

DirectJN_{gine} 1.3

beta 1 for ExtJs

User's Guide

DirectJN_{gine} 1.3 beta 1 for ExtJs 3.2.1

Copyright © Pedro Agulló Soliveres, 2009, 2010

1. What is DirectJNgin?	4
DirectJNgin in 30 seconds	4
2. Features	6
3. Installing DirectJNgin in your system	7
4. Configuring a new project to use DirectJNgin	8
5. DirectJNgin by example	9
Running ExtJs examples against DirectJNgin	9
Step by step "Hello world" with DirectJNgin	9
Step 1: configure DirectJNgin servlet in web.xml	9
Step 2: make our server methods available to Javascript	10
Step 3: decide what your server methods will be like	11
Step 4: write the server methods in Java	12
Step 5: tell DirectJNgin where to look for server methods	13
Step 6: register your methods with Ext Direct	13
Step 7: call the methods from Javascript	13
Other issues	14
Wrap up	15
6. Form posts and DirectJNgin	16
Step 1: writing the Javascript code that loads/submits data	16
Step 2: configure the form	17
Step 3: write the Java code for loading form data	17
Step 4: write the Java code for submitting form data	18
An alternative way to submit data	19
7. Polling Providers and DirectJNgin	21
8. DirectStore with DirectJNgin	23
Passing parameters to DirectStore's directFn	24
Passing unknown parameters to DirectStore's directFn	25
9. Google AppEngine support	26
10. Servlet configuration	27
11. State management and session/application scope support	30
Stateful actions	30
Generating multiple action instances for a Java class	31
12. Customizing data conversion and Gson configuration	32
Changing Gson's configuration	32
Adding your own serializers/deserializers	33
Closing thoughts	35

13. Handling JSON data directly	36
14. Adding actions and methods programmatically	37
15. Checking client-side parameters	40
16. DirectJNgin Optimization.....	42
Optimizing api files generation and usage	42
Optimizing batch requests handling using multiple threads	43
17. Diagnostics and logging	44
Measuring request execution time	44
Understanding which logs go together.....	44
18. How reliable is all of this?	45
19. Licensing.....	46

Acknowledgments

I would like to thank José María Martínez and Judith Marcet for their feedback, as well as for the nice time we have together as part of the *softwarementors agile team*.

Thanks!

1. What is DirectJNgin?

DirectJNgin (or *djn*, for short) is a Java implementation of the *Ext Direct API*. This API allows applications using *ExtJs* to call Java methods from Javascript almost transparently, making things that used to be more or less cumbersome or time consuming much easier.

New to Ext Direct?

If you are new to *Ext Direct*, please check the *ExtJs* documentation and examples, or go to <http://extjs.com/blog/2009/05/13/introducing-ext-direct/> or <http://extjs.com/products/extjs/direct.php> for details. From now on, we will assume that you have a basic understanding of *Ext Direct*, as well as of its “vocabulary” (*action*, *method*, etc.)

DirectJNgin in 30 seconds

Now, how is *everyday* life with *DirectJNgin*?

Let's assume that you want your Javascript code to call a **sayHello** Java method, which will receive a person name and return a greeting message. That is as easy as writing the following Java code:

```
public class Action1 {
    @DirectMethod
    public String sayHello( String name) {
        return "Hello, " + name + ". Nice to meet you!";
    }
}
```

Basically, you write your Java code without much concern for whether it will be called by some Javascript code living in a remote server or not. The secret? Using the **@DirectMethod** annotation *DirectJNgin* provides. Once you do that, you will get automatic remote method call support: no need for boring, cumbersome and error-prone glue code.

Using the newly written method is as easy as writing the following Javascript:

```
Action1.sayHello( "Pedro", function(p, response) {
    Ext.MessageBox.alert( "Greetings", response.result );
});
```

The only remarkable thing here is the function passed as a parameter to the **Action1.sayHello** method, a Javascript function that will be called when the server response arrives, to avoid blocking the application.

If you look at the client and server code, you will notice that there is no “extra fat”: what you see is what you get.

Of course, things can't be that easy, we are talking about *remote communication*, *Javascript on one side*, *Java on the other*, and *the net in the middle*. So, yes, there will be things to configure, issues to take into account, and best practices to follow in order to stay sane.

But once you start to master them, things will be almost *that* easy.

2. Features

In its current version, we think *DirectJNgin* is very much feature-complete, providing the following features:

- Annotations-based configuration.
- Support for all kinds of requests:
 - JSON requests.
 - Batched JSON requests.
 - Simple Form Posts (no files to upload).
 - Upload Form Posts.
 - PollingProvider requests.
- Multithreaded processing of batched requests, for better performance.
- Method name based configuration.
- Automatic Javascript API files generation.
- Detailed User's Guide.
- Demos: implements all the server side functionality required to run the demos provided by ExtJs in *examples/direct*.
- API files consolidation: consolidate several provider apis into one file to minimize network traffic.
- API files minification and comment removal.
- Support for programmatic Api generation + hook to allow custom generation on startup.
- Debug mode support.
- Fully automated tests: more than 80 unitary tests are executed every time there are changes to the code.
- Tested against Firefox, Internet Explorer, Safari and Chrome.
- Possibility to call public, private, package and protected instance or static methods in public or private classes.
- Detailed logging, to support easy diagnostic of problems and performance measurements.
- Open Source, free for commercial projects too.
- Stateful actions: actions can have session and application scopes.
- Support for accessing the current session, servlet context, servlet configuration, etc., from within action methods.
- Support for Google AppEngine.

3. Installing DirectJNgine in your system

To install the library, decompress the appropriate *directjengine.xxx.zip* file into a directory (*xxx* is the library version).

You will need to install *ExtJs* too: due to licensing issues, we can't redistribute *ExtJs* with this library. You will have to download it from <http://extjs.com>. Just make sure you are using the right version, please!

Once installed, copy it in an *extjs* subdirectory under the *WebContent* directory in our distribution.

4. Configuring a new project to use DirectJNgin

In order to use DirectJNgin in a new application, you will need to add the following JARs to your web app *WEB-INF/lib* directory:

- *DirectJNgin* itself: the file is *deliverables/directjngine.xxx.jar*, where *xxx* is the version number, such as *1.0*.
- Third party libraries used by *DirectJNgin*:
 - All JARs in the *lib* directory. Please, ignore its subdirectories.
 - All JARs in the *lib/runtimeonly* directory.

If you use the client-side parameter checking debug-time support (take a look at the *Checking client-side parameters* chapter), you will need to add several javascript files to your web app:

- *djn-remote-call-support.js*: put it in *djn/djn-remote-call-support.js* under the web root directory.
You can find this file in *deliverables/djn-remote-call-support.js*.
- *ejn-assert.js*: put it in *ejn/ejn-assert.js* under the web root directory.
You can find this file in *deliverables/ejn-assert.js*.

Finally, you will need to provide the *ExtJs* files: due to licensing issues you need to download them separately, and then install them in your web app.

The enclosed demo app might use some other files, but they are *not* needed in order to use *DirectJNgin*, and they are *not* part of it.

Compilation only JARs

In order to compile, you might need the libraries in *lib/compiletimeonly* or not, depending on how your environment is set up.

This directory contains JAR files the web server will already provide, so you must not add them to your app *WEB-INF/lib* directory.

5. DirectJNgin by example

ExtJs provides several examples of how to use the *Direct* API. You can find them in the *extjs/examples/direct* subdirectory. These examples work beautifully...but they use PHP in the server side.

However, it is very easy to make them work against Java in the server side, using *DirectJNgin*. In fact, we will use them in order to show how *DirectJNgin* works.

For *ExtJs* examples to work, you will need to modify slightly several files, as follows:

- *direct.php*: substitute the *php/api.php* string with *../demo/Api.js*.
- *direct-form.php*: substitute *php/api.php* with *../demo/Api.js*.
- *direct-tree.php*: substitute *php/api.php* with *../demo/Api.js*.
- *direct.js*: substitute '*php/poll.php*' with *Ext.app.POLLING_URLS.message* (yes, remove the single quotes, unlinke in the prior modifications)

That's all. From now on, the examples will work directly with against a *DirectJNgin* based backend.

In fact, we have provided the application we use to run the automated *DirectJNgin* tests with the distribution, and have added support to run the *ExtJs Direct* demos once "converted" to *DirectJNgin*.

Running ExtJs examples against DirectJNgin

To run *Ext Direct* examples you need to install the *djn_test* war. To do that, follow these steps:

1. Install our *demos/test_war/djn_test.war* in your web server.
2. Start the web application, making sure it is decompressed.
3. Stop the web application, and add the *ExtJs* libraries in an *extjs* subdirectory under the web root directory of the decompressed war.

Do not forget the *extjs examples* directory, as we use some of its gadgets and examples.

4. Modify the *extjs/examples/direct* files as explained above.
5. Restart the web application.
6. Navigate to the *demo/DjnDemo.html* page: you can run all examples from there.

Step by step "Hello world" with DirectJNgin

Step 1: configure DirectJNgin servlet in web.xml

Open the *WebContent/web.xml* file included with your *DirectJNgin* distribution, and take a look at the following lines, used to configure the *DirectJNgin* servlet:

```
<!-- DirectJNgin servlet -->
<servlet>
  <servlet-name>DjnServlet</servlet-name>
```

```
<servlet-class>
    com.softwarementors.extjs.djn.servlet.DirectJNginServlet
</servlet-class>

<init-param>
    <param-name>providersUrl</param-name>
    <param-value>djn/directprovider</param-value>
</init-param>

<!-- more parameters... -->

<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>DjnServlet</servlet-name>
    <url-pattern>/djn/directprovider/*</url-pattern>
</servlet-mapping>
```

Of course, there are more parameters, but ignore them by now, please.

The servlet `url-pattern` must always end with “/*”, and we recommend that you use the default url, `/djn/directprovider/*`.

The `providersUrl` parameter is essential, because it will be used by *Ext Direct* to communicate with *DirectJNgin*: make sure it is the same as the servlet’s `url-pattern`, minus the ending “/*”.

Step 2: make our server methods available to Javascript

Open the *direct.php* file: it is a plain html file, so do not worry. We want to call your attention to the following line:

```
<script type="text/Javascript" src="../../../demo/Api.js"></script>
```

This line must be there, because *Api.js* is the Javascript file that provides access to the Java methods we implemented in the server. How do you write it? Well, you don’t, *DirectJNgin* will generate it on your behalf.

But, how does *DirectJNgin* know how to create it? Open *web.xml* again, and take a look at the following lines:

```
<init-param>
    <param-name>apis</param-name>
    <param-value>
        test,
        demo,
    </param-value>
</init-param>

<init-param>
    <param-name>demo.apiFile</param-name>
    <param-value>demo/Api.js</param-value>
</init-param>

<init-param>
    <param-name>demo.apiNamespace</param-name>
    <param-value>Ext.app</param-value>
```

</init-param>

Our demo application provides two different *provider apis*, one for test methods (called *test*), and another one for demo methods (called *demo*). You must provide the **api s** parameters in order to tell *DirectJNgine* the Direct provider apis you want to define.

You specify the file the *demo* api will be writtent to using the **demo. api File** parameter. Its value is the file path relative to the web app root directory. In our demo, since it is *demo/Api.js*, *DirectJNgine* will generate *Api.js* in the *WebContent/demo* directory of your installation.

Besides, you can specify the namespace where the api will live, using the **demo. api Namespace** parameter.

Of course, if we were setting the *test* api configuration, the parameter names would have been *test.apiFile* and *test.apiNamespace*, respectively.

For additional configuration parameters, check the *Servlet configuration* chapter.

Alternative API handling

DirectJNgine generates javascript files containing the API used to access the Java code, but there is in alternative way to access the API. Please, check the chapter explaining how to support *Google's AppEngine* for details on how to access the API in an environment in which it is not possible to create/update files in the server.

Step 3: decide what your server methods will be like

If you open the *extjs/examples/direct/direct.js* example file, you will find that the demo calls two server methods, as follows:

```
TestAction.doEcho(text.getValue(), function(result, e){
// ...
TestAction.multiply(num.getValue(), function(result, e){
// ...
```

As you already know, the functions at the end of the method calls are the callbacks that will be invoked by *Ext Direct* to handle the server result. Ignore them, they are not passed to the server – and we will get back to them later.

Ignoring the functions, the call would be a lot more like

```
TestAction.doEcho(text.getValue());
// ...
TestAction.multiply(num.getValue());
// ...
```

TestAction.doEcho receives a string and returns it. **TestAction.multiply** receives a string, tries to multiply it by eight, and returns the result as a number. And, yes, that means the server can

receive a string that is not a valid number, so we will have to take care of this in some way. But, again, let us postpone those details.

Step 4: write the server methods in Java

This is the Java code for the methods:

```
public class TestAction {

    @DirectMethod
    public String doEcho( String data ) {
        return data;
    }

    @DirectMethod
    public double multiply( String num ) {
        double num_ = Double.parseDouble(num);
        return num_ * 8.0;
    }

    public static class Node {
        public String id;
        public String text;
        public boolean leaf;
    }
}
```

We have grouped the methods for the *TestAction* action in a **TestAction** class. But if you need to have a class that has not the same name as the action, use the **@DirectAction** annotation as follows:

```
@DirectAction( action="TestAction")
public class MyTestActionClass {
    // ...
}
```

We have implemented the methods with exactly the same names the Ext Direct methods have, adding the **@DirectMethod** annotation to them.

Again, if you had to write the Java methods with a different name, you could use the **@DirectMethod** annotation as follows:

```
@DirectMethod( method="multiply")
public double myMultiplyMethod( String num ) {
    // ...
}
```

If you look at **doEcho**, you will find that the code is absolutely straightforward, it receives a string and returns it. Nothing to worry about - unless there is some internal server error, but let me talk about that later.

Now, if you take a look at **multiply**, things get a bit more interesting. What if we receive as an argument something like *"hello world"*? If that's the case, the call to **Double.parseDouble** will throw a **NumberFormatException**. *DirectJNgine* will take care of this, and return information that

allows *Ext Direct* to know that something went wrong, so that your Javascript code can handle the problem.

Coping with method's results will be explained later, just let me give you reassurance that even unexpected server errors can be handled very easily.

Step 5: tell DirectJNgin where to look for server methods

Now, how does DirectJNgin know what are the classes that contain action methods, so that it can look for all those nice annotations?

We use the servlet `demo. classes` parameter to tell djn the classes to check, as follows:

```
<init-param>
  <param-name>demo. classes</param-name>
  <param-value>
    com.softwarementors.extjs.djn.demo.Poll,
    com.softwarementors.extjs.djn.demo.TestAction,
    com.softwarementors.extjs.djn.demo.Profile
  </param-value>
</init-param>
```

Remember, here *demo* is the api definition for ExtJs Direct examples, if we were configuring the *tests* api, the parameter to configure would have been `tests. classes`.

Step 6: register your methods with Ext Direct

In order for ExtJs to be able to call our java methods we need to register a remoting provider. The way it's been done in *direct.js* is as follows:

```
Ext.Direct.addProvider(
  Ext.app.REMOTING_API,
  // ...
);
```

Please, note that `Ext. app` is the namespace we specified via the `demo. apiNamespace` servlet parameter, and `REMOTING_API` is the provider configuration we have provided in *Api.js* (we always use the same name, `REMOTING_API`, to make your life easier).

Step 7: call the methods from Javascript

The *WebContent/extjs/examples/directscript.js* file calls our `TestAction. doEcho` Java method as follows:

```
TestAction.doEcho(text.getValue(), function(result, e) {
  var t = e.getTransaction();
  out.append(String.format(
    '<p><b>Successful call to {0}.{1} with ' +
    ' response: </b><xmp>{2}</xmp></p>',
    t.action, t.method, Ext.encode(result)));
  out.el.scrollTo('t', 100000, true);
});
```

Note we are passing a second parameter, a Javascript function that will be called with the data returned by the server (it is not sent to the server!). We need to use a function to handle the result because remote calls are asynchronous, as it would not be a good idea to block the program waiting for the result.

The function receives the call result in the `result` parameter, and additional data in the `e` event, including the transaction, which holds the invoked *action* and *method* names, among other things.

The call to `multiply` is a bit more interesting, because it shows how to handle server errors:

```
TestAction.multiply(num.getValue(), function(result, e) {
    var t = e.getTransaction();
    if(e.status) {
        out.append(String.format(
            '<p><b>Successful call to {0}.{1} with ' +
            'response: </b><xmp>{2}</xmp></p>',
            t.action, t.method, Ext.encode(result)));
    } else {
        out.append(String.format(
            '<p><b>Call to {0}.{1} failed with message: </b><xmp>{2}</xmp></p>',
            t.action, t.method, e.message));
    }
    out.el.scrollTo('t', 100000, true);
});
```

Here, we get the event transaction and check its `status`: if it is true, the execution of the application method finished successfully, and you can safely use the `result`. Else, the execution finished with a server error. For all intents and purposes this is considered to be a server error by *DirectJNgin*, and is notified as such to *Ext Direct*.

When there is a server error, the event received by the function handling the result will have a `message` field, providing some kind of explanation about the problem, and if in debug mode, a `where` field providing additional information. This field will always be an empty string when not in debug mode.

DirectJNgin provides as `message` the name of the Java exception and the message it contains, while `where` contains the full stack trace of the exception.

Other issues

We mentioned that while in debug mode you will get additional information about server errors. Now, how do you specify whether the application is in debug mode or not? Just use the servlet debug parameter, as follows:

```
<init-param>
  <param-name>debug</param-name>
  <param-value>true</param-value>
</init-param>
```

Finally, in case you are wondering what the generated api file looks like, here is (part of) the code:

```
Ext.namespace( 'Ext.app' );

Ext.app.PROVIDER_BASE_URL=window.location.protocol + '//' +
```

```
window.location.host + '/' +
(window.location.pathname.split('/')[1]) + '/' + 'djn/directprovider';

// ...

Ext.app.REMOTING_API = {
  url: Ext.app.PROVIDER_BASE_URL,
  type: 'remoting',
  actions: {
    TestAction: [
      {
        name: 'doEcho' /*( String) => String */,
        len: 1,
        formHandler: false
      },
      {
        name: 'multiply' /*(String) => double */,
        len: 1,
        formHandler: false
      }
    ]
  }
  // ...
}
```

We think that it might be interesting for the api users to know the Java types of the method parameters and result, and therefore we added it to the generated source code: the parameter types are specified in parentheses, while the return type is added after the “=>” string.

Wrap up

Now, that’s a lot of steps!

However, once you have finished with basic configuration, you will find that writing a new method involves just three steps: thinking what your method has to look like, writing the java method itself, and calling it from Javascript. This is not much more difficult than creating a new Java method to be used by other Java code in your application.

6. Form posts and DirectJNgin

Form processing with *DirectJNgin* is quite easy. But, before we start to take a look at it, make sure that you have followed the steps we outlined before to “port” the demos from PHP to a *DirectJNgin* based Java backend.

Now, let's check how to load and submit form data using the demo included with *ExtJs* itself, in its *demo/direct/direct-form.js* file.

Step 1: writing the Javascript code that loads/submits data

To begin with, we are going to study only the first form in the demo (see **Figure 1**). The data is some kind of *basic person info* that includes a **name**, an **email** and a **company**. Even if it is not visible, there is an **id** that uniquely identifies the person, and an additional hidden **foo** field, just to make things interesting.

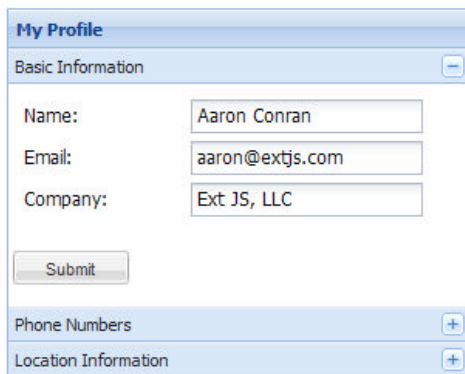


Figure 1. Loading and submitting data

Now, how are we going to load that basic person info? The first thing that comes to mind is to pass the person **id** to the **load** method. And, again, to make things interesting, we will pass a *foo* value.

The Javascript code to load the form data will be as follows:

```
basicInfo.getForm().load({
  params: {
    foo: 'bar',
    uid: 34
  }
});
```

You pass the parameters to load as a **params** object, not directly, that's the way it is.

The next step is to submit the form data for processing. Here is the Javascript code:

```
basicInfo.getForm().submit({
  params: {
    foo: 'bar',
    uid: 34
  }
});
```


But, wait! Where are the **name**, **email** and **company** we should be passing to the server? Well, the form has fields with *exactly* those names: therefore *ExtJs* will pass their value automatically to the submit handler waiting for them in the backend. But this *only* happens for the submit method!

Step 2: configure the form

Ok, now, how does *Ext Direct* know what Java methods to call? You tell *Ext Direct* via an **api** configuration parameter, as follows:

```
api: {  
    // The server-side method to call for load() requests  
    load: Profile.getBasicInfo,  
    // The server-side must mark the submit handler as a 'formHandler'  
    submit: Profile.updateBasicInfo  
}
```

You will have to tell *Ext Direct* in what order will the **load** parameters be passed to the Java method. Use the **paramOrder** configuration parameter to make sure that the **id** will be the first parameter and **foo** the second one, as follows:

```
paramOrder: [ 'id' , 'foo' ]
```

And, no, you need not specify the order in which the fields and parameters will be passed to the Java method that handles the form submit, they are passed as a map of name-value to the Java method.

Step 3: write the Java code for loading form data

You can find the Java code in our *Profile.java* file. Lets' take a look at our **getBasicInfo** method, the one that handles data loading:

```
@DirectMethod  
public BasicInfo getBasicInfo( Long userId, String foo ) {  
    assert userId != null;  
    assert foo != null;  
  
    BasicInfo result = new BasicInfo();  
    result.data.foo = foo;  
    result.data.name = "Aaron Conran";  
    result.data.company = "Ext JS, LLC";  
    result.data.email = "aaron@extjs.com";  
    return result;  
}
```

Remember, we told *Ext Direct* that the **id** is the first parameter, and **foo** the second one, and that's how we have to write the Java method. Besides, the method must be annotated with the **@DirectMethod** annotation.

If you look carefully, you will see that we are returning a **BasicInfo** object to the client. The code for it is as follows:

```
public static class BasicInfo {  
    public static class Data {
```

```

        public String foo;
        public String name;
        public String company;
        public String email;
    }

    public boolean success = true;
    public Data data = new Data();
}

```

This looks complicated, but it is not, I promise. What's happening here is that the *Ext Direct* expects data returned from a `load` call to have a certain format.

First of all, *Ext Direct* expects you to provide a `success` value that you set to true if everything went well, and to false if there was a problem. Then, it expects the returned data to be put in a `data` value. Both of them are there in the `BasicInfo` class, which is nothing more than a value holder.

It is you who decides what goes inside of the `data`. In this case, we just have `foo`, `name`, `company` and `email`, *exactly* the names we gave the form fields and/or parameters passed back and forth. We created the `Data` class to reflect this structure, nothing more. By following these conventions *Ext Direct* will perform all magic required to fill the form fields and pass the data to our Java method.

Once your method returns the data, *DirectJNgin* will perform all magic required to send it to *Ext Direct* for processing at the client.

Step 4: write the Java code for submitting form data

In order to handle the form submit we have to define a Java method annotated with `DirectFormPostMethod`. Besides, the Java method must receive two parameters. The first one must be a map of *field name-field value* pairs, representing all form fields, except input file fields.

The second parameter must be a map of *field name-file items* representing only the input file fields: you can access each file using the `FileItem`'s `getInputStream` method. If your form has no input file fields, this map will be empty.

Here is the Java code:

```

@DirectFormPostMethod
public SubmitResult updateBasicInfo( Map<String, String> formParameters,
    Map<String, FileItem> fileFields )
{
    assert formParameters != null;
    assert fileFields != null;

    SubmitResult result = new SubmitResult();

    String email = formParameters.get( "email" );
    result.success = !email.equals("aaron@extjs.com");
    if( !result.success ) {
        result.errors = new HashMap<String, String>();
        result.errors.put( "email", "already taken" );
    }

    result.debug_formPacket = formParameters;
    return result;
}

```

Here, we are performing data validation by checking the **email** value: if it is *aaron@extjs.com*, we will consider that the data is wrong. In that case we set the **success** field to false, and add an entry to the **errors** result, a pair having a *field-name* as key and an *error text* as value.

By following this convention *ExtJs* will handle the error on its own, providing feedback as shown in **Figure 2**.

Figure 2. Validation error, as reported by the server.

The method returns a *SubmitResult* object, defined as follows:

```
private static class SubmitResult {
    public boolean success = true;
    public Map<String, String> errors;
    public Map<String, String> debug_formPacket;
}
```

The interesting thing in this class is the fact that we are using a map to return error information to Javascript. When the email is wrong, the data returned to Javascript will be as follows:

```
result = {
    success : false,
    errors: {
        email: 'already taken'
    },
    // ...
}
```

An alternative way to submit data

Just for the record, there is another way to submit data. The Javascript will be as follows:

```
handler: function() {
    Profile.updateBasicInfo(form.getForm().el, function(result, e) {
        //...
    });
}
```

In this case, we just pass the form's `el` element as the first and only parameter to the remote method. The Java method implementation will not change.

7. Polling Providers and DirectJNgine

Polling providers make it possible to make periodical requests to the server. The example in *extjs/examples/direct/direct.php* creates a polling provider that periodically calls the server to get its current date and time. Let's see how this can be accomplished with *DirectJNgine*.

The first thing you have to do is register the polling provider. This is done in the “ported” version of *extjs/examples/direct/direct.js* as follows:

```
Ext.Direct.addProvider(
    // ...
    {
        type: 'polling',
        url: Ext.app.POLLING_URLS.message
    }
);
```

Here, we are telling *Ext Direct* to create a polling provider whose url will be the one in `Ext.app.POLLING_URLS.message`. We will explain where this comes from in minute, bear with me.

Now, *Ext Direct* will receive a periodic notification sent by the server, which it needs to handle in a callback function, as usual. The code for the callback is as follows:

```
Ext.Direct.on('message', function(e) {
    out.append(String.format('<p><i>{0}</i></p>', e.data));
    out.el.scrollTo('t', 100000, true);
});
```

In the end, *Ext Direct* is just handling an *event*, called *message* in the example. Why *message*? Take a look at the Java code handling the request, and you'll get the answer:

```
@DirectPollMethod( event="message")
public String handleMessagePoll(Map<String, String> parameters) {
    assert parameters != null;

    Date now = new Date();
    SimpleDateFormat formatter =
        new SimpleDateFormat("yyyy/MM/dd 'at' hh:mm:ss");
    return "Current time in server: " + formatter.format( now );
}
```

You will have noticed the `@DirectPollMethod(event="message")` annotation, and that the event name is *message* because we have specified that as the *event* value in the *DirectPollMethod* annotation. This is the annotation we need to add to a method used to handle polling provider requests.

Poll handlers receive as their only parameter a Java map with parameter names as keys and parameter values as values. In the example no parameters are passed to the request handler, but you can provide parameters using the provider `baseParams` config option.

Now, back to the url, `Ext.app.POLLING_URLS.message`. Where does it come from? It is part of the generated api file: `Ext.app` is the value we specified for `apiNamespace` in the servlet configuration

for the *demo* api, and **POLLING_URLS** is the object holding the urls of all polling events in that api. Lastly, *message* is the event name.

It is possible to handle polling provider events in Javascript via the provider's **data** listener as well. The following code is from one of our internal tests, and shows how to do that. Besides, it shows how to pass parameters to the request, using the **baseParams** config option.

```
var pollingProvider = Ext.Direct.addProvider({
  type: 'polling',
  interval: 1000,
  url: test.POLLING_URLS.test_pollWithBaseParams,
  baseParams : {
    arg1 : 'value',
  },
  listeners: {
    data: function(provider, event) {
      Ext.log( 'test_pollWithBaseParams' );
      timesCalled++;
      if (timesCalled === 2) {
        pollingProvider.disconnect();
        Djn.Test.check('test_pollWithBaseParams',
          event.data !== undefined && event.data === 'arg1=value',
          "Expected to receive 'arg1=value' as event.data");
      }
    }
  }
});

pollingProvider.connect();
```

Just for completeness, let us show the Java code:

```
@DirectPollMethod
public String test_pollWithBaseParams( Map<String,String> parameters ) {
  assert parameters != null;

  // ...

  String result = "";
  for( String key : parameters.keySet() ) {
    String value = parameters.get(key);
    result += key + "=" + value;
  }

  return result;
}
```

Please, note that since we haven't specified a value for the **event** parameter in the **DirectPollMethod** annotation, the name of the event is the method name.

8. DirectStore with DirectJNgin

The `DirectStore` is an *Ext* store that uses *Ext Direct* to load data. We provide an example of how to use *DirectJNgin* for that in *djn_test* web app, in the *demo/DirectStoreDemo.html* page.

The Javascript code needed to create the store is as follows:

```
var experienceStore = new Ext.data.DirectStore( {
    paramsAsHash: false,
    root: '',
    directFn: DirectStoreDemo.loadExperienceData,
    idProperty: 'description',
    fields: [
        {name: 'startDate' },
        {name: 'endDate' },
        {name: 'description' },
    ],
    listeners: {
        load: function(s, records){
            Ext.MessageBox.alert( "Information", "Loaded " +
                records.length + " records");
        }
    },
});

experienceStore.load();
```

Setting up a `DirectStore` is very similar to setting up any other store: the main difference is the fact that you have to specify the server side method you want to be called to load the store data using the `directFn` config parameter.

The server side code is as follows:

```
private static class Experience {
    public String startDate;
    public String endDate;
    public String description;

    private Experience( String startDate, String endDate,
                        String description ) {
        this.startDate = startDate;
        this.endDate = endDate;
        this.description = description;
    }
}

@DirectMethod
public List<Experience> loadExperienceData() {
    List<Experience> items = new ArrayList<Experience>();
    Collections.addAll( items,
        new Experience( "2009/05/10", "...",
            "Implementation of <b>DirectJNgin</b> for ExtJs")
        // ...
    );

    return items;
}
```

First of all, we define a very simple Java class, **Experience**, that has all data for items in the store, (**startDate**, **endDate** and **description**, as defined in the **fields** config option). The server method just returns a list of **Experience** objects.

There is really nothing remarkable about the server method, which as you probably expected just needs to have the **DirectMethod** annotation, and must return a list of objects, in this case of type **Experience**. However, you *must* set the **DirectStore paramsAsHash** configuration parameter to false, for this to work.

Passing parameters to DirectStore's directFn

In real life, you will have to pass some parameter to your **directFn** method. You can do this in several ways:

- Use the **DirectStore baseParams** configuration property: lookup **argFromBaseParams** in the example code below to learn how to do this.
- Use the **load** method's **params** argument: lookup **argPassedInLoadCall** in the example code to check how to do this.
- Add the parameter to the **options** object received by the **DirectStore beforeload** method: lookup **argPassedInBeforeLoadEvent** to learn how to do this.

Here is the javascript source code for the *DirectJNgin* test method demonstrating how to use this:

```
test_loadWithArguments : function() {
  var myStore = new Ext.data.DirectStore( {
    paramsAsHash: false,
    root: 'items',
    paramOrder: [ 'argFromBaseParams', 'argPassedInLoadCall',
                  'argPassedInBeforeLoadEvent' ],
    totalProperty : 'rowCount',
    directFn: DirectStoreTest.test_loadWithArguments,

    idProperty: 'id',
    fields: [
      {name: 'id' },
      {name: 'name' }
    ],
    listeners: {
      beforeload : function(store, options) {
        options.params.argPassedInBeforeLoadEvent = false
      },
      load: function(s, records){
        Djn.Test.check( "test_loadWithArguments", records.length === 2,
          "If there is an error, this will never be called: a " +
          "timeout should happen if there is some error!" );
      }
    },
    baseParams: {argFromBaseParams: 'aValue' }
  });

  myStore.load({
    params: {
      argPassedInLoadCall: 34
    }
  });
}
```



```
    }  
  });  
}
```

Remember that you must set **paramsAsHash** to false. Besides, you *must* specify the order in which the parameters will be passed using the *DirectStore* **paramOrder** configuration parameter.

Here is the server side test method:

```
@DirectMethod  
public DirectStoreResult dj_n_test_loadWithArguments(  
    String argFromBaseParams, int argPassedInLoadCall,  
    boolean argPassedInBeforeLoadEvent )  
{  
    if( !argFromBaseParams.equals("aValue") || argPassedInLoadCall != 34 ||  
        argPassedInBeforeLoadEvent ) {  
        throw new DirectTestFailedException(  
            "Did no receive expected values");  
    }  
  
    Output[] result = new Output[2];  
    result[0] = new Output();  
    result[0].id = 99;  
    result[0].name = "name1";  
  
    result[1] = new Output();  
    result[1].id = 100;  
    result[1].name = "name2";  
  
    DirectStoreResult r = new DirectStoreResult( result, 347);  
    return r;  
}
```

Passing unknown parameters to DirectStore's directFn

If you need to pass unknown parameters (i.e., parameters whose name you don't know beforehand), you will need to set the **paramsAsHash** to true, and then write a direct JSON handling method: we will study them later.

9. Google AppEngine support

DirectJNgin usually creates/updates the javascript files containing the API code when the web application is started.

This is done on purpose, because it makes it easier for the web server to perform optimizations such as telling the browser that the .js file has not changed, avoiding the need to transfer its content. For very large applications, this can make a difference.

Unfortunately, Google AppEngine does not allow writing files to disk (see <http://www.mail-archive.com/google-appengine@googlegroups.com/msg14572.html>). In order to solve this problem, we are allowing an alternative way to reference the API source code. If you were referencing a *test/MyApi.js* API file as follows,

```
<script type="text/javascript"
  src="test/DjnTestApi - debug.js">
</script>
```

now you should reference it like this:

```
<script type="text/javascript"
  src="/XXX/djn/directprovider/src=test/DjnTestApi - debug.js">
</script>
```

Here, *XXX* is the context name, *djn/directprovider* is the value you provided in the *DirectJNgin providersUrl* servlet parameter, and */src=* is a standard prefix you need to add before the api file name.

This way, *DirectJNgin* will generate the source code on the fly, making the API files unnecessary. In fact, the source will be generated and minified at start time, and placed in a cache.

Now, if you don't want that *DirectJNgin* creates/updates the source files, you will need to set the *createSourceFiles* servlet parameter to false (it is true by default), so that *AppEngine* does not complain.

Last, but not least, you will need to disable multithreaded support for batched requests, because *AppEngine* will not like your app if it creates multiple threads. To do this, set the *batchRequestsMultithreadingEnabled* servlet parameter to false.

Once all of this is done, you should be able to run *Google AppEngine* without a problem.

10. Servlet configuration

The following table includes the list of global configuration parameters, which can be set via servlet initialization parameters.

Global parameter	Description
<code>providersUrl</code>	Required. This URL will be used by <i>Ext Direct</i> to communicate with <i>DirectJNginx</i> : it <i>must</i> be the same as the servlet's <code>url-pattern</code> , minus the ending <code>/*</code> .
<code>debug</code>	If set to true, default generated api files will be readable and include comments (they will not be minified), logs for JSON data will be pretty printed, etc. Set to false in your production releases. Not required. Default value: false.
<code>apis</code>	Required. A comma-separated value of Direct provider api names: you can configure each api using the api configuration parameters in the next table. Api names must not be duplicated.
<code>gsonBuilderConfiguratorClass</code>	It is possible to provide custom Gson configuration and type handling by specifying a custom class for this parameter, as explained in the <i>State management and session/application scope support</i> chapter. Not required. Default value: <code>com.softwarementors.extjs.djn.gson.DefaultGsonBuilderConfigurator</code>
<code>registryConfiguratorClass</code>	It is possible to register actions and methods programmatically, for all kinds of requests. You do this by specifying a custom class for this parameter, as explained in the <i>Customizing DirectJNginx programmatically</i> section. Not required Default value: none.
<code>minify</code>	If set to false, <i>DirectJNginx</i> will not generate a minified version of api files. Not required. Default value: true
<code>batchRequestsMultithreadingEnabled</code>	Please, take a look at the <i>Optimizing batch requests handling using multiple threads</i> section for an

	<p>explanation.</p> <p>Not required.</p> <p>Default value: true.</p>
batchRequestsMinThreadsPoolSize	<p>Please, take a look at the <i>Optimizing batch requests handling using multiple threads</i> section for an explanation.</p> <p>Not required.</p> <p>Default value: 16.</p>
batchRequestsMaxThreadsPoolSize	<p>Please, take a look at the <i>Optimizing batch requests handling using multiple threads</i> section for an explanation.</p> <p>Not required.</p> <p>Default value: 80.</p>
batchRequestsMaxThreadKeepAliveSeconds	<p>Please, take a look at the <i>Optimizing batch requests handling using multiple threads</i> section for an explanation.</p> <p>Not required.</p> <p>Default value: 60.</p>
batchRequestsMaxThreadsPerRequest	<p>Please, take a look at the <i>Optimizing batch requests handling using multiple threads</i> section for an explanation.</p> <p>Not required.</p> <p>Default value: 8.</p>
dispatcherClass	<p>The class implementing the <i>Dispatcher</i> that will handle action methods: useful for extending <i>DirectJNgin</i>.</p> <p>Not required.</p>
jsonRequestProcessorThreadClass	<p>The class implementing the <i>JsonRequestProcessorThread</i> that will handle batched JSON requests: useful for extending <i>DirectJNgin</i>.</p> <p>Not required</p>
contextPath	<p>Forces the web app context path, instead of calculating it via Javascript: this is usually not needed, but you MUST specify if the web app context path is the default context ('/' or ''), because the Javascript code will not be able to handle this special case.</p> <p>Not required.</p>
createSourceFiles	<p>When set to false, DirectJNgin will not create source files. This is needed for Google's AppEngine, for example, which does not allow file creation at</p>

	run-time. Not required. Default value: true
--	---

Every Direct provider api we define can be configured by using the parameters in the following table. Please, note that **xxx** must be the api name, as declared in the global **api s** configuration parameter.

Api parameter	Description
xxx. api Fi l e	<p>The file the api will be written to.</p> <p>It is possible to have several apis written to the same file: just provide the same file name.</p> <p>Not required.</p> <p>Default value: <i>xxx-api.js</i>, <i>xxx</i> being the api name.</p>
xxx. cl asses	<p>A comma separated list of classes to scan for direct methods.</p> <p>Not required.</p>
xxx. act i onsNamespace	<p>The namespace for the actions. For example, if we set this to <i>MyNamespace</i>, then to call <i>myMethod</i> in the <i>MyAction</i> class you will have to write the following code:</p> <pre>MyNamespace. MyActi on. myMethod(. . .);</pre> <p>If no value is specified, then the actions will not belong to a namespace, and you'll write the following code to invoke <i>myMethod</i>:</p> <pre>MyActi on. myMethod(. . .);</pre> <p>Not required.</p>
xxx. api Namespace	<p>The namespace for the api. For example, if we set this to <i>Ext.app</i>, then you will have to write the following code to register the api:</p> <pre>Ext. Di rect. addProvi der(Ext. app. REMOTING_API);</pre> <p>Not required.</p> <p>Default value: if <i>actionsNamespace</i> is set, its value will be used. Else, the default value will be <i>Djn.xxx</i>, <i>xxx</i> being the api name.</p>

11. State management and session/application scope support

While *DirectJNgin* is just a communication protocol, many users have requested the ability to access the web session, the servlet context, the servlet configuration, etc. from within their action methods.

To this end, we have implemented two new classes, **WebContext** and **WebContextManager**. The following code shows how to use them in order to store the number of times a method has been called in a session:

```
@DirectMethod
public WebContextInfo test_webContext() {
    WebContext context = WebContextManager.get();
    HttpSession session = context.getSession();
    ServletContext application = context.getServletContext();

    // Keep a counter of how many times we have called this method
    // in this session
    Integer callsInSession=(Integer) session.getAttribute("callsInSession");
    if( callsInSession == null ) {
        callsInSession = new Integer(0);
    }
    callsInSession = new Integer(callsInSession.intValue() + 1);
    session.setAttribute("callsInSession", callsInSession);

    // ...
}
```

As you can see, you just get a **WebContext** by calling the **WebContextManager.get()**, and then you call its **getSession** method -or **getServletContext**, **getRequest**, **getResponse**, **getServletConfig**, depending on your needs. That's all!

Stateful actions

By default, all action objects are stateless, meaning that they are recreated (at least, conceptually) every time a new request is handled at the server. However, sometimes it would be nice to have actions whose state is kept between requests. *DirectJNgin* has added support both for session scoped and application scoped action objects, using the **ActionScope** annotation, as follows:

```
@ActionScope(scope=Scope.SESSION)
public class SessionStatefulActionTest {
    private int count = 0;

    @DirectMethod
    public synchronized int test_getSessionCallCount() {
        this.count++;
        return this.count;
    }

    // ...
}
```

The supported scopes are **SESSION**, **APPLICATION** and **STATELESS**. If not explicitly set, the default action scope is **STATELESS**.

Generating multiple action instances for a Java class

Now that we have stateful actions, it makes sense to be able to create different instances so that each one has its own differentiated state.

To this end, we have modified the **DirectMethod** annotation, so that we can create multiple instances by providing multiple names, as follows:

```
@DirectAction( action={"action1", "action2"})
public class ClassWithMultipleActionsTest {
    // ...
}
```

From now on, you will be able to access **action1** and **action2** in your Javascript code as usual.

12. Customizing data conversion and Gson configuration

We are using Gson to handle data conversion from JSON to Java data and back. Gson is very powerful, and its default configuration is quite acceptable, but it might happen that you need to customize it.

What are the Gson configuration options? Just take a look at the *Gson User's Guide*, and then the documentation for its `GsonBuilder` class, they are all explained there.

Among the configuration options, there is the possibility to control how to serialize/deserialize certain Java types, such as a hypothetical `DateTime` class provided by a third party that Gson does not even know about.

To allow you to handle these issues, we have provided support for you to configure the `GsonBuilder` *DirectJNgin* uses to parse JSON.

Changing Gson's configuration

To take control of Gson configuration you have to create a class that implements the `GsonBuilderConfigurator` interface. As an example, here is the implementation of the class that defines the *default* configuration for *DirectJNgin*:

```
public class DefaultGsonBuilderConfigurator
    implements GsonBuilderConfigurator
{
    @Override
    public void configure(GsonBuilder builder,
        GlobalConfiguration configuration) {
        assert builder != null;
        assert configuration != null;

        if( configuration.getDebug() ) {
            builder.setPrettyPrinting();
        }
        builder.serializeNulls();
        builder.disableHtmlEscaping();
    }
}
```

The only method you need to override is `configure`, which receives our `GsonBuilder` as its first parameter, and the global *DirectJNgin* configuration as the second one. We think the code is pretty much self-explanatory.

Now you have to tell *DirectJNgin* that you want to use your own custom configurator. To do that, use the `gsonBuilderConfiguratorClass` servlet parameter, which must be the full name of the configurator class:

```
<init-param>
  <param-name>gsonBuilderConfiguratorClass</param-name>
  <param-value>
com.softwarementors.extjs.djn.test.config.GsonBuilderConfiguratorForTesting
  </param-value>
</init-param>
```


If you don't specify a value for `GsonBuilderConfiguratorClass`, the default configurator will be used.

I want the *default* Gson configuration back!

Just create your own configurator class as follows:

```
public class MyGsonBuilderConfigurator
    implements GsonBuilderConfigurator {
    @Override
    public void configure(GsonBuilder builder,
        GlobalConfiguration configuration)
    {
        // Do nothing!
    }
}
```

Adding your own serializers/deserializers

Once you define your own Gson configurator class, you will be able to configure how JSON data is transformed from JSON to a Java type and back.

As an example, we have implemented support to convert a Javascript object representing a date (with no time data) to a Java `Date`. The Javascript object can be defined as follows:

```
var aDate = {year: 2005, month: 3, day: 20};
MyAction.callMethodWithDate( aDate );
```

What we want is this kind of Javascript object to be converted to a plain Java date, so that we can implement the Java method like this:

```
@DirectMethod
public void callMethodWithDate(Date date) // ...
```

And, of course, we want to be able to handle dates returned by a Java method in Javascript. To do these two things, we need to define Gson serializers and deserializers. Here is the code:

```
public class GsonBuilderConfiguratorForTesting
    extends DefaultGsonBuilderConfigurator
{
    @Override
    public void configure(GsonBuilder builder,
        GlobalConfiguration configuration)
    {
        super.configure(builder, configuration);
        addCustomSerializationSupport(builder);
    }
}
```

```

private void addCustomSerializationSupport(GsonBuilder builder) {
    // Convert our own custom Javascript "date" to a Java Date
    builder.registerTypeAdapter( Date.class, new JsonSerializer<Date>() {
        public JsonElement serialize( Date src, Type typeOfSrc,
                                     JsonSerializerContext context) {

            assert src != null;
            assert context != null;
            assert typeOfSrc != null;

            JsonObject result = new JsonObject();
            setIntValue( result, "year", src.getYear() + 1900);
            setIntValue( result, "month", src.getMonth() + 1);
            setIntValue( result, "day", src.getDate());

            return result;
        }
    });

    // Convert a Java Date to our own custom Javascript "date"
    builder.registerTypeAdapter( Date.class, new JsonDeserializer<Date>() {
        @Override
        public Date deserialize( JsonElement json, Type typeOfT,
                                JsonDeserializationContext context)
            throws JsonParseException
        {
            assert json != null;
            assert context != null;
            assert typeOfT != null;

            if( !json.isJsonObject() ) {
                throw new JsonParseException( "A Date must be a JSON object");
            }

            JsonObject jsonObject = json.getAsJsonObject();
            int year = getIntValue( jsonObject, "year" ) - 1900;
            int month = getIntValue( jsonObject, "month" ) - 1;
            int day = getIntValue( jsonObject, "day" );

            Date result = new Date( year, month, day);
            return result;
        }
    });
}

```

The code relies in a pair of utility functions that are really not part of the serializer/deserializer, which we include here for completeness:

```

private static void setIntValue( JsonObject parent, String elementName,
                                int value ) {
    parent.add( elementName, new JsonPrimitive( new Integer(value)));
}

private static int getIntValue( JsonObject parent, String elementName )
{
    assert parent != null;
    assert !StringUtils.isEmpty(elementName);
}

```

```
JsonElement element = parent.get( elementName );
if( !element.isJsonPrimitive() ) {
    throw new JsonParseException( "Element + ' " + elementName +
        "' must be a valid integer");
}
JsonPrimitive primitiveElement = (JsonPrimitive)element;
if( !primitiveElement.isNumber() ) {
    throw new JsonParseException( "Element + ' " + elementName +
        "' must be a valid integer");
}
return primitiveElement.getAsInt();
}
}
```

We hope the code is not too difficult to understand: just take a look at *Gson User's Guide* for details -it is very well written.

Closing thoughts

We have provided this as an example of how to handle non-trivial types, such as Java's **Date**. We decided against providing default serializers/deserializers for classes such as **Date** or **Calendar** because we didn't want to impose a Javascript format for these types. In our example we defined our own custom Javascript "date" like this:

```
var aDate = {year: 2005, month: 3, day: 20};
```

But, why not use a more compact alternative? Something like this, for example:

```
var aDate = [2005, 3, 20];
```

We thought that the decision should be yours: we hope this example will make it very easy for you to implement the solution that better fits you.

13. Handling JSON data directly

We have made every effort to handle serialization from JSON to Java for you, so that you can write methods that receive good old Java data types.

However, there can be cases when you might need to access the JSON data directly for maximum flexibility, and *DirectJNgin* allows you to do that too.

Here is some code that shows how to write such a method:

```
@DirectMethod
public boolean test_handleJsonDataMethod( JSONArray data ) {
    assert data != null;

    // Write your own custom code here...
    if( data.size() != 1) {
        throw new DirectTestFailedException(
            "We expected a json array with just one element");
    }

    JsonElement element = data.get(0);
    if( !element.isJsonPrimitive()) {
        throw new DirectTestFailedException(
            "We expected the first json item to be a json primitive");
    }

    JsonPrimitive primitive = (JsonPrimitive)element;
    if( !primitive.isBoolean()) {
        throw new DirectTestFailedException(
            "We expected a primitive json boolean element");
    }
    if( primitive.getAsBoolean()) {
        throw new DirectTestFailedException( "We expected a false value");
    }

    return primitive.getAsBoolean();
}
```

As you can see in the example code, the method must receive a **JSONArray**, because the data sent by *Ext Direct* is encoded in a JSON array. For information on how to handle a **JSONArray**, take a look at Gson's documentation, please.

Of course, this makes sense only for JSON requests, and this means that this is only supported for standard methods (i.e., those annotated with **@DirectMethod**).

14. Adding actions and methods programmatically

DirectJNgine registers actions and methods by scanning those classes you specify via the servlet configuration. However, there will be cases in which you will want to register your own methods programmatically.

To do this, you must create a class that implements the **RegistryConfigurator** interface, and then set the **registryConfiguratorClass** servlet parameter to the full class name.

As always, we have written automated tests to make sure things really work, and we will use our own tests to illustrate how to use the feature. Here is the code implementing **RegistryConfigurator** for our test implementation:

```
public class RegistryConfiguratorForTesting implements
    ServletRegistryConfigurator
{
    private Method getMethod( Class<?> cls, String name,
        Class<?> parameterTypes)
    {
        assert cls != null;
        assert !StringUtils.isEmpty(name);

        try {
            Method m = cls.getMethod(name, parameterTypes);
            return m;
        }
        catch (SecurityException e) {
            // Do not do this in production quality code!
            throw new RuntimeException(e);
        }
        catch (NoSuchMethodException e) {
            // Do not do this in production quality code!
            throw new RuntimeException(e);
        }
    }
}

public void configure(Registry registry, ServletConfig config) {
    assert registry != null;
    assert config != null;

    // Create a new api programmatically
    String apiFile =
        config.getServletContext().getRealPath("test/ProgrammaticApi.js");
    RegisteredApi api = registry.addApi( "programmaticApi",
        "test/ProgrammaticApi.js", apiFile,
        "Djn.programmaticNamespace", "Djn.programmaticNamespace" );

    // Register a new action with a method
    RegisteredAction action = api.addAction(
        CustomRegistryConfiguratorHandlingTest.class,
        "MyCustomRegistryConfiguratorHandlingTest");
    Method m = getMethod( CustomRegistryConfiguratorHandlingTest.class,
        "test_programmaticMethod", String.class);
    action.addStandardMethod( "myProgrammaticMethod", m, false);

    // Register a poll method
    Method pm = getMethod( CustomRegistryConfiguratorHandlingTest.class,
```

```

        "test_programmaticPollMethod", Map.class);
    action.addPollMethod( "myProgrammaticPollMethod", pm);
}
}

```

As you can see, the programmatic API uses extensively several *DirectJNgin* classes: **RegisteredApi**, **RegisteredAction**, **RegisteredMethod** and **RegisteredPollMethod**.

Now, let's take a look at how to use this functionality in Javascript. Here is the test code for this feature:

```

Dj n. CustomRegistryConfiguratorHandlingTest = {
    testClassName : 'CustomRegistryConfiguratorHandlingTest',

    test_programmaticMethod : function() {
        Dj n. programmaticNamespace. MyCustomRegistryConfiguratorHandlingTest.
            myProgrammaticMethod( 'programmatic', function(result, response) {
                Dj n. Test. checkSuccessfulResponse("test_programmaticMethod",
                    response, result === 'programmatic');
            });
    },

    test_programmaticPollMethod : function() {
        var pollingProvider = Ext.Direct.addProvider({
            type: 'polling',
            interval: 100,
            baseParams : {
                myParameter : 'myValue'
            },
            url:
                Dj n. programmaticNamespace. POLLING_URLS. myProgrammaticPollMethod,
            listeners: {
                data: function(provider, event) {
                    Ext.log( 'test_programmaticPollMethod' );
                    pollingProvider.disconnect();
                    Dj n. Test. check('test_programmaticPollMethod',
                        event.data === 'ok',
                        "Expected to receive 'ok' as event.data");
                }
            }
        });
        pollingProvider.connect();
    }
}

```

In order to understand what's going on, just find where and how the following code/strings are used both in Java and Javascript code:

- **Dj n. programmaticNamespace**: the namespace for the javascript provider and actions.
- **MyCustomRegistryConfiguratorHandlingTest**: the javascript action name. The corresponding java class is **CustomRegistryConfiguratorHandlingTest**.
- **myProgrammaticMethod**: a javascript method name. The corresponding java method is **CustomRegistryConfiguratorHandlingTest. test_programmaticMethod**.

- **myProgrammaticPollMethod**: a javascript poll method name. The corresponding java method is **CustomRegistryConfiguratorHandlingTest.test_programmaticPollMethod**.
- **test/ProgrammaticApi.js**: the generated api file, which you must add to your HTML with a **script** tag.

Just spend a handful of minutes to understand how the java and javascript code are related, and you'll be able to write your own code to perform programmatic creation of action methods.

For completeness, here is the code for the action and poll methods:

```
public class CustomRegistryConfiguratorHandlingTest {
    public String test_programmaticMethod( String value ) {
        if( !value.equals( "programmatic" ) ) {
            throw new DirectTestFailedException(
                "We expected to receive 'programmatic' as value");
        }

        return value;
    }

    public String test_programmaticPollMethod(
        Map<String, String> parameters )
    {
        assert parameters != null;

        if( parameters.size() != 1 || !parameters.containsKey("myParameter")
            || !parameters.get("myParameter").equals("myValue") )
        {
            throw new DirectTestFailedException(
                "We expected to receive 'myParameter' with a value of 'myValue'");
        }
        return "ok";
    }
}
```

Note that we are not annotating these methods because we are not going to process them using the default class scanning functionality built into *DirectJNgine* –well, that's the whole point!

Besides, you need not specify the **CustomRegistryConfiguratorHandlingTest** class in the servlet configuration as one of the Java classes to scan, for the same reason.

As an aside, let me tell you that *DirectJNgine* is chock full of assertions (as I write this, I just found that there are more than 400 assertions spread in the code!). Make sure that you enable them while in development mode, especially when you are writing code that customizes *DirectJNgine* itself. They will be invaluable for debugging.

15. Checking client-side parameters

Due to the ways of Javascript and how *Ext Direct* works, there are cases in which strange things can happen when a server methods is called. *DirectJNgine* provides support for checking some of those cases, in order to avoid potentially dangerous situations.

In order to better understand some of the problem, let's take a look at the following Java code:

```
class MyAction {
    @DirectMethod
    double sum( double d1, double d2) {
        return d1 + d2;
    }
}
```

Now, let's call our `sum` method with the following Javascript code:

```
MyAction.sum(3, undefined, 5);
```

Due to the way *Ext Direct* serializes json, it will ignore the *undefined* argument, and the request will look as if the Javascript method had been called as follows:

```
MyAction.sum(3, 5);
```

I think this can be dangerous, because the client and the server are seeing very different things, with the server not being able to know that something potentially problematic is going on.

The problem gets worse if you pass an array with an undefined value, as it will be ignored, and the same will happen for an object inside an object inside...that has an array with one of its values set to *undefined*. And this last scenario can be *really* hard to debug.

There are more problematic situations. For example, take a look at the following Javascript calls:

```
MyAction.sum( 3, 5, 7522 );
MyAction.sum( 3 );
```

In both cases the server will be called with the wrong number of arguments. While the server can handle this, I think it would be nice if *Ext Direct* checked that the number of arguments is right before sending a request.

All in all, we think it would be nice if these things were checked at the client. In fact, we have implemented support to check for these issues while debugging, as follows:

```
var remoti ngProvider = Ext. Di rect. addProvi der( Dj n. test. REMOTING_API );
Dj n. RemoteCal l Support. addCal l Val i dat i on(remoti ngProvi der);
Dj n. RemoteCal l Support. val i dateCal ls = true;
```

This functionality is provided in the *djn-remote-call-support.js* file, which is located in the *deliverables* directory in our distribution.

It is *very important* that you use this for debug only, for several reasons. The first one is that the algorithm traverses the whole object graph for every function argument, something that might be expensive, and you will not want to incur this overhead once you have your application fully debugged and tested.

The second reason is that the algorithm does not check for cycles: therefore, if you pass object *a*, which references object *b*, and object *b* references object *a*, then you will have infinite recursion. The Javascript interpreter will detect this, and raise an exception. Unfortunately, this is a limitation in our algorithm. We feel that we can live with this, because we prefer the additional debugging support these checks give. Besides, you can deactivate temporary parameter checking as follows:

```
Dj n. RemoteCallSupport.validateCalls = false;
```

We have to confess that we thought twice before adding this feature, but in the end we arrived to the compromise of using it for **debugging purposes only**. Just use it judiciously, or simply avoid it if you don't like it.

16. DirectJNgin Optimization

Optimizing api files generation and usage

Minimizing network traffic is one of the most important optimizations we can perform for a web application.

We have worked hard on minimizing both the number of requests, as well as their size when it comes to api file. Here is the list of optimizations:

- We only regenerate an api file when its contents changes: that way the web server does not send exactly the same content just because we have rewritten a file and its date and time has changed. The server will communicate the client that the file has not changed, saving bandwidth.

Restarting the application server *will not* force the api files to be rewritten -unless their content has changed.

- You can consolidate several apis in just one file: as you know, you specify the api file name for an **xxx** api via the servlet **xxx. api File** parameter. If you want two different apis to be written to the same file, use the same file name in the **api File** parameter.

This minimizes the *number of requests* the client makes to the server to retrieve a web page.

- We generate minified versions of api files to save *bandwidth*.

In fact, *DirectJNgin* generates three versions of a file. If you specified *abc.js* in the **api File**, you will get the following files:

- *abc-debug.js*: the debug version of the api file.

This is very readable, and includes comments for every method, including the Java types for the method parameters and the returned value.

- *abc-min.js*: a minified version of the api file. It does away with unnecessary whitespace as well as comments.

In our test files we have obtained a file whose size is *less* than 50% of the debug file size.

- *abc.js*: if you have the servlet **debug** parameter to true, this file will contain debug code, else it will contain minified code.

Why this file? Because this way you can change what file your application really uses without having to modify your HTML files code so that they link to *abc-debug.js* instead of *abc-min.js*.

You can disable generation of minified files using the global **minify** servlet initialization parameter.

By the way, it is highly unlikely that minification fails: we use the *YUI Compressor*, a very well tested minifier. However, if the YUI Compressor raises some exception or reports some error, we make sure that the minified file will contain at least standard code, so that your application does not break because there is no “*-min.js*” file.

Optimizing batch requests handling using multiple threads

When several requests reach the web server, it invokes the *DirectJNgine* servlet in different threads, giving us multitasking for free.

However, *Ext Direct* has a feature that allows independent logical requests to be batched, so they are all sent grouped in a single physical request. This is a really nice optimization, because it minimizes the number of data exchanges going on between the client and the server.

The web app knows nothing about this, so it just makes one call to our servlet, instead of distributing the logical calls among several threads, as it might have done had it received the requests separately. Since one of our goals is to provide excellent performance, we have decided to provide support for this feature in *DirectJNgine*.

Multithreaded handling of batched requests is enabled by default: however, if you need to disable it for some reason, you can set the `batchRequestsMultiThreadedEnabled` servlet initialization parameter to false.

There several additional servlet parameters you can use to customize thread usage:

- `batchRequestsMinThreadsPoolSize`: equivalent to Java's `ThreadPoolExecutor.getCorePoolSize`.
- `batchRequestsMaxThreadsPoolSize`: equivalent to Java's `ThreadPoolExecutor.getMaxPoolSize`.
- `batchRequestsMaxThreadKeepAliveSeconds`: equivalent to Java's `ThreadPoolExecutor.getKeepAliveTime`.
- `batchRequestsMaxThreadsPerRequest`: explained later.

In order to understand these parameters, take a look at the Javadoc documentation for *ThreadPoolExecutor*: it is quite good. We create our thread pool instance passing the parameters as follows:

```
new ThreadPoolExecutor( batchRequestsMinThreadsPoolSize,
                        batchRequestsMaxThreadsPoolSize,
                        batchRequestsThreadKeepAliveSeconds,
                        TimeUnit.SECONDS,
                        new LinkedBlockingQueue<Runnable>() );
```

The `batchRequestsMaxThreadsPerRequest` is not passed to the thread pool handler. This parameter limits the number of threads that will be devoted to handle the individual requests for a single batched request. We added this limit so that no single client is able to end up consuming all threads in the pool.

Customizing thread usage is not easy, because this kind of optimization is very context dependent. That said, I wholeheartedly recommend that you take a look at *Java Concurrency in Practice*, by Brian Goetz, especially the sections on *thread pool sizing* and *configuration of ThreadPoolExecutor*.

On the other hand, we think that the default values we provide will be quite adequate for most users.

17. Diagnostics and logging

At times, debugging Javascript↔JSON↔Java interactions can be *really* daunting. Configuration issues are easier to deal with, but it is always nice to have as much help as possible in that area too.

While programming *DirectJNgin* we have paid *lots* of attention to getting accurate diagnostics when things go awry. In fact, if you take a look at the source code, you'll see lots of things that could have been solved with *much* less code: we have been writing lots of extra code to be very specific about what the cause of an error is -that's why we have a whole hierarchy of exceptions.

DirectJNgin uses *log4j* for logging. All *DirectJNgin* classes live under the `com. softwarementors. extj s. dj n` package, so you can adjust the log level adding a *logger* to your *log4j.properties* configuration, as follows:

```
log4j.logger.com. softwarementors. extj s. dj n=INFO
```

The traces at the **INFO** level are completely adequate for production, and we recommend you use that level, unless you are diagnosing an application. In any case, do not set the logging level to something less than **WARN**.

We recommend that you set the trace level to **ALL** at least once or twice to become familiar with *DirectJNgin* logs: running the automated tests in our *djn_test* WAR might be interesting, because those tests provoke errors and exercise lots of features, and you will be exposed to all kinds of logging info.

If you suspect that *DirectJNgin* is not working correctly, or just to learn what's going on, you might find it useful to look at the *request* and *response contents*: to take a look at these, set the logging level to **DEBUG**.

Measuring request execution time

If you want to get execution time data, you can enable a especial timer logger, as follows:

```
log4j.logger.com. softwarementors. extj s. dj n. Ti mer=ALL
```

Here you will find the time it takes to process every servlet call, the time per individual request (when you receive a bunch of requests in a batch), the time it takes to invoke your Java method (so that you can know how much time is consumed by *DirectJNgin*, and how much by your own code), etc.

Understanding which logs go together

Given that a web app can receive several requests concurrently, you will probably find their logs intertwined, making it *very* difficult to know what log message belongs to which request. To help with this we provide a unique *request id* per request, setting it as the *log4j NDC* value for every log message. This id will look like "*rid: xxx*", *xxx* being the id.

You can control whether and how this request id is written to logs using the `'%x'` parameter in your appender layouts. For example, in the *log4j.properties* in our *djn_test* application, we have our console layout defined as follows:

```
log4j.appender.Console.layout.ConversionPattern=
%-5p: %c - "%m" (%x)%n
```

18. How reliable is all of this?

At the moment of writing this document, we have more than 90 automated tests that check all kinds of situations: *undefined* values being passed to a remote method, form posts, form upload posts, batched JSON posts, complex object structures being returned from the server, etc.

We developed our testing infrastructure as a precondition to develop this library with guarantees: remote communication is a very tricky subject, and we felt that automated tests were a must. We have been writing unit tests for years, and *test driven development* works very well for us. Therefore, we plan to keep the test list to keep growing as time passes.

If you want to run our battery test, just make sure you have installed the *djn_demo.war* web app, as explained before. Once it is up and running, navigate to the *test/DjnTests.html* page, and all automated tests will be run...automatically.

To run manual tests, navigate to the *test/DjnManualTests.html* page, and follow the instructions.

Finally, it will make me feel better if we tell you we run our first battery test against Firefox (3.0.10 at the moment): that's just so you can use it to run our tests if you find that something goes awry with *whizzbang-explorer 0.3*, or something just looks ugly in it.

Why “manual tests”?

We have been developing application using *Test Driven Development* for almost a decade now, writing several thousand unitary tests during this time. To *TDD* advocates, manual tests are “evil”. Therefore, why do we have several manual test?

Well, it happens that *you can't set a form INPUT field of type FILE programmatically*, due to security concerns. Therefore, we have developed several manual tests, but **only** to check file uploads.

19. Licensing

DirectJNgin is open source. Please, check the *readme.txt* file in your distribution for details about both *DirectJNgin* and *ExtJs* licensing.