Drupal 9 configuration schema cheat sheet

2.0 - December 22. 2021.

Configuration schema is a YAML based declarative format in Drupal 9 to describe the structure of configuration files. It is then applied to:

- Typecast configuration values to ensure type consistency (see StorableConfigBase::castValue())
- Automated persistence of configuration entity properties (see ConfigEntityBase::toArray())
- Extraction of translatable configuration strings for localize.drupal.org and Interface Translation
- Automated generation of the configuration translation user interface (see the core module)
- Ordering config keys automatically for consistent configuration structure (from Drupal 9.3.0)

Module settings example

config/install/my_module.settings.yml

```
type: warning
message: 'Hello!'
langcode: en
```

config/schema/my_module.schema.yml

```
my_module.settings:
  type: config_object
  mapping:
    type:
      type: string
      label: 'Message type'
    message:
      type: label
      label: 'Message text'
```

Settings in config

The langcode key is inherited from config_object

Schema keys and base types

Define configuration object: schema key should be the filename of the configuration object. Typically modulename.settings. Inherit from config_object *(see earlier example).*

1

Define configuration entity: schema key should be the module name followed by the config entity type ID (from the PHP annotation), followed by an asterisk (for matching the name of the config entity). Inherit from config_entity.

2

Examples of text format config entities:
filter/config/install/filter.format.plain_text.yml
standard/config/install/filter.format.basic_html.yml

Are all matched by this config schema in filter/config/schema/filter.schema.yml:

```
filter.format.*:
  type: config_entity
  label: 'Text formats'
  mapping:
    name:
      type: label
      label: 'Name'
  (....)
```

Lots of built in properties inherited from config_entity.

Define dynamic part of another config structure: the key used and potential base type used would be as dictated by the extension point defined where your data structure slots into. Views handlers, display plugins, etc. would be in this group. Examples later.

3

Filename and file location

Put schema files in your module or theme's config/schema directory. A simple extension would define a modulename.schema.yml file, but any number of schema YAML files are possible to logically group definitions (see Views module for an extensive example).

Schema is always subtyping

All of config schema is about subtyping from existing types. The my_module.settings example earlier is subtyping config_object (which is a subtype of the mapping type). All keys from the subtype and sub-subtype add up to the final type. Core base types are defined in core.data_types.schema.yml.

Base scalar types		boolean
string	integer	timestamp
uri	float	email

Common string subtypes

There are some purpose specific subtypes of the simple string type, particularly for UI generation and translatability purposes, such as:

- label: short and translatable string
- plural_label: a label that contains plural variants
- text: long and translatable string
- uuid: a string that is a UUID
- path: a string that is a Drupal path
- date_format: a string that is a PHP date format
- color_hex: a string that is a hex color value

Translatability

The label, plural_label, date_format and text types are defined as translatable: true.

For drupal.org hosted projects, translatable values are exported to localize.drupal.org for community translation. When an extension is installed, translatable strings from default configuration are saved into the interface translation database and may be replaced in active configuration with translated values as available.

To define your own translatable value that is not logically of any other already defined type, add translatable: true. Only use this on single string value types. If possible, use the base translatable types as they provide specific translation UI elements. Add translation context as needed.

config/schema/my_module.schema.yml

```
type: warning
message: 'Hello!'
langcode: en
```

config/schema/my_module.schema.yml

```
my_module.settings:
  type: config_object
  mapping:
    type:
      type: string
      label: 'Message type'
    message:
      type: label
      translation context: 'Message to print'
      label: 'Message text'
```

Already translatable

Context is useful for short strings

Mappings and sequences

There are two base list types defined by configuration schema. If all keys can be predefined in the schema structure, then you should use a mapping, otherwise you should use a sequence (even if the list's keys are strings).

config/install/my_module.settings.yml

```
time: 1640116813
messages:
  - 'Hello!'
  - 'Hi!'
langcode: en
```

config/schema/my_module.schema.yml

```
my_module.settings:
  type: config_object
  mapping:
    time:
      type: timestamp
      label: 'When to print the messages'
    messages:
      type: sequence
      label: 'Message list'
      sequence:
        type: label
        label: 'Message text'
```

Extending an already mapping based type

A list of any number of things

Each item is a translatable short string

In the above example, the settings structure has a known list of keys, including the time key that defines a timestamp and a messages key that defines a list of strings. The later list can have any number of items though, so it is a sequence. Sequences don't need to be numeric, they could have string keys.

Sequence with string keys

The previous schema matches this configuration too, because sequence does not specify anything about the keys or the number of items. String keyed sequences can be one way to do dynamic typing though, see later.

config/install/my_module.settings.yml

```
time: 1640116813
messages:
  long: 'Hello!'
  short: 'Hi!'
  extra: 'Welcome friend!'
langcode: en
```

Common mapping subtypes

Core defines some common mapping structures as reusable patterns for your configuration.

- theme_settings: base type for a theme settings file
- mail: a mapping with subject and body keys, use to store an email subject and body combination
- text_format: a mapping with text and format keys, use for text to be formatted with a text format
- route: a route_name string and route_params sequence

Ignore and undefined

Finally, these two special types are to be avoided at all cost. The undefined type is used internally to represent unspecified schema, so it's pointless to use. You may use ignore if a sub-structure of the config is absolutely impossible to define. In this case that part will not benefit from any of the advantages of config with schema.

Three ways of dynamic typing

Parts of the configuration structure may depend on data elsewhere in configuration. There are three ways to define dynamic types based on data:

1. Use `[%parent]` to define type based on a parent property value.
2. Use `[childkey]` to define type based on an embedded value.
3. Use `[%key]` to define type based on the key of a list item (especially useful for sequences).

Take this data structure where the type of the `value` key depends on the contents of the `variant` key. It is impossible to define schema for this without considering the data.

config/install/my_module.settings.yml

```
message:
  variant: single
  value: 'Hello!'
langcode: en
```

Two potential versions of the same file

config/install/my_module.settings.yml

```
message:
  variant: multiple
  value:
    - 'Hello!'
    - 'Hi!'
langcode: en
```

There are two ways to approach this, either define the `value` key based on it's sibling (the parent's) `variant` key or the top `message` key based on the underlying `variant` value. Use what is more appropriate based on other data elements.

Dynamic type with `[%parent]`

Defining the type of a key based on a parent's child.

config/schema/my_module.schema.yml

```
my_module.settings:
  type: config_object
  mapping:
    message:
      type: mapping
      mapping:
        variant:
          type: string
          label: 'Message variant'
        value:
          type: my_module_value.[%parent.variant]

my_module_value.single:
  type: label
  label: 'Message text'

my_module_value.multiple:
  type: sequence
  label: 'Message list'
  sequence:
    type: label
    label: 'Message text'
```

Use the `variant` key from parent data

Our own types prefixed with module name to avoid conflict with top level types

Here, the type for the `message` key is a custom type that is dynamic based on the value of the parent's (from the point of view of `message`) `variant` key. This opens the door for other variants down the line that have their own structure without the need to change the main wrapping structure.

Chaining is possible as `%parent.%parent....` Combine with `%type` to get the parent's type: `%parent.%type`.

Dynamic type with `[childkey]`

Defining the type of the wrapper based on data within.

config/schema/my_module.schema.yml

```
my_module.settings:
  type: config_object
  mapping:
    message:
      type: my_module_message.[variant]

my_module_message_base:
  type: mapping
  mapping:
    variant:
      type: string
      label: 'Message variant'

my_module_message.single:
  type: mymodule_message_base
  mapping:
    value:
      type: label
      label: 'Message text'

my_module_message.multiple:
  type: mymodule_message_base
  mapping:
    value:
      type: sequence
      label: 'Message list'
      sequence:
        type: label
        label: 'Message text'
```

Use the `variant` key from the data structure under it

Custom base type to define shared mapping format and variant key

Only need to define the unique keys of the mapping as the rest are inherited from the base type

This looks more complicated at first but there are cases where this approach makes most sense to use.

Dynamic type with [%key]

This is useful when a list of items is typed based on (part of) the list item key.

config/install/my_module.settings.yml

```
messages:
  'single:long': 'Hello!'
  'single:short': 'Hi!'
  'multiple:mix':
    - 'Good morning!'
    - 'Good night!'
langcode: en
```

config/schema/my_module.schema.yml

```
my_module.settings:
  type: config_object
  mapping:
    messages:
      type: sequence
      label: 'List of messages'
      sequence:
        type:  my_module_message.[%key]

my_module_message.single:*:
  type: label
  label: 'Message text'

my_module_message.multiple:*:
  type: sequence
  label: 'Message list'
  sequence:
    type: label
    label: 'Message text'
```

Type is the prefix of the key, e.g. 'single:1'

Use wildcard match to match to the prefix regardless of further content of the key

Built-in extension points

There are various built-in schema extension points in core fore common situations. Some examples:

- Theme settings have a `third_party_settings` key which is a sequence of type `theme_settings.third_party.[%key]`, allowing to add arbitrary third party settings to theme settings.
- Configuration entities have a `third_party_settings` key which is a sequence of type `[%parent.%parent.%type].third_party.[%key]`. So that is based on the config entity type name, such as `contact.form.*.third_party.contact_storage` would be settings for contact storage on all contact forms.
- Views has probably the most extensive list, eg. `views.display.[%parent.display_plugin]` for displays, `views.style.[%parent.type]` for styles, `views.relationship.[plugin_id]` for relationships, etc.

Advanced properties

Config schema elements can take some advanced properties that are rarely used.

- Set `nullable: true` to explicitly define that a key is optional and may not be present or have value. Practically used for mappings and sequences to accept a `NULL` value in their place, see `SchemaCheckTrait::checkValue()`.
- Set `deprecated: 'Deprecation message text...'` to specify a key as deprecated.
- Very rarely set the `class` and `definition_class` keys to assign PHP classes implementing the parsing and definition of the value.

Schema debugging

To debug configuration schemas, use the Configuration Inspector module (http://drupal.org/project/config_inspector) which helps you find schema mismatches with active configuration and inspect how your schema is applied to your configuration.

Schema testing

- All BrowserTestBase and KernelTestBase extending tests define `$strictConfigSchema = TRUE` by default, which results in strict scheme adherence testing for all configuration saved. Only opt out of this if you *really* need to. Your schema should match your data and pass this test.
- Use SchemaCheckTestTrait in your test to explicitly validate with specific config schemas.

More documentation

See https://www.drupal.org/node/1905070 for even more configuration schema documentation and examples.

Issues?

- For issues with core configuration schemas, tag them with 'Configuration schema' and 'Configuration system' and pick the appropriate module as component.
- For issues with the configuration schema system itself, use the 'configuration system' component and also tag with 'Configuration schema'.

Cheat sheet created by Gábor Hojtsy
https://www.drupal.org/user/4166/contact