The GWT-RPC wire protocol

Published to the Internet on December 10, 2012
Author: Brian Slesinsky
To enable commenting, join google-web-toolkit

Contents

Objective

Document the GWT-RPC wire protocol as used in GWT 2.5.

Background

GWT-RPC is a framework that many GWT apps use to communicate between a client application running in a web page and a front-end server written in Java. It's a fairly traditional RPC framework that's loosely based on Java serialization but adapted to work over HTTP. Like Java serialization, it supports serialization of arbitrary Java object graphs, including cycles, provided that they meet certain requirements that make translation between client and server-side Java feasible. The official GWT documentation explains how to use it from a developer's perspective.

Overview

Usually each GWT-RPC service appears at a separate URL. The web page makes RPC calls by sending HTTP(S) POST requests containing a UTF-8 string as the request body. The response is a string in a format similar to but not exactly JSON.

Example

Here's a request taken from the Validation sample app. (You can see these in the "Network" tab of the Chrome Developer Tools.)

```
7|0|6|http://127.0.0.1:8888/validation/|D031DD0CECD85E06AF1E383A0EC73E6E|com.
google.gwt.sample.validation.client.GreetingService|greetServer|com.google.gw
t.sample.validation.shared.Person/2669394933|Hello|1|2|3|4|1|5|5|0|6|0|A|
```

Here's the meaning of each field::
7 => The current protocol version.
0 => No flags are set.
6 => The string table contains six strings, which follow.
[string table] => Six strings that the following fields will refer to via a one-based index
1 => `http://127.0.0.1:8888/validation/` => The base URL of the GWT app.
2 => `D031DD0CECD85E06AF1E383A0EC73E6E` => The strong name of the policy file.
3 => `com.google.gwt.sample.validation.client.GreetingService` => The service interface. (Source.)
4 => `greetServer` => The name of the method to call.
1 => The method call has one parameter.
5 => `com.google.gwt.sample.validation.shared.Person/2669394933` => This is the declared type of the method's first parameter, which is needed to look up the method.
5 => (same) => This is the runtime type of the first parameter, which happens to be the same as the declared type. This type is checked against the policy file to make sure it's deserializable (the last two flags are true). Here's the entry in the policy file for Person:

```
com.google.gwt.sample.validation.shared.Person, false, false, true, true,
com.google.gwt.sample.validation.shared.Person/2669394933, 2669394933
```

The Person class has four fields which are then serialized in alphabetical order. In this case they all happen to contain primitive Java types. If a field referred to another Java object then its fields would be serialized (recursively).
0 => null => The "address" field , which is null.
6 => "Hello" => The "name" field contains a string.
0 => null => The "otherAddresses" field is null.
A => 0 in base64 => The "ssn" field's value is zero. ("A" is the first base64 character, which decodes to six zero bits.)

Here's the response:

```
//OK[2,1,["com.google.gwt.safehtml.shared.SafeHtmlString/235635043","Hello,
Hello!<br>I am running jetty-6.1.x.<br><br>It looks like you are using:
```

`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.4 (KHTML, like Gecko) Chrome/22.0.1229.94 Safari/537.4"],0,7]`

Here's the meaning:

"//" The response starts with a JavaScript comment.
"OK" indicates that the method returned success (rather than an exception).

Next is a JavaScript expression that evaluates to an array, which is written in reverse order so that JavaScript can read values via pop(). Here's the meaning of the values (starting from the end):

7 => The required serialization version.
0 => No flags are set.
[strings] => A table with two strings in it. The rest of the response will use one-based indices (in forward order) to refer to strings.
1 => `"com.google.gwt.safehtml.shared.SafeHtmlString/235635043"` => The result type.
2 => "Hello, hello..." => The value of the "html" field in SafeHtmlString.

Detailed Design

Envelope

How the client code chooses the URL

Each GWT-RPC service is represented by a client-side stub (a subclass of RemoteServiceProxy). When constructed, the stub calculates the service's default URL by appending a service-specific suffix to the GWT application's base URL. Developers typically set this suffix using the @RemoteServiceRelativePath annotation on the service interface. The GWT application may override the stub's destination URL by calling setServiceEntryPoint() on the stub and providing a different URL.

It's possible for a servlet to two implement two different services by implementing both their Java interfaces, in which case two client stubs might send RPC calls to the same URL.

How GWT calculates the base URL

A GWT application's base URL comes from a call to GWT.getModuleBaseURL(), which normally returns a URL pointing to the directory containing the GWT application's javascript files. A JavaScript function named computeScriptBase in the GWT bootstrap script finds this directory. The default script tries various strategies (see the code) but normally it searches the HTML page for a <script> tag pointing to the GWT application's bootstrap script, which has a URL ending with ".nocache.js". The computeScriptBase function may be replaced depending on GWT linker options.

How the frontend server routes requests to a GWT-RPC service

GWT developers must configure their web frontend to accept requests at the appropriate URL for each service. How configuration happens varies, but typically in a Java web app, the developer adds entries to a web.xml file to bind each subclass of RemoteServiceServlet to its expected URL.

HTTP(S) Requirements

- Must be a POST request. (Handling starts in AbstractRemoteServiceServlet.doPost)
- Unless overridden, the Content-type must be "gwt/x-gwt-rpc; charset=utf-8". (Checked in RPCServletUtils.readContentAsGwtRpc.)
- The request must have a header named "X-GWT-Permutation". This is a partial guard against XSRF attacks. (Checked in RemoteServiceServlet.processCall.)
- The post body must be a non-empty UTF-8 string.

Request format

The post body is considered as a sequence of fields, using the '|' character as the field terminator. Unescaping (if any) depends on the field.

Response format

The response type is: application/json; charset=utf8. However, the response body is not actually JSON. Responses start with either "//OK" to indicate that the call returned successfully or "//EX" for an exception, followed by a JavaScript expression for an array.

The JavaScript exception need not be a literal array; as a workaround for a bug in IE 6 and 7, large arrays (above 32k elements) are actually encoded as a sequence of "concat" calls to create the array.

The fields in a GWT-RPC request

Version

The protocol version (int). GWT currently accepts versions 5-7 on the server.

Flags

A bitset (int) containing the options for this request. Currently, only two options are allowed:

FLAG_ELIDE_TYPE_NAMES = 1

This flag will be set if the GWT application turns on type obfuscation by including the RemoteServiceObfuscateTypeNames module. When enabled, type names will be represented

as opaque keys referring to records in the serialization policy file. (The GWT-RPC generator allocates unique ids using a counter and encodes them as base-36 numbers.)

FLAG_RPC_TOKEN_INCLUDED = 2

If set, the request includes an RpcToken used to guard against XSRF attacks.

The string table

This is a count followed by a list of escaped strings. Escape sequences for decoding:

- \0 -> nul character (U+0000)
- \\ -> \
- \! -> |
- \u followed by four hex digits -> decodes to the unicode character

The remaining fields in the request can use an index into the string table to represent a Java string. This allows repeated strings to be represented compactly. String indexes are one-based, with a 0 representing a Java null.

Module base URL

The base URL of the GWT app that sent this request. (String index.) This is used to find the serialization policy file (unless overridden).

Strong name

An index into the string table for the "strong name" of the policy file. Policy files may be different for each GWT-RPC service and permutation, or may be shared if the policies turn out to be identical.

RPC Token (optional)

If the RPC_TOKEN_INCLUDED flag is set, a serialized RpcToken object comes next. (See below for the format of a serialized Object.)

Service name

The next field is a string index pointing to the type name of the GWT-RPC service. Note that a servlet might implement multiple interfaces, and this field disambiguates between them. If the servlet doesn't implement the interface, an IncompatibleRemoteServiceException is thrown.

Method name

A string index pointing to the method name.

Parameter count

An integer with the number of parameters to the method.

Parameter types

The declared type of each method parameter for the method to be called. (See below.)

Parameter values

A sequence of Java values, one per parameter. (See Java Values.)

Java Types

Each type is represented as a string in the string table. For example:
`com.google.gwt.safehtml.shared.SafeHtmlString/235635043`
The class name comes from Class.getName() except for primitive types, which have one-letter abbreviations. The number is used to check type compatibility; it is a CRC32 checksum of the type's name and fields. (See SerializabilityUtil.generateSerializationSignature.)

Type name obfuscation

If the ELIDE_TYPE_NAMES option is set, the compiler generates a unique id and writes it to the policy file and generated code in place of the type name. (It generates the id using a base-36 encoding of a counter. See TypeSerializerCreator.realize.)

Java Values

Java values are represented similarly in the request and response, except that in a request, they are terminated by "|" and in the response, they are JavaScript expressions separated by commas.

Boolean

"0" is false, any other string is true. The type is serialized as "Z".

Byte

An integer between -128 and 127. Type is "B".

Char

Read as a Java integer, but only the low 16 bits are used. (Cast from int to char.) Type is "C".

Short

Read using Short.parseShort(). Type is "S".

Integer

Read using Integer.parseInt(). Type is "I".

Long

Version 5 represented a long as two doubles. Versions 6 and above represent a long as a base64 value, using '$' and '_' for 62 and 63. Unexpected ascii characters are treated as zero. If the base64 number doesn't fit into a long, the low 64 bits are taken. Type is "J". In the response, the base64 value has single quotes around it.

Float

Read as a Java double, then cast to float. Type is "F".

Double

Read as a Java double. Type is "D".

String

Represented as a one-based index into the string table. A '0' indicates a null. The type is "java.lang.String".

Java Objects

The first field of an object is a number indicating how to decode it. A '0' means null.

A negative numbers means to repeat a previously decoded object. (Note that this could be a parent of the object currently being deserialized, allowing for cycles.)

A positive number is a string index of the Java type, which is used for type-checking and to decode the rest of the value. (Decoding starts in AbstractSerializationReader.readObject.)

Type checking

If type name wasn't obfuscated, the stream reader checks its signature to make sure none of its fields changed. (See ServerSerializationStreamReader.deserialize.) Also, if it knows the declared type then resolves any generic types as much as possible. For example, if an RPC call passes a parameter to a method and the parameter's declared type is ArrayList<String>, the runtime type will be just ArrayList, but the stream reader can infer that the each element should be a String.
The stream reader then checks that the serialization policy allows this type to be instantiated (validateDeserialize).

Arrays

For an array, the type is followed by the array's length and the serialization of each item in the array. Each item is deserialized recursively, so arrays of non-null non-primitives will have each item preceded by its type (unless it was previously seen).

Enums

For enums, the type is followed by the Enum's ordinal (an integer).

Custom serialization

GWT-RPC supports custom serialization (see doc). This is used both within GWT itself for types such as collections and by applications. Custom serialization uses the methods on SerializationStreamReader and SerializationStreamWriter to read and write Java primitives using the format described here, and to recurse using readObject() and writeObject().

When a type has a custom serializer, the GWT-RPC generator doesn't know at compile time which fields will be recursively serialized using writeObject(). It makes a conservative assumption:

- If a field can be serialized, the custom serializer will call writeObject() on it.

If a type is serializable but the custom serializer won't call writeObject() on it, adding the "transient" keyword to the field will suppress visiting it.

Other objects

For other Java objects, the type is followed by the value for each serializable field in the leaf class, sorted by field name, followed by the fields in its ancestor classes (recursively). Fields are deserialized recursively.

Serialization policies

A GWT-RPC serialization policy controls whether each Java type is whitelisted for serialization or deserialization over GWT-RPC. (This is from the server's point of view; deserialization happens for request parameters and serialization for objects returned or thrown.) GWT-RPC supports two levels of serializability:

- Field-serializable means that the fields of the class may be serialized.
- Instantiable means that in addition, a new instance of the class may be created.

For example, in the Validation sample app, the RPC call may throw NumberFormatException, so this type is both field-serializable and instantiable for the serialization direction. NumberFormatException inherits from RuntimeException and Throwable but they're never thrown directly, so these classes have field serialization turned on but aren't instantiable.

API

GWT-RPC calls methods on a subclass of SerializationPolicy to check the policy. It calls validateSerialize() and validateDeserialize() to determine whether instances of a type may be sent over the wire. It calls shouldSerializeFields() and shouldDeserializeFields() to determine whether a supertype's fields may be serialized when its subtype is sent over the wire.

How RemoteServiceServlet locates the serialization policy file

RemoteServiceServlet assumes that serialization policies are stored as *servlet resources*. (Servlet resources aren't the same as Java resources; they're defined as part of J2EE, not the JDK. Servlet resources are typically provided as part of a war file.)

For this to work, the servlet must be running in a web app that also serves the static files that make up the GWT application. Therefore, it can use the GWT app's base URL (sent in the request) and servlet container methods to locate files generated by the GWT compiler.

Example: suppose a web page loads a GWT app using this script tag:

```
<script src="http://example.com/contextPath/something/SomeModule.nocache.js">
</script>
```

At startup, the GWT app finds its own script tag and calculates that its base URL is:
http://example.com/contextPath/something/

Then suppose it makes a GWT-RPC request to a servlet at:

http://example.com/contextPath/somethingElse/myServlet

The servlet calls request.getContextPath() and this returns "http://example.com/contextPath". It removes this prefix from the base URL (sent in the RPC call) to get "/something/". To this it adds the strong name and a standard file extension to get:

```
servlet.getServletContext().getResourceAsStream("/something/<strong name>.gwt.rpc");
```

If any of the above assumptions aren't true, the developer will have to override RemoteServiceServlet.doGetSerializationPolicy() to provide an alternate way to load the policy file.

Policy file format

A policy is defined by a UTF8 text file with one record per line, containing comma-separated fields. Blank lines are ignored. There may 2 or 7 fields per line. In the two-field format, the fields are:

```
typeName, isSerializable
```

In this format, the typeName is both the name of the Java type (suitable for Class.forName() and the id used on the wire to identify this type. There is only one flag, so it's all or nothing.

In the seven-field format, the fields are:

```
binaryTypeName, fieldSerializable, instantSerializable, fieldDeserializable,
instantDeserializable, typeId, (unused)
```

Here, binaryTypeName must be instantiable using Class.forName() and typeId is used to identity the class on the wire. This format has support for more fine-grained control and type name obfuscation.

(Not covered: lines beginning with @ClientFields.)

The parser is implemented in SerializationPolicyLoader.

How is a serialization policy calculated?

SerializationTypeOracleBuilder determines whether a type is serializable or not. This builder is run twice for each RPC stub, to calculate the types that may potentially be sent in each direction.
The results from the two SerializationTypeOracles are combined into a single text file, its MD5 hash is taken to get the strong name, and it's written to disk.

Arrays

GWT generates code to handle covariant arrays. For example, suppose an RPC call uses an array type such as Foo[][], and Foo has subtype Bar. Then GWT will generate code supporting the following array types: Foo[][], Foo[], Bar[][], Bar[]. This is to allow for covariant arrays. So if an array has rank R there are S instantiable subtypes (including the root), the number of array types generated is R*S.

Note that covariant arrays are also generated for fields of type List<Foo>, because an array of type Foo[] could be wrapped via Arrays.asList(). If you don't want this, you should use an ArrayList, which has a custom field serializer.

It's slightly more complicated if not all subtypes of Foo are instantiable. If Baz is a subtype of Foo that isn't instantiable for some reason, then the array type for Baz will be generated only if Baz has a subtype that's instantiable. (For example, this could happen if Baz is abstract or has no zero-arg constructor.)

Custom Serializers

If a type has a custom serializer, its hash will be calculated using the fields on the *serializer* class instead of on the original class. (See SerializabilityUtil.java line 895.) It may be useful to add a dummy field on the serializer and change its name if the custom serializer's protocol changes.

How many serialization policies does a GWT app have?

The worst case is quite large:
permutations = supported browsers * locales
policies = permutations * GWT-RPC services

However, normally GWT-RPC services don't use any types that are specific to a browser or locale. In that case, each permutation should generate an identical serialization policy for a given service and so there's one policy per service.

If for any reason the serialization policies turn out to be different for each permutation, this can cause server-side memory usage to blow up due to the large number of policies. It's a good idea to write a test on the output of the GWT compiler to make sure this doesn't happen.

How can I find out which GWT-RPC service generated a policy?

The GWT compiler writes output to two directories, controlled by the -deploy and -extra flags. In the extra directory, there is a file named rpcPolicyManifest/manifest.txt that has the mapping from RPC services to policy files. For example, the validation sample app generates this file:

```
# Module validation
# RPC service class, partial path of RPC policy file
com.google.gwt.sample.validation.client.GreetingService, D031DD0CECD85E06AF1E383A0EC73E6E.gwt.rpc
```

There is also a directory named "manifests/" which contains a file for each mapping, in a somewhat more machine-readable format. (The filenames in this directory are meaningless; they're just the MD5 of the contents.)

Caveats

- This document is based on reading GWT source code and experimentation. However, since I didn't implement GWT-RPC I may have misunderstood some things.
- This document skips some advanced features.