RFP-169-Object-Conversion

Draft

10 Pages

Abstract

Java is a type safe language that can be used to create applications that are easy to navigate in an IDE and that significantly reduce time to write tests. However, there is a tendency in Java to bypass the type system because it is often deemed easier to use strings instead of proper types: logging, JAX-RS, configuration, records, etc. This RFP investigates the issues that surrounding the use of type safe interfaces and DTOs where traditionally properties and other string based solutions are used.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

`Source code is shown in this typeface.`

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	*01/12/14*	*Initial* ***Peter.Kriens@aQute.biz***
0.1	*06/05/15*	*David Bosschaert – Small tweaks to the requirements as discussed at the Cologne F2F.*

1 Introduction

This RFP originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that needs to be solved.

Java is a type safe language that be used to create applications that are easy to navigate in an IDE and that significantly reduce time to write tests. However, there is a tendency in Java to bypass the type system because it is often deemed easier to use strings instead of proper types: logging, JAX-RS, configuration, records, etc. This RFP investigates the issues that surrounding the use of type safe interfaces and DTOs where traditionally properties and other strings are used.

2 Application Domain

Today, many programs directly interface directly with the outside world through REST, HTTP, and other protocols that are frequently require conversion from strings or byte streams. Data conversion is an inherent part of writing software in a type safe language. In Java, converting strings to proper types or to convert one type to a more convenient type is often done manually. Any errors are then handled inline.

A common problem is interacting with Javascript. Since Javascript has no user defined types, it can get away with relatively clean looking code; the same code in Java usually requires significantly more code due to the required conversions to the fine grained types of Java. This code bypasses the built-in facilities like fields and methods and instead defines constant key strings and embeds the knowledge of the types in a piece of code instead of relying on a central declaration that is then verified by the compiler.

In release 6, the OSGi Alliance introduced Data Transfer Objects (*DTOs*). DTOs are public objects without generics that only contain public fields based on simple types, arrays, and collections. In many ways DTOs replace Java beans. Java beans are hiding their fields and provide access methods but that separated the contract (the public interface) from the internal usage. Though this model has advantages in technical applications (many types, few fields) it tend to be a large overhead when there are relatively few types with lots of fields. i.e. the more common web applications. DTOs unify the specification with the data since the data is what is already public when it is sent to another process or serialized.

By limiting the allowed data types in DTOs and ensuring they have no cycles they can be easily (de)serialized using JSON, providing easy interactions with Javascript.

In enRoute, a *DTO+* is a DTO but it additionally allows many additional types and defines the rules for creating these types in a conversion.

In applications, a DTO provides the same role as a Javascript hash/object. In java, however, the fields are typed, providing type checks and content assist in the browser. However, there are similar needs to what Javascript provides when objects are used that way:

- Deep copy – Create two DTOs that are equal but do not share any instances
- Deep equals – Compare two DTOs for equality
- Shallow copy – Create a new DTO but share the fields
- Diff – Calculate the difference between 2 DTOs, providing where and why the the objects are different.
- Path based access – Provide a path based access into a DTO. E.g. a path can be `foo.4.abc`

2.1 Conversions

The Java language has the concept of a *type*. A type is defined by either a class or one of the ~~generic~~primitive types.

The first level of conversion of a *value* is limited to *simple types*. Simple types are either booleans, characters, numbers, and String. Simple types do not use generics and have no cardinality~~, they are atomic~~. For example,

converting a `String` to an `int`. In general the developer that is writing the conversion expects a specific type as input and then calls an appropriate method to do the actual conversion: `int integer = Integer.valueOf(string).` ~~Simple types are either booleans, characters, numbers, and String.~~

The second level are *cardinal* types:

- *Collection* – An enumeration of zero or more values.
- *Arrays* – An enumeration of zero or more values.
- *Map* – A mapping from one value to another value

2.2 Reflection

Java is a type safe language that provides access to the type information during runtime. This information is quite extensive and includes generic type information. Though it is impossible to know the type parameters are for an object from a generic class, the places where a generic type is used (a call, extending, method arguments, return type) actually do contain the full generic signatures.

Java does not have a built in concept to create a *type reference* for generic types since an instance does not contain the generic information it was compiled with, this information is erased. A common pattern to provide a generic type signature is to create a `TypeReference<T>` class. To create a reference, an inner subclass is created that then encodes the T in its generic signature for the super relation:

```
new TypeReference<List<Map<Pair<Integer,String>,String>>() {}
```

The TypeReference class then can inspect this information and provides the desired type information with a `Type getType()` method.

Libraries like the bnd Converter can use the reflective information to create another object of a desired type. For example:

```
byte[] barray = Converter.cnv( byte[].class, "1"); // new byte[]{1}
List<Short> shorts = Converter.cnv(
      new TypeReference<List<Short>>(){}, new String[]{"1","2"}); // [1, 2]
FooConfig fooConfig = Converter.cnv( FooConfig.class, map );
Map<String,Object> map = Converter.cnv(
      new TypeReference<Map<String,Object>>(){}, dto );
```

The bnd JSON Codec extend this model to *JSON*. JSON is a syntax to transfer data with only a limited set of types: string, booleans, numbers, arrays, and maps. In general, it is straightforward to map a DTO+ to JSON stream since Java has so much more type information than is required. However, the bnd JSON Codec can take an input stream and a DTO+ and map the JSON input stream to the fields and types defined in the DTO, recursively. Since these types contain the full generic information it is possible to support quite rich DTO+ objects.

2.3 Terminology + Abbreviations

- DTO+ – a DTO with an identity.

3 Problem Description

Experience shows clearly that leveraging the Java type system more and reducing the use of key constants and DSLs in the code can increase the productivity of developers significantly. Java is an excellent language to act as a specification language, which the huge benefit that it can be executed and is extensively supported by IDEs like Eclipse and Intellij.

The DTO model is already powerful in replacing where properties were used but requires more extensive support to match capabilities in Javascript, but then in a type safe way.

However, moving to a more type safe use of Java requires a powerful and flexible data handling that currently lacks. This RFP therefore is seeking proposals for a service that provides the following services:

- General any-to-any type conversion
- Extension to the DTO model that allows more types to be used in its fields
- Extension to the DTO that provides DTOs with an identity and if applicable comparable.
- DTO support for copying, equals, and diffing
- JSON encoding/decoding

4 Use Cases

4.1 Configuration Admin Properties Handling

Al Bundle uses Configuration Admin and gets his bundle's properties. These properties are:

```
port          “20”
host          “localhost”
debug         “true”
pattern       “.*”
```

Al has defined these property keys in an interfaces:

```
interface AlsConfig {
  int port();
  String host();
  boolean debug();
  List<Pattern> pattern();
```

```
}
```

Using the DTOs service, AI converts the properties to the interface:

```
Configuration c = cm.getConfiguration(alsPid,"?");
FooConfig config = dtos.convert( c.getProperties() ).to( FooConfig.class);
```

AI now has type safe access to its properties, any conversion errors throw exceptions.

4.2 REST API 1

The OSGi enRoute REST API uses methods to define the URI layout. The first segment of the URI is the REST processor, the next segment is the name of the endpoint service, and the remaining segments are mapped to the arguments of a method. E.g. `GET /rest/foo/widget/12` is mapped to `getWidget(WidgetRequest opts, int id )`. The `WidgetOptions` extends a standard interface to provide access to the actual servlets but is further used to define the body type for POST or PUT requests with the `_body` method's return type. Any additional methods on this interface are treated as search parameters on the URI (the part after the '?'). For example:

```
interface WidgetOptions extends RESTRequest {
  Widget _body();
  List<Glob> q();
  boolean ignore();
}
```

The implementation of this REST API must convert the last segments of the URI to the generic parameter types. The implementer uses this code (well, a bit simplified) to map the segments to the arguments:

```
Object[] mapper( Method m, List<String> segments) {
  Type types[] = m.getGenericParameterTypes()
  Object parameters[] = new Object[types.length];
  assert types.length == segments.size();

  for ( int i=0; i<types.length; i++) {
    parameters[i] = dtos.convert(segments.get(i)).to(types[i]);
  }
  return parameters;
}
```

4.3 REST API 2

In the REST API sketched in the previous use case the body was defined in the RESTRequest interface as the _body method. The implementation of this method is done through a proxy.

```
Type type = requestClass.getMethod("_body").getGenericReturnType();
if (type != null && (rq.getMethod().equalsIgnoreCase("POST")
     || rq.getMethod().equalsIgnoreCase("PUT"))) {
  Object body = dtos.encoder(type).get(rsp.getInputStream());
  args.put("_body", body);
}
```

4.4 REST API 3

AI Bundle needs to provide detailed access to a DTO over a REST interface. He defines the URI as the path into the DTO. That is, he uses an integer for an index in an array or a collection and the name of the field for a DTO or a Map. He uses the enRoute REST API so he defines a getWidget method that provides this access:

```
public Object getWidget( RESTRequest req, int id, String [] path ) {
  Widget w = getWidget(id);
```

```
    return w.get( path );
}
```

4.5 Configurable

The bnd Configurable class creates a proxy on an interface. The proxy has access to a map with the properties. The name of the method is then used as a key in the properties. Though the actual Configurable uses a more fancy name mapping, the code looks like:

```
public Object invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
  Object v = map.get(method.getName());
  return dtos.convert(v).to( method.getGenericReturnType());
}
```

5 Requirements

5.1 General

- G0010 – Provide a service that can convert any object to a given type. The specification must clearly outline what conversions are possible but must at least allow the simple types, maps, collections, and arrays.
- G0020 – Provide a type reference class
- G0030 – It must be possible to specify the destination type with a class, a generic type (Type<T>), or a type reference.
- G0040 – It must be possible to convert Strings ~~from~~to popular Java types like Pattern, File, Date, Java Date/Time, et~~.~~ al. The specification must clearly define the rules for these classes.

5.2 Maps

- M0010 – It must be possible to convert a Map or Dictionary to an interface where the method names are used as keys
- M0020 – It must be possible to convert a DTO+ to a Map<String,Object> and vice versa

5.3 DTOs

- D0005 – It must be possible to assign an identity to a DTO. This shall be referred to as a DTO+.
- D0010 – It must be possible to diff two objects of the same type returning information where the DTO+'s differ and in what way.

- D0020 – Provide a proper deepEquals that assumes DTO+
- D0030 – Provide~~s~~ a way for types to handle conversion from and to strings for non-specified types
- D0040 – Provide a way to set/get fields from a DTO+ through a string path.
- D0050 – Provide a base class for identity DTO+s
- D0060 – Provide a compare function for identity DTOs that have a primary key that is comparable
- D0070 – Provide a way to find out if a DTO+ is complex
- D0080 – Provide a way to find out an object is DTO+
- D0090 – Provide a way to verify that an object is a DTO+ and has no cycles
- D0100 – Provide a deep copy routine for a DTO+
- D0110 – Provide a shallow copy routine for a DTO+

5.4 JSON

- ~~D~~J0010 – Provide a JSON encoder and decoder that uses the conversion rules for the conversion from JSON types to destination types
- ~~D~~J0020 – JSON decoding must be able to provide a value without specifying any type for the destination
- ~~D~~J0030 – The output must be an OutputStream, ~~Writer~~Appendable, or String
- ~~D~~J0040 – The input must be an InputStream, Readable~~er~~, or String
- ~~D~~J0050 – It must be possible to pretty print the output
- J0055 – It must be possible to generate canonical, compact output
- ~~D~~J0060 – It must be possible to specify the output character set for a stream
- ~~D~~J0070 – It must be possible to specify if nulls are outputed or not
- ~~D~~J0080 – It must be possible to add hook to the conversions for custom types for encoding and decoding

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	+33 467542167
e-mail	Peter.kriens@aQute.biz

6.3 End of Document