



ad DZone's 2019 Migrating to Microservices Trend Report to learn about the next phase of microservices adop

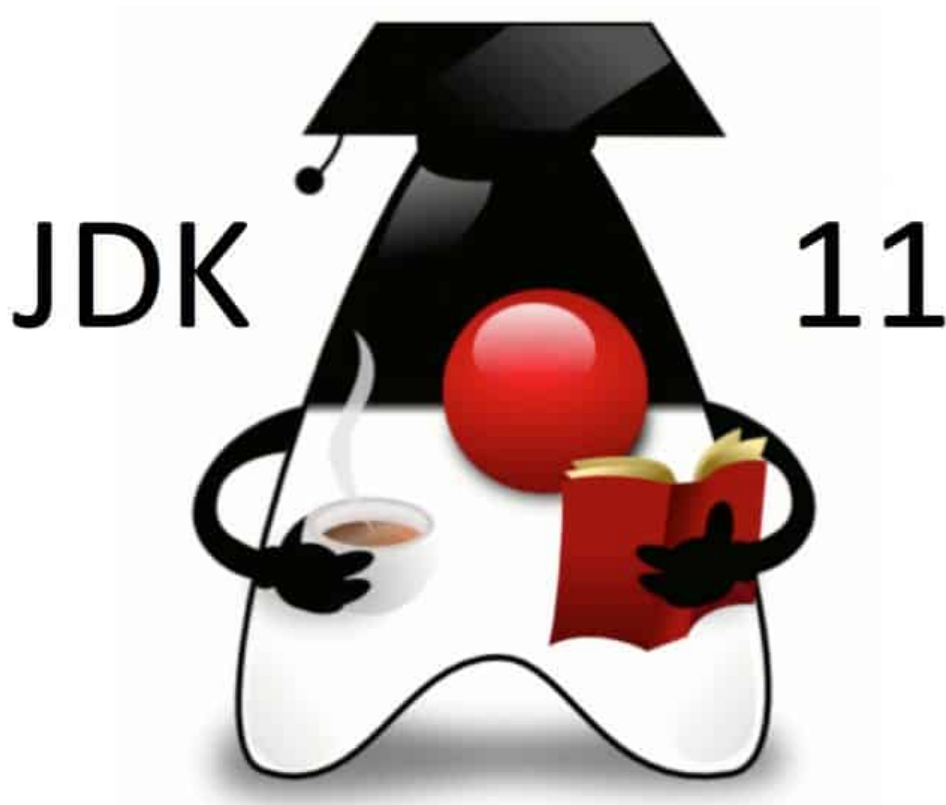
[Read Now▶](#)

# 90 New Features and APIs in JDK 11 (Part 1)

by Simon Ritter  MVB  · Sep. 27, 18 · Java Zone · Presentation

Build, test, and launch Java applications faster using a JVM. [Get your Free Trial of Zing with ReadNow! to stay fast.](#)

Presented by Azul Systems



The new six-month release cadence of the JDK means that before we've even really figured out what the new features are in JDK 10 along comes JDK 11. I posted an earlier blog where I listed all 109 new features and APIs I could find in JDK 10, so it seemed obvious to do the same thing for JDK 11. I'm going to use a different format from the previous post. In this post, I'll divide everything into two sections: features that are visible to developers and everything else. This way, if you're interested in just what will affect your development work, you can check out the second installment.

The total I counted was 90 (that's JEPs plus new classes and methods, excluding the individual ones for the HTTP

client and Flight Recorder). Although that's eleven less than I found in JDK 10, I think it's fair to say more functionality has been added to JDK 11 — certainly at the JVM level.

## Developer Visible Features

JDK 11 is pretty light on things that change the way you code. There is one small change to the language syntax, a fair number of new APIs, and the ability to run single-file applications without the need to use the compiler. Also, the removal of the `java.se.ee` aggregator module is now visible, which may impact migrating an existing application to JDK 11.

### JEP 323: Local-Variable Syntax for Lambda Parameters

JDK 10 introduced the Local-Variable Type Inference (JEP 286). This simplifies the code, as you no longer need to explicitly state the type of a local-variable but can, instead, use `var`. JEP 323 extends the use of this syntax to the parameters of Lambda expressions. Here's a simple example:

```
1 list.stream()  
2     .map((var s) -> s.toLowerCase())  
3     .collect(Collectors.toList());
```

Of course, the astute Java programmer would point out that Lambda expressions already have type inference so the use of `var` would (in this case) be superfluous. We could just as easily write the same code as:

```
1 list.stream()  
2     .map(s -> s.toLowerCase())  
3     .collect(Collectors.toList());
```

Why add `var` support, then? The answer is for one special case, which is when you want to add an annotation to the Lambda parameter. It is not possible to do this without a type being involved. To avoid having to use the explicit type, we can use `var` to simplify things, thus:

```
1 list.stream()  
2     .map((@NotNull var s) -> s.toLowerCase())  
3     .collect(Collectors.toList());
```

This feature has required changes to the Java Language Specification (JLS), specifically:

- Page 24: The description of the `var` special identifier.
- Page 627-30: Lambda parameters
- Page 636: Runtime evaluation of Lambda expressions
- Page 746: Lambda syntax

### JEP 330: Launch Single-File Source-Code Programs

One of the criticisms of Java is that it can be verbose in its syntax and the 'ceremony' associated with running even a trivial application can make it hard to approach as a beginner. To write an application that just prints "Hello

World!,” it requires you to write a class with a `public static void main` method and use the `System.out.println` method. Having done this, you must then compile the code using `javac`. Finally, you can run the application to be welcomed into the world. Doing the same thing in most scripting languages is *significantly* simpler and quicker.

JEP 330 eliminates the need to compile a single-file application, so now you can type:

```
1 java HelloWorld.java
```

The Java launcher will identify that the file contains Java source code and will compile the code to a class file before executing it.

Parameters placed *after* the name of the source file are passed as parameters when executing the application. Parameters placed *before* the name of the source file are passed as parameters to the Java launcher after the code has been compiled. This allows for things like the classpath to be set on the command line. Parameters that are relevant to the compiler (such as the classpath) will also be passed to `javac` for compilation.

As an example:

```
1 java -classpath /home/foo/java Hello.java Bonjour
```

would be equivalent to:

```
1 javac -classpath /home/foo/java Hello.java
2 java -classpath /home/foo/java Hello Bonjour
```

This JEP also provides ‘shebang’ support. To reduce the need to even mention the Java launcher on the command line, this can be included on the first line of the source file. For example:

```
1 #!/usr/bin/java --source 11
2     public class HelloWorld {
3         ...
```

It is necessary to specify the `--source` flag with the version of Java to use.

## JEP 321: HTTP Client (Standard)

JDK 9 introduced a new API to provide support for the HTTP Client protocol (JEP 110). Since JDK 9 introduced the Java Platform Module System (JPMS), this API was included as an *incubator module*. Incubator modules are intended to provide new APIs without making them part of the Java SE standard. Developers can try the API and provide feedback. Once any necessary changes have been made (this API was updated in JDK 10), the API can be moved to become part of the standard.

The HTTP Client API is now part of the Java SE 11 standard. This introduces a new module and package to the JDK, `java.net.http`. The main types defined are:

- `HttpClient`

- `HttpRequest`
- `HttpResponse`
- `WebSocket`

The API can be used synchronously or asynchronously. The asynchronous mode makes use of `CompletableFuture` and `CompletionStages`.

## JEP 320: Remove The Java EE and CORBA Modules

With the introduction of JPMS in JDK 9, it was possible to divide the monolithic `rt.jar` file into multiple modules. An additional advantage of JPMS is it is now possible to create a Java runtime that only includes the modules you need for your application, reducing the size considerably. With cleanly defined module boundaries, it is now simpler to remove parts of the Java API that are outdated. This is what this JEP does; the `java.se.ee` meta-module includes six modules that will no longer be part of the Java SE 11 standard and not included in the JDK. The affected modules are:

- `corba`
- `transaction`
- `activation`
- `xml.bind`
- `xml.ws`
- `xml.ws.annotation`

These modules have been deprecated since JDK 9 and were not included by default in either compilation or runtime. If you had tried compiling or running an application that used APIs from these modules on JDK 9 or JDK 10, they would have failed. If you use APIs from these modules in your code, you will need to supply them as a separate module or library. From asking audiences at my presentations, it seems that the `java.xml` modules, which are part of the JAX-WS, SOAP-based web services support are the ones that will cause most problems.

## New APIs

A lot of the new APIs in JDK 11 result from the HTTP client module now being part of the standard, as well as the inclusion of Flight Recorder.

For a complete list of API changes, I refer the reader to the excellent comparison of different JDK versions produced by Gunnar Morling, which is available on Github.

What I list here are all the new methods other than those in the `java.net.http` and `jdk.jfr` modules. I've also not listed the new methods and classes in the `java.security` modules, which are pretty specific to the changes of JEP 324 and JEP 329 (there are six new classes and eight new methods).

### **`java.io.ByteArrayOutputStream`**

- `void writeBytes(byte[]):` Write all the bytes of the parameter to the output stream

### **`java.io.FileReader`**

Two new constructors that allow a `Charset` to be specified.

## **java.io.FileWriter**

Four new constructors that allow a Charset to be specified.

## **java.io.InputStream**

- `io.InputStream nullInputStream()` : Returns an `InputStream` that reads no bytes. When you first look at this method (and the ones in `OutputStream`, `Reader`, and `Writer`), you wonder what use they are. You can think of them like `/dev/null` for throwing away an output you don't need or providing an input that always returns zero bytes.

## **java.io.OutputStream**

- `io.OutputStream nullOutputStream()`

## **java.io.Reader**

- `io.Reader nullReader()`

## **java.io.Writer**

- `io.Writer nullWriter()`

## **java.lang.Character**

- `String toString(int)` : This is an overloaded form of the existing method but takes an `int` instead of a `char`. The `int` is a Unicode code point.

## **java.lang.CharSequence**

- `int compare( CharSequence , CharSequence )` : Compares two `CharSequence` instances lexicographically. Returns a negative value, zero, or a positive value if the first sequence is lexicographically less than, equal to, or greater than the second, respectively.

## **java.lang.ref.Reference**

- `lang.Object clone()` : I must admit, this one confuses me. The `Reference` class does not implement the `Cloneable` interface and this method will always throw a `CloneNotSupportedException`. There must be a reason for its inclusion, presumably for something in the future.

## **java.lang.Runtime**

## **java.lang.System**

No new methods here but worth mentioning that the `runFinalizersOnExit()` method has now been removed from both these classes (this could be a compatibility issue).

## **java.lang.String**

I think this is one of the highlights of the new APIs in JDK 11. There are several useful new methods here.

- `boolean isBlank()` : Returns true if the string is empty or contains only white space codepoints, otherwise false.
- `Stream lines()` : Returns a stream of lines extracted from this string, separated by line terminators.

- `String repeat(int)` : Returns a string whose value is the concatenation of this string repeated count times.
- `String strip()` : Returns a string whose value is this string, with all leading and trailing whitespace removed.
- `String stripLeading()` : Returns a string whose value is this string, with all leading whitespace removed.
- `String stripTrailing()` : Returns a string whose value is this string, with all trailing whitespace removed.

You probably look at `strip()` and ask, “How is this different to the existing `trim()` method?” The answer is that how whitespace is defined differs between the two.

### **java.lang.StringBuffer**

### **java.lang.StringBuilder**

Both these classes have a new `compareTo()` method that takes a `StringBuffer / StringBuilder` and returns an `int`. The lexicographical comparison method is the same as the new `compareTo()` method of the `CharSequence`.

### **java.lang.Thread**

No additional methods but the `destroy()` and `stop(Throwable)` methods have been removed. The `stop()` method that takes no arguments is still present. This might present a compatibility issue.

### **java.nio.ByteBuffer**

### **java.nio.CharBuffer**

### **java.nio.DoubleBuffer**

### **java.nio.FloatBuffer**

### **java.nio.LongBuffer**

### **java.nio.ShortBuffer**

All these classes now have a `mismatch()` method that finds and returns the relative index of the first mismatch between this buffer and a given buffer.

### **java.nio.channels.SelectionKey**

- `int interestOpsAnd(int)` : Atomically sets this key’s interest set to the bitwise intersection (“and”) of the existing interest set and the given value.
- `int interestOpsOr(int)` : Atomically sets this key’s interest set to the bitwise union (“or”) of the existing interest set and the given value.

### **java.nio.channels.Selector**

- `int select(java.util.function.Consumer, long)` : Selects and performs an action on the keys whose corresponding channels are ready for I/O operations. The long parameter is a timeout.
- `int select(java.util.function.Consumer)` : As above, except without the timeout.
- `int selectNow(java.util.function.Consumer)` : As above, except it is non-blocking.

### **java.nio.file.Files**

- `String readString(Path)` : Reads all content from a file into a string, decoding from bytes to characters

using the UTF-8 charset.

- `String readString(Path, Charset)` : As above, except decoding from bytes to characters using the specified Charset.
- `Path writeString(Path, CharSequence, java.nio.file. OpenOption[])` : Write a CharSequence to a file. Characters are encoded into bytes using the UTF-8 charset.
- `PathwriteString(Path, CharSequence, java.nio.file. Charset, OpenOption[])` : As above, except Characters are encoded into bytes using the specified Charset.

### **java.nio.file.Path**

- `Path of(String, String[])` : Returns a Path by converting a path string, or a sequence of strings that when joined form a path string.
- `Path of(net.URI)` : Returns a Path by converting a URI.

### **java.util.Collection**

- `Object[] toArray(java.util.function.IntFunction)` : Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.

### **java.util.concurrent.PriorityBlockingQueue**

#### **java.util.PriorityQueue**

- `void forEach(java.util.function.Consumer)` : Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
- `boolean removeAll(java.util.Collection)` : Removes all of this collection's elements that are also contained in the specified collection (optional operation).
- `boolean removeIf(java.util.function.Predicate)` : Removes all of the elements of this collection that satisfy the given predicate.
- `boolean retainAll(java.util.Collection)` : Retains only the elements in this collection that are contained in the specified collection (optional operation).

### **java.util.concurrent.TimeUnit**

- `long convert(java.time.Duration)` : Converts the given time duration to this unit.

### **java.util.function.Predicate**

- `Predicate not(Predicate)` . Returns a predicate that is the negation of the supplied predicate.

This is one of my favourite new APIs in JDK 11. As an example, you can convert this code:

```
1 lines.stream()  
2     .filter(s -> !s.isBlank())
```

to

```
1 lines.stream()  
2     .filter(Predicate.not(String::isBlank))
```

And, if we use a static import, it becomes:

```
1 lines.stream()  
2     .filter(not(String::isBlank))
```

Personally, I think this version is more readable and easier to understand.

### **java.util.Optional**

### **java.util.OptionalInt**

### **java.util.OptionalDouble**

### **java.util.OptionalLong**

- `boolean isEmpty()` : If a value is not present, it returns true, otherwise it is false.

### **java.util.regex.Pattern**

- `Predicate asMatchPredicate()` : I think this could be a hidden gem in the new JDK 11 APIs. It creates a predicate that tests if this pattern matches a given input string.

### **java.util.zip.Deflater**

- `int deflate(ByteBuffer)` : Compresses the input data and fills the specified buffer with compressed data.
- `int deflate(ByteBuffer, int)` : Compresses the input data and fills the specified buffer with compressed data. Returns the actual number of bytes of data compressed.
- `void setDictionary(ByteBuffer)` : Sets the preset dictionary for compression to the bytes in the given buffer. This is an overloaded form of an existing method that can now accept a ByteBuffer, rather than a byte array.
- `void setInput(ByteBuffer)` : Sets input data for compression. Also an overloaded form of an existing method.

### **java.util.zip.Inflater**

- `int inflate(ByteBuffer)` : Uncompresses bytes into the specified buffer. Returns the actual number of bytes uncompressed.
- `void setDictionary(ByteBuffer)` : Sets the preset dictionary to the bytes in the given buffer. An overloaded form of an existing method.
- `void setInput(ByteBuffer)` : Sets input data for decompression. An overloaded form of an existing method.

### **javax.print.attribute.standard.DialogOwner**

- This is a new class in JDK 11 and is an attribute class used to support requesting a print or page setup dialog be kept displayed on top of all windows or some specific window.

### **javax.swing.DefaultComboBoxModel**

### **javax.swing.DefaultListModel**

- `void addAll(Collection)` : Adds all of the elements present in the collection.



- `void addAll(int, Collection)` : Adds all of the elements present in the collection, starting from the specified index.

### **javax.swing.ListSelectionModel**

- `int[] getSelectedIndices()` : Returns an array of all of the selected indices in the selection model in increasing order.
- `int getSelectedItemsCount()` : Returns the number of selected items.

### **jdk.jshell.EvalException**

- `jshell.JShellException getCause()` : Returns the wrapped cause of the throwable in the executing client represented by this `EvalException` or null if the cause is non-existent or unknown.

Like what you see? Click here for Part 2.

---

Discover how you can stop fighting Java application timeouts, jitter and glitches. [Get a free a better JVM.](#)

Presented by Azul Systems

---

## **Like This Article? Read More From DZone**



**API Updates in Java SE 11 (18.9)**



**90 New Features and APIs in JDK 11 (Part 2)**



**JDK 12 News: Switch Expressions and Raw String Literals**



**Free DZone Refcard**  
**Getting Started With Headless CMS**

Topics: JAVA , APIS , JDK 11 , FEATURES , JLS

Published at DZone with permission of Simon Ritter , DZone MVB. [See the original article here.](#)

Opinions expressed by DZone contributors are their own.

# **Best Performance Practices for Hibernate 5 and Spring Boot 2 (Part 1)**

by Anghel Leonard MVB · Nov 16, 18 · Java Zone · Tutorial

Learn how enabling deeper real-time analytics can benefit your users and your bottom line, the benefits of moving to a hybrid transactional/analytical processing (HTAP) system. [Watch demand webinar now!](#)

Presented by MariaDB

---

# Item 1: Attribute Lazy Loading Via Bytecode Enhancement

By default, the basic attributes of an entity are loaded eager (all at once). Are you sure that you want that?

**Description:** If not, then it is important to know that basic attributes can be loaded lazily as well via Hibernate *Bytecode Enhancement (instrumentation)*. This is useful for lazy loading the column types that store large amounts of data: CLOB , BLOB , VARBINARY , etc. or columns that should be loaded on demand.

## Key points:

- For Maven, in `pom.xml` , activate Hibernate *BytecodeEnhancement* (e.g. use Maven bytecode enhancement plugin as follows)
- Mark the columns that should be loaded lazily with `@Basic(fetch = FetchType.LAZY)`
- In `application.properties` , disable Open Session in View

Source code can be found [here](#).

You may also like:

[Default Values For Lazy Loaded Attributes](#)

[Attribute Lazy Loading And Jackson Serialization](#)

If this approach is not proper for you then the same result can be obtained via subentities. Consider reading [Attributes Lazy Loading Via Subentities](#).

# Item 2: View Binding Parameter Values Via Log4J 2

Without seeing and inspecting the SQL fired behind the scenes and the corresponding binding parameters, we are prone to introduce performance penalties that may remain there for a long time (e.g. N+1).

**Update (please read):** The solution described below is useful if you **already** have Log4J 2 in your project. If not, it is better to rely on TRACE (thank you Peter Wippermann for your suggestion) or `log4jdbc` (thank you, Sergei Poznanski, for your suggestion and SO answer). Both approaches don't require the exclusion of Spring Boot's Default Logging. An example of TRACE can be found [here](#), and an example of `log4jdbc` [here](#).

**Description based on Log4J 2:** While the application is under development, maintenance is useful to view and inspect the prepared statement binding parameter values instead of assuming them. One way to do this is via Log4J 2 logger setting.

## Key points:

- For Maven, in `pom.xml` , exclude Spring Boot's Default Logging (read update above)
- For Maven, in `pom.xml` , add the Log4j 2 dependency
- In `log4j2.xml` , add the following:

## Output sample:

```

insert into author (age, genre, name) values (?, ?, ?)
binding parameter [1] as [INTEGER] - [34]
binding parameter [2] as [VARCHAR] - [History]
binding parameter [3] as [VARCHAR] - [Joana Nimar]

select author0_.id as id1_0_0_, author0_.age as age2_0_0_, author0_.genre as genre3_0_0_,
       author0_.name as name4_0_0_ from author author0_ where author0_.id=?
binding parameter [1] as [BIGINT] - [1]
extracted value ([age2_0_0_] : [INTEGER]) - [34]
extracted value ([genre3_0_0_] : [VARCHAR]) - [History]
extracted value ([name4_0_0_] : [VARCHAR]) - [Joana Nimar]

```

Source code can be found [here](#).

## Item 3: How To View Query Details Via DataSource-Proxy

Without ensuring that batching is actually working, we are prone to serious performance penalties. There are different cases when batching is disabled, even if we have it set up and think that it is working behind the scene. For checking, we can use `hibernate.generate_statistics` to display details (including batching details), but we can go with the DataSource-Proxy library, as well.

**Description:** View the query details (query type, binding parameters, batch size, etc.) via DataSource-Proxy.

### Key points:

- For Maven, add in the `pom.xml` the DataSource-Proxy dependency
- Create a bean post-processor to intercept the `DataSource` bean
- Wrap the `DataSource` bean via the `ProxyFactory` and implementation of the `MethodInterceptor`

### Output sample:

```

Name:DATA_SOURCE_PROXY, Connection:5, Time:48, Success:True
Type:Prepared, Batch:False, QuerySize:1, BatchSize:0
Query:["insert into author (age, genre, name) values (?, ?, ?)"]
Params:[(34,History,Joana Nimar)]

```

Source code can be found [here](#).

## Item 4: Batch Inserts Via `saveAll(Iterable<S> entities)`

By default, 100 inserts will result in 100 SQL INSERT statements and this is bad since it results in 100 database round trips.

**Description:** Batching is a mechanism capable of grouping INSERTs, UPDATEs, and DELETEs, and as a consequence, it significantly reduces the number of database round trips. One way to achieve batch inserts consists of using the `SimpleJpaRepository#saveAll(Iterable<S> entities)` method. Here, we do this with MySQL. The recommended batch size is between 5 and 30.

### Key points:

- In `application.properties`, set `spring.jpa.properties.hibernate.jdbc.batch_size`
- In `application.properties`, set `spring.jpa.properties.hibernate.generate_statistics` (just to check that batching is working)
- In `application.properties`, set JDBC URL with `rewriteBatchedStatements=true` (optimization specific for MySQL)
- In `application.properties` set JDBC URL with `cachePrepStmts=true` (enable caching and is useful if you decide to set `prepStmtCacheSize`, `prepStmtCacheSqlLimit`, etc as well; without this setting the cache is disabled)
- In `application.properties` set JDBC URL with `useServerPrepStmts=true` (this way you switch to server-side prepared statements (may lead to significant performance boost))
- In the entity, use the assigned generator since MySQL `IDENTITY` will cause insert batching to be disabled
- In the entity, add a property annotated with `@Version` to avoid the extra- `SELECT` fired before batching (also prevent lost updates in multi-request transactions). Extra- `SELECT`s are the effect of using `merge()` instead of `persist()`. Behind the scenes, `saveAll()` uses `save()`, which in case of non-new entities (entities having IDs), will call `merge()`, which instructs Hibernate to fire to a `SELECT` statement to ensure that there is no record in the database having the same identifier.
- Pay attention to the number of inserts passed to `saveAll()` to not "overwhelm" the Persistence Context. Normally, the `EntityManager` should be flushed and cleared from time to time, but during the `saveAll()` execution, you simply cannot do that, so if in `saveAll()` there is a list with a high amount of data, all that data will hit the Persistence Context (1st level cache) and will be in-memory until flush time. Using a relatively small amount of data should be OK.
- The `saveAll()` method return a `List<S>` containing the persisted entities; each persisted entity is added into this list; if you just don't need this `List` then it is created for nothing
- **For a large amount of data, call `saveAll()` per batch and set batch size between 5 and 30.** Moreover, please check the next item as well (item 5).

#### Output sample:

```
441797 nanoseconds spent acquiring 1 JDBC connections;
0 nanoseconds spent releasing 0 JDBC connections;
5423247 nanoseconds spent preparing 1 JDBC statements;
0 nanoseconds spent executing 0 JDBC statements;
55628238 nanoseconds spent executing 1 JDBC batches;
0 nanoseconds spent performing 0 L2C puts;
0 nanoseconds spent performing 0 L2C hits;
0 nanoseconds spent performing 0 L2C misses;
```

Source code can be found [here](#).

## Item 5: How To Optimize Batch Inserts of Parent-Child Relationships And Batch Per Transaction

**Description:** Let's suppose that we have a one-to-many relationship between `Author` and `Book` entities. When we save an author, we save his books as well thanks to cascading `all/persist`. We want to create a bunch of authors with books and save them in the database (e.g., a MySQL database) using the batch technique. By default, this will

result in batching each author and the books per author (one batch for the author and one batch for the books, another batch for the author and another batch for the books, and so on). In order to batch authors and books, we need to **order inserts** as in this application.

Moreover, this example commits the database transaction after each batch execution. This way we avoid long-running transactions and, in case of a failure, we rollback only the failed batch and don't lose the previous batches. For each batch, the Persistent Context is flushed and cleared, therefore we maintain a thin Persistent Context. This way the code is not prone to memory errors and performance penalties caused by slow flushes.

### Key points:

- Besides all settings specific to batching inserts in MySQL (see Item 4), we need to set up in `application.properties` the following property: `spring.jpa.properties.hibernate.order_inserts=true`

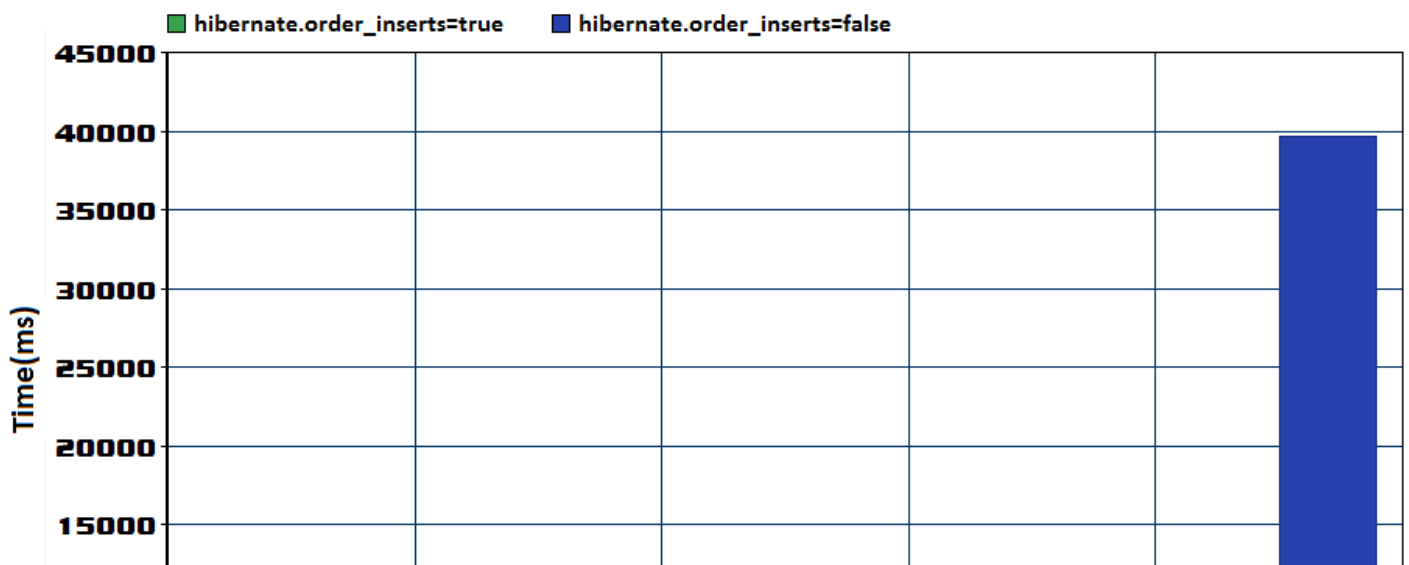
### Output sample without ordering inserts (80 batches):

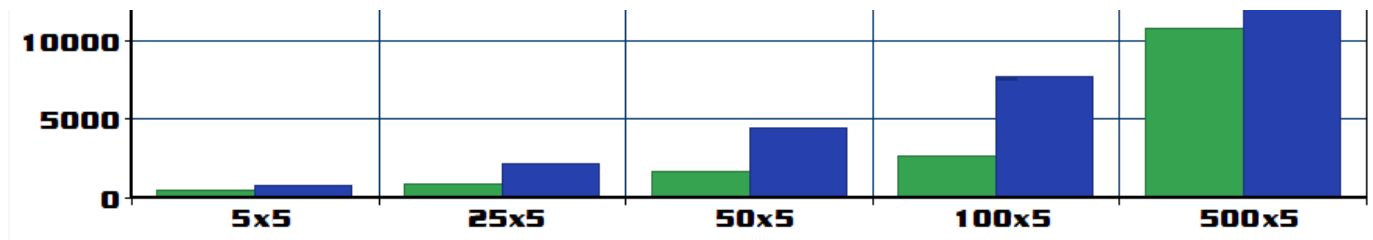
```
471614 nanoseconds spent acquiring 1 JDBC connections;
0 nanoseconds spent releasing 0 JDBC connections;
25202794 nanoseconds spent preparing 80 JDBC statements;
0 nanoseconds spent executing 0 JDBC statements;
3065506548 nanoseconds spent executing 80 JDBC batches;
0 nanoseconds spent performing 0 L2C puts;
0 nanoseconds spent performing 0 L2C hits;
0 nanoseconds spent performing 0 L2C misses;
3238487470 nanoseconds spent executing 3 flushes (flushing a total of 240 entities and 40 collections);
0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
```

### Output sample with ordering inserts (17 batches):

```
460863 nanoseconds spent acquiring 1 JDBC connections;
0 nanoseconds spent releasing 0 JDBC connections;
18206769 nanoseconds spent preparing 6 JDBC statements;
0 nanoseconds spent executing 0 JDBC statements;
761763240 nanoseconds spent executing 17 JDBC batches;
0 nanoseconds spent performing 0 L2C puts;
0 nanoseconds spent performing 0 L2C hits;
0 nanoseconds spent performing 0 L2C misses;
870081646 nanoseconds spent executing 3 flushes (flushing a total of 240 entities and 40 collections);
0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
```

How much it counts? Check this out for 5 authors with 5 books each to 500 authors with 5 books each:





Source code can be found [here](#).

You may also like the following:

"Item 6: Batch Inserts Via EntityManager With Batch Per Transaction"

"Item 7: Batch Inserts In Spring Boot Style Via CompletableFuture"

## Item 8: Direct Fetching Via Spring Data / EntityManager / Session

The way, we fetch data from the database determines how an application will perform. In order to build the optimal fetching plan, we need to be aware of each fetching type. *Direct fetching* is the simplest (since we don't write any explicit query) and very useful when we know the entity *Primary Key*.

**Description:** Direct fetching via Spring Data, EntityManager, and Hibernate Session examples.

### Key points:

- Direct fetching via Spring Data, `findById()`
- Direct fetching via `EntityManager#find()`
- Direct fetching via Hibernate `Session#get()`

Source code can be found [here](#). Direct fetching multiple entities by id can be done via Spring Data, `findAllById()` and the great Hibernate `MultiIdentifierLoadAccess` interface.

## Item 9: DTOs Via Spring Data Projections

Fetching more data than needed is one of the most common issues causing performance penalties. Fetching entities without the intention of modifying them is also a bad idea.

**Description:** Fetch only the needed data from the database via Spring Data Projections (DTOs). See also items 25-32.

### Key points:

- Write an interface (projection) containing getters only for the columns that should be fetched from the database
- Write the proper query returning a `List <projection>`
- If possible, limit the number of returned rows (e.g., via `LIMIT`). Here, we can use the Query Builder mechanism built into the Spring Data repository infrastructure.

**Output example (select first 2 rows; select only "name" and "age"):**

id	age	genre	name
1	23	Anthology	Mark Janel
2	43	Horror	Olivia Goy
3	51	Anthology	Quartis Young
4	34	History	Joana Nimar
5	38	Anthology	Alicia Tom
6	56	Anthology	Katy Loin

```
select
  author0_.name as col_0_0_,
  author0_.age as col_1_0_
from
  author author0_
where
  author0_.genre = ?
limit ?
```

Number of authors:2

Author name: Mark Janel | Age: 23

Author name: Quartis Young | Age: 51

Source code can be found [here](#).

**Note:** Using projections is not limited to use the Query Builder mechanism built into Spring Data repository infrastructure. We can fetch projections via JPQL or native queries as well. For example, in this application, we use a JPQL. Moreover, Spring projection can be nested. Consider this application and this application. Pay extra attention on using nested projection and the involved performance penalties. This topic is detailed in my book, **Spring Boot Persistence Best Practices**.

## Item 10: How To Store UTC Timezone In MySQL

Storing date-time and timestamps in the database in different/specific formats can cause real issues when dealing with conversions.

**Description:** This recipe shows you how to store date-time, and timestamps in UTC time zone in MySQL. For other RDBMSs (e.g. PostgreSQL), just remove " useLegacyDatetimeCode=false " and adapt the JDBC URL.

### Key points:

- `spring.jpa.properties.hibernate.jdbc.time_zone=UTC`
- `spring.datasource.url=jdbc:mysql://localhost:3306/db_screenshot?useLegacyDatetimeCode=false`

Source code can be found [here](#).

## Item 11: Populating a Child-Side Parent Association Via Proxy

Executing more SQL statements than needed is always a performance penalty. It is important to strive to reduce their number as much as possible, and relying on references is one of the easy to use optimization.

**Description:** A Hibernate proxy can be useful when a child entity can be persisted with a reference to its parent ( @ManyToOne or @OneToOne lazy association). In such cases, fetching the parent entity from the database (execute the SELECT statement) is a performance penalty and a pointless action. Hibernate can set the underlying foreign key value for an uninitialized proxy.

### Key points:



- Rely on `EntityManager#getReference()`
- In Spring, use `JpaRepository#getOne()`
- Used in this example, in Hibernate, use `load()`
- Assume two entities, `Author` and `Book`, involved in a unidirectional `@ManyToOne` association (`Author` is the parent-side)
- We fetch the author via a proxy (this will not trigger a `SELECT`), we create a new book, we set the proxy as the author for this book and we save the book (this will trigger an `INSERT` in the `book` table)

### Output sample:

- The console output will reveal that only an `INSERT` is triggered, and no `SELECT`

Source code can be found here.

## Item 12: Reproducing N+1 Performance Issue

N+1 is another issue that may cause serious performance penalties. In order to eliminate it, you have to find/recognize it. It is not always easy, but here is one of the most common scenarios that lead to N+1.

**Description:** N+1 is an issue of lazy fetching (but, eager is not exempt). Just in case you didn't have the chance to see it in action, this application reproduces the N+1 behavior. In order to avoid N+1 is better to rely on JOIN+DTO (there are examples of JOIN+DTOs in items 36-42).

### Key points:

- Define two entities, `Author` and `Book` in a lazy bidirectional `@OneToMany` association
- Fetch all `Book` lazy, so without `Author` (results in 1 query)
- Loop the fetched `Book` collection and for each entry fetch the corresponding `Author` (results N queries)
- Or, fetch all `Author` lazy, so without `Book` (results in 1 query)
- Loop the fetched `Author` collection and for each entry fetch the corresponding `Book` (results N queries)

### Output sample:

author				book			
id	age	genre	name	id	isbn	title	author_id
1	23	Anthology	Mark Janel	1	001-JN	A History of Ancient Prague	4
2	43	Horror	Olivia Goy	2	001-QY	Modern Anthology	3
3	51	Anthology	Quartis Young	3	001-MJ	The Beatles Anthology	1
4	34	History	Joana Nimar	4	001-OG	Carrie	2

1

N

```
SELECT
  book0_.id AS id1_1_,
  book0_.author_id AS author_i4_1_,
  book0_.isbn AS isbn2_1_,
  book0_.title AS title3_1_
FROM book book0_
```

+

Book: A History of Ancient Prague Author: Joana Nimar

```
SELECT
  author0_.id AS id1_0_0_,
  author0_.age AS age2_0_0_,
  author0_.genre AS genre3_0_0_,
  author0_.name AS name4_0_0_
FROM author author0_
WHERE author0_.id = ?
```

Book: Modern Anthology Author: Quartis Young

```
SELECT
  author0_.id AS id1_0_0_,
  author0_.age AS age2_0_0_,
  author0_.genre AS genre3_0_0_,
  author0_.name AS name4_0_0_
FROM author author0_
WHERE author0_.id = ?
```

Book: The Beatles Anthology Author: Mark Janel

```
SELECT
  author0_.id AS id1_0_0_,
  author0_.age AS age2_0_0_,
  author0_.genre AS genre3_0_0_,
  author0_.name AS name4_0_0_
FROM author author0_
WHERE author0_.id = ?
```

Book: Carrie Author: Olivia Goy

```
SELECT
  author0_.id AS id1_0_0_,
  author0_.age AS age2_0_0_,
  author0_.genre AS genre3_0_0_,
  author0_.name AS name4_0_0_
FROM author author0_
WHERE author0_.id = ?
```



Source code can be found [here](#).

## Item 13: Optimize Distinct SELECTs Via HINT\_PASS\_DISTINCT\_THROUGH Hint

**Description:** Starting with Hibernate 5.2.2, we can optimize JPQL (HQL) query entities of type `SELECT DISTINCT` via `HINT_PASS_DISTINCT_THROUGH` hint. Keep in mind that this hint is useful only for JPQL (HQL) `JOIN FETCH` -ing queries. It is not useful for scalar queries (e.g., `List`), DTO or HHH-13280. In such cases, the `DISTINCT` JPQL keyword is needed to be passed to the underlying SQL query. This will instruct the database to remove duplicates from the result set.

### Key points:

- Use `@QueryHints(value = @QueryHint(name = HINT_PASS_DISTINCT_THROUGH, value = "false"))`

### Output sample:

```

Fetching authors with duplicates ...
Hibernate: select author0_.id as id1_0_0_, books1_.id as id1_1_1_, author0_.age as age2_0_0_, author0_.genre as genre3_0_0_, a
uthor0_.name as name4_0_0_, books1_.author_id as author_i4_1_1_, books1_.isbn as isbn2_1_1_, books1_.title as title3_1_1_, boo
ks1_.author_id as author_i4_1_0_, books1_.id as id1_1_0_ from author author0_ left outer join book books1_ on author0_.id=bo
oks1_.author_id
Id: 1: Name: Joana Nimar
Id: 1: Name: Joana Nimar

Fetching authors without HINT_PASS_DISTINCT_THROUGH hint ...
Hibernate: select distinct author0_.id as id1_0_0_, books1_.id as id1_1_1_, author0_.age as age2_0_0_, author0_.genre as genre
3_0_0_, author0_.name as name4_0_0_, books1_.author_id as author_i4_1_1_, books1_.isbn as isbn2_1_1_, books1_.title as title3_
1_1_, books1_.author_id as author_i4_1_0_, books1_.id as id1_1_0_ from author author0_ left outer join book books1_ on autho
r0_.id=books1_.author_id
Id: 1: Name: Joana Nimar

Fetching authors with HINT_PASS_DISTINCT_THROUGH hint ...
Hibernate: select author0_.id as id1_0_0_, books1_.id as id1_1_1_, author0_.age as age2_0_0_, author0_.genre as genre3_0_0_, a
uthor0_.name as name4_0_0_, books1_.author_id as author_i4_1_1_, books1_.isbn as isbn2_1_1_, books1_.title as title3_1_1_, boo
ks1_.author_id as author_i4_1_0_, books1_.id as id1_1_0_ from author author0_ left outer join book books1_ on author0_.id=bo
oks1_.author_id
Id: 1: Name: Joana Nimar

```

Source code can be found [here](#).

## Item 14: Enable Dirty Tracking

Java Reflection is considered slow and, therefore, a performance penalty.

**Description:** Prior to Hibernate version 5, the *dirty checking* mechanism relies on the Java Reflection API. Starting with Hibernate version 5, the *dirty checking* mechanism relies on **Bytecode Enhancement**. This approach sustains better performance, especially when you have a relatively large number of entities.

### Key points:

- Add the corresponding plugin in `pom.xml` (e.g. use Maven *Bytecode Enhancement* plugin)

### Output sample:

```

Jul 17, 2019 10:26:17 AM org.hibernate.bytecode.enhance.internal.bytebuddy.EnhancerImpl doEnhance
INFO: Enhancing [com.bookstore.entity.Author] as Entity
Successfully enhanced class [D:\BPSB\GitHub\Hibernate-SpringBoot\HibernateSpringBootEnableDirtyTracking\target\classes\com\boo
kstore\entity\Author.class]

```

- The bytecode enhancement effect can be seen on `Author.class`, [here](#).

Source code can be found [here](#).

## Item 15: Use Java 8 Optional in Entities and Queries

Treating Java 8 `Optional` as a "silver bullet" for dealing with nulls can cause more harm than good. Using things for what they were designed is the best approach. A detailed chapter of good practices for `Optional` API is available in my book, *Java Coding Problems*.

**Description:** This application is a proof of concept of how is correct to use the Java 8 `Optional` in entities and queries.

### Key points:

- Use the Spring Data built-in query-methods that return `Optional` (e.g. `findById()` )
- Write your own queries that return `Optional`
- Use `Optional` in entities getters
- In order to run different scenarios check the file, `data-mysql.sql`

Source code can be found [here](#).

## Item 16: How to Correctly Shape an @OneToMany Bidirectional Relationship

There are a few ways to screw up your `@OneToMany` bidirectional relationship implementation. And, I am sure that this is a thing that you want to do it correctly right from the start.

**Description:** This application is a proof of concept of how is correct to implement the bidirectional `@OneToMany` association.

### Key points:

- **Always** cascade from parent to child
- Use `mappedBy` on the parent
- Use `orphanRemoval` on the parent in order to remove children without references
- Use helper methods on the parent to keep both sides of the association in sync
- **Always** use lazy fetch
- As entities identifiers, use assigned identifiers (business key, natural key ( `@NaturalId` )) and/or database-generated identifiers and override (on child-side) properly the `equals()` and `hashCode()` methods as [here](#)
- If `toString()` needs to be overridden, then pay attention to involve only the basic attributes fetched when the entity is loaded from the database

**Note:** Pay attention to remove operations, especially to removing child entities. The `CascadeType.REMOVE` and `orphanRemoval=true` may produce too many queries. In such scenarios, relying on *bulk* operations is most of the time the best way to go for deletions. Check this out.

Source code can be found [here](#).

## Item 17: JPQL Query Fetching

When *direct fetching* is not an option, we can think of JPQL query fetching.

**Description:** This application is a proof of concept of how to write a query via JpaRepository , EntityManager and Session .

### Key points:

- For JpaRepository, use @Query or Spring Data Query Creation
- For EntityManager and Session, use the createQuery() method

Source code can be found [here](#).

## Item 18: MySQL and Hibernate 5 Avoid AUTO Generator Type

In MySQL, the TABLE generator is something that you will **always** want to avoid. Never use it!

**Description:** In MySQL and Hibernate 5, the GenerationType.AUTO generator type will result in using the TABLE generator. This adds a significant performance penalty. Turning this behavior to IDENTITY generator can be obtained by using GenerationType.IDENTITY or the *native generator*.

### Key points:

- Use GenerationType.IDENTITY instead of GenerationType.AUTO
- Use the *native generator* exemplified in this source code

### Output sample:

#### GenerationType.AUTO (IDENTITY)

```
insert into author_good (age, genre, name) values (?, ?, ?)
```



#### GenerationType.AUTO (TABLE)

```
select next_val as id_val from hibernate_sequence for update
update hibernate_sequence set next_val= ? where next_val=?
insert into author_bad (age, genre, name, id) values (?, ?, ?, ?)
```



Source code can be found [here](#).

## Item 19: Redundant save() Call

We love to call this method, don't we? But, calling it for managed entities is a bad idea since Hibernate uses Hibernate *dirty checking* mechanism to help us to avoid such redundant calls.

**Description:** This application is an example when calling save() for a managed entity is redundant.

### Key points:

- Hibernate triggers UPDATE statements for managed entities without the need to explicitly call

- Hibernate triggers UPDATE statements for managed entities without the need to explicitly call the `save()` method
- Behind the scenes, this redundancy implies a performance penalty as well (see here)

Source code can be found here.

## Item 20: PostgreSQL (BIG)SERIAL and Batching Inserts

In PostgreSQL, using `GenerationType.IDENTITY` will disable insert batching.

**Description:** The ( BIG ) SERIAL is acting "almost" like MySQL, `AUTO_INCREMENT`. In this example, we use the `GenerationType.SEQUENCE`, which enables insert batching, and we optimize it via the `hi/lo` optimization algorithm.

### Key points:

- Use `GenerationType.SEQUENCE` instead of `GenerationType.IDENTITY`
- Rely on the `hi/lo` algorithm to fetch a *hi* value in a database roundtrip (the *hi* value is useful for generating a certain/given number of identifiers in-memory; until you haven't exhausted all in-memory identifiers there is no need to fetch another *hi*).
- You can go even further and use the Hibernate `pooled` and `pooled-lo` identifier generators (these are optimizations of `hi/lo` that allows external services to use the database without causing duplication keys errors). Check this out! And this!

### Output sample:

```
Hibernate: select nextval ('sequence')
Hibernate: insert into author (age, genre, name, id) values (?, ?, ?, ?)
Hibernate: insert into author (age, genre, name, id) values (?, ?, ?, ?)
Hibernate: insert into author (age, genre, name, id) values (?, ?, ?, ?)
Hibernate: insert into author (age, genre, name, id) values (?, ?, ?, ?)
Hibernate: insert into author (age, genre, name, id) values (?, ?, ?, ?)
Hibernate: insert into author (age, genre, name, id) values (?, ?, ?, ?)
2019-08-05 11:28:32.680 INFO 6740 --- [main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    460856 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    3516290 nanoseconds spent preparing 2 JDBC statements;
    6583452 nanoseconds spent executing 1 JDBC statements;
    3762601 nanoseconds spent executing 2 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    22024337 nanoseconds spent executing 1 flushes (flushing a total of 6 entities and 0 collections);
    0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```

Source code can be found here.

## Item 21: JPA Inheritance — Single Table

JPA supports `SINGLE_TABLE`, `JOINED`, `TABLE_PER_CLASS` inheritance strategies. Each of them has its pros and cons. For example, in the case of `SINGLE_TABLE`, reads and writes are fast, but as the main drawback, *NOT NULL* constraints are not allowed for columns from subclasses.

**Description:** This application is a sample of JPA Single Table inheritance strategy ( `SINGLE_TABLE` )

**Key points:**

- This is the default inheritance strategy ( `@Inheritance(strategy=InheritanceType.SINGLE_TABLE)` )
- All the classes in a hierarchy are mapped to a single table in the database
- Subclasses attributes non-nullability is ensured via `@NotNull` and MySQL triggers
- The default discriminator column memory footprint was optimized by declaring it of type `TINYINT`

**Output example:**

dtype	id	isbn	title	format	size_in	weight_lbs	author_id
1	1	001-AT	The book of swords	NULL	NULL	NULL	1
3	2	002-AT	The beatles anthology	NULL	7.5 x 1.3 x 9.2	2.7	1
2	3	003-AT	Anthology myths	kindle	NULL	NULL	1

Source code can be found [here](#).

You may also like:

[JPA Inheritance - JOINED](#)

[JPA Inheritance - TABLE\\_PER\\_CLASS](#)

[JPA Inheritance - @MappedSuperclass](#)

## Item 22: How to Count and Assert SQL Statements

Without counting and asserting SQL statements, it is very easy to lose control of the SQL executed behind the scene and, therefore, introduce performance penalties.

**Description:** This application is a sample of counting and asserting SQL statements triggered "behind the scenes." It is very useful to count the SQL statements in order to ensure that your code is not generating more SQLs that you may think (e.g., N+1 can be easily detected by asserting the number of expected statements).

**Key points:**

- For Maven, in `pom.xml`, add dependencies for `DataSource-Proxy` and Vlad Mihalcea's `db-util`
- Create the `ProxyDataSourceBuilder` with `countQuery()`
- Reset the counter via `SQLStatementCountValidator.reset()`
- Assert `INSERT`, `UPDATE`, `DELETE`, and `SELECT` via `assertInsert{Update/Delete/Select}Count(long expectedNumberOfSql)`

**Output example (when the number of expected SQLs is not equal with the reality an exception is thrown):**

```
com.vladmihalcea.sql.exception.SQLInsertCountMismatchException: Expected 0 statements but recorded 1 instead!
```

Source code can be found [here](#).

## Item 23: How To Use JPA Callbacks

Don't reinvent the wheel when you need to tie up specific actions to a particular entity lifecycle event. Simply rely <https://dzone.com/articles/90-new-features-and-apis-in-jdk-11>

on built-in JPA callbacks.

**Description:** This application is a sample of enabling the JPA callbacks ( `Pre / PostPersist` , `Pre / PostUpdate` , `Pre / PostRemove` , and `PostLoad` ).

#### Key points:

- In the entity, write callback methods and use the proper annotations
- Callback methods annotated on the bean class must return `void` and take no arguments

#### Output sample:

```
@PrePersist callback ...
    insert into author (age, genre, name) values (?, ?, ?)
@PostPersist callback ...
    select author0_.id as id1_0_0_, author0_.age as age2_0_0_, author0_.genre as genre3_0_0_, ...
@PostLoad callback ...
@PreUpdate callback ...
    update author set age=?, genre=?, name=? where id=?
@PostUpdate callback ...
@PreRemove callback ...
    delete from author where id=?
@PostRemove callback ...
```

Source code can be found [here](#).

## Item 24: @OneToOne and @MapsId

**Description:** Instead of *regular* unidirectional/bidirectional `@OneToOne` better rely on a unidirectional `@OneToOne` and `@MapsId` . This application is a proof of concept.

#### Key points:

- Use `@MapsId` on the child side
- Use `@JoinColumn` to customize the name of the *primary key* column
- Mainly, for `@OneToOne` associations (unidirectional and bidirectional), `@MapsId` will share the *primary key* with the parent table (id property acts as both primary key and foreign key)

**Note:** `@MapsId` can be used for `@ManyToOne` as well.

Source code can be found [here](#). A detailed dissertation is available in my book, **Spring Boot Persistent Best Practices**.

## Item 25: DTOs Via SqlResultSetMapping

Fetching more data than needed is bad. Moreover, fetching entities (add them in the Persistence Context) when you don't plan to modify them is one of the most common mistakes that draw implicitly performance penalties. Items 25-32 show different ways of extracting DTOs.

**Description:** Using DTOs allows us to extract only the needed data. In this application, we rely on `SqlResultSetMapping` and `EntityManager` .

#### Key points:

- Use `SqlResultSetMapping` and `EntityManager`
- For using Spring Data Projections, check issue number 9 above.

Source code can be found [here](#).

Stay tuned for our next installment where we explore the remaining top 25 best performance practices for Spring Boot 2 and Hibernate 5!

If you liked the article, you might also like my books:

Java Persistence Performance Illustrated Guide

**Spring Boot Persistence Best Practices** (coming soon)

See you in part 2!

---

The growing popularity of IoT, sensor networks, and other telemetry applications lead to the vast amounts of time series data. Learn how to automate large scale processing of thousands of series and millions of data points [in this on-demand webinar](#).

Presented by InfluxData

---

## Like This Article? Read More From DZone

**Best Performance Practices for  
Hibernate 5 and Spring Boot 2 (Part  
3)**

**How to Create a REST API With  
Spring Boot**

**Best Performance Practices for  
Hibernate 5 and Spring Boot 2 (Part  
2)**

**Free DZone Refcard  
Getting Started With Headless CMS**

Topics: HIBERNATE 5, PERSISTENCE, JAVA, SPRING DATA, SPRING BOOT, TUTORIAL, PERFORMANCE, SPRING BOOT 2

Opinions expressed by DZone contributors are their own.