DirectJNgine
User's Guide

Version 1.0 RC1
for ExtJs 3.0.0

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 DirectJNgine

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

DirectJNgine in 30 seconds

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

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 *DirectJNgine* provides. Once you do that, you will get automatic remoting support: no need for boring, cumbersome and error-prone glue code at the server side.

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 in 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.

Now, what about those using JDKs prior to 1.5, that do not support annotations? No problem. Remove the `@DirectMethod` annotation from our example, and then rename the `sayHello` method to `djn_sayHello`: the `djn_` prefix will be enough to get the magic!

2. Features

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

- Support for JSON requests.
- Support for batched JSON requests.
- Support for Simple Form Posts (no files to upload).
- Support for Upload Form Posts.
- Support for PollingProvider requests.
- Multithreaded processing of batched requests.
- Annotations-based configuration for JDK 1.5, 1.6, etc.
- 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.
- Support for generation of multiple API Files.
- Api files minification and comment removal.
- Api consolidation: consolidate several apis into just one file to minimize network traffic.
- 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.

3. Installing DirectJNgine

To install the library, decompress the appropriate *directjngine.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. DirectJNgine by example

> **More about 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.).

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 with Java in the server side, using *DirectJNgine*. In fact, we will use them in order to show how *DirectJNgine* works.

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

- *direct.php*: subsitute the *php/api.php* string with *../../../demo/Api.js*.
- *direct-form.php*: subsitute *php/api.php* with *../../../demo/Api.js*.
- *direct-tree.php*: subsitute *php/api.php* with *../../../demo/Api.js*.
- *direct.js*: subsitute *'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 *DirectJNgine*.

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

Running ExtJs examples against DirectJNgine

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 DirectJNgine

Step 1: configure DirectJNgine servlet in web.xml

Open the *WebContent/web.xml* file included with your *DirectJNgine* distribution, and take a look at the following lines:

```
<!-- DirectJNgine servlet -->
<servlet>
  <servlet-name>DjnServlet</servlet-name>
  <servlet-class>
    com.softwarementors.extjs.djn.servlet.DirectJNgineServlet
  </servlet-class>

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

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

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

Here, we configure the *DirectJNgine servlet*. 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 *DirectJNgine*: 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, DirectJNgine will generate it on your behalf.

But, how does DirectJNgine 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.namespace</param-name>
  <param-value>Ext.app</param-value>
</init-param>
```

Our applications provides two different *apis*, one for test methods (called *test*), and another one for demo methods (called *demo*). You must provide the `apis` parameters in order to tell *DirectJNgine* the apis you want to define. Most of the time an "api" is little more than an independent Javascript file *DirectJNgine* generates on your behalf.

You have to specify the file the *demo* api will end up in, using the `demo.apiFile` 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.

In order to be a good *ExtJs* citizen, you will have to specify the namespace where all things in the *Api.js* file will live, using the `demo.namespace` parameters.

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

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. If the string we receive is convertible to a number, there is not much to worry about, but, what if we receive a null string, or something like *"hello world"*? If that's the case, the call to `Double.parseDouble` will throw a `NumberFormatException`. Well, *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.

By the way, what if you are stuck with a JDK that does not support annotations, such as JDK 1.4? No problem, just add "`djn_`" as a prefix to the method name, as follows:

```
public double djn_myMultiplyMethod( String num ) {
  // ...
```

Of course, the method name as seen by Javascript will still be `myMultiplyMethod`, the `djn_` prefix is just a hint for *DirecJNgine*, you don't want to carry it to the Javascript code.

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 DirectJNgine where to look for server methods

Now, how does DirectJNgine 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.namespace` 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 *DirectJNgine*, 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.

DirectJNgine 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'/*(java.lang.String) => java.lang.String */,
        len: 1,
        formHandler: false
      },
      {
        name: 'multiply'/*(java.lang.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 that creating a new Java method to be used by other Java code in your app.

5. Form posts and DirectJNgine

To learn how to handle forms, including file uploads, just go to the *demo/FormPostDemo.html* page in the *djn_test* web app.

First of all, you need to invoke the server method in your javascript code. If you look at *FormPostDemo.js*, you will see that the form *'Submit'* button has a handler, defined as follows:

```
handler: function(){
  FormPostDemo.handleSubmit(form.getForm().el, function(result, e){
    if( e.type === 'exception') {
      Ext.MessageBox.alert("Unexpected server error", e.message );
      return;
    }
    Ext.MessageBox.alert("Posted values", result.fieldNamesAndValues);
    textArea1.setValue( result.fileContents);
  });
}
```

In this case, we have to pass the form's `el` element as the first and only parameter to a server method annotated with `@DirectFormPostMethod`, which is implemented as follows:

```
public class FormPostDemo {
  public static class Result {
    public String fileContents = "";
    public String fieldNamesAndValues = "";
  }

  @DirectFormPostMethod
  public Result handleSubmit( Map<String, String> formParameters,
                  Map<String, FileItem> fileFields ) throws IOException
  {
    assert formParameters != null;
    assert fileFields != null;

    Result result = new Result();
    String fieldNamesAndValues = "";

    fieldNamesAndValues += "<p>Non file fields:</p>";
    for( String fieldName : formParameters.keySet()) {
      fieldNamesAndValues += "<b>" + fieldName + "</b>='" +
         formParameters.get(fieldName) + "'<br>";
    }

    fieldNamesAndValues += "<p></p><p>FILE fields:</p>";
    for( String fieldName : fileFields.keySet() ) {
      FileItem fileItem = fileFields.get(fieldName);
      result.fileContents = IOUtils.toString(
                              fileItem.getInputStream() );
      fileItem.getInputStream().close();

      fieldNamesAndValues += "<b>" + fieldName + "</b>:";

      boolean fileChosen = !fileItem.getName().equals("");
      if( fileChosen ) {
        fieldNamesAndValues += " file=" + fileItem.getName() +
          " (size=" + fileItem.getSize() + ")";
```

```
        }
        else {
          fieldNamesAndValues += " --no file was chose--";
        }
      }

      result.fieldNamesAndValues = fieldNamesAndValues;
      return result;
    }

  }
```

Remember: methods handling form posts use the `DirectFormPostMethod` annotation, instead of `DirectMethod` or `DirectPollMethod`.

The method that handles the request must receive two parameters. The first parameter 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, as shown in the example. If your form has no input file fields, this map will be empty.

Now, what if you are using a JDK that does not support annotations, such as JDK 1.4? No problem, just add "`djnform_`" as a prefix to the method name, as follows:

```
public Result djnform_handleSubmit( // ...
```

Again, the method name as seen by Javascript will still be `handleSubmit`.

By the way, there is an interesting feature in this demo: we are returning a `Result` class in the Java code, which is a complex object. How do we access its fields from Javascript? Just access the fields using the names they have in Java. In our example the file content is accessed as `result.fileContents`.

6. 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 your 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 `"message"` as the `event` 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 namespace we specified 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, as specified in the `RequestPollMethod` annotation.

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: DjnTestApi.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.

If you are using a JDK that does not support annotations, just add "`djnpoll_`" as a prefix to the method name, as follows:

```
public String djnpoll_test_pollWithBaseParams( //...
```

Again, the method name as seen by Javascript will still be `handleSubmit`.

7. DirectStore with DirectJNgine

The `DirectStore` is an *Ext* store that uses *Ext Direct* to load data. We provide en example of how to use *DirectJNgine* 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>DirectJNgine</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.

8. 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, there will be a time when you will 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`: you'll have access to all the configuration options 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` *DirectJNgine* 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 *DirectJNgine*:

```
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 *DirectJNgine* configuration as the second one. We think the code is pretty much self-explanatory.

Now you have to tell *DirectJNgine* that you want to use a 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!
  }
}
```

Do not forget to set the `gsonBuilderConfiguratorClass` servlet parameter too!

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 too. 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,
                                  JsonSerializationContext 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.

9. DirectJNgine Optimization

Optimizing api file generation and usage

Minimizing network traffic is one of the most important optimizations we can perform for a web application. Therefore, we need to take care of how we handle api files.

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.apiFile` parameter. If you want two different apis to be written to the same file, use the same file name in the `apiFile` 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 *bandwith.*

  In fact, *DirectJNgine* generates three versions of a file. If you specified *abc.js* int the `apiFile`, 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 unnecesary 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.*

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 `batchRequestsMultithreadingEnabled` 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.getMaximumPoolSize`.
- `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 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.

10. 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 *DirectJNgine* 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.

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

```
log4j.logger.com.softwarementors.extjs.djn=INFO
```

The traces at the `INFO` level are completely adecuate 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 *DirectJNgine* 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 all kinds of logging info.

If you suspect that DirectJNgine is not working correctly, or just to learns 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.extjs.djn.Timer=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 *DirectJNgine*, and how much by your own code), etc.

Undestanding 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
```

11. Not explained here -but available

Unfortunately, there are several features that are fully implemented and tested, but are not explained in this *User's Guide*.

In many cases you can learn how they work by looking at *DirectJNgine* tests in *DjnTests.js*. In other cases, you can take a look at our own demos, or how we ported the ExtJs *examples/direct* demos.

Some of the most important of these features are:

- Form loading and submitting via ExtJs's `api` configuration parameter: check the *direct-form.js* example "port".
- Server based validation of forms (i.e., providing per-field errors): check the *direct-form.js* example "port".
- Tree loading: check the *direct-tree.js* example "port".

12. How reliable is all of this?

At the moment of writing the first version of this document, we have more than 70 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 adecade 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.

13. Call for help

We are releasing this library in the hope that it is useful to the programming community.

We understand that this is the first public beta release of the library, which has been tested in a very restricted environment. Unfortunately, that can only guarantee that there is not way for it to be feature complete or bug free.

It is only natural that we will be happy to receive feedback.

Now, receiving feedback it in the form of automated tests that can be added to those in *DjnTests.js*, if at all possible, will allow both you and us to remain focused -the key to quality. And, of course, it will make it much more likely that your concerns are addressed, for our time is very limited.

Thanks in advance!

14. Licensing

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