Exposing SiteAccess-aware configuration for your bundle

In this topic:

Description

Symfony Config component makes it possible to define *semantic configuration*, exposed to the end developer. This configuration is validated by rules you define, e.g. validating type (string, array, integer, boolean, etc.). Usually, once validated and processed, this semantic configuration is then mapped to internal *key/value* parameters stored in the `ServiceContainer`.

eZ Platform uses this for its core configuration, but adds another configuration level, the **siteaccess**. For each defined siteaccess, we need to be able to use the same configuration tree in order to define siteaccess-specific config. These settings then need to be mapped to siteaccess-aware internal parameters that you can retrieve via the `ConfigResolver`. For this, internal keys need to follow the format `<namespace>.<scope>.<parameter_name>`, `namespace` being specific to your app/bundle, `scope` being the siteaccess, siteaccess group, `default` or `global`, `parameter_name` being the actual setting *identifier*.

For more information on ConfigResolver, namespaces and scopes, see eZ Platform configuration basics.

The goal of this feature is to make it easy to implement a *siteaccess-aware* semantic configuration and its mapping to internal config for any eZ bundle developer.

Solution

Semantic configuration parsing

An abstract `Configuration` class has been added, simplifying the way to add a siteaccess settings tree like the following:

ezplatform.yml or config.yml

```
acme_demo:
    system:
        my_siteaccess:
            hello: "world"
            foo_setting:
                an_integer: 456
                enabled: true

        my_siteaccess_group:
            hello: "universe"
            foo_setting:
                foo: "bar"
                some: "thing"
                an_integer: 123
                enabled: false
```

Class FQN is `eZ\Bundle\EzPublishCoreBundle\DependencyInjection\Configuration\SiteAccessAware\Configuration`.
All you have to do is to extend it and use `$this->generateScopeBaseNode()`:

```
namespace Acme\DemoBundle\DependencyInjection;

use
eZ\Bundle\EzPublishCoreBundle\DependencyInjection\Config
uration\SiteAccessAware\Configuration as
SiteAccessConfiguration;
use
Symfony\Component\Config\Definition\Builder\NodeBuilder;
use
Symfony\Component\Config\Definition\Builder\TreeBuilder;

class Configuration extends SiteAccessConfiguration
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root( 'acme_demo' );

        // $systemNode will then be the root of
siteaccess aware settings.
        $systemNode = $this->generateScopeBaseNode(
$rootNode );
        $systemNode
            ->scalarNode( 'hello' )->isRequired()->end()
            ->arrayNode( 'foo_setting' )
                ->children()
                    ->scalarNode( "foo" )->end()
                    ->scalarNode( "some" )->end()
                    ->integerNode( "an_integer" )->end()
                    ->booleanNode( "enabled" )->end()
                ->end()
            ->end();

        return $treeBuilder;
    }
}
```

Default name for the *siteaccess root node* is `system`, but you can customize it. For this, just pass the name you want to use as a second argument of `$this->generateScopeBaseNode()`.

Mapping to internal settings

Semantic configuration must always be *mapped* to internal *key/value* settings within the `ServiceContainer`. This is usually done in the DIC extension.

For siteaccess-aware settings, new `ConfigurationProcessor` and `Contextualizer` classes have been introduced to ease the process.

```
namespace Acme\DemoBundle\DependencyInjection;

use
eZ\Bundle\EzPublishCoreBundle\DependencyInjection\Config
uration\SiteAccessAware\ConfigurationProcessor;
```

```
use
eZ\Bundle\EzPublishCoreBundle\DependencyInjection\Config
uration\SiteAccessAware\ContextualizerInterface;
use
Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use
Symfony\Component\HttpKernel\DependencyInjection\Extensi
on;
use Symfony\Component\DependencyInjection\Loader;

/**
 * This is the class that loads and manages your bundle
configuration
 *
 * To learn more see {@link
http://symfony.com/doc/current/cookbook/bundles/extensio
n.html}
 */
class AcmeDemoExtension extends Extension
{
    public function load( array $configs,
ContainerBuilder $container )
    {
        $configuration = $this->getConfiguration(
$configs, $container );
        $config = $this->processConfiguration(
$configuration, $configs );

        $loader = new Loader\YamlFileLoader( $container,
new FileLocator( __DIR__.'/../Resources/config' ) );
        $loader->load( 'default_settings.yml' );

        // "acme_demo" will be the namespace as used in
ConfigResolver format.
        $processor = new ConfigurationProcessor(
$container, 'acme_demo' );
        $processor->mapConfig(
            $config,
            // Any kind of callable can be used here.
            // It will be called for each declared
scope/SiteAccess.
            function ( $scopeSettings, $currentScope,
ContextualizerInterface $contextualizer )
            {
                // Will map "hello" setting to
"acme_demo.<$currentScope>.hello" container parameter
                // It will then be possible to retrieve
this parameter through ConfigResolver in the application
code:
                // $helloSetting =
$configResolver->getParameter( 'hello', 'acme_demo' );
                $contextualizer->setContextualParameter(
'hello', $currentScope, $scopeSettings['hello'] );
            }
        );

        // Now map "foo_setting" and ensure keys defined
for "my_siteaccess" overrides the one for
```

```
"my_siteaccess_group"
        // It is done outside the closure as it is
needed only once.
        $processor->mapConfigArray( 'foo_setting',
$config );
    }
}
```

Tip
You can map simple settings by calling `$processor->mapSetting()`, without having to call `$processor->mapConfig()` with a callable.

```
$processor = new ConfigurationProcessor(
$container, 'acme_demo' );
$processor->mapSetting( 'hello', $config );
```

Important
Always ensure you have defined and loaded default settings.

@AcmeDemoBundle/Resources/config/default_settings.yml

```
parameters:
    acme_demo.default.hello: world
    acme_demo.default.foo_setting:
        foo: ~
        some: ~
        planets: [Earth]
        an_integer: 0
        enabled: false
        j_adore: les_sushis
```

Merging hash values between scopes

When you define a hash as semantic config, you sometimes don't want the siteaccess settings to replace the default or group values, but *enrich* them by appending new entries. This is made possible by using `$processor->mapConfigArray()`, which needs to be called outside the closure (before or after), in order to be called only once.

Consider the following default config:

default_settings.yml

```
parameters:
    acme_demo.default.foo_setting:
        foo: ~
        some: ~
        planets: [Earth]
        an_integer: 0
        enabled: false
        j_adore: les_sushis
```

And then this semantic config:

ezplatform.yml or config.yml

```
acme_demo:
    system:
        sa_group:
            foo_setting:
                foo: bar
                some: thing
                an_integer: 123

        # Assuming "sa1" is part of "sa_group"
        sa1:
            foo_setting:
                an_integer: 456
                enabled: true
                j_adore: le_saucisson
```

What we want here is that keys defined for `foo_setting` are merged between default/group/siteaccess:

Expected result

```
parameters:
    acme_demo.sa1.foo_setting:
        foo: bar
        some: thing
        planets: [Earth]
        an_integer: 456
        enabled: true
        j_adore: le_saucisson
```

Merge from *second level*

In the example above, entries were merged in respect to the scope order of precedence. However, if we define the `planets` key for`sa1`, it will completely override the default value since the merge process is done at only 1 level.

You can add another level by passing `ContextualizerInterface::MERGE_FROM_SECOND_LEVEL` as an option (3rd argument) to`$contextualizer->mapConfigArray()`.

default_settings.yml

```
parameters:
    acme_demo.default.foo_setting:
        foo: ~
        some: ~
        planets: [Earth]
        an_integer: 0
        enabled: false
        j_adore: [les_sushis]
```

Semantic config (ezplatform.yml / config.yml)

```
acme_demo:
    system:
        sa_group:
            foo_setting:
                foo: bar
                some: thing
                planets: [Mars, Venus]
                an_integer: 123

        # Assuming "sa1" is part of "sa_group"
        sa1:
            foo_setting:
                an_integer: 456
                enabled: true
                j_adore: [le_saucisson,
la_truite_a_la_vapeur]
```

Result of using `ContextualizerInterface::MERGE_FROM_SECOND_LEVEL` option:

```
parameters:
    acme_demo.sa1.foo_setting:
        foo: bar
        some: thing
        planets: [Earth, Mars, Venus]
        an_integer: 456
        enabled: true
        j_adore: [les_suhis, le_saucisson,
la_truite_a_la_vapeur]
```

There is also another option, `ContextualizerInterface::UNIQUE`, to be used when you want to ensure your array setting has unique values. It will only work on normal arrays though, not hashes.

Limitations

A few limitation exist with this scope hash merge:

- Semantic setting name and internal name will be the same (like `foo_setting` in the examples above).

- Applicable to first level semantic parameter only (i.e. settings right under the siteaccess name).
- Merge is not recursive. Only second level merge is possible by using `ContextualizerInterface::MERGE_FROM_SECOND_LEVEL` option.

Dedicated mapper object

Instead of passing a callable to `$processor->mapConfig()`, an instance of `eZ\Bundle\EzPublishCoreBundle\DependencyInjection\Configuration\SiteAccessAware\ConfigurationMapperInterface` can be passed.

This can be useful if you have a lot of configuration to map and don't want to pollute your DIC extension class (better for maintenance).

Merging hash values between scopes

As specified above, `$contextualizer->mapConfigArray()` is not to be used within the *scope loop*, like for simple values. When using a closure/callable, you usually call it before or after `$processor->mapConfig()`. For mapper objects, a dedicated interface can be used: `HookableConfigurationMapperInterface`, which defines 2 methods: `preMap()` and `postMap()`.