Software Engineering Conference Russia 2018

October 12-13
Moscow

SOLID: the core principles of success of the Symfony web framework and of your applications

RIABCHENKO Vladyslav

RIABCHENKO Vladyslav

Technical architect at Webnet (France)

5+ years full stack web-developer

Symfony certified developer and contributor

https://vria.eu

contact@vria.eu

https://twitter.com/RiaVlad

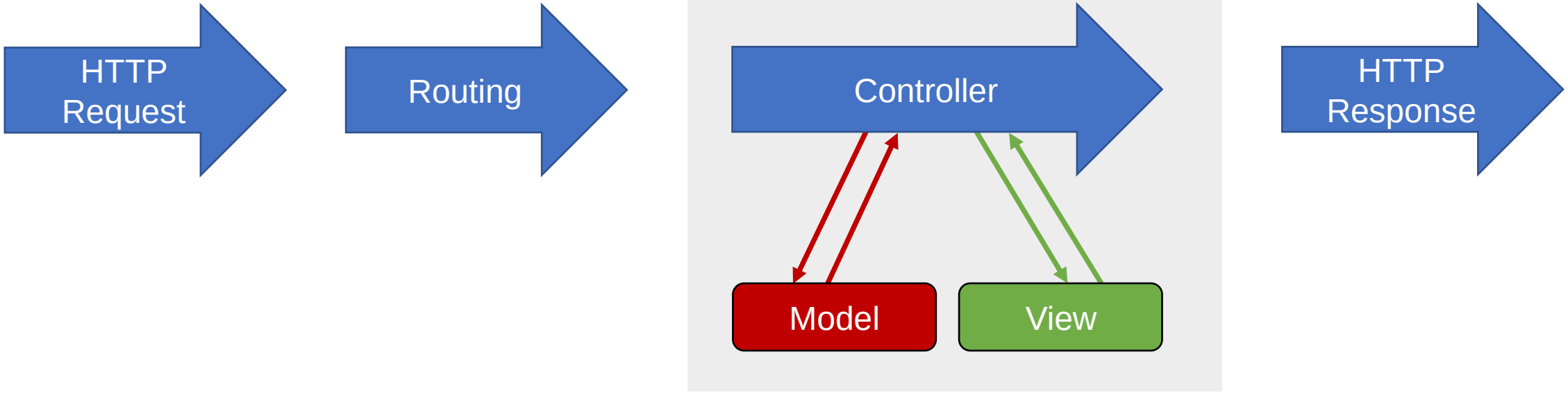
Plan

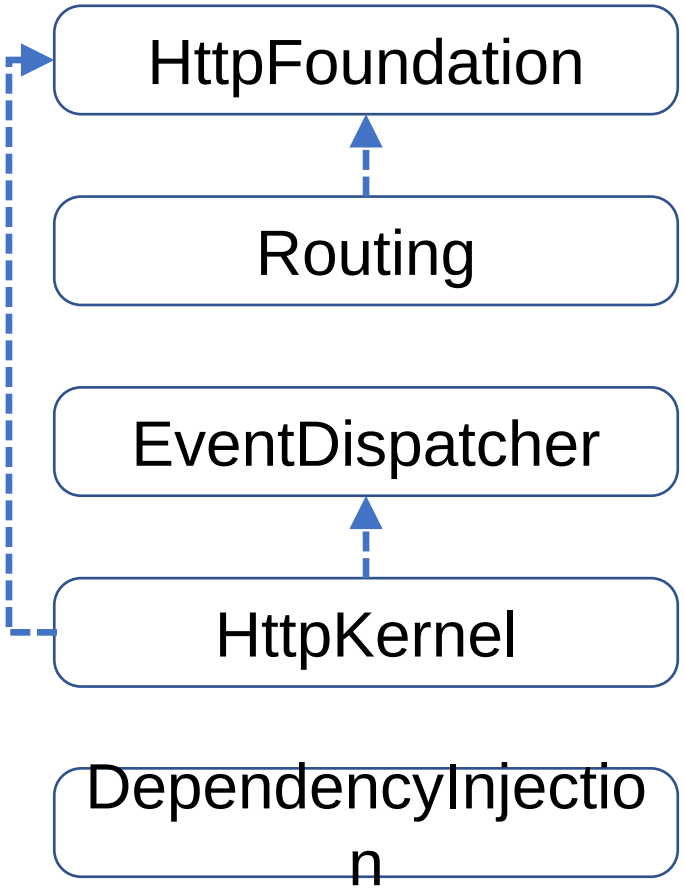
1. Symfony
2. SOLID
3. Single responsibility principle
4. Open/closed principle
5. Liskov substitution principle
6. Interface segregation principle
7. Dependency inversion principle

Symfony framework

Symfony

1. Set of reusable PHP components
2. PHP framework for web projects





Object-oriented layer for request, response, session, etc.

Maps requests to variables (controller, path parameters)

Mediator that allows decoupled components to communicate by dispatching events

Heart of framework that manages HTTP request-response cycle

Service container that instantiates service objects out of configuration and injects their dependencies

and 45 others including Security, Form, Console, Cache, Asset, Validator, Serializer, etc.

Bundles are reusable plugins that provide services, routes, controllers, templates, etc.

FrameworkBundle	Configures Symfony components and glues them in web app. It registers services and event listeners.
doctrine/orm	DBAL and ORM libraries to communicate with databases.
twig/twig	Template engine.
Your code	Provides routes, controllers, services, business logic, templates for your purpose.

SOLID principles

S – Single Responsibility Principle

O – Open/Closed Principle

L – Liskov Substitution Principle

I – Interface Segregation Principle

D – Dependency Inversion Principle

SOLID helps to design a system that is

- Reusable
- Flexible
- Maintainable
- Understandable

… and avoid a system that is

- Fragile
- Duplicated
- Viscose
- Unreasonably complex

Single responsibility principle

A class should have only a single responsibility. Responsibility is a reason to change.

```php
class ReportService
{
    public function createReport($data)
    {
        // working with report manager at domain layer
    }

    public function getLastThreeReportsByAuthor($author)
    {
        // working with database at ORM or DBAL layer
    }

    public function searchBy($criteria)
    {
        // working with database at ORM or DBAL layer
    }

    public function export(Report $report)
    {
        // working with filesystem at by means of PHP functions
    }
}
```

Manager

```php
// working with report manager at domain layer (business logic)
class ReportManager
{
    public function createReport($data)
    { ... }
}
```

Exporter

```php
// working with filesystem and pdf, excel, csv, etc.
class ReportExporter
{
    public function export(Report $report)
    { ... }
}
```

Repository

```php
// working with database at ORM and/or DBAL layer
class ReportRepository
{
    public function getLastThreeReportsByAuthor($author)
    { ... }

    public function searchBy($criteria)
    { ... }
}
```

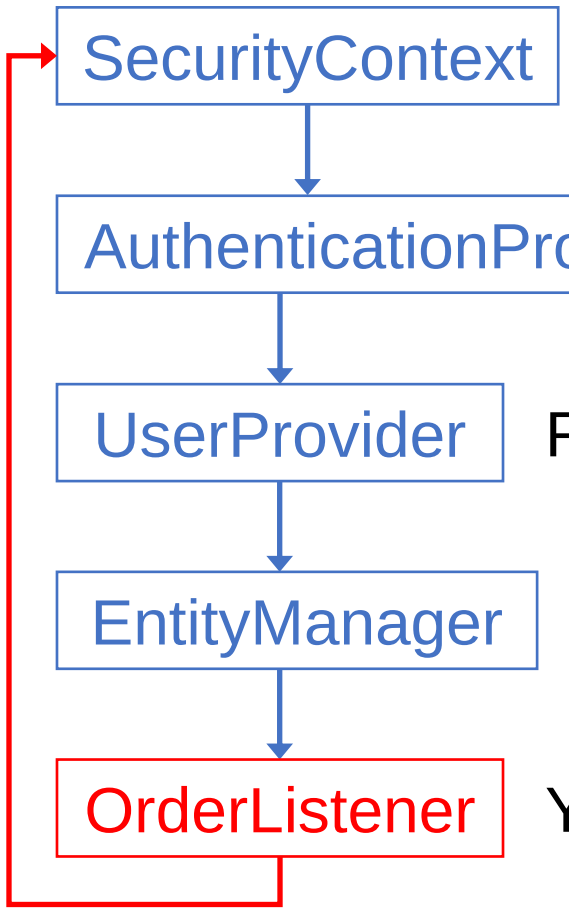
Feature : Store users in database

Feature : Authenticate users against database (login form, HTTP auth, whatever...)

Feature : Store user credentials in session and authenticate users at latter requests

Feature : Store orders in database

Feature : On creation of order assign a currentrly connected user to it.



Stores a user token setToken($token), getToken()
Checks if a current user authorization isGranted('attr', $sub)

Authenticates a token

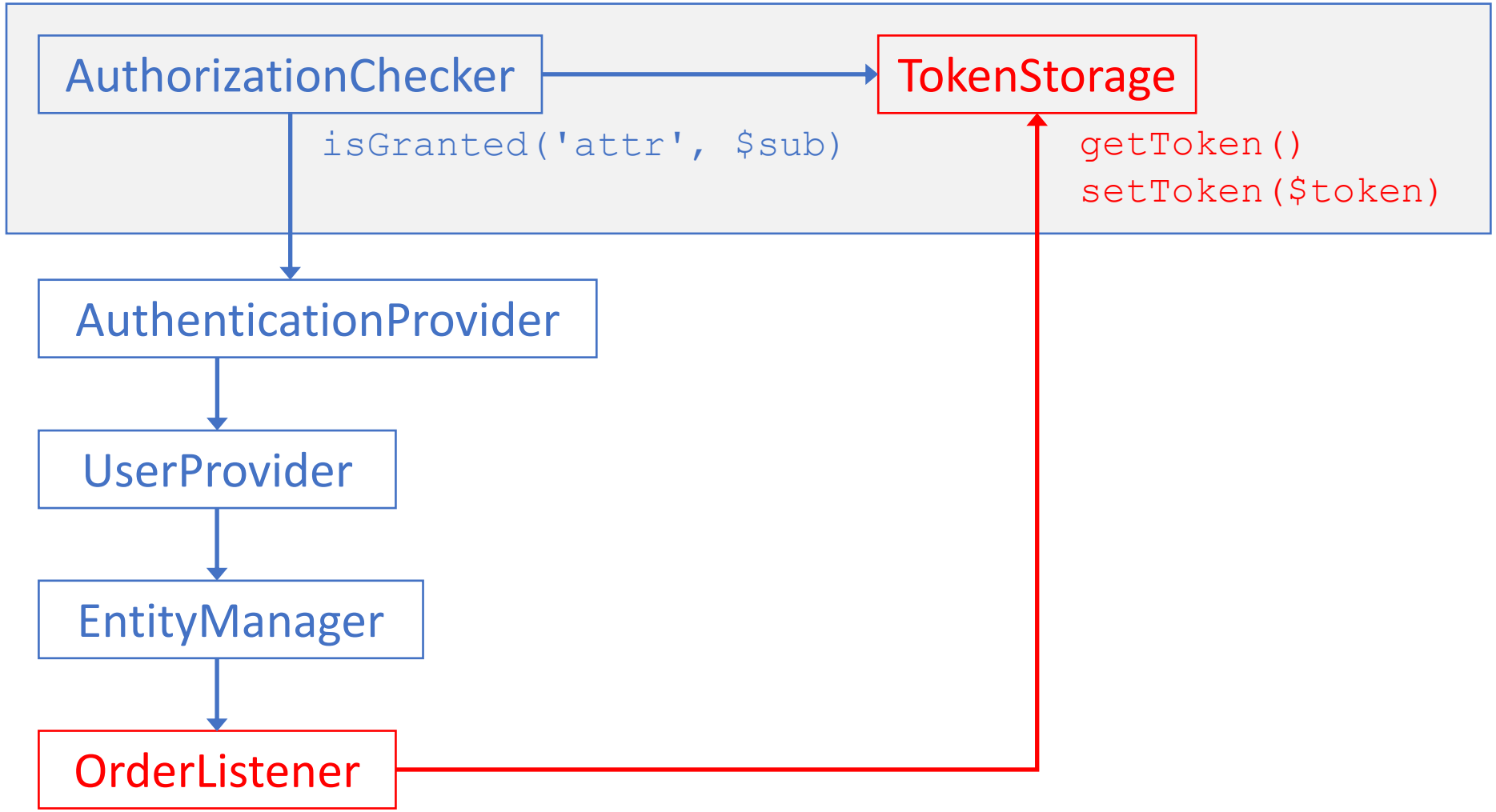
Retrieves or refreshes user from database

All interaction with database at ORM and DBAL levels

Your listener that modifies order before it gets inserted

Injecting SecurityContext service in any EntityManager's listener created a circular dependency.

Solution is splitting SecurityContext into AuthorizationChecker and TokenStorage



Open/closed principle

Entities should be open for extension, but closed for modification.

Open for extension:

- Alter or add behavior
- Extend an entity with new properties

Closed for modification:

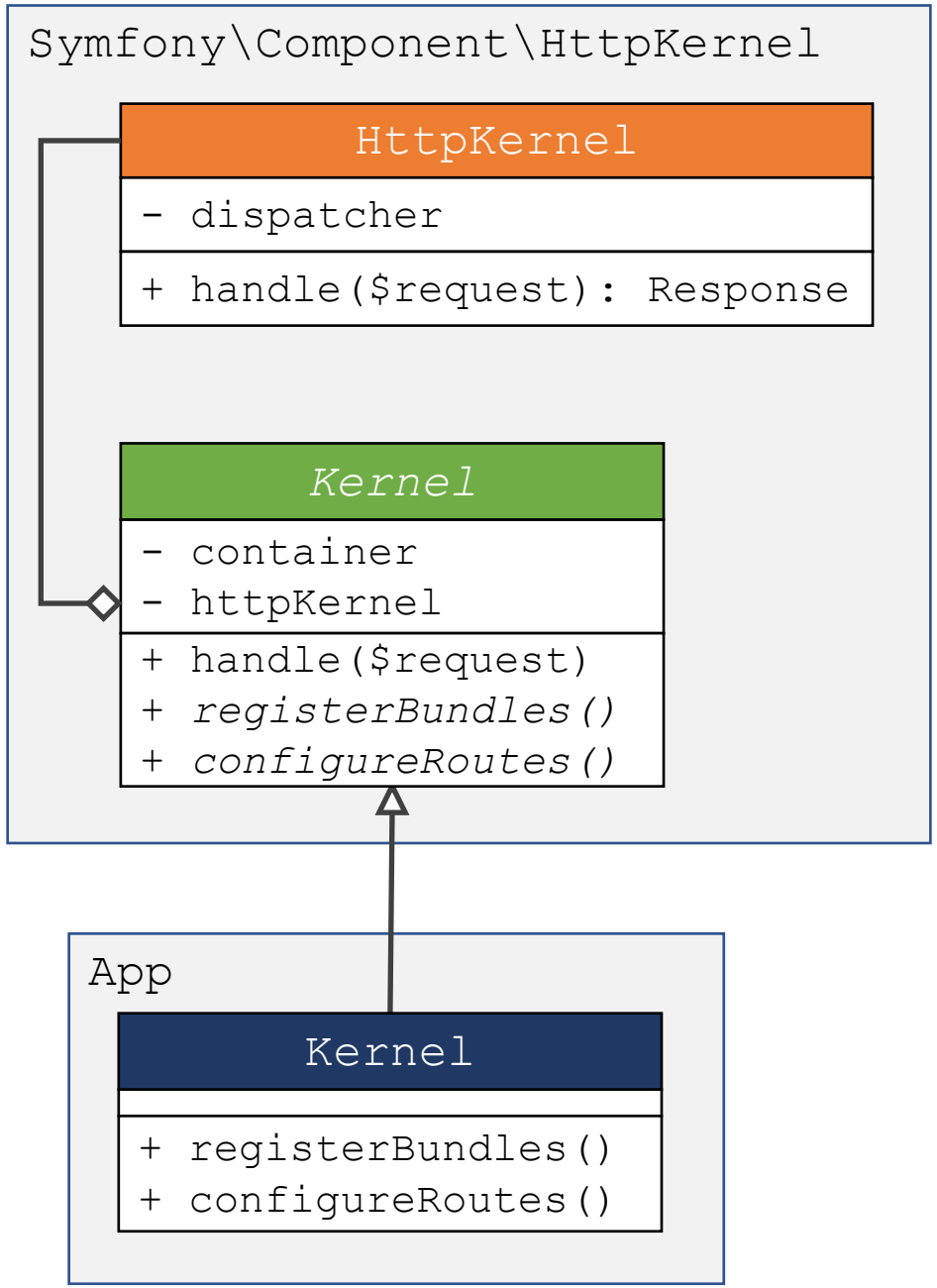
- Adapt entity without modifying its source code
- Provide an entity with clear and stable interface

Techniques to stay in compliance with the open/closed principle:

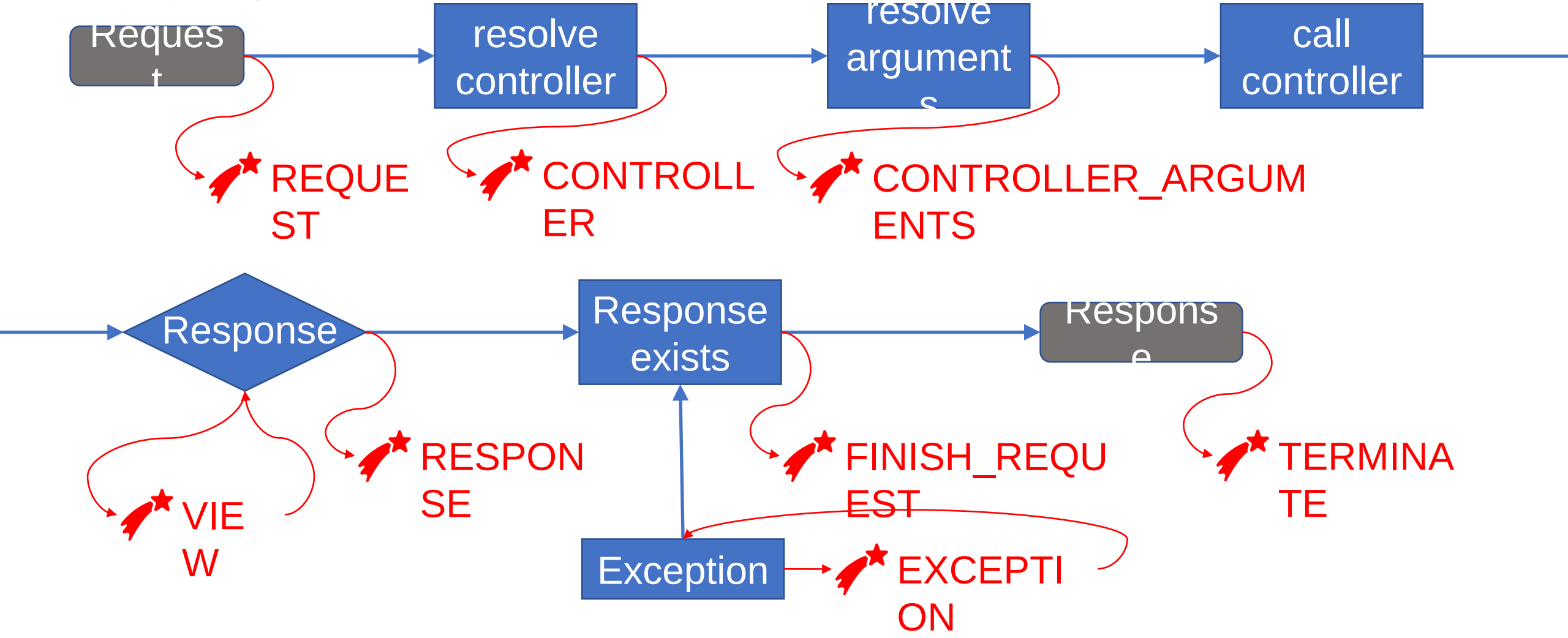
- Abstraction / Inheritance / Polymorphism
- Dependency injection
- Single responsibility principle
- Design patterns. For example, those of GoF:
  - Abstract Factory, Factory Method, Builder, Prototype,
  - Bridge, Decorator, Proxy,
  - Command, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

All HTTP requests are treated by front controller.

```php
// public/index.php
use App\Kernel;
use Symfony\Component\HttpFoundation\Request;

$kernel = new Kernel($env, $debug);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
```



Request-Response cycle is open for extension thanks to events dispatched at every step.

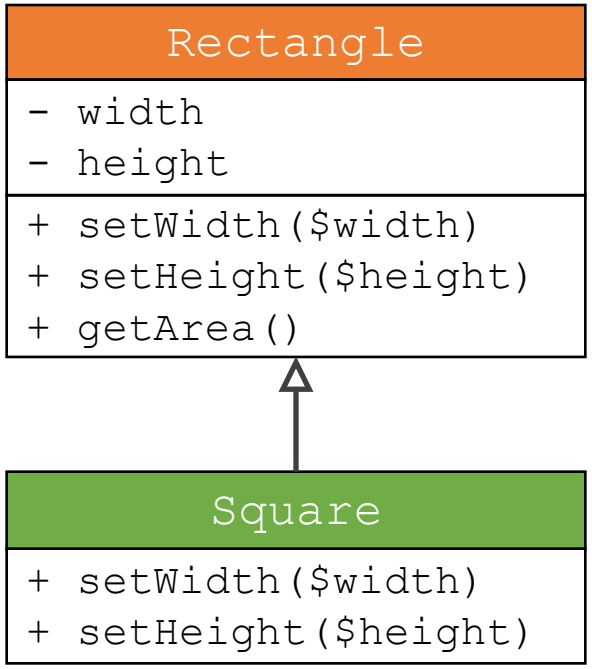


Liskov substitution principle

Objects should be replaceable with instances of their subtypes without altering the correctness of the program.

LSP describes behavioral subtyping – semantic rather than merely syntactic relation.

Square **is a** regular rectangle with four equal sides.



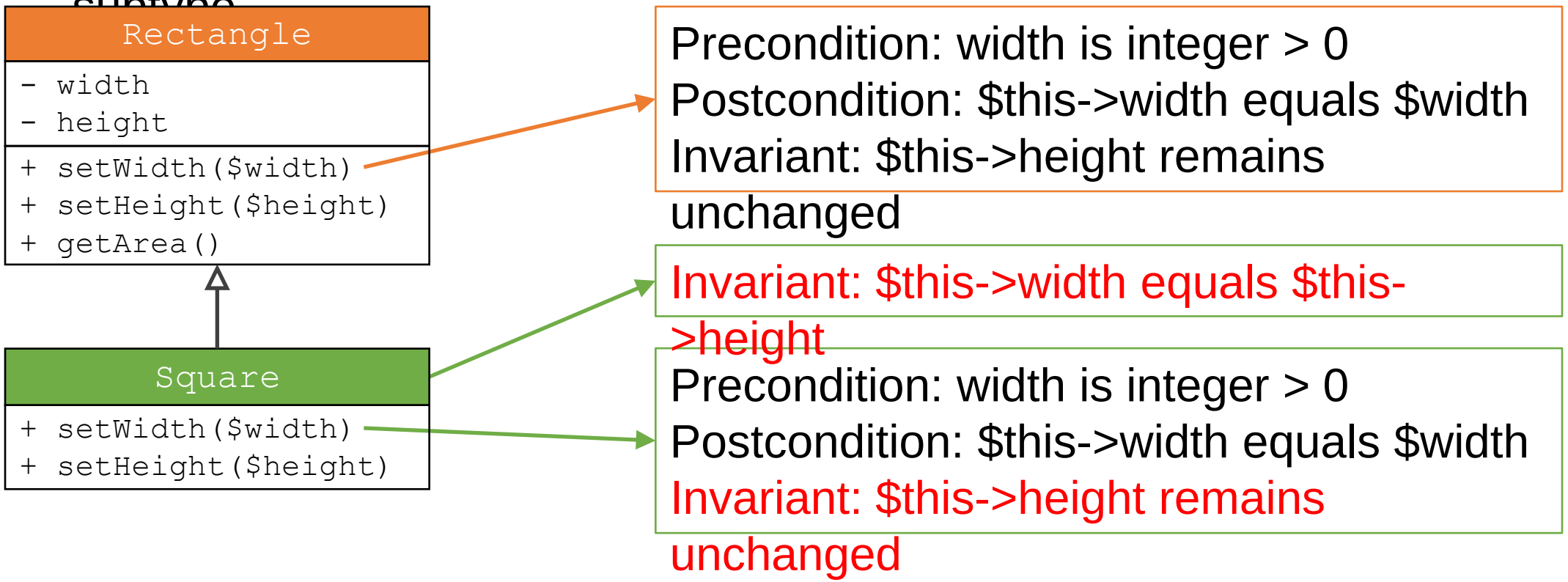

```php
class Square extends Rectangle
{
    public function setWidth($width)
    {
        $this->width = $width;
        $this->height = $width;
    }

    public function setHeight($height)
    {
        $this->height = $height;
        $this->width = $height;
    }
}
```

```php
function client(Rectangle $rect)
{
    $rect->setHeight(8);
    $rect->setWidth(3);

    assert($rect->area() == 24);
}
```

Client will fail when you pass Square object

LSP suggests behavioral conditions resembling those of design by contract methodology:

- **Preconditions** cannot be strengthened in a subtype
- **Postconditions** cannot be weakened in a subtype
- **Invariants** of the supertype must be preserved in a subtype

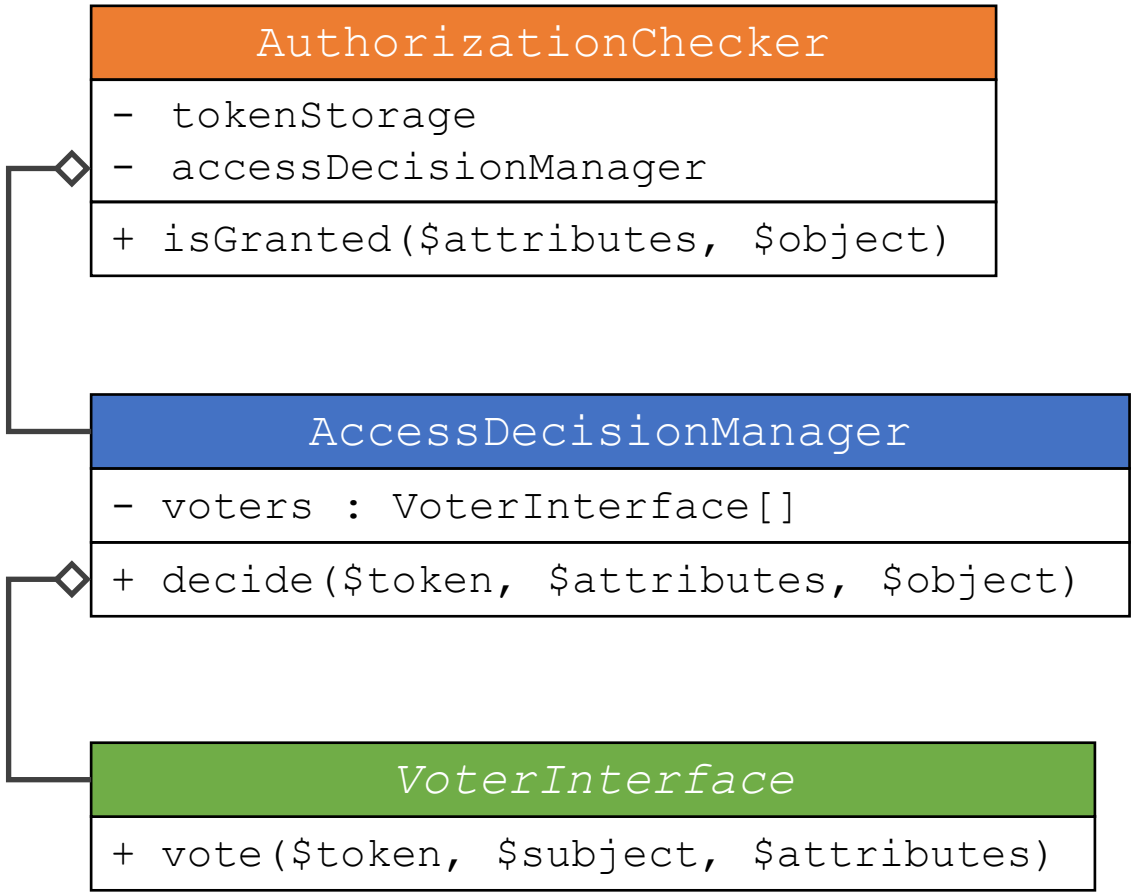


Authorization – verifying access rights/privileges of currently authenticated user.

In Symfony authorization is based on:

- Token: the user's authentication information (roles, etc.),
- Attributes: rights/privileges to verify the access to,
- Object: Any object for which access control needs to be checked.

```
$authorizationChecker->isGranted('ROLE_ADMIN');
$authorizationChecker->isGranted('EDIT', $comment);
$authorizationChecker->isGranted(['VIEW', 'MANAGE'], $order);
```

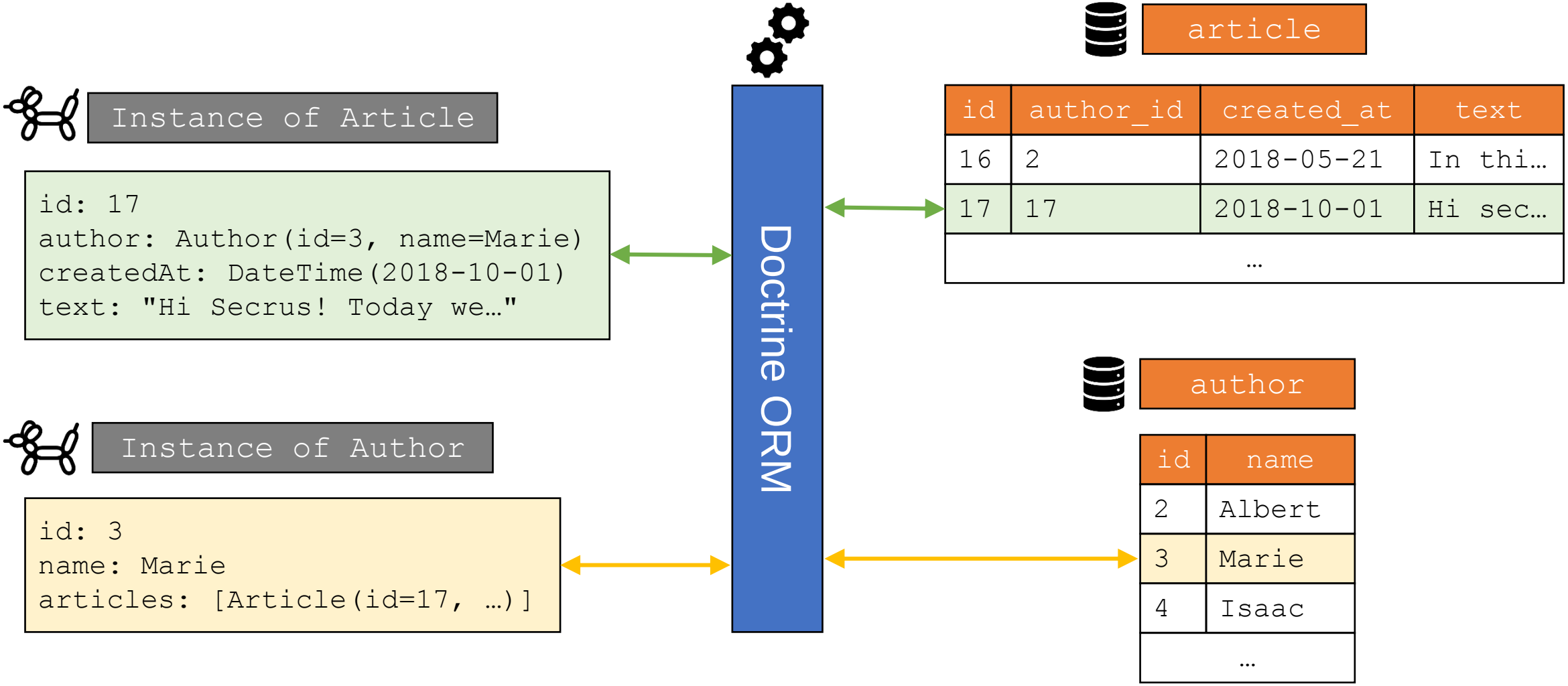


isGranted returns true/false. It delegates decision to AccessDecisionManager::decide.

decide asks all voters to vote and returns final decision.

vote must return one of these values : ACCESS_GRANTED, ACCESS_ABSTAIN or ACCESS_DENIED.

Doctrine is several PHP libraries focused on database storage and object mapping.



Doctrine triggers a bunch of events during the life-time of stored entities: prePersist and postPersist, preUpdate and postUpdate, preRemove and

```php
use Doctrine\ORM\Event\LifecycleEventArgs;
use Doctrine\ORM\Mapping as ORM;

class Article
{
    private $createdAt;
    private $updatedAt;
    private $createdBy;
    private $updatedBy;

    /** @ORM\PrePersist() */
    public function prePersist(LifecycleEventArgs $event)
    {
        $this->createdAt = new \DateTime();
    }

    /** @ORM\PreUpdate() */
    public function preUpdate(LifecycleEventArgs $event)
    {
        $article = $event->getEntity();
        $article->setUpdatedAt(new \DateTime());
    }
}
```

Precondition: $event->getEntity() is an instance of Article

Doctrine listeners as services have access to other services via dependency injection :

```php
use App\Entity\Article;
use Doctrine\ORM\Event\LifecycleEventArgs;

class ArticleLifecycleListener
{
    /** @var App\Entity|Author */
    private $user;

    public function prePersist(LifecycleEventArgs $event)
    {
        $article = $event->getEntity();
        if ($article instanceof Article) {
            $article->setCreatedBy($this->user);
        }
    }

    public function preUpdate(LifecycleEventArgs $event)
    {
        $article = $event->getEntity();
        if ($article instanceof Article) {
            $article->setUpdatedBy($this->user);
        }
    }
}
```

Precondition: $event->getEntity() is an instance of any entity

Interface segregation principle

Many client-specific interfaces are better than one general-purpose interface.
No client should be forced to depend on methods it does not use.

Container manages services which are any useful objects.

`Configuraion: yaml, xml, php`

`Service container`

For each service:

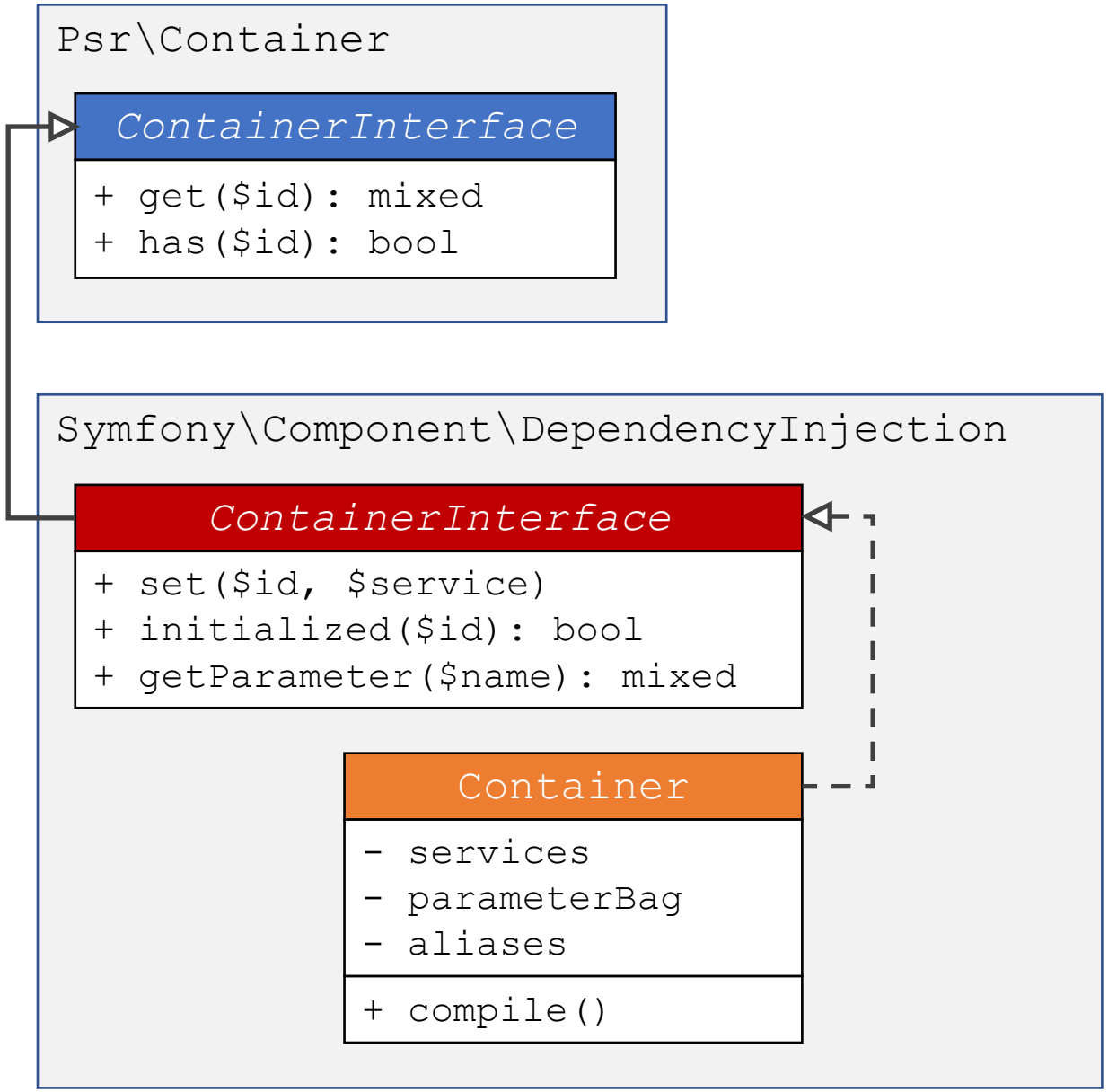
- id
- class name
- dependencies
- …

- Instantiate service object on demand
- Instantiate its dependencies
- Inject dependencies into service
- Store and return the service
- Return the same instance on consecutive demands

Fetch services
(Any client code)

Configure services, parameters
(Extensions)

Compile container (init, cache)
(Kernel)



Routes configuration

```yaml
# config/routes.yaml
blog_list:                                  # route name
    path: /blog/{category}/{page}           # path pattern
    controller: App\Controller\Blog::list # controller to execute
    requirements:                           # param constraints
        category: '[a-z0-9\-_]+'
        page: '\d+'
    defaults:                               # param default values
        page: 1
```

Url matcher

```php
$matcher->match('/blog/symfony-routing/1');
```

```
'_route'      => string 'blog_list'
'_controller' => string 'App\Controller\Blog::list'
'page'        => string '1'
'category'    => string 'symfony-routing'
```

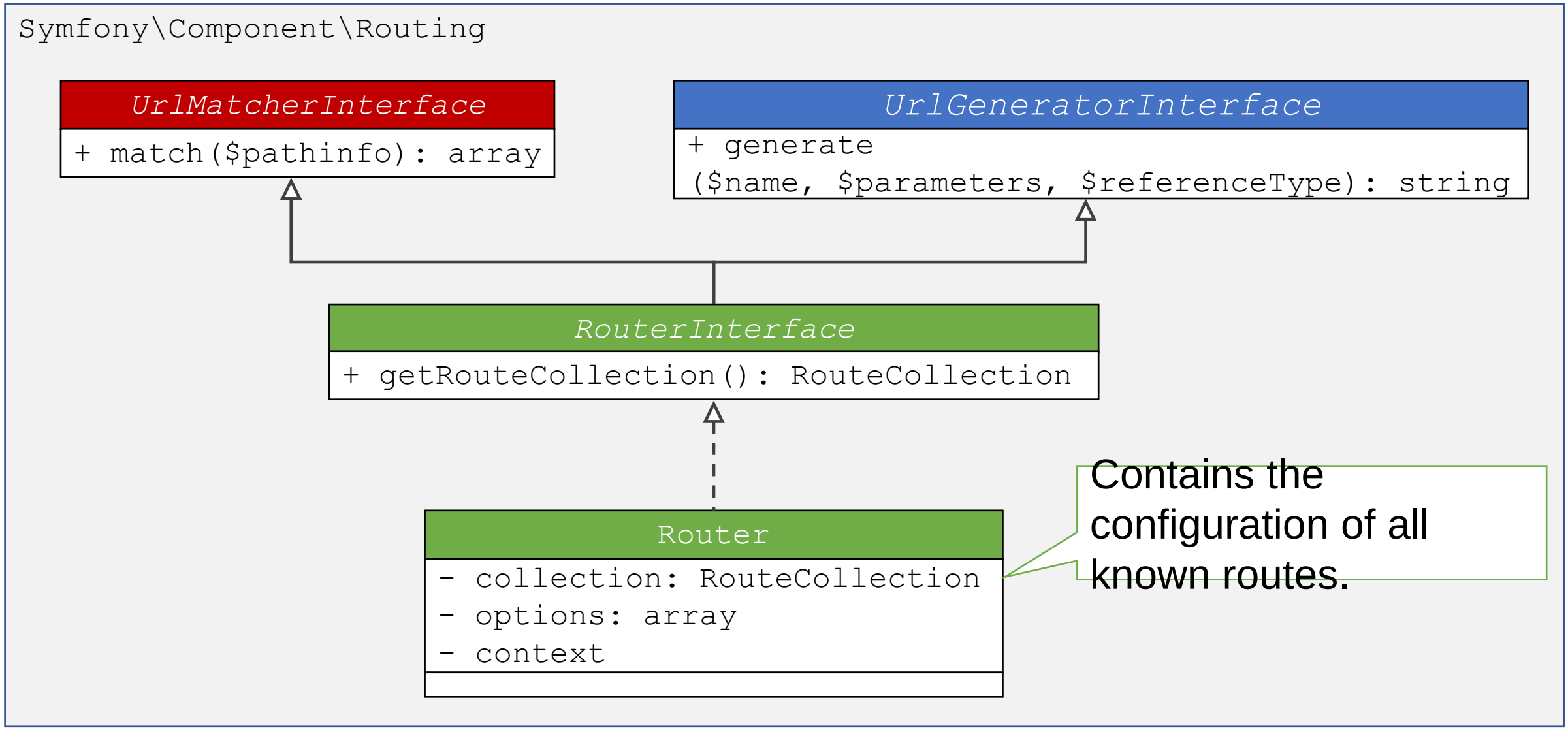
Url generator

```php
$generator->generate('blog_list', ['category' => 'symfony-routing']);
```

```
'/blog/symfony-routing'
```

```php
$generator->generate('blog_list', ['category' => 'secr', 'page' => 2]);
```

```
'/blog/secr/2'
```


Symfony\Component\Routing
UrlMatcherInterface
+ match($pathinfo): array
UrlGeneratorInterface
+ generate
($name, $parameters, $referenceType): string
RouterInterface
+ getRouteCollection(): RouteCollection
Router
- collection: RouteCollection
- options: array
- context
Contains the configuration of all known routes.

Dependency inversion principle

High-level modules should not depend on low-level modules.
Both should depend on abstractions.

Abstractions should not depend on details.
Details should depend on abstractions.

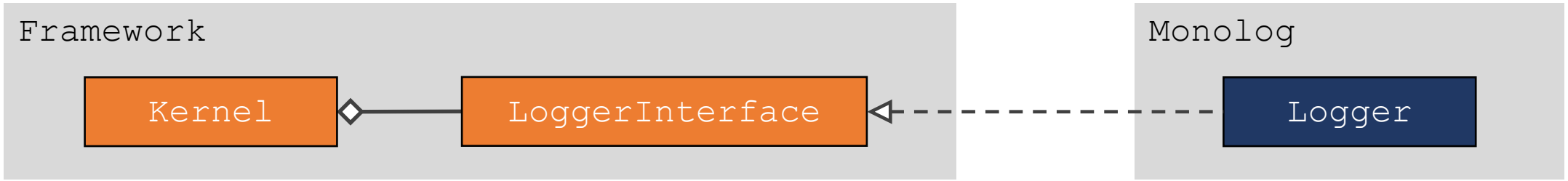
```php
namespace Framework;

use Monolog\Logger;

class Kernel
{
    private $logger;

    public function __construct(Logger $logger)
    {
        $this->logger = $logger;
    }

    public function handle()
    {
        $this->logger->log('...');
        // ...
    }
}
```

```php
namespace Monolog;

class Logger
{
    public function log($message)
    {
        echo $message;
    }
}
```

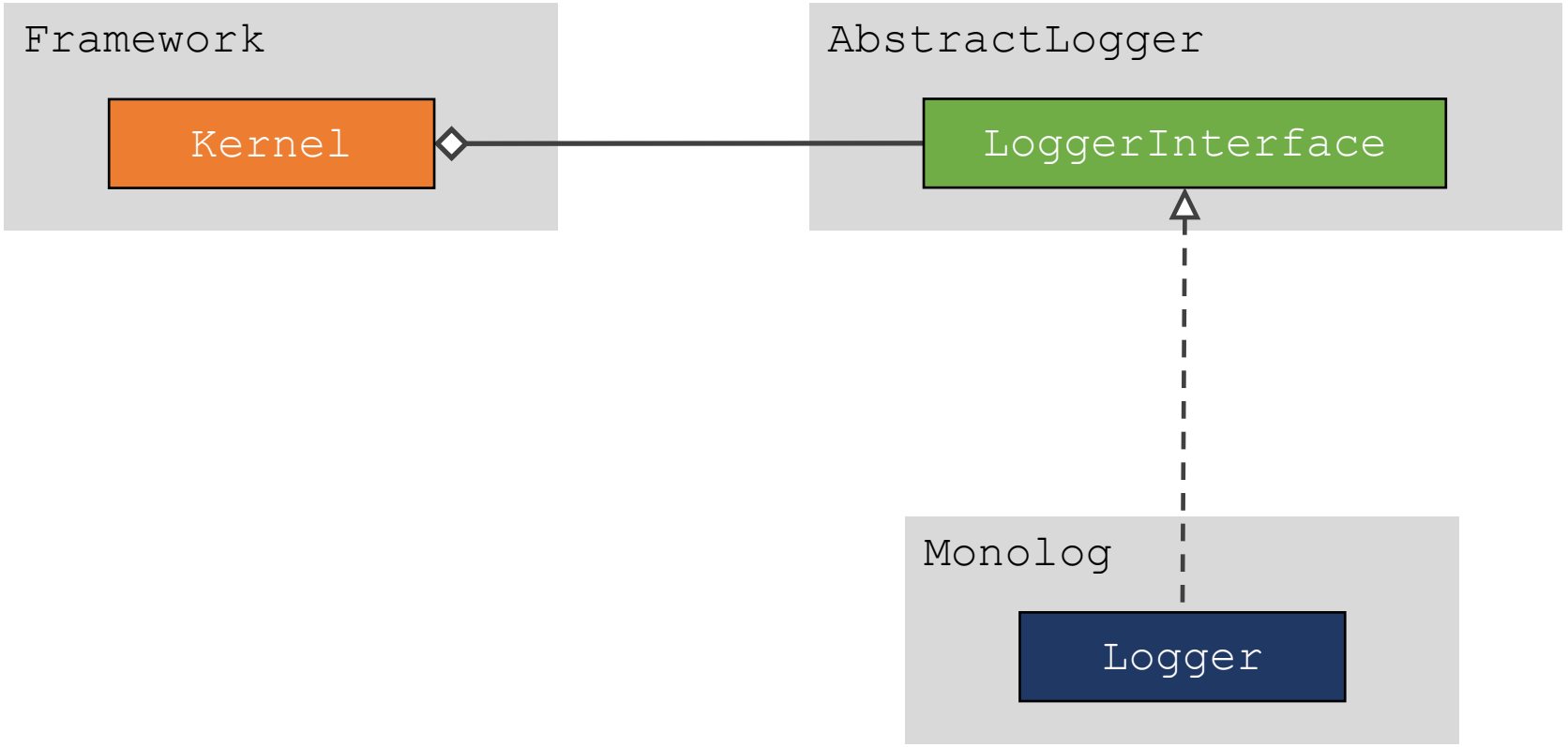
```php
namespace Framework;

class Kernel
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }
}
```

```php
namespace Monolog;

use Framework\LoggerInterface;

class Logger implements LoggerInterface
{
    public function log($message)
    {
        echo $message;
    }
}
```

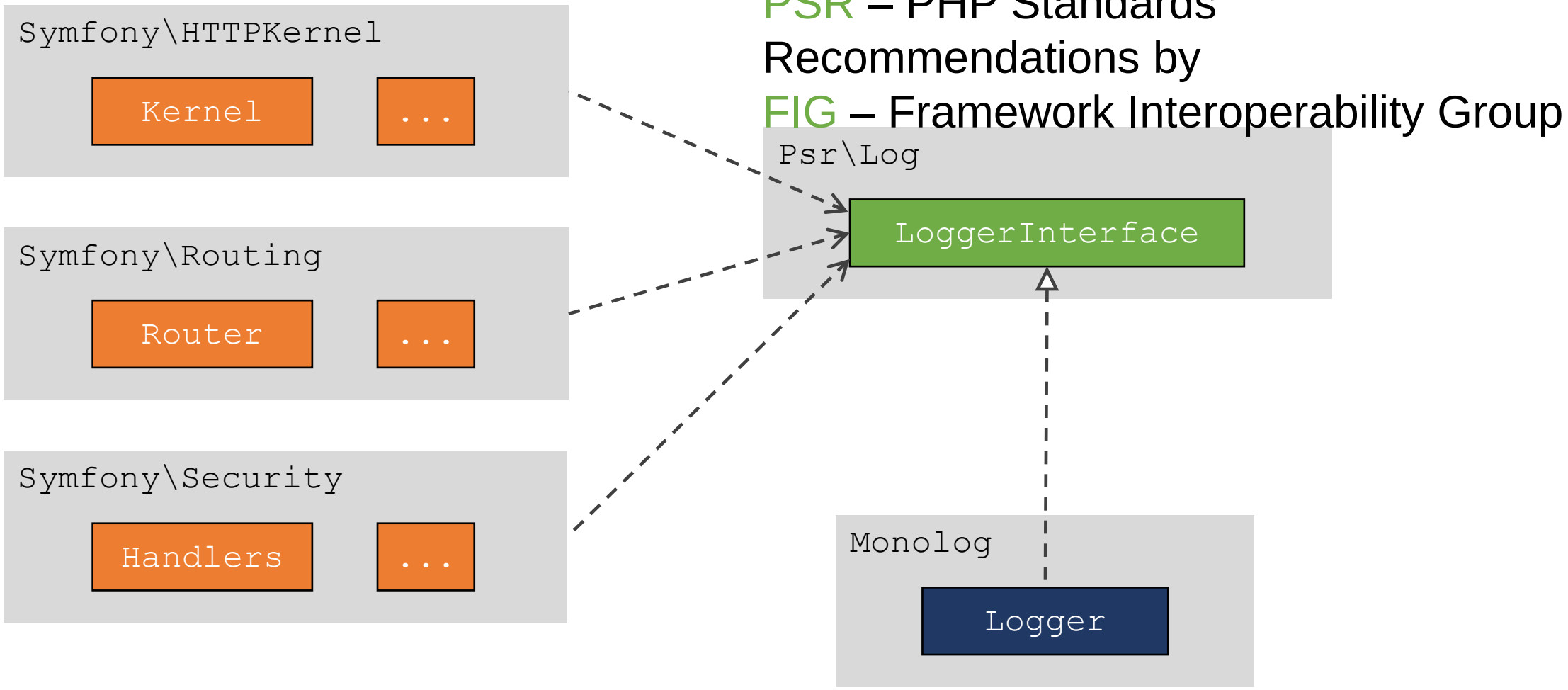
```php
namespace Framework;

interface LoggerInterface
{
    public function log($message);
}
```



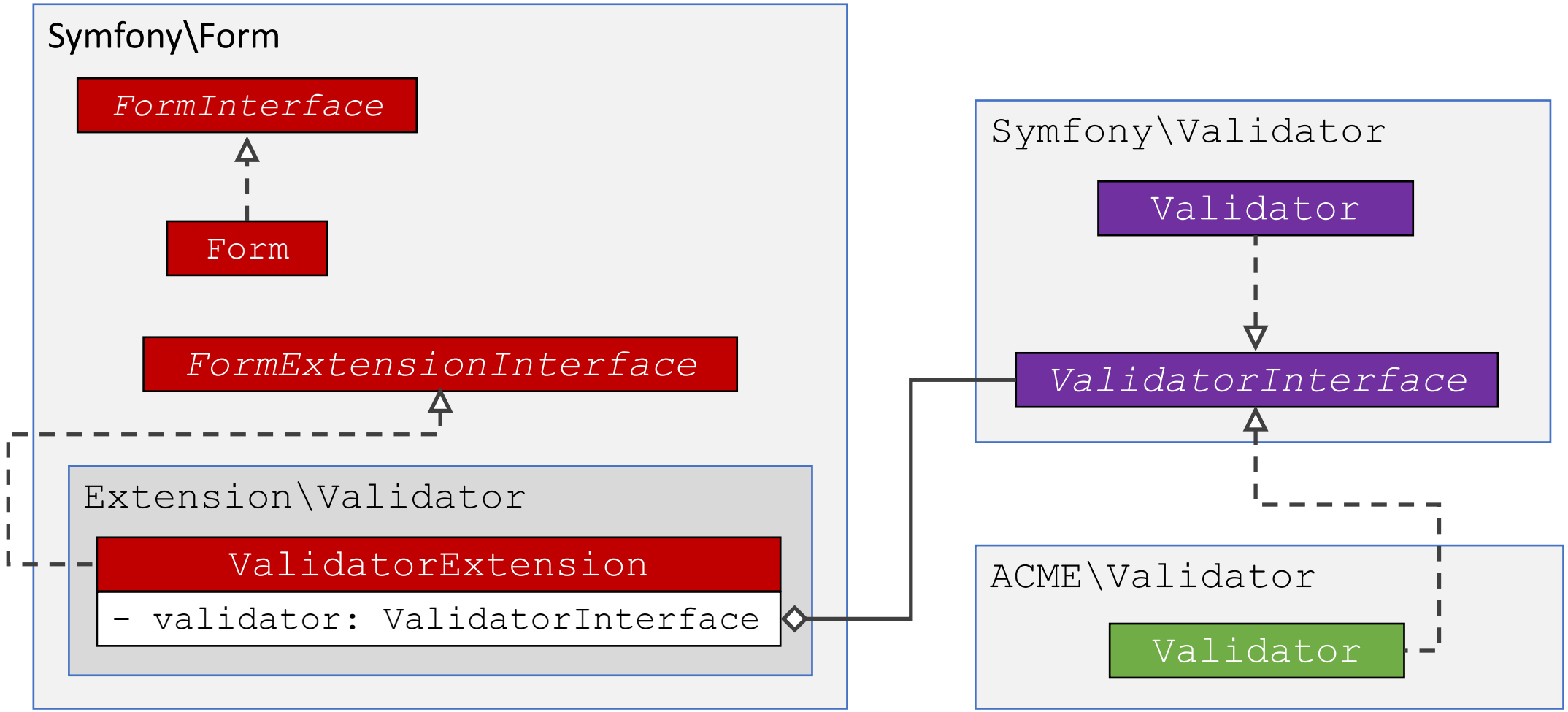
More flexible solution



PSR – PHP Standards Recommendations by FIG – Framework Interoperability Group



Symfony uses interfaces and dependency injection everywhere.



Thank you!

https://vria.eu

contact@vria.eu

https://twitter.com/RiaVlad

https://webnet.fr