For Version 2.0.x

XENO-REMOTING

Reference Documentation

SINCE10™

TABLE OF CONTENT

1 OVERVIEW

The "xeno-remoting" (the "XR") is a Java library aim to simplify the development process between the client side and the server side for AJAX based web applications, it provides below features:

- Enable the client side JavaScript to call the server side Java methods directly.
- Enable the server side Java to call the client side JavaScript functions directly.

This document could be used as a guide and help you to get the idea of how to use this library, and it is suitable for all version 2.0.x based releases.

1.1 License

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

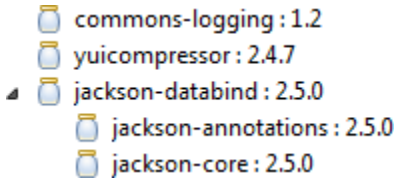
1.2 Runtime Requirements

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

- Install Java 7 runtime environment with correct settings.
- Install Servlet 3.0 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, for your web application, you should copy the "xeno-remoting-2.0.0.jar" together with all dependencies in the distributed files folder "libraries" to the web application folder "<webRoot>/WEB-INF/libs".

1.2.2 Browsers Compatibility

There are 4 kernel browsers are tested and fully compatible with this library, they are:

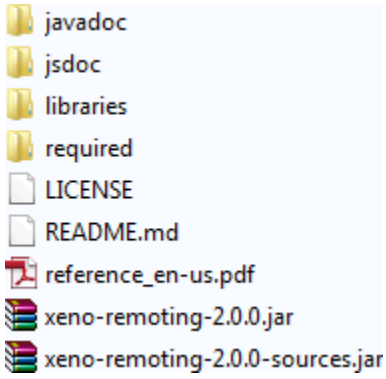
- Trident
- Geckos

- Webkit
- 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:

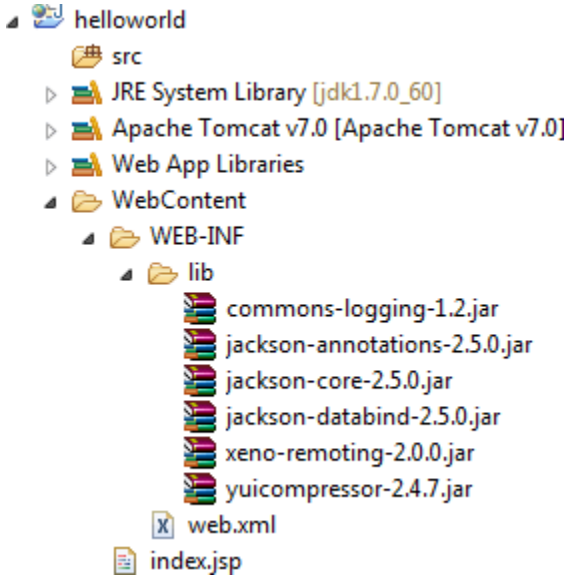


Item	Description
javadoc	This folder contains Java API documents in HTML format, the "index.html" is the home page.
jsdoc	This folder contains JavaScript API documents in HTML format, the "index.html" is the home page.
libraries	This folder contains both direct and indirect 3rd party dependencies.
required	This folder only contains direct 3rd party dependencies.
LICENSE	The license document.

reference_en-us.pdf	The reference documentation in English.
xeno-remoting-2.0.0.jar	The released XR Java archive file.
xeno-remoting-2.0.0.sources.jar	The released XR Java archive file's source code.

2 GETTING STARTED

Let's start to use it through a simple web application. We are using Apache Tomcat 7 as the web container, and configure the service port for "9080", and the context path is "/helloworld". In the Eclipse IDE we create a Dynamic Web Project with below structure:



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>
   <async-supported>true</async-supported>
```

```
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>MessageServlet</servlet-name>
    <url-pattern>/xr/*</url-pattern>
</servlet-mapping>
```

For more information about configurations, please see "3.1 Configurations".

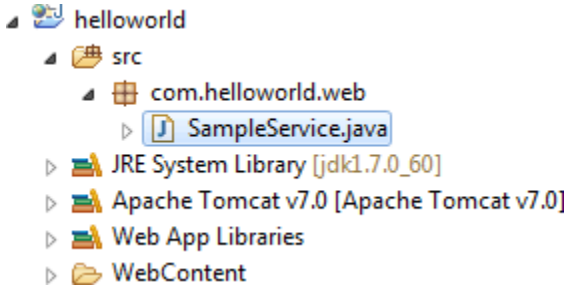
2.2 Call the Server Side Java Methods

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

1. Declare the remote proxy.
2. Add web methods in the remote proxy.
3. Import the remote proxy on JSP and process the remote call.

2.2.1 Declare the Remote Proxy

Create a Java class "com.helloworld.web.SampleService" and add the "@RemoteProxy" annotation to declare it as a remote proxy, see below screenshot and code:



```
package com.helloworld.web;


import xeno.remoting.bind.RemoteProxy;


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

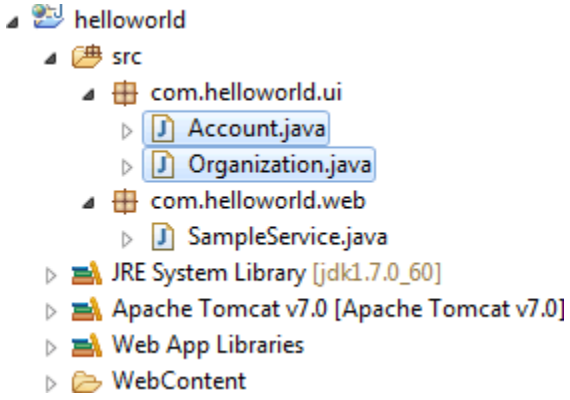
2.2.2 Add the Web Method

Continue editing the Java class "com.helloworld.web.SampleService", and add a method "getOrganization" with the "@WebMethod" annotation to declare a web method, see below highlighted code:

```
package com.helloworld.web;


import java.util.ArrayList;
import java.util.List;


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


import com.helloworld.ui.Account;
import com.helloworld.ui.Organization;
```

```
@RemoteProxy
public class SampleService {

   @WebMethod
   public Organization getOrganization(Account acct, String auth) {
      Account acct1 = new Account();
      acct1.setId(9);
      acct1.setName("Jane Lin");

      Account acct2 = new Account();
      acct2.setId(7);
      acct2.setName("Tim Zhang");

      List<Account> accounts = new ArrayList<Account>();
      accounts.add(acct1);
      accounts.add(acct2);
      accounts.add(acct);

      Organization organization = new Organization();
      organization.setCode("org_a_" + auth);
      organization.setName("Organization A");
      organization.setAccounts(accounts);

      return organization;
   }
}
```

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




```
// Code snippet for the Java model "com.helloworld.ui.Account".

package com.helloworld.ui;


public class Account {

    private long id = 0;

    private String name = null;


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

}


// Code snippet for the Java model "com.helloworld.ui.Organization".

package com.helloworld.ui;


import java.util.List;
```

```
public class Organization {

    private String code = null;

    private String name = null;

    private List<Account> accounts = null;


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

}
```

2.2.3 Process the Remote Call

Open the "index.jsp" with any text editor 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" />

        <xr:proxy>

            import com.helloworld.web.SampleService;

        </xr:proxy>

    </head>

    <body>

        <input id="btnGetOrganization" type="button" value="Get Organization" />

        <div id="txtOutputConsole"></div>

    </body>

</html>
```

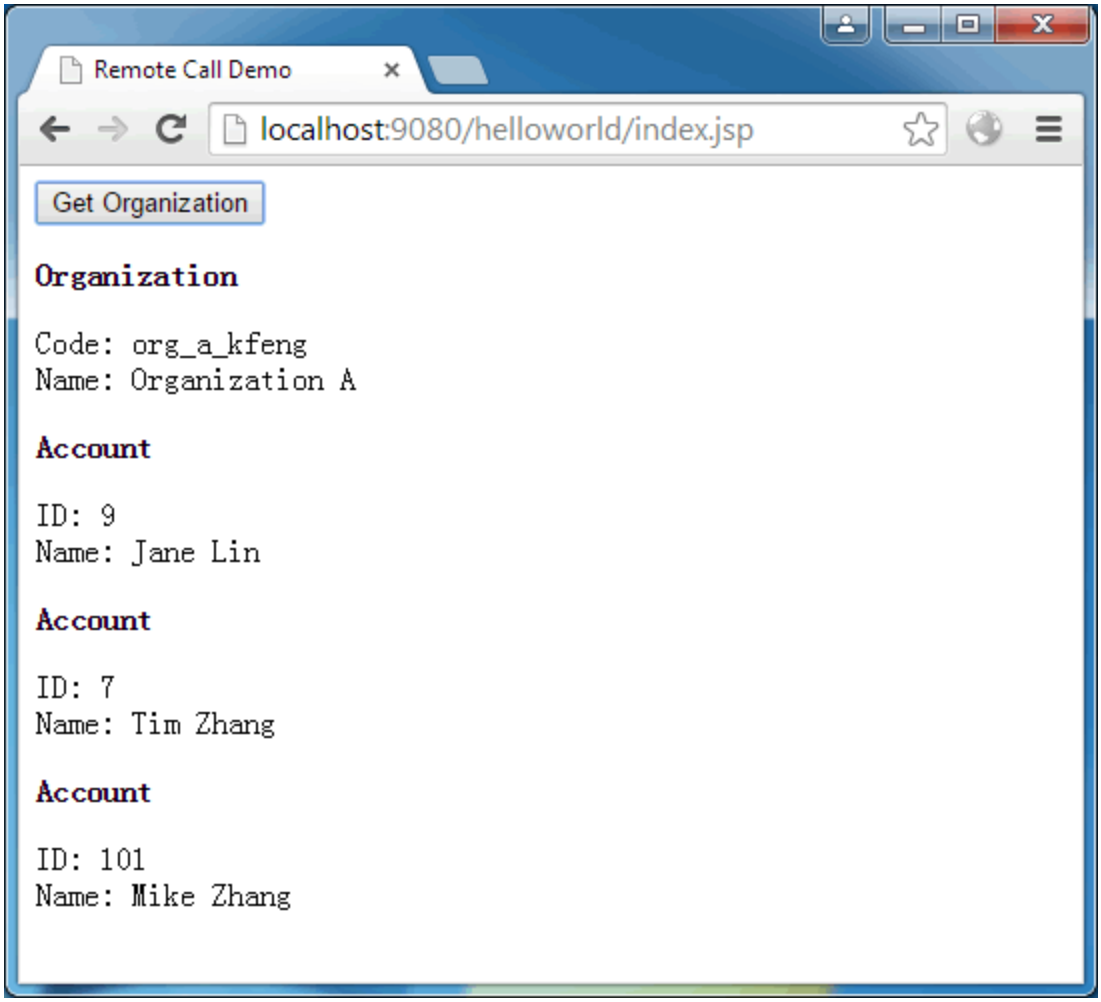
In this page, the remote proxy "com.helloworld.web.SampleService" created in "2.2.1 Declare the Remote Proxy" will be imported by the JSP custom tag "<xr:proxy>". We also provide a button "btnGetOrganization" to trigger the remote call and the "txtOutputConsole" to track the result.

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

```
<script type="text/javascript">
   window.onload = function() {
      document.getElementById("btnGetOrganization").onclick = function(evt) {

         var account = {
            id: 101,
            name: "Mike Zhang"
         };

         var options = {
            onSuccess: function(result, xhr) {
               var organization = result.data;
               var content = "<p><b>Organization</b></p>";
               content += "Code: " + organization.code + "<br />";
               content += "Name: " + organization.name + "<br />";

               for(var i in organization.accounts) {
                  content += "<p><b>Account</b></p>";
                  content += "ID: " + organization.accounts[i].id + "<br />";
                  content += "Name: " + organization.accounts[i].name + "<br />";
               }
```

```
            document.getElementById("txtOutputConsole").innerHTML = content;
          }
        };

        com.helloworld.web.SampleService.getOrganization(account, "kfeng", options);
      };
    };
</script>
```

Now you could start up the server and type "http://localhost:9080/helloworld/index.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".

2.3 Call the Client Side JavaScript Functions

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

1. Enable the reverse AJAX feature at both client & server sides.
2. Encapsulate JavaScript functions at the server side and process the callback.

2.3.1 Enable the Reverse AJAX Feature

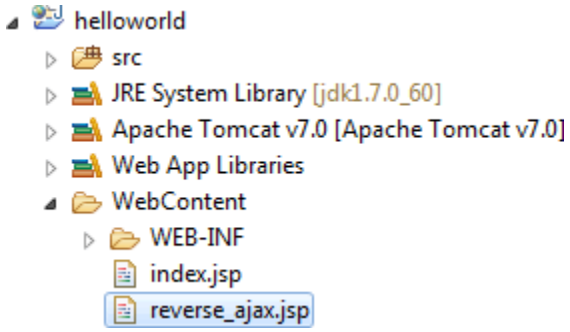
To use the reverse AJAX, you should set the attribute "async-supported" in the Servlet's configurations to "true", please see below highlighted code snippet from "2.1 Configure the Servlet Definition":

```
<servlet>
    <servlet-name>MessageServlet</servlet-name>
    <servlet-class>xeno.remoting.web.MessageServlet</servlet-class>
    <async-supported>true</async-supported>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Then for those pages which want to receive the server side data push back, you should also enable this feature through below code snippet:

```
xeno.remoting.web.Engine.setReverseAjaxEnabled(true);
```

To demonstrate this feature, create another JSP "reverse_ajax.jsp" 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" />
      <xr:proxy>
         import com.helloworld.web.SampleService;
      </xr:proxy>
      <script type="text/javascript">
         window.onload = function() {
            xeno.remoting.web.Engine.setReverseAjaxEnabled(true);

            document.getElementById("btnTriggerCallback").onclick = function(evt) {
               com.helloworld.web.SampleService.performCallback();
            };
         };

         var updateOrganization = function(organization, timestamp) {
            var content = "<p>" + timestamp + "</p>";

            content += "<p><b>Organization</b></p>";
            content += "Code: " + organization.code + "<br />";
            content += "Name: " + organization.name + "<br />";

            for(var i in organization.accounts) {
               content += "<p><b>Account</b></p>";
               content += "ID: " + organization.accounts[i].id + "<br />";
```

```
                content += "Name: " + organization.accounts[i].name + "<br />";
            }

            document.getElementById("txtOutputConsole").innerHTML = content;
        };
    </script>
</head>
<body>
    <input id="btnTriggerCallback" type="button" value="Trigger Callback" />
    <div id="txtOutputConsole"></div>
</body>
</html>
```

In this page, we put a button "btnTriggerCallback" to trigger the server side to call the JavaScript function "updateOrganization" at the client side with arguments and the "txtOutputConsole" to track the result.

2.3.2 Process the JavaScript Callback

Continue working on the Java class "com.helloworld.web.SampleService", and add another web method "performCallback" into it:

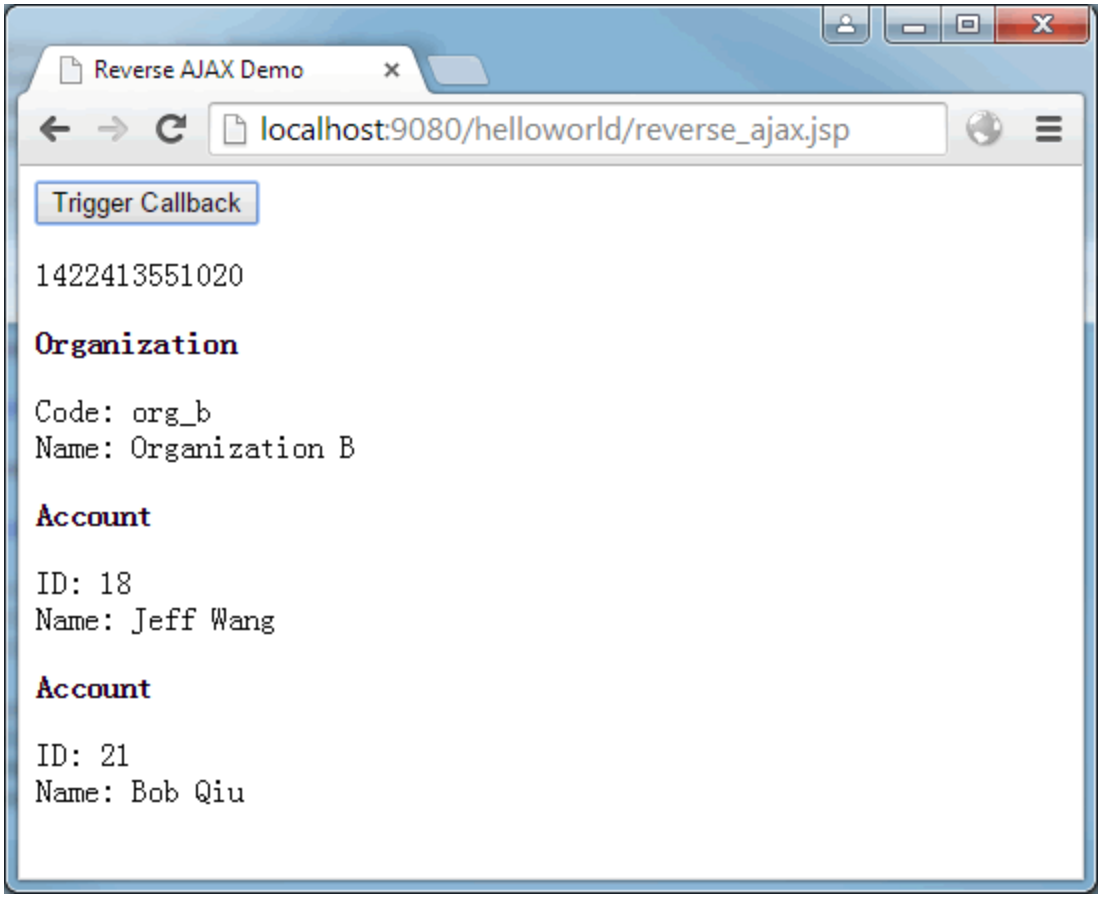
```
@WebMethod
public void performCallback() {
    Account acct1 = new Account();
    acct1.setId(18);
    acct1.setName("Jeff Wang");
```

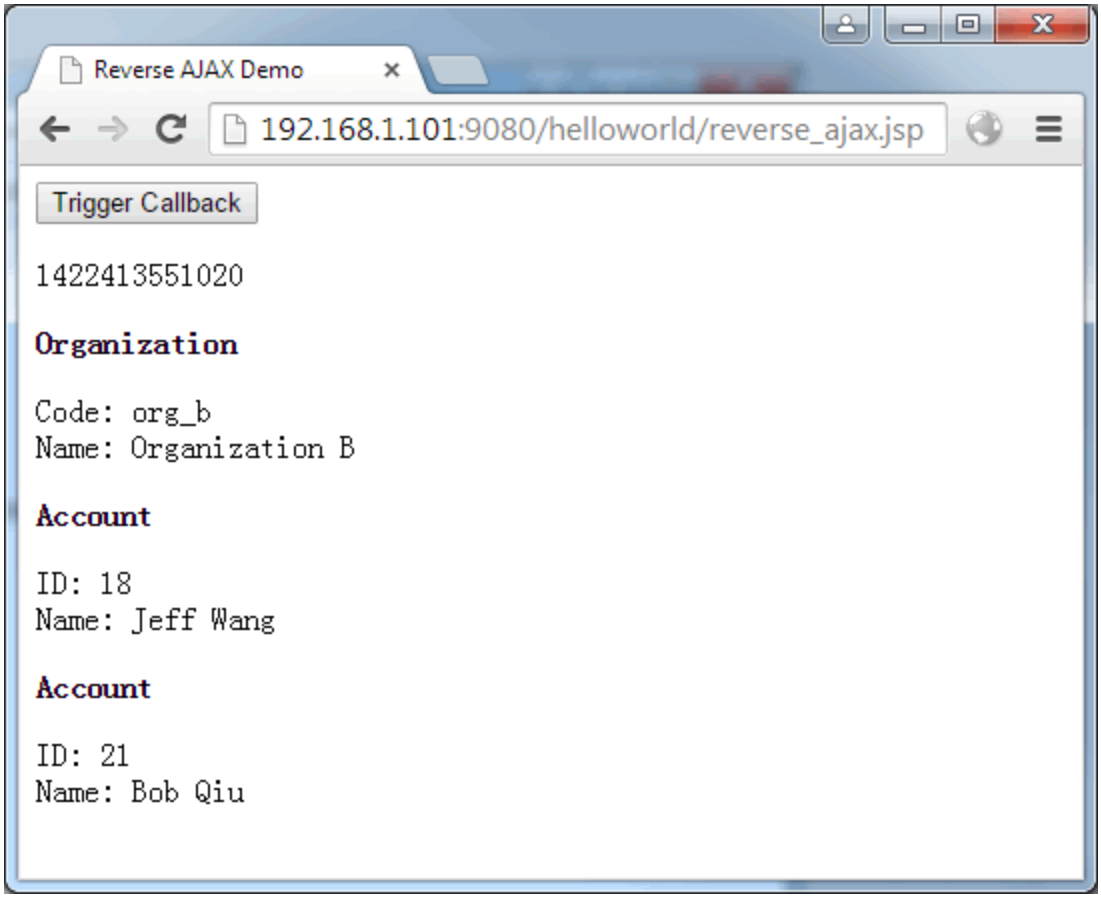
```
    Account acct2 = new Account();
    acct2.setId(21);
    acct2.setName("Bob Qiu");

    List<Account> accounts = new ArrayList<Account>();
    accounts.add(acct1);
    accounts.add(acct2);

    Organization organization = new Organization();
    organization.setCode("org_b");
    organization.setName("Organization B");
    organization.setAccounts(accounts);

    JavaScriptCallback callback = Browser.withPage("/reverse_ajax.jsp");
    callback.invoke("updateOrganization", organization, new Date().getTime());
}
```

From the highlighted code snippet, all pages with the URI "/reverse_ajax.jsp" will be notified after this method has been executed.

Start up the server again and type "http://localhost:9080/helloworld/reverse_ajax.jsp" in the browser's address bar, and use another machine to open another browser and type the address "http://192.168.1.101:9080/helloworld/reverse_ajax.jsp"[1]. Then click the button "Trigger Callback" on the page of either machine to see the similar result according to below pictures.

[1] Assume the server's IP address of the "helloworld" application has been deployed is "192.168.1.101".


Reverse AJAX Demo
localhost:9080/helloworld/reverse_ajax.jsp
Trigger Callback
1422413551020
Organization
Code: org_b
Name: Organization B
Account
ID: 18
Name: Jeff Wang
Account
ID: 21
Name: Bob Qiu



For more information about the reverse AJAX, please see "3.3 Reverse AJAX".

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 requests & responses operation. 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>
    <async-supported>true</async-supported>
    <load-on-startup>1</load-on-startup>
```

```
    <init-param>
        <param-name>characterEncoding</param-name>
        <param-value>GB2312</param-value>
    </init-param>
    <init-param>
        <param-name>debugMode</param-name>
        <param-value>true</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, all 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. Furthermore, you could use "<contextPath>/xr/index.htm" to see all available remote proxies.	false

includeJsonLibrary	A parameter to determine whether this Servlet will take "json2.js" as an implementation or not. If the value of this parameter sets to "true" (case insensitive), the content of "json2.js" will be exported. For some modern browsers (e.g. IE8+, Firefox, Chrome, etc.), JSON has already be implemented and embed in the browser. Please see "3.2.2.2 Serialization / Deserialization" for more information.	false
scanGlobalDependencies	A parameter to determine whether this Servlet will scan global dependencies directory (e.g. for Apache Tomcat, <Installation Root>\lib) to find remote proxy classes. In most cases, you will not allow to put dependencies to that directory, so there is no need to take time to scan it.	false
reverseAjaxMaxRetryCount	A parameter to determine the maximum continuous error retry count when the reverse AJAX thread failed. If this value provides, the reverse AJAX thread will be stopped after specific continuous error retry count occurs to reduce the network traffic. 0 or less than 0 means unlimited retry count.	0
reverseAjaxIdleLiveTime	A parameter to determine the time (in millisecond) of idle reverse AJAX thread live. If this value provides, each reverse AJAX thread will be aborted and setup a new request thread when there is no server side response occurs during this time period. 0 or less than 0 means unlimited idle live time. It is recommended this value should be set less than the session timeout to prevent the reverse AJAX to be inactivated automatically.	0

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 methods 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 Methods" 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 on JSP 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 to be able to called from the client side directly, it should meet below requirements:

- The class should provide the public modifier, and the abstract class or the 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" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <xr:proxy>
            import test.MyService;
        </xr:proxy>
```

```
    </head>
</html>
```

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

```
xeno.remoting.web.Engine.registerClass("test.MyService");


test.MyService.calculate = function(arg0, arg1, opts) {
    return xeno.remoting.web.Engine.processRemoteCall("test.MyService", "calculate", [arg0, arg1], opts);
};


test.MyService.check = function(arg0, opts) {
    return xeno.remoting.web.Engine.processRemoteCall("test.MyService", "check", [arg0], opts);
};
```

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, {
    onSuccess: function(result, xhr) {
        // Implements your logic code here...
    },
    onError: function(fault, xhr) {
        // Implements your logic code here...
    }
});
```

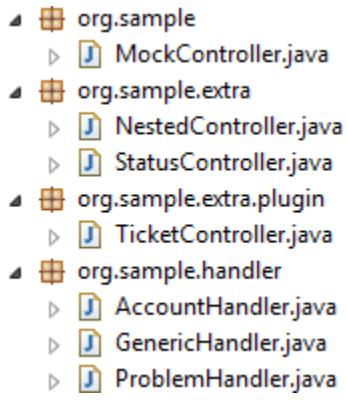
For each web method translated JavaScript function, the last optional argument allows you to do some settings of the remote call, please see "3.2.2.1 Calling Options" for more information.

3.2.1.1 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>proxy</name>
   <tag-class>xeno.remoting.web.ProxyJspTag</tag-class>
</tag>
```

Contents between 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:proxy>
         import org.sample.extra.NestedController;
         import org.sample.extra.StatusController;
         import org.sample.handler.GenericHandler;
      </xr:proxy>
   </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:proxy>
   import org.sample.extra.*;
   import org.sample.handler.GenericHandler;
</xr:proxy>
```

And the line comment "//" is supported in the body content of this tag as well, in below code snippet, the "org.sample.handler.GenericHandler" will be excluded:

```
<xr:proxy>
   import org.sample.extra.*;
```

```
   // import org.sample.handler.GenericHandler;
</xr:proxy>
```

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:proxy>
         import org.sample.extra.*;
         import org.sample.handler.GenericHandler;
      </xr:proxy>
      <xr:proxy>
         import org.sample.MockController;
         import org.sample.extra.plugin.TicketController;
         import org.sample.handler.*;
      </xr:proxy>
   </head>
</html>
```

3.2.2 Web Method

"@WebMethod", the Java class "xeno.remoting.bind.WebMethod", this annotation uses to mark a method as a web method, 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, see below Java code:

```
package net.corp;


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


@RemoteProxy
public class OrgService {


   @WebMethod
   public Organization find(Account acct, String auth) {
      // Implements your logic code here...
   }
}
```

Below JavaScript code is after the translation:

```
xeno.remoting.web.Engine.registerClass("net.corp.OrgService");
```

```
net.corp.OrgService.find = function(arg0, arg1, opts) {
   return xeno.remoting.web.Engine.processRemoteCall("net.corp.OrgService", "find", [arg0, arg1], opts);
};
```

Arguments type and position are the same as those at the server side, an optional argument "opts" will be generated for each translated JavaScript function, please see "3.2.2.1 Calling Options" for more information.

3.2.2.1 Calling Options

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:

```
net.corp.OrgService.find = function(arg0, arg1, opts) {
   return xeno.remoting.web.Engine.processRemoteCall("net.corp.OrgService", "find", [arg0, arg1], opts);
};
```

Below code snippet is an example of calling options usage:

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

// Processes the remote call with calling options.
var progress = net.corp.OrgService.find(acct, "kfeng", {
   timeout: 120000,
```

```
    onSuccess: function(result, xhr) {
        // Implements your logic code here...
    }
});

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

Calling options use in the translated JavaScript function is called local calling options, there are also some global calling options set through functions for the application's high level usage, see below table:

Local Prop.	Global Function	Description
timeout	xeno.remoting.web.Engine.setTimeout	A millisecond describes the request timeout.
onBeforeSend	xeno.remoting.web.Engine.setBeforeSendHandler	The function will be triggered before the request to be sent.
onSuccess	xeno.remoting.web.Engine.setSuccessHandler	The function will be triggered when the request success. (No error or timeout occurs.)
onError	xeno.remoting.web.Engine.setErrorHandler	The function will be triggered when error occurs during the request.
onTimeout	xeno.remoting.web.Engine.setTimeoutHandler	The function will be triggered when the request timeout.
onComplete	xeno.remoting.web.Engine.setCompleteHandler	The function will be trigger when the request completes whatever it is success, error or even timeout.

To enable both local & global calling options work together well, each option item should follow below rules:

- The option item in local calling options of the remote call always has the highest priority, it will override that one in global calling options if has.
- If there is no specific option item set in calling options of the remote call, use that one in global calling options if has.
- If there is no specific option item set in both calling options of the remote call and global calling options, this option item feature will be ignored.

Below table is an example of the remote call with full calling options setting:

```
var timeoutVal = 12000;
var beforeSendFunc = function(header) { };
var successFunc = function(result, xhr) { };
var errorFunc = function(fault, xhr) { };
var timeoutFunc = function(xhr) { };
var completeFunc = function(status, xhr) { };
```

Local Calling Options Setting	**Global Calling Options Setting**

net.corp.OptionService.test({
timeout: timeoutVal,
onBeforeSend: beforeSendFunc,
onSuccess: successFunc,
onError:errorFunc,
onTimeout: timeoutFunc,
onComplete: completeFunc
});
	xeno.remoting.web.Engine.setTimeout(timeoutVal);
	xeno.remoting.web.Engine.setBeforeSendHandler(beforeSendFunc);
	xeno.remoting.web.Engine.setSuccessHandler(successFunc);
	xeno.remoting.web.Engine.setErrorHandler(errorFunc);
	xeno.remoting.web.Engine.setTimeoutHandler(timeoutFunc);
	xeno.remoting.web.Engine.setCompleteHandler(completeFunc);

Please see JavaScript API documents in the folder "jsdoc" of the released distributed files for more information, such as callback functions arguments, object reference, etc.

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, for most modern browsers (e.g. Internet Explorer 8+, Firefox, Chrome, etc.), actually they has already implemented serialization & deserialzation of JavaScript object and embed this feature in the browser directly. For older browsers (e.g. Internet Explorer 7) support, you could add the initialization parameter "includeJsonLibrary" of the configuration for the Servlet "xeno.remoting.web.MessageServlet", and the "json2.js" (https://github.com/douglascrockford/JSON-js) library will be used as a JSON adopter, please see "3.1 Configurations" for more information.

At the server side, the "Jackson JSON Processor" (http://jackson.codehaus.org) 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, 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 argument with the class type of 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 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.

3.3 Reverse AJAX

Generally, the web was not designed to allow the server side to make connections to the client side, so it can be tricky to get data to a browser in a timely manner. The good news is from the Servlet 3.0, the asynchronous context feature has been implemented, the XR take it for the reverse AJAX. A new attribute "async-supported" has been introduced to

the Servlet's definition. Any time if you want it you should set this attribute to "true", see the below highlighted code snippet:

```
<servlet>
   <servlet-name>MessageServlet</servlet-name>
   <servlet-class>xeno.remoting.web.MessageServlet</servlet-class>
   <async-supported>true</async-supported>
   <load-on-startup>1</load-on-startup>
</servlet>
```

Then for those pages which want to receive the server side data push back, below highlighted code snippet should be added:

```
<%@ 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:proxy />
      <script type="text/javascript">
         window.onload = function() {
            xeno.remoting.web.Engine.setReverseAjaxEnabled(true);
         };
      </script>
   </head>
</html>
```

The XR provide a Java class "xeno.remoting.web.Browser" with static methods for you to notify the client side's JavaScript functions, and you should encapsulate them at the server side and process the callback, please see Java API documents in the folder "javadoc" of the released distributed files.

Session IDs, page URIs and their combinations are used to locate which clients you want to notify, see below code snippet:

```
@WebMethod
public void updateOrganization() {
    Account user = new Account();
    user.setId(9);
    user.setName("Jame He");

    Organization organization = new Organization();
    organization.setCode("org_c");
    organization.setName("Organization C");

    String sessionId = "363F33878FD45FBFF63850D711031691"
    String pageUri = "/home/list";
    JavaScriptFunction func1 = new JavaScriptFunction("org.corp.homePage.refresh", organization, user);
    JavaScriptFunction func2 = new JavaScriptFunction("findAllOrganizations");

    Browser.with(sessionId, pageUri).invoke(func1, func2);
}
```

When above code to be executed, the client with the specific session ID and page URI will be invoked, and JavaScript functions "org.corp.homePage.refresh" and "findAllOrganizations" will be called with arguments on the page. Like the remote call,

data serialization & deserialization use the same mechanism for the reverse AJAX, please see "3.2.2.2 Serialization / Deserialization" for more information.

For the session ID, it is the same meaning of the HTTP session ID, get through the interface "javax.servlet.http.HttpSession".

For the page URI, it is the value comes from the browser address bar, and start after your application context path. For an example, if current URL is showing as below:

```
http://www.abc.com:8030/testapp/account/list#bookmark?p1=1&p2=a
```

Assume your application has been deployed with the context path "testapp", then the page URI to be used to notify the client side is "/account/list".

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()
- xeno.remoting.web.MessageServlet.getCurrentPageUriThread()

Reverse AJAX will be inactivated after session timeout.

If you configure the session timeout in the "web.xml" less than the value of the initialization parameter "reverseAjaxIdleLiveTime" for the Servlet "xeno.remoting.web.MessageServlet", and if there is no any data exchange during that period, the reverse AJAX will be inactivated because of the HTTP session life cycle mechanism.

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 reverse AJAX 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.

This issue exists in many widely used web frameworks (e.g. Spring MVC, Struts, DWR, etc.), currently there is no perfect solution. 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.getWebApplicationServletContext();
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".

Is there any possibility to monitor the reverse AJAX status?

There are 2 JavaScript handlers are used to do it, see below code snippet:

```
xeno.remoting.web.Engine.setReverseAjaxProgressHandler(function() {
   // Implements your logic code here...
});

xeno.remoting.web.Engine.setReverseAjaxCompleteHandler(function() {
   // Implements your logic code here...
});
```

For more information about these handlers, please see JavaScript API documents in the folder "jsdoc" of the released distributed files.

Is there any possibility to see all available remote proxies?

In "3.1 Configurations" you will notice there is an initialization parameter "debugMode", if you set this value to "true", when your application startup, you could see all available remote proxies and their web methods by typing this URL in your browser address bar: "<contextPath>/xr/index.htm", see below screenshot: