Smart Home Automation

✅ 1. Device Class

Purpose:

Represents a smart home device like a light, thermostat, or security system.

Attributes:

- `deviceID (int)` → Unique identifier for the device.
- `type (string)` → Type of the device (`"light"`, `"thermostat"`, `"camera"`, etc.).
- `status (string)` → Current status (`"on"`, `"off"`, `"active"`, `"inactive"`).
- `settings (map<string, string>)` → Key-value settings specific to the device (e.g., temperature for thermostat).

Methods:

- `turnOn()` → Turns the device on.
- `turnOff()` → Turns the device off.
- `updateSettings(key: string, value: string)` → Updates specific settings of the device.
- `getStatus()` → Returns the current status of the device.

Relationships:

- **Server** owns the devices using a **composition**.
- **DeviceManager** manages the lifecycle of devices.

✅ 2. DeviceManager Class

Purpose:

Handles device registration, management, and lookup. Acts as a middle layer between the server and the devices.

Attributes:

- `devices (map<int, Device>)` → Stores all devices using their `deviceID` as the key.

Methods:

- `registerDevice(device: Device)` → Registers a new device.
- `lookupDevice(deviceID: int)` → Returns a reference to a device by its ID.

- `removeDevice(deviceID: int)` → Removes a device from the system.

Relationships:

- **Server** delegates device management tasks to the **DeviceManager**.
- **DeviceManager** has an aggregation relationship with **Device** (Devices can exist independently, but the manager tracks them).

✅ 3. Server Class

Purpose:

Acts as a central control point, managing devices, handling client requests, and communicating over the network.

Attributes:

- `deviceManager (DeviceManager)` → Manages all devices.
- `serverSocket (Network)` → Listens for client connections.
- `clients (list<Client>)` → Tracks connected clients.

Methods:

- `startServer()` → Starts the server and begins listening for client connections.
- `handleRequest(client: Client, request: string)` → Parses and handles client commands.
- `sendCommand(deviceID: int, command: string)` → Sends control commands to devices.

Relationships:

- **Client** connects to the **Server** via TCP using the **Network** class.
- **Server** uses **ProtocolHandler** to interpret client requests.
- **ThreadManager** handles requests concurrently using multithreading.

✅ 4. Client Class

Purpose:

Provides an interface for users to send commands and view device statuses.

Attributes:

- `clientSocket (Network)` → Manages the connection to the server.
- `deviceID (int)` → Specifies which device the client is controlling.

Methods:

- `connectToServer()` → Establishes a connection with the server.
- `sendCommand(command: string)` → Sends commands like `"GET /light/on"`.
- `receiveData()` → Receives and displays responses from the server.

Relationships:

- **Client** communicates with the **Server** using TCP/IP through the **Network** class.

✅ 5. Network Class

Purpose:

Simulates network communication using TCP/IP.

Attributes:

- `ipAddress (string)` → IP address of the server or client.
- `port (int)` → Port used for the connection.

Methods:

- `sendData(data: string)` → Sends data over the network.
- `receiveData()` → Receives data from the network.
- `startCommunication()` → Establishes the network connection.

Relationships:

- Used by both **Server** and **Client** for communication.
- Simulates network-level interactions (e.g., routing, ARP) through the **Router** class.

✅ 6. ProtocolHandler Class

Purpose:

Handles custom application-level protocols. Implements a simple HTTP-like protocol.

Methods:

- `encodeMessage(message: string)` → Encodes commands into a simple protocol format (`"GET /light/on"`).
- `decodeMessage(message: string)` → Decodes incoming server responses.
- `validateMessage(message: string)` → Ensures proper formatting of requests.

Relationships:

- **Server** uses this to parse and interpret client requests.
- **Client** uses this to format its requests.

✅ 7. Router Class

Purpose:

Handles routing of network packets between subnets.

Attributes:

- `routingTable (map<string, string>)` → Maps destination IPs to next-hop IPs.
- `activeConnections (list<Connection>)` → Manages active network connections.

Methods:

- `routeMessage(deviceID: int, message: string)` → Routes messages to appropriate devices.
- `updateTopology()` → Adjusts the network routing table.

Relationships:

- **Network** uses the **Router** to simulate network-level routing.

✅ 8. ThreadManager Class

Purpose:

Manages multithreading operations, ensuring the server handles multiple client requests.

Attributes:

- `threads (list<Thread>)` → Active threads running on the server.

Methods:

- `startThread(task: function)` → Creates and starts a new thread.
- `stopThread(threadID: int)` → Stops a running thread.
- `assignTaskToThread(task: function)` → Allocates tasks to threads.

Relationships:

- Used by the **Server** to ensure efficient handling of multiple client requests.

✅ 9. ConfigurationManager Class

Purpose:

Handles configuration management, loading initial settings and saving changes.

Methods:

- `loadConfig(file: string)` → Reads configuration from a file.
- `saveConfig(file: string)` → Saves updated configuration to a file.
- `updateSettings(key: string, value: string)` → Updates specific settings.

Relationships:

- **Server** uses this to load and apply configurations at startup.

✅ 10. Logger Class

Purpose:

Logs system events for debugging and analysis.

Methods:

- `logInfo(message: string)` → Logs informational messages.
- `logError(message: string)` → Logs errors.
- `clearLogs()` → Clears logs.
- `retrieveLogs()` → Retrieves log history.

Relationships:

- **Server** uses this for logging client requests and system events.

✅ How it All Comes Together

1. **Client** sends a request (`"GET /light/on"`) using the **ProtocolHandler**.
2. The request is routed using the **Network** and reaches the **Server**.
3. The **Server** decodes the request using the **ProtocolHandler**.
4. The **Server** checks with the **DeviceManager** to find the correct **Device**.
5. The **Device** processes the command using its methods (`turnOn()` or `turnOff()`).
6. The **Server** sends a response back to the **Client**.
7. **Logger** records all interactions, while **ThreadManager** ensures efficient processing using multithreading.