Data entry flow

Data Entry Flow is a data entry framework that is part of Home Assistant. Data entry is done via data entry flows. A flow can represent a simple login form or a multi-step setup wizard for a component. A Flow Manager manages all flows that are in progress and handles creation of new flows.

Data Entry Flow is used in Home Assistant to login, create config entries, handle options flow, repair issues.

Flow manager

This is the class that manages the flows that are in progress. When instantiating one, you pass in two async callbacks:

```
async def async_create_flow(handler, context=context, data=data):
    """Create flow."""
```

The manager delegates instantiating of config flow handlers to this async callback. This allows the parent of the manager to define their own way of finding handlers and preparing a handler for instantiation. For example, in the case of the config entry manager, it will make sure that the dependencies and requirements are setup.

```
async def async_finish_flow(flow, result):
    """Finish flow."""
```

This async callback is called when a flow is finished or aborted. i.e. `result['type'] in [FlowResultType.CREATE_ENTRY, FlowResultType.ABORT]`. The callback function can modify result and return it back, if the result type changed to `FlowResultType.FORM`, the flow will continue running, display another form.

If the result type is `FlowResultType.FORM`, the result should look like:

```
{
    # The result type of the flow
    "type": FlowResultType.FORM,
    # the id of the flow
    "flow_id": "abcdfgh1234",
    # handler name
    "handler": "hue",
    # name of the step, flow.async_step_[step_id] will be called when form submitted
    "step_id": "init",
    # a voluptuous schema to build and validate user input
    "data_schema": vol.Schema(),
    # an errors dict, None if no errors
    "errors": errors,
    # a detail information about the step
    "description_placeholders": description_placeholders,
}
```

If the result type is `FlowResultType.CREATE_ENTRY`, the result should look like:

```
{
    # Data schema version of the entry
    "version": 2,
    # The result type of the flow
    "type": FlowResultType.CREATE_ENTRY,
```

```
    # the id of the flow
    "flow_id": "abcdfgh1234",
    # handler name
    "handler": "hue",
    # title and data as created by the handler
    "title": "Some title",
    "result": {
        "some": "data"
    },
}
```

If the result type is `FlowResultType.ABORT`, the result should look like:

```
{
    # The result type of the flow
    "type": FlowResultType.ABORT,
    # the id of the flow
    "flow_id": "abcdfgh1234",
    # handler name
    "handler": "hue",
    # the abort reason
    "reason": "already_configured",
}
```

Flow handler

Flow handlers will handle a single flow. A flow contains one or more steps. When a flow is instantiated, the `FlowHandler.init_step` step will be called. Each step has several possible results:

- Show Form
- Create Entry
- Abort
- External Step
- Show Progress
- Show Menu

At a minimum, each flow handler will have to define a version number and a step. This doesn't have to be `init`, as `async_create_flow` can assign `init_step` dependent on the current workflow, for example in configuration, `context.source` will be used as `init_step`.

For example, a bare minimum config flow would be:

```
from homeassistant import data_entry_flow

@config_entries.HANDLERS.register(DOMAIN)
class ExampleConfigFlow(data_entry_flow.FlowHandler):

    # The schema version of the entries that it creates
    # Home Assistant will call your migrate method if the version changes
    # (this is not implemented yet)
    VERSION = 1

    async def async_step_user(self, user_input=None):
        """Handle user step."""
```

Data entry flows depend on translations for showing the text in the steps. It depends on the parent of a data entry flow manager where this is stored. For config and option flows, this is in `strings.json` under `config` and `option`, respectively.

For a more detailed explanation of `strings.json` see the backend translation page.

Show form

This result type will show a form to the user to fill in. You define the current step, the schema of the data (using a mixture of voluptuous and/or selectors) and optionally a dictionary of errors.

```
from homeassistant.data_entry_flow import section
from homeassistant.helpers.selector import selector

class ExampleConfigFlow(data_entry_flow.FlowHandler):
    async def async_step_user(self, user_input=None):
        # Specify items in the order they are to be displayed in the UI
        data_schema = {
            vol.Required("username"): str,
            vol.Required("password"): str,
            # Items can be grouped by collapsible sections
            vol.Required("ssl_options"): section(
                vol.Schema(
                    {
                        vol.Required("ssl", default=True): bool,
                        vol.Required("verify_ssl", default=True): bool,
                    }
                ),
                # Whether or not the section is initially collapsed (default = False)
                {"collapsed": False},
            )
        }

        if self.show_advanced_options:
            data_schema[vol.Optional("allow_groups")] = selector({
```

```
                "select": {
                    "options": ["all", "light", "switch"],
                }
            })

        return self.async_show_form(step_id="init", data_schema=vol.Schema(data_schema))
```

Grouping of input fields

As shown in the example above, input fields can be visually grouped in sections.

Each section has a translatable name and description, and it's also possible to specify an icon.

Grouping input fields by sections influences both how the inputs are displayed to the user and how user input is structured. In the example above, user input will be structured like this:

```
{
    "username": "user",
    "password": "hunter2",
    "ssl_options": {
        "ssl": True,
        "verify_ssl": False,
    },
}
```

Only a single level of sections is allowed; it's not possible to have sections inside a section.

To specify an icon for a section, update `icons.json` according to this example:

```
{
  "config": {
    "step": {
      "user": {
        "sections": {
          "ssl_options": "mdi:lock"
        }
      }
    }
  }
}
```

Labels & descriptions

Translations for the form are added to `strings.json` in a key for the `step_id`. That object may contain the folowing keys:

Key	Value	Notes
`title`	Form heading	Do not include your brand name. It will be automatically injected from your manifest.
`description`	Form instructions	Optional. Do not link to the documentation as that is linked automatically. Do not include "basic" information like "Here you can set up X".
`data`	Field labels	Keep succinct and consistent with other integrations whenever appropriate for the best user experience.
`data_description`	Field descriptions	Optional explanatory text to show below the field.

Key	Value	Notes
`section`	Section translation	Translations for sections, each section may have `name` and `description` of the section and `data` and `data_description` for its fields.

More details about translating data entry flows can be found in the core translations documentation.

The field labels and descriptions are given as a dictionary with keys corresponding to your schema. Here is a simple example:

```
{
  "config": {
    "step": {
      "user": {
          "title": "Add Group",
          "description": "Some description",
          "data": {
              "entities": "Entities"
          },
          "data_description": {
              "entities": "The entities to add to the group"
          },
          "sections": {
              "additional_options": {
                  "name": "Additional options",
                  "description": "A description of the section",
                  "data": {
                      "advanced_group_option": "Advanced group option"
                  },
                  "data_description": {
                      "advanced_group_option": "A very complicated option which does abc"
```

```
                },
            }
        }
    }
  }
}
```

Enabling browser autofill

Suppose your integration is collecting form data which can be automatically filled by browsers or password managers, such as login credentials or contact information. You should enable autofill whenever possible for the best user experience and accessibility. There are two options to enable this.

The first option is to use Voluptuous with data keys recognized by the frontend. The frontend will recognize the keys `"username"` and `"password"` and add HTML `autocomplete` attribute values of `"username"` and `"current-password"` respectively. Support for autocomplete is limited to `"username"` and `"password"` fields and is supported primarily to quickly enable auto-fill on the many integrations that collect them without converting their schemas to selectors.

The second option is to use a text selector. A text selector gives full control of the input type and allows any permitted value for `autocomplete` to be specified. A hypothetical schema collecting specific fillable data might be:

```
import voluptuous as vol
from homeassistant.const import CONF_PASSWORD, CONF_USERNAME
from homeassistant.helpers.selector import (
    TextSelector,
    TextSelectorConfig,
    TextSelectorType,
)
```

```
STEP_USER_DATA_SCHEMA = vol.Schema(
    {
        vol.Required(CONF_USERNAME): TextSelector(
            TextSelectorConfig(type=TextSelectorType.EMAIL, autocomplete="username")
        ),
        vol.Required(CONF_PASSWORD): TextSelector(
            TextSelectorConfig(
                type=TextSelectorType.PASSWORD, autocomplete="current-password"
            )
        ),
        vol.Required("postal_code"): TextSelector(
            TextSelectorConfig(type=TextSelectorType.TEXT, autocomplete="postal-code")
        ),
        vol.Required("mobile_number"): TextSelector(
            TextSelectorConfig(type=TextSelectorType.TEL, autocomplete="tel")
        ),
    }
)
```

Defaults & suggestions

If you'd like to pre-fill data in the form, you have two options. The first is to use the `default` parameter. This will both pre-fill the field, and act as the default value in case the user leaves the field empty.

```
    data_schema = {
        vol.Optional("field_name", default="default value"): str,
    }
```

The other alternative is to use a suggested value - this will also pre-fill the form field, but will allow the user to leave it empty if the user so wishes.

```
    data_schema = {
        vol.Optional(
            "field_name", description={"suggested_value": "suggested value"}
        ): str,
    }
```

You can also mix and match - pre-fill through `suggested_value`, and use a different value for `default` in case the field is left empty, but that could be confusing to the user so use carefully.

Using suggested values also make it possible to declare a static schema, and merge suggested values from existing input. A `add_suggested_values_to_schema` helper makes this possible:

```
OPTIONS_SCHEMA = vol.Schema(
    {
        vol.Optional("field_name", default="default value"): str,
    }
)

class ExampleOptionsFlow(config_entries.OptionsFlow):
    async def async_step_init(
        self, user_input: dict[str, Any] | None = None
    ) -> FlowResult:
        return self.async_show_form(
            data_schema = self.add_suggested_values_to_schema(
                OPTIONS_SCHEMA, self.entry.options
            )
        )
```

Note: For select type inputs (created from a `vol.In(...)` schema), if no `default` is specified, the first option will be selected by default in the frontend.

Displaying read-only information

Some integrations have options which are frozen after initial configuration. When displaying an options flow, you can show this information in a read-only way, so that users may remember which options were selected during the initial configuration. For this, define an optional selector as usual, but with the `read_only` flag set to `True`.

```
# Example Config Flow Schema
DATA_SCHEMA_SETUP = vol.Schema(
    {
        vol.Required(CONF_ENTITY_ID): EntitySelector()
    }
)

# Example Options Flow Schema
DATA_SCHEMA_OPTIONS = vol.Schema(
    {
        vol.Optional(CONF_ENTITY_ID): EntitySelector(
            EntitySelectorConfig(read_only=True)
        ),
        vol.Optional(CONF_TEMPLATE): TemplateSelector(),
    }
)
```

This will show the entity selected in the initial configuration as a read-only property whenever the options flow is launched.

Validation

After the user has filled in the form, the step method will be called again and the user input is passed in. Your step will only be called if the user input passes your data schema. When the user passes in data, you will have to do extra validation of the data. For example, you can verify that the passed in username and password are valid.

If something is wrong, you can return a dictionary with errors. Each key in the error dictionary refers to a field name that contains the error. Use the key `base` if you want to show an error unrelated to a specific field. The specified errors need to refer to a key in a translation file.

```
class ExampleConfigFlow(data_entry_flow.FlowHandler):
    async def async_step_user(self, user_input=None):
        errors = {}
        if user_input is not None:
            # Validate user input
            valid = await is_valid(user_input)
            if valid:
                # See next section on create entry usage
                return self.async_create_entry(...)

            errors["base"] = "auth_error"

        # Specify items in the order they are to be displayed in the UI
        data_schema = {
            vol.Required("username"): str,
            vol.Required("password"): str,
        }

        return self.async_show_form(
            step_id="init", data_schema=vol.Schema(data_schema), errors=errors
        )
```

Multi-step flows

If the user input passes validation, you can return one of the possible step types again. If you want to navigate the user to the next step, return the return value of that step:

```
class ExampleConfigFlow(data_entry_flow.FlowHandler):
    async def async_step_init(self, user_input=None):
        errors = {}
        if user_input is not None:
            # Validate user input
            valid = await is_valid(user_input)
            if valid:
                # Store info to use in next step
                self.init_info = user_input
                # Return the form of the next step
                return await self.async_step_account()

        ...
```

Create entry

When the result is "Create Entry", an entry will be created and passed to the parent of the flow manager. A success message is shown to the user and the flow is finished. You create an entry by passing a title, data and optionally options. The title can be used in the UI to indicate to the user which entry it is. Data and options can be any data type, as long as they are JSON serializable. Options are used for mutable data, for example a radius. Whilst Data is used for immutable data that isn't going to change in an entry, for example location data.

```
class ExampleConfigFlow(data_entry_flow.FlowHandler):
    async def async_step_user(self, user_input=None):
        return self.async_create_entry(
            title="Title of the entry",
```

```
            data={
                "username": user_input["username"],
                "password": user_input["password"]
            },
            options={
                "mobile_number": user_input["mobile_number"]
            },
        )
```

Note: A user can change their password, which technically makes it mutable data, but for changing authentication credentials, you use reauthentication, which can mutate the config entry data.

Abort

When a flow cannot be finished, you need to abort it. This will finish the flow and inform the user that the flow has finished. Reasons for a flow to not be able to finish can be that a device is already configured or not compatible with Home Assistant.

```
class ExampleConfigFlow(data_entry_flow.FlowHandler):
    async def async_step_user(self, user_input=None):
        return self.async_abort(reason="not_supported")
```

External step & external step done

It is possible that a user needs to finish a config flow by doing actions on an external website. For example, setting up an integration by being redirected to an external webpage. This is commonly used by integrations that use OAuth2 to authorize a user.

The example is about config entries, but works with other parts that use data entry flows too.

The flow works as follows:

1. The user starts config flow in Home Assistant.
2. Config flow prompts the user to finish the flow on an external website.
3. The user opens the external website.
4. Upon completion of the external step, the user's browser will be redirected to a Home Assistant endpoint to deliver the response.
5. The endpoint validates the response, and upon validation, marks the external step as done and returns JavaScript code to close the window: `<script>window.close()</script>`.

   To be able to route the result of the external step to the Home Assistant endpoint, you will need to make sure the config flow ID is included. If your external step is an OAuth2 flow, you can leverage the oauth2 state for this. This is a variable that is not interpreted by the authorization page but is passed as-is to the Home Assistant endpoint.
6. The window closes and the Home Assistant user interface with the config flow will be visible to the user again.
7. The config flow has automatically advanced to the next step when the external step was marked as done. The user is prompted with the next step.

Example configuration flow that includes an external step.

```
from homeassistant import config_entries

@config_entries.HANDLERS.register(DOMAIN)
class ExampleConfigFlow(data_entry_flow.FlowHandler):
    VERSION = 1
    data = None
```

```
    async def async_step_user(self, user_input=None):
        if not user_input:
            return self.async_external_step(
                step_id="user",
                url=f"https://example.com/?config_flow_id={self.flow_id}",
            )

        self.data = user_input
        return self.async_external_step_done(next_step_id="finish")

    async def async_step_finish(self, user_input=None):
        return self.async_create_entry(title=self.data["title"], data=self.data)
```

Avoid doing work based on the external step data before you return an `async_mark_external_step_done`. Instead, do the work in the step that you refer to as `next_step_id` when marking the external step done. This will give the user a better user experience by showing a spinner in the UI while the work is done.

If you do the work inside the authorize callback, the user will stare at a blank screen until that all of a sudden closes because the data has forwarded. If you do the work before marking the external step as done, the user will still see the form with the "Open external website" button while the background work is being done. That too is undesirable.

Example code to mark an external step as done:

```
from homeassistant import data_entry_flow


async def handle_result(hass, flow_id, data):
    result = await hass.config_entries.async_configure(flow_id, data)

    if result["type"] == data_entry_flow.FlowResultType.EXTERNAL_STEP_DONE:
        return "success!"
```

```
    else:
        return "Invalid config flow specified"
```

Show progress & show progress done

If a data entry flow step needs a considerable amount of time to finish, we should inform the user about this.

The example is about config entries, but works with other parts that use data entry flows too.

The flow works as follows:

1. The user starts the config flow in Home Assistant.
2. The config flow creates an `asyncio.Task` to execute the long running task.
3. The config flow informs the user that a task is in progress and will take some time to finish by calling `async_show_progress`, passing the `asyncio.Task` object to it. The flow should pass a task specific string as `progress_action` parameter to represent the translated text string for the prompt.
4. The config flow will be automatically called once the task is finished, but may also be called before the task has finished, for example if frontend reloads.

- If the task is not yet finished, the flow should not create another task, but instead call `async_show_progress` again.
- If the task is finished, the flow must call the `async_show_progress_done`, indicating the next step

5. The frontend will update each time we call show progress or show progress done.
6. The config flow will automatically advance to the next step when the progress was marked as done. The user is prompted with the next step.
7. The task can optionally call `async_update_progress(progress)` where progress is a float between 0 and 1, indicating how much of the task is done.

Example configuration flow that includes two show sequential progress tasks.

```
import asyncio

from homeassistant import config_entries
from .const import DOMAIN

class TestFlow(config_entries.ConfigFlow, domain=DOMAIN):
    VERSION = 1
    task_one: asyncio.Task | None = None
    task_two: asyncio.Task | None = None

    async def async_step_user(self, user_input=None):
        uncompleted_task: asyncio.Task[None] | None = None

        if not self.task_one:
            coro = asyncio.sleep(10)
            self.task_one = self.hass.async_create_task(coro)
        if not self.task_one.done():
            progress_action = "task_one"
            uncompleted_task = self.task_one
        if not uncompleted_task:
            if not self.task_two:
                self.async_update_progress(0.5) # tell frontend we are 50% done
                coro = asyncio.sleep(10)
                self.task_two = self.hass.async_create_task(coro)
            if not self.task_two.done():
                progress_action = "task_two"
                uncompleted_task = self.task_two
        if uncompleted_task:
            return self.async_show_progress(
                progress_action=progress_action,
```

```
                progress_task=uncompleted_task,
            )

        return self.async_show_progress_done(next_step_id="finish")

    async def async_step_finish(self, user_input=None):
        if not user_input:
            return self.async_show_form(step_id="finish")
        return self.async_create_entry(title="Some title", data={})
```

Show menu

This will show a navigation menu to the user to easily pick the next step. The menu labels can be hardcoded by specifying a dictionary of `{step_id: label}` or translated via `strings.json` when specifying a list.

```
class ExampleConfigFlow(data_entry_flow.FlowHandler):
    async def async_step_user(self, user_input=None):
        return self.async_show_menu(
            step_id="user",
            menu_options=["discovery", "manual"],
            description_placeholders={
                "model": "Example model",
            }
        )
        # Example showing the other approach
        return self.async_show_menu(
            step_id="user",
            menu_options={
                "option_1": "Option 1",
                "option_2": "Option 2",
```

```
            }
        )
```

```
{
  "config": {
    "step": {
      "user": {
        "menu_options": {
          "discovery": "Discovery",
          "manual": "Manual ({model})",
        }
      }
    }
  }
}
```

Initializing a config flow from an external source

You might want to initialize a config flow programmatically. For example, if we discover a device on the network that requires user interaction to finish setup. To do so, pass a source parameter and optional user input when initializing a flow:

```
await flow_mgr.async_init(
    "hue", context={"source": data_entry_flow.SOURCE_DISCOVERY}, data=discovery_info
)
```

The config flow handler will not start with the `init` step. Instead, it will be instantiated with a step name equal to the source. The step should follow the same return values as a normal step.

```
class ExampleConfigFlow(data_entry_flow.FlowHandler):
    async def async_step_discovery(self, info):
        """Handle discovery info."""
```

The source of a config flow is available as `self.source` on `FlowHandler`.

Edit this page

*Last updated on **Jun 27, 2025***