[How to use JPA correctly to avoid complaints of a slow application](https://zeroturnaround.com/rebellabs/how-to-use-jpa-correctly-to-avoid-complaints-of-a-slow-application/" \o "How to use JPA correctly to avoid complaints of a slow application)

* January 23, 2015

* [Roberto Cortez](https://zeroturnaround.com/author/robertocortez/)

* [15 Comments](https://zeroturnaround.com/rebellabs/how-to-use-jpa-correctly-to-avoid-complaints-of-a-slow-application/#disqus_thread)
* [inShare](javascript:void(0);)69

Setting the stage with a short story

*The day is clear and sunny, birds chirping happily as Lea, a project manager, makes her way to a meeting with Robert, a development team lead. “Dude, our application is slow,” she tells Robert.*

*Robert smirks. “Our code is impeccable, so this is probably a database problem. Let me ask our database admin, Dan,” and he picks up the phone.*

*“Hey Dan,” Robert says, “Lea says the application is slow. Can’t you create some indexes, or do something to speed it up?”*

*“Well Robert, it seems the database is flooded with queries! What the hell is this application doing??” cries Dan.*

*“Just deal with it!” exclaims Robert, hanging up.*

*“%$@#!!” mutters Dan.*

Over the last few years, I’ve come across with several enterprise applications that use JPA to manage their data, which is cool since JPA is a very powerful and awesome specification. Unfortunately, I came to realize that this technology is commonly used improperly, which generates a lot of complaints and even full-scale wars between database administrators (DBAs) and developers.

If you have some basic knowledge of JPA (which you should to get the most out of this article), then I can bet that many of you have heard a similar exchange before.

The double-edged sword of JPA

A great thing about JPA is that abstracts your interaction with the underlying database. A bad thing about JPA is that abstracts your interaction with the underlying database.

You can write database access code very easily and get most of the general database operations out of the box without having to write all that tedious JDBC code. On the other hand, you also need to have some knowledge of what’s going on behind the scenes or you’ll be in for some unpleasant surprises.

Believe it or not, I met a few developers that had no idea that JPA uses a database underneath. I feel that the majority of the developers are only concerned about getting the data they need, and don’t worry about anything else.

And this is why I decided to write this article: I’ve seen the same mistakes repeated over and over again, and they actually have a huge performance impact.

I’ve written down four areas in which I usually find all the issues: these are the ones I check first when I have to hunt down JPA performance problems.

* Eager fetching
* Lazy fetching
* Pagination
* Column select

Shamefully, I always find them out during my analysis after the fact, even though they seem completely obvious! In any case, I believe that these mistakes are not deliberately put into the application–most of the time, it’s poor knowledge about the technology itself. I hope this article could raise some awareness and guide developers to write better and faster JPA code.

There’s also another great post by Thorben Janssen where he talks about JPA features that can help with the application performance: [boosting your application performance with JPA](http://zeroturnaround.com/rebellabs/three-jpa-2-1-features-that-will-boost-your-applications-performance/).

Eager Fetching

Eager Fetching is a strategy that allows you to get additional data when you are loading an entity to have everything available for your needs…(but at what cost?)

Let’s imagine that you have a Department entity with an **@OneToMany**relationship to Employee defined with Eager Fetching. If you have a page where you are only displaying the Departments, you are also selecting the Employees for each Department. This scenario suffers from an increase of the loading time of data that you don’t need to fulfill your requirement.

So ask yourself: Do you really need all the data that you eagerly fetched? Most likely you don’t, not for every situation. So unless you know what you are doing, don’t use it.

Lazy Fetching

The Lazy Fetching strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first accessed. It seems like a good thing and usually it is, but sometimes it can slow down your application. Have you ever heard about the [N+1 select](http://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue) problem? This happens when you select an entity and then iterate the results to access a collection in a lazy fashion.

List departments = entityManager.createQuery("select d from

Department d").getResultList();

for (Department department : departments) {

// Issues a “select \* from Employee where departmentId = ?”

List employees = department.getEmployees();

}

This is incredibly inefficient, since you have to go to the database and bring the results one row at a time. If you already know that you’re going to need the Employee data, you could write the query like this:

List departments = entityManager.createQuery("select d from

Department d left join fetch d.employees").getResultList();

In this case, only a single database query is performed with all the data already populated in the return results.  
Keep in mind, that Lazy Fetching is just giving a hint to the provider. The implementation is permitted to eagerly fetch data for which the Lazy strategy hint has been specified, but the most popular ones have the same behavior regarding collections and the example I just gave.

Pagination for a quick win

Paginating your results is probably one of the best ways to increase the performance of your JPA application. If you have a table with 1 million records you are not going to display them all, right? RIGHT? Performing pagination on the client side is not the answer either, because the database had to return all the records anyway. Pagination should be done directly into the database, and you only have to call…

setFirstResult();

setMaxResults();

…in the Query object to paginate results.

Smart column select

Pagination deals with the amount of records (lines) in your table, but what about the number of columns? What if you have 100 columns, including BLOB’S, TEXT’s or other big sized data types? Even if you don’t, you should only select the required columns for the operation you are trying to perform. This will reduce the amount of data sent by the database to your application and speedup the query that you are executing.

Instead of writing:

entityManager.createQuery("select d from Department d")

You can write, if you only need the Department id:

entityManager.createQuery("select d.id from Department d")

The numbers that prove it all

Ok, so you probably thinking that this is all hogwash, and until you see some real numbers, there is no way this could be true.

I compiled a few test cases with some of the scenarios presented above and you can check the results in the table below. These are very simple tests: I added 2000 Departments with 100 columns each, and Departments have a one-to-many relationship with Employees, where each Department record has two Employees and Employees each have 100 columns.

|  |  |
| --- | --- |
| Find all records with relationships set to Lazy (n+1 selects) | 502 ms |
| Find all records with relationships set to Eager | 210 ms |
| Find all records with relationships set to Lazy, but relationships are fetched on the query | 206 ms |
| Find all records without relationships | 59 ms |
| Find all records without relationships, only with 10 columns | 12 ms |
| **Find all records without relationships, only with 10 columns, paginated** | **8 ms** |

The tests were run one at a time on three separate occasions, using Wildfly, Hibernate and H2 as the provider and with a database in a local environment. If you’d like to try it out yesterday, get the code on my GitHub page:  <https://github.com/radcortez/jpa-performance>

Two things you can do to avoid this mess

**1. Know your JPA Provider** – Hibernate, EclipseLink or OpenJPA are probably the most well known JPA Providers. While providers have to comply with the standards, the specification is open in a few scenarios, which may cause different behaviors for each implementation.  Think about the Lazy hint strategy that I explained a couple of lines ago as an example.

The Provider has a considerable amount of impact on your JPA application performance, so you should try to understand these little bits to get the most out of your chosen implementation. Some providers extended the specification and have a few features that you can use to improve performance if you don’t mind to sacrifice portability. Stay tuned; I’m planning to write another blog post exploring these on my own blog.

**2. Consider finding a DBA to join your team** – Databases are complex pieces of software. There are a lot of ways you can optimize and increase the performance of your queries and this also depends on the database engine that you’re using. In my opinion, having a specialist with you that can help you write optimized queries and monitor the application load is most of the times underrated.

In one of my previous jobs, I used to hear this a lot: “Database guys are not needed!” and I couldn’t disagree more. The most successful projects I have ever worked I always had a DBA backing me up. Probably without their help, I would be stuck working in some random car wash.

Conclusion: Boost app performance significantly by using JPA the right way

I think the numbers speak by themselves. The n + 1 select had the worst performance as expected. Both tests for relationships set to Eager or fetched on the query have similar results, since the query performed is the same, but remember it may be getting data that you don’t really need. Now, performing the query without the relationships have a very good boost, but you get better results if you just select the columns you need and squeeze the last extra bit by paginating the results.

Keep in mind that these times are only to demonstrate an order of magnitude between the different scenarios. Applying these techniques may not give you a flat linear performance increase, since you have to account in other factors like: database engine, network latency, system load etc., but for sure it’s going to help you to develop faster applications.

I hope you enjoyed this article, and feel free to leave me comments below, or ping me on Twitter [@radcortez](http://twitter.com/radcortez/). For more tech goodness, check out some **downloadable RebelLabs Reports!**

# [Dealing with org.hibernate.LazyInitializationException: could not initialize proxy - no Session in Hibernate Java](https://javarevisited.blogspot.com/2014/04/orghibernatelazyinitializationException-Could-not-initialize-proxy-no-session-hibernate-java.html)

If you are working in Hibernate framework, then you know that one of the key feature of Hibernate is "lazy initialization", which allows framework to lazily initialize dependencies, relationship or association lazily from database on need basis. For example, if you are dealing with User object, which has relationship with Permission object like one user can have multiple permissions, then Hibernate may choose not to initialize the collection which holds all permissions at the time it initialized User object and instead returns a proxy object. At this point, if you close your session and letter tries to access an attribute from Permission object, you will get *"org.hibernate.LazyInitializationException: could not initialize proxy - no Session in Hibernate"*.  
  
Why this error comes, because hibernate needs to go database to initialize the proxy object, and connection is already closed. If you remember, what we discussed in[difference between get vs load in hibernate](http://javarevisited.blogspot.sg/2012/07/hibernate-get-and-load-difference-interview-question.html) that Proxy object is only initialized in Hibernate if you access any attribute other than id itself, that's why you would only seeLazyInitializationException if you try to access an attribute other than id.  
  
In this article, we will see different scenarios on which you could possibly get"org.hibernate.LazyInitializationException: could not initialize proxy - no Session in Hibernate" and how to solve them appropriately.  
  
I have tried to explain reasons which caused the error, and explained the solution as why it will work, but if you still face issues, then feel free to post it here.  
  
By the way, good understanding of lazy initialization is also a [good Hibernate interview question](http://javarevisited.blogspot.sg/2013/05/10-hibernate-interview-questions-answers-java-j2ee-senior.html), so this not only help you to solve this error but also to do well during interviews.

### 1) Code tries to access a lazy initialized property or collection and session is not available.

This is by far most common reason of "LazyInitializationException: could not initialize proxy". In order to find the reason you need to look your code carefully. Here is one example to understand, how lazy initialization exception comes in Hibernate :

**Session** s **=** sessions.openSession();

**Transaction** tx **=** s.beginTransaction();

**Employee** e **=** (**Employee**) s.createQuery("from Employee e where e.name=:empName").setString("empName", eName).uniqueResult();

**List** roles **=** u.getRoles();

tx.commit();

s.close();

**String** role **=** roles.get(**0**); // This line will throw error

**Exception** in thread "main" org.hibernate.**LazyInitializationException:** **could** **not** **initialize** **proxy** **-** **no** **Session**

at org.hibernate.proxy.**AbstractLazyInitializer**.initialize(**AbstractLazyInitializer**.java**:**57)

at org.hibernate.proxy.**AbstractLazyInitializer**.getImplementation(**AbstractLazyInitializer**.java**:**111)

at org.hibernate.proxy.pojo.cglib.**CGLIBLazyInitializer**.invoke(**CGLIBLazyInitializer**.java**:**150)

**Easy Solution**  
Use lazy=false in hibernate mapping file.  
  
**Advantage and Disadvantages of lazy=false in Hibernate**  
Well, one clear advantage is that it's easy to apply, all you only need to change in Hibernate configuration files e.g. Employee.hbm.xml. It also guarantees that object will be fully initialized. On the other hand, main disadvantage of this approach can be slow performance. Since dependent objects are loaded at the time of persistent object loading, it will increase loading time. Also since now object is fully initialized, there memory consumption would be very high. This can become more severe if your [Collection classes](http://javarevisited.blogspot.sg/2014/01/ow-to-remove-objects-from-collection-arraylist-java-iterator-traversing.html) are big list of other objects, which are not always accessed.  
  
**Better Solution :**  
The real problem is that you are trying to access a collection in an object that is detached or associated session is closed. You need to re-attach the object before accessing the collection to the current session. You can either [reattach](http://javarevisited.blogspot.sg/2012/09/difference-hibernate-save-vs-persist-and-saveOrUpdate.html) the object by callingsession.update(object); Or you can move the code which access proxy object to the line before you close the session.  
  
In short, though making lazy=false is simple and sure short way to solve "Exception in thread "main" org.hibernate.LazyInitializationException: could not initialize proxy - no Session" it is not a good solution because you are throwing away the Lazy Initialization feature of hibernate. When lazy=false, the collection is loaded in memory at the same time that the object is requested. This means that if we have a collection with 1000 items, they all will be loaded in memory, despite we are going to access them or not. This can result in more memory consumption and slow initialization of object with lot of [association](http://javarevisited.blogspot.sg/2014/02/ifference-between-association-vs-composition-vs-aggregation.html)or dependency.

### 2) Upgrading from Hibernate 2.1 to Hibernate 3.0

Cause : Hibernate 3.0 provide lazy loading default as true i.e. lazy ="true"  
Solution : In hibernate mapping file set lazy= "false"  
  
Context : You may see **"org.hibernate.LazyInitializationException: could not initialize proxy - no Session"** while upgrading from hibernate 2.1 to hibernate 3.0. You will suddenly find yourself puzzling what happened, it was working before update. Reasons is, Hibernate 3 introduced lazy loading as the default i.e. lazy="true". If you want it to work the same as before you can mark everything as lazy="false". Alternatively you'll have to start eagerly initialising your entities and associations.

### 3) Hibernate with JPA Annotation

If you are using hibernate with JPA annotations and manually managing your transactions, then you can also try this as well to deal with LazyInitializationException in Hibernate . In your service class there should be a setter for entity manager with@PersistenceContext. change this to @PersistenceContext(type = PersistenceContextType.EXTENDED). Then you can access lazy property in any where. By the way, its worth remember that, Spring EXTENDED persistence context type is for long conversation pattern, not the session-per-request pattern.

### 4) Application wide Solution

There are situation, when we want an easy solution and doesn't care anything about performance e.g. for testing or prototyping purpose. In that case you can make following configuration change into your application to avoid this error, but remember the impact eager initialization can cause if this code makes its way to production.  
  
if you are using XML configuration: add default-lazy="false" to your element  
if you are using annotation configuration: add @Proxy(lazy=false) to all your [entity classes](http://javarevisited.blogspot.sg/2014/02/ifference-between-association-vs-composition-vs-aggregation.html).  
  
That's all about *how to fix Exception in thread "main" org.hibernate.LazyInitializationException: could not initialize proxy - no Session*. We have seen that this error mainly comes when you have closed the connection and trying to access the proxy object which is no fully initialized. Since Proxy object needs a connection, you can either reattach object to the session or carefully avoid writing such code, which access uninitialized Proxy object. Another way to avoid*LazyInitializationException*is to disable lazy initialization feature of hibernate for your entity classes by using lazy="false" or disable it completely for your application by using default-lazy="false". This solution is not recommended for production use due to performance reason but can be used during prototyping, testing and demo. Don't surprise if you first time see this error, when upgrading from Hibernate 2.1 to 3.0, because that's the version when Hibernate made lazy initialization enabled by default. If you have face this error in any other scenario or trying to solve "org.hibernate.LazyInitializationException: could not initialize proxy - no Session", you can also post your error and code here and we can take a look together.