For Version 3.2.0

XENO-REMOTING

Reference Documentation

SINCE10™

TABLE OF CONTENT

1 OVERVIEW

The "xeno-remoting" (the "XR") is a Java library aim to enable the client side JavaScript to call the server side Java methods directly for AJAX based web applications. This document could be used as a guide to help you to get an idea of how to use it.

1.1 License

The XR is free software, licensed under the Apache License, Version 2.0 (the "License"). Commercial and non-commercial use are permitted in compliance with the License. You may obtain a copy of the License at: "http://www.apache.org/licenses/LICENSE-2.0". In addition, a copy of the License is included with each distribution.

The source code is available at: "https://github.com/kfeng2015/xeno-remoting".

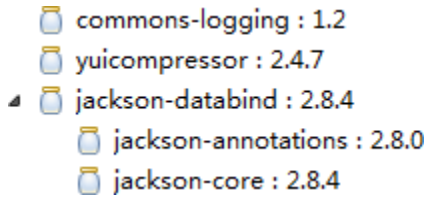
1.2 Runtime Requirements

To run the XR successfully, the server side system should meet below requirements:

- Install Java 8+ runtime environment with correct settings.
- Install Servlet 3.1+ specification implemented web container.

1.2.1 3rd Party Dependencies

In each release, all 3rd party dependencies will be put into the distributed package, please see "1.3 Distributed Files" for more information. The full dependency hierarchy is showing as below:



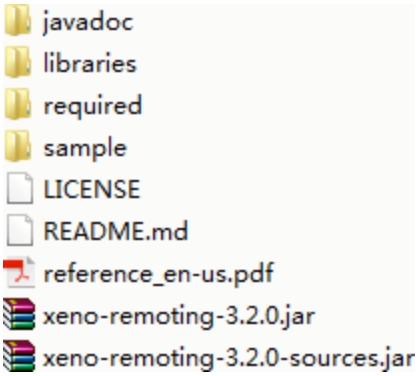
In generally, you should copy the "xeno-remoting-3.2.0.jar" together with all dependencies in the distributed files folder "libraries" to your web application folder "<webRoot>/WEB-INF/libs". To simplify the work, we recommend you to use Maven to manage dependencies, please see "2 Getting Started" for more information.

1.2.2 Browsers Compatibility

There are 4 kernels based browser are tested and fully compatible with the XR, they are: Trident, Geckos, Webkit and Presto. This means the XR could be run well in most modern browsers, e.g. Internet Explorer, Mozilla Firefox, Google Chrome, Apple Safari, Opera, Maxthon, etc.

1.3 Distributed Files

The XR is released in a ZIP file, you could extract it on any directory on your disk, and each distribution contains below structure:



Item	Description
javadoc	This folder contains Java API documents in HTML format, the "index.html" is the home page.
libraries	This folder contains both direct and indirect 3rd party dependencies.
required	This folder only contains direct 3rd party dependencies.
sample	This folder contains the "Hello World" sample described in "2 Getting Started".
LICENSE	The license document.
README.md	The document contains a short instruction for this library.
reference_en-us.pdf	The reference documentation in English.
xeno-remoting-3.1.0.jar	The released XR Java archive file.
xeno-remoting-3.1.0.sources.jar	The released XR Java archive file's source code.

2 GETTING STARTED

Let's start to use this library through a simple web application. We are using Apache Tomcat 8 as the web server, and configure the service port to "9080", and set the context path to "/helloworld". Then in the Eclipse IDE we create a Maven Project and add below dependency configuration code snippet in the "pom.xml" file:

```
<repositories>
   <repository>
      <id>since10-repo</id>
      <url>http://www.since10.com:8081/nexus/content/groups/public/</url>
   </repository>
</repositories>

<dependencies>
   <dependency>
      <groupId>com.since10</groupId>
      <artifactId>xeno-remoting</artifactId>
      <version>3.2.0</version>
   </dependency>
</dependencies>
```

2.1 Configure the Servlet Definition

Open the "web.xml" with any text editor and add below code snippet:

```
<servlet>
   <servlet-name>MessageServlet</servlet-name>
   <servlet-class>xeno.remoting.web.MessageServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>MessageServlet</servlet-name>
   <url-pattern>/xr/*</url-pattern>
</servlet-mapping>
```

That all for this time, and if you want to know more information about configurations, please see "3.1 Configurations".

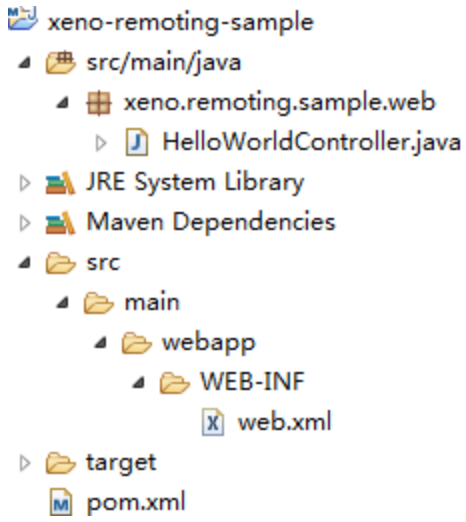
2.2 Call the Server Side Java Method

To call the server side Java method by the client side JavaScript, you should:

1. Declare the remote proxy.
2. Add the web method in the remote proxy.
3. Import the remote proxy into the JSP.
4. Process the remote call.

2.2.1 Declare the Remote Proxy

Create a Java class "xeno.remoting.sample.web.HelloWorldController" and add the "@RemoteProxy" annotation to declare it as a remote proxy, see below screenshot and code:




```
package xeno.remoting.sample.web;


import xeno.remoting.bind.RemoteProxy;


@RemoteProxy
public class HelloWorldController {
   // Implements your logic code here...
}
```

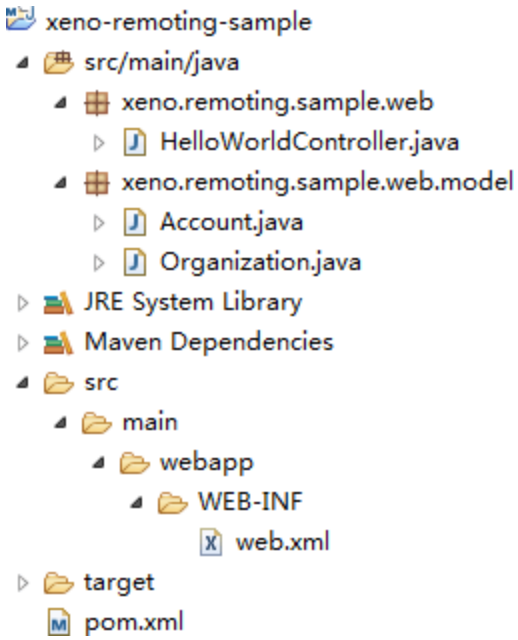
2.2.2 Add the Web Method in the Remote Proxy

Continue editing the Java class " xeno.remoting.sample.web.HelloWorldController", and add a method "getOrganization" with the "@WebMethod" annotation to declare it as a web method, see below code snippet:

```
@WebMethod
```

```
public Organization getOrganization(Account account, String token) {

    Account account_1 = new Account();

    account_1.setId(9);

    account_1.setName("Jane Lin");


    Account account_2 = new Account();

    account_2.setId(7);

    account_2.setName("Tim Zhang");


    Organization organization = new Organization();

    organization.setCode("org_a_" + token);

    organization.setRegion("Red Zone");

    organization.setAccounts(Arrays.asList(account, account_1, account_2));


    return organization;

}
```

To demonstrate the auto serialization & deserialization feature, we also create 2 Java models for the web method's input and output, see below screenshot and code snippet:




```
// Code snippet for the Java model "xeno.remoting.sample.web.model.Account".
package xeno.remoting.sample.web.model;


public class Account {
    private long id = 0;
    private String name = null;

    // Public getter & setter should be added here...
}

// Code snippet for the Java model "xeno.remoting.sample.web.model.Organization".
package xeno.remoting.sample.web.model;


import java.util.List;
```

```
public class Organization {
    private String code = null;
    private String region = null;
    private List<Account> accounts = null;

    // Public getter & setter should be added here...
}
```

2.2.3 Import the Remote Proxy into the JSP

Create a JSP named as "default.jsp" with any text editor in the web application root directory and add below code:

```
<%@ page contentType="text/html; charset=utf-8"%>
<%@ taglib prefix="xr" uri="http://www.since10.com/xeno-remoting" %>
<!doctype html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
      <title>Hello World</title>
      <xr:script>
         import xeno.remoting.sample.web.HelloWorldController;
      </xr:script>
   </head>
   <body>
      <input id="btnTest" type="button" value="Get Organization" />
```

```
    </body>
</html>
```

In this page, the remote proxy "xeno.remoting.sample.web.HelloWorldController" created in "2.2.1 Declare the Remote Proxy" will be imported by the JSP custom tag "<xr:script>". We also provide a button "btnTest" to trigger the remote call.

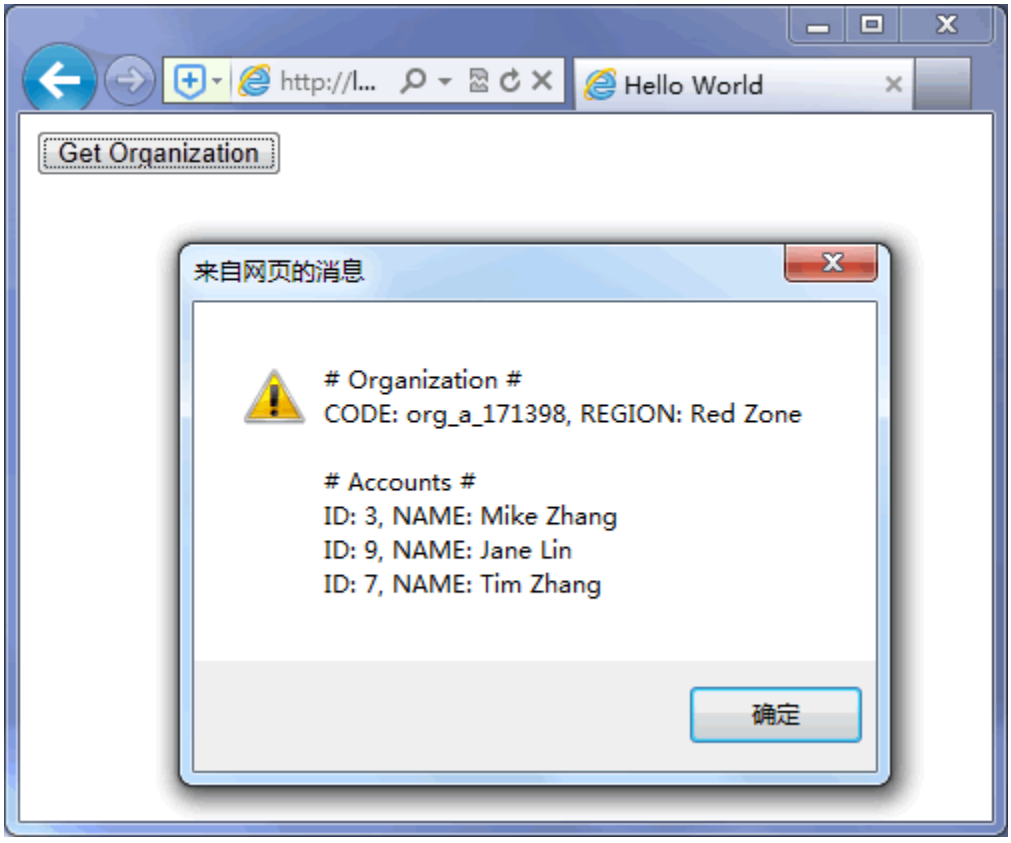
2.2.4 Process the Remote Call

Continue working on this page, and add below code snippet into the "<head>" tag:

```
<script type="text/javascript">

   window.onload = function(evt) {

      document.getElementById("btnTest").onclick = function(evt) {

         var account = {
            id: 3,
            name: "Mike Zhang"
         };

         var token = "171398";

         var option = {

            successCallback: function(result, xhr) {
```

```
            var organization = result.data;


            var organizationText = "# Organization #" + "\n";
            organizationText += "CODE: " + organization.code + ", ";
            organizationText += "REGION: " + organization.region + "\n";


            var accountsText = "# Accounts #" + "\n";


            for(var i = 0; i < organization.accounts.length; i += 1) {
               var account = organization.accounts[i];


               accountsText += "ID: " + organization.accounts[i].id + ", ";
               accountsText += "NAME: " + organization.accounts[i].name + "\n";
            }


            alert(organizationText + "\n" + accountsText);
         }
      };


      xeno.remoting.sample.web.HelloWorldController.getOrganization(account, token, option);
   };
};
</script>
```

Now you could start up the server and type "http://localhost:9080/helloworld/default.jsp" in the browser's address bar, and then click the button "Get Organization" on the page to see the similar result according to below picture:



For more information about the remote call, please see "3.2 Remote Call".

3 DEVELOPER'S GUIDE

Through "2 Getting Started" you may have got a rough idea about this library, now we will take a deeper look at it. As most Java web applications, the XR provides a Servlet "xeno.remoting.web.MessageServlet" to do initializations and handle request & response operations. Once you have configured it in the "web.xml", when the application startups it will scan classes in the class path and extract remote proxies' metadata information and store them in the memory. Then for each request handled by this Servlet, it will look up the matched remote proxy, invoke the target method with arguments by the Java refection mechanism, and serialize the result into JSON string before response it to the client side. All requests are based on the traditional HTTP URL request, but the XR has encapsulated it by providing a more convenient way for developing web applications between the client side and the server side.

3.1 Configurations

To use all features provided by the XR, you should configure the Servlet "xeno.remoting.web.MessageServlet" in the "web.xml", below code snippet shows the configuration with a few initialization parameters:

```
<servlet>
   <servlet-name>MessageServlet</servlet-name>
   <servlet-class>xeno.remoting.web.MessageServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
   <init-param>
```

```
        <param-name>characterEncoding</param-name>
        <param-value>GB2312</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>MessageServlet</servlet-name>
    <url-pattern>/xr/*</url-pattern>
</servlet-mapping>
```

Nothing special for this Servlet's configurations except the "url-pattern", this value should always be "/xr/*". This is because all URL requests go through under this pattern that will be handled by the Servlet. Below table lists acceptable initialization parameters for this Servlet, both of them are optional, if not provides, the default value will be used:

Parameter Name	**Description**	**Default**
characterEncoding	A parameter to determine the way of characters encoding.	UTF-8
debugMode	A parameter to determine whether this Servlet will be startup in a debug mode or not. If the value of this parameter sets to "true" (case insensitive), all JavaScripts generated by this Servlet will in a readable format. And you could use "<contextPath>/xr/api" to see all available remote proxies and their APIs documentation. Otherwise, all JavaScripts generated by this Servlet will be combined together and minified to reduce the size.	false

3.2 Remote Call

Different from traditional web applications use URL requests with key-value pair parameters to send data to the server side and retrieve those request parameters, the XR provides the ability for the client side JavaScript to call the server side Java method directly. We give the name of this feature as the "Remote Call".

Do not be fooled by the name, actually the XR is still using traditional HTTP URL request and response, and it takes the responsibility to encapsulate operations such as data serialization & deserialization, invoke Java method, handle exception, etc.

In "2.2 Call the Server Side Java Method" you have got steps about how to use the remote call, you should:

1. Declare the remote proxy by marking the "@RemoteProxy" annotation for the Java class.
2. Add Java methods and marking the "@WebMethod" annotation for those methods which want to be called from the client side.
3. As an optional step, you could also add Java methods and marking the "@ExceptionHandler" annotation to handle the exception when the target method throws exception during the execution.
4. Import remote proxies of using into the JSP.
5. And process the remote call.

3.2.1 Remote Proxy

"@RemoteProxy", the Java class "xeno.remoting.bind.RemoteProxy", this annotation uses to mark a class as a remote proxy and visible to the client side, it should meet below requirements

- The class should provide the public modifier, any abstract class or interface is not allowed.
- The class should provide a default (public & non-argument) constructor.
- Inner nested remote proxy (whatever exists in a valid remote proxy or other general classes) is not allowed.
- Unless explicit marked with this annotation on the class, any sub classes of this class will not be recognized as a remote proxy automatically.

Once the remote proxy has been defined, it should be imported into the page if you want to use it. The XR will translate the Java code into the JavaScript code for the client side uses automatically, for an example, see below remote proxy code:

```
package test;


import xeno.remoting.bind.RemoteProxy;
import xeno.remoting.bind.WebMethod;


@RemoteProxy
public class MyService {


    @WebMethod
    public int calculate(int a, int b) {
```

```
        return sum(a, b);
    }

    public int sum(int... v) {
        int r = 0;

        for (int i : v) {
            r += i;
        }

        return r;
    }

    @WebMethod
    public void check(String auth) throws IllegalArgumentException {

        if (auth == null) {
            throw new IllegalArgumentException("No auth detected");
        }
    }
}
```

Then import this remote proxy into the page by the JSP custom tag (see "3.2.1.1 Import Remote Proxies" for more information):

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="xr" uri="http://www.since10.com/xeno-remoting" %>
<!doctype html>
```

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
      <xr:script>
         import test.MyService;
      </xr:script>
   </head>
</html>
```

Finally, when you access this page, the "test.MyService" will be translated into below JavaScript code:

```
xr_global_register_class("test.MyService", function(id, engine) {

   return {

      calculate: function(arg0, arg1, option) {
         return engine.sendRequest(id, "calculate", [arg0, arg1], option);
      },

      check: function(arg0, option) {
         return engine.sendRequest(id, "check", [arg0], option);
      }
   };
});
```

So this is why you could call the Java method by the similar JavaScript function at the client side like below code snippet:

```
var progress = test.MyService.calculate(1, 2, {
   successCallback: function(result, xhr) {
      // Implements your logic code here...
   },
   errorCallback: function(fault, xhr) {
      // Implements your logic code here...
   }
});
```

For each web method translated JavaScript function, the last optional argument allows you to do some settings of the remote call, please see "3.2.2.1 AJAX Option" for more information.

3.2.1.1 Declare Prototype Remote Proxy

As an optional step, by default, each remote proxy will be created as singleton to handle the remote call. So if you want to use the remote proxy as new instances for each remote call, you could do like this:

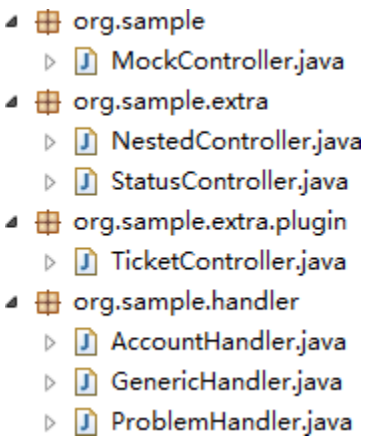
```
@RemoteProxy(BeanScope.PROTOTYPE)
public class MyService {
      // Ignore code here...
}
```

3.2.1.2 Import Remote Proxies

Before you call the Java web method at the client side, related remote proxies should be imported into current page. The XR provides a JSP custom tag to do it, the URI is "http://www.since10.com/xeno-remoting", see below specification of it:

```
<tag>
   <name>script</name>
   <tag-class>xeno.remoting.web.JsContentTagSupport</tag-class>
</tag>
```

Contents between in this tag will be parsed and translated into JavaScript automatically. To explain how to use it, look below packages and classes structure:



Assume all classes in above screenshot are remote proxies, so if you want to import some of them into the page, you could add below highlighted code into your page:

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="xr" uri="http://www.since10.com/xeno-remoting" %>
<!doctype html>
<html>
```

```
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <xr:script>
            import org.sample.extra.NestedController;
            import org.sample.extra.StatusController;
            import org.sample.handler.GenericHandler;
        </xr:script>
    </head>
</html>
```

This tag also provide to use "*" to import all remote proxies direct under the package, below code snippet has the same effect as above, and those remote proxies under the package "org.sample.extra.plugin" will not be included:

```
<xr:script>
    import org.sample.extra.*;
    import org.sample.handler.GenericHandler;
</xr:script>
```

The line comment "//" and the block comment "/* */" are both supported in the body content, in below code snippet, the "org.sample.handler.GenericHandler" will be excluded:

```
<xr:script>
    import org.sample.extra.*;
    // import org.sample.handler.GenericHandler;
</xr:script>
```

Furthermore, you could reuse this tag in the page with multi-blocks:

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="xr" uri="http://www.since10.com/xeno-remoting" %>
<!doctype html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
      <xr:script>
         import org.sample.extra.*;
         import org.sample.handler.GenericHandler;
      </xr:script>
      <xr:script>
         import org.sample.MockController;
         import org.sample.extra.plugin.TicketController;
         import org.sample.handler.*;
      </xr:script>
   </head>
</html>
```

3.2.2 Web Method

"@WebMethod", the Java class "xeno.remoting.bind.WebMethod", this annotation uses to mark a method in the remote proxy as a web method and provides ability to call this method from JavaScript directly, it should meet below requirements:

- The method should be used in a remote proxy or its super classes.
- The method should provide the public modifier, and the static modifier is also allowed.

- All methods marked with this annotation could be inherited from its super classes, and overrides of these methods are allowed.
- If a method marked with this annotation in its super class, but remove this annotation in its inherited class, this method in its inherited class would not be recognized as a web method.
- Overload methods which both/all marked with this annotation is not allowed.

As mentioned in "3.2.1 Remote Proxy", each web method could be translated into the JavaScript function at the client side, arguments type and position are the same as those at the server side, an optional argument "option" will be generated for each translated JavaScript function, please see "3.2.2.1 AJAX Option" for more information.

3.2.2.1 AJAX Option

The XR provides the AJAX mechanism of data exchanging, the last argument of each translated JavaScript function give you opportunities for setting request timeout and adding handlers to monitor the status, see below translated JavaScript code:

```
xr_global_register_class("net.corp.OrgService", function(id, engine) {

   return {

      find: function(arg0, arg1, option) {
         return engine.sendRequest(id, "find", [arg0, arg1], option);
      }
   };
});
```

Below code snippet is an example of the AJAX option usage:

```
var acct = {
   id: 101,
   name: "Mary Wang"
};

// Processes the remote call with the AJAX option.
var progress = net.corp.OrgService.find(acct, "kfeng", {
   timeout: 120000,
   successCallback: function(result, xhr) {
      // Implements your logic code here...
   }
});

// Or processes the remote call without the AJAX option.
var progress = net.corp.OrgService.find(acct, "kfeng");
```

Below table lists all supported properties of the AJAX option, for more information about how to use it, please see "3.1 Configuration" to startup the application in debug mode and type "<contextPath>/xr/api" in the browser's address bar to check "AjaxOption" and "Engine" objects in the APIs documentation:

Property	Description
timeout	A millisecond describes the request timeout.
beforeSendCallback	The function will be triggered before the request to be sent.

successCallback	The function will be triggered when the request success. (No error, timeout, or abort occurs.)
errorCallback	The function will be triggered when error occurs during the request.
timeoutCallback	The function will be triggered when the request timeout.
abortCallback	The function will be triggered when the request abort.
completeCallback	The function will be trigger when the request completes whatever it is success, error, timeout or even abort.

3.2.2.2 Serialization / Deserialization

The XR uses JSON to transfer data between requests and responses. The data to be sent to the server side will be serialized into JSON string at the client side before the request start, then restore it by deserializing the JSON string into the Java object at the server side. When the remote call complete, the data will be serialized into JSON string at the server side and response back, then restore it by deserializing the JSON string into the JavaScript object at the client side.

At the client side, most modern browsers (e.g. IE8+, Firefox, Chrome, etc.), JSON has already be implemented and embed in the browser. If end user's browser does not support this feature, you should pick up one JSON adopter manually, for an example, the "json2.js" (https://github.com/douglascrockford/JSON-js) is one of a good choice.

At the server side, the "Jackson" (https://github.com/FasterXML/jackson) library is used as a JSON adopter, you could visit the project website for more information. It is recommended each Java object which wants to be serialized or to be deserialized should meet below requirements:

- Provide a default (public & non-argument) constructor.

- Provide public getter and setter methods for properties.
- Do not provide self-reference / infinity recursively objects.

3.2.2.3 Abort Progress

It is possible to abort a progressing remote call when you need, each web method at the client side will return an "AjaxProgress" object, and you could abort the progress through below code snippet:

```
var progress = test.MyService.calculate(1, 2);


// After a long time or some condition else…
progress.abort();
```

One important thing you should know, when the abort function has been called, it will just cut off the communication between the client side and the server side. Actually, at the server side, this progress is still running but no response will be returned according to this request.

3.2.3 Exception Handler

"@ExceptionHandler", the Java class "xeno.remoting.bind.ExceptionHandler", this annotation uses to mark a class as an exception handler to be able to handle exception when calling a web method failed, it should meet below requirements:

- The method should be used in a remote proxy or its super classes.
- The method should provide the public modifier, and the static modifier is also allowed.

- The method should provide only one pass in argument with the class type of "java.lang.Exception" or its sub classes.
- All methods marked with this annotation could be inherited from its super classes, and overrides & overloads of these methods are allowed.
- No multiple exception handlers for the same exception type are allowed.

Normally, if any exception occurs during your web method, you could set the error handler at client to get the information about it:

```
package com.report.web;


import xeno.remoting.bind.RemoteProxy;
import xeno.remoting.bind.WebMethod;


@RemoteProxy
public class UserService {


   @WebMethod
   public Account createAccount(Account obj) {


      if (obj == null) {
         throw new IllegalArgumentException("The 'obj' is null");
      }


      // Implements your logic code here...
   }
}
```

Then at the client side, below code snippet could get the information about the exception "java.lang.IllegalArgumentException" if you pass null into the web method "createAccount":

```
com.report.web.UserService.createAccount(null, {
    onError: function(fault, xhr) {
        alert(fault.type);  // The "java.lang.IllegalArgumentException" will be displayed.
        alert(fault.message);  // The "The 'obj' is null" will be displayed.
        alert(fault.data);  // The null will be displayed.
    }
});
```

The main purpose of the exception handler is designed to perform some additional works when the exception occurs, add below highlighted code into the Java class "com.report.web.UserService", now it will return the string "Invalid data detected!" if you pass null into the web method "createAccount":

```
package com.report.web;

import xeno.remoting.bind.ExceptionHandler;
import xeno.remoting.bind.RemoteProxy;
import xeno.remoting.bind.WebMethod;

@RemoteProxy
public class UserService {

    @ExceptionHandler
    public String handleException(IllegalArgumentException ex) {
```

```
        return "Invalid data detected!";
    }

    @WebMethod
    public Account createAccount(Account obj) {

        if (obj == null) {
            throw new IllegalArgumentException("The 'obj' is null");
        }

        // Implements your logic code here...
    }
}
```

This time at the client side, you could get the string "Invalid data detected!" through the JavaScript callback argument "fault":

```
com.report.web.UserService.createAccount(null, {
    onError: function(fault, xhr) {
        alert(fault.type);  // The "java.lang.IllegalArgumentException" will be displayed.
        alert(fault.message);  // The "The 'obj' is null" will be displayed.
        alert(fault.data);  // The "Invalid data detected!" will be displayed.
    }
});
```

The XR use the same mechanism for data serialization & deserialization for the exception handler, please see "3.2.2.2 Serialization / Deserialization" for more information. And it will find the closest type of exception to be handled, that means it will look up the same exception type's handler, if it does not exist, it will look up its direct super class

recursively. Take below Java code for an example, the final result is "Runtime exception detected!", if you pass null into the web method "createAccount":

```
package com.report.web;


import xeno.remoting.bind.ExceptionHandler;
import xeno.remoting.bind.RemoteProxy;
import xeno.remoting.bind.WebMethod;


@RemoteProxy
public class UserService {


    @ExceptionHandler
    public String handler1(RuntimeException ex) {
        return "Runtime exception detected!";
    }


    @ExceptionHandler
    public String handler2(Exception ex) {
        return "Exception detected!";
    }


    @WebMethod
    public Account createAccount(Account obj) {


        if (obj == null) {
            throw new IllegalArgumentException("The 'obj' is null");
```

```
    }


    // Implements your logic code here...
  }
}
```

3.2.3.1 Change Response Status Code

In general, once the exception handler invoked, whatever the handling result is success or not, the response status code will become 500 and trigger the error callback at the client side if has. As an optional way, you could also determine the response status code when handling exception in an exception handler like below code snippet:

```
@ExceptionHandler(HttpStatus.OK)
public String handleException(IllegalArgumentException ex) {
  return "Invalid data detected!";
}
```

Then the final response status code will become 200 and trigger the success callback at the client side if has. So if you want, you could modify the response status code to change the result when you needed, and:

- For status code 200 to 299, it will trigger the success callback if has.
- For status code 408 and 504, it will trigger the timeout callback if has.
- For others status codes, it will trigger the error callback if has.

3.2.3.2 Unhandable Exception

The exception "xeno.remoting.web.UnhandlableException" will be thrown when the XR could not handle the exception, for an example, it is possible to throw an exception even inside the exception handler itself, see below highlighted code:

```
package com.report.web;


import xeno.remoting.bind.ExceptionHandler;
import xeno.remoting.bind.RemoteProxy;
import xeno.remoting.bind.WebMethod;


@RemoteProxy
public class UserService {


   @ExceptionHandler
   public int handler1(IllegalArgumentException ex) {
      return 1 / 0;
   }


   @ExceptionHandler
   public String handler2(ArithmeticException ex) {
      return "Arithmetic exception detected!";
   }


   @WebMethod
   public Account createAccount(Account obj) {
```

```
        if (obj == null) {
            throw new IllegalArgumentException("The 'obj' is null");
        }

        // Implements your logic code here...
    }
}
```

The exception "java.lang.IllegalArgumentException" will be thrown if you pass null into the web method "createAccount" and captured by the exception handler "handler1". Unfortunately "handler1" will also throw the exception "java.lang.ArithmeticException". Although there is another exception handler "handler2" has been defined in this remote proxy to handle such kind of exception, the exception "java.lang.ArithmeticException" will not be captured any more.

4 KNOWN ISSUES

Web methods annotation will be lost if the super class is with default modifier.

If the remote proxy's super class is with default modifier, and you have defined web methods in its super class, annotation about web methods in this super class will be lost.

Please see "http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6815786" for more information, this issue causes since Java 1.6, a walk around solution is that do not define any web method in the class which marked as default modifier.

Method "MessageServlet.getCurrentHttpRequestThread" throws the exception.

The XR uses "java.lang.ThreadLocal" to store each request's information so as to you could get them anywhere in your code except you planned to use them in another thread:

```
public void foo() {
      Request req1 = MessageServlet.getCurrentHttpRequestThread();

      new Thread(new Runnable() {

         public void run() {
            Request req2 = MessageServlet.getCurrentHttpRequestThread();
         }
      }).start();
}
```

Above highlighted code will be run in a new thread which is different from current request thread context, so you will get "req1" with correct information bound to your current request thread, but the exception "java.lang.IllegalStateException" will be thrown for "req2".

This issue will also be reported for below methods if this situation occurs, because they are using the "java.lang.ThreadLocal" to store information:

- xeno.remoting.web.MessageServlet.getCurrentHttpRequestThread()
- xeno.remoting.web.MessageServlet.getCurrentHttpResponseThread()
- xeno.remoting.web.MessageServlet.getCurrentHttpSessionThread()

Remote call will be broken if the input stream has been used before the XR.

The XR uses the input stream of each request to carry data from the client side. If it has been used before it comes to the Servlet "xeno.remoting.web.MessageServlet", the remote call and the subscribe/publish mechanism will be broken. This situation mostly taken place when you add a filter to do some content filtering works, for an example, you have configured a filter like below code snippet:

```
<filter>
   <filter-name>ContentFilter</filter-name>
   <filter-class>com.report.web.ContentFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>ContentFilter</filter-name>
   <url-pattern>*</url-pattern>
</filter-mapping>
```

Then in the "doFilter" method, it will be performed as below code snippet:

```
InputStream stream = request.getInputStream();


// Reads the input stream here...

// For an example: stream.read();


chain.doFilter(request, response);
```

The "ContentFilter" will handle all requests URLs, actually the "url-pattern" of the Servlet "xeno.remoting.web.MessageServlet" also be included in it. Some walk around solutions are always focus on do not operate the input stream in the filter chain, or configure the filter's "url-pattern" to avoid match that "url-pattern" of the Servlet "xeno.remoting.web.MessageServlet".

5 FAQ

How to get original HTTP Servlet request?

The Servlet "xeno.remoting.web.MessageServlet" provides 3 static methods to get wrapped instances of the HTTP Servlet request, the HTTP Servlet response, and the HTTP session:

- xeno.remoting.web.MessageServlet.getCurrentHttpRequestThread()
- xeno.remoting.web.MessageServlet.getCurrentHttpResponseThread()
- xeno.remoting.web.MessageServlet.getCurrentHttpSessionThread()

For most cases, you do not need the original instance to handle operations because the wrapper object provides similar methods of that instance it wrapped. If you really want to get the original instance you could do it as below code snippet:

```
// Gets the HTTP Servlet request, the interface "javax.servlet.http.HttpServletRequest".
Request wrappedRequest = MessageServlet.getCurrentHttpRequestThread();
HttpServletRequest httpServletRequest = wrappedRequest.getOriginalHttpServletRequest();

// Gets the HTTP Servlet response, the interface "javax.servlet.http.HttpServletResponse".
Response wrappedResponse = MessageServlet.getCurrentHttpResponseThread();
HttpServletResponse httpServletResponse = wrappedResponse.getOriginalHttpServletResponse();

// Gets the HTTP session, the interface "javax.servlet.http.HttpSession".
Session wrappedSession = MessageServlet.getCurrentHttpSessionThread();
HttpSession httpSession = wrappedSession.getOriginalHttpSession();
```

How to get the Spring bean in the remote proxy?

Currently the XR could not integrate with the Spring framework directly because it does not go through the Servlet "org.springframework.web.servlet.DispatcherServlet". If you want to get the Spring bean, you could do it as below code snippet:

```
ServletContext webContext = MessageServlet.getCurrentServletContext();
ApplicationContext appContext = WebApplicationContextUtils.getWebApplicationContext(webContext);

// Gets the Spring bean from the application context.
SiteDao siteDao = appContext.getBean("newSiteDao", SiteDao.class);
```

The Java class "org.springframework.web.context.support.WebApplicationContextUtils" is a utilities provides by the Spring framework. And for the Servlet context argument, you could get it through the Servlet "xeno.remoting.web.MessageServlet".