© 2000 - 2007 JetBrains, Inc. All rights reserved.

Alexey Efimov

The Basics of Plugin Development for IntelliJ IDEA

This article helps you to quickly understand the basics of plugin development for IntelliJ IDEA in order to start writing your own plugins. It describes general plugin design principles, descriptor syntax, and publishing. It also contains a plugin example with step-by-step instructions on how to create it.

Constraints

This article assumes you are using IntelliJ IDEA **5.1** or higher, or any *EAP* build of IntelliJ IDEA starting from #**4121**. Some functionality or classes from the **IntelliJ IDEA Open API** library described in this article may be missing in earlier versions.

Introduction

In IntelliJ IDEA, *plugin* is a separate module that can be attached to IntelliJ IDEA either manually, or using embedded IDE tools.

IntelliJ IDEA plugins can be grouped according to their functionality. Major groups include:

- Code inspections and refactoring (Inspection Gadgets, Intension Power Pack, Refactor-J, Refactor-X, etc.)
- Custom editors (such as Images) and tool windows (SQLQuery, PsiViewer, etc.)
- Language support (Groovy, JavaScript, etc.)
- Application server and profiler integrations (Resin, JBoss, Tomcat, etc.)
- Version Control System integrations (CVS, Clearcase, etc.)
- Frameworks and technologies support (Hibernate Tools, IdeaSpring, XPathView, Struts, etc.)
- Integration with external applications (Jira Browser, JFormdesigner, etc.)
- User interface improvements, instrument panels, and menus (TabSwitch, CVS bar, etc.)
- Compilers (Native2Ascii)
- Others, including utility plugins and games (IdeaJad, simpleUML, Tetris, Sokoban, etc.)

IntelliJ IDEA now provides plugin developers with almost all its functionality, so any standard IntelliJ IDEA feature can be extended by plugins. Before IntelliJ IDEA 5.0, there was no possibility to add support for a new language. Today there are plugins

© 2000 - 2007 JetBrains, Inc. All rights reserved.

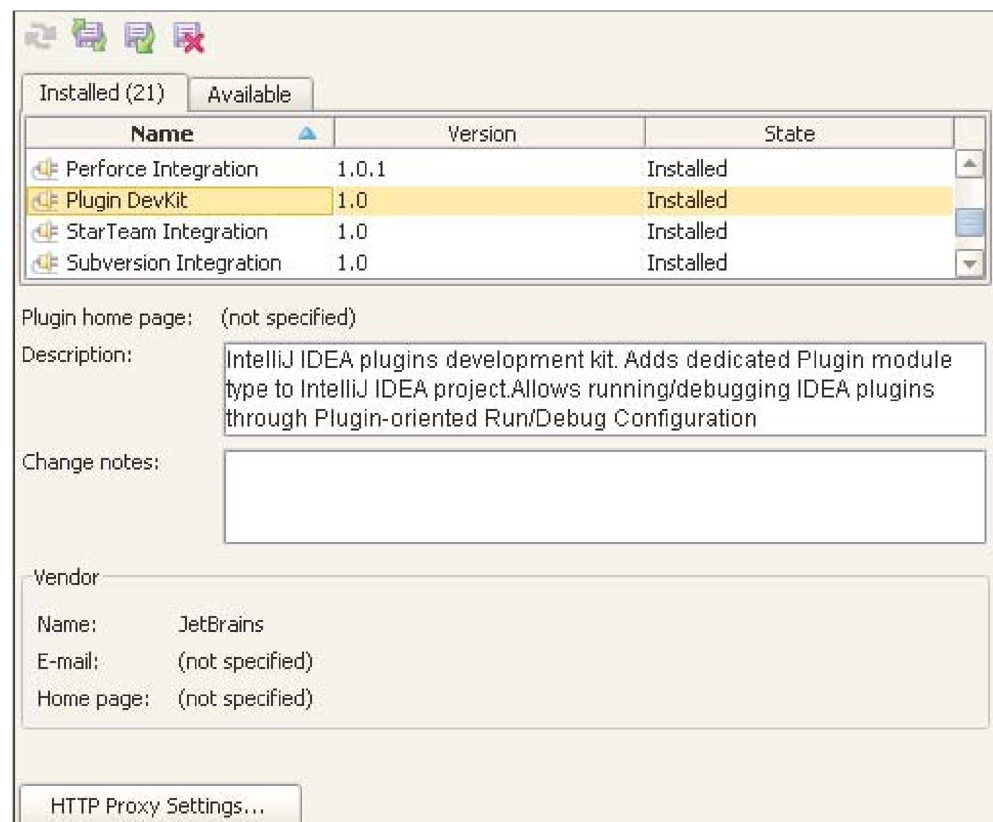
for JavaScript, CSS, Groovy, SQL and others. Little by little, from version to version, IntelliJ IDEA provided plugin developers with new interfaces in the **IntelliJ IDEA Open API** library which has steadily enabled new plugin concepts.

Among the most popular plugins to date are the following:

- **TabSwitch**: a small, but very popular plugin that allows you to switch between open files in IntelliJ IDEA in the same way you switch between Windows applications.
- **SQL Query**: enables writing and executing queries right from the IDE. Sometimes, this plugin is as essential as breathing!
- **Regex**: enables validation of regular expressions right in IntelliJ IDEA.
- **XPathView**: provides the possibility to evaluate XPath expressions against any XML document open in the editor.

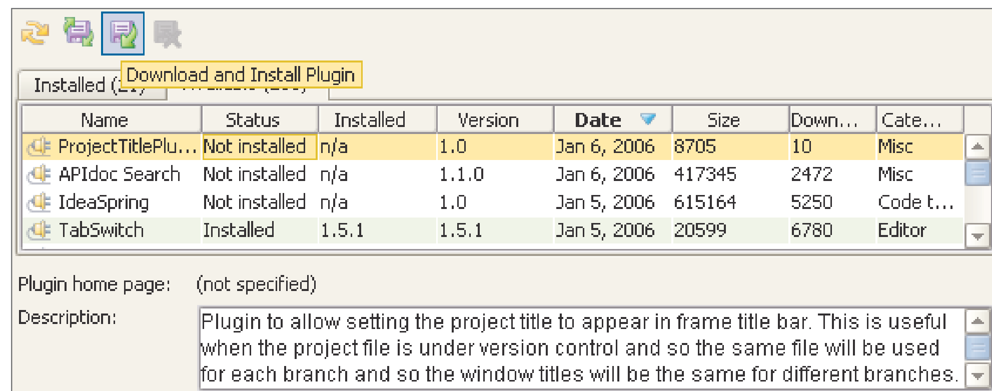
Obtaining Plugins

Registered IntelliJ IDEA plugins are available for download from the **File->Settings->Plugins** menu. The Plugin Manager enables adding, updating, and deleting of plugins. After one or more of these operations, it is necessary to restart IntelliJ IDEA to apply changes.



© 2000 - 2007 JetBrains, Inc. All rights reserved.

To download the list of plugins, open the **Available** tab. If you connect to the Internet via a proxy server, make the corresponding settings after clicking the **HTTP Proxy Settings** button. IntelliJ IDEA will request the plugin repository (*http://plugins.intellij.net*) and show the list of available plugins for your IntelliJ IDEA version. You can install one or more plugins at once. Just select the plugins in the list, and click the download icon.



Where to Begin?

Suppose that you have decided to write a plugin that you and your colleagues desperately need. Well, let's try.

First, it is necessary to create a project that will contain your plugins.

Step 1

Download the **Plugin Development Package** if you don't already have it.

Those who develop using an IntelliJ IDEA EAP version need to download an archive file named like `idea4121-dev.zip` located at the *IntelliJ IDEA EAP Access* page (4121 is the number of the EAP build).

Users of a released version should visit the *IntelliJ IDEA Plugin Developers* page and click the **Plugin Development** link to download the archive.

Unpack the archive to the same directory/folder where IntelliJ IDEA is installed.

The package contains only plugin source code and **IntelliJ IDEA Open API** library with JavaDocs. Therefore, it is possible to develop plugins without this package. However, we highly recommend that you do download and install it. This can make your plugin development much easier.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

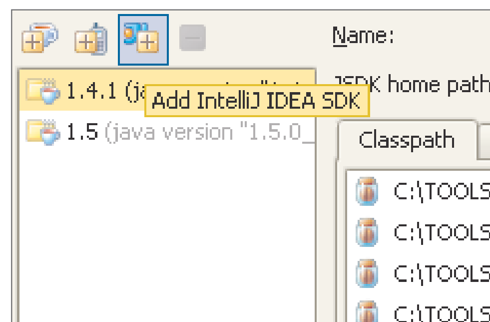
Step 2

After successfully acquiring the Plugin Development Package, we create a project for writing our plugin.

Click **File->New Project**, enter the project name, for example "idea-plugins", then choose its location, and click **Next**.

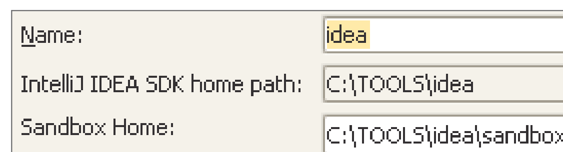
Plugin development requires a special type of Java™ SDK (JDK): the **IntelliJ IDEA SDK**. It is a combination of the standard libraries and the **IntelliJ IDEA Open API** library. You can attach the source code and JavaDocs from the Plugin Development Package to this JDK.

Click the **Configure** button, and the JDK settings dialog box will appear. Click the **Add IntelliJ IDEA SDK** button with the IntelliJ IDEA icon (the button is shown in the screenshot below).



Now select the folder where IntelliJ IDEA is installed (actually, IntelliJ IDEA will select this folder automatically, and you need just to click **OK**).

After that, change the name to something like "**idea**" if necessary. **Sandbox Home** is a folder where IntelliJ IDEA will copy plugins for debugging. If necessary, change the default path for the **sandbox** folder.



It is recommended that you avoid using build numbers in the JDK name. It is inconvenient when you develop using EAP versions, since the set of libraries changes not so frequently. The JDK name like «IDEA 4121» can be confusing when setting up future projects.

Click **OK**. You will then see **IntelliJ IDEA SDK** in the list of available project JDKs. Setting up the JDK is a one-time operation – in future you can simply select this JDK from the list.

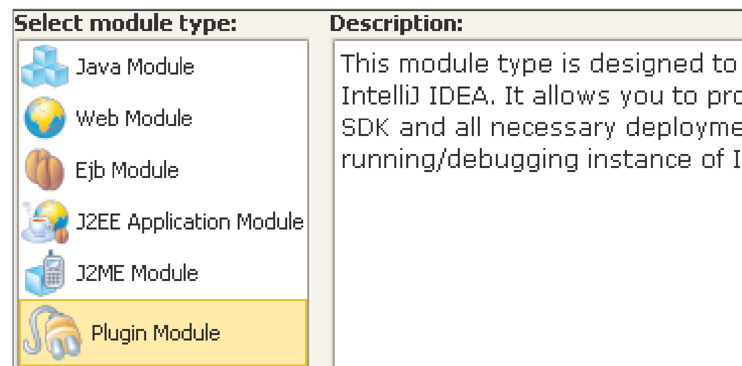
Now click **Next**.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Step 3

On the next screen, select **Create single-module project** and click **Next** once again.

Now it is necessary to select the module type. We need **Plugin Module**.



Then we have to define the module name and paths. Assuming we are newbies, let's create the **HelloWorld** plugin. So the module name is self-evident.

Module name:
helloWorld
Module content root:
C:\IdeaProjects\idea_plugins\helloWorld

Click **Next** again; then at last, click **Finish**.

Step 4

Congratulations! You have just created a plugin project. Now let's investigate what we've got.

Expand the project tree in the **Project** tool window, and find the `plugin.xml` file (in the **META-INF** folder). This is the plugin descriptor that contains general information about the plugin including the plugin name and description, version, the lowest IntelliJ IDEA build number with which it works, as well as the component and actions description. This file is a starting point for loading a plugin in IntelliJ IDEA. If you have created a component but forgotten to specify it in this file – be sure, the component will not be loaded.

So then, the `plugin.xml` file contains various descriptions that are better not to forget about.

Let's make the file consistent – that is, specify its name, description, and other information. Look at the `since-build` attribute. Later we will consider it in detail, but for now just change its value to the IntelliJ IDEA build number in the **Help->About** dialog.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

File: META-INF/plugin.xml

```
<!DOCTYPE idea-plugin PUBLIC
"Plugin/DTD""http://plugins.intellij.net/plugin.dtd">
<idea-plugin>
   <name>HelloWorld</name>
   <description>This plugin does nothing</description>
   <version>1.0</version>
   <vendor>JetBrains</vendor>
   <idea-version since-build="4121" />
</idea-plugin>
```

Now we are completely ready to start writing our first plugin.

How Plugins Work?

Before continuing to develop the Hello World plugin, I would like to tell you a little about how IntelliJ IDEA plugins work. All classes and interfaces described in this section belong to the **IntelliJ IDEA Open API** library included in the Plugin Development Package.

IntelliJ IDEA's Component Model

IntelliJ IDEA has a multilevel component model based on *PicoContainer*. Components belong to some containers, and containers are located on different levels.

There are three levels: *application*, *project*, and *module*. On the *application* level there can be several *project*-level containers; on the *project* level there can be several *module*-level containers.

Application-level components. There can be only one application-level component instance for the whole IntelliJ IDEA application. To create an application-level component it is necessary to implement the *ApplicationComponent* interface, and then register this implementation in the `plugin.xml` file, in the `application-components` section.

Project-level components. There can be only one project-level component instance for each opened project. So, there can be several instances for the IntelliJ IDEA application. To create a project-level component, it is necessary to implement the *ProjectComponent* interface, and then register this implementation in the `plugin.xml` file, in the `project-components` section.

Module-level components. There can be only one module-level component instance for each project module. So, there can be several instances for a project. The total number of instances is equal to the number of the loaded modules of all projects. To create a module-level component, it is necessary to implement the *ModuleComponent* interface, and then register this implementation in the `plugin.xml` file, in the `module-components` section.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

The number of project-level component instances is not equal to, but greater than the number of opened projects in IntelliJ IDEA. Besides components belonging to the opened projects, one more project-level component is loaded because IntelliJ IDEA always has one hidden project that is **Template Project**. This project is loaded when the first project is opened and is not unloaded while IntelliJ IDEA runs.

Loading Components

Application-level components are loaded at the application start. Project- and module-level components are loaded together with the project.

A component is initialized by calling the following methods (implemented for each component):

1. The component is created by its constructor.
2. If the component implements the *JDOMExternalizable* interface, the object is deserialized from XML (*JDOM* model) by the *readExternal* method.
3. The *initComponent* method is called.
4. If the component belongs to the project or module level, the *projectOpened* method is called.

   If the component belongs to the module level, the *moduleAdded* method is called.

If we need to access other components from our component, we can simply list them in the constructor. In this case, IntelliJ IDEA automatically initializes dependent component's data, so the constructor will get already-initialized components.

Dependent components initialization example

```
package com.intellij.tutorial.helloWorld;
import ...

public class MyComponent implements ApplicationComponent {
  private final MyOtherComponent otherComponent;
  public MyComponent(MyOtherComponent otherComponent) {
    this.otherComponent = otherComponent;
  }
  ...
}
```

Unloading Components

Application-level components are unloaded on application close. Project- and module-level components are unloaded on project close.

The following methods are called when unloading a component:

1. If a component implements the *JDOMExternalizable* interface, the component state is saved in XML (*JDOM* model) by the *writeExternal* method.
2. If a component belongs to the project or module level, the *projectClosed* method is called.
3. The *disposeComponent* method is called.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Component Containers

As mentioned above, there are three types of containers. An application containerimplements the *Application* interface; a project container implements *Project*; a module container implements *Module*. Each container has its own methods for acquiring its components.

Getting components from the application-level container

```
Application application = ApplicationManager.getApplication();
MyOtherComponent otherComponent =
  application.getComponent(MyOtherComponent.class);
```

It is rather easy to get an application-level container, since there is only one container of this type (see example above). It is more difficult to get project- and module- level containers. Usually it is necessary to specify them directly in the component constructor or get them from an event handler’s context of some action.

Passing a container as a constructor parameter

```
package com.intellij.tutorial.helloWorld;
import ...
public class MyProjectComponent implements ProjectComponent
{
  private final Project project;
  public MyProjectComponent(Project project) {
    this.project = project;
  }
  ...
  public void foo() {
    MyOtherProjectComponent otherProjectComponent =
      project.getComponent(MyOtherProjectComponent.class);
  }
}
```

In the previous example, the component gets its container and saves the reference to it inside the component instance. If it requires any other component in one of its methods, it will use this reference. Be careful when passing this reference to other components (especially application-level ones). If an application-level component does not release the reference, but saves it inside itself, all the resources used by a project or module will not be unloaded from the memory on the project closing.

A more complex example for getting a container inside the event handler is presented below. In this example, the *DataContext* class is used for passing context references for the considered action. This class deserves special attention, and we will consider it in detail in another article devoted to actions and passing context parameters principles.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Getting a container inside an action's event handler

```
package com.intellij.tutorial.helloWorld;
import ...
public class MyAction extends AnAction {
   public void actionPerformed(AnActionEvent e) {
      DataContext dataContext = e.getDataContext();
      Project project =
         (Project)dataContext.getData(DataConstants.PROJECT);
      Module module =
         (Module)dataContext.getData(DataConstants.MODULE);
   }
}
```

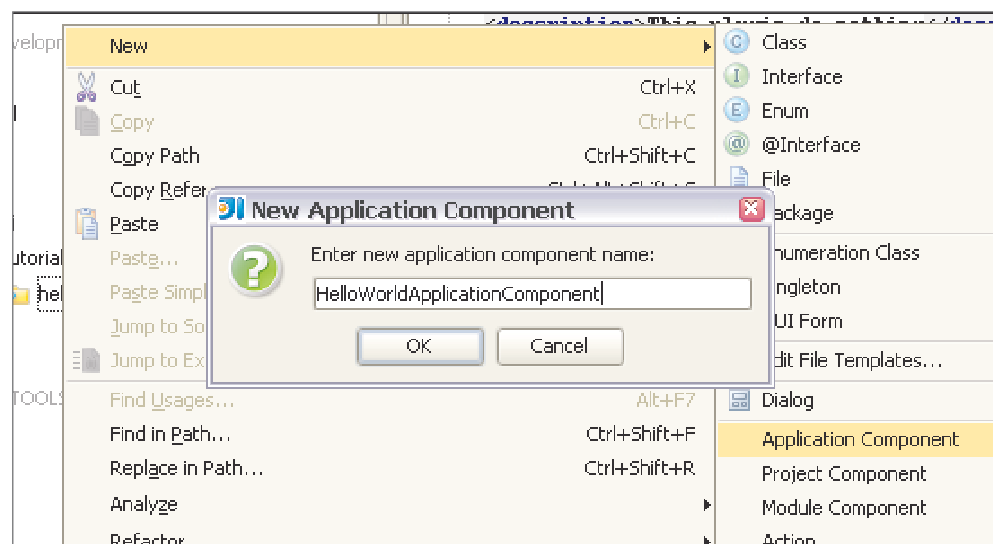
HelloWorld Plugin

Now let's come back to our HelloWorld plugin. Let's add a component to it that will display the "Hello World!" message in a dialog box.

Creating Component

Create a package named `com.intellij.tutorial.helloWorld` inside your project. Right-click this package, and then on the context menu, click **New->Application Component**.

Enter the component name: `HelloWorldApplicationComponent`.



Click **OK**, and IntelliJ IDEA will create the component and register it in the `plugin.xml` file.

Let's add the `sayHello` method to the component. We will use the *Messages* utility class for displaying the message.

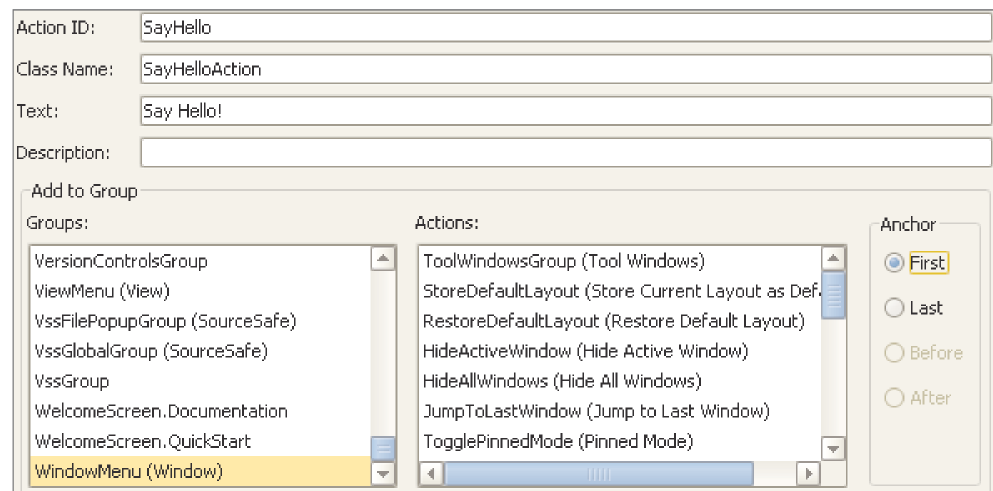
© 2000 - 2007 JetBrains, Inc. All rights reserved.

File: /com/intellij/tutorial/helloWorld/HelloWorldApplicationComponent.java

```
package com.intellij.tutorial.helloWorld;
import ...
public class HelloWorldApplicationComponent implements
   ApplicationComponent {
   public void initComponent() {
   }
   public void disposeComponent() {
   }
   public String getComponentName() {
      return "HelloWorldApplicationComponent";
   }
   public void sayHello() {
      // Show dialog with message
      Messages.showMessageDialog(
         "Hello World!",
         "Sample",
         Messages.getInformationIcon()
      );
   }
}
```

Creating Action

Now we have a component that says "Hello World" like a parrot. Let's add a menu item that allows the user to see this phrase. Once again, right-click the `helloWorld` package, and on the shortcut menu, click **New->Action**. Then enter the action ID and class name, and then select the **WindowMenu (Window)** from the **Groups** list (see the screenshot below).



Click **OK**, and IntelliJ IDEA will create the menu item's event handler and register it in the `plugin.xml` file. Now we need to call the `sayHello` method of our component.

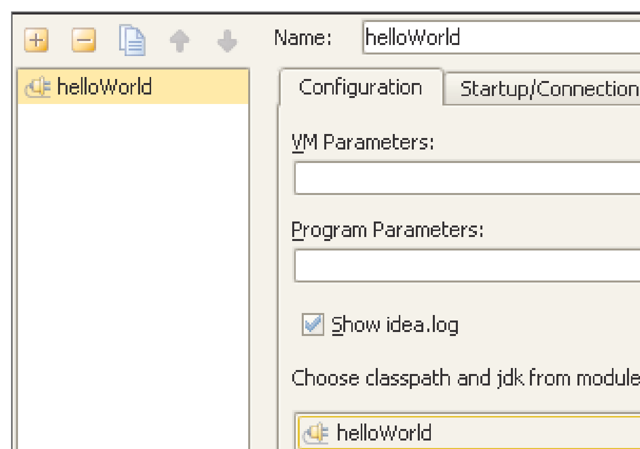
© 2000 - 2007 JetBrains, Inc. All rights reserved.

File: /com/intellij/tutorial/helloWorld/SayHelloAction.java

```
package com.intellij.tutorial.helloWorld;
import ...
public class SayHelloAction extends AnAction {
  public void actionPerformed(AnActionEvent e) {
    Application application =
    ApplicationManager.getApplication();
    HelloWorldApplicationComponent helloWorldComponent =
      application.getComponent(
        HelloWorldApplicationComponent.class);
    helloWorldComponent.sayHello();
  }
}
```

First Plugin Launching

We need to set up a run configuration for our plugin so we can launch and/or debug it. Click the **Run->Edit Configurations** menu. In the **Run/Debug Configuration** dialog box, click the **Plugin** tab, and then add a new configuration. Then click **OK**.



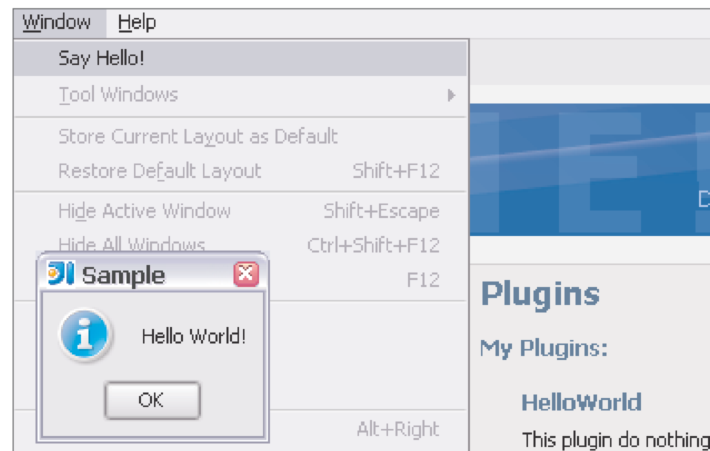
Now we need to just press **Shift+F9** (or click the **Run->Debug** menu) to start the plugin.

For debugging and testing a plugin, IntelliJ IDEA starts one more instance of itself. In this instance the plugin will be available. IntelliJ IDEA uses the **sandbox** folder for saving settings of this instance and copying the plugins for debugging and testing.

Wait until the IntelliJ IDEA's debug instance starts, and then switch to that window. Open the **Window** menu, and click the **Say Hello**! menu item.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

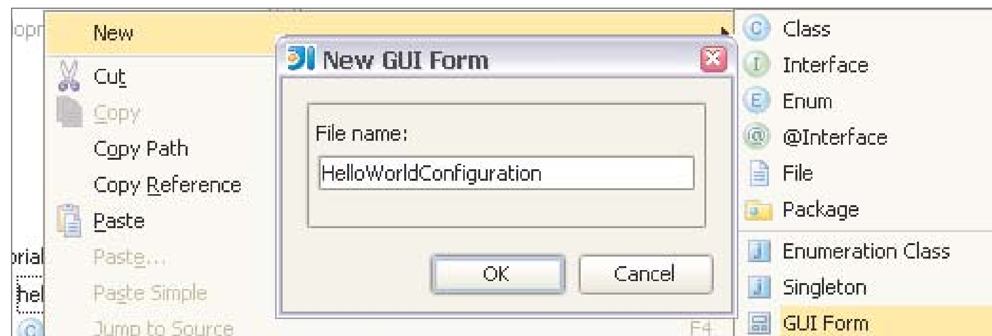
Wait until the IntelliJ IDEA's debug instance starts, and then switch to that window. Open the **Window** menu, and click the **Say Hello**! menu item.



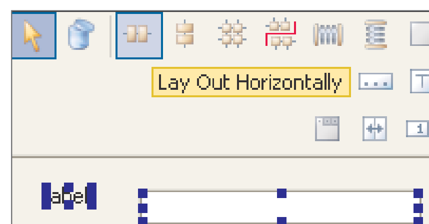
Adding Configuration

So far, our plugin can only say "Hello World!" like some idiot parrot. Let's make it a smart parrot and teach it some other phrases. To do this, we need to add a configuration form for the plugin and a component to be responsible for saving this configuration. The form will contain just one text field.

Right-click the `helloWorld` package, and on the shortcut menu click **New->GUI Form**. Enter the form name: `HelloWorldConfiguration`.

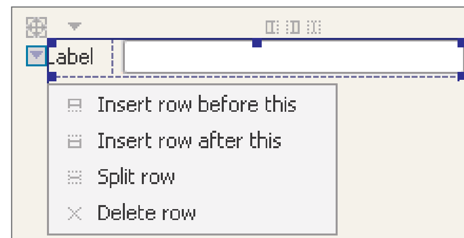


In the form editor, drag **JLabel** and **JTextField** elements and drop them anywhere inside the form. Put them one next to another as shown on the screenshot below. Then select the elements and click the **Lay Out Horizontally** button.

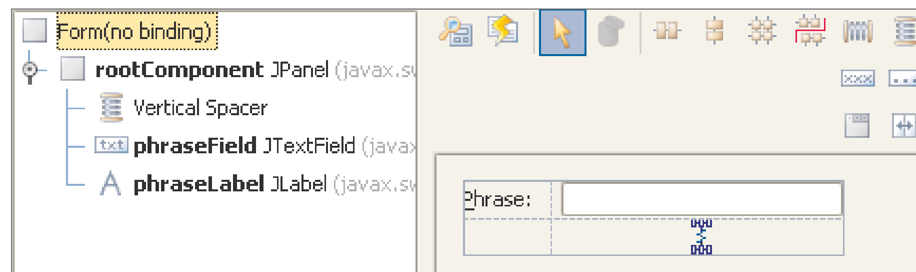


© 2000 - 2007 JetBrains, Inc. All rights reserved.

IntelliJ IDEA creates a horizontal panel and places elements inside it. Select the JPanel element in the editor and you will see the small arrow to the left of the panel. Click the arrow, and then click **Insert row after this** to add one more row to the grid.



Then select the **Vertical Spacer** element on in the instrument panel and drag it to t he bottom right cell of the grid.



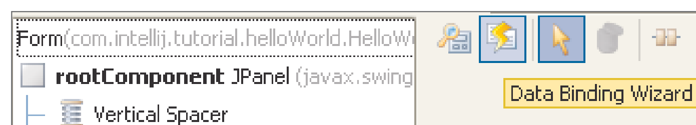
In the element tree (on the left side of the form editor), select the **JPanel** element. In the property table below the tree, set the JPanel's **binding** property to `rootComponent`. For the text field (**JTextField**) set the **binding** property to `phraseField`, and for the label (**JLabel**) to `phraseLabel`. Now the form is ready for binding to a Java class.

Create a class in the same package and name it `HelloWorldConfigurationForm`.

Switch back to visual form editor and set the **binding** property of the whole form to this class.

After that, IntelliJ IDEA will highlight the elements in the tree indicating that it is necessary to create fields representing the form elements in the `HelloWorldConfigurationForm` class. You can solve these problems by pressing **Alt + Enter** for each element.

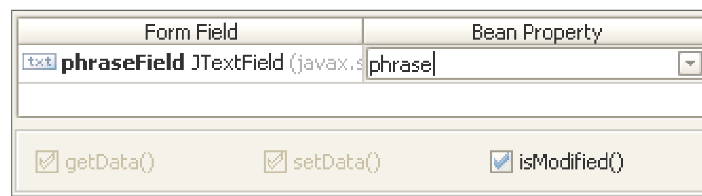
When all the conflicts are resolved, on the instrument panel of visual form editor, click the **Data Binding Wizard** button (see the screenshot below).



© 2000 - 2007 JetBrains, Inc. All rights reserved.

The wizard will create all the necessary code for reading and configuring the form state. On the first screen, select the **Bind to existing bean** option, and specify the `HelloWorldApplicationComponent` class. Then click **Next**.

Now, in the **Bean Property** cell, specify the class field name: `phrase`. Then click **Finish**. IntelliJ IDEA will create the code for writing and configuring the form state in the `HelloWorldConfigurationForm` class (i.e. the phrase field, its getter and setter will be generated). It also will add the `isModified` method that helps to detect whether the form data has changed.



Now we need to modify the `HelloWorldConfigurationForm` class, adding a method for getting the root form element - `getRootComponent`.

File: /com/intellij/tutorial/helloWorld/HelloWorldConfigurationForm.java

```
public class HelloWorldConfigurationForm {
   private JPanel rootComponent;
   private JTextField phraseField;
   private JLabel phraseLabel;
   public HelloWorldConfigurationForm() {
      // Enable mnemonic action
      phraseLabel.setLabelFor(phraseField);
   }
   // Method returns the root component of the form
   public JComponent getRootComponent() {
      return rootComponent;
   }
   public void setData(HelloWorldApplicationComponent data) {
      phraseField.setText(data.getPhrase());
   }
   public void getData(HelloWorldApplicationComponent data) {
      data.setPhrase(phraseField.getText());
   }
   public boolean isModified(HelloWorldApplicationComponent data) {
      return phraseField.getText() != null ?
         !phraseField.getText().equals(data.getPhrase()) :
         data.getPhrase() != null;
   }
}
```

© 2000 - 2007 JetBrains, Inc. All rights reserved.

So then, we have a visual configuration form for editing. This form uses the `phrase` field of the `HelloWorldApplicationComponent` class for saving its state. Now we need to change the `sayHello` method of our component and use the `phrase` field for displaying a message.

Modified sayHello method of the HelloWorldApplicationComponent class

```
public void sayHello() {
   // Show dialog with message
   Messages.showMessageDialog(
      phrase,
      "Sample",
      Messages.getInformationIcon()
   );
}
```

To register the component in the IntelliJ IDEA settings (opened by the **File->Settings** menu) we need to change the original plugin component slightly. Now this component should also implement the *Configurable* interface. Below you can see the implementation of this interface in our component.

Implementing the Configurable inteface in the HelloWorldApplicationComponent class

```
package com.intellij.tutorial.helloWorld;
import ...
public class HelloWorldApplicationComponent
   implements ApplicationComponent, Configurable {
   private HelloWorldConfigurationForm form;
   private String phrase;
   public void initComponent() {
   }
   public void disposeComponent() {
   }
   public String getComponentName() {
      return "HelloWorldApplicationComponent";
   }
   public void sayHello() {
      // Show dialog with message
      Messages.showMessageDialog(
         phrase,
         "Sample",
         Messages.getInformationIcon()
      );
   }
   public String getDisplayName() {
      // Return name of configuration icon in Settings dialog
      return "HelloWorld";
   }
   public Icon getIcon() {
      return null;
   }
```

© 2000 - 2007 JetBrains, Inc. All rights reserved.


```
    public String getHelpTopic() {
      return null;
    }
    public JComponent createComponent() {
      if (form == null) {
        form = new HelloWorldConfigurationForm();
      }
      return form.getRootComponent();
    }
    public boolean isModified() {
      return form != null && form.isModified(this);
    }
    public void apply() throws ConfigurationException {
      if (form != null) {
        // Get data from form to component
        form.getData(this);
      }
    }
    public void reset() {
      if (form != null) {
        // Reset form data from component
        form.setData(this);
      }
    }
    public void disposeUIResources() {
      form = null;
    }
    public String getPhrase() {
      return phrase;
    }
    public void setPhrase(final String phrase) {
      this.phrase = phrase;
    }
  }
```

To implement the *Configurable* interface it is necessary to define how to work with the visual form. For example, in the *createComponent* method we create a visual form and return its root element. When the form is shown on the screen, the *apply*, *reset* and *isModified* methods are enabled.

When the user clicks **OK** or **Apply** buttons, the *apply* method is called and the string from the input field is saved in the phrase field of the `HelloWorldApplicationComponent` class. When the form is initialized or the user clicks **Cancel**, the *reset* method is called and all the changes are replaced by old values. The form is checked for changes at regular intervals by calling the *isModified* method. If the method returns `false`, the **Apply** button is disabled.

When the user closes the form, the *disposeUIResources* method is called. In this method you can release the resources used by the visual form (in our example the reference to the form is set to `null`).

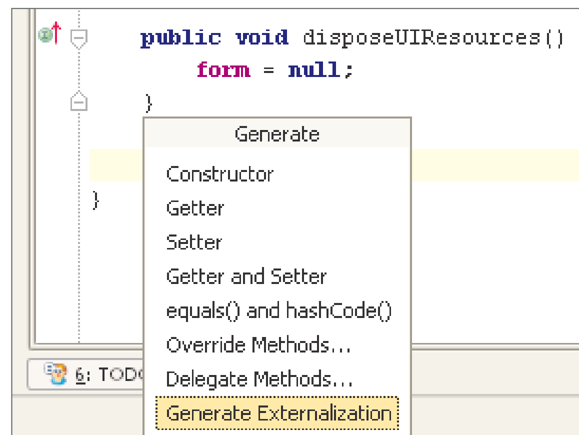
© 2000 - 2007 JetBrains, Inc. All rights reserved.

Perhaps, you have noticed that in our example the form is created in the *createComponent* method and then is disposed in the *disposeUIResources* method. During the next call, the *createComponent* method creates the form once again. This is because IntelliJ IDEA does not set the component to the current Look&Feel at each call of the *createComponent* method. So, to make the form look the same as all other forms (i.e. inherit the current IntelliJ IDEA Look&Feel) it is necessary to recreate the form each time when the *createComponent* method is called.

The *Configurable* interface declares three more methods that can be thought of as "informational". The *getDisplayName* method defines the icon name in the settings panel (opened by **File->Settings**), while the *getIcon* method defines the icon itself. The *getHelpTopic* method returns ID of the corresponding help section. If this method returns some not-null value, the configuration form will have the **Help** button.

Now there is one last modification for our component. We must implement how the state of our component will be saved in the `phrase` field of the `HelloWorldApplicationComponent` class. To enable saving and restoring the state, our component should implement the *JDOMExternalizable* interface.

In IntelliJ IDEA you can quickly generate the methods for saving and restoring states of components. Open to the `HelloWorldApplicationComponent` class, press **Alt+Insert**, and in the appeared list click **Generate Externalization**.



This action is available only for IntelliJ IDEA components that do not already implement the *JDOMExternalizable* interface.

After this action, IntelliJ IDEA will implement the *JDOMExternalizable* interface in the component and add the basic implementation of the *readExternal* and *writeExternal* into it.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Results of generating saving and restoring state methods

```
package com.intellij.tutorial.helloWorld;
import ...
public class HelloWorldApplicationComponent
      implements ApplicationComponent, Configurable,
      JDOMExternalizable {
  ...
  public void readExternal(Element element)
      throws InvalidDataException {
      DefaultJDOMExternalizer.readExternal(this, element);
  }
  public void writeExternal(Element element)
      throws WriteExternalException {
      DefaultJDOMExternalizer.writeExternal(this, element);
  }
}
```

As you can see from the example, the *DefaultJDOMExternalizer* utility class is used for saving and restoring the state. This class operates with the public class fields of the passed object. In our case, the phrase field is private, so it will be ignored. To make this field available for the *DefaultJDOMExternalizer* utility class we need to change its access modifier to public. We should also specify the default value for this field.

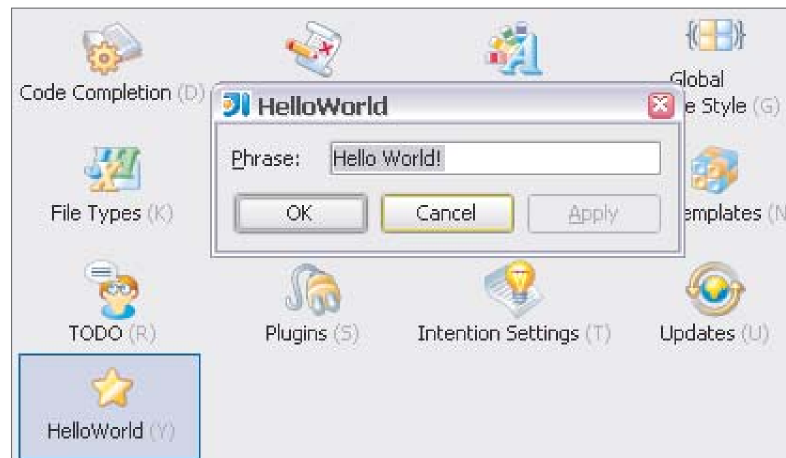
Changing the phrase field's access modifier in the HelloWorldApplicationComponent class

```
package com.intellij.tutorial.helloWorld;
import ...
public class HelloWorldApplicationComponent
      implements ApplicationComponent, Configurable,
      JDOMExternalizable {
  ...
  public String phrase = "Hello World!";
  ...
}
```

If it is impossible to make public the fields that require saving and restoring their states, you can work directly with the *JDOM* model (or use the *JDOMExternalizerUtil* class) instead of using the **DefaultJDOMExternalizer** class. The *JDOMExternalizerUtil* class allows you to read and write *JDOM* model's attributes with its *writeField* and *readField* methods.

The plugin is now ready to start. Press **Shift + F9**, switch to open IntelliJ IDEA instance and click the **File->Settings** menu.

© 2000 - 2007 JetBrains, Inc. All rights reserved.



Since we have not specified the icon for our configuration component (the `getIcon` method returns `null`), IntelliJ IDEA displays the default yellow star.

Enter some other string instead of "Hello World!", for example, "Caramba! Corrida!". Save the changes, and then click the **Window->Say Hello** menu.

To check whether our settings are really saved, close the IntelliJ IDEA debug instance and then start it once again. Click the **Window** menu again, and then select **Say Hello**. Voila!

Some Words About Plugin Descriptor

We have just created and started a plugin, but we have hardly edited the `plugin.xml` file because all the necessary operations with it were performed by IntelliJ IDEA. In other cases you may need to edit this file manually, which is why I have decided to present some information about it.

Plugin Identifier

Besides the plugin name you can specify the plugin identifier using the `id` tag. Unlike plugin names, the identifier must not contain spaces. You can use the identifier when specifying plugin dependencies.

Plugin Dependencies

If a plugin uses the API of any other plugin, that dependency must be specified in the `plugin.xml` file in order to provide access to the classes of the plugin on which it depends. By default, plugins do not have access to other plugins' API.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Dependencies between plugins are set up with the help of the `depends` tag where the name or ID of the required plugin is specified. If necessary, you can specify multiple `depends` tags.

Plugin dependencies example

```
<idea-plugin>
  ...
  <depends>MyOtherPluginId</depends>
  <name>MyPlugin</name>
  ...
</idea-plugin>
```

In the example above, the `MyPlugin` plugin uses the API of the plugin with the `MyOtherPluginId` id, or in other words, it depends on that plugin.

Version and Build Number

The plugin version is specified in the `version` tag to inform IntelliJ IDEA whether plugin is up-to-date or should be updated. That is why each new plugin version in the plugin repository should have a unique number. In this case the users can easily update their versions.

The build number is specified in the `idea-version` tag for compatibility of the plugin with IntelliJ IDEA versions. Usually, the `since-build` attribute contains the build number on which the plugin was tested. Otherwise the user may install the plugin on some older IntelliJ IDEA version with which plugin may not be compatible.

The `until-build` attribute is used to remove the plugin from the list of available plugins starting from some build number.

Version and build number specification

```
<idea-plugin>
  ...
  <name>MyPlugin</name>
  <version>1.0</version>
  <idea-version
    since-build="4081"
    until-build="4109"/>
  ...
</idea-plugin>
```

The example above shows the `MyPlugin` plugin, version `1.0` that works with IntelliJ IDEA from build `4081` till build `4109`, inclusive.

The `since-build` attribute value may not be equal to the IntelliJ IDEA build number on which the plugin was tested. If you are sure that changes in the new plugin version are compatible with the previous IntelliJ IDEA builds, it is possible *not* to change the attribute value. But it is better to test the plugin on the version specified in `since-build`.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Components Registration

Components are registered in the `application-components`, `project-components`, and `module-components` sections. When registering a component, the implementation class specification is mandatory, while interface class specification is optional. Hiding the implementation can help you when extracting interface components.

Interface registration for the component

```
<component>
  <interface-class>org.MyComponent</interface-class>
  <implementation-class>org.impl.MyComponentImpl</implementation-class>
</component>
```

With such a registration you can ignore the component implementation and operate only with its interfaces. The `org.MyComponen` interface (or class) does not necessarily extend or implement the *ApplicationComponent* interface. It can be an ordinary interface or class. But its implementation must be *ApplicationComponent* in this case.

An advantage of such an approach is that nobody can gain access to `org.impl.MyComponentImpl` directly (for example, it can be package visible), but rather will work only with an interface.

We have seen earlier how to get components directly from the container. It is easy, but in most cases it makes the code cumbersome. That is why some IntelliJ IDEA components have special static methods that access the container.

Example of the static method implementation for getting project-level component

```
package com.intellij.tutorial.helloWorld;
import com.intellij.openapi.project.Project;
...
public abstract class MyProjectComponent {
  public static MyProjectComponent getInstance(Project project) {
    return project.getComponent(MyProjectComponent.class);
  }
  public abstract void foo();
...
}
```

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Example of the hidden implementation of the project-level component

```
package com.intellij.tutorial.helloWorld.impl;
import ...
final class MyProjectComponentImpl extends MyProjectComponent
   implements ProjectComponent {
   ...
   public String getComponentName() {
      return "MyProjectComponent";
   }
   public void foo() {
      // Sample
   }
}
```

As can be seen from the examples above, implementation of this component is not visible – the class is not public and is located in another package. The user of this API will see nothing but the interface or the abstract class.

When only one class is specified (it does not matter whether it is specified through `interface-class` or `implementation-class`) it is considered as the component implementation.

Project-level components that implement *JDOMExternalizable* save their states in the project file (IPR) by default. This behavior can be changed by adding a special directive into the component description.

Example of the project-level component saving its data in the IWS file

```
<component>
   <implementation-class>org.MyProjectComponent</implementation-class>
   <option name="workspace" value="true" />
</component>
```

Finally, check that there are no spaces or line breaks within the reference to the interface or implementation – this causes an error when the plugin loads.

! Example of incorrect component registration

```
<component>
   <interface-class>
      org.MyComponent
   </interface-class>
   <implementation-class>
      org.impl.MyComponentImpl
   </implementation-class>
</component>
```

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Localization

You can localize the plugin description. For this purpose, specify the resource bundle with the plugin description. For example:

File: plugin.xml

```
<idea-plugin>
  <name>MyPlugin</name>
  <description>My description</description>
  <resource-bundle>org.MyBundle</resource-bundle>
  ...
</idea-plugin>
```

File: /org/MyBundle_en.properties:

```
plugin.MyPlugin.description=My description
...
```

The property key is based on the plugin name with the "plugin" prefix and tag name suffix. The `description` tag is available for localization.

The `description` tag must be presented in the `plugin.xml` file even if you have specified it in all the resource bundles. This is required by the plugin repository server that reads the plugin description only from the `plugin.xml` file when loading plugin and ignores the localization.

Along with the plugin description you can also localize actions described in the `plugin.xml` file. For this purpose, the specified resource bundle should contain all properties the keys which should contain action identifiers with the "action" prefix and the ".text" or ".description" suffix.

File: plugin.xml

```
<idea-plugin>
  <resource-bundle>org.MyBundle</resource-bundle>
  ...
  <action class="org.MyAction" id="MyAction" />
  ...
</idea-plugin>
```

File: /org/MyBundle_en.properties

```
action.MyAction.text=My Action
action.MyAction.description=My action description
...
```

For action localization, `text` and `description` attributes in the action description in the `plugin.xml` file are not required.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Plugins Portability

When developing plugins, it is important to remember that IntelliJ IDEA is a cross-platform application. The most popular operating systems among IntelliJ IDEA users are **Windows**, **Linux**, and **MacOS**. On Windows and Linux IntelliJ IDEA usuallyworks under JRE 5.0, but on MacOS it is forced to use JRE 1.4. To make the plugin universal, i.e. working on any operating system, you have to make it compatible with JRE 1.4.

The IntelliJ IDEA SDK is based on the same JDK that is used by IntelliJ IDEA itself. If it is JDK 5.0, there can be problems with making the source code that works under JDK 5.0 compatible with JDK 1.4. To translate the code from JDK 5.0 to JDK 1.4 you can use code translators such as *RetroWeaver* or *RetroTranslator*.

Additionally, in IntelliJ IDEA you can enable highlighting of the new API. For this purpose, open **File->Settings->Errors**, find the **J2SDK5.0 specific issues and migration** category, and switch on the **Usage of API documented as @since 1.5** inspection.

There is an undocumented feature that can compile JDK 5.0 code to JDK 1.4 compatible bytecode. To do this, in the Javac compiler settings (**File->Settings->Compiler**) of the project, specify `-target jsr14` parameter in the **Additional command line parameters** field. But remember that this is an "undocumented feature", so the code can be prepared for loading on JRE 1.4 only roughly, or approximately.

Plugin Publishing

Before concluding this article, I'd like to tell you about uploading developed plugins to the plugin repository (*http://plugins.intellij.net*).

Creating Archive

Before uploading a plugin, you need to create a ZIP or JAR plugin archive. For this purpose, right-click the module in the Project view, and on the shortcut menu, click **Prepare Plugin Module '<name>' For Deployment**.

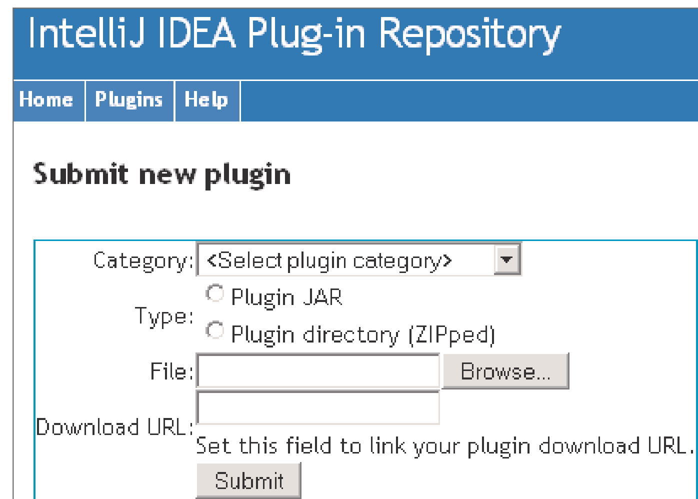
If the module does not depend on libraries, the JAR archive will be created. Otherwise, a ZIP archive will be created that will include all the plugin libraries specified in the project settings.

Uploading Plugin into Repository

To upload a new plugin to the repository or update an existing plugin, visit the *http://plugins.intellij.net* site. On the site, click the **Plugins** tab, and then click the **Add Plugin** link for loading a new plugin or the **Add Version** link to update an existing plugin. Then select the already created ZIP or JAR archive, specify the archive type, and click **Submit**. After that the archive will be uploaded to the server.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

If registration succeeds, you will see information about the uploaded plugin version. Now the plugin can be downloaded by other users through their versions of IntelliJ IDEA (of course, only if a plugin is compatible with their versions).



It may take some time for a plugin to appear in the list of available plugins in IntelliJ IDEA Plugin Manager, since the server needs time to build the list of plugins in the whole repository. Usually, it takes less than a minute.

Notifying About New Plugin Version

Usually after a new plugin version is loaded, the plugin author posts a new message in the *Plugins* forum saying that a new version is available through the Plugin Manager starting from a certain build number. The message also describes changes made in the new version. This forum is probably the best place to inform the world about your new plugin.

Summary and Useful Links

So – we have examined the IntelliJ IDEA component model, gotten some useful information about the `plugin.xml` file, written our first plugin, configured it, and learned how to publish it.

Of course, it is impossible to tell you everything about plugin development in IntelliJ IDEA just in one article. But I hope it helped you to make your first step to writing your own plugins for IntelliJ IDEA.

- *IntelliJ IDEA Plugins* – IntelliJ IDEA Plugins Home Page.
- *IntelliJ.org TWiki* – Wiki site about IntelliJ IDEA and its plugins. Some plugins have its own pages on this site.
- *onBoard* – online magazine for developers. Here you can find articles written by IntelliJ IDEA developers and other JetBrains staff.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Acknowledgements

I am extremely grateful to the great people at JetBrains who helped me create this article. Kudoz go to:

Alexandra Rusina, for painstakingly going through my work and fully translating it into English;

Dmitry Jemerov, for his active assistance and helpful technical advice;

Alex Tkachman and Ann Oreshnikova, for their invaluable guidance and constructive criticism;

Robert Palomo, for masterfully adapting the English text;

Julia Repina, for lending us her wonderful design skills.

© 2000 - 2007 JetBrains, Inc. All rights reserved.

Contents

Constraints	1
Introduction	1
Obtaining Plugins	2
Where to Begin?	3
Step 1	3
Step 2	4
Step 3	5
Step 4	5
How Plugins Work?	6
IntelliJ IDEA's Component Model	6
Loading Components	7
Unloading Components	7
Component Containers	8
HelloWorld Plugin	9
Creating Component	9
Creating Action	10
First Plugin Launching	11
Adding Configuration	12
Some Words About Plugin Descriptor	19
Plugin Identifier	19
Plugin Dependencies	19
Version and Build Number	20
Components Registration	21
Localization	23
Plugins Portability	24
Plugin Publishing	24
Creating Archive	24
Uploading Plugin into Repository	24
Notifying About New Plugin Version	25
Summary and Useful Links	25