PokeriPalvelin plugin tutorial

Hands-on guide to development and deployment of plugins

Contents

Overview....3
Prerequisites....4
  Interface API jar package....4
  Compilation....4
Development....5
  New poker bot AI....5
  New poker variant....5
  New game....5
Deployment....6
Example: Hello game....7

Overview

PokeriPalvelin is a client-server software allowing users to play different kind of multiplayer games. Software is designed from the ground up to be extensible on every aspect. There are few designed points for extension on the architecture but user is at the end free to replace any component if needed.

Typical plugins allow new game types or variations for existing ones. PokeriPalvelin is distributed with plugins for certain poker variants. Poker plugin itself also contains interface for programming new bot players. Software is written in Java and it is encouraged to be used to extend it. In the case where other languages are desired to be used unsupported options are available. There are few popular languages that compile to Java bytecode such as Scala and Groovy. These languages can be used directly with extension API and in case of other languages Java offers JNI (Java Native Interfaces) library to access native languages outside of JVM. Other options include using JSR-223 (Scripting for the Java Platform) that offers easy interfaces to certain scripting languages.

This document describes how to develop and deploy plugins using predefined extension points with Java. Programming poker bots and new variants are also covered.

Prerequisites

Interface API jar package

When project's main sources are compiled ant script generates extension API package named extension_api.jar in the directory Plugins/lib. This package contains interfaces required to develop new plugins.

Compilation

Compiling new plugins requires only Java SDK and ant version 1.70 or newer if you want to use our build files to compile your plugins. Plugins directory distributed with the system use Java 5 EE webservices-tools.jar package containing XJC ant task used to generate Java code directly from XML schema files.

Development

This document assumes that user is using Java as a language and is using Plugins directory as template for new plugins.

New poker bot AI

New bot is done by implementing com.asdf.plugins.pokergames.Player interface. Interface definition is documented with Javadoc styled comments that explain purpose of every to be implemented method. There exists a method for every interesting event happening in poker game. Bot is queried for action with the getAction() method. Bot is passed information about game events through various methods defined by the Player interface. You should read through very carefully our API documentation for extensions. Also we have included two example bots that can be used as a reference.

New poker variant

New poker variant is done by extending abstract com.asdf.plugins.pokergames.PokerGame class. User is free to override any method needed. There are few abstract methods that are mandatory. In addition of PokerGame user is required to implement com.asdf.gui.lobby.CreateGamePanel to have custom create game panel in client portion of the application.

Package com.asdf.plugins.pokergames.gui contains a graphical poker table implementation which should work with most common poker variants without any modifications. You should consider using it with your own poker game implementation at client side.

New game

New game is developed by inheriting com.asdf.common.Game class. Class contains methods for receiving GameRequest and GameResponse objects. Games are sending responses with send method found from Game class. On join games get client session ID, which is used to identify different players and clients. On send session ID is required to be specified. Games should not be starting own threads. Everything should be done by storing game state in it's member variables. To use timer functions developer must use setTimer methods from the Game class. These timers synchronize thread access properly.

In addition developer must inherit com.asdf.client.GameController class that defines logic for parsing messages on the client side and CreateGamePanel described above.

Deployment

Poker variants and bots should be packaged to existing poker.jar package. This happens automatically if you place your extensions on existing Plugins sources in the package com.asdf.plugins or it's subpackages it's automatically included in poker.jar. If you place it outside you must modify the compile task in the Plugins/build.xml ant script. New games can copy Plugins structure and change appropriate names in build.xml file.

Alternatively you can compile your bots or game plugins anyway you like. Then you should add them to .jar package and place the jar files in /bots or /plugins directory. It is not strictly necessary to add your class files in jar package. Placing your class files in subdirectory of plugins or bots will also work if you add that path to to client.conf and server.conf

There must be plugins/meta directory containing .meta files that describe available plugins in the working directory where application is started. There exists info.txt in that directory describing exact syntax for meta files. Basically files include fully qualified class names for Game, GameController and CreateGamePanel implementations and possible variants of the plugin.

Poker bots should be described in bots/meta directory also in application's working directory. Meta files contain fully qualified name of the bot and human readable name visible in GUI.

Client.conf and server.conf should contain entries for new plugins if placed somewhere else than in existing jar files.

Example: Hello game

Hello game distributed in plugins directory contains very basic implementations of all required interfaces. You can use these classes as a base for your own implementation.

1. Directory structure

   Lets look at the directory structure of our game first:

   pokeripalvelin – this is the workin directory

   pokeripalvelin\bin – this directory contains the jar files of the main program, server.jar, client.jar

   pokeripalvelin\lib – contains library files, including plugins_api.jar, which you should get familiarized with

   pokeripalvelin\plugins – this is the directory where all the plugins go and it should have hellogame.jar in it

   pokeripalvelin\plugins\meta – here are the metafiles for plugins and it should have hellogame.meta in it

   pokeripalvelin\bots - this is the directory where all the bots go (we do not need to care about that in this tutorial)

   pokeripalvelin\bots\meta - this is the directory where all the metafiles for bots go (we do not need to care about that in this tutorial)

2. Meta file

   Our meta file for hello game, *hellogame.meta*, is in directory *\plugins\meta* and it looks like this:

```
# Example game provided by ASDF-group with the server/client software
#
gamename Hello
panelname com.asdf.plugins.hello.CreatePanel
servergame com.asdf.plugins.hello.ServerSideHelloGameLogic
clientgame com.asdf.plugins.hello.HelloGame
variantnames ExampleGame
```

   After the comments, marked with # is the interesting stuff. We will get to all of these components in detail later, for now we just introduce them shorty.

   - **gamename** is used by client and server when querying this game and also this is what is shown to clients in the lobby.
   - **panelname** is the fully qualified classname of the CreateGamePanel implementation (a class extending com.asdf.client. CreateGamePanel). This is read from the metafile and send by server to clients when available games are queried. You do not need to implement your own CreateGamePanel, but it is strongly recommended since otherwise it will be impossible to create games to our server.
   - **servergame** is the fully qualified classname of the server-side Game implementation. This is the logic that actually runs the game in the server. Or at least the class that starts it. This tells server which class to load when client creates a new hellogame.

- **clientgame** is the fully qualified classname of the client-side GameController implementation. This is the game client that runs in the client side. GameController does not actually need to contain full game logic as server-side Game already does so. GameController only needs to take clients input, send it to server which runs the actual game and receive moves made by other player or whatever. But you could run the full game logic at client-side in GameController and then the server-side Game would act as a hub between clients. This tells lobby which class to load when client joins a game of this type running at server.
- **variantnames** is not strictly required but if you have different variants of your game you can use this field to tell server which those are. These will be shown at lobby and can be used by clients to create games of different variants and also filter results of game search. Note that you can give several **variantnames**, you only need to separate them with whitespace.

3. The main components

This chapter deals with the actual implementation of our interfaces and abstract classes when creating a new game for our server. We start from the server-side and then move on to the client-side. Example code will be shown along with comments explaining what is what, what it does and why it is necessary. Full example code of hellogame is included with the distribution package.

1. com.asdf.plugins.hello.ServerSideHelloGameLogic

This one is quite obvious starting point when creating a new game to this server from scratch – Using the abstract skeleton Game which the creators of this server have kindly made for us.

```
public class ServerSideHelloGameLogic extends Game {
```

`columnNames()` is used to list those columns we want to show at the gamelist for clients. `description()` is used to to set the information contained by those columns. This is done per game basis so that every game fills its own details to an `com.asdf.parser.GameListResponse.Games.Game` which is then returned to the server. You should check details about which fields you need to fill and which are optional from our protocol description message.xsd when developing your own game. If you forget to fill a mandatory field, don't worry. Our schema parser will tell you where you made a mistake when you first try to run or compile your game.

```
@Override
protected String[] columnNames() {
      return new String[]{"Jou", "Jeah", "Jaa", "123"};
}
```

```
/**
 * A small description about the game to the game listing.
 * The bare minimum of information is sent for the purposes of
 * clarity.
 */
@Override
protected com.asdf.parser.GameListResponse.Games.Game description(){

      com.asdf.parser.GameListResponse.Games.Game game = new
com.asdf.parser.GameListResponse.Games.Game();
      game.setGameID(this.getId());
      game.setName(this.getName());
      game.setGameType("Example game");
      game.setMaxPlayers(1);
      game.setPlayers(0);

      // Variant name is required, and affects the categories
      //in client game tree, in which this game can be found.
      game.setGameVariant("ExampleGame");

      return game;
}
```

`details(GameDetailsResponse a)` works the same way as `description()` above, except it is passed an object of type GameDetailsResponse which it needs to fill with the information we want to be displayed when client asks details about this game.You should check details about which fields you need to fill and which are optional from our protocol description message.xsd when developing your own game. If you forget to fill a mandatory field, don't worry. Our schema parser will tell you where you made a mistake when you first try to run or compile your game.

```
/**
 * Sends details about this game instance to the clients.
 * The example is not very interested in telling things about
 * it self,
 * as there is very little to say. So the bare minimum of
 * information is sent.
 */
@Override
protected void details(GameDetailsResponse a) {
      OptionalStringParameters b = new
OptionalStringParameters();
      Players plrs = new Players();

      a.setColumnNames(b);
      a.setPlayers(plrs);
      a.setRules(b);
}
```

Rest of the methods are quite well covered by the comments in the code and they actually are very game specific and might do whatever you like them to do. So there is no further need to comment these.

```
/**
 * Kills the game program. If a game is deemed as trash (no
 * players playing it for one minute)
 * it is killed [note that when a game is created, it has an
 * extra 1 minute of time when it is
 * not checked for trashyness at all, giving it a minimum of 2
 * minutes life time].
 * This method is called to give the game a chance to make any
 * final deeds
 * that might be necessary for proper client service.
 */
@Override
public void die() {
      // I'm ready to die ANYTIME. As soon as Server deems me
      // as trash,
      // I will die. (About 2 minutes after creation)
}

/**
 * User sends a request. Reacting to those requests should be
 * done in this method.
 * In this example we just check if the client wants to join
 * the game, and if so,
 * send a response that he is welcome to start the example
 * application. The client is not registered as a player,
 * as this example game doesn't really have players. If it
 * did, it would have to keep track
 * of clients in it's own data structures and use that
 *information in game details responses
 * and game list responses.
 */
@Override
protected void handleRequest(long sessionID, GameRequest request) {

      GameResponse msg = new GameResponse();
      GenericResponse gr = new GenericResponse();
      msg.setJoin(gr);

      gr.setSuccess(true);
      gr.setDescription("HelloGame");

      if (request.getJoin() != null)
            send(sessionID, msg);

}


/**
 * If we want to react to clients' responses, this is the
 * place.
 * This example will not cover any specific things one might
 * wish to do in this method, your hands are free. Try things
 * out and build your perfect game.
 */
@Override
protected void handleResponse(long sessionID, GameResponse response)
{
}
```

2. com.asdf.plugins.hello.HelloGame

Starting point is abstract class GameController and we begin by extending it. It provides few useful ready made methods and a bunch of abstract methods you need to implement. You should check the javadoc API documentation of class GameController when starting your own implementation. It gives you a good information about every method you might need.

```
/**
 * An example game framework provided by the ASDF group to get you a
* quick start
 * on creating your own great games in our game server system. Intended
* to function
 * as a first tutorial to working with our server software, not as a
 * tutorial in game programming.
 */

public class HelloGame extends GameController {

   // ConnectionHandlers will make your life a lot easier. See the
   // javadoc for greater understanding.
   ConnectionHandler connection = null;

   // The game window.
   JFrame helloframe = null;

   /**
    * When the game is started, this method is called. Gives you the
    * chance to do what ever
    * initializing you might wish to do.
    */
   @Override
   public void start() {
         helloframe = new JFrame();
         helloframe.setTitle("Hello World!");
         helloframe.setPreferredSize(new Dimension(640,480));
         /*For now we dont need anything else but the title saying
         Hello World!*/
```

handleResponse, handleRequest and handleStateUpdate are your main ways of communicating with the master game running at the server. Server sends you these kinds of messages and you write what you want to do with them in these methods. Also you can send messages to server using the send method in connectionHandler which you can find using getConnection() method.

```
   /**
    * Game logics has sent a response. If you wish to react in some
    * way, write some code here.
    */
   @Override
   public void handleResponse(Response rsp) {

   }
```

```
/**
 * Game logics requests something of you, if you want to do
 * something about it, write some code here.
 */
@Override
public void handleRequest(Request req) {
}

/**
 * Game logics has sent a state update message, informing you of the
 * new state of the game world.
 * If you wish to react in some way, for example store the
 * information somewhere, or process your
 * own game world model, this would be a good time to do so.
 */
@Override
public void handleStateUpdate(StateUpdate stateUp) {
}
```

This method works like a constructor. Parameters which you might need when starting your game client are passed as GameResponse object sent by the game running at server-side.

```
/**
 * When the client is joining the game for the first time, if he
 * wants to something special about it,
 * this is the place.
 */
@Override
public void init(GameResponse arg0, String arg1, AccountType arg2) {
}
```

3. com.asdf.plugins.hello.CreatePanel

Once again we should start with the abstract class that has been given as so kindly by the makers of this server program. This time we actually don't have a choice or our panel will not work in the lobby program. So we begin by extending CreateGamePanel and implementing all the abstract methods. You should really check the javadoc api of CreateGamePanel to get the idea of all the ready made methods you get by extending CreateGamePanel. Also you should check the documentation of LobbyController since all the communication you can do is through it.

```
/**
 * Game creation panel for HelloGame game example. It is recommended to
* create panels that can
 * be used by several games, instead of tailoring a single one for
* each. See pre created panels
 * before creating your own, might save a lot of trouble.
 *
 * Game type is not decided in the game creation panel, but on the
* upper level of hierarchy.
 * This means that the panels can be used by any game, as long as the
* transmitted information
 * suffices for all games that use it.
 */
```

```
public class CreatePanel extends CreateGamePanel{

   private static final long serialVersionUID = -7975533206336492104L;
   private JLabel helloLabel;

   /**
    * Creates a simple game creation panel.
    *
    */
   public CreatePanel() {
         super();
         initGUI();
         setInitialized(true);
   }


   /**
    * Creates a HelloGame creation panel. The name of the classes to
    * launch with game start
    * are transmitted automatically when the game meta files are
    * properly in place.
    *
    * This means that you can use the same game creation panel for
    * different games that ask
    * for similar parameters for creation from the user. For example
    * many poker games can
    * easily share their game creation panel.
    *
    * This game creation panel asks for no information from the user,
    * and doesn't suit for
    * games that have some creation options (for example time per turn,
    * number of players
    * before starting game, required player level, difficulty, game
    * name, etc..). The only thing the panel
    * does, is listen to mouse clicks. When the mouse is clicked on the
    * panel, it sends a
    * game creation request to server.
    */
   public void initGUI() {

         helloLabel = new JLabel();
         this.add(helloLabel);
         helloLabel.setText("Hello World!");
         helloLabel.setPreferredSize(new java.awt.Dimension(218, 178));
         //And other components we want to add to this panel...
```

When we click a mouse in this panel, we send a message to server that we want to create a new HelloGame. Notice that the only way we can get communication to server is through LobbyController which we can find with getController() method. We then request LobbyController that it would create a new game for us. We pass CreateGameRequest as a parameter to that method which contains all the information server needs to create a new HelloGame.

```
helloLabel.addMouseListener(new MouseListener(){
      @Override
      public void mouseClicked(MouseEvent e) {
            CreateGameRequest createReq = new CreateGameRequest();

            String gameName = "Helloworld";
            createReq.setName(gameName);

            Parameters param = new Parameters();
            param.getAny().add(new BasicElement("nimi"));

            createReq.setParameters(param);
            getController().requestCreateGame(createReq);
      }
});
```

4. What else should I know?

You should definitely familiarize yourself with the API documentation of the extension_api.jar package. Especially you should pay attention to the key classes which we went through in the example game. API documentation contains a lot of information about which methods you can use to communicate with the server and what kind of methods are ready made. Especially you should pay attention to our timer mechanism when implementing a new server-side game. Also you might want to spend some time figuring out how our XML message protocol which is defined by message.xsd -schema file actually works. It should be pretty versatile so you actually shouldn't need to do anything else but use the accessors of the message “objects” passed to you or the ones that you create and pass on to be sent. There are “any” and “custom” fields which can contain any kind of payload. You can insert a Java object or even a picture or sound file in it!