Sync Smart Home API Docs

None

None

None

Table of contents

1. Sync Smart Home API Documentation

2. API Reference

2.1 `app.main`

This module initializes & configures the FastAPI application

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

2.1.1 `app.main.RootResponse`

Bases: `BaseModel`

Response model for the root endpoint

Attributes:

Name	Type	Description
`message`	`str`	Welcome message

Source code in `app/main.py`

```python
class RootResponse(BaseModel):
    """
    Response model for the root endpoint

    Attributes:
        message (str): Welcome message
    """
    message: str
```

2.1.2 `app.main.lifespan(app_context)` `async`

Defines application's lifespan event handler

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

Parameters:

Name	Type	Description	Default
`app_context`	`FastAPI`	The FastAPI application instance	*required*

Yields:

Name	Type	Description
`None`		Allows FastAPI to manage application life-cycle

Raises:

Type	Description
`Exception`	If database initialization fails

Source code in `app/main.py`

```python
@asynccontextmanager
async def lifespan(app_context: FastAPI):
    """
    Defines application's lifespan event handler

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

    Args:
        app_context (FastAPI): The FastAPI application instance

    Yields:
        None: Allows FastAPI to manage application life-cycle

    Raises:
        Exception: If database initialization fails
    """
    try:
        await init_db()
        app_context.state.custom_attribute = "value"    # Placeholder for future app-wide state
        logger.info("Application startup complete.")
        yield
    except Exception as e:
        logger.error("Failed to initialize the database: %e.")
        raise e
    finally:
        logger.info("Application is shutting down.")
```

2.1.3 `app.main.read_root()`

Root endpoint

Returns:

Name	Type	Description
`RootResponse`	`RootResponse`	A welcome message for the API

Source code in `app/main.py`

```python
@app.get("/", response_model = RootResponse)
def read_root() -> RootResponse:
    """
    Root endpoint

    Returns:
        RootResponse: A welcome message for the API
    """
    return RootResponse(message = "Welcome to the Sync Smart Home.")
```

2.2 `app.routes.user_routes`

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

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

2.2.1 `app.routes.user_routes.get_admin_dashboard()` async

Admin dashboard endpoint (restricted access)

Returns:

Name	Type	Description
`dict`		A welcome message confirming admin access

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

```python
@router.get("/admin/dashboard", dependencies = [Depends(role_required("admin"))])
async def get_admin_dashboard():
    """
    Admin dashboard  endpoint (restricted access)

    Returns:
        dict: A welcome message confirming admin access
    """
    return {"message": "Welcome, admin!"}
```

2.2.2 `app.routes.user_routes.login(form_data=Depends())` async

Authenticate user & return a JWT token

Parameters:

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

Returns:

Name	Type	Description
`dict`		A dictionary containing the access token & token type

Raises:

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

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

```python
@router.post("/token")
async def login(form_data: OAuth2PasswordRequestForm = Depends()):
    """
    Authenticate user & return a JWT token

    Args:
        form_data (OAuth2PasswordRequestForm): User-provided login credentials

    Returns:
        dict: A dictionary containing the access token & token type

    Raises:
        HTTPException (400): If the username or password is incorrect
    """
    user = users_collection.find_one({"email": form_data.username})
    if not user or not verify_password(form_data.password, user["password_hash"]):
        raise HTTPException(status_code = 400, detail = "Invalid username or password")

    access_token = create_access_token(
        data = {"sub": user["email"], "role": user.get("role", "user")},
        expires_delta = timedelta(minutes = 30)
    )

    return {"access_token": access_token, "token_type": "bearer"}
```

2.2.3 `app.routes.user_routes.register_user(user)` async

Register a new user in the database

Parameters:

Name	Type	Description	Default
`user`	`UserCreate`	User registration data including email, password & optional role	*required*

Returns:

Name	Type	Description
`UserResponse`		The created user information excluding password

Raises:

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

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

```python
@router.post("/register", response_model = UserResponse)
async def register_user(user: UserCreate):
    """
    Register a new user in the database

    Args:
        user (UserCreate): User registration data including email, password & optional role

    Returns:
        UserResponse: The created user information excluding password

    Raises:
        HTTPException (400): If the email is already registered
        HTTPException (500): If there is an error inserted the user into the database
    """
    # Check if email exists already
    if users_collection.find_one({"email": user.email}):
        raise HTTPException(status_code = 400, detail = "Registration failed, please try again.")

    # Prepare user data for insertion
    user_data = {
        "email": user.email,
        "password_hash": hash_password(user.password),
        "is_verified": False,
        "created_at": datetime.now(timezone.utc),
        "updated_at": datetime.now(timezone.utc),
        "role": user.role or "user"
    }

    # Database user insert
    try:
        result = users_collection.insert_one(user_data)
    except Exception as e:
        raise HTTPException(status_code = 500, detail = f"Failed to register user: {e}") from e

    return UserResponse(
        id = str(result.inserted_id),
        role = user_data["role"],
        email = user.email,
        is_verified = False,
        created_at = user_data["created_at"]
    )
```

2.3 `app.routes.data_routes`

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

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

2.3.1 `app.routes.data_routes.add_energy_data(data)` async

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

Parameters:

Name	Type	Description	Default
`data`	`EnergyData`	The energy data record to be added	*required*

Returns:

Name	Type	Description
`dict`		A confirmation message upon successfully insertion

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

```
@router.post("/add", dependencies = [Depends(role_required("admin"))])
async def add_energy_data(data: EnergyData):
    """
    Add new energy consumption data to the database (admin-only)

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

    Returns:
        dict: A confirmation message upon successfully insertion
    """
    energy_collection.insert_one(data.model_dump())       # Insert into MongoDB
    return {"message": "Energy data added successfully"}
```

2.3.2 `app.routes.data_routes.get_admin_dashboard()` async

Admin dashboard endpoint (restricted access)

Returns:

Name	Type	Description
`dict`		A welcome message confirming admin access

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

```
@router.get("/admin/dashboard", dependencies = [Depends(role_required("admin"))])
async def get_admin_dashboard():
    """
    Admin dashboard endpoint (restricted access)

    Returns:
        dict: A welcome message confirming admin access
    """
    return {"message": "Welcome, admin!"}
```

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

Retrieve aggregated energy consumption data based on time interval & filters

Parameters:

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

Returns:

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

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

```python
@router.get("/aggregate")
async def get_aggregated_data(
    start_date: Optional[str] = Query(None, description = "Start date (YYYY-MM-DD)"),
    end_date: Optional[str] = Query(None, description = "End date (YYYY-MM-DD)"),
    device_id: Optional[str] = Query(None, description = "Device ID filter"),
    location: Optional[str] = Query(None, description = "Location filter"),
    interval: Literal["hour", "day", "week"] = "day",
):
    """
    Retrieve aggregated energy consumption data based on time interval & filters

    Args:
        start_date (Optional[str]): The start date for filtering (YYYY-MM-DD)
        end_date (Optional[str]): The end date for filtering (YYYY-MM-DD)
        device_id (Optional[str]): The ID of the device to filter data
        location (Optional[str]): The location filter
        interval (Literal["hour", "day", "week"]): The time interval for aggregation (defaults to "day")

    Returns:
        dict: Aggregated energy consumption data grouped by the selected interval
    """
    query = {}

    if start_date and end_date:
        query["timestamp"] = {
            "$gte": datetime.strptime(start_date, "%Y-%m-%d"),
            "$lte": datetime.strptime(end_date, "%Y-%m-%d")
        }
    if device_id:
        query["device_id"] = device_id
    if location:
        query["location"] = location

    time_group = {
        "year": {"$year": "$timestamp"},
        "month": {"$month": "$timestamp"},
        "day": {"$dayOfMonth": "$timestamp"}
    }

    if interval == "hour":
        time_group = {
            "device_id": "$device_id",
            "year": {"$year": "$timestamp"},
            "month": {"$month": "$timestamp"},
            "day": {"$dayOfMonth": "$timestamp"},
            "hour": {"$hour": "$timestamp"}
        }
    elif interval == "day":
        time_group = {
            "device_id": "$device_id",
            "year": {"$year": "$timestamp"},
            "month": {"$month": "$timestamp"},
            "day": {"$dayOfMonth": "$timestamp"}
        }
    elif interval == "week":
        time_group = {
            "device_id": "$device_id",
            "year": {"$year": "$timestamp"},
            "week": {"$isoWeek": "$timestamp"}
        }

    aggregation_pipeline = [
        {"$match": query},
        {"$group": {
            "_id": {
                "device_id": "$device_id",  # Group by device
                **time_group                # Group by time interval
            }, "total_energy": {"$sum": "$energy_consumed"}
        }}
    ]

    result = list(energy_collection.aggregate(aggregation_pipeline))

    return {"aggregated_data": result}
```

2.4 `app.routes.report_routes`

This module provides API endpoints for generating energy consumption reports

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

2.4.1

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

Generate an energy consumption report in CSV or PDF format

Parameters:

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

Returns:

Name	Type	Description
`FileResponse`		The generated report file

Raises:

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

Source code in `app/routes/report_routes.py`

```python
@router.post("/report", dependencies = [Depends(role_required("admin"))])
async def generate_report(
        format: str = Query("csv", enum = ["csv", "pdf"]),
        start_date: str = Query(None, description = "Start date (YYYY-MM-DD)"),
        end_date: str = Query(None, description = "End date (YYYY-MM-DD)")
):
    """
    Generate an energy consumption report in CSV or PDF format

    Args:
        format (str): The desired report format, either "csv" or "pdf"
        start_date (str, optional): The start date for filtering data (YYYY-MM-DD)
        end_date (str, optional): The end date for filtering data (YYYY-MM-DD)

    Returns:
        FileResponse: The generated report file

    Raises:
        HTTPException (400): If an invalid date format is provided
        HTTPException (404): If no energy data is available for the selected range
    """
    try:
        energy_data = get_energy_data(start_date, end_date)
    except ValueError as exc:
        raise HTTPException(status_code = 400, detail = str(exc)) from exc

    if not energy_data:
        raise HTTPException(status_code = 404, detail = "No energy data available for the selected range.")

    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = f"{REPORTS_DIR}/energy_report_{timestamp}.{format}"

    if format == "csv":
        # Convert data to a DataFrame & format timestamps
        df = pd.DataFrame(energy_data)
        df["timestamp"] = pd.to_datetime(df["timestamp"]).dt.strftime("%Y-%m-%d %H:%M:%S")  # Format timestamp
        df.columns = ["Device ID", "Timestamp", "Energy Consumed (kWh)", "Location"]
        df.to_csv(filename, index = False)
    elif format == "pdf":
        # Create a PDF report with a structure table
        doc = SimpleDocTemplate(filename, pagesize=letter)
        elements = []

        # Prepare table data
        data = [["Device ID", "Timestamp", "Energy Cosumed (kWh)", "Location"]]
        for entry in energy_data:
            data.append([
                entry.get("device_id", "N/A"),
                entry.get("timestamp", "").strftime('%Y-%m-%d %H:%M:%S') if entry.get("timestamp") else "N/A",
                entry.get("energy_consumed", "N/A"),
                entry.get("location", "N/A")
            ])

        # Create & style table
        table = Table(data)
        table.setStyle(TableStyle([
            ('BACKGROUND', (0, 0), (-1, 0), colors.grey),
            ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
            ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
            ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
            ('FONTSIZE', (0, 0), (-1, 0), 12),
            ('BOTTOMPADDING', (0, 0), (-1, 0), 10),
            ('GRID', (0, 0), (-1, -1), 1, colors.black),
            ('ROWBACKGROUNDS', (0, 1), (-1, -1), [colors.whitesmoke, colors.lightgrey])
        ]))

        elements.append(table)
        doc.build(elements)

    return FileResponse(
        path  = os.path.abspath(filename),
        filename = os.path.basename(filename),
        media_type = "application/octet-stream"
    )
```

2.5 `app.models.user`

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

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

2.5.1 `app.models.user.UserCreate`

Bases: `BaseModel`

Pydantic model for user registration input

Attributes:

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

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

```python
class UserCreate(BaseModel):
    '''
    Pydantic model for user registration input

    Attributes:
        email (EmailStr): The user's email address
        password (str): The user's password (validated for strength)
        role (Optional[str]): The user's role, defaulting to "user"
    '''
    email: EmailStr
    password: str
    role: Optional[str] = "user"    # Default role

    @field_validator("password")
    @classmethod
    def validate_password(cls, value: str) -> str:
        """
        Validate the password to ensure it meets security requirements

        Args:
            value (str): The password provided by the user

        Returns:
            str: The validated password

        Raises:
            ValueError: If the password doesn't meet security requirements
        """
        if len(value) < 8:
            raise ValueError("Password must be at least 8 characters long.")
        if not any(char.isdigit() for char in value):
            raise ValueError("Password must contain at least 1 number.")
        if not any(char.isalpha() for char in value):
            raise ValueError("Password must contain at least 1 letter.")
        if not any(char.islower() for char in value):
            raise ValueError("Password must contain at least 1 lower letter.")
        if not any(char.isupper() for char in value):
            raise ValueError("Password must contain at least 1 upper letter.")
        if not any(char in '!@#$%^&*()_+-=[]{}|;\':",.<>?/' for char in value):
            raise ValueError("Password must contain at least 1 special character.")
        return value

    @field_validator("email")
    @classmethod
    def validate_email(cls, value: EmailStr) -> EmailStr:
        """
        Validate the user's email address

        Args:
            value (EmailStr): The email address provided

        Returns:
            EmailStr: The validated email

        Raises:
            ValueError: If the email doesn't contain `@` or a valid domain
        """
        if not "@" in value:
            raise ValueError("Email must contain @.")
        if not "." in value.split("@")[1]:
            raise ValueError("Email must contain a dot.")
        return value
```

`app.models.user.UserCreate.validate_email(value)` `classmethod`

Validate the user's email address

Parameters:

Name	Type	Description	Default
`value`	`EmailStr`	The email address provided	*required*

Returns:

Name	Type	Description
`EmailStr`	`EmailStr`	The validated email

Raises:

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

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

```python
@field_validator("email")
@classmethod
def validate_email(cls, value: EmailStr) -> EmailStr:
    """
    Validate the user's email address

    Args:
        value (EmailStr): The email address provided

    Returns:
        EmailStr: The validated email

    Raises:
        ValueError: If the email doesn't contain `@` or a valid domain
    """
    if not "@" in value:
        raise ValueError("Email must contain @.")
    if not "." in value.split("@")[1]:
        raise ValueError("Email must contain a dot.")
    return value
```

`app.models.user.UserCreate.validate_password(value)` `classmethod`

Validate the password to ensure it meets security requirements

Parameters:

Name	Type	Description	Default
`value`	`str`	The password provided by the user	*required*

Returns:

Name	Type	Description
`str`	`str`	The validated password

Raises:

Type	Description
`ValueError`	If the password doesn't meet security requirements

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

```python
@field_validator("password")
@classmethod
def validate_password(cls, value: str) -> str:
    """
    Validate the password to ensure it meets security requirements

    Args:
        value (str): The password provided by the user

    Returns:
        str: The validated password

    Raises:
        ValueError: If the password doesn't meet security requirements
    """
    if len(value) < 8:
        raise ValueError("Password must be at least 8 characters long.")
    if not any(char.isdigit() for char in value):
        raise ValueError("Password must contain at least 1 number.")
    if not any(char.isalpha() for char in value):
        raise ValueError("Password must contain at least 1 letter.")
    if not any(char.islower() for char in value):
        raise ValueError("Password must contain at least 1 lower letter.")
    if not any(char.isupper() for char in value):
        raise ValueError("Password must contain at least 1 upper letter.")
    if not any(char in '!@#$%^&*()_+-=[]{}|;\':",.<>?/' for char in value):
        raise ValueError("Password must contain at least 1 special character.")
    return value
```

2.5.2 `app.models.user.UserResponse`

Bases: `BaseModel`

Pydantic model for user response data

Attributes:

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

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

```python
class UserResponse(BaseModel):
    """
    Pydantic model for user response data

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

    model_config = ConfigDict(from_attributes = True)
```

2.6 `app.models.energy_data`

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

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

2.6.1 `app.models.energy_data.EnergyData`

Bases: `BaseModel`

Pydantic model representing energy consumption data for a smart device

Attributes:

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

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

```python
class EnergyData(BaseModel):
    """
    Pydantic model representing energy consumption data for a smart device

    Attributes:
        device_id (str): Unique identifier for the device
        timestamp (datetime): The date & time of the recorded energy consumption
        energy_consumed (float): The amount of energy consumed in kilowatt-hours (kWh)
        location (Optional[str]): The optional physical location of the device
    """
    device_id: str
    timestamp: datetime
    energy_consumed: float          # tracking in kWh (kilowatt hours)
    location: Optional[str] = None
```

2.7 `app.db.database`

This module handles MongoDB database connections & operations

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

2.7.1 `app.db.database.get_energy_data(start_date=None, end_date=None)`

Fetch energy data consumption from MongoDB with optional date filtering

Parameters:

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

Returns:

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

Raises:

Type	Description
`ValueError`	If the provided date format is incorrect

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

```python
def get_energy_data(start_date: str = None, end_date: str = None) -> List[Dict]:
    """
    Fetch energy data consumption from MongoDB with optional date filtering

    Args:
        start_date (str, optional): Start date in `YYYY-MM-DD` format
        end_date (str, optional): End date in `YYYY-MM-DD` format

    Returns:
        List[Dict]: A list of energy consumption records

    Raises:
        ValueError: If the provided date format is incorrect
    """
    query = {}

    if start_date and end_date:
        try:
            start_dt = datetime.strptime(start_date, "%Y-%m-%d")
            end_dt = datetime.strptime(end_date, "%Y-%m-%d")
            query["timestamp"] = {"$gte": start_dt, "$lte": end_dt}
        except ValueError as exc:
            raise ValueError("Invalid date format. Use `YYYY-MM-DD`.") from exc

    energy_data = list(energy_collection.find(query, {"_id": 0}))   # Exclude MongoDB _id field

    return energy_data
```

2.7.2 `app.db.database.init_db()` async

Initialize the database by creating necessary indexes

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

Raises:

Type	Description
`RuntimeError`	If there is an error during database initialization

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

```python
async def init_db():
    """
    Initialize the database by creating necessary indexes

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

    Raises:
        RuntimeError: If there is an error during database initialization
    """
    try:
        # Create uniqueness of user emails
        users_collection.create_index("email", unique = True)

        # Optimize queries by creating indexes on frequently queried fields
        energy_collection.create_index("device_id")
        energy_collection.create_index("timestamp")

        print("Database initialized successfully.")
    except Exception as e:
        raise RuntimeError(f"Error during database initialization: {e}") from e
```

2.8 `app.core.config`

2.9 `app.core.security`

This module implements security features

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

2.9.1 `app.core.security.create_access_token(data, expires_delta=None)`

Create a JWT token with an expiration time

Parameters:

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

Returns:

Name	Type	Description
`str`		The encoded JWT token

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

```python
def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    """
    Create a JWT token with an expiration time

    Args:
        data (dict): The payload to encode in the token
        expires_delta (Optional[timedelta], optional): The expiration time delta
            Defaults to `ACCESS_TOKEN_EXPIRE_MINUTES`

    Returns:
        str: The encoded JWT token
    """
    to_encode = data.copy()
    expire = datetime.now(timezone.utc) + (expires_delta or timedelta(minutes = ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm = ALGORITHM)
```

2.9.2 `app.core.security.get_current_user(token=Depends(oauth2_scheme))`

Extract & validate the current user's JWT token

Parameters:

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

Returns:

Name	Type	Description
`dict`	`dict`	The decoded token payload

Raises:

Type	Description
`HTTPException`	If the token is invalid or missing required fields

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

```python
def get_current_user(token: str = Depends(oauth2_scheme)) -> dict:
    """
    Extract & validate the current user's JWT token

    Args:
        token (str): The OAuth2 token obtained from authentication

    Returns:
        dict: The decoded token payload

    Raises:
        HTTPException: If the token is invalid or missing required fields
    """
    payload = verify_access_token(token)
    if not payload or "role" not in payload:
        raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail = "Invalid token")
    return payload
```

2.9.3 `app.core.security.hash_password(password)`

Hash a plain-text password using bcrypt

Parameters:

Name	Type	Description	Default
`password`	`str`	The password to hash	*required*

Returns:

Name	Type	Description
`str`	`str`	The hashed password

Raises:

Type	Description
`ValueError`	If an error occurs during hashing

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

```python
def hash_password(password: str) -> str:
    """
    Hash a plain-text password using bcrypt

    Args:
        password (str): The password to hash

    Returns:
        str: The hashed password

    Raises:
        ValueError: If an error occurs during hashing
    """
    try:
        return pwd_context.hash(password)
    except Exception as e:
        raise ValueError(f"Error hashing password: {e}") from e
```

2.9.4 `app.core.security.needs_rehash(hashed_password)`

Check if a stored hashed password required rehashing

Parameters:

Name	Type	Description	Default
`hashed_password`	`str`	The existing hashed password	*required*

Returns:

Name	Type	Description
`bool`	`bool`	True if rehashing is required & False if otherwise

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

```python
def needs_rehash(hashed_password: str) -> bool:
    """
    Check if a stored hashed password required rehashing

    Args:
        hashed_password (str): The existing hashed password

    Returns:
        bool: True if rehashing is required & False if otherwise
    """
    return pwd_context.needs_update(hashed_password)
```

2.9.5 `app.core.security.role_required(required_role)`

Dependency function to check to enforce role-based access control

Parameters:

Name	Type	Description	Default
`required_role`	`str`	The required user role	*required*

Returns:

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

Raises:

Type	Description
`HTTPException`	If the user lacks the required role

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

```python
def role_required(required_role: str):
    """
    Dependency function to check to enforce role-based access control

    Args:
        required_role (str): The required user role

    Returns:
        function: A dependency function that verifies the user's role

    Raises:
        HTTPException: If the user lacks the required role
    """
    def role_checker(current_user: dict = Depends(get_current_user)):
        if current_user.get("role") != required_role:
            raise HTTPException(
                status_code = status.HTTP_403_FORBIDDEN,
                detail = f"Permission denied: {required_role} role required"
            )
        return current_user
    return role_checker
```

2.9.6 `app.core.security.verify_access_token(token)`

Verify & decode a JWT access token

Parameters:

Name	Type	Description	Default
`token`	`str`	The JWT token to verify	*required*

Returns:

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

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

```python
def verify_access_token(token: str) -> Optional[dict]:
    """
    Verify & decode a JWT access token

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

    Returns:
        Optional[dict]: The decoded token payload if valid, otherwise None
    """
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms = [ALGORITHM])
        return payload
    except JWTError:
        return None
```

2.9.7 `app.core.security.verify_password(plain_password, hashed_password)`

Verify a plain-text password against a hashed password

Parameters:

Name	Type	Description	Default
`plain_password`	`str`	The input password	*required*
`hashed_password`	`str`	The stored hashed password	*required*

Returns:

Name	Type	Description
`bool`	`bool`	True if the password matches & False if otherwise

Raises:

Type	Description
`ValueError`	If an error occurs during verification

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

```python
def verify_password(plain_password: str, hashed_password: str) -> bool:
    """
    Verify a plain-text password against a hashed password

    Args:
        plain_password (str): The input password
        hashed_password (str): The stored hashed password

    Returns:
        bool: True if the password matches & False if otherwise

    Raises:
        ValueError: If an error occurs during verification
    """
    try:
        return pwd_context.verify(plain_password, hashed_password)
    except Exception as e:
        raise ValueError(f"Error verifying password: {e}") from e
```