BONAPARTE - Binary Object Notation As Portable And Readable Text Extension*

Michael Bischoff

May 26, 2013

Abstract

Bonaparte is an Eclipse XText based domain specific language (DSL), with the purpose to generate code for fast object serialization / deserialization. There is a preferred serialization format (the "Bonaparte format"), but the DSL itself is independent of the serialization format. The Bonaparte serialization format combines features of CSV, JSON, Java serialization and Google protocol buffers.

1 Why another serialization format?

Object serialization is an important function in today's distributed computing environments. No matter if you want to exchange synchronous or asynchronous messages in a service-oriented modular software environment, persist arbitrary data in some NoSQL database, use distributed caches, or just dump data to a file. Because it's used in so many places, often transparent to the application, serialization / deserialization (or marshalling /unmarshalling) must be fast. In addition, a compact presentation is important, because it helps reducing required storage space and network traffic. Many implementations sacrifice readability or portability for size. Others (like XML) emphasize readability but neglect the message size.

The Bonaparte message format was designed with the following goals in mind:

- Small message size without sacrifizing portability (computer byte order / programming language independence)
- fast serialization / deserialization
- good support for typical data types found in nowadays programming languages
- support of object inheritance
- extensibility of the message format
- compatibility of serialized formats if the underlying objects are changed

Normally, Bonaparte serialized formats are defined as UTF-8 encoded character byte streams. Using an extension mechanism or via external setting of the encoding, other encodings are also supported (provided they cover all required code points of transmitted text fields). In fact, the serialized format can be converted between UTF-8 and other formats during transmission and parsed in a different encoding.

*SHIPANWABE - Should Have I Picked Another Name With A Better Expansion?

2 Why a specialized DSL?

The key driver for the development of the DSL was the desire to have a notation which is independent of the target language and also very compact. The advantage of a DSL (and especially DSLs based on Eclipse XText) is, that you can get full syntax checking and syntax highlighting while entering the code (this of course has the prerequisite that you use the Eclipse IDE). In addition, DSLs offer the most compact notation for a given purpose. If you don't like boilerplate code, DSLs are for you!

The grammar elements are

- packages / modules
- type definitions (C / C++ developers miss those in Java)
- classes / objects (including enums)
- data fields / member variables

The grammar of the DSL is loosely derived from the Java language syntax.

At the moment, the only open source code generator is for the Java programming language. A generator for Google DART is planned as soon as dynamic class loading is supported by DART.

Generic key properties are defined in the core DSL. The DSL can be extended to provide additional plausibility checks / restrictions, additional language components (keywords, annotations) or output formats. Thus, the core DSL can be kept clean of aspects used for exotic languages only, for example restrictions on field name lengths (SQL, COBOL), or annotations specifying alternate names (as in JPA).

In addition to the code generator, for every supported language a basic library providing one or more implementations of support functions for the marshaller / unmarshaller are provided.

Lastly, adapters to use the functionality in commonly used libraries are provided.

3 Version compatibility

Software systems are a volatile world. New features require extensions of the data model on a regular base. In distributed environments, new releases cannot be deployed on all nodes at the same time. This means that some nodes run the older software release and must cooperate with nodes using the new software release already. Even in single node environments, where objects have been persisted using an older release, and will be read back into memory with the new software release, we require interoperability of different versions. Therefore, version compatibility is one of the core concepts of Bonaparte.

Bonaparte distinguishes two kinds of object changes.

3.1 Compatible changes

A compatible change is defined as a change of the object / class definition which allows straightforward deserialization of an object serialized with the older version. For the Bonaparte format, the following object changes are compatible:

- Extension of the maximum length of a field. Every field is stored using its actual size only, not its maximum size. Therefore, length increases do not cause any issues.
- Adding optional (nullable) fields at the end of an object. If an "end of object" code is encountered during the parse process, while an additional object is expected, the object is interpreted as if it had been stred with a null value.

- Certain type conversions. Conversions from unsigned to signed numbers, conversions from ascii to unicode types, or conversion of a boolean to a numeric type or from any numeric data type to an ascii or unicode type is possible. This is possible, because alphanumeric types are not enclosed in quotes in the serialized format.
- Changing a required field to optional / nullable.
- Renaming a field (unless meta data is used, but in that case, it is typically used to dynamically create the receiving objects anyway, which implies that both sides will be in sync).

The reverse will usually cause parsing errors. It is planned as a future extension to allow configuration of the parser to accept the following reverse changes:

- Reduction of maximum length of alphanumeric fields (implemented by truncation of data where the data length exceeds the maximum allowed field length).
- Ignoring extra fields at the end of an object. (Null fields at the end of an object will be ignored by default)

3.2 Incompatible changes

Sometimes, an incompatible change cannot be avoided (adding required fields, cleaning up by removing no longer used fields). A parser encountering an incompatible change will break. Therefore, the concept of object revisions (version numbers) is part of the grammar definition. The revision can be seen as a second part of an object's class name. A later release will add functionality to provide parsers for all current object revisions, and allow manually written converters to be hooked into, such that all object revisions will be converted to the latest revision.

4 The Bonaparte serialization format in detail

The Bonaparte serialization format supports the following specific features. We believe that no other currently existing format supports all of them.

- Smart escaping: Instead of just prepending escape characters to characters of special meaning (with the effect that some occurences will be escape characters, some others not, like in CSV (,) or SQL ('')), escape characters or other special characters will always have their special meaning, while payload characters of this value will be represented by tokens (such as in markup languages). In this case, due to the limited number of special characters, the token will be a single character.
- Object oriented features: Whenever an instance of a specific class is serialized, the grammar can specify if exactly that class, or also any subclass can be (de-)serialized instead.
- Data domains. The implemention can provide a mapping of HQONs (Half Qualified Object Names) to class names.
- Auto-detection of encoding conversions (for certain formats: one single-byte, one multi-byte) => use the EURO currency sign as criteria: ISO 8859-15, UTF-8? (Future feature, requires extension parsing)
- DSL to define message format without boilerplate code
- Object versioning for incompatible changes (a high level concept exists at the moment, implementation is planned as a future extension)

- Smart metadata handling. Some formats do not store metadata at all, some accompany parts of it with every record (field names for JSON / XML). Bonaparte support configurable metadata sending at transmission begin, or lazyly, once an object is required. The metadata contains enough information to allow the dynamic creation of the data classes and is sent as data objects defined in the "meta" package.

- Generics support. At the moment, it's the same halfhearted approach as in Java (namely, with type erasure), but the plan is to extend it to a full-featured implementation, comparable to C# or C++'s templates.

5 Description of the format

A serialized object in Bonaparte fromat is a series of bytes, where text fields (strings) are represented by their UTF-8 encoded presentation, unless they contain control characters, which are the only characters requiring escaping.

Bonaparte can directly support the following elementary data types:

- booleans. A boolean is either true (represented by the digit 1) or false (represented by the digit 0).

- numbers. A number is of the pattern [-]0-9*[.]0-9*[e[-]0-9+]. The Bonaparte DSL allows to distinguish between int, long, float, double, or a specification of an integral type with a maximum number of digits.

- strings. (Either ASCII or Unicode.) A string is a sequence of characters, in message format corresponding notation. Control characters (characters with codepoint ¡ 0x20) are escaped as follows: Escape character (ctrl-C), followed by a character with codepoint 0x40 to 0x5f. The resulting character will be the one with the codepoint of the second byte, modulus 32. Often such control characters are not permitted due to business reasons, in which case the string serialized form does not rely on escaping at all. A special control character is directly converted into a space (ctrl-E), with the intent that parsers stripping leading and trailing spaces treat this as a non-strippable space character.

- Timestamp. A timestamp is represented in the following form: YYYYMMDDHHMISS(fff), or any subsequence of this which at least defines the day. The timestamp is assumed to be in UTC, Gregorian Calendar. The maximum number of sub-second fractional digits is implementation defined, implementations which see more digits presented than they support are expected to silently ignore them.

- raw data (binary bytes). Binary data is transmitted in base64 encoding.

Any elementary field is terminated by a field terminator character (ctrl-F). Any field may be empty, which means, it consists only of the field terminator character. If during parsing of an object the object termination character (ctrl-O) is found, any additional fields are assumed to be empty.

Arrays are represented by an introduction character ctrl-B, followed by an arbitrary number of objects or elementary data items, followed by the array termination character, ctrl-A. Converting a singular data item in an object into an array therefore is (from a message presentation form) an upwards compatible operation.

Objects are represented by an identifier giving their name (any valid Java identifier of up to 30 characters length is fine) and a version number (default 0). Both object ID and version number are treated as regular fields, i.e. are terminated by a field termination character.

5.1 Special characters

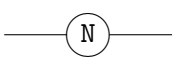
Hex	Char	Meaning	Comments
00	^@	N/A	Does not appear, because it is used as a string terminator character in some languages
01	^A	array terminator	is inserted after the last array entry. Is not followed by an additional field terminator
02	^B	array start	Immediately followed by the number of entries (0 .. n), then field terminator, then the entries itself. Used for array, List and Set data types.
03	^C	comment record	everything until the next record terminator is a comment. Escaping of special characters does apply within comments.
04	^D	unused	allows creation of input files via Unix console
05	^E	escape char	allows the transmission of control characters in String fields. Is followed by the control character, offset by 0x40 (@)
06	^F	field terminator	is sent after every atomic field
07	^G	-	
08	^H	unused	reserved for backspace while typing
09	^I	tab	if escape characters are allowed, a tab is represented via the normal tab character inside unicode fields
0A	^J	record terminator	terminates a record (in files, matches the newline character)
0B	^K	-	
0C	^L	-	
0D	^M	ignored	is used immediately ahead of the record terminator character in some operating systems, will cause an error if not immediately followed by ^J
0E	^N	NULL	used to serialise a field with null contents. This token is not followed by a field terminator. Allows distinguishing between an empty string (zero characters) and a null string.
0F	^O	former object terminator	was sent after the last field of an object has been serialized, now replaced by ^P
10	^P	parent separator	separates fields of parent object from next derived object in order to allow extensions of objects at every level. Now also sent at the end of an object.
11	^Q	quit	close the connection (optional, after transmission end)
12	^R	record start	starts every record, is followed by a version number (or blank) and a field terminator, to indicate the format version (currently always 0 / blank)

Hex	Char	Meaning	Comments
13	^S	start object	start a new (sub-) object. Is followed by the object name (with optional package prefix in Java-style dot notation (minus a configurable start package prefix)), then a field terminator, then the object's version number (empty if default / 0), and another field terminator, then the object's fields
14	^T	transmission start	starts a multi-record block, is followed by a version number (or blank) and a field terminator, to indicate the format version (currently always 0 / blank)
15	^U	transmission end	follows the last record of a transmission. Optionally followed by a ctrl-Z character
16	^V	-	
17	^W	-	
18	^X	extension start	provides optional extension information, usually sent once at the start of a transmission. Is followed by one character defining the type of the extension, then extension specific data. Currently, only E is defined (charset encoding), followed by a Euro sign, which defines either UTF-8 or ISO-8859-15 or we1252-? (standard character set converters will convert this character as well, allowing an auto-detection of the char set)
19	^Y	end of extension	is sent after extension information ends. Allows parsers to skip all (unknown) extensions.
1A	^Z	-	optionally send after the end of data transmission character. closes connection (for example for sockets).
1B	^[	-	
1C	^\		
1D	^]	-	
1E	^^	begin Map data	Starts a block immediately followed by an index type identifier (Integer / Long / String), the number of entries (0 .. n), then field terminator, then the with key / value pairs (serialized form of Java maps)
1F	^_	unused	

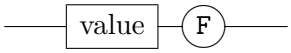
5.2 Message format railroad diagrams

In the following diagrams, a single uppercase letter inside a circle represents the single byte control code represented by the ASCII code of the letter minus 64 (0x40).

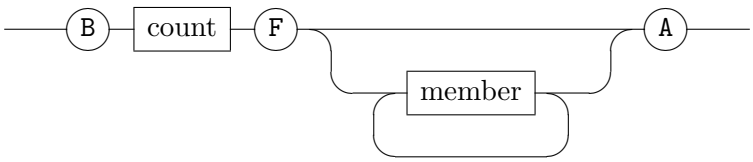
null



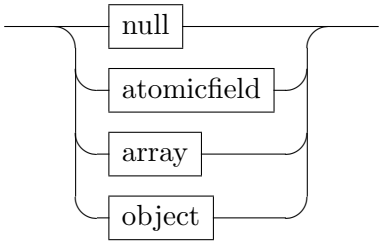
atomicfield



array

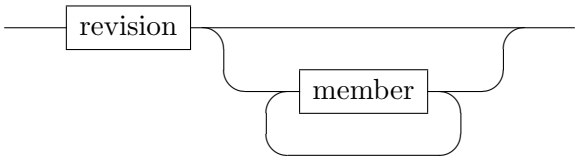


member

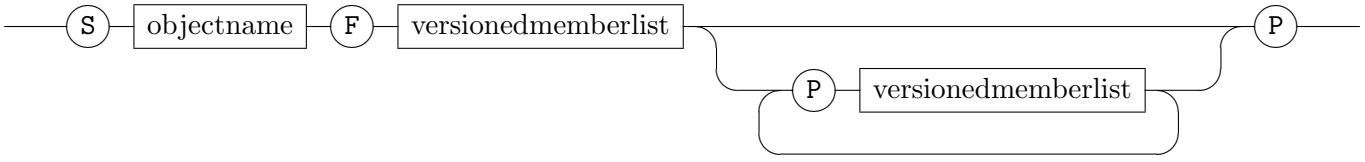


revision

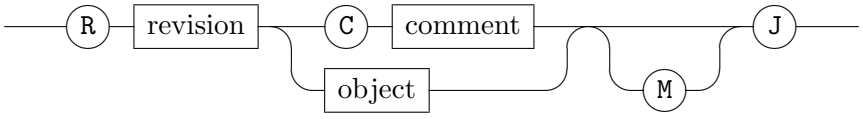
versionedmemberlist



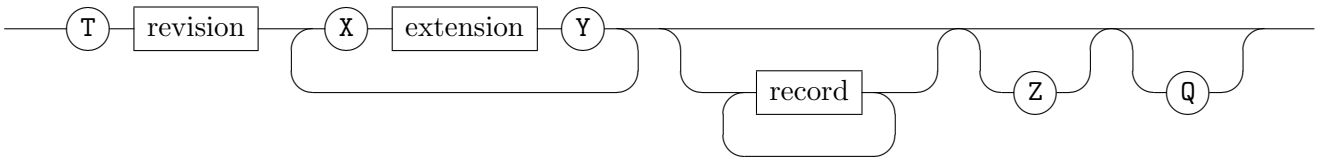
object



record



transmission



6 Message compatibility rules

In a large distributed platform, extensions of data objects frequently occur. Atbon is designed to support upgrades of platform components at different points in time. In order to support this, the "cooperative extension" model is used. This means, that some data model changes are supported, while others are not.

The changes not supported (while keeping the same version number) are:

- Swapping of field order within a class. This is due to skipping the field names from the message for sake of compactness.
- Changing a data type (some changes are allowed, see exceptions below)
- Changing a class name
- adding required (not NULLable) fields
- Modifying the inheritance structure (i.e. switching between C extends A and C extends B, B extends A)
- Changing the index type of a map

The following changes are supported:

- renaming a field within a class
- adding extra fields at the end of a class (even if this class is used as a base class and subclasses are serialized)
- changing a boolean into a number or character field
- changing an ASCII field into a UNICODE field
- removing an existing field, which is not the last field of a class.
- Swapping between array, List and Set data structures

If incompatible changes cannot be avoided, use of the "versioning feature" is required. This requires additional development at the application side.

7 Backend implementations

7.0.1 Domain / package name mapping

In order to convert between domain names and Java / Scala packages, the following rules are implemented, in order of increasing priority:

- Use of `de.jpaw.bonaparte.pojos.`*Domainname* as a package name (fallback)
- Use of a customized *packageNamePart*1.*Domainname*.*packageNamePart*2 package name (configurable by static setter function).
- Use of a Java collections `Map<String,String>`, which maps between Domain and package, using the previous rules as a fallback, if the domain has not been found in the map.

7.1 Interoperability / Message transport

The core Bonaparte project focusses on POJO serialization only, but a number of implementations are provided as separate add-on projects, which offer seamless support for different transport libraries:

7.1.1 Netty4

Bonaparte integrates well with Netty4 (currently available as alpha release only, but works very well). The bonaparte-netty project offers an API to transmit objects via Netty4. This has been tested with Netty 4.0.0.CR1

7.1.2 vert.x

This has been tested with vert.x 1.3.0

7.1.3 HornetQ (JMS API)

This has been tested with HornetQ 2.2.21 GA.

7.1.4 Apache activeMQ Apollo (STOMP API)

This has been tested only with an external Apollo server.

7.1.5 Apache Camel

This has been tested with Camel 2.10. There is an unexplained performance issue that Camel using the Bonaparte serialization is slower than Camel with GSON or Jackson, despite the serializer / deserializer of Bonaparte standalone is 2 to 5 times faster than GSON or Jackson. (It looks like the integration is not yet optimal.)

7.1.6 Akka

Akka is a very promising actor framework. Actors can communicate via the Bonaparte serialization format as shown in the bonaparte-akka sample project.

8 License

The Bonaparte DSL and the default serialization / deserialization implementation for Java are available under the Apache license, version 2 (`http://www.apache.org/licenses/`).

9 References and Prerequisites

The source of the Bonaparte DSL can be obtained here: `https://github.com/jpaw/bonaparte-dsl`.

At `https://github.com/jpaw/bonaparte-java`, there is a Java implementation of the (de)serializers for the Bonaparte message format. This repository also holds the transport integration projects.

All implementations require Java Standard Edition, version 7 (1.7.0_09 or better), the DSL is based on Eclipse 4.2.2 (Juno SR2) with XText / Xtend 2.4.1 or better (for Bonaparte-1.5.1 or later).