Drupal 8 configuration schema cheat sheet

1.0 - Dec 12. 2014.

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

- Typecast configuration to ensure type consistency (to only get useful diffs on deployment)
- Automated persistence of configuration entity properties (on the top level)
- Automated generation of the configuration translation user interface

A simple example

config/install/my_module.settings.yml

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

config/schema/my_module.schema.yml

```
my_module.settings:          ← Config key /name
  type: mapping
  mapping:
    type:                    ← Settings in config
      type: string
      label: 'Message type'
    message:
      type: label
      label: 'Message text'
    langcode:                ← Used internally
      type: string             for translation
      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

Subtyping

All of configuration schema is basically subtyping from existing types. The simple example earlier is subtyping *mapping* with defined keys that have their own types.

Label and *text* are the most important subtypes for translatability. Types *route*, *filter*, *mail*, etc. are provided for common complex Drupal data structures.

Dynamic type with [type]

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 with 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.*:      ← Applies to a set of
  type: mapping              config keys
  mapping:
    type:
      type: string
      label: 'Message type'
    message:
      type: my_module_message.[type]   ←
      label: 'Message'
    langcode:                  Dynamic element
      type: string             type based on data
      label: 'Language code'

my_module_message.warning:
  type: string

my_module_message.multiple:
  type: sequence
  sequence:
    - type: string
      label: 'Individual message'
```

Fields, entity displays, views, blocks, etc. use this extensively to define pluggable types.

Dynamic type with [%parent]

All dynamic references are enclosed in [ ], like with [key] above. If the data is not on the same level, you can reference the parent as well with %parent. Restructuring the previous example:

config/install/my_module.message.single.yml

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

config/install/my_module.message.multiple.yml

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

Now the type indication is one level up:

config/schema/my_module.schema.yml

```
my_module_message.*:
  type: mapping
  mapping:
    message:                          Type one level up
      type: mapping                           ↓
      mapping:
        data:
          type:  my_module_message.[%parent.type]
          label: 'Message'

… rest is same as above …
```

Dynamic type with [%key]

config/install/my_module.messages.yml

```
messages:
  'single:1': 'Hello!'
  'single:2': 'Hi!'   ←—— Arbitrary message list
  'multiple:1':
    - 'Good morning!'
    - 'Good night!'
langcode: en
```

This is now a list of arbitrary message element.

config/schema/my_module.schema.yml

```
my_module_messages:
  type: mapping
  mapping:
    message:
      type: sequence          Type is in the key
      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'           Wildcard to match
  sequence:                   any item of this
    - type: string            type.
      label: 'Message'
```

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 is you *really* need to. Your schema should match your data and pass this test.

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.

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