**Improving the performance of the Spring-Petclinic sample application (part 1 of 5)**

[Julien Dubois](https://blog.ippon.tech/author/jdubois/) Mar 11, 2013 [0](https://blog.ippon.tech/improving-the-performance-of-the-spring-petclinic-sample-application-part-1-of-5/#disqus_thread)

This post is a performance audit of the [“spring-petclinic” sample application](https://github.com/SpringSource/spring-petclinic) provided by VMWare/SpringSource as a demonstration of the latest Spring features and configurations.

We are going to do a step-by-step audit of the application, and show:

* Tools & techniques we usually work with when doing a performance audit
* Common pitfalls and how to avoid them

For doing this audit, we will use our own fork of the project, which is located at: <https://github.com/jdubois/spring-petclinic>

As this tuning is a bit long, we have split it into 5 parts, which we will post every day this week:

* Monday: setting up the tests
* Tuesday: going stateless and Tomcat tuning
* Wednesday: removing the JVM locks
* Thursday: should we stay on JDBC, or is JPA better?
* Friday: adding a cache, and final words

Many thanks to:

* [Pierre Templier](https://twitter.com/ptemplier) and [Christophe Parageaud](http://blog.ippon.fr/author/Christophe_PARAGEAUD/), from Ippon Technologies, for proof-reading this analysis.
* [Michaël Isvy](https://twitter.com/michaelisvy) from SpringSource/VMWare, who is the main developer of the Spring-petclinic application, for helping me and allowing me to torture his application.
* [Thibault Duchateau](https://twitter.com/tduchateau) and [Romain Lespinasse](https://twitter.com/rlespinasse) from the Dandelion Project, for correcting their library at an unbelievable speed.
* [James Ward](https://twitter.com/_JamesWard), the author of WebJars, for best practices in using his library.

**Creating a JMeter test**

First of all, we have configured the application to be in “production” mode:

* We defined the logging level to error [[Source code]](https://github.com/jdubois/spring-petclinic/commit/2c9a37179d7abe3aa057fee200966a2412e4c2e8)
* We have configured the application to use MySQL (as in this project HSQLDB runs in the same JVM as the application, and thus will distort profiling) [[Source code]](https://github.com/jdubois/spring-petclinic/commit/58ce7d6d794c69696d23d62616221a4708d13455)

We’ve created a JMeter test, which is available [here](https://github.com/jdubois/spring-petclinic/commit/69e55e406db37a386ff8348a5a84343801169f85).

We have run this test a first time, with the “POST new visit” step activated, in order to create 10 visits per pet. This means running this test with 10 threads, doing 13 loops each (as there are 13 pets). The “POST new visit” step was then disabled: the application does not paginate visits, so doing inserts will just break it down really quickly.

As visits are the main objects that are being created in the application, they should have been paginated, or there should be at least a limit on the dataset that is returned. On the other hand, it would be really strange that a user has a pet that goes 1000 times to the vet, so this would not be a realistic test case.

That’s why we have decided to limit this test to 10 visits per pet: this is important, as this will have an impact in the future tests.

We have have then run the test normally (using “*mvn clean tomcat7:run*“), with a 2011 Macbook Pro, with 500 concurrent users each doing the test 10 times.

It is also important to mention that:

* Everything is on the same computer: Tomcat, MySQL, JMeter… This is not a “production” architecture but it’s easier to diagnose/debug it that way
* We use Tomcat 7, with Java 1.6 and 128M of RAM
* We have been careful in doing a “*mvn clean*” in the process, in order to remove older HTTP session states
* We do a “warm up” of the application before launching the tests, so the JIT compiler has the time to optimize the application

We have also run those tests on several other computers, with the same results (proportionally, some having more CPU power than the others).

The first results are as follow: the application quickly goes up to **285 requests per second (req/sec)**, then slows down and throws “**java.lang.OutOfMemoryError: Java heap space**” errors after nearly 700 users. We then have lots of HTTP errors, and the application gets so slow it is not usable anymore.

Of course, we can give more memory to the application, so that it can handle our 5000 users. With 1 Gigabytes of heap, the application can handle the load, with a result of **548 req/sec and 0,2% of HTTP errors**.

**Conclusion of part 1**

The application cannot handle a lot of load: with 128 MB of RAM, the application starts to **fail after 600 simultaneous users**, which is a rather small number. Of course, we can give it more RAM to push that limit: we will be able to have more users, but as the GC will have more work, it is likely that the application does not perform very well.

We will see tomorrow, in the next episode of this series, how we started to solve that problem.

You can find the other episodes of this series here : [part 2](http://blog.ippon.fr/?p=7500), [part 3](http://blog.ippon.fr/?p=7512), [part 4](http://blog.ippon.fr/?p=7520) and [part 5](http://blog.ippon.fr/?p=7527).

**Improving the performance of the Spring-Petclinic sample application (part 2 of 5)**

[Julien Dubois](https://blog.ippon.tech/author/jdubois/) Mar 12, 2013 [0](https://blog.ippon.tech/improving-the-performance-of-the-spring-petclinic-sample-application-part-2-of-5/#disqus_thread)

This is part 2 of our 5-part series on improving the performance of the Spring-petclinic application. You can find the first part [here](http://blog.ippon.fr/2013/03/11/improving-the-performance-of-the-spring-petclinic-sample-application-part-1-of-5/).

**Let’s profile our application**

The error from part 1 is quite clear: we fill up all the server memory until the application slows down and crashes.

This is time to launch our profiler! Our two favorite tools are:

* [JProfiler](http://www.ej-technologies.com/products/jprofiler/overview.html), which is the most complete and is a little bit expensive. This is the one we usually recommend to our clients.
* [YourKit](http://www.yourkit.com/), which is easier to use and is less expensive. This is the one we use for Tatami, as they provide free licenses for Open Source projects. If you have never used a profiler before, we recommend that you start with Yourkit.

We have used YourKit for this profiling session, mainly because we found its screenshots look better.

As we can see from this first screenshot, we have found our first culprit: [Dandelion](http://dandelion.github.com/datatables/) (which is a tag library used to display nice-looking HTML tables) is using most of our memory.  
Graphical user interface, text, application, email

Description automatically generated

Dandelion is a great project, but it is using too much memory on this version. As we would love to use Dandelion again, [we have filled a bug](https://github.com/dandelion/issues/issues/34) on the project’s website, and the project’s developpers have been very quick at resolving it!

So **the next version of Dandelion doesn’t have this problem anymore**, and you can safely use it on high-volume applications.

**Solving the memory issue with Dandelion**

Of course we will upgrade to the next version of Dandelion, which will resolve this issue, but for the moment, as we need to move forward, we will replace it with a classic HTML table, which is then beautified using JavaScript. We have used [JQuery DataTables](http://www.datatables.net/), which provides a similar, but pure-JavaScript, solution:

[[Source code]](https://github.com/jdubois/spring-petclinic/commit/b2d628354f327006650506bd531213bd7f3130a2)

We then ran our tests again: we now **go up to 560 req/sec and then down again**… The application now **fails at 3000-4000 users**. We have just pushed our memory limit further, but as soon as the heap space is filled up, the whole application starts to fail again.

This is already a big improvement, but it looks like we still have a memory problem. Let’s fire YourKit again:

Graphical user interface, text, application, email

Description automatically generated

The heap memory is mostly used by “*org.apache.catalina.session.StandardManager*”, which is Tomcat’s class that manages HTTP sessions. This means the HTTP sessions are using all the free heap space, until the JVM cannot handle connections anymore.

**Going stateless**

We have fallen into a classical pitfall in Web application design: using stateful data prevents the application from scaling up.

On our application, it is rather easy to become stateless:

[[Source code]](https://github.com/jdubois/spring-petclinic/commit/32b14575c085c85ab4f2d0c3922a82cfd186bb52)

For this project, going stateless is mostly a matter of reloading data from the database instead of using the HTTP session as a kind of cache. Of course things are not always that easy, for instance when you manage “conversations” in a business application. Our goal is to lower the amount of data stored in the user’s HTTP session, as it is one of main scalability issues we encounter: we are here rather lucky, as we can remove all of this data.

Once those modifications have been done, let’s launch our stress test again: YourKit confirms that we do not use the HTTP Session anymore, and we can now handle the load without any problem. The biggest object in memory is now “*org.sonatype.aether.util.DefaultRepositoryCache*”, which comes from Maven (remember that we launch the application with “*mvn clean tomcat7:run*”).

The application can now handle our 500 threads doing 10 loops for the first time. However, our results are not perfect: we can serve **532 req/sec, but we still have 0.70% HTTP errors**.

This result is a little bit slower than what we got during the previous step (we reached 560 req/sec before failing), as we now read more data from the database, instead of using the HTTP Session as a cache.

**Tuning Tomcat**

We have HTTP errors, which are distributed on all pages: this is a classical problem with Tomcat, which is using blocking IO by default. Let’s use the new Tomcat NIO connector:

[[Source code]](https://github.com/jdubois/spring-petclinic/commit/3388f317f46bfc2a0c9e6c42a7d84a4aa90a197f)

(Many thanks to [Olivier Lamy](https://twitter.com/olamy/status/306140401901899778) for this configuration, which was not explained in the official documentation!).

Now, **we have no HTTP error at all, and we are able to handle 867 req/sec**.

**Conclusion of part 2**

Now the application is starting to work! We can handle our 5000 users without any error at all, and the performance is rather good, at 867 req/sec.

On part 3, we will see that we can do even better.

[edit]

You can find the other episodes of this series here : [part 1](http://blog.ippon.fr/?p=7496), [part 3](http://blog.ippon.fr/?p=7512), [part 4](http://blog.ippon.fr/?p=7520) and [part 5](http://blog.ippon.fr/?p=7527).

Share this article:

This is part 3 of our 5-part series on improving the performance of the Spring-petclinic application. Here are [part 1](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-1-of-5/) and [part 2](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-2-of-5/).

Yesterday we managed to make the application handle 5000 users, with an average performance of 867 req/sec. Let’s see if we can improve this today.

**Removing the JVM locks**

Running the tests from part 2 showed us that some requests are taking a lot longer than the others, in fact our application does not answer requests fairly to all users.

As this is rather strange, we run up YourKit again, this time to check if we have some locked threads in the JVM:

Graphical user interface, application, table, Excel

Description automatically generated

This YourKit screenshot shows a profiling session, where each time a thread is locked it becomes red.

It seems we have a few locks! This explains why some requests are really slow, they are getting locked by the JVM.

YourKit also shows us 3 main culprits for those locks, which we are going to remove one at a time.

**Removing Commons DBCP**

The first cause of those locks is our database connection pool, Commons DBCP, which is well-known for doing a lot of synchronization.

Let’s switch to tomcat-jdbc, which is the new Tomcat connection pool :

[[Source code]](https://github.com/jdubois/spring-petclinic/commit/bb1b399771fe6748294ec410136aebbdbb327d3f)

We have limited the number of connections to 8, which is a very small number, for two reasons:

* This is the default configuration of Commons DBCP, and we wanted to have a comparable configuration
* For this test, everything runs on the same machine, so we can’t have too many concurrent connections, or the performance will in fact decrease

On a real production system, it is very likely that a number between 30 and 100 would be a better configuration.

Testing the application again, **we now achieve 910 req/sec**.

**Removing Webjars**

[Webjars](http://www.webjars.org/) is a library that is used to manage client-side libraries like JQuery or Twitter Bootstrap.

We have included a request to get JQuery in the stress test, in fact we expected this to cause some trouble: Webjars is using the same mechanism as Richfaces to provide static resources, and we already had that exact same issue with Richfaces (by the way, many other Web frameworks, like Play!, are doing the same). Indeed, the JVM is doing a big lock each time someone tries to load a Web library.

We have experienced this being an issue at a client site: we are only testing here with one Web library, but if we were using all the libraries from the application, things would be much worse.

For this particular client:

* We had a few hundred users, not enough to bother setting up a specific system (like a CDN) to handle static resources
* Every user was using the application through HTTPS, which minimized browser caching

After discussing with James Ward, the author of Webjars, we have the following best practices for using Webjars:

* Use a caching HTTP Server as a front-end (or a CDN for a bigger Web application)
* Use a framework that automatically caches Web resources. Unfortunately Spring doesn’t do it by default, But it shouldn’t be too hard to add [using an interceptor](http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/servlet/mvc/WebContentInterceptor.html)

We have decided to simulate that we have a specific caching mechanism (of course, another solution is to remove Webjars and manage your Web libraries manually):

[[Source code]](https://github.com/jdubois/spring-petclinic/commit/2b3ed81a9b294587573cf0bd43402264355dd15b)

For our next steps, we have now disabled the “JS” step in our JMeter test, and have enabled the “JS no webjar” step instead. This will make our stress test use the JQuery script that is served directly by the server, without using Webjars.

So let’s get down to the results: **we are now at 942 req/sec**.

**Removing the monitoring aspect**

This last issue was created on purpose: we have a small lock in the aspect which is provided to monitor the application with JMX.

It’s in fact a very good idea, but it has a negative impact on performance. Let’s remove it:

[[Source code]](https://github.com/jdubois/spring-petclinic/commit/197888fef0ad5066006f817c801c99f57e44103d)

**We now reach 959 req/sec**.

**Conclusion of part 3**

We have used our profiler to quickly see that we had some JVM locks on the application. We have removed each of them, and saw each time an increase in the performance of the application.

Using YourKit again, we don’t see any JVM locks any more on the application (no red threads anymore!):

Graphical user interface, application, table

Description automatically generated

Our application **went up from 867 req/sec to 959 req/sec**. This is of course a better result, and it would probably be even better on a real production server, which has more cores and threads than the Macbook we are using for the tests.

On part 4, we will test if we can do better with JPA than with JDBC.

[edit]

You can find the other episodes of this series here : [part 1](http://blog.ippon.fr/?p=7496), [part 2](http://blog.ippon.fr/?p=7500), [part 4](http://blog.ippon.fr/?p=7520) and [part 5](http://blog.ippon.fr/?p=7527).

**Improving the performance of the Spring-Petclinic sample application (part 4 of 5)**

[Julien Dubois](https://blog.ippon.tech/author/jdubois/) Mar 14, 2013 [0](https://blog.ippon.tech/improving-the-performance-of-the-spring-petclinic-sample-application-part-4-of-5/#disqus_thread)

This is part 4 of our 5-part series on improving the performance of the Spring-petclinic application. Here are [part 1](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-1-of-5/), [part 2](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-2-of-5/) and [part 3](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-3-of-5/).

**Testing the other persistence mechanisms: is JDBC really faster?**

The good thing with the Spring-petclinic application is that we can easily switch between different persistence solutions: JDBC, JPA and Spring Data JPA.

**First tests: JDBC wins!**

We are first doing a very simple test, by switching the Spring profile in the web.xml file and comparing the results.

Here are the results:

* JDBC: 959 req/sec
* JPA: 902 req/sec
* Spring Data JPA: 797 req/sec

One of the reasons of the difference between the JPA and the Spring Data JPA tests is that there is a cache on the [JpaVetRepositoryImpl](https://github.com/jdubois/spring-petclinic/blob/197888fef0ad5066006f817c801c99f57e44103d/src/main/java/org/springframework/samples/petclinic/repository/jpa/JpaVetRepositoryImpl.java#L44). and not on the [SpringDataVetRepository](https://github.com/jdubois/spring-petclinic/blob/197888fef0ad5066006f817c801c99f57e44103d/src/main/java/org/springframework/samples/petclinic/repository/springdatajpa/SpringDataVetRepository.java#L28). So this is rather a non-issue: both repositories have roughly the same performance, and we will generally recommend using Spring Data over JPA as there is a lot less code to write. However, for the rest of the tests we will use JPA as this cache gives us better performance.

**Using Lazy loading and Open Session In View**

The previous tests might have given some people the impression that JPA is not a good technology for performance tuning an application.  
This is just plain wrong: JPA is an excellent solution, as long as you use it correctly.

For instance, the biggest problem here is that we have a collection of visits which is eagerly fetched:

@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.EAGER)

[[Source code]](https://github.com/jdubois/spring-petclinic/blob/197888fef0ad5066006f817c801c99f57e44103d/src/main/java/org/springframework/samples/petclinic/model/Pet.java#L51)

Let’s discuss this problem:

* As this is a sample application, demonstrating that we can switch persistence solutions easily, we are fetching all data in the repository layer. So this decision is understandable.
* However, that means that each time we show the “owners” page, we fetch all the visits, for no reason.

In order to solve this issue, we have decided to go back to normal and use lazy-loading (which is the default in Hibernate/JPA). As this will cause lazy loading exceptions in the visits page, we have to find a solution, and there are two of them:

* Create a specific business method that can return the visits eagerly when needed. This is the most performant solution, but it requires recoding a lot of things.
* Use the “Open Session In View” pattern

You will find a lot of people on the Internet telling that the “Open Session In View” is a bad pattern: indeed, it will make your database transaction live a lot longer than needed. Instead of just using a transaction at the service layer, you will now use it as long as your whole JSP view is not completely generated.  
For example, if you configure the Open Session In View filter on your whole application (on the “/\*” pattern), and run our stress test again, you will have some transactions errors.

However, we only need it on the owners page, so we are going to configure it on this page only:

|  |
| --- |
| <filter-mapping> |
|  |

|  |
| --- |
| <filter-name>openEntityManagerFilter</filter-name> |
|  |

|  |
| --- |
| <url-pattern>/owners/\*</url-pattern> |
|  |

</filter-mapping>

[[Source code]](https://github.com/jdubois/spring-petclinic/blob/681026758d2f80082c4597ec0393bff07c95be65/src/main/webapp/WEB-INF/web.xml#L78)

I have personally used this pattern on a lot of production applications, and it really eases development, for a small performance cost. So unless you have some very big performance needs, you can use it without worrying too much.

The resulting code is rather simple to write:

@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet")

[[Source code]](https://github.com/jdubois/spring-petclinic/commit/681026758d2f80082c4597ec0393bff07c95be65)

And **here are the results: 1066 req/sec** ! So using JPA with the Open Session In View is not such a bad solution after all!

**Conclusion of part 4**

Switching to JPA for our persistence layer has given us a performance boost, as soon as we have tuned it, of course! Besides, we have room for improvement: more tuning can probably be done, and the Open Session In View is not a good performance pattern (but it is an excellent productivity pattern!).

But most importantly, switching to JPA (or, even better, Spring Data JPA) has given us an excellent developer productivity and code quality boost.

We will finish this series with part 5, where we will add more cache to the application, to make it even more performant.

[edit]

You can find the other episodes of this series here : [part 1](http://blog.ippon.fr/?p=7496), [part 2](http://blog.ippon.fr/?p=7500), [part 3](http://blog.ippon.fr/?p=7512), and [part 5](http://blog.ippon.fr/?p=7527).

**Petclinic sample application (part 5 of 5)**

[Julien Dubois](https://blog.ippon.tech/author/jdubois/) Mar 15, 2013 [0](https://blog.ippon.tech/improving-the-performance-of-the-spring-petclinic-sample-application-part-5-of-5/#disqus_thread)

This is part 5 of our 5-part series on improving the performance of the Spring-petclinic application. Here are [part 1](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-1-of-5/), [part 2](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-2-of-5/), [part 3](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-3-of-5/) and [part 4](https://test-ippon.ghost.io/improving-the-performance-of-the-spring-petclinic-sample-application-part-4-of-5/).

**Adding more cache**

To increase application performance, one of the classic solutions is to add more cache. We already have a cache configured in the application, it is on the [JpaVetRepositoryImpl](https://github.com/jdubois/spring-petclinic/blob/681026758d2f80082c4597ec0393bff07c95be65/src/main/java/org/springframework/samples/petclinic/repository/jpa/JpaVetRepositoryImpl.java#L44).

We are going to generalize this cache on 2 different parts of the application:

* Using the [Spring Cache abstraction](http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/cache.html), like on the [JpaVetRepositoryImpl](https://github.com/jdubois/spring-petclinic/blob/681026758d2f80082c4597ec0393bff07c95be65/src/main/java/org/springframework/samples/petclinic/repository/jpa/JpaVetRepositoryImpl.java#L44), to cache frequently used business methods. The [JpaOwnerRepositoryImpl.findByLastName()](https://github.com/jdubois/spring-petclinic/blob/05a60b774011c41d522293ec3b2f4ed89b680ec5/src/main/java/org/springframework/samples/petclinic/repository/jpa/JpaOwnerRepositoryImpl.java#L46) method is a perfect candidate for this: doing a “like” clause is very slow on a database, and people’s last names do not change frequently. So we decided to cache this result for 60 seconds.
* Using the Hibernate second level cache: we have cached most entities and collections of the application, and re-coded parts of the application to use this cache. As a result, using Hibernate’s “*showSql=true*” parameter, we can see that the application does not execute any unnecessary SQL request (as long as there is no write, requests are only executed once).

You can see those changes in the following commit:

[[Source code]](https://github.com/jdubois/spring-petclinic/commit/05a60b774011c41d522293ec3b2f4ed89b680ec5)

**Our final result goes up to 1225 req/sec**. That’s only a 15% performance boost, but please note that we have a very small database, and that it is running locally. On a real-world system, the improvement should be better, especially for the JpaOwnerRepositoryImpl.findByLastName() method.

**Extreme testing**

We decided to do an extreme testing session, in order to see if we really stabilized the application (using Tomcat’s NIO connector and going stateless, specifically).

So we ran our test again on the Macbook, still with 500 threads, but this time we let it run until we had **100 000 user sessions**.

Our results are extremely good:

* No HTTP error at all
* Memory stays at the same level, no matter how many users are using it
* The application runs smoothly, and stabilizes at **1565 req/sec** !!

This is, of course, an excellent result.

**Conclusion and final thoughts**

During those five days, we have gone through the classical steps we use at Ippon Technologies when auditing a project:

* Creating a “real” test case
* Removing data in the HTTP Sessions
* Removing the JVM locks
* Tuning the persistence layer
* Adding a cache

For each of these steps, JMeter and YourKit were our best tools to stress test the application and monitor how it responded to the test.

Of course, more work could have been done on JVM tuning, and on database tuning (but changing the database schema is outside the scope of this article).

At the beginning of the tests, we had to increase our heap memory size to 1 Gb, and could only serve 548 req/sec, with some HTTP errors. After completing our audit, we are now back to 128 M, and can serve 1225 req/sec with no error at all. We expect those results to be even better on a real server, with many cores and threads, and where removing JVM locks will have a more significant impact.

We also had a great improvement in the application stability, having 0 errors and 1565 req/sec with 100 000 users on our “extreme tests”. The application is now ready to handle a lot of users without any trouble.

Last but not least, we have switched the persistence layer from JDBC to JPA: a quick look at the code shows how much clearer and smaller the JPA code is. And the Spring Data JPA code is even clearer and smaller. It’s great to see that quality code can also be more performant than low-level, hard-to-code classes.

[edit]

You can find the other episodes of this series here : [part 1](http://blog.ippon.fr/?p=7496), [part 2](http://blog.ippon.fr/?p=7500), [part 3](http://blog.ippon.fr/?p=7512) and [part 4](http://blog.ippon.fr/?p=7520).