8 Advanced logging

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 ch7
Cloning into 'symfony-tutorial'...
remote: Counting objects: 838, done.
remote: Compressing objects: 100% (93/93), done.
remote: Total 838 (delta 27), reused 0 (delta 0), pack-reused 737
Receiving objects: 100% (838/838), 433.96 KiB | 319.00 KiB/s, done.
Resolving deltas: 100% (405/405), done.
Checking connectivity... done.
```

Don't forget to `composer install`

8.1 Logger for JSON-RPC bundle

- Edit **src/JsonRpcBundle/Resources/config/services.yml** and add the following service defintition

```
json_rpc.logger:
        class: JsonRpcBundle\Logger
        arguments: ["@?logger"]
        tags:
            - { name: monolog.logger, channel: json_rpc }
```

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

```php
<?php

namespace JsonRpcBundle;

use Symfony\Bridge\Monolog\Logger as LoggerBridge;

class Logger
{
    const ID = 'json_rpc.logger';

    private $logger;

    public function __construct(LoggerBridge $logger = null)
    {
        if (!$logger) {
            $logger = new LoggerBridge();
        }
        $this->logger = $logger;
    }

    public function getLogger()
    {
        return $this->logger;
    }

}
```

- Update **src/JsonRpcBundle/Controller/ServerController.php** to use the new logger

```php
<?php

namespace JsonRpcBundle\Controller;

use JsonRpcBundle\Logger;
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)
    {
        $requestContent = $request->getContent();
        $logger = $this->get(Logger::ID)->getLogger();
        $logger->addInfo('request', array('content' => $requestContent));
        $server = $this->get(Server::ID);
        $result = $server->handle($requestContent, $service);
        $result = $result->toArray();
        $logger->addInfo('response', $result);
        return new JsonResponse($result);
    }

}
```

Now if we try to make a json-rpc call, post tohttp://127.0.0.1:8000/json-rpc/warehouse

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

We can see in **app/logs/dev.log** (or app/logs/prod.log) the following entries

```
[] json_rpc.INFO: request {"content":"{\n  \"jsonrpc\": \"2.0\", \n  \"method\":\
\"getAll\", \n \"id\": 1\n}"} []

[] json_rpc.INFO: response {"id":1,"jsonrpc":"2.0","result":\[{"id":1,"name":"w1",\
"address":null},{"id":2,"name":"w2","address":null}]} []
```

Note that the logs entries are prepended by **json_rpc** that's the new channel we created.

Now we want to log entries for this channel to a different file, and stop logging them to the main log file.

- Edit **app/config/config_dev.yml**
  - Add `channels: ["!json_rpc"]` to **monolog.handlers.main**
  - And add the following handler to **monolog.handlers**

    ```
    json_rpc:
    type:   stream
    path:   "%kernel.logs_dir%/json-rpc-%kernel.environment%.log"
    level:  debug
    channels: ["json_rpc"]
    ```
- Do the same changes to **app/config/config_prod.yml**.

Now you can find the log entries with **json_rpc** channel in **app/logs/json-rpc-dev.log** and **app/logs/json-rpc-prod.log**

8.2 Log all events

Previously, we created the createOrder method in OrderService that triggers two events. And we created OrderListener that performed some tasks, especially logging those events.

However, the order listener logic should perform actions related to the order flow. Moreover, if we will register more than one listener for a specific event, where should we perform the logging? Logically, in none of them.

One option would be to have a Logging listener service that will listen to all events we want to log, and just perform the logging. This option could work if we have few events, but when the event list will grow, it will be hard to maintain the service definition tags list.

To achieve this in a smooth way, we will define a LoggableEventInterface that we will implement in any event we want to be logged. Then we will override the default event dispatcher service with a custom event dispatcher that supports logging.

- Create **src**/**AppBundle**/**Event**/**LoggableEventInterface.php**

```php
<?php

namespace AppBundle\Event;

interface LoggableEventInterface
{
    public function getLogContext();
}
```

- Create **src**/**AppBundle**/**Event**/**EventDispatcher.php**

```
<?php

namespace AppBundle\Event;

use Symfony\Component\EventDispatcher\ContainerAwareEventDispatcher;
use Symfony\Component\EventDispatcher\Event;

class EventDispatcher extends ContainerAwareEventDispatcher
{
    public function dispatch($eventName, Event $event = null)
    {
        if ($event instanceof LoggableEventInterface) {
            $this->getContainer()
              ->get('logger')
              ->addInfo($eventName, $event->getLogContext());
        }
        return parent::dispatch($eventName, $event);
    }
}
```

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

```
event_dispatcher:
        class: AppBundle\Event\EventDispatcher
        arguments: [@service_container]
```

Our parent servie uses TraceableEventDispatcher, we will update it to make it more independent and uses EventDispatcherInterface

- 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\EventDispatcher\EventDispatcherInterface;

class AbstractDoctrineAware
{
    const ID = 'app.doctrine_aware';

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

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

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

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

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

Everything should be fine now. Let's update the events we created previously to implement LoggableEventInterface

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

```php
<?php

namespace AppBundle\Event\Order;

use AppBundle\Event\LoggableEventInterface;

class OrderBeforeCreate extends OrderEvent implements LoggableEventInterface
{

    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;
    }

    public function getLogContext()
    {
        return array(
            'customerId' => $this->customerId,
            'products' => $this->products
        );
    }

}
```

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

```php
<?php

namespace AppBundle\Event\Order;

use AppBundle\Entity\Order;
use AppBundle\Event\LoggableEventInterface;

class OrderAfterCreate extends OrderEvent implements LoggableEventInterface
{

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

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

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

    public function getLogContext()
    {
        return array('id' => $this->order->getId());
    }

}
```

Done, if you try to call the createOrder method, you should see the events logged

```
app.INFO: order.before_create {"customerId":"1","products":[{"id":"1","quantity":"5"},
{"id":"2","quantity":"8"}]} []
app.INFO: order.after_create {"id":54} []
```

You may notice that we have duplicated logs now

```
app.INFO: order.after_create {"id":54} []
app.INFO: Order created {"orderId":54} []
```

We have to update our OrderListener and remove all loggings from it. Also there is no need anymore that it inherits from AbstractDoctrineAware

- Edit **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\WarehouseService;

class OrderListener
{

    /**
     *
     * @var WarehouseService
     */
    private $warehouseService;

    public function onBeforeCreate(OrderBeforeCreate $event)
    {

    }

    public function onAfterCreate(OrderAfterCreate $event)
    {
        $this->warehouseService->reserveProducts($event->getOrder());
    }

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

}
```

- Edit **src/AppBundle/Resources/config/listeners.yml** and remove `parent: app.doctrine_aware` from `app.order_listener`.

Did you noticed the follwoing entry in the log files?
[2015-06-27 13:17:32] php.INFO: The Symfony\Component\Validator\Constraints\True class is deprecated since version 2.7 and will be removed in 3.0. Use the IsTrue class in the same namespace instead.

To fix it, edit **src/AppBundle/Entity/Category.php** and change `@Assert\True` to `@Assert\IsTrue` in the **isNotSameAsParent**'s annotation.

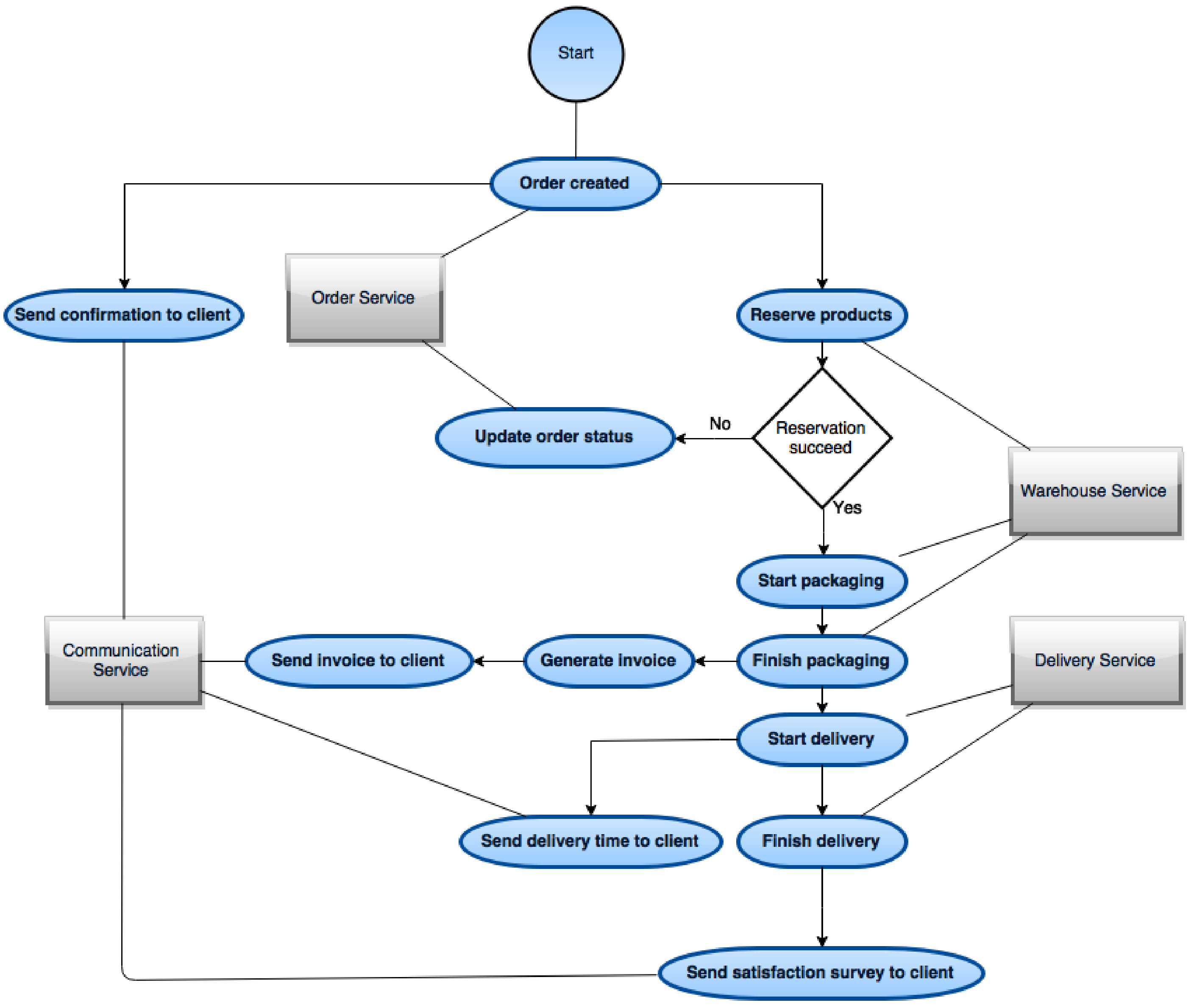
> The fix we just did is very important. If you track any warning or deprecation message and fix it, it will take you few minutes effor once every few weeks or few months.
> You may get some of those warnings every time you upgrade from one minor version to another. However, if you try to update several versions at once (ie: 2.3 to 2.7) you will feel that there is too much to rework and probabaly even withdraw on the upgrade.
> I recommand upgrading to the latest stable release for many reasons
>
> - Your codebase is 6 months older since the last release, and relatively easy to update
> - Few (sometimes none) updates are needed to ajust your code
> - Your application potentially gets better performance (depending on how and what you are using from the framework)

As a real example, for a small application (about 15000 lines) I kept upgrading every minor release from 2.3 to 2.7 (at the time of writing this book). I also applied the same strategy to the used bundles (Doctrine, Monolog...). Each upgrade didn't took more than one hour. I saw teams getting stuck to a LTS (Long Term Support) version which will loose maintainance soon. And upgrading to the next LTS version appears to be a huge refactoring in their application, they are right.

For more details about Symfony release process, visit
https://symfony.com/doc/current/contributing/community/releases.html

8.3 Extended Order flow

We are going to extend the order creation flow as follwoing



We are not going to implement all the logic required to execute this flow right now. As usual, we will implement as less as to see the flow working.

As we are going to have a more detailed flow, we will need more detailed order status.

- Edit **src**/**AppBundle**/**Entity**/**Order.php** and update the status constants block as following

```
const STATUS_NEW = 1;
const STATUS_PROCESSING = 10;
const STATUS_PROCESSING_PRODUCTS_MISSING = 11;
const STATUS_PROCESSING_PRODUCTS_RESERVED = 14;
const STATUS_PROCESSING_PACKAGING = 15;
const STATUS_DELIVERY_STARTED = 20;
const STATUS_DELIVERED = 29;
const STATUS_CANCELLED = 30;
```

- Delete **src**/**AppBundle**/**Event**/**Order**/**OrderAfterCreate.php** since we will have more detailed order events

- Edit **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\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();
        $this->eventDispatcher->dispatch(
            OrderEvent::AFTER_CREATE,
            new OrderEvent($order)
        );
        $this->entityManager->flush();
        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);
    }
}
```

We will refactor OrderBeforeCreate to extend from Event instead of OrderEvent

- Edit **src/AppBundle/Event/Order/OrderBeforeCreate.php**, update the class header

```
namespace AppBundle\Event\Order;

use AppBundle\Event\LoggableEventInterface;
use Symfony\Component\EventDispatcher\Event;

class OrderBeforeCreate extends Event implements LoggableEventInterface
{
```

We will refactor OrderEvent to implement LoggableEventInterface and add new events constants

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

```php
<?php
namespace AppBundle\Event\Order;

use AppBundle\Entity\Order;
use AppBundle\Event\LoggableEventInterface;
use Symfony\Component\EventDispatcher\Event;

class OrderEvent extends Event implements LoggableEventInterface
{
    const BEFORE_CREATE = 'order.before_create';
    const AFTER_CREATE = 'order.after_create';
    const PRODUCTS_RESERVATION_FAILED = 'order.products_reservation_failed';
    const PRODUCTS_RESERVED = 'order.products_reserved';
    const PACKAGING_START = 'order.packaging_start';
    const PACKAGING_END = 'order.packaging_end';
    const INVOICE_GENERATED = 'order.invoice_generated';
    const DELIVERY_START = 'order.delivery_start';
    const DELIVERY_END = 'order.delivery_end';
    /**
     *
     * @var Order
     */
    private $order;

    public function __construct(Order $order)
    {
        $this->order = $order;
    }
    public function getOrder()
    {
        return $this->order;
    }
    public function getLogContext()
    {
        return array('id' => $this->order->getId());
    }
}
```

Let's add the new events to our events parameters list

- Edit **src/AppBundle/Resources/config/events.xml** add the following arguments

```
<parameter key="app.order_reservation_failed" type="constant"
>AppBundle\Event\Order\OrderEvent::PRODUCTS_RESERVATION_FAILED</parameter>
<parameter key="app.order_products_reserved"
type="constant">AppBundle\Event\Order\OrderEvent::PRODUCTS_RESERVED</parameter>
<parameter key="app.order_packaging_start"
type="constant">AppBundle\Event\Order\OrderEvent::PACKAGING_START</parameter>
<parameter key="app.order_packaging_end"
type="constant">AppBundle\Event\Order\OrderEvent::PACKAGING_END</parameter>
<parameter key="app.order_invoice_generated"
type="constant">AppBundle\Event\Order\OrderEvent::INVOICE_GENERATED</parameter>
<parameter key="app.order_delivery_start"
type="constant">AppBundle\Event\Order\OrderEvent::DELIVERY_START</parameter>
<parameter key="app.order_delivery_end"
type="constant">AppBundle\Event\Order\OrderEvent::DELIVERY_END</parameter>
```

We will create a communication service that will handle communication (human, not technical) with the exterior world.

- Create **src/AppBundle/Service/Communication/CommunicationService.php**

```php
<?php

namespace AppBundle\Service\Communication;

class CommunicationService
{

    const ID = 'app.communication';

    private $emailService;

    public function __construct(EmailService $emailService)
    {
        $this->emailService = $emailService;
    }
    public function sendConfirmationEmail($emailAddress, $orderNumber)
    {
    }
    public function sendDeliveryEmail($emailAddress, $orderNumber)
    {
    }
    public function sendInvoice($emailAddress, $orderNumber)
    {
    }
    public function sendSatisfactionSurvey($emailAddress, $orderNumber)
    {
    }

}
```

- Move **src/AppBundle/Service/EmailService.php** to **src/AppBundle/Service/Communication/EmailService.php** and change the namespace to `AppBundle\Service\Communication`

We need a delivery service that will handle delivery related logic.

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

```php
<?php

namespace AppBundle\Service;

use AppBundle\Entity\Order;
use AppBundle\Event\Order\OrderEvent;

class DeliveryService extends AbstractDoctrineAware
{
    const ID = 'app.delivery';

    public function deliverProducts(Order $order)
    {
        $this->eventDispatcher->dispatch(
            OrderEvent::DELIVERY_START, new OrderEvent($order)
        );
        $this->eventDispatcher->dispatch(
            OrderEvent::DELIVERY_END, new OrderEvent($order)
        );
    }

}
```

To generate an invoice, we will create a document service that will be responsible of several documents management in the future.

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

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

use AppBundle\Entity\Order;
use AppBundle\Event\Order\OrderEvent;

class DocumentService extends AbstractDoctrineAware
{
    const ID = 'app.document';
    public function generateInvoice(Order $order)
    {
        $this->eventDispatcher->dispatch(
            OrderEvent::INVOICE_GENERATED, new OrderEvent($order)
        );
    }
}
```

We will update OrderListener to handle the new events

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

```php
<?php

namespace AppBundle\Event\Listener;

use AppBundle\Entity\Order;
use AppBundle\Event\Order\OrderBeforeCreate;
use AppBundle\Event\Order\OrderEvent;
use AppBundle\Service\DeliveryService;
use AppBundle\Service\WarehouseService;

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

    /**
     *
     * @var DeliveryService
     */
    private $deliveryService;

    public function __construct(
        WarehouseService $warehouseService,
        DeliveryService $deliveryService
    )
    {
        $this->warehouseService = $warehouseService;
        $this->deliveryService = $deliveryService;
    }

    public function onBeforeCreate(OrderBeforeCreate $event)
    {

    }

    public function onAfterCreate(OrderEvent $event)
    {
        $this->warehouseService->reserveProducts($event->getOrder());
    }

    public function onReservationFailed(OrderEvent $event)
    {
        $event->getOrder()->setStatus(Order::STATUS_PROCESSING_PRODUCTS_MISSING);
    }
    public function onProductsReserved(OrderEvent $event)
    {
        $event->getOrder()->setStatus(Order::STATUS_PROCESSING_PRODUCTS_RESERVED);
        $this->warehouseService->packageProducts($event->getOrder());
    }

    public function onPackagingStart(OrderEvent $event)
    {
        $event->getOrder()->setStatus(Order::STATUS_PROCESSING_PACKAGING);
    }

    public function onPackagingEnd(OrderEvent $event)
    {
        $this->deliveryService->deliverProducts($event->getOrder());
    }

    public function onDeliveryStart(OrderEvent $event)
    {
        $event->getOrder()->setStatus(Order::STATUS_DELIVERY_STARTED);
    }

    public function onDeliveryEnd(OrderEvent $event)
    {
        $event->getOrder()->setStatus(Order::STATUS_DELIVERED);
    }

}
```

- Edit **src**/**AppBundle**/**Resources**/**config**/**services.yml** to add the new service definitions

```
imports:
    - { resource: listeners.yml }
    - { resource: events.xml }
    - { resource: email_providers.yml }

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}
    order: @app.order
    warehouse: @app.warehouse
    app.email:
        class: AppBundle\Service\Communication\EmailService
        public: false
        calls:
            - ["addProvider", ["@=service('kernel').getEnvironment() === 'dev' ?
            service('app.provider_email_dev') : service('app.provider_email_php')"]]
    app.communication:
        class: AppBundle\Service\Communication\CommunicationService
        arguments: [@app.email]
    event_dispatcher:
        class: AppBundle\Event\EventDispatcher
        arguments: [@service_container]
    app.document:
        class: AppBundle\Service\DocumentService
        parent: app.doctrine_aware
    app.delivery:
        class: AppBundle\Service\DeliveryService
        parent: app.doctrine_aware
```

- Edit **src**/**AppBundle**/**Resources**/**config**/**listeners.yml** to add new tags with the new events

```
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
        arguments: [@app.warehouse, @app.delivery]
        tags: #Put every tag on single line, I split them 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}
            - {name: kernel.event_listener,event:
              %app.order_reservation_failed%,method:onReservationFailed}
            - {name: kernel.event_listener,event:
              %app.order_products_reserved%,method:onProductsReserved}
            - {name: kernel.event_listener,event:
              %app.order_packaging_start%,method:onPackagingStart}
            - {name: kernel.event_listener,event:
              %app.order_packaging_end%,method:onPackagingEnd}
            - {name: kernel.event_listener,event:
              %app.order_delivery_start%,method:onDeliveryStart}
            - {name: kernel.event_listener,event:
              %app.order_delivery_end%,method:onDeliveryEnd}

    app.communication_listener:
        class: AppBundle\Event\Listener\OrderCommunicationListener
        arguments: [@app.communication]
        tags: #Put every tag on single line, I split them here for readability
            - {name: kernel.event_listener,event:
              %app.order_after_create%,method:onAfterCreate}
            - {name: kernel.event_listener,event:
              %app.order_invoice_generated%,method:onInvoiceGenerated}
            - {name: kernel.event_listener,event:
              %app.order_delivery_start%,method:onDeliveryStart}
            - {name: kernel.event_listener,event:
              %app.order_delivery_end%,method:onDeliveryEnd}
```

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

```php
<?php

namespace AppBundle\Service;

use AppBundle\Entity\Order;
use AppBundle\Entity\ProductStock;
use AppBundle\Entity\Warehouse;
use AppBundle\Event\Order\OrderEvent;

class WarehouseService extends AbstractDoctrineAware
{

    const ID = 'app.warehouse';

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

    public function getProductStocks($productId)
    {
        $stocks = $this->entityManager->
                getRepository(ProductStock::REPOSITORY)
                ->findBy(array('product' => $productId));
        if (empty($stocks)) {
            $this->logger->addNotice(
                sprintf('No stocks found for product %s', $productId)
            );
        }

        return $stocks;
    }

    public function reserveProducts(Order $order)
    {
        if ($this->checkProductsStock($order)) {
            $this->doReserveProducts($order);
            $this->eventDispatcher->dispatch(
                    OrderEvent::PRODUCTS_RESERVED, new OrderEvent($order)
            );
        } else {
            $this->eventDispatcher->dispatch(
                    OrderEvent::PRODUCTS_RESERVATION_FAILED, new OrderEvent($order)
            );
        }
    }
    private function doReserveProducts(Order $order)
    {
    }

    private function checkProductsStock(Order $order)
    {
        $quantities = array();
        foreach ($order->getProductLines() as $productLine) {
            $id = $productLine->getProductSale()->getProduct()->getId();
            $quantities[$id] = $productLine->getQuantity();
        }
        $productStocks = $this
                        ->getProductStocksByProductIds(array_keys($quantities))
                        ->execute();
        foreach ($productStocks as $productStock) {
            $id = $productStock->getProduct()->getId();
            if ($quantities[$id] <= $productStock->getQuantity()) {
                unset($quantities[$id]);
            }
        }
        return empty($quantities);
    }

    public function getProductStocksByProductIds(array $productIds)
    {
        return $this->entityManager
                    ->createQueryBuilder()
                    ->select('productStock')
                    ->from(ProductStock::REPOSITORY, 'productStock')
                    ->where('productStock.product in (:products)')
                    ->setParameter('products', $productIds)
                    ->getQuery();
    }

    public function packageProducts(Order $order)
    {
        $this->eventDispatcher->dispatch(
            OrderEvent::PACKAGING_START, new OrderEvent($order)
        );
        $this->eventDispatcher->dispatch(
            OrderEvent::PACKAGING_END, new OrderEvent($order)
        );
    }

}
```

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

```php
<?php

namespace AppBundle\Event\Listener;

use AppBundle\Event\Order\OrderEvent;
use AppBundle\Service\Communication\CommunicationService;

class OrderCommunicationListener
{
    /** @var CommunicationService */
    private $communicationService;

    public function __construct(CommunicationService $communicationService)
    {
        $this->communicationService = $communicationService;
    }
    public function onAfterCreate(OrderEvent $event)
    {
        $emailAddress = $this->getEmailAddress($event);
        $this->communicationService->sendConfirmationEmail(
            $emailAddress, $event->getOrder()->getId()
        );
    }
    public function onInvoiceGenerated(OrderEvent $event)
    {
        $emailAddress = $this->getEmailAddress($event);
        $this->communicationService->sendDeliveryEmail(
            $emailAddress, $event->getOrder()->getId()
        );
    }
    public function onDeliveryStart(OrderEvent $event)
    {
        $emailAddress = $this->getEmailAddress($event);
        $this->communicationService->sendDeliveryEmail(
            $emailAddress, $event->getOrder()->getId()
        );
    }
    public function onDeliveryEnd(OrderEvent $event)
    {
        $emailAddress = $this->getEmailAddress($event);
        $this->communicationService->sendSatisfactionSurvey(
            $emailAddress, $event->getOrder()->getId()
        );
    }
    private function getEmailAddress(OrderEvent $event)
    {
        return $event->getOrder()->getCustomer()->getContact()->getEmail();
    }
}
```

Perfect, if you try to call the createOrder method with valid product sale id and quantity, you will see the following logs

```
app.INFO: order.before_create {"customerId":"1","products":[{"id":"3","quantity":"5"}]} []
app.INFO: order.after_create {"id":84} []
app.INFO: order.products_reserved {"id":84} []
app.INFO: order.packaging_start {"id":84} []
app.INFO: order.packaging_end {"id":84} []
app.INFO: order.delivery_start {"id":84} []
app.INFO: order.delivery_end {"id":84} []
```

8.4 View logs in Kibana

Logging is not just a matter of writing history events to files. With a proper logging setup, we can have a near realtime view on the application behaviour while running on production.

We will install the following setup: update the application to log to an elasticsearch server, setup and run elasticsearch server, install kibana and connect it to the elasticsearch instance.

The following instructions is the minimum required for a working development environment. For more details about installation options and configurations, please visit the online documentation
https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html
https://www.elastic.co/guide/en/kibana/current/index.html

For more download options, please visit https://www.elastic.co/downloads/elasticsearch

Install elasticsearch

```
$ curl -L -O \
https://download.elastic.co/elasticsearch/elasticsearch/elasticsearch-1.6.0.tar.gz
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                               Dload  Upload   Total   Spent    Left  Speed
100 27.0M  100 27.0M    0     0  2294k      0  0:00:12  0:00:12 --:--:-- 4431k
$ tar -xvf elasticsearch-1.6.0.tar.gz
$ cd elasticsearch-1.6.0/bin/
$ ./elasticsearch --node.name logs
[datetime][INFO ][node            ] [logs] version[1.6.0], pid[30511], build...
[datetime][INFO ][node            ] [logs] initializing ...
[datetime][INFO ][plugins         ] [logs] loaded [], sites []
[datetime][INFO ][env             ] [logs] using [1] data paths, mounts [[/ ...
[datetime][INFO ][node            ] [logs] initialized
[datetime][INFO ][node            ] [logs] starting ...
[datetime][INFO ][transport       ] [logs] bound_address
{inet[/0:0:0:0:0:0:0:0:9300]}, publish_address {inet[/192.168.1.143:9300]}
[datetime][INFO ][discovery       ] [logs] elasticsearch/1KtA1thHTGCfh9v7wts6vQ
[datetime][INFO ][cluster.service ] [logs] new_master [logs][1KtA1thHTGCfh9v7wts6vQ]
      [asus][inet[/192.168.1.143:9300]], reason: zen-disco-join (elected_as_master)
[datetime][INFO ][http            ] [logs] bound_address
{inet[/0:0:0:0:0:0:0:0:9200]}, publish_address {inet[/192.168.1.143:9200]}
[datetime][INFO ][node            ] [logs] started
[datetime][INFO ][gateway         ] [logs] recovered [0] indices into cluster_state
```

The server is up and running. Leave it running and let's update our application to send the logs to this instance.

Next, we need to install **ruflin/elastica** bundle, it provides an elasticsearch client.

Add `ruflin/elastica` to composer.json and run `composer update`

- Edit **app/config/config_dev.yml** and **app/config/config_prod.yml** add the following handler to both of them, under **monolog.handlers**

```
elasticsearch:
    type:   elasticsearch
    elasticsearch:
        host: localhost
    level: info
    channels: [app, json_rpc]
```

Now try to run any action that produces logs and watch the elasticsearch console output

```
[datetime][INFO ][cluster.metadata] [logs] [monolog] creating index,
    cause [auto(bulk api)], templates [], shards [5]/[1], mappings [logs]
[datetime][INFO ][cluster.metadata] [logs] [monolog] update_mapping [logs] (dynamic)
[datetime][INFO ][cluster.metadata] [logs] [monolog] update_mapping [logs] (dynamic)
[datetime][INFO ][cluster.metadata] [logs] [monolog] update_mapping [logs] (dynamic)
```

Done, now our application is sending logs to the elasticsearch instance, we need to install kibana to view the logs.
For more download options, please visit https://www.elastic.co/downloads/kibana

```
$ curl -L -O https://download.elastic.co/kibana/kibana/kibana-4.1.0-linux-x64.tar.gz
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10.1M  100 10.1M    0     0  1181k      0  0:00:08  0:00:08 --:--:-- 2001k
$ tar -xvf kibana-4.1.0-linux-x64.tar.gz
$ cd kibana-4.1.0-linux-x64/bin/
$ $ ./kibana
{"name":"Kibana","hostname":"asus","pid":31331,"level":30,"msg":
"No existing kibana index found","time":"2015-06-28T19:31:45.416Z","v":0}
{"name":"Kibana","hostname":"asus","pid":31331,"level":30,"msg":"Listening on
0.0.0.0:5601","time":"2015-06-28T19:31:45.436Z","v":0}
```

Done, visit http://127.0.0.1:5601/ and enter **monolog** in the index pattern, click create. Everything is up and running, click on Discover to see the logs.

8.4 Homework

1. Every time the order status changes, log the change (old status, new status).
2. Update the order.products_reserved event to log the list of reserved products IDs and their quantities.