7. JSON-RPC Server

We are going to be working on the repository that you have created up to this point. If you skipped ahead, don't worry. You can clone the repository from its Github repository.

```
$ git clone git@github.com:skafandri/symfony-tutorial.git --branch ch6
Cloning into 'symfony-tutorial'...
remote: Counting objects: 737, done.
remote: Total 737 (delta 0), reused 0 (delta 0), pack-reused 737
Receiving objects: 100% (737/737), 418.24 KiB | 66.00 KiB/s, done.
Resolving deltas: 100% (378/378), done.
Checking connectivity... done.
```

We are going to implement a JSON-RPC server to expose some of our application's logic as webservices.
This functionality is clearly not specific to our application.
For this reason, we will implement it as a stand alone bundle.

7.1 JsonRpcBundle

We will start to partially implement JSON-RPC 2.0 specifications. For a full specification reference please check http://www.jsonrpc.org/specification

Let's start by creating a new JsonRpcBundle

- Create **src/JsonRpcBundle/JsonRpcBundle.php**

```php
<?php

namespace JsonRpcBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class JsonRpcBundle extends Bundle
{

}
```

To enable the new bundle, edit **app/AppKernel.php** and add `new JsonRpcBundle\JsonRpcBundle()` to the $bundles array.

- Create **src/JsonRpcBundle/Server.php**

```php
<?php

namespace JsonRpcBundle;

use Symfony\Component\DependencyInjection\ContainerAware;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Exception\UnexpectedValueException;

class Server extends ContainerAware
{

    const ID = 'json_rpc.server';

    public function handle($request, $serviceId)
    {
        $encoder = new JsonEncoder();
        try {
            $request = $encoder->decode($request, JsonEncoder::FORMAT);
        } catch (UnexpectedValueException $exception) {
            return new ErrorResponse(
                ErrorResponse::ERROR_CODE_PARSE_ERROR, 'Invalid JSON'
            );
        }

        $request = $this->resolveOptions($request);

        if (!$this->isAllowed($serviceId, $request['method'])) {
            return new ErrorResponse(
                    ErrorResponse::ERROR_CODE_METHOD_NOT_FOUND,
                    sprintf('%s does not exist', $request['method'])
            );
        }

        $service = $this->container->get($serviceId);
        $result = call_user_func_array(
                array(
                    $service,
                    $request['method']
                ),
                $request['params']
        );

        return new SuccessResponse($request['id'], $result);
    }

    private function isAllowed($serviceId, $method)
    {
        return true;
    }
    private function resolveOptions($request)
    {
        $resolver = new OptionsResolver();
        $resolver
                ->setRequired('id')
                ->setRequired('method')
                ->setRequired('jsonrpc')
                ->setDefault('params', array())
                ->addAllowedValues('jsonrpc', Response::VERSION);
        return $resolver->resolve($request);
    }

}
```

- Create **src/JsonRpcBundle/Resources/config/services.yml**

```
services:
    json_rpc.server:
        class: JsonRpcBundle\Server
        calls:
            - [setContainer, ["@service_container"]]
```

- Create **src/JsonRpcBundle/DependencyInjection/JsonRpcExtension.php**

```
<?php

namespace JsonRpcBundle\DependencyInjection;

use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Extension\Extension;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;

class JsonRpcExtension extends Extension
{
    public function load(array $config, ContainerBuilder $container)
    {
        $loader = new YamlFileLoader(
            $container,
            new FileLocator(__DIR__.'/../Resources/config')
        );
        $loader->load('services.yml');
    }
}
```

- Create **src/JsonRpcBundle/Response.php**

```php
<?php

namespace JsonRpcBundle;

abstract class Response
{

    const VERSION = '2.0';

    private $id;
    private $jsonrpc = self::VERSION;
    private $resultKey;
    private $result;

    public function __construct($id, $resultKey, $result)
    {
        $this->id = $id;
        $this->resultKey = $resultKey;
        $this->result = $result;
    }

    public function getId()
    {
        return $this->id;
    }

    public function getJsonrpc()
    {
        return $this->jsonrpc;
    }

    public function getResultKey()
    {
        return $this->resultKey;
    }

    public function getResult()
    {
        return $this->result;
    }

    public function setId($id)
    {
        $this->id = $id;
        return $this;
    }

    public function setJsonrpc($jsonrpc)
    {
        $this->jsonrpc = $jsonrpc;
        return $this;
    }
    public function setResultKey($resultKey)
    {
        $this->resultKey = $resultKey;
        return $this;
    }

    public function setResult($result)
    {
        $this->result = $result;
        return $this;
    }

    public function toArray()
    {
        return array(
            'id' => $this->getId(),
            'jsonrpc' => $this->getJsonrpc(),
            $this->getResultKey() => $this->getResult()
        );
    }

}
```

- Create **src/JsonRpcBundle/SuccessResponse.php**

```php
<?php

namespace JsonRpcBundle;

class SuccessResponse extends Response
{

    public function __construct($id, $result)
    {
        parent::__construct($id, 'result', $result);
    }

}
```

- Create **src/JsonRpcBundle/ErrorResponse.php**

```php
<?php

namespace JsonRpcBundle;

class ErrorResponse extends Response
{
    const ERROR_CODE_PARSE_ERROR = -32700;
    const ERROR_CODE_METHOD_NOT_FOUND = -32601;
    const ERROR_CODE_SERVER_ERROR = -32000;

    public function __construct($code, $message, $id = null)
    {
        parent::__construct(
          $id,
          'error',
          array('code' => $code, 'message' => $message)
        );
    }

}
```

- Create **src/JsonRpcBundle/Exception/JsonRpcException.php**

```php
<?php

namespace JsonRpcBundle\Exception;

class JsonRpcException extends \Exception
{

}
```

- Create **src/JsonRpcBundle/Exception/InvalidMethodException.php**

```php
<?php

namespace JsonRpcBundle\Exception;

class InvalidMethodException extends JsonRpcException
{

}
```

- Create **src/JsonRpcBundle/Controller/ServerController.php**

```php
<?php

namespace JsonRpcBundle\Controller;

use JsonRpcBundle\Server;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;

class ServerController extends Controller
{

    public function handleAction(Request $request, $service)
    {
        $server = $this->get(Server::ID);
        $result = $server->handle($request->getContent(), $service);
        return new JsonResponse($result->toArray());
    }

}
```

- Create **src/JsonRpcBundle/Resources/config/routing.yml**

```
json_rpc:
    path:     /json-rpc/{service}
    defaults: { _controller: "JsonRpcBundle:Server:handle" }
    methods: [POST]
```

- Edit **app/config/routing.yml** and add the fowlling route

```
json_rpc:
    resource: "@JsonRpcBundle/Resources/config/routing.yml"
```

- Edit **src/AppBundle/Service/WarehouseService.php** update getAll method

```
public function getAll()
{
    return $this->entityManager
            ->createQueryBuilder()
            ->select('warehouse')
            ->from(Warehouse::REPOSITORY, 'warehouse')
            ->getQuery()
            ->getArrayResult();
}
```

Done, you can check if this webservice is working by posting

```
{
  "jsonrpc": "2.0",
  "method": "getAll",
  "id": 1
}
```

to http://127.0.0.1:8000/json-rpc/app.warehouse

You should get a response similar to

```
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [
        {
            "id": 1,
            "name": "warehouse 1",
            "address": null
        },
        {
            "id": 2,
            "name": "warehouse 2",
            "address": null
        }
    ]
}
```

7.2 Service tagging

The bundle we just created will expose any method from any service available in our application.

If you try for example to post

```
{
  "jsonrpc": "2.0",
  "method": "trans",
  "params": ["order.status.20"],
  "id": 1
}
```

to http://127.0.0.1:8000/json-rpc/translator you will get

```
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "delivered"
}
```

Which is not ok. We have to update the method/service filtering that we have it now hard coded to allow everything.

The `isAllowed` method from our server service just returns true. We will update this logic to effectively check if the method is allowed.

- Edit **src**/**JsonRpcBundle**/**Server.php**

Add private $allowedMethods = array();

Update isAllowed method

```
private function isAllowed($serviceId, $method)
{
    return in_array(sprintf('%s->%s', $serviceId, $method), $this->allowedMethods);
}
```

Add addAllowedMethod

```
public function addAllowedMethod($serviceId, $method)
{
    $this->allowedMethods[] = sprintf('%s->%s', $serviceId, $method);
}
```

Now we need a way to gather the list of serviceId/method.

We will define a custom tag [json_rpc.service] and add it to each service method we want to enable.

- Edit **src**/**AppBundle**/**Resources**/**config**/**services.yml** and update app.warehouse definition

```
app.warehouse:
        class: AppBundle\Service\WarehouseService
        parent: app.doctrine_aware
        tags:
            - {name: json_rpc.service, method: getAll}
```

To parse all the services tagged with our custom tag, we will need to write a custom compiler pass.

- Create **src**/**JsonRpcBundle**/**DependencyInjection**/**Compiler**/**ServicePass.php**

```php
<?php

namespace JsonRpcBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class ServicePass implements CompilerPassInterface
{

    public function process(ContainerBuilder $container)
    {
        $serverDefinition = $container->findDefinition(\JsonRpcBundle\Server::ID);
        $resolvers = $container->findTaggedServiceIds('json_rpc.service');

        foreach ($resolvers as $id => $tagAttributes) {
            foreach ($tagAttributes as $attributes) {
                $method = $attributes['method'];
                $serverDefinition->addMethodCall(
                                  'addAllowedMethod',
                                  array($id, $method)
                                );
            }
        }
    }

}
```

Now we need to add the previous compiler pass when building our bundle.

- Edit **src/JsonRpcBundle/JsonRpcBundle.php**

```php
<?php

namespace JsonRpcBundle;

use JsonRpcBundle\DependencyInjection\Compiler\ServicePass;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\HttpKernel\Bundle\Bundle;

class JsonRpcBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);
        $container->addCompilerPass(new ServicePass());
    }

}
```

Done, clear your cache, now only app.warehouse->getAll is accessible as a json-rpc service.

When our application will grow and we will have many services exposed as json-rpc services, we can inspect the methods list by running

```
$ app/console debug:container --tag=json_rpc.service
[container] Public services with tag json_rpc.service
Service ID    method Class name
app.warehouse getAll AppBundle\Service\WarehouseService
```

7.3 Order service

We will create an Order service with only one method for the moment `createOrder`.

When creating an order, we want to do the following:

- Reserve the order products in warehouses
- Send a 'thank you' email to the customer (we will implement this later)
- Log informations about the order creation

An obvious function schema may look like this

```
function createOrder(){``
    logOrderCreate();
    saveToDatabase();
    reserveProductsInWarehosue();
    sendThankYouEmail();
    logOrderCreateFinished();
}
```

This method may have several problems:

- It break the SRP (Single Responsibility Principle) createOrder is doing many things besides creating an order.
- It breaks the Open/closed principle. If we want to change the way we log the order creation for example, we will have to change the createOrder method.
- The order service will need to be coupled with many services (warehouse service, communication service, logging service...)

We will rather implement the createOrder as following

```
function createOrder(){
    dispatchEvent(creating order);
    saveToDatabase();
    dispatchEvent(order created);
}
```

This way, the order creation logic doesn't know anything about logging or customer communication logic. And we can add/remove/change event subscribers without touching the order creation logic. Since event dispatching will be used in many services, we will add this functionality to our parent service.

- Edit **src/AppBundle/Service/AbstractDoctrineAware.php**

```php
<?php

namespace AppBundle\Service;

use Doctrine\Bundle\DoctrineBundle\Registry;
use Doctrine\ORM\EntityManager;
use Symfony\Bridge\Monolog\Logger;
use Symfony\Component\HttpKernel\Debug\TraceableEventDispatcher;

class AbstractDoctrineAware
{

    const ID = 'app.doctrine_aware';

    /**
     *
     * @var Registry
     */
    protected $doctrine;

    /**
     *
     * @var EntityManager
     */
    protected $entityManager;

    /**
     *
     * @var Logger
     */
    protected $logger;

    /**
     *
     * @var TraceableEventDispatcher
     */
    protected $eventDispatcher;

    public function __construct(
        Registry $doctrine,
        Logger $logger,
        TraceableEventDispatcher $eventDispatcher
    )
    {
        $this->doctrine = $doctrine;
        $this->entityManager = $doctrine->getManager();
        $this->logger = $logger;
        $this->eventDispatcher = $eventDispatcher;
    }

}
```

- Create **src/AppBundle/Service/OrderService.php**

```php
<?php
namespace AppBundle\Service;

use AppBundle\Entity\Customer;
use AppBundle\Entity\Order;
use AppBundle\Entity\OrderProductLine;
use AppBundle\Entity\ProductSale;
use AppBundle\Event\Order\OrderAfterCreate;
use AppBundle\Event\Order\OrderBeforeCreate;
use AppBundle\Event\Order\OrderEvent;

class OrderService extends AbstractDoctrineAware
{
    const ID = 'app.order';

    public function createOrder($customerId, $products)
    {
        $this->eventDispatcher->dispatch(
            OrderEvent::BEFORE_CREATE,
            new OrderBeforeCreate($customerId, $products)
            );
        $order = new Order();
        $order->setCustomer(
          $this->doctrine->getRepository(Customer::REPOSITORY)->find($customerId)
        );
        foreach ($products as $product) {
            $this->createProductLine($order, $product['id'], $product['quantity']);
        }
        $this->entityManager->persist($order);
        $this->entityManager->flush();
        $this->eventDispatcher->dispatch(
            OrderEvent::AFTER_CREATE,
            new OrderAfterCreate($order)
        );
        return $order->getId();
    }

    private function createProductLine(Order $order, $productSaleId, $quantity)
    {
        $productLine = new OrderProductLine();
        $productLine->setProductSale(
            $this->doctrine
                 ->getRepository(ProductSale::REPOSITORY)
                 ->find($productSaleId)
        );
        $productLine->setQuantity($quantity);
        $productLine->setOrder($order);
        $order->addProductLine($productLine);
        $this->entityManager->persist($productLine);
    }
}
```

- Create **src**/**AppBundle**/**Event**/**Order**/**OrderEvent.php**

```
<?php

namespace AppBundle\Event\Order;

use Symfony\Component\EventDispatcher\Event;

class OrderEvent extends Event
{
    const BEFORE_CREATE = 'order.before_create';
    const AFTER_CREATE = 'order.after_create';
    const PRODUCTS_RESERVED = 'order.products_reserver';
}
```

- Create **src**/**AppBundle**/**Event**/**Order**/**OrderBeforeCreate.php**

```
<?php

namespace AppBundle\Event\Order;

class OrderBeforeCreate extends OrderEvent
{

    private $customerId;
    private $products;

    public function __construct($customerId, $products)
    {
        $this->customerId = $customerId;
        $this->products = $products;
    }

    public function getCustomerId()
    {
        return $this->customerId;
    }

    public function getProducts()
    {
        return $this->products;
    }

}
```

- Create **src/AppBundle/Event/Order/OrderAfterCreate.php**

```php
<?php

namespace AppBundle\Event\Order;

use AppBundle\Entity\Order;

class OrderAfterCreate extends OrderEvent
{

    /**
     *
     * @var Order
     */
    private $order;

    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    public function getOrder()
    {
        return $this->order;
    }

}
```

- Edit **src/AppBundle/Resources/config/services.yml**

```yaml
imports:
    - { resource: listeners.yml }
    - { resource: events.xml }
services:
    app.doctrine_aware:
        class: AppBundle\Service\AbstractDoctrineAware
        arguments: [@doctrine, @logger, @event_dispatcher]
        abstract: true
    app.warehouse:
        class: AppBundle\Service\WarehouseService
        parent: app.doctrine_aware
        tags:
            - {name: json_rpc.service, method: getAll}
    app.catalog:
        class: AppBundle\Service\CatalogService
        parent: app.doctrine_aware
    app.order:
        class: AppBundle\Service\OrderService
        parent: app.doctrine_aware
        tags:
            - {name: json_rpc.service, method: createOrder}
```

- Create **src**/**AppBundle**/**Resources**/**config**/**listeners.yml**

```
services:
    app.listener.soft_delete:
            class: AppBundle\Event\Listener\SoftDelete
            tags:
                - { name: doctrine.event_listener, event: onFlush }

    app.order_listener:
        class: AppBundle\Event\Listener\OrderListener
        parent: app.doctrine_aware
        calls:
            - [setWarehouseService, ["@app.warehouse"]]
        tags: #each tag should be on a single line, I split it here for readability
            - {name: kernel.event_listener,
              event:%app.order_before_create%,method:onBeforeCreate}
            - {name: kernel.event_listener,
              event: %app.order_after_create%,method:onAfterCreate}
```

- Create **src**/**AppBundle**/**Resources**/**config**/**events.xml**

```
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://symfony.com/schema/dic/services
            http://symfony.com/schema/dic/services/services-1.0.xsd">

    <parameters>
        <parameter key="app.order_before_create"
        type="constant">AppBundle\Event\Order\OrderEvent::BEFORE_CREATE</parameter>
        <parameter key="app.order_after_create"
        type="constant">AppBundle\Event\Order\OrderEvent::AFTER_CREATE</parameter>
    </parameters>
</container>
```

- Create **src/AppBundle/Event/Listener/OrderListener.php**

```php
<?php

namespace AppBundle\Event\Listener;

use AppBundle\Event\Order\OrderAfterCreate;
use AppBundle\Event\Order\OrderBeforeCreate;
use AppBundle\Service\AbstractDoctrineAware;
use AppBundle\Service\WarehouseService;

class OrderListener extends AbstractDoctrineAware
{
    /** @var WarehouseService */
    private $warehouseService;

    public function onBeforeCreate(OrderBeforeCreate $event)
    {
        $this->logger->addInfo(
                'Creating order', array(
            'customerId' => $event->getCustomerId(),
            'products' => $event->getProducts()
        ));
    }

    public function onAfterCreate(OrderAfterCreate $event)
    {
        $this->logger->addInfo(
                'Order created', array('orderId' => $event->getOrder()->getId())
        );
        $this->warehouseService->reserveProducts($event->getOrder());
    }

    public function setWarehouseService(WarehouseService $warehouseService)
    {
        $this->warehouseService = $warehouseService;
        return $this;
    }
}
```

- Edit **src/AppBundle/Service/WarehouseService.php** and add

```
public function reserveProducts(Order $order)
{

}
```

Now, if you post

```
{
  "jsonrpc": "2.0",
  "method": "createOrder",
  "params": {
    "customerId": "1",
    "products": [
      {"id": "1", "quantity":"5"},
      {"id": "2", "quantity":"8"}
    ]
  },
  "id": 1
}
```

To http://127.0.0.1:8000/json-rpc/app.order

If you check **app/logs/dev.log** (or **prod.log** if you run your application in prod mode) you should see similar entries like

```
app.INFO: Creating order {"customerId":"1","products": \
  [{"id":"1","quantity":"5"},{"id":"2","quantity":"8"}]} []
app.INFO: Order created {"orderId":20} []
```

7.4 Service aliases

Sometimes you would like to access a service through another name. In our case, to use the order webservice you should post to http://127.0.0.1:8000/json-rpc/app.order
We would like to provider a more convinient URL to access our webservice.http://127.0.0.1:8000/json-rpc/order would look better. We will do the same with http://127.0.0.1:8000/json-rpc/app.warehouse

Setting aliases is very simple.

- Edit **src/AppBundle/Resources/config/services.yml** and add

```
order:
    alias: app.order

warehouse:
    alias: app.warehouse
```

At the same level with `app.order`

Now, calling `$container->get('app.order')` is equivalent to `$container->get('order')`

We need to update our compiler pass to take in consideration aliases. We will also add a validation to make sure that the methods exposed are callables.

- Edit **src/JsonRpcBundle/DependencyInjection/Compiler/ServicePass.php**

```php
<?php

namespace JsonRpcBundle\DependencyInjection\Compiler;

use JsonRpcBundle\Server;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Definition;
use JsonRpcBundle\Exception\InvalidMethodException;

class ServicePass implements CompilerPassInterface
{

    public function process(ContainerBuilder $container)
    {
        $serverDefinition = $container->findDefinition(Server::ID);
        $resolvers = $container->findTaggedServiceIds('json_rpc.service');
        foreach ($resolvers as $id => $tags) {
            $this->registerMethods($container, $serverDefinition, $id, $tags);
            foreach ($container->getAliases() as $aliasName => $alias) {
                if ($alias->isPublic() && (string) $alias === $id) {
                    $this->registerMethods(
                            $container, $serverDefinition, $aliasName, $tags
                    );
                }
            }
        }
    }

    private function registerMethods(
    ContainerBuilder $container, Definition $server, $serviceId, $tags
    )
    {
        $class = $container->findDefinition($serviceId)->getClass();
        foreach ($tags as $tag) {
            $method = $tag['method'];
            if (!is_callable(array($class, $method))) {
                throw new InvalidMethodException(
                sprintf('%s::%s is not callable', $class, $method)
                );
            }
            $server->addMethodCall('addAllowedMethod', array($serviceId, $method));
        }
    }

}
```

We will update the Server to make the allowed methods data structure more explicit (now we have it as a list of concatinated **serviceId->method**).

- Edit **src/JsonRpcBundle/Server.php**

Update `isAllowed` and `addAllowedMethod` methods

```
private function isAllowed($serviceId, $method)
{
    return array_key_exists($serviceId, $this->allowedMethods) &&
            in_array($method, $this->allowedMethods[$serviceId]);
}

public function addAllowedMethod($serviceId, $method)
{
    if (empty($this->allowedMethods[$serviceId])) {
        $this->allowedMethods[$serviceId] = array();
    }
    $this->allowedMethods[$serviceId][] = $method;
}
```

7.5 Configuration expressions

We will create an EmailService that is capable of using several external email providers to send emails.

As a start, we will have two email providers. PhpProvider that will use PHP's**mail** function, and DevProvider that will just do nothing.

- Create **src/AppBundle/Communication/Email/Message.php**

```php
<?php

namespace AppBundle\Communication\Email;

class Message
{

    private $to;
    private $subject;
    private $message;
    private $additionalHeaders;
    private $additionalParameters;

    public function getTo()
    {
        return $this->to;
    }
    public function getSubject()
    {
        return $this->subject;
    }

    public function getMessage()
    {
        return $this->message;
    }

    public function getAdditionalHeaders()
    {
        return $this->additionalHeaders;
    }

    public function getAdditionalParameters()
    {
        return $this->additionalParameters;
    }

    public function setTo($to)
    {
        $this->to = $to;
        return $this;
    }

    public function setSubject($subject)
    {
        $this->subject = $subject;
        return $this;
    }

    public function setMessage($message)
    {
        $this->message = $message;
        return $this;
    }

    public function setAdditionalHeaders($additionalHeaders)
    {
        $this->additionalHeaders = $additionalHeaders;
        return $this;
    }

    public function setAdditionalParameters($additionalParameters)
    {
        $this->additionalParameters = $additionalParameters;
        return $this;
    }

}
```

- Create **src**/**AppBundle**/**Communication**/**Email**/**ProviderInterface.php**

```php
<?php

namespace AppBundle\Communication\Email;

interface ProviderInterface
{

    public function send(Message $message);
}
```

- Create **src**/**AppBundle**/**Communication**/**Email**/**PhpProvider.php**

```php
<?php

namespace AppBundle\Communication\Email;

class PhpProvider implements ProviderInterface
{

    public function send(Message $message)
    {
        return mail(
                $message->getTo(),
                $message->getSubject(),
                $message->getMessage(),
                $message->getAdditionalHeaders(),
                $message->getAdditionalParameters()
        );
    }

}
```

- Create **src/AppBundle/Communication/Email/DevProvider.php**

```php
<?php

namespace AppBundle\Communication\Email;

class DevProvider implements ProviderInterface
{

    public function send(Message $message)
    {
        return true;
    }

}
```

- Create **src/AppBundle/Service/EmailService.php**

```php
<?php

namespace AppBundle\Service;

use AppBundle\Communication\Email\Message;
use AppBundle\Communication\Email\ProviderInterface;

class EmailService
{

    private $providers = array();
    private $providerIndex = -1;

    public function addProvider(ProviderInterface $provider)
    {
        $this->providers[] = $provider;
    }

    public function send(Message $message)
    {
        $this->incrementIndex();
        $provider = $this->providerIndex[$this->providerIndex];

        return $provider->send($message);
    }

    private function incrementIndex()
    {
        $this->providerIndex++;
        if ($this->providerIndex > count($this->providers) - 1) {
            $this->providerIndex = 0;
        }
    }

}
```

Now we are ready to define the desired services, however we want to add some logic to the service definition. If the application is running in **dev** mode add only DevProvider, otherwise add PhpProvider

- Create **src/AppBundle/Resources/config/email_providers.yml**

```
services:
    app.provider_email_php:
        class: AppBundle\Communication\Email\PhpProvider

    app.provider_email_dev:
        class: AppBundle\Communication\Email\DevProvider
```

- Edit **src/AppBundle/Resources/config/services.yml** and add the following service definition

```
app.email:
        class: AppBundle\Service\EmailService
        calls: #The following should be on one line, I split it here for readability
            - ["addProvider", ["@=service('kernel').getEnvironment() === 'dev' ?
            service('app.provider_email_dev') : service('app.provider_email_php')"]]
```

7.6 Homework

1. If a logger service is available, update the json-rpc service to log the received request and/or the response. Logging the request or the response should be switchable on/off from a configuration. The logger will use a new chanel **json-rpc** that will write to "app/log/json-rpc-{environment}.log"
2. When an order is created, send the follwoing email to the customer: Thank you {client name}, your order number {orderId} is being processed.