Entity

For a generic introduction of entities, see entities architecture.

Basic implementation

Below is an example switch entity that keeps track of its state in memory. In addition, the switch in the example represents the main feature of a device, meaning the entity has the same name as its device.

Please refer to Entity naming for how to give an entity its own name.

```
from homeassistant.components.switch import SwitchEntity


class MySwitch(SwitchEntity):
    _attr_has_entity_name = True

    def __init__(self):
        self._is_on = False
        self._attr_device_info = ...  # For automatic device registration
        self._attr_unique_id = ...

    @property
    def is_on(self):
        """If the switch is currently on or off."""
        return self._is_on

    def turn_on(self, **kwargs):
        """Turn the switch on."""
        self._is_on = True

    def turn_off(self, **kwargs):
        """Turn the switch off."""
        self._is_on = False
```

That's all there is to it to build a switch entity! Continue reading to learn more or check out the video tutorial.

Updating the entity

An entity represents a device. There are various strategies to keep your entity in sync with the state of the device, the most popular one being polling.

Polling

With polling, Home Assistant will ask the entity from time to time (depending on the update interval of the component) to fetch the latest state. Home Assistant will poll an entity when the `should_poll` property returns `True` (the default value). You can either implement your update logic using `update()` or the async method `async_update()`. This method should fetch the latest state from the device and store it in an instance variable for the properties to return it.

Subscribing to updates

When you subscribe to updates, your code is responsible for letting Home Assistant know that an update is available. Make sure you have the `should_poll` property return `False`.

Whenever you receive a new state from your subscription, you can tell Home Assistant that an update is available by calling `schedule_update_ha_state()` or async callback `async_schedule_update_ha_state()`. Pass in the boolean `True` to the method if you want Home Assistant to call your update method before writing the update to Home Assistant.

Generic properties

The entity base class has a few properties common among all Home Assistant entities. These properties can be added to any entity regardless of the type. All these properties are optional and don't need to be implemented.

These properties are always called when the state is written to the state machine.

TIP

Properties should always only return information from memory and not do I/O (like network requests). Implement `update()` or `async_update()` to fetch data.

Because these properties are always called when the state is written to the state machine, it is important to do as little work as possible in the property.

To avoid calculations in a property method, set the corresponding entity class or instance attribute, or if the values never change, use entity descriptions.

Name	Type	Default	Description
assumed_state	`bool`	`False`	Return `True` if the state is based on our assumption instead of reading it from the device.
attribution	`str | None`	`None`	The branding text required by the API provider.
available	`bool`	`True`	Indicate if Home Assistant is able to read the state or control the underlying device, see entity-unavailable for more details.
device_class	`str | None`	`None`	Extra classification of what the device is. Each domain specifies their own. Device classes can come with extra requirements for unit of measurement and supported features.
entity_picture	`str | None`	`None`	Url of a picture to show for the entity.
extra_state_attributes	`dict | None`	`None`	Extra information to store in the state machine. It needs to be information that further explains the state, it

Name	Type	Default	Description
			should not be static information like firmware version.
has_entity_name	`bool`	`False`	Return `True` if the entity's `name` property represents the entity itself (required for new integrations). This is explained in more detail below.
name	`str | None`	`None`	Name of the entity. Avoid hard coding a natural language name, use a translated name instead.
should_poll	`bool`	`True`	Should Home Assistant check with the entity for an updated state. If set to `False`, entity will need to notify Home Assistant of new updates by calling one of the schedule update methods.
state	`str | int | float | None`	`None`	The state of the entity. In most cases this is implemented by the domain base entity and should not be implemented by integrations.
supported_features	`int | None`	`None`	Flag features supported by the entity. Domains specify their own.
translation_key	`str | None`	`None`	A key for looking up translations of the entity's state in `entity` section of the integration's `strings.json` and for translating the state into a matching icon.
translation_placeholders	`dict | None`	`None`	Placeholder definitions for translated entity name.

⚠ WARNING

It's allowed to change `device_class`, `supported_features` or any property included in a domain's `capability_attributes`. However, since these entity properties often are not expected to change at all and some entity consumers may not be able to update them at a free rate, we recommend only changing them when absolutely required and at a modest interval.

As an example, such changes will cause voice assistant integrations to resynchronize with the supporting cloud service.

⚠ WARNING

Entities that generate a significant amount of state changes can quickly increase the size of the database when the `extra_state_attributes` also change frequently. Minimize the number of `extra_state_attributes` for these entities by removing non-critical attributes or creating additional `sensor` entities.

Registry properties

The following properties are used to populate the entity and device registries. They are read each time the entity is added to Home Assistant. These properties only have an effect if `unique_id` is not None.

Name	Type	Default	Description
device_info	`DeviceInfo | None`	`None`	Device registry descriptor ' automatic device registrat
entity_category	`EntityCategory | None`	`None`	Classification of a non-prir entity. Set to `EntityCategory.CONFIG` an entity that allows chanç the configuration of a devi for example, a switch entit

Name	Type	Default	Description
			making it possible to turn background illumination o switch on and off. Set to `EntityCategory.DIAGNOS` for an entity that exposes some configuration param or diagnostics of a device l does not allow changing it example, a sensor showing RSSI or MAC address. Use also for button entities tha trigger a device identificat mechanism (with `IDENTIF` device class).
entity_registry_enabled_default	`bool`	`True`	Indicate if the entity shoul enabled or disabled when added to the entity registr This includes fast-changing diagnostic entities or, assumingly less commonly used entities. For example sensor exposing RSSI or battery voltage should typically be set to `False`; t prevent unneeded (record state changes or UI clutter these entities.
entity_registry_visible_default	`bool`	`True`	Indicate if the entity shoul hidden or visible when firs added to the entity registr
unique_id	`str | None`	`None`	A unique identifier for this entity. It must be unique

Name	Type	Default	Description
			within a platform (like `light.hue`). It should not configurable or changeabl the user. Learn more.

Additional properties

The following properties are also available on entities, but should be used with caution. These properties are always called when the state is written to the state machine.

Name	Type	Default	Description
capability_attributes	`dict | None`	`None`	State attributes which are stored in the entity registry. This property is implemented by the domain base entity and should not be implemented by integrations.
force_update	`bool`	`False`	Write each update to the state machine, even if the data is the same. Example use: when you are directly reading the value from a connected sensor instead of a cache. Use with caution, will spam the state machine.
icon	`str |`	`None`	Icon to use in the frontend. Using this

Name	Type	Default	Description
	`None`		property is not recommended. More information about using icons.
state_attributes	`dict | None`	`None`	State attributes of a base domain. This property is implemented by the domain base entity and should not be implemented by integrations.
unit_of_measurement	`str | None`	The unit of measurement that the entity's state is expressed in. In most cases, for example for the `number` and `sensor` domains, this is implemented by the domain base entity and should not be implemented by integrations.	

System properties

The following properties are used and controlled by Home Assistant, and should not be overridden by integrations.

Name	Type	Default	Description
enabled	`bool`	`True`	Indicate if entity is enabled in the entity registry. It also returns `True` if the platform doesn't support the entity registry. Disabled entities will not be added to Home Assistant.

Entity naming

Avoid setting an entity's name to a hard coded English string, instead, the name should be translated. Examples of when the name should not be translated are proper nouns, model names, and name provided by a 3rd-party library.

Some entities are automatically named after their device class, this includes `binary_sensor`, `button`, `number` and `sensor` entities and in many cases don't need to be named. For example, an unnamed sensor which has its device class set to `temperature` will be named "Temperature".

> **NOTE**
>
> If an entity provides translations for the entity name, the used name depends on the system (backend) language at creation time, not the user's UI language. For example, if your backend is set to German, new entities will be named in German — even if a user later switches their UI to French. Changing the backend language will only affect entities created after the change; existing entities retain their original names.

`has_entity_name` True (Mandatory for new integrations)

The entity's name property only identifies the data point represented by the entity, and should not include the name of the device or the type of the entity. So for a sensor that represents the power usage of its device, this would be "Power usage".

If the entity represents a single main feature of a device the entity should typically have its name property return `None`. The "main feature" of a device would for example be the `LightEntity` of a smart light bulb.

The `friendly_name` state attribute is generated by combining the entity name with the device name as follows:

- The entity is not a member of a device: `friendly_name = entity.name`
- The entity is a member of a device and `entity.name` is not `None`: `friendly_name = f"{device.name} {entity.name}"`
- The entity is a member of a device and `entity.name` is `None`: `friendly_name = f"{device.name}"`

The `entity_id` is generated by combining the entity name with the device name as follows:

- The entity is not a member of a device, for example, a helper "Everyone is home": `entity_id = binary_sensor.everyone_is_home`
- The entity is a member of a device and `entity.name` is not `None`, for example, the battery of device named "nightlight": `entity_id = sensor.nightlight_battery`
- The entity is a member of a device and `entity.name` is `None`, for example, the light of a device named "nightlight": `entity_id = light.nightlight`

Entity names should start with a capital letter, the rest of the words are lower case (unless it's a proper noun or a capitalized abbreviation of course).

Example of a switch entity which is the main feature of a device

Note: The example is using class attributes to implement properties, for other ways to implement properties see Property implementation. *Note: The example is incomplete, the `unique_id` property must be implemented, and the entity must be registered with a device.

```
from homeassistant.components.switch import SwitchEntity


class MySwitch(SwitchEntity):
    _attr_has_entity_name = True
    _attr_name = None
```

Example of a switch entity which is either not the main feature of a device, or is not part of a device:

Note: The example is using class attributes to implement properties, for other ways to implement properties see Property implementation. *Note: If the entity is part of a device, the `unique_id` property must be implemented, and the entity must be registered with a device.

```
from homeassistant.components.switch import SwitchEntity


class MySwitch(SwitchEntity):
    _attr_has_entity_name = True

    @property
    def translation_key(self):
        """Return the translation key to translate the entity's name and
states."""
        return "my_switch"
```

Example of an untranslated switch entity which is either not the main feature of a device, or is not part of a device:

```
from homeassistant.components.switch import SwitchEntity


class MySwitch(SwitchEntity):
    _attr_has_entity_name = True

    @property
    def name(self):
        """Name of the entity."""
        return "Model X"
```

`has_entity_name` not implemented or False (Deprecated)

The entity's name property may be a combination of the device name and the data point represented by the entity.

Property implementation

Property function

Writing property methods for each property is just a couple of lines of code, for example

```
class MySwitch(SwitchEntity):

    @property
    def icon(self) -> str | None:
        """Icon of the entity."""
        return "mdi:door"

    ...
```

Entity class or instance attributes

Alternatively, a shorter form is to set Entity class or instance attributes according to either of the following patterns:

```
class MySwitch(SwitchEntity):

    _attr_icon = "mdi:door"

    ...
```

```
class MySwitch(SwitchEntity):

    def __init__(self, icon: str) -> None:
        self._attr_icon = icon

    ...
```

This does exactly the same as the first example but relies on a default implementation of the property in the base class. The name of the attribute starts with `_attr_` followed by the property name. For example, the default `device_class` property returns the `_attr_device_class` class attribute.

Not all entity classes support the `_attr_` attributes for their entity specific properties, please refer to the documentation for the respective entity class for details.

> **TIP**
>
> If an integration needs to access its own properties it should access the property (`self.name`), not the class or instance attribute (`self._attr_name`).

Entity description

The third way of setting entity properties is to use an entity description. To do this set an attribute named `entity_description` on the `Entity` instance with an `EntityDescription` instance. The entity description is a dataclass with attributes corresponding to most of the available `Entity` properties. Each entity integration that supports an entity platform, eg the `switch` integration, will define their own `EntityDescription` subclass that should be used by implementing platforms that want to use entity descriptions.

By default the `EntityDescription` instance has one required attribute named `key`. This is a string which is meant to be unique for all the entity descriptions of an implementing platform. A common use case for this attribute is to include it in the `unique_id` of the described entity.

The main benefit of using entity descriptions is that it defines the different entity types of a platform in a declarative manner, making the code much easier to read when there are many different entity types.

Example

The below code snippet gives an example of best practices for when to implement property functions, when to use class or instance attributes and when to use entity descriptions.

```
from __future__ import annotations

from collections.abc import Callable
from dataclasses import dataclass

from example import ExampleDevice, ExampleException

from homeassistant.components.sensor import (
```

```
    SensorDeviceClass,
    SensorEntity,
    SensorEntityDescription,
    SensorStateClass,
)
from homeassistant.config_entries import ConfigEntry
from homeassistant.const import (
    EntityCategory,
    UnitOfElectricCurrent,
)
from homeassistant.core import HomeAssistant
from homeassistant.helpers.entity_platform import
AddConfigEntryEntitiesCallback
from homeassistant.helpers.typing import StateType

from .const import DOMAIN, LOGGER


@dataclass(kw_only=True)
class ExampleSensorEntityDescription(SensorEntityDescription):
    """Describes Example sensor entity."""

    exists_fn: Callable[[ExampleDevice], bool] = lambda _: True
    value_fn: Callable[[ExampleDevice], StateType]


SENSORS: tuple[ExampleSensorEntityDescription, ...] = (
    ExampleSensorEntityDescription(
        key="estimated_current",
        native_unit_of_measurement=UnitOfElectricCurrent.MILLIAMPERE,
        device_class=SensorDeviceClass.CURRENT,
        state_class=SensorStateClass.MEASUREMENT,
        value_fn=lambda device: device.power,
        exists_fn=lambda device: bool(device.max_power),
    ),
)


async def async_setup_entry(
    hass: HomeAssistant,
    entry: ConfigEntry,
    async_add_entities: AddConfigEntryEntitiesCallback,
) -> None:
    """Set up Example sensor based on a config entry."""
```

```
    device: ExampleDevice = entry.runtime_data
    async_add_entities(
        ExampleSensorEntity(device, description)
        for description in SENSORS
        if description.exists_fn(device)
    )


class ExampleSensorEntity(SensorEntity):
    """Represent an Example sensor."""

    entity_description: ExampleSensorEntityDescription
    _attr_entity_category = (
        EntityCategory.DIAGNOSTIC
    )  # This will be common to all instances of ExampleSensorEntity

    def __init__(
        self, device: ExampleDevice, entity_description:
ExampleSensorEntityDescription
    ) -> None:
        """Set up the instance."""
        self._device = device
        self.entity_description = entity_description
        self._attr_available = False  # This overrides the default
        self._attr_unique_id = f"{device.serial}_{entity_description.key}"

    def update(self) -> None:
        """Update entity state."""
        try:
            self._device.update()
        except ExampleException:
            if self.available:  # Read current state, no need to prefix
with _attr_
                LOGGER.warning("Update failed for %s", self.entity_id)
            self._attr_available = False  # Set property value
            return

        self._attr_available = True
        # We don't need to check if device available here
        self._attr_native_value = self.entity_description.value_fn(
            self._device
        )  # Update "native_value" property
```

Lifecycle hooks

Use these lifecycle hooks to execute code when certain events happen to the entity. All lifecycle hooks are async methods.

`async_added_to_hass()`

Called when an entity has their entity_id and hass object assigned, before it is written to the state machine for the first time. Example uses: restore the state, subscribe to updates or set callback/dispatch function/listener.

`async_will_remove_from_hass()`

Called when an entity is about to be removed from Home Assistant. Example use: disconnect from the server or unsubscribe from updates.

Icons

Every entity in Home Assistant has an icon, which is used as a visual indicator to identify the entity more easily in the frontend. Home Assistant uses the Material Design Icons icon set.

In most cases, Home Assistant will pick an icon automatically based on the entity's domain, `device_class`, and `state`. It is preferred to use the default icon if possible, to provide a consistent experience and to avoid confusion for the user. However, it is possible to override the default and provide a custom icon for an entity.

Regardless of the provided icon, it is always possible for the user to customize the icon to their liking in the frontend.

There are two ways to provide a custom icon for an entity, either by providing icon translations or by providing an icon identifier.

Icon translations

This is the preferred way to provide a custom icon for an entity. Icon translations work similarly to our regular translations, but instead of translating the state of an entity, they

translate the states of an entity to icons.

Note that translated states must be `snake_case` just like all other translation keys.

The `translation_key` property of an entity defines the icon translation to use. This property is used to look up the translation in the `entity` section of the integration's `icons.json` file.

To differentiate entities and their translations, provide different translation keys. The following example shows `icons.json` for a Moon domain `sensor` entity with its `translation_key` property set to phase:

```
{
  "entity": {
    "sensor": {
      "phase": {
        "default": "mdi:moon",
        "state": {
          "new_moon": "mdi:moon-new",
          "first_quarter": "mdi:moon-first-quarter",
          "full_moon": "mdi:moon-full",
          "last_quarter": "mdi:moon-last-quarter"
        }
      }
    }
  }
}
```

Notice that icons start with `mdi:` plus an identifier. The `default` icon is used when the entity's state is not in the `state` section. The `state` section is optional, and if not provided, the `default` icon will be used for all states.

Icons for entity state attributes can also be provided in cases where the frontend shows icons for the state attributes. Examples include climate presets and fan modes. It's not possible to provide icons for other state attributes. The following example provides icons for a `climate` entity with its `translation_key` property set to `ubercool`. This entity has a `preset_mode` state attribute, which can be set to `vacation` or `night`. The frontend will use these in, for example, the climate card.

Note that translated state attributes must be `snake_case` just like all other translation keys.

```
{
  "entity": {
    "climate": {
      "ubercool": {
        "state_attributes": {
          "preset_mode": {
            "default": "mdi:confused",
            "state": {
              "vacation": "mdi:umbrella-beach",
              "night": "mdi:weather-night"
            }
          }
        }
      }
    }
  }
}
```

Icon property

Another way to provide an icon for an entity is by setting the `icon` property of an entity, which returns a string referencing the `mdi` icon. As this property is a method, it is possible to return different icons based on custom logic unlike with icon translations. For example, it's possible to calculate the icon based on the state as in the example below, or return different icons based on something that is not part of the entity's state.

```python
class MySwitch(SwitchEntity):

    @property
    def icon(self) -> str | None:
        """Icon of the entity, based on time."""
        if now().hour < 12:
            return "mdi:weather-night"
        return "mdi:weather-sunny"

    ...
```

It is not possible to provide icons for state attributes using the `icon` property. Please note that using the `icon` property is discouraged; using the above-mentioned icon translations is

preferred.

Excluding state attributes from recorder history

State attributes which are not suitable for state history recording should be excluded from state history recording by including them in either of `_entity_component_unrecorded_attributes` or `_unrecorded_attributes`.

- `_entity_component_unrecorded_attributes: frozenset[str]` may be set in a base component class, for example, in `light.LightEntity`
- `_unrecorded_attributes: frozenset[str]` may be set in an integration's platform, for example, in an entity class defined in platform `hue.light`.

The `MATCH_ALL` constant can be used to exclude all attributes instead of typing them separately. This can be useful for integrations providing unknown attributes or when you simply want to exclude all without typing them separately.

Using the `MATCH_ALL` constant does not stop recording for `device_class`, `state_class`, `unit_of_measurement`, and `friendly_name` as they might also serve other purposes and, therefore, should not be excluded from recording.

Examples of platform state attributes which are excluded from recording include the `entity_picture` attribute of `image` entities which will not be valid after some time, the `preset_modes` attribute of `fan` entities which is not likely to change. Examples of integration specific state attributes which are excluded from recording include `description` and `location` state attributes in platform `trafikverket.camera` which do not change.

> **TIP**
> The `_entity_component_unrecorded_attributes` and `_unrecorded_attributes` must be declared as class attributes; instance attributes will be ignored.

Changing the entity model

If you want to add a new feature to an entity or any of its subtypes (light, switch, etc), you will need to propose it first in our architecture repo. Only additions will be considered that are common features among various vendors.

*Last updated on **Jun 17, 2026***