12 Locking and caching

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
Cloning into 'symfony-tutorial'...
remote: Counting objects: 1020, done.
remote: Total 1020 (delta 0), reused 0 (delta 0), pack-reused 1020
Receiving objects: 100% (1020/1020), 456.82 KiB | 373.00 KiB/s, done.
Resolving deltas: 100% (518/518), done.
Checking connectivity... done.
```

Don't forget to `composer install`

Every section's code is in a separate branch with the section's number. So to get the code for the first section of this chapter, just `git checkout 12.1`

12.1 Critical section lock

When reserving products for an order, the scenarios is simple: check if the requested quantity is available, if yes proceed with the reservation.

In a concurrent environment, the following risky scenario is possible:

1. **Process 1** check if the requested quantity is available (productId: 1, requested: 2, available: 4)
2. **Process 2** check if the requested quantity is available (productId: 1, requested: 3, available: 4)
3. **Process 1** reserves the requested quantity (productId: 1, remaining: 2)
4. **Process 2** reserves the requested quantity (productId: 1, remaining: -1)

Our business logic tells us that a product quantity can be reserved only if the requested quantity is lower or equals to the available stock.

We will implement a simple MySQL lock to achieve this behavior.

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

```
<?php

namespace AppBundle\Service;

use AppBundle\Exception\LockException;
use Symfony\Bridge\Doctrine\RegistryInterface;
use Symfony\Bridge\Monolog\Logger;

class UtilService
{

    const ID = 'app.util';

    /**
     *
     * @var RegistryInterface
     */
    private $registry;

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

    private $prefix;

    function __construct(RegistryInterface $registry, Logger $logger, $prefix = 'app.') {
        $this->registry = $registry;
        $this->logger = $logger;
        $this->prefix = $prefix;
    }
```

```
    public function getLock($lock, $timeout = 10) {
        $lock = $this->prefix.$lock;
        $this->logger->addInfo(sprintf('acquiring named lock "%s"', $lock));
        $connection = $this->registry->getConnection();
        $result = $connection->executeQuery(
            "SELECT GET_LOCK(':lock', :timeout)",
            array('lock' => $lock, 'timeout' => $timeout)
        );
        if (!$result) {
            throw new LockException(sprintf('Cannot get named lock %s', $lock));
        }
        $this->logger->addInfo(sprintf('acquired named lock "%s"', $lock));
    }

    public function releaseLock($lock) {
        $lock = $this->prefix.$lock;
        $connection = $this->registry->getConnection();
        $connection->executeQuery(
                "SELECT RELEASE_LOCK(':lock')", array('lock' => $lock)
        );
        $this->logger->addInfo(sprintf('released named lock "%s"', $lock));
    }

}
```

- Create **src/AppBundle/Exception/LockException.php**

```
<?php

namespace AppBundle\Exception;

class LockException extends AppException
{

}
```

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

Add app.util service definition

```
app.util:
    class: AppBundle\Service\UtilService
    arguments: [@doctrine, @logger]
```

Add `calls` to app.warehouse definition `- ["setUtilService", [@app.util]]`

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

Add a UtilService member and setter

```
/**
 *
 * @var UtilService
 */
private $utilService;

function setUtilService(UtilService $utilService) {
    $this->utilService = $utilService;
}
```

Update reserveProducts method to lock/reserve product/release lock

```
public function reserveProducts(Order $order) {
    $lockName = 'reserve_products';
    $this->utilService->getLock($lockName);
    try {
        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)
            );
        }
    } finally {
        $this->utilService->releaseLock($lockName);
    }
}
```

Done, now only one process can reserve products at once.

12.2 Maximum processes lock

We have got a new business requirement. We should be able to limit the number of sent emails per second. Initially, we will limit the sending to one email per second.

We will keep track of the number of sent emails in a memcached server.

If you don't have memcached server installed or the PHP memcached extension, you can easily install them it by running `sudo apt-get install memcached php5-memcached` (you must restart apache to enable the new extension)

- Edit **app/config/parameters.yml.dist** and add the following parameters

```
memcached.servers:
  - { host: 127.0.0.1, port: 11211 }
maximum_emails_per_second: 1
```

We will implement a generic feature to limit access to given resource to a limited number of processes during a timeframe.

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

Add the service definition

```
memcached:
    class: Memcached
    calls:
        - [ addServers, [ %memcached.servers%
```

Add `@memcached` as third argument to `app.util` arguments.
Add `arguments: [@app.util, %maximum_emails_per_second%]` to `app.email` definition.

- Create **src/AppBundle/Exception/ConcurrentAccessException.php**

```
<?php

namespace AppBundle\Exception;

class ConcurrentAccessException extends AppException
{

}
```

- Edit **src/AppBundle/Service/UtilService.php**
  Add uses `use AppBundle\Exception\ConcurrentAccessException;` and `use Memcached;`
  Add a Memcached member and set it from the constructor argument

```
/**
 *
 * @var Memcached
 */
private $memcached;

function __construct(RegistryInterface $registry,Logger $logger,Memcached $memcached) {
    $this->registry = $registry;
    $this->logger = $logger;
    $this->memcached = $memcached;
}
```

Add the following methods

```
public function enterConcurrentSection($resource, $limit, $timeLimit) {
    $key = sprintf('limit_concurrents.%s', $resource);
    $this->memcached->add($key, 0, $timeLimit);
    if ($this->memcached->increment($key) > $limit) {
        $this->memcached->decrement($key);
        throw new ConcurrentAccessException(
          sprintf(
            'Only %s processes can access %s per %s seconds',
            $limit, $resource, $timeLimit
          )
        );
    }
}

public function exitConcurrentSection($resource) {
    $this->memcached->decrement(sprintf('limit_concurrents.%s', $resource));
}
```

We will update **src/AppBundle/Service/Communication/EmailService.php** to use UtilService to manage the concurrent access.

```php
<?php

namespace AppBundle\Service\Communication;

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

class EmailService
{

    const ID = 'app.email';

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

    /**
     *
     * @var UtilService
     */
    private $utilService;
    private $maxPerSecond;

    function __construct(UtilService $utilService, $maxPerSecond) {
        $this->utilService = $utilService;
        $this->maxPerSecond = $maxPerSecond;
    }

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

    public function send(Message $message) {
        $this->utilService->enterConcurrentSection('email', $this->maxPerSecond, 1);

        $this->incrementIndex();
        $provider = $this->providers[$this->providerIndex];
        $result = $provider->send($message);

        $this->utilService->exitConcurrentSection('email');
        return $result;
    }

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

}
```

12.3 Application benchmarking

We want to enable the catalog service as a webservice and add a getProducts method.

- Edit **src/AppBundle/Resources/config/services.yml**
  Add the following tags to app.catalog definition

```
- {name: json_rpc.service, method: getCategories}
- {name: json_rpc.service, method: getProductSales}
```

Add the alias the service alias `catalog: @app.catalog`.

- Edit **src/AppBundle/Service/CatalogService.php** and add getProductSales method

```
public function getProductSales() {
    return $this->entityManager
        ->createQueryBuilder()
        ->select('product.id, product.code, product.title, product.description, '
                . 'productSale.price, category.label as categoryName, '
                . 'SUM(productStock.quantity) as stock')
        ->from(ProductSale::REPOSITORY, 'productSale')
        ->innerJoin('productSale.product', 'product')
        ->leftJoin('product.category', 'category')
        ->leftJoin(
          ProductStock::REPOSITORY, 'productStock',
          Join::WITH, 'product = productStock.product'
        )
        ->where('productSale.active = true')
        ->andWhere('CURRENT_DATE() BETWEEN productSale.startDate AND productSale.endDate')
        ->groupBy('productSale.id')
        ->getQuery()
        ->getResult();
}
```

Now you can test how this webservice works from a GUI or from a command line script. But we want to see how our application performs under a production load, which certainly won't serve one user at a time.

For this purpose, we are going to use Apache HTTP server benchmarking tool (AB). For the full AB documentation, please visit https://httpd.apache.org/docs/2.4/programs/ab.html

First, install AB using the command `sudo apt-get install apache2-utils`

Create a file named **post** with the content `{"jsonrpc":"2.0","method":"getProductSales","id":1}`

A sample AB command:

```
$ ab -d -S -q -p post -s 600 -T application/json -A 'api_all:pass' -c 1 -n 10 \
http://symfony.local/json-rpc/catalog

This is ApacheBench, Version 2.3 <$Revision: 1528965 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking symfony.local (be patient).....done

Server Software:        Apache/2.4.7
Server Hostname:        symfony.local
Server Port:            80

Document Path:          /json-rpc/catalog
Document Length:        933092 bytes

Concurrency Level:      1
Time taken for tests:   6.376 seconds
Complete requests:      10
Failed requests:        0
Total transferred:      9334180 bytes
Total body sent:        2410
HTML transferred:       9330920 bytes
Requests per second:    1.57 [#/sec] (mean)
Time per request:       637.640 [ms] (mean)
Time per request:       637.640 [ms] (mean, across all concurrent requests)
Transfer rate:          1429.55 [Kbytes/sec] received
                        0.37 kb/s sent
                        1429.92 kb/s total

Connection Times (ms)
              min   avg   max
Connect:        0     0    4
Processing:   584   637  696
Waiting:      576   624  688
Total:        584   637  700
```

We are interested in the application's response time. In the command output, note **Time per request: 637.640 [ms] (mean, across all concurrent requests)**

So our application's average response time is about 600ms for this action.

Not bad.. no? But wait, what means `-c 1 -n 10`? It means to run **10** HTTP requests, **1** at once. Let's increase the number of concurrent connections and see if there is any difference.

```
$ ab -d -S -q -p post -s 600 -T application/json -A 'api_all:pass' -c 200 -n 400\
http://symfony.local/json-rpc/catalog
This is ApacheBench, Version 2.3 <$Revision: 1528965 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking symfony.local (be patient).....done


Server Software:        Apache/2.4.7
Server Hostname:        symfony.local
Server Port:            80

Document Path:          /json-rpc/catalog
Document Length:        933092 bytes

Concurrency Level:      200
Time taken for tests:   378.236 seconds
Complete requests:      400
Failed requests:        1
   (Connect: 0, Receive: 0, Length: 1, Exceptions: 0)
Non-2xx responses:      1
Total transferred:      372434651 bytes
Total body sent:        96400
HTML transferred:       372304203 bytes
Requests per second:    1.06 [#/sec] (mean)
Time per request:       189117.819 [ms] (mean)
Time per request:       945.589 [ms] (mean, across all concurrent requests)
Transfer rate:          961.58 [Kbytes/sec] received
                        0.25 kb/s sent
                        961.83 kb/s total

Connection Times (ms)
              min   avg   max
Connect:        0    37   88
Processing: 67544 170254284994
Waiting:    63744 168164281694
Total:      67628 170290285050
```

Over 900ms. We have to do something..

12.4 Redis cache

In order to enhance the application's performance, we are going to cache the JsonRpc results in Redis.

First, let's install Redis. `sudo apt-get install redis-server`

- Edit **composer.json** add `"snc/redis-bundle": "1.1.9"` and `"predis/predis": "1.0.3"` and `composer update`
- Edit *app/config/config.yml** and add

```
snc_redis:
    clients:
        default:
            type: predis
            alias: default
            dsn: redis://localhost
```

- Edit **src/JsonRpcBundle/Controller/ServerController.php**
  before `$result = $server->handle($requestContent, $service);` insert

  ```
  $cacheKey = base64_encode($service.$requestContent);
  $redis = $this->get('snc_redis.default');
  if($redis->exists($cacheKey)){
    return new JsonResponse(unserialize($redis->get($cacheKey)));
  }
  ```

  after `$result = $result->toArray();` insert `$redis->set($cacheKey, serialize($result));`

Rerun the previous benchmarking and see if there are any performance differences.

12.5 Exercises

1. Update the JsonRpc bundle to let the user decide if the service results should be cached or not.
2. Add an a timeout option for the JsonRpc caching feature.
3. Avoid the cache dogpile effect using two mechanisms
    - The first request that finds that the cache expired will rebuild it while any incoming request will wait.
    - The first request that finds that the cache expired will rebuild it while any incoming request will be served the latest cached data.
      Which technique to use should be configurable per method
4. Gather statistics about cache hits and misses and display them