Drupal 8 configuration schema cheat sheet

1.5 - April 9. 2015.

Configuration schema in Drupal 8 is used to describe the structure of configuration files. It is then applied to:

- Typecast configuration to ensure type consistency (see StorableConfigBase::castValue())
- Automated persistence of configuration entity properties (see ConfigEntityBase::toArray())
- Automated generation of the configuration translation user interface (see the core module)

A simple 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:                    <- Settings in config
      type: string
      label: 'Message type'
    message:
      type: label
      label: 'Message text'
    langcode:                <- Used internally for translation
      type: string
      label: 'Language code'
```

Basic schema types

Core provides the following data types. Contributed modules may define new base types. More are defined in *core.data_types.schema.yml*.

Scalar types
boolean
integer
float
string
uri
email
List types
mapping: known keys
sequence: unknown keys
Common subtypes
label: short & translatable
text: long & translatable
config_object: object root
config_entity: entity root

Subtyping

All of config schema is about subtyping from existing types. The simple example earlier is subtyping *config_object* (which is a subtype of *mapping*) further defining keys with their own types.

The *config_object* or config_entity types are for the root of config schema definitions as appropriate.

Types *route*, *filter*, *mail*, etc. are provided for common complex internal data structures.

Dynamic type with [%parent]

Exact types may not be known ahead of time and may depend on the data. Schema allows to define types based on the data as well. Let's say the type of *message* may depend on the *type* value: either a list of messages or a simple warning message. Let's use *'multiple'* for the list case and keep *'warning'* for the single line message.

config/install/my_module.message.single.yml

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

config/install/my_module.message.multiple.yml

```
type: multiple
message:
  - 'Hello!'
  - 'Hi!'
langcode: en
```

config/schema/my_module.schema.yml

```
my_module.message.*:          <- Used wildcard so it applies to a set of config names.
  type: config_object
  mapping:
    type:
      type: string
      label: 'Message type'
    message:
      type: my_module_message.[%parent.type]
    langcode:                 <- Dynamic element type based on data
      type: string
      label: 'Language code'

my_module_message.warning:
  type: string
  label: 'Message'

my_module_message.multiple:
  type: sequence              <- Internal types prefixed with module name to avoid conflict with top level types.
  label: 'Messages'
  sequence:
    type: string
    label: 'Message'
```

Chaining is possible as *%parent.%parent.type*, etc.

Dynamic type with [type]

If the data to vary your type by is under the data to be typed, that is when [type] becomes useful.

config/install/my_module.message.single.yml

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

config/install/my_module.message.multiple.yml

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

config/schema/my_module.schema.yml

```
my_module.message.*:
  type: config_object
  mapping:
    message:
      type: my_module_message.[type]
  […]

my_module_message.warning:
  type: mapping
  […]

my_module_message.multiple:
  type: mapping
  […]
```

Use the *type* key under the message

Need to define *type* and *value* keys as appropriate.

You may also define a **my_module_message_base** base type that includes common keys like 'type' and extend from that with any custom keys per type.

Dynamic type with [%key]

config/install/my_module.messages.yml

```
messages:
  'single:1': 'Hello!'
  'single:2': 'Hi!'
  'multiple:1':
    - 'Good morning!'
    - 'Good night!'
langcode: en
```

Arbitrary message list

This is now a list of arbitrary message elements.

config/schema/my_module.schema.yml

```
my_module.messages:
  type: config_object
  mapping:
    messages:
      type: sequence
      label: 'Messages'
      sequence:
        type: my_module_message.[%key]
    langcode:
      type: string
      label: 'Language code'

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

my_module_message.multiple:*:
  type: sequence
  label: 'Messages'
  sequence:
    type: string
    label: 'Message'
```

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

Wildcard to match prefix.

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 TestBase deriving tests in core now use *$strictConfigSchema = TRUE* 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 check for specific config files only.

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'.

Created by Gábor Hojtsy
https://www.drupal.org/user/4166/contact