# DirectJNgine
## User's Guide

*Version 0.7.1 beta*
*for ExtJs 3.0 RC2*

## Acknowledgments

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

## 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.
- Annotations-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.
- Debug mode support.
- Fully automated tests: more than 70 unitary tests are executed every time there are changes to the code.
- Possibility to call public, private, package and protected instance or static methods in public or private classes.
- Client-side call parameter checks.
- Detailed logging, to support easy diagnostic of problems.
- 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*: 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.

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 `@DirectFormMethod`, which is implemented as follows:

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

  @DirectPostMethod
  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 `DirectPostMethod` 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, 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.

## 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. Other features*

Due to a lack of time, there are several features that are fully implemented but are not explained in this *User's Guide*.

*However*, the features are fully implemented and tested, and you can get to know how they work by looking at the tests in *DjnTests.js*. Besides, in some cases, you can take a look at how we ported the ExtJs *examples/direct* demos for further information.

These features are:

- Form loading: check the *direct-form.js* example "port".
- Client parameter checking: check the tests.
- Tree loading: check the *direct-tree.js* example "port".

## 9. 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.

## *10. 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!

## 11. Licensing

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