

# Introduction to Hessian4J

Roger Laenen

Bruno Ranschaert

June 2, 2008

## Contents

<b>1</b>	<b>What is Hessian?</b>	<b>1</b>
<b>2</b>	<b>What's it good for?</b>	<b>2</b>
<b>3</b>	<b>Basic concepts</b>	<b>2</b>
<b>4</b>	<b>The serialization phase</b>	<b>4</b>
4.1	What's a "helper"?	4
4.2	What's a "namer"?	5
4.3	Annotations	6
<b>5</b>	<b>The renderer</b>	<b>6</b>
<b>6</b>	<b>The parser</b>	<b>6</b>
<b>7</b>	<b>The deserializer</b>	<b>7</b>
<b>8</b>	<b>A simple example showing it all</b>	<b>7</b>
<b>9</b>	<b>What next ?</b>	<b>8</b>

## 1 What is Hessian?

In a nutshell, Hessian is a binary data transfer protocol with the following characteristics:

- It is completely self describing, class definitions are contained within the data stream. The stream is self contained, no separate definition files are needed.
- It is language agnostic, various language implementations currently exist.
- It is compact in comparison with the hyped XML<sup>1</sup> stuff. XML was invented to annotate documents whereas Hessian was invented to represent binary data in a standard way.
- It's a stateless protocol.

All grants for the protocol go to Caucho<sup>2</sup> who defined the protocol and submitted it in 2007 as an IETF<sup>3</sup> draft. This article doesn't cover the internals of the protocol, more information on this can be found on the Caucho website itself.

The Java implementation discussed in this document, named Hessian4J, differs from the Caucho implementation in 2 ways :

---

<sup>1</sup>Extensible Markup Language. See <http://en.wikipedia.org/wiki/XML>.

<sup>2</sup><http://www.caucho.com/resin-3.0/protocols/hessian.xtp>

<sup>3</sup><http://www.ietf.org>

- It only supports the more recent (and more compact) version 2 of the protocol.
- The transformation from Java objects to the Hessian wire data is performed through an intermediate model.

This last item was our primary motivation to create an alternative implementation. The existence of an intermediate Hessian model permits processing of the data (transformation, validation, routing, ...) without requiring the existence of the associated Java business model. If you are creating a web interface for example, all classes used in the definition of the interface have to be available on the client and on the server. An intermediate agent is now able to read and parse the Hessian stream without having to reconstruct the business model. The intermediate model in our Hessian implementation is what the DOM<sup>4</sup> model is for XML.

## 2 What's it good for?

Hessian is a replacement for XML when you need something fast and simple while direct access to the data (using a text editor) is less important. It is fast because it was designed that way, the Hessian format allows very fast parsers. It is simpler than XML, compact and complete implementations can be found for many languages.

Basically you can use it to stream any data to a file or over the network. Because Hessian is a very compact protocol most benefit will be gained in high load environments. In a RESTful<sup>5</sup> environment you could use Hessian as the payload of your HTTP-POST request. Specifically for this purpose there is a separate draft proposal submitted to the IETF by Caucho. Examples where Hessian can be used:

- Persist data to a file.
- Use it to call a network service.
- Use it to post messages on an ESB.

## 3 Basic concepts

As mentioned before the Hessian4J implementation transforms your Java model to the Hessian format through an intermediate model. This is illustrated in Figure 1. Translating this to the software architecture we get the following :

- `HessianSerializer` class which does the serializing and deserializing. The terms "serialize" and "deserialize" mean the conversion between your Java object model and the Hessian object model.
- `HessianParser` class which parses the Hessian data stream to the Hessian model. The term "parse" means to read something in Hessian and convert it to the Hessian object model.
- The Hessian model classes that are able to render themselves to the Hessian wire data format. The term "render" means to write a Hessian object representation to a Hessian encoded stream.

The Hessian model in fact is a hierarchical Java representation of the Hessian data types. Figure 2 illustrates a part of its basic structure.

---

<sup>4</sup>Document Object Model. See [http://en.wikipedia.org/wiki/Document\\_Object\\_Model](http://en.wikipedia.org/wiki/Document_Object_Model).

<sup>5</sup>Representational state transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. See [http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer)

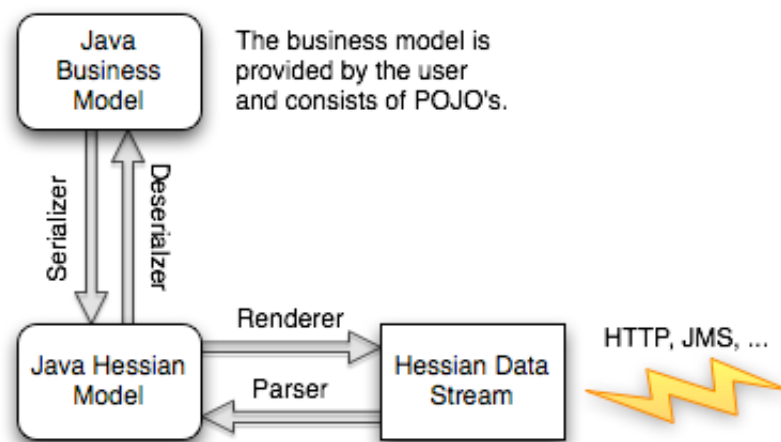


Figure 1: 2-phase transformation

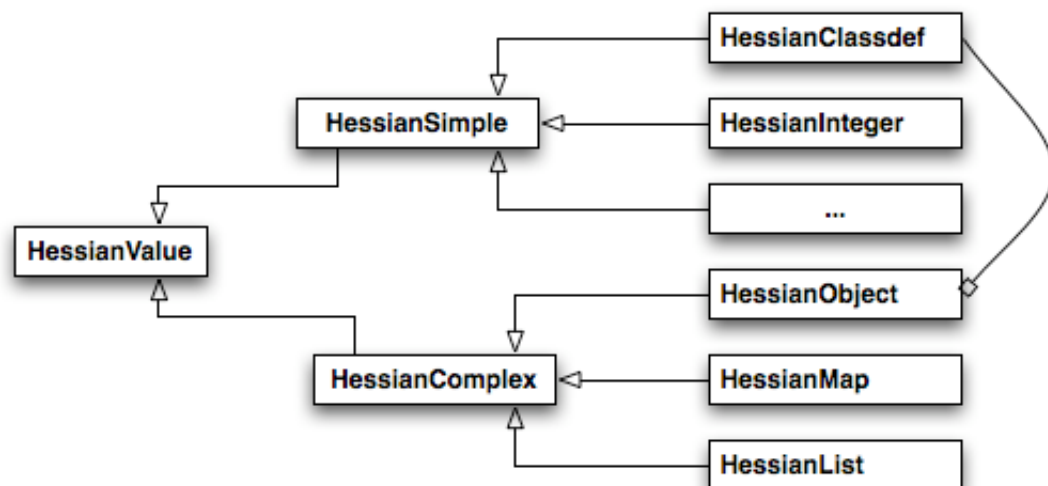


Figure 2: Structure of the Hessian model

## 4 The serialization phase

Let's first discuss the serialization from a Java object to its Hessian model representation. This process is driven by the `HessianSerializer`. The serializer performs its task with the assistance of a set of hierarchically structured helpers. A so-called helper knows how to (de)serialize a specific class to/from the Hessian model. The `HessianSerializer` contains a repository with a predefined set of helper classes. You can add your own helper class to this repository if needed. It is the responsibility of these helpers to convert your Java objects to a corresponding Hessian model and vice-versa. Let's look at the following example:

```
1 HessianSerializer lSer = new HessianSerializer();
2 HessianValue lVal = lSer.serialize(myObject);
```

That's all there is. On line 1 a serializer is created. On line 2 your Java object is transformed to a Hessian representation and is ready to be rendered to a stream.

### 4.1 What's a "helper"?

Like we said before, you can add your own helper to the serializer repository. One of the most common cases you will need to do this is when you are dealing with Java classes that have no default constructor. Normally, the deserializer will instantiate the Java object using reflection to call the default constructor. For classes without a default constructor you have to write a Helper class that tells the serializer how to serialize and deserialize objects of it. Another common use of Helpers is where you want to transform your object (eg limit the number of attributes, convert a type..) before putting it on the wire. To give you an idea of the use of these helpers, take the following snippet:

```
1 final HessianSerializer lSerializer = new HessianSerializer();
2 lSerializer.getRepo().addHelper(new BigDecimalHelper());
3 HessianValue lVal = lSerializer.serialize(new BigDecimal("10.00"));
4 System.out.println(lVal.prettyPrint());
5 BigDecimal lResult = (BigDecimal) lSerializer.deserialize(lVal);
```

We (de)serialize a `BigDecimal` here. `BigDecimal` is one of those classes without a default constructor, that's where our `BigDecimalHelper`<sup>6</sup> comes in. What the helper basically does is converting our Java `BigDecimal` representation to a `HessianObject` with a `HessianString` argument. The value of this argument is the `toString()` value of the original `BigDecimal`. When consequently deserializing, the same helper will call the `BigDecimal(String)` constructor with the value of the string argument. Basically, the real (de)serialization work is done within the Helper classes, the `HessianSerializer` mainly keeps the context (more on this later).

As you notice on line 4, you can always `prettyPrint()` a `HessianValue` to inspect the Hessian representation.

```
1 <classdef#0:'java.math.BigDecimal'>
2 [0] val
3
4 <object@0:'java.math.BigDecimal'#0>
5 [0] <string:"10.00">
```

The dump shows that the Hessian data contains a Hessian class definition on line 1. The index of the classdefs as they are written to the stream is mentioned in the dump: "#0". The classdef contains a single field named "`val`" which is written out on line 2. The dump contains a single object on line 4 of the classdef. The only field contains a value of type string which is a textual representation of a `BigDecimal`.

---

<sup>6</sup>This helper is defined by default in the repository.

When writing your own helper, there is a generic `GenObjectHelper` class you can extend from or you can make your own from scratch by implementing the `HessianHelper` interface. When (de)serializing Java objects that don't have a specific Helper class we fall back to one of the 2 generic object helpers, `ObjectHelper` or `ObjectHelperDirect`. The difference between the 2 is the approach they take to read/write the attributes. `ObjectHelper` uses the JavaBean approach and as such calls the getters/setters of the object. The `ObjectHelperDirect` implementation on the contrary uses reflection to directly access your attributes. By calling the `setFieldAccess()` method on the `HessianSerializer` you can switch between these 2. By default, direct field access is used.

## 4.2 What's a "namer"?

Besides the Helpers another facility that is used by the (de)serializer is the Namer. The Namer translates between Java class names and corresponding Hessian names and vice-versa. This can be useful if the consumer of your data uses another naming convention or uses another programming language altogether. The type names and class names in the Hessian model do not have to correspond to Java class names. The namers allow you to replace the automatically generated Java names with your own names. Namers are chained, when serializing each class-name encountered is passed to the namer chain.

The following example illustrates this :

```
1 public class NamerSample
2 {
3     public static class BigDecimalNamer
4     implements Namer
5     {
6         public String mapHessian2Java(String aHessianName)
7         throws HessianSerializerException
8         {
9             if ("FOO".equals(aHessianName))
10                 return "java.math.BigDecimal";
11             return aHessianName;
12         }
13
14         public String mapJava2Hessian(String aJavaName)
15         throws HessianSerializerException
16         {
17             if ("java.math.BigDecimal".equals(aJavaName))
18                 return "FOO";
19             return aJavaName;
20         }
21     }
22
23     public static void main(String[] args)
24     throws HessianSerializerException
25     {
26         final HessianSerializer lSerializer = new HessianSerializer();
27         lSerializer.addNamer(new BigDecimalNamer());
28         HessianValue lVal = lSerializer.serialize(new BigDecimal("10"));
29         System.out.println(lVal.prettyPrint());
30
31         BigDecimal bd = (BigDecimal)lSerializer.deserialize(lVal);
32     }
33 }
```

In this sample we serialize a `BigDecimal` as class FOO and deserialize it again to a `BigDecimal`. One example of the use of namers is the `WipeHibernateNamer`. When working with Hibernate

you don't always want to transfer the Hibernate specific collection classes to your client application. In that case, you can add the `WipeHibernateNamer` which will translate the respective collection names to their Java equivalents.

## 4.3 Annotations

When talking about helpers we mentioned the usage for classes without a default constructor. However, when you have full control over the class there is a simpler way to come around this problem. Hessian4J provides 2 annotations for this purpose, namely `@HessianConstruct` and `@HessianSerialize`.

The method annotated with `HessianSerialize` should return an array of objects that can be passed (in the given order) to the constructor annotated with the `HessianConstruct` annotation. Using this approach avoids having to write a Helper class. The following snippet illustrates this :

```
1 public static class MyDate
2 {
3     private Date theDate;
4     private String theTimeZone;
5
6     @HessianConstruct
7     public MyDate(long aTime, String aTimeZone)
8     {
9         theDate = new Date(aTime);
10        theTimeZone = aTimeZone;
11    }
12
13    @HessianSerialize
14    public Object[] getTime()
15    {
16        return new Object[] {theDate.getTime(), theTimeZone};
17    }
18 }
```

## 5 The renderer

There is no real Hessian renderer, instead each Hessian model class is able to render itself to the Hessian wire protocol format by calling its `render()` method. The result of the rendering is put in the `OutputStream` you pass to it. If you want to play a bit around with it, then just pass it a `FileOutputStream` and use your favorite hexadecimal editor to view the Hessian data.

Besides the `OutputStream` the render method expects another 3 list-arguments, 'types', 'class-defs' and 'objects'. A renderer uses these to make references to classes or objects that have already been rendered in the same context. The types list is used for to store the type of maps or lists (eg `ArrayList`, `LinkedList`, ...) in order to be able to reference it once the same collection type occurs more then once in the data stream.

## 6 The parser

The `HessianParser` sequentially reads and parses the given Hessian input stream to transform it into a `HessianValue`. It's a one-pass parser which makes the parsing very fast. Note that the parser is not thread-safe (parsing information is stored internally), so every parsing thread should instantiate its own parser.

The use of the parser is very straightforward, instantiate it giving it an `InputStream` and call the `nextValue()` method to parse the first available data element in the stream.

```

1 HessianParser lMyParser = new HessianParser(new FileInputStream(tempFile));
2 final HessianValue lMyString = lMyParser.nextValue();

```

## 7 The deserializer

The deserialization takes the same approach as the serialization, it uses the same mechanism of namers and helpers to convert your Hessian value to the corresponding Java object.

Just invoke it by calling the `deserialize()` method passing it your `HessianValue` like in the following example :

```

1 Order lOrder2 = (Order) serializer.deserialize(aHessianValue);

```

## 8 A simple example showing it all

To summarize this quick start, consider the following snippet, which serializes and renders an order to a `ByteArrayOutputStream` and subsequently parses and deserializes it again to its Java representation.

```

1 public class OrderSample
2 {
3     public static class Order
4     {
5         private Date orderDate = new Date();
6         private BigDecimal totalAmt = BigDecimal.ZERO;
7         private Set<OrderLine> orderLines = new HashSet();
8
9         public void addOrderLine(OrderLine aLine)
10        {
11            orderLines.add(aLine);
12            totalAmt = totalAmt.add(
13                aLine.unitAmount.multiply(
14                    new BigDecimal(aLine.quantity)));
15        }
16    }
17
18    public static class OrderLine
19    {
20        private String description;
21        private int quantity;
22        private BigDecimal unitAmount;
23
24        public OrderLine() {}
25
26        OrderLine(String aDescription, int aQuantity, BigDecimal anAmount)
27        {
28            description = aDescription;
29            quantity = aQuantity;
30            unitAmount = anAmount;
31        }
32    }
33
34    public static void main(String[] args)
35    throws HessianSerializerException,
36           HessianRenderException,
37           HessianParserException

```

```

38 {
39     Order lOrder = new Order();
40     lOrder.addOrderLine(new OrderLine("firstLine",1,new BigDecimal("10")));
41     lOrder.addOrderLine(new OrderLine("secondLine",5,new BigDecimal("5")));
42
43     HessianSerializer serializer = new HessianSerializer();
44     HessianObject lVal = (HessianObject)serializer.serialize(lOrder);
45
46     final ByteArrayOutputStream lOut = new ByteArrayOutputStream();
47     lVal.render(lOut);
48     System.out.println(lVal.prettyPrint());
49
50     HessianParser lParser = new HessianParser(
51         new ByteArrayInputStream(lOut.toByteArray()));
52     HessianValue lVal2 = lParser.nextValue();
53
54     Order lOrder2 = (Order) serializer.deserialize(lVal2);
55 }
56 }

```

## 9 What next ?

This concludes this quick guide to Hessian4J. In another issue we will cover the Hessian web service specification which defines the use of Hessian in a web service context. The current implementation provides already the basic elements as defined in the draft<sup>7</sup>. Please download the software (its licensed as LGPL<sup>8</sup>) and try it out. All feedback, positive or negative, is welcome. Currently we're working on integration with the Spring framework for a next release.

<sup>7</sup><http://hessian.caucho.com/doc/hessian-ws.html>

<sup>8</sup>GNU Lesser General Public License. See more at <http://www.gnu.org/licenses/lgpl.html>. In short, you can use it for commercial and non-commercial purposes, as long as you distribute the license text of Hessian4J as well.