ChatGPT

Building a Home Assistant Custom Integration for Lights and Occupancy Sensors

Overview of Custom Component Structure

To integrate custom lights and occupancy sensors into Home Assistant (HA), you'll build a **custom integration** (also called a custom component) in Python. At a high level, this involves creating a new integration folder (domain) under `custom_components/` with the appropriate files (like `manifest.json`, `__init__.py`, etc.), and implementing HA entity platforms for your devices (lights, sensors).

Key points about the integration structure:

- **Integration Folder & Manifest:** Create a folder named after your integration's domain (e.g. `my_custom_lights`) in `custom_components`. In this folder, include a `manifest.json` with at least the domain, name, version, and indicate support for config flow (UI setup) by adding `"config_flow": true` [1]. This tells Home Assistant that your integration can be configured via the UI rather than YAML.
- **Platform Files:** For each type of entity (device) you want to expose, create a platform file. For example, to integrate lights, create a `light.py` file in your integration folder, and for occupancy sensors (binary sensors), create a `binary_sensor.py`. Home Assistant will load these if you forward the config entry setup to those platforms. **Each platform file will define entity classes inheriting from HA base classes** (e.g., `LightEntity` for lights, `BinarySensorEntity` for occupancy) and handle the communication with your API for that domain [2].
- **Config Flow (UI Setup):** Since you want to avoid YAML and use the UI, implement a `config_flow.py`. This defines how a user provides configuration (like API host, credentials) through the Home Assistant UI. Your `config_flow.py` will subclass `ConfigFlow` and prompt the user for necessary info (e.g., host IP, auth token or username/password), then create a config entry. Once the config entry is created, Home Assistant will call your integration's `async_setup_entry` to initialize the component [3] [1].

Initializing the Integration and Discovering Devices

When Home Assistant starts (or when the user adds the integration via the UI), your integration will set up and discover the lights and sensors from your local API:

1. **`async_setup_entry` in `__init__.py`:** This function is called by HA when the config entry is added (or on startup for an existing entry). Here you typically:
2. Create an API client object or otherwise store connection info. For example, use the data from `config_entry.data` (like host, token) to initialize your API wrapper (since you mentioned you'll handle the API calls, this could be where you set up your API connection object).

3. Store the API object and any other shared data in `hass.data[DOMAIN][entry.entry_id]` so it can be accessed by platform code [4] [5].

4. Forward the entry setup to platform files. For instance, call `hass.config_entries.async_forward_entry_setups(entry, ["light", "binary_sensor"])` to load your platform entities (or use the newer `async_forward_entry_setups` which can forward multiple domains at once). This will invoke the `async_setup_entry` function defined in your `light.py` and `binary_sensor.py` files.

5. **Device Discovery on Startup:** In the platforms' `async_setup_entry`, fetch the current list of devices (lights and sensors) from your local API. This is where you "search for new lights when the component starts up" – effectively querying the API for all lights and sensors and creating HA entities for each. For example, in `light.py`'s `async_setup_entry`, you might call an API endpoint that returns the JSON list of lights (like the example you provided) and then create `LightEntity` objects for each light in the response. Similarly, fetch occupancy sensor data in `binary_sensor.py`.

6. **Dynamic Device Addition:** Home Assistant will register whatever entities you add during setup. If new devices are added to your system later, you have a few options:

   - **On each startup**, the integration will again fetch all devices, so any newly added lights/ sensors will appear (this requires restarting or reloading the integration).
   - **Periodic discovery:** You could periodically poll for new devices even while running. For example, schedule a job (using HA's helpers like `async_track_time_interval`) to call the API for the device list and compare with known devices, adding any new ones via `async_add_entities`. This isn't done automatically by HA; you'd implement it if hot-plugging devices is important.
   - **Manual service:** Alternatively, expose a Home Assistant **service** (via `async_register_entity_service` or define one in `services.yaml`) that when called will trigger a discovery scan for new devices. This way, from the HA UI or automations, you could invoke a "find new lights" action.

However, the simplest approach initially is to load all devices during `async_setup_entry` so that when the component starts, it discovers and registers all current lights and sensors.

1. **Unique IDs and Device Info:** As you create each entity, assign it a `unique_id` (for example, use the device's ID from your API, such as `"light_10"` for the light with id 10). Providing a unique ID allows Home Assistant to preserve the entity (and user customizations) across restarts. Also implement the entity's `device_info` property to register devices in HA's **Device Registry**. This ties entities to a device container in HA. In `device_info`, supply an `identifiers` value (e.g., `(DOMAIN, device_id)`), a device name, and other info. You can also include a `suggested_area` to hint the Area name. For example, if your API provides a room name for a device, you can pass that to HA. Home Assistant will use `suggested_area` to automatically assign the device to an Area (if the area exists or when created) [6]. This addresses the requirement of "an easy way to assign them to Areas in HA": the integration can suggest area names based on your system's room info.

Implementing the Light Entity (Custom Light with Brightness & Fade)

For each light from your API data, create a class that inherits from `homeassistant.components.light.LightEntity`. This class wraps the actual device and implements properties and methods to interface with HA. Key aspects to implement:

- **On/Off and Brightness:** Implement the `is_on` property to reflect the light's on/off state, and if the light is dimmable, implement the `brightness` property (0-255 in HA). Your raw API shows `level` as a 16-bit value (0–65535). Home Assistant uses 0–255 for brightness values, so you'll need to scale accordingly. You can do this conversion manually (e.g., `ha_brightness = int(api_level * 255 / 65535)`) or use HA's utility function `homeassistant.util.color.value_to_brightness(range, value)` to map a value from a given range into 0–255 [7]. For example, if your brightness range is 0–65535, you could define `BRIGHTNESS_SCALE = (0, 65535)` and use `value_to_brightness(BRIGHTNESS_SCALE, api_level)` to get the HA brightness value, and likewise use `percentage_to_ranged_value` for converting back when setting brightness [8] [9].

- **Supported Features (Fade/Transition):** Since your custom lights support fade duration for brightness changes, you'll want to leverage HA's **transition** feature. Home Assistant's light service calls include an optional `transition` parameter (in seconds) which indicates the fade duration. To advertise this capability, set the light entity's `supported_features` to include `LightEntityFeature.TRANSITION` [10]. Then, in your `async_turn_on` / `async_turn_off` methods, handle the `transition` parameter if present in `**kwargs`. For example:

```python
async def async_turn_on(self, **kwargs):
    brightness = kwargs.get("brightness")
    transition = kwargs.get("transition")  # seconds
    # Convert HA brightness to API level if needed
    if brightness is not None:
        level = int(brightness * 65535 / 255)
    else:
        level = None
    # Call your API to turn on the light, passing the level and transition
    await api.set_light(self.device_id, level=level, transition=transition)
    # Optionally update internal state and request a refresh
    self._attr_is_on = True
    if brightness is not None:
        self._attr_brightness = brightness
    # Request a state update (especially if not immediately reflected)
    await self.coordinator.async_request_refresh()  # if using coordinator
```

  Home Assistant will manage calling your `async_turn_on` or `async_turn_off` when the user toggles the light or changes brightness. After changing a state, you might want to trigger an immediate data refresh (as shown above) so the new state is quickly reflected in HA. If using a

DataUpdateCoordinator (discussed below), you can call `coordinator.async_request_refresh()` [11] to fetch updated data after sending a control command.

- **Brightness and Color Mode:** If your lights are purely dimmable (no color), you should set `supported_color_modes = {ColorMode.BRIGHTNESS}` (for dimmable lights) or `{ColorMode.ONOFF}` (for non-dimmable switches). This modern approach replaces legacy flags like `SUPPORT_BRIGHTNESS`. By declaring support for brightness, HA will show a brightness slider for those lights. In your case, lights with `subType: "dimmer"` would use brightness, whereas `"switch"` (on/off only) might use `ColorMode.ONOFF`. Ensure you only implement `brightness` property for the dimmer lights, and for on/off switches, you can omit the `brightness` property so HA knows they are just binary lights (the UI will then show a simple toggle).

- **Example**: In practice, your `LightEntity` subclass will hold an internal reference to the API or coordinator data for that specific light (e.g., an ID). It will have something like:

```
class CustomLight(LightEntity):
    def __init__(self, api, device_info):
        self.api = api
        self.device = device_info  # e.g., the dict of light data
        self._attr_unique_id = f"{DOMAIN}_light_{device_info['id']}"
        self._attr_name = device_info['name']
        # Set supported features and color modes
        if device_info['subType'] == 'dimmer':
            self._attr_supported_color_modes = {ColorMode.BRIGHTNESS}
            self._attr_color_mode = ColorMode.BRIGHTNESS
        else:
            self._attr_supported_color_modes = {ColorMode.ONOFF}
            self._attr_color_mode = ColorMode.ONOFF
        # If transitions (fade) are supported generally:
        self._attr_supported_features = LightEntityFeature.TRANSITION
    @property
    def is_on(self):
        # Determine from device info if light is on (e.g., level > 0 or a
separate status field)
        return self.device_status['is_on']
    @property
    def brightness(self):
        # Only for dimmers: convert device brightness to 0-255
        if self.device['subType'] == 'dimmer':
            level = self.device_status['level']  # 0-65535
            return int(level * 255 / 65535)
        return None
    async def async_turn_on(self, **kwargs):
        # (Implementation as described above)
        ...
    async def async_turn_off(self, **kwargs):
        await self.api.set_light(self.device['id'], level=0)
        self._attr_is_on = False
        await self.coordinator.async_request_refresh()
```

This is just illustrative; in a real integration, you might integrate with the DataUpdateCoordinator for state updates (see below). The key is that the LightEntity class wraps the behavior of your physical light device so HA can interact with it seamlessly.

Implementing Occupancy Sensors (Binary Sensors)

Occupancy sensors can be represented as **binary sensors** in Home Assistant. HA's `BinarySensorEntity` is used for sensors that have two states (on/off). For an occupancy sensor:

- **State as `is_on`:** You can interpret `"presence": "Vacant"` vs `"Occupied"` as a boolean. For example, set `is_on = True` if the sensor reports "Occupied", and `False` if "Vacant".
- **Device Class:** Set the binary sensor's `device_class` to `BinarySensorDeviceClass.OCCUPANCY` (or `BinarySensorDeviceClass.MOTION` if more appropriate). The Occupancy device class is ideal for presence detection – in HA this class means *On = occupied, Off = not occupied (clear)* [12] . This will also display an appropriate icon (e.g., a person icon) and integrate with HA's semantics (like marking a room as occupied).

- **Example:** Suppose your API sensor data is like:

```
{"id": 7, "name": "Hallway Occupancy", "subType": "OccupancySensor",
"presence": "Vacant", ...}
```

You would create a `CustomOccupancySensor(BinarySensorEntity)` for this. In its properties:

```
class CustomOccupancySensor(BinarySensorEntity):
    def __init__(self, api, device_info):
        self.api = api
        self.device = device_info
        self._attr_unique_id = f"{DOMAIN}_sensor_{device_info['id']}"
        self._attr_name = device_info['name']
        self._attr_device_class = BinarySensorDeviceClass.OCCUPANCY
    @property
    def is_on(self):
        # Return True if occupied, False if vacant
        return self.device_status['presence'].lower() == "occupied"
```

You might update `self.device_status` periodically via the coordinator or direct polling. If the sensor has other types (like the `"Front Porch Light Sensor"` with a numeric level), those could be implemented as a regular `SensorEntity` (for illuminance, you could use an illuminance device class or unit if available). But focusing on occupancy, a binary sensor is appropriate.

- **Polling Updates:** Since your API is local and doesn't push updates, these sensor entities will also need periodic polling to update their state. This can be handled by the same mechanisms as lights (discussed next). Typically, you would update their internal state on each poll from the API.

Polling the API for Updates (DataUpdateCoordinator)

Your integration will need to **poll** the local API at regular intervals to get the latest state of lights and sensors, because the API does not push updates (no webhook or subscription support). Home Assistant offers a pattern to manage polling efficiently:

- **DataUpdateCoordinator:** This is a helper class that coordinates fetching data for multiple entities in one go [13] . For example, if the API has an endpoint to get **all lights** (and possibly sensors) at once, you can use one coordinator to fetch all data periodically, and have all your entities use that data. This avoids each entity polling separately and ensures consistency.

How to use it: You would create a subclass or instance of `DataUpdateCoordinator` in your integration. For example, in `__init__.py` after setting up the API, do something like:

```
coordinator = DataUpdateCoordinator(
    hass,
    _LOGGER,
    name="my_lights_and_sensors",
    update_interval=timedelta(seconds=30),
    update_method=async_fetch_data  # your coroutine to fetch all data
)
await coordinator.async_config_entry_first_refresh()  # get initial data
hass.data[DOMAIN][entry.entry_id]['coordinator'] = coordinator
```

Here, `async_fetch_data` is a coroutine you write to call the API (e.g., GET `/lights` and `/sensors`) and return combined data. The coordinator will call this every 30 seconds (in this example) or whatever interval you set. It will handle throttling, error logging, etc. If the first refresh fails, `async_config_entry_first_refresh` will cause the integration setup to retry or mark the entry as not ready.

- **Coordinator Entities:** In your platform files, instead of directly subclassing `LightEntity` or `BinarySensorEntity` alone, you can subclass `CoordinatorEntity` as well. Home Assistant provides a mixin `CoordinatorEntity` that ties an entity to the coordinator. For example:

```
from homeassistant.helpers.update_coordinator import CoordinatorEntity

class CustomLight(CoordinatorEntity, LightEntity):
    def __init__(self, coordinator, light_id):
        super().__init__(coordinator)
        self.light_id = light_id
        # ... set up attributes like name, unique_id as before
    @property
    def is_on(self):
        # Use data from coordinator
        light_data = self.coordinator.data["lights"].get(self.light_id)
        return light_data["level"] > 0   # or use a status field
    @property
    def brightness(self):
        # Calculate brightness from coordinator data
        light_data = self.coordinator.data["lights"].get(self.light_id)
        return int(light_data["level"] * 255 / 65535)
    async def async_turn_on(self, **kwargs):
        # Call API (maybe via coordinator.my_api stored) to turn on light
        await self.coordinator.api.set_light(self.light_id, ...)
        # optionally update coordinator data or request refresh:
        await self.coordinator.async_request_refresh()
```

The `CoordinatorEntity` will automatically handle some things: for example, it sets `should_poll = False` because the entity will update from the coordinator instead of its own polling [14]. It also provides an `async_added_to_hass` hook to add itself as a listener to coordinator updates. When the coordinator fetches new data, it will call `entity.async_write_ha_state()` for each attached entity. The above `is_on` and `brightness` properties always read the latest data from `self.coordinator.data`, so when the coordinator updates, the entity state is refreshed in HA.

- **Polling Interval/SCAN_INTERVAL:** If you choose not to use DataUpdateCoordinator, you can have each entity poll individually by implementing `async_update()` or setting a module-level `SCAN_INTERVAL`. For instance, in a simpler custom platform you might do:

```
SCAN_INTERVAL = timedelta(seconds=30)
class CustomOccupancySensor(BinarySensorEntity):
    # ... properties as above
    async def async_update(self):
        # fetch sensor state from API for just this sensor
        self.device_status = await api.get_sensor(self.device['id'])
```

Home Assistant will call `async_update()` automatically every `SCAN_INTERVAL` seconds for entities that have `should_poll = True` (the default) [15]. However, the coordinator approach is generally preferred if a single API call can get the status of many or all devices at once – it's more efficient to make one bulk request every interval, rather than many small requests.

- **Handling Only Polling:** By default, Home Assistant assumes polling for normal entities. If we had a push/subscribe API we could override `should_poll` to `False` and manually push state updates, but in your case *"the API only supports polling"*, so using the default polling or coordinator (which itself polls) is the right approach [16].

- **Error Handling and Auth Refresh:** Implement your data fetch such that it gracefully handles failures. If an API call fails, you can raise `UpdateFailed` in the coordinator to log a warning without breaking the integration [17]. If the failure is due to an authentication issue (e.g., token expired), you have a couple of options:

- **Automatic Re-auth:** Home Assistant has a mechanism for config entries to handle auth errors. If your coordinator or API wrapper detects an expired token, you can raise a `ConfigEntryAuthFailed` exception [18]. This signals HA that the config entry needs re-authentication, which can trigger a reauth flow (if you implemented one in `config_flow.py`). The user would be prompted to re-login or provide new credentials/token.
- **Silent Token Refresh:** If your API provides a refresh token or a way to get a new token without user intervention, you can catch the auth error and instead call the token refresh method, then retry the data fetch. For example:

```
try:
    data = await api.get_all_devices()
except ExpiredTokenError:
    await api.refresh_token()  # use stored refresh token
    data = await api.get_all_devices()  # retry
```

You can integrate this into the coordinator's `_async_update_data`. If the token has a known expiry time, you could even proactively refresh it on an interval (e.g., schedule a job with `async_track_time_interval` to refresh auth periodically in the background). It depends on your API specifics. Home Assistant doesn't mandate how you refresh; it provides the tools to either handle it internally or escalate to a reauth flow if manual intervention is needed.

Putting It All Together (Step-by-Step Summary)

To summarize the crash course for building this custom component:

1. **Set Up Integration Skeleton:** Create the `custom_components/your_domain/` directory with `manifest.json` (include basic info and `"config_flow": true` for UI config [1]), `__init__.py`, and platform files (`light.py`, `binary_sensor.py`, etc.). Also include a `const.py` for constants (like DOMAIN name, possibly default polling interval, etc.), and optionally `config_flow.py` for UI flow and `translations/` for UI strings.

2. **Config Flow (UI Setup):** In `config_flow.py`, implement a user step (and any other needed steps) to accept configuration. Likely this will prompt for the local API base URL or host, and credentials (or token). Validate the input by attempting a test API call (to ensure the API is reachable and auth is correct). On success, create the config entry (store host and auth info in `config_entry.data`). This makes the integration easily installable via **Settings -> Devices & Services -> Add Integration** in HA, without editing YAML.

3. **Initialize Integration (`async_setup_entry`):** In `__init__.py`, set up the API client using the config entry data. For example, create an HTTP client or use your library to connect to the local API. Store this client in `hass.data[DOMAIN][entry.entry_id]` for access in other parts. Also create

a **DataUpdateCoordinator** to manage polling: define the async function to fetch all data (lights and sensors) from the API, and set an update interval (e.g., 30 seconds, or whatever is appropriate for your use case and API limits). Initialize the coordinator and call `async_config_entry_first_refresh()` to fetch initial data (this ensures that when entities are added, data is available) [19] . If the initial refresh fails (e.g., API down), HA will retry or mark the setup as failed (and you can handle `ConfigEntryNotReady` exceptions for retry logic). Finally, forward the config entry to your platforms:

```
await hass.config_entries.async_forward_entry_setups(entry, ["light",
"binary_sensor"])
```

This tells HA to set up those domains for this entry, which will call each platform's `async_setup_entry`.

4. **Platform Setup (`async_setup_entry` in light.py & binary_sensor.py):** In each platform file, retrieve the coordinator and/or API object from `hass.data[DOMAIN]`. For example:

```
async def async_setup_entry(hass, entry, async_add_entities):
    coordinator = hass.data[DOMAIN][entry.entry_id]["coordinator"]
    api = hass.data[DOMAIN][entry.entry_id]["api"]
    # Create list of entity objects
    entities = []
    for light in coordinator.data["lights"]:  # assuming coordinator.data
has a dict/list of lights from initial fetch
        entities.append(CustomLight(coordinator, light["id"]))
    for sensor in coordinator.data["sensors"]:
        if sensor["subType"] == "OccupancySensor":
            entities.append(CustomOccupancySensor(coordinator,
sensor["id"]))
        elif sensor["subType"] == "PhotoSensor":
            entities.append(CustomIlluminanceSensor(coordinator,
sensor["id"]))
    async_add_entities(entities)
```

This example assumes you fetched data already. If you are not using a coordinator, you might directly call `api.get_lights()` here to get data, but using the coordinator (which already fetched) is more efficient. In either case, create entity instances for each device and use `async_add_entities(...)` to register them with HA. By doing this at startup, you **"search for new lights when the component starts up"** automatically (loading all devices into HA entities).

5. **Entity Implementation:** As discussed, implement the LightEntity and BinarySensorEntity classes with the appropriate properties (`is_on`, `brightness`, etc.) and methods (`async_turn_on/off`). Make sure each entity has:

6. A `unique_id` (so the entity is tracked in the entity registry).
7. A `name` (either set `_attr_name` or define a `name` property).
8. A proper `device_info` linking it to a device (common for grouping entities). Use the API's device identifiers. For example, if all lights are controlled by a single hub, you might have one device (the hub) with `via_device` relationships. But if each light is its own device, you can simply use the light's ID as an identifier. Include `suggested_area` if you have room names [6]. For instance, if the API provides `"roomId": 3` and you know room 3 is "Kitchen", set `suggested_area="Kitchen"` for that device.

9. Proper handling of state updates: If using CoordinatorEntity, override `_handle_coordinator_update` or simply rely on properties pulling from `coordinator.data` to update state when data changes [20]. If not using a coordinator, ensure `async_update` pulls fresh data. In both cases, Home Assistant will periodically call your update logic (either via the coordinator or via polling intervals).

10. **Token Refresh Mechanism:** If the local API uses an auth token that can expire, incorporate a refresh strategy:

11. If the token's lifetime is known, you might schedule a periodic refresh. For example, if it's valid for an hour, schedule a refresh job 5 minutes before expiry each time. Use `hass.helpers.event.async_track_time_interval` or `async_call_later` to schedule these in `async_setup_entry`. This function can call your API client's refresh and update the stored token.
12. If you only know a token expired when an API call fails, then handle exceptions in your API calls. As noted, if an API call raises an auth error, you can either attempt a refresh inside the coordinator's `_async_update_data` or raise `ConfigEntryAuthFailed` [18] to let HA prompt the user to re-auth (depends on whether the user needs to log in again or if you can do it automatically).

13. For simplicity, many custom integrations implement a refresh in the coordinator: e.g., every time `_async_update_data` runs, check if now > token_expiry and refresh it before making the data call. This keeps token management in one place.

14. **Assigning Areas:** As mentioned, use the `suggested_area` field in device registration. When you create the device (either via returning `device_info` in each entity, or manually using `device_registry.async_get_or_create`), you can pass a name for the area [6]. For example, if your API has `roomId` and you have a mapping to room names (perhaps via another API call or a static map), you can do:

```python
@property
def device_info(self):
    return {
        "identifiers": {(DOMAIN, self.device_id)},
        "name": self.device_name,
        "manufacturer": "MyLightingSystem",
        "model": "CustomLight",
        "suggested_area": self.device_room_name  # e.g., "Living Room"
    }
```

Home Assistant will use this to place the device in the "Living Room" area if that area exists (or it might create it, or at least suggest it in the UI). The user can always change areas later, but this provides a convenient default assignment.

15. **Testing and Iteration:** Develop with logging enabled (use `_LOGGER.debug` generously) to ensure your integration is fetching data and updating states as expected. You can install the custom component in HA and watch the logs for any errors. Use the Developer Tools in HA to verify that entities appear and update.

Additional Tips

- **No YAML Mode:** By using config entries (the UI flow), you avoid requiring the user to edit configuration.yaml. Ensure your `manifest.json` has `"config_flow": true` and you implement the `async_setup_entry` (and `async_unload_entry` for cleanup) so that the integration can be managed entirely from the UI. You do *not* need to define anything in `configuration.yaml` (unless you want to support an optional YAML setup, which is usually not needed for new integrations [21]).
- **Cleanup on Removal:** Optionally implement `async_unload_entry` in `__init__.py` to handle when the user removes the integration. This would unload platforms (use `hass.config_entries.async_unload_platforms`) and clean up any timers or tasks (e.g., cancel the token refresh interval if you set one).
- **Use of Constants:** Define constants for things like your integration DOMAIN, perhaps default scan interval, etc., in a `const.py` for clarity. E.g., `DOMAIN = "my_custom_lights"`, `SCAN_INTERVAL = timedelta(seconds=30)`.
- **Leverage Existing Examples:** The Home Assistant developer docs and community have many examples. The official *"Creating your first integration"* guide and the DataUpdateCoordinator docs are very useful [13] [22] . You can also find community examples of custom light components (for instance, ones that implement dimmers) to see how they handle brightness and transitions.

By following this approach, you'll have a custom integration that **dynamically discovers lights and sensors**, updates their state via polling, handles authentication renewal, and is configurable through the Home Assistant UI. This will expose your custom lights (with brightness and fade support) and occupancy sensors as first-class citizens in Home Assistant, allowing you to control them and incorporate them into automations just like any native device. Enjoy building your integration!

References

- Home Assistant Developer Docs – *Integration File Structure* (how to structure custom components with platforms) [2]
- Home Assistant Developer Docs – *Config Flow* (enabling UI configuration for integrations) [1]
- Home Assistant Developer Docs – *Fetching Data* (using DataUpdateCoordinator for polling APIs) [13] [15]

- Home Assistant Developer Docs – *DataUpdateCoordinator Example* (coordinator + CoordinatorEntity usage) [23] [20]
- Home Assistant Developer Docs – *Light Entity* (implementing lights, brightness scaling, transitions) [7] [9] [10]
- Home Assistant Developer Docs – *Binary Sensor* (device classes for binary sensors like occupancy) [12]
- Home Assistant Developer Docs – *Device Registry* (using device_info with suggested_area for area assignment) [6]
- Home Assistant Developer Docs – *Config Entry Auth* (raising ConfigEntryAuthFailed on token failure to trigger reauth) [18]

---

[1] [3] [21] Config flow | Home Assistant Developer Docs
https://developers.home-assistant.io/docs/config_entries_config_flow_handler/

[2] Integration file structure | Home Assistant Developer Docs
https://developers.home-assistant.io/docs/creating_integration_file_structure/

[4] [5] [11] [13] [14] [15] [16] [17] [18] [19] [20] [22] [23] Fetching data | Home Assistant Developer Docs
https://developers.home-assistant.io/docs/integration_fetching_data/

[6] Device registry | Home Assistant Developer Docs
https://developers.home-assistant.io/docs/device_registry_index/

[7] [8] [9] [10] Light entity | Home Assistant Developer Docs
https://developers.home-assistant.io/docs/core/entity/light/

[12] Binary sensor entity | Home Assistant Developer Docs
https://developers.home-assistant.io/docs/core/entity/binary-sensor/