SiteAccess

Introduction

eZ Platform allows you to maintain multiple sites in one installation using a feature called **siteaccesses**.

In short, a siteaccess is a set of configuration settings that is used when the site is reached through a specific address.

When the user accesses the site, the system analyzes the uri and compares it to rules specified in the configuration. If it finds a set of fitting rules, this siteaccess is used.

What does a siteaccess do?

A siteaccess overrides the default configuration. This means that if the siteaccess does not specify some aspect of the configuration, the default values will be used. The default configuration is also used when no siteaccess can be matched to a situation.

A siteaccess can decide many things about the website, for example the database, language or var directory that are used.

How is a siteaccess selected?

A siteaccess is selected using one or more matchers – rules based on the uri or its parts. Example matching criteria are elements of the uri, host name (or its parts), port number, etc.

For detailed information on how siteaccess matchers work, see Siteaccess Matching.

What can you use siteaccesses for?

Typical uses of a siteaccess are:

- different language versions of the same site identified by a uri part; one siteaccess for one language
- two different versions of a website: one siteaccess with a public interface for visitors and one with a restricted interface for administrators

Both the rules for siteaccess matching and its effects are located in the main **app/config/ezplatform.yml** configuration file.

Use case: multilanguage sites

A site has content in two languages: English and Norwegian. It has one URI per language: http://example.com/eng and http://example.com/nor. Uri parts of each language (eng, nor) are mapped to a *siteaccess*, commonly named like the uri part: `eng`, `nor`. Using semantic configuration, each of these siteaccesses can be assigned a prioritized list of languages it should display:

- The English site would display content in English and ignore Norwegian content;
- The Norwegian site would display content in Norwegian but also in English *if it does not exist in Norwegian*.

Such configuration would look like this:

In this topic:

Related topics:

Cross-siteaccess links

Setting the Index Page

```
ezpublish:
 siteaccess:
  # There are two siteaccess
  list: [eng, nor]

  # eng is the default one if no prefix is specified
        default_siteaccess: eng

  # the first URI of the element is used to find a
siteaccess with a similar name
        match:
      URIElement: 1


ezpublish:
 # root node for configuration per siteaccess
 system:
  # Configuration for the 'eng' siteaccess
        eng:
   languages: [eng-GB]
        nor:
            languages: [nor-NO, eng-GB]
```

The default scope

When no particular context is required, it is fine to use the `default` scope instead of specifying a siteaccess.

Configuration

Basics

> **Important**
> Configuration is tightly related to the service container.
> To fully understand the following content, you need to be familiar with Symfony's service container and its configuration.

Basic configuration handling in eZ Platform is similar to what is commonly possible with Symfony. Regarding this, you can define key/value pairs in your configuration files, under the main **parameters** key (like in **parameters.yml**).

Internally and by convention, keys follow a **dot syntax** where the different segments follow your configuration hierarchy. Keys are usually prefixed by a *namespace* corresponding to your application. Values can be anything, **including arrays and deep hashes**.

> eZ Platform core configuration is prefixed by **ezsettings** namespace, while *internal* configuration (not to be used directly) is prefixed by **ezpublish** namespace.

> For configuration that is meant to be exposed to an end-user (or end-developer), it's usually a good idea to also implement semantic configuration.
>
> Note that it is also possible to implement SiteAccess aware semantic configuration.

Example

Configuration

```
parameters:
    myapp.parameter.name: someValue
    myapp.boolean.param: true
    myapp.some.hash:
        foo: bar
        an_array: [apple, banana, pear]
```

Usage from a controller

```
// Inside a controller
$myParameter = $this->container->getParameter(
'myapp.parameter.name' );
```

Dynamic configuration with the ConfigResolver

In eZ Platform it is fairly common to have different settings depending on the current siteaccess (e.g. languages, view provider configuration).

Scope

Dynamic configuration can be resolved depending on a *scope*.

Available scopes are (in order of precedence) :

1. *global*
2. SiteAccess
3. SiteAccess group
4. *default*

It gives the opportunity to define settings for a given siteaccess, for instance, like in the legacy INI override system.

This mechanism is not limited to eZ Platform internal settings (aka ***ezsettings* namespace**) and is applicable for specific needs (bundle-related, project-related, etc.).

> Always prefer semantic configuration especially for internal eZ settings.
> Manually editing internal eZ settings is possible, but at your own risk, as unexpected behavior can occur.

ConfigResolver Usage

Dynamic configuration is handled by a **config resolver**. It consists in a service object mainly exposing `hasParameter()` and `getParameter()` methods. The idea is to check the different *sc opes* available for a given *namespace* to find the appropriate parameter.

In order to work with the config resolver, your dynamic settings must comply internally with the following name format: `<namespace>.<scope>.parameter.name`.

> The following configuration is **an example of internal usage** inside the code of eZ Platform.

Namespace + scope example

```
parameters:
    # Some internal configuration
    ezsettings.default.content.default_ttl: 60
    ezsettings.ezdemo_site.content.default_ttl: 3600

    # Here "myapp" is the namespace, followed by the
siteaccess name as the parameter scope
    # Parameter "foo" will have a different value in
ezdemo_site and ezdemo_site_admin
    myapp.ezdemo_site.foo: bar
    myapp.ezdemo_site_admin.foo: another value
    # Defining a default value, for other siteaccesses
    myapp.default.foo: Default value

    # Defining a global setting, used for all
siteaccesses
    #myapp.global.some.setting: This is a global value
```

```
// Inside a controller, assuming siteaccess being
"ezdemo_site"
/** @var $configResolver
\eZ\Publish\Core\MVC\ConfigResolverInterface **/
$configResolver = $this->getConfigResolver();

// ezsettings is the default namespace, so no need to
specify it
// The following will resolve
ezsettings.<siteaccessName>.content.default_ttl
// In the case of ezdemo_site, will return 3600.
// Otherwise it will return the value for
ezsettings.default.content.default_ttl (60)
$locationViewSetting = $configResolver->getParameter(
'content.default_ttl' );

$fooSetting = $configResolver->getParameter( 'foo',
'myapp' );
// $fooSetting's value will be 'bar'

// Force scope
$fooSettingAdmin = $configResolver->getParameter( 'foo',
'myapp', 'ezdemo_site_admin' );
// $fooSetting's value will be 'another value'

// Note that the same applies for hasParameter()
```

Both `getParameter()` and `hasParameter()` can take 3 different arguments:

1. `$paramName` (i.e. the name of the parameter you need)
2. `$namespace` (i.e. your application namespace, *myapp* in the previous example. If null, the default namespace will be used, which is **ezsettings** by default)
3. `$scope` (i.e. a siteaccess name. If null, the current siteaccess will be used)

Inject the ConfigResolver in your services

> Instead of injecting the whole ConfigResolver service, you may directly inject your SiteAccess-aware settings (aka dynamic settings) into your own services.

You can use the **ConfigResolver** in your own services whenever needed. To do this, just inject the **`ezpublish.config.resolver`** `service`:

```
parameters:
    my_service.class: My\Cool\Service

services:
    my_service:
        class: %my_service.class%
        arguments: [@ezpublish.config.resolver]
```

```
<?php
namespace My\Cool;

use eZ\Publish\Core\MVC\ConfigResolverInterface;

class Service
{
    /**
     * @var \eZ\Publish\Core\MVC\ConfigResolverInterface
     */
    private $configResolver;

    public function __construct( ConfigResolverInterface
$configResolver )
    {
        $this->configResolver = $configResolver;
        $myParam = $this->configResolver->getParameter(
'foo', 'myapp' );
    }

    // ...
}
```

Custom locale configuration

If you need to use a custom locale they can also be configurable in `ezplatform.yml`, adding them to the *conversion map*:

```
ezpublish:
    # Locale conversion map between eZ Publish format
(i.e. fre-FR) to POSIX (i.e. fr_FR).
    # The key is the eZ Publish locale. Check locale.yml
in EzPublishCoreBundle to see natively supported
locales.
    locale_conversion:
        eng-DE: en_DE
```

A locale *conversion map* example can be found in the `core` bundle, on `locale.yml`.

Siteaccess Matching

Siteaccess matching is done through **eZ\Publish\MVC\SiteAccess\Matcher** objects. You can configure this matching and even develop custom matchers.

To be usable, every siteaccess must be provided a matcher.

You can configure siteaccess matching in your main **app/config/ezplatform.yml**:

ezplatform.yml

```
ezpublish:
    siteaccess:
        default_siteaccess: ezdemo_site
        list:
            - ezdemo_site
            - eng
            - fre
            - fr_eng
            - ezdemo_site_admin
        groups:
            ezdemo_site_group:
                - ezdemo_site
                - eng
                - fre
                - fr_eng
                - ezdemo_site_admin
        match:
            Map\URI:
                ezdemo_site: ezdemo_site
                fre: fre
                ezdemo_site_admin: ezdemo_site_admin
```

You need to set several parameters:

- **ezpublish.siteaccess.default_siteaccess**
- **ezpublish.siteaccess.list**
- *(optional)* **ezpublish.siteaccess.groups**
- **ezpublish.siteaccess.match**

ezpublish.siteaccess.default_siteaccess is the default siteaccess that will be used if matching was not successful. This ensures that a siteaccess is always defined.

ezpublish.siteaccess.list is the list of all available siteaccesses in your website.

(optional) **ezpublish.siteaccess.groups** defines which groups siteaccesses belong to. This is useful when you want to mutualize settings between several siteaccesses and avoid config

duplication. Siteaccess groups are treated the same as regular siteaccesses as far as configuration is concerned.

> A siteaccess can be part of several groups.
>
> A siteaccess configuration has always precedence on the group configuration.

ezpublish.siteaccess.match holds the matching configuration. It consists in a hash where the key is the name of the matcher class. If the matcher class doesn't start with a **** , it will be considered relative to `eZ\Publish\MVC\SiteAccess\Matcher` (e.g. `Map\Host` will refer to `eZ\Publish\MVC\SiteAccess\Matcher\Map\Host`)

> Every **custom matcher** can be specified with a **fully qualified class name** (e.g. `\My\SiteAccess\Matcher`) or by a **service identifier prefixed by @** (e.g. `@my_matcher_service`).
>
> - In the case of a fully qualified class name, the matching configuration will be passed in the constructor.
> - In the case of a service, it must implement `eZ\Bundle\EzPublishCoreBundle\SiteAccess\Matcher`. The matching configuration will be passed to `setMatchingConfiguration()`.

> Make sure to type the matcher in correct case. If it is in wrong case like "Uri" instead of "URI," it will happily work on systems like Mac OS X because of case insensitive file system, but will fail when you deploy it to a Linux server. This is a known artifact of PSR-0 autoloading of PHP classes.

Available matchers

Name	Description	Configuration	Example

`URIElement`	Maps a URI element to a siteaccess.
	This is the default matcher used when choosing
	URI matching in setup wizard.
		The element number you want to match (starting from 1).
		
		`ezpublish:`
		`siteaccess:`
		`match:`
		`URIElement: 1`
		
		Important: When using a value > 1, it will concatenate the elements with _
			URI: `/ezdemo_site/foo/bar`
			
			Element number: 1
			Matched siteaccess: ezdemo_site
			
			Element number: 2
			Matched siteaccess: ezdemo_site_foo

`URIText`	Matches URI using *pre* and/or *post* sub-strings in the first URI segment	The prefix and/or suffix (none are required)
		`ezpublish:`
		`siteaccess:`
		`match:`
		`URIText:`
		`prefix: foo`
		`suffix: bar`
			URI: `/footestbar/my/content`
			Prefix: foo
			Suffix: bar
			Matched siteaccess: test

`HostElement`	Maps an element in the host name to a siteaccess.	The element number you want to match (starting from 1).
		`ezpublish:`
		`siteaccess:`
		`match:`
		`HostElement: 2`
			Host name: `www.example.com`
			Element number: 2
			Matched siteaccess: example

`HostText`	Matches a siteaccess in the host name, using *pre* and/or *post* sub-strings.	The prefix and/or suffix (none are required)
		`ezpublish:`
		`siteaccess:`
		`match:`
		`HostText:`
		`prefix: www.`
		`suffix: .com`
			Host name: `www.foo.com`
			Prefix: www.
			Suffix: .com
			Matched siteaccess: foo

`Map\Host`	Maps a host name to a siteaccess.	A hash map of host/siteaccess
		
		``
		ezpublish:
		
		siteaccess:
		
		match:
		
		Map\Host:
		
		www.foo.com: foo_front
		
		adm.foo.com: foo_admin
		
		www.bar-stuff.fr: bar_front
		
		adm.bar-stuff.fr: bar_admin
		``
		
		In eZ Enterprise, when using the `Map\Host` matcher, you need to provide the following line in your Twig template (e.g. in the head of the main template file):
		
		`{{ multidomain_access() }}`
			Map:
			
			- www.foo.com => foo_front
			- admin.foo.com => foo_admin
			
			Host name: www.example.com
			
			Matched siteaccess: foo_front

`Map\URI`	Maps a URI to a siteaccess	A hash map of URI/siteaccess ``` ezpublish: siteaccess: match: Map\URI: something: ezdemo_site foobar: ezdemo_site_admin ``` The name of the `Map\URI` matcher must be the same as the siteaccess name. This also means that only one URI can be addressed by the same matcher.	**URI:** `/something/my/content` Map: • something => ezdemo_site • foobar => ezdemo_site_admin Matched siteaccess: ezdemo_site

`Map\Port`	Maps a port to a siteaccess	A has map of Port/siteaccess
		`ezpublish:`
		`siteaccess:`
		`match:`
		`Match\Port:`
		`80: foo`
		`8080: bar`
			URL: `http://ezpublish.dev:8080/my/content`
			Map:
			• 80: foo
			• 8080: bar
			Matched siteaccess: bar

`Regex\Host`	Matches against a regexp and extracts a portion of it	The regexp to match against
		and the captured element to use
		`ezpublish:`
		`siteaccess:`
		`match:`
		`Regex\Host:`
		`regex: "^(\\w+_sa)$"`
		`# Default is 1`
		`itemNumber: 1`
			Host name: `example_sa`
			regex: `^(\\w+)_sa$`
			itemNumber: 1
			Matched siteaccess: example

`Regex\URI`	Matches against a regexp and extracts a portion of it	The regexp to match against and the captured element to use ``` ezpublish: siteaccess: match: Regex\URI: regex: "^/foo(\\w+)bar" # Default is 1 itemNumber: 1 ```	**URI:** `/footestbar/something` regex: ^/foo(\\w+)bar itemNumber: 1 Matched siteaccess: test

Compound siteaccess matcher

The Compound siteaccess matcher allows you to combine several matchers together:

- http://example.com/en matches site_en (match on host=example.com *and* URIElement(1)=en)
- http://example.com/fr matches site_fr (match on host=example.com *and* URIElement(1)=fr)
- http://admin.example.com matches site_admin (match on host=admin.example.com)

Compound matchers cover the legacy ***host_uri*** matching feature.

They are based on logical combinations, or/and, using logical compound matchers:

- `Compound\LogicalAnd`
- `Compound\LogicalOr`

Each compound matcher will specify two or more sub-matchers. A rule will match if all the matchers, combined with the logical matcher, are positive. The example above would have used `Map\Host` and `Map\Uri`., combined with a `LogicalAnd`. When both the URI and host match, the siteaccess configured with "match" is used.

ezplatform.yml

```
ezpublish:
    siteaccess:
        match:
            Compound\LogicalAnd:
                # Nested matchers, with their
configuration.
                # No need to precise their matching
values (true will suffice).
                site_en:
                    matchers:
                        Map\URI:
                            en: true
                        Map\Host:
                            example.com: true
                    match: site_en
                site_fr:
                    matchers:
                        Map\URI:
                            fr: true
                        Map\Host:
                            example.com: true
                    match: site_fr
            Map\Host:
                admin.example.com: site_admin
```

Matching by request header

It is possible to define which siteaccess to use by setting an **X-Siteaccess** header in your request. This can be useful for REST requests.

In such case, **X-Siteaccess** must be the **siteaccess name** (e.g. *ezdemo_site*).

Matching by environment variable

It is also possible to define which siteaccess to use directly via an **EZPUBLISH_SITEACCESS** environment variable.

This is recommended if you want to get **performance gain** since no matching logic is done in this case.

You can define this environment variable directly from your web server configuration:

Apache VirtualHost example

```
# This configuration assumes that mod_env is activated
<VirtualHost *:80>
    DocumentRoot "/path/to/ezpublish5/web/folder"
    ServerName example.com
    ServerAlias www.example.com
    SetEnv EZPUBLISH_SITEACCESS ezdemo_site
</VirtualHost>
```

This can also be done via PHP-FPM configuration file, if you use it. See PHP-FPM

documentation for more information.

Note about precedence
The precedence order for siteaccess matching is the following (the first matched wins):

1. Request header
2. Environment variable
3. Configured matchers

URILexer and semanticPathinfo

In some cases, after matching a siteaccess, it is neecessary to modify the original request URI. This is for example needed with URI-based matchers since the siteaccess is contained in the original URI and it is not part of the route itself.

The problem is addressed by *analyzing* this URI and by modifying it when needed through the **URI Lexer** interface.

URILexer interface

```
/**
 * Interface for SiteAccess matchers that need to alter
the URI after matching.
 * This is useful when you have the siteaccess in the
URI like "/<siteaccessName>/my/awesome/uri"
 */
interface URILexer
{
    /**
     * Analyses $uri and removes the siteaccess part, if
needed.
     *
     * @param string $uri The original URI
     * @return string The modified URI
     */
    public function analyseURI( $uri );
    /**
     * Analyses $linkUri when generating a link to a
route, in order to have the siteaccess part back in the
URI.
     *
     * @param string $linkUri
     * @return string The modified link URI
     */
    public function analyseLink( $linkUri );
}
```

Once modified, the URI is stored in the ***semanticPathinfo*** request attribute, and the original pathinfo is not modified.

Usage

Cross-siteacess links

When using the *multisite* feature, it is sometimes useful to be able to **generate cross-links** betwee n the different sites.

This allows you to link different resources referenced in the same content repository, but configured independently with different tree roots.

Twig example

```
 {# Linking a location #}
<a href="{{ url( 'ez_urlalias', {'locationId': 42,
'siteaccess': 'some_siteaccess_name'} ) }}">{{
ez_content_name( content ) }}</a>

{# Linking a regular route #}
<a href="{{ url( "some_route_name", {"siteaccess":
"some_siteaccess_name"} ) }}">Hello world!</a>
```

See ez_urlalias documentation page, for more information about linking to a Location

PHP example

```
namespace Acme\TestBundle\Controller;

use eZ\Bundle\EzPublishCoreBundle\Controller as
BaseController;
use
Symfony\Component\Routing\Generator\UrlGeneratorInterfac
e;

class MyController extends BaseController
{
    public function fooAction()
    {
        // ...

        $location =
$this->getRepository()->getLocationService()->loadLocati
on( 123 );
        $locationUrl = $this->generateUrl(
            $location,
            array( 'siteaccess' =>
'some_siteaccess_name' ),
            UrlGeneratorInterface::ABSOLUTE_PATH
        );

        $regularRouteUrl = $this->generateUrl(
            'some_route_name',
            array( 'siteaccess' =>
'some_siteaccess_name' ),
            UrlGeneratorInterface::ABSOLUTE_PATH
        );

        // ...
    }
}
```

Important
As siteaccess matchers can involve hosts and ports, it is **highly recommended** to generate cross-siteaccess links in an absolute form (e.g. using `url()` Twig helper).

Troubleshooting

- The **first matcher succeeding always wins**, so be careful when using *catch-all* matchers like `URIElement`.
- If passed siteaccess name is not a valid one, an `InvalidArgumentException` will be thrown.
- If matcher used to match the provided siteaccess doesn't implement `VersatileMatcher`, the link will be generated for the current siteaccess.
- When using `Compound\LogicalAnd`, all inner matchers **must match**. If at least one matcher doesn't implement `VersatileMatcher`, it will fail.
- When using `Compound\LogicalOr`, the first inner matcher succeeding will win.

Under the hood

To implement this feature, a new `VersatileMatcher` was added to allow siteaccess matchers to be able to *reverse-match*.
All existing matchers implement this new interface, except the Regexp based matchers which have been deprecated.

The siteaccess router has been added a `matchByName()` method to reflect this addition. Abstract URLGenerator and `DefaultRouter` have been updated as well.

Note
Siteaccess router public methods have also been extracted to a new interface, `SiteAccessRouterInterface`.

Landing Page - Known limitation
In eZ Studio's Landing Page you can encounter a 404 error when clicking a relative link which points to a different siteaccess (if the Content item being previewed does not exist in the previously used siteaccess). This is because detecting siteaccesses when navigating in preview is not functional yet. This is a known limitation that is awaiting resolution.

Dynamic Settings Injection

Before 5.4, if you wanted to implement a service needing siteaccess-aware settings (e.g. language settings), you needed to inject the whole `ConfigResolver` (`ezpublish.config.resolver`) and get the needed settings from it. This was neither very convenient nor explicit.

The goal of this feature is to allow developers to inject these dynamic settings explicitly from their service definition (yml, xml, annotation, etc.).

Syntax

Static container parameters follow the `%<parameter_name>%` syntax in Symfony.

Dynamic parameters have the following: `$<parameter_name>[; <namespace>[; <scope>]]$`, default namespace being `ezsettings`, and default scope being the current siteaccess.

For more information, see ConfigResolver documentation.

DynamicSettingParser

This feature also introduces a *DynamicSettingParser* service that can be used for adding support of the dynamic settings syntax.
This service has `ezpublish.config.dynamic_setting.parser` for ID and implements `eZ\Bundle\EzPublishCoreBundle\DependencyInjection\Configuration\SiteAccessAware\DynamicSettingParserInterface`.

Limitations

A few limitations still remain:

- It is not possible to use dynamic settings in your semantic configuration (e.g. `config.yml` or `ezplatform.yml`) as they are meant primarily for parameter injection in services.
- It is not possible to define an array of options having dynamic settings. They will not be parsed. Workaround is to use separate arguments/setters.
- Injecting dynamic settings in request listeners is **not recommended**, as it won't be resolved with the correct scope (request listeners are **instantiated before SiteAccess match**). Workaround is to inject the ConfigResolver instead, and resolving the setting in your `onKernelRequest` method (or equivalent).

Examples

Injecting an eZ parameter

Defining a simple service needing `languages` parameter (i.e. prioritized languages).

> **Note**
> Internally, `languages` parameter is defined as `ezsettings.<siteaccess_name>.languages`, `ezsettings` being eZ internal *namespace*.

Before 5.4

```
parameters:
    acme_test.my_service.class:
Acme\TestBundle\MyServiceClass

services:
    acme_test.my_service:
        class: %acme_test.my_service.class%
        arguments: [@ezpublish.config.resolver]

namespace Acme\TestBundle;
```

```
use eZ\Publish\Core\MVC\ConfigResolverInterface;

class MyServiceClass
{
    /**
 * Prioritized languages
 *
 * @var array
 */
    private $languages;

    public function __construct( ConfigResolverInterface
$configResolver )
    {
        $this->languages =
$configResolver->getParameter( 'languages' );
    }
}
```

After, with setter injection (preferred)

```
parameters:
    acme_test.my_service.class:
Acme\TestBundle\MyServiceClass

services:
    acme_test.my_service:
        class: %acme_test.my_service.class%
        calls:
            - [setLanguages, ["$languages$"]]
```

```
namespace Acme\TestBundle;

class MyServiceClass
{
    /**
 * Prioritized languages
 *
 * @var array
 */
    private $languages;

    public function setLanguages( array $languages =
null )
    {
        $this->languages = $languages;
    }
}
```

Important: Ensure you always add `null` as a default value, especially if the argument is type-hinted.

After, with constructor injection

```
parameters:
    acme_test.my_service.class:
Acme\TestBundle\MyServiceClass

services:
    acme_test.my_service:
        class: %acme_test.my_service.class%
        arguments: ["$languages$"]
```

```
namespace Acme\TestBundle;

class MyServiceClass
{
    /**
 * Prioritized languages
 *
 * @var array
 */
    private $languages;

    public function __construct( array $languages )
    {
        $this->languages = $languages;
    }
}
```

Tip
Setter injection for dynamic settings should always be preferred, as it makes it possible to update your services that depend on them when ConfigResolver is updating its scope (e.g. when previewing content in a given SiteAccess). **However, only one dynamic setting should be injected by setter** .

Constructor injection will make your service be reset in that case.

Injecting 3rd party parameters

```
parameters:
    acme_test.my_service.class:
Acme\TestBundle\MyServiceClass
    # "acme" is our parameter namespace.
    # Null is the default value.
    acme.default.some_parameter: ~
    acme.ezdemo_site.some_parameter: foo
    acme.ezdemo_site_admin.some_parameter: bar

services:
    acme_test.my_service:
        class: %acme_test.my_service.class%
        # The following argument will automatically
resolve to the right value, depending on the current
SiteAccess.
        # We specify "acme" as the namespace we want to
use for parameter resolving.
        calls:
            - [setSomeParameter,
["$some_parameter;acme$"]]
```

```
namespace Acme\TestBundle;
class MyServiceClass
{
    private $myParameter;
    public function setSomeParameter( $myParameter =
null )
    {
        // Will be "foo" for ezdemo_site, "bar" for
ezdemo_site_admin, or null if another SiteAccess.
        $this->myParameter = $myParameter;
    }
}
```