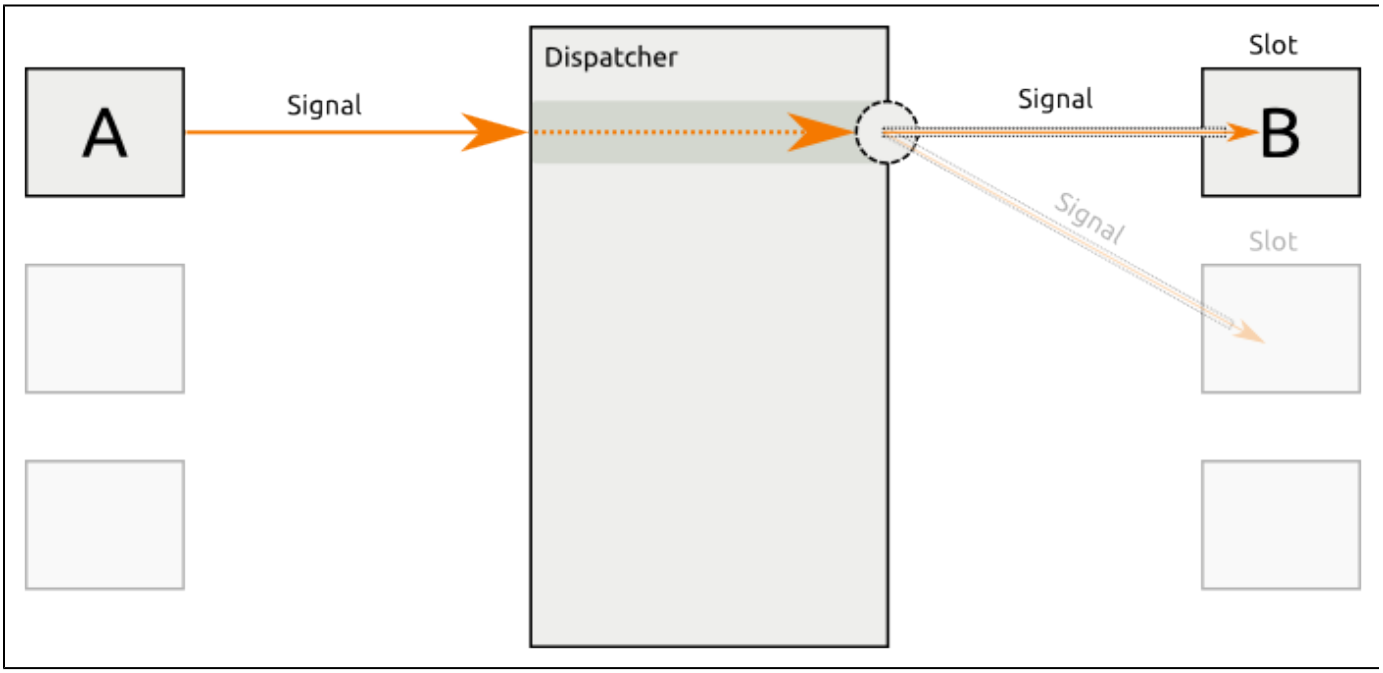
Signal-Slot

The Signal-Slot system provides a means for realizing loosely coupled dependencies in the sense that a code entity A can react on an event occurring in code entity B, without A and B knowing each other directly. This works by dispatching event information through a central third instance, the so called dispatcher:



In the shown schematics, object B and one other object are interested in a certain signal. B is a so-called Slot that can be announced to be interested in receiving a Signal (indicated by the circular connector to the dispatcher). Object A now sends the corresponding Signal. The Dispatcher takes care of realizing the dependency and informs the Slot A (and one other Slot) about the occurrence of the Signal.

Signals roughly equal events, while Slots roughly equal event handlers. An arbitrary number (0…n) of Slots can listen for a specific Signal. Every object that receives the Dispatcher as a dependency can send signals. However, the following conditions apply (that typically do not apply to event handling systems):

- A Slot cannot return anything to the object that issued a signal
- It is not possible for a Slot to stop the propagation of a Signal, i.e. all listening Slots will eventually receive the Signal

Those two conditions allow the asynchronous processing of Slots. That means: It is possible to determine, by configuration, that a Slot must not receive a Signal in the very same moment it occurs, but to receive it on a later point in time, maybe after other Signals from a queue have been processed or even on a completely different server.

Signal

A Signal represents a specific event, e.g. that a content version has been published. It consists of information that is significant to the event, e.g. the content ID and version number. Therefore, a Signal is represented by an object of a class that is specific to the Signal and that extends from e `Z\Publish\Core\SignalSlot\Signal`. The full qualified name of the Signal class is used to uniquely identify the Signal. For example, the class `eZ\Publish\Core\SignalSlot\Signal\ContentService\PublishVersionSignal` identifies the example Signal.

In order to work properly with asynchronous processing, Signals must not consist of any logic and must not contain complex data structures (such as further objects and resources). Instead, they must be exportable using the `__set_state()` method, so that it is possible to transfer a Signal to a different system.

> **Note**
> Signals can theoretically be sent by any object that gets hold of a SignalDispatcher (`eZ\Publish\Core\SignalSlot\SignalDispatcher`). However, at a first stage, **Signals are only sent by special implementations of the Public API to indicate core events**. These services must and will be used by default and will wrap the original service implementations.

Slot

A Slot extends the system by realizing functionality that is executed when a certain Signal occurs. To implement a Slot, you must create a class that derives from `eZ\Publish\Core\SignalSlot\Slot`. The full qualified name of the Slot class is also used as the unique identifier of the Slot. The Slot base class requires you to implement the single method `receive()`. When your Slot is configured to listen to a certain Signal and this Signal occurs, the `receive()` method of your Slot is called.

Inside the `receive()` method of your Slot you can basically realize any kind of logic. However, it is recommended that you only dispatch the action to be triggered to a dedicated object. This allows you to trigger the same action from within multiple Slots and to re-use the implementation from a completely different context.

Note that, due to the nature of Signal-Slot, the following requirements must be fulfilled by your Slot implementation:

- A Slot must not return anything to the Signal issuer
- A Slot must be aware that it is potentially executed delayed or even on a different server

> **Important**: A single Slot should not take care of processing more than one Signal. Instead, if you need to trigger same or similar actions as different Signals occur. you should encapsulate these actions into a dedicated class, of which your Slots receive an instance to trigger this action.

Example: Updating URL aliases

Updating URL aliases is a typical process that can be realized through a Signal-Slot extension for different reasons:

- The action must be triggered on basis of different events (e.g. content update, location move, etc.)
- Direct implementation would involve complex dependencies between otherwise unrelated services
- The action is not critical to be executed immediately, but can be made asynchronous, if necessary

As a first step it needs to be determined for which Signals we need to listen in order to update URL aliases. Some of them are:

- `eZ\Publish\Core\SignalSlot\Signal\ContentService\PublishVersionSignal`
- `eZ\Publish\Core\SignalSlot\Signal\LocationService\CopySubtreeSignal`
- `eZ\Publish\Core\SignalSlot\Signal\LocationService\MoveSubtreeSignal`
- …

There are of course additional Signals that trigger an update of URL aliases, but these are left out for simplicity here.

Now that we identified some Signals to react upon, we can start implementing Slots for these signals. For the first Signal, which is issued as soon as a new version of Content is published, there exists a method in `eZ\Publish\SPI\Persistence\Content\UrlAlias\Handler` for exactly that purpose: `publishUrlAliasForLocation()`. The Signal contains the ID of the content object and its newly published version number. Using this information, the corresponding Slot can fulfill its purposes with the following steps:

1. Load the corresponding content and its locations
2. Call the URL alias creation method for each location

To achieve this, the Slot has 2 dependencies:

- `eZ\Publish\SPI\Persistence\Content\Handler`
  to load the content itself in order to retrieve the names
- `eZ\Publish\SPI\Persistence\Content\Location\Handler`
  to load the locations
- `eZ\Publish\SPI\Persistence\Content\UrlAlias\Handler`
  to create the aliases for each location

So, a stub for the implementation could look like this:

```
namespace Acme\TestBundle\Slot;

use eZ\Publish\Core\SignalSlot\Slot as BaseSlot;
use eZ\Publish\API\Repository\Repository;
use eZ\Publish\SignalSlot\Signal;

class CreateUrlAliasesOnPublishSlot extends BaseSlot
{
    /**
     * @var \eZ\Publish\API\Repository\Repository
     */
    private $repository;
    public function __construct( Repository $repository )
    {
        $this->repository = $repository;
    }

    public function receive( Signal $signal )
    {
        if ( !$signal instanceof Signal\ContentService\PublishVersionSignal )
        {
            return;
        }
        // Load content
        // Load locations
        // Create URL aliases
    }
}
```

In order to make the newly created Slot react on the corresponding Signal, the following steps must be performed:

1. Make the Slot available through the Symfony service container as a service
2. Register the Slot to react to the Signal of type `eZ\Publish\Core\SignalSlot\Signal\ContentService\PublishVersionSignal`

See How to listen to Core events recipe in the developer cookbook for more information.