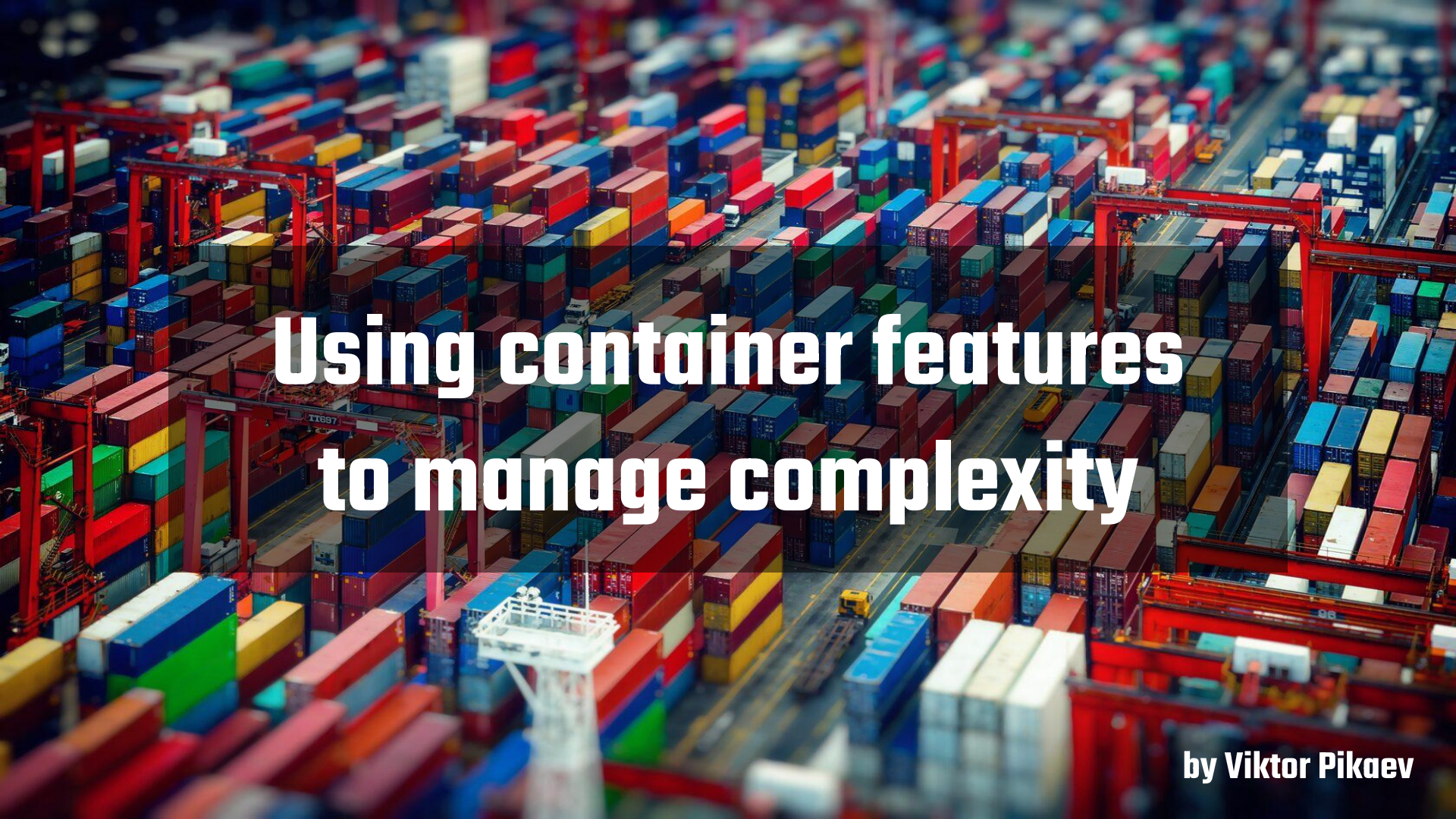


Using container features to manage complexity

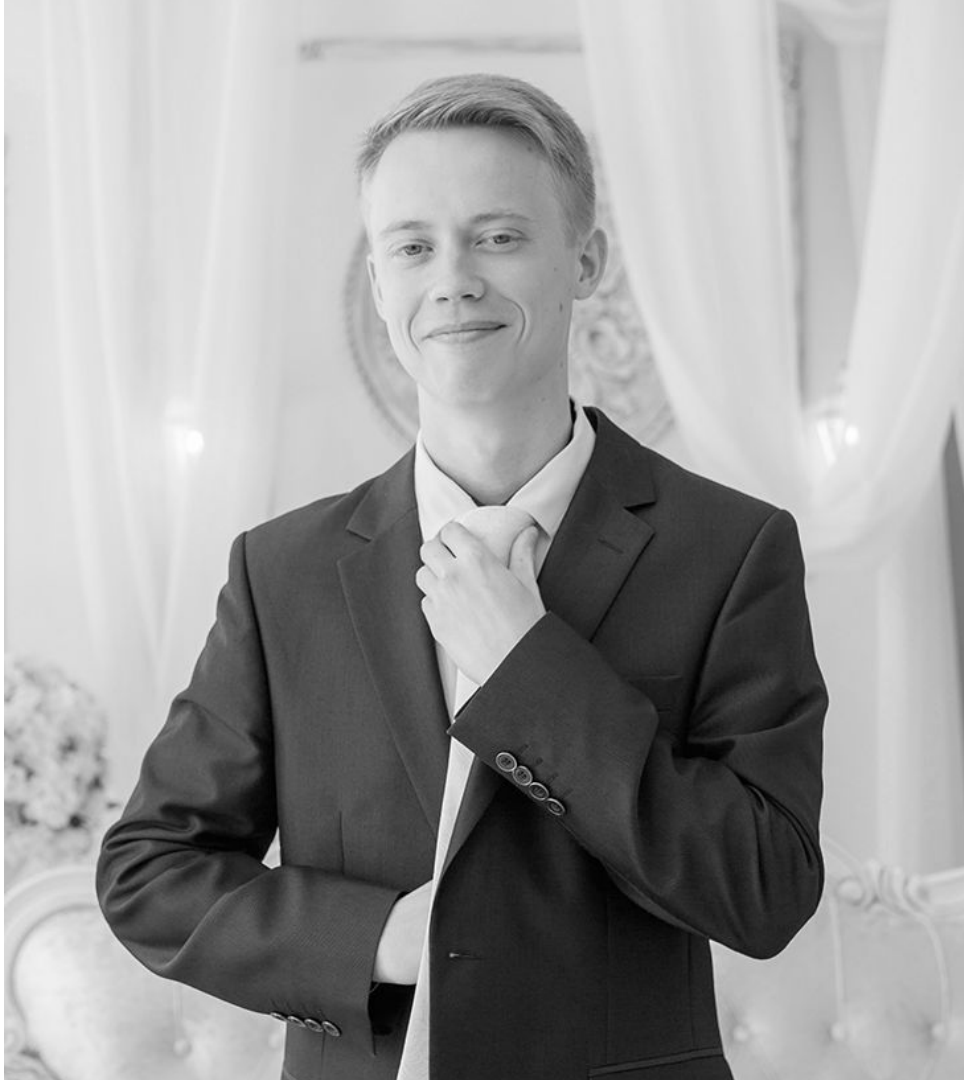
by Viktor Pikaev

Viktor Pikaev

- Senior Backend Developer at Boozt
- More than 16 years of experience
- A big fan of Rust and Haskell
- Play Factorio

haru-atari.com

linkedin.com



Boozt Platform

- Boozt - Nordic technology company
- More than 200 developers and engineers
- More than 40 tailor-made systems (mostly using Symfony)
- Platform People post on LinkedIn with #boozttech

linkedin.com

medium.com

Boozt.com
PLATFORM
POWER TO THE PEOPLE

Symfony container

The container allows you to centralize the way objects are constructed.
It makes your life easier, promotes a strong architecture and is super fast!

In other words, it's the heart of your application.

Autowiring

- Defines dependencies automatically
- Covers most of use cases
- Needs configuration
  for rare cases only

```yaml
# services.yaml
services:
  _defaults:
    autowire: true # <---

  DemoApp\:
    resource: './src'

  # Specify a class for an interface
  GuzzleHttp\ClientInterface:
    class: GuzzleHttp\Client

  # Specify a value for an argument
  DemoApp\SpamChecker:
    arguments:
      $textAnalyser: '@DemoApp\PlainTextAnalyser'
```

Advanced configuration with Service Tags

Definition:

Service tags are a way to tell Symfony or other third-party bundles that our service should be registered in some special way.

- Are used to group services and use them in a specific way
- Are used for advanced configuration that still can be described declaratively

Manual tagging

- Service can be tagged manually in the configuration file
- Service can have more than one tag
- Tags can have parameters

Symfony built-in Service Tags

```yaml
# services.yaml
services:
  _defaults:
    autowire: true

  DemoApp\:
    resource: './src'

  # Assign a tag
  DemoApp\CsvReportGenerator:
    tags: [ 'report_generator' ]

  # Assign a tag with parameters
  DemoApp\ExcelReportGenerator:
    tags: [ { name: 'report_generator', oldFormat:
true } ]
```

Tagging by the interface

- Services implementing the interface will be tagged automatically

```yaml
# services.yaml
services:
  _defaults:
    autowire: true

  _instanceof:
    DemoApp\ReportGeneratorInterface:
      tags: [ 'report_generator' ]

  DemoApp\:
    resource: './src'
```

Autoconfiguration

- Autoconfiguration will enable tagging based on the code
- No configuration required
- Allows us to use interfaces and PHP attributes for tagging

```yaml
# services.yaml
services:
  _defaults:
    autowire: true
    autoconfigure: true # <---

  DemoApp\:
    resource: './src'
```

Autoconfiguration (example)

```
class EventSubscriber implements EventSubscriberInterface
{
    // This class will be automatically tagged as "kernel.event_subscriber"
    // Because it implements the EventSubscriberInterface
}
```

Autoconfiguration (example)

```php
#[AsCommand(name: 'app:healthcheck')]
class HealthcheckCommand extends Command
{
    // This class will be automatically tagged as "console.command"
    // Tag's parameter "name" will be filled with the "app:healthcheck" value
    // Because it has the AsCommand  attribute
}
```

Autoconfiguration (example)

```php
#[AutoconfigureTag('report_generator')]
class CsvReportGenerator
{
    // This class will be tagged as "report_generator"
}


#[AutoconfigureTag('report_generator', ['oldFormat' => true])]
class ExcelReportGenerator
{
    // This class will be tagged as "report_generator"
    // Tag's parameter "oldFormat" will be filled with the true value
}
```

Autoconfiguration registration

- Autoconfiguration rules should be registered
  in the Kernel::build() method with a container builder
- Use ContainerBuilder::registerForAutoconfiguration()
  for interfaces autoconfiguration
- Use ContainerBuilder::registerAttributeForAutoconfiguration()
  for PHP attributes autoconfiguration

Use of tagged services

- Tagging without subsequent use makes no sense
- Services are tagged to be grouped into collections
- These collections can be injected to another services

```php
#[AutoconfigureTag('spam_checker')]
interface SpamChecker
{ /* ... */ }

class TextChecker implements SpamChecker
{ /* ... */ }

class AuthorChecker implements SpamChecker
{ /* ... */ }
```

Tagged iterator

Injects services tagged by the specified tag as a lazy iterator

```php
class SpamFilter
{
    /**
     * @param iterable<array-key, SpamChecker> $checkers
     */
    public function __construct(#[TaggedIterator('spam_checker')] iterable $checkers)
    {
        foreach ($checkers as $checker) {
            // $checker instanceof SpamChecker
        }
    }
}
```

Tagged locator

Injects services tagged by the specified tag as a service locator

```php
class SpamFilter
{
    public function __construct(#[TaggedLocator('spam_checker')] ServiceLocator $checkers)
    {
        $checkers->has('...');
        $checkers->get('...');
    }
}
```

Control container compilation with Compiler Pass

Definition:

Compiler passes give us an opportunity to manipulate other service definitions that have been registered with the service container.

- Indispensable for describing complex logic for the container
- Manipulate service definitions during container compilation
- Can terminate compilation

Compiler pass registration

- Should implement a CompilerPassInterface interface
- Should be registered in the Kernel::build() method

- Kernel itself can be a compiler pass
- In that case it should not be registered

Compiler pass registration (example)

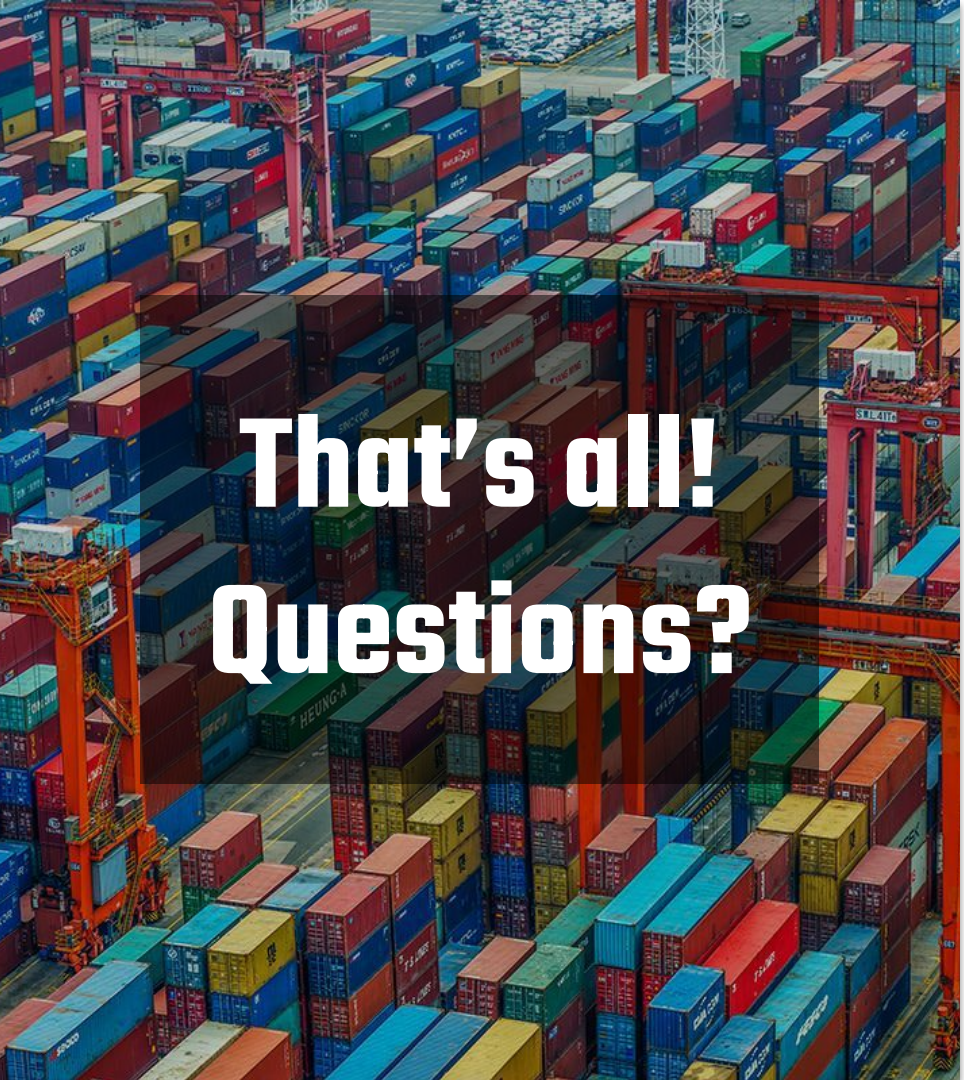
```php
// DemoCompilerPass.php
class DemoCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container) { /* ... */ }
}

// Kernel.php
class Kernel extends BaseKernel
{
    protected function build(ContainerBuilder $container):void
    {
        parent::build($container);
        $container->addCompilerPass(new DemoCompilerPass());
    }
}
```

Compiler pass usage

- Compiler pass works with service definitions, not the services themselves

- Can access service definitions and their metadata
- Can modify or remove definitions
- Can add new definitions
- Can terminate the compilation process

```php
class DemoCompilerPass
implements CompilerPassInterface
{
    public function
    process(ContainerBuilder $container)
    {
        $taggedServiceIds = $container
            ->findTaggedServiceIds(/* ... */);

        $container->hasDefinition(/* ... */);

        $definition = $container
          ->getDefinition(/* ... */);

        $definition->addError(/* ... */);

        $definition->addMethodCall(/* ... */);

        $container->addDefinitions(/* ... */);
    }
}
```



That's all!
Questions?

slides

blog post

haru-atari.com

linkedin.com