Marcelo Madeira

OSGI – Modularizando sua aplicação

novembro 12, 2009 às 11:27 pm | Publicado em Java | 3 Comentários
Semana passada fui conhecer a maior livraria de Vancouver. Como sempre, gastei mais do que devia. Dentre minhas aquisições, não podia faltar um livro sobre OSGI. Como sou um grande defensor de modularização, baixo acoplamento, alta coesão, etc… creio que esta tecnologia é um grande avanço na forma como criamos aplicações utilizando a plataforma Java. Com OSGI, é muito simples criar aplicações altamente extensíveis, veja como exemplo a IDE Eclipse.

Meu objetivo aqui não é mostrar a fundo como funciona a tecnologia e sim demonstrar em um pequeno exemplo algumas de suas vantagens. O exemplo consiste em um sistema de envio de mensagens. O usuário digita uma mensagem em um textfield e esta mensagem pode ser enviada de diversas formas, como email ou sms. Porém, no exemplo, teremos quatro módulos. A interface gráfica, o domínio, o enviador de mensagens por email e o enviador por SMS.

Seguindo a nomenclatura do OSGI, cada módulo é um Bundle. Um Bundle nada mais é do que um "jar" com algumas informações adicionais do MANIFEST.MF. Estas informações são utilizadas pelo framework OSGI. Como quase tudo no Java, a tecnologia OSGI é uma especificação e portanto, temos diversas implementações para escolher. Dentre elas, as mais famosas são Equinox (Projeto Eclipse) e Felix (Apache). Neste artigo utilizaremos o Equinox.

Faça o download no Equinox. Para este artigo precisaremos apenas do Jar. Execute o Jar para ter acesso ao console do Equinox.

java -jar org.eclipse.osgi_3.4.2.R34x_v20080826-1230.jar -console

Para visualizar os Bundles instalados, basta digitar o comando ss.

```
C:\Users\Marcelo\Desktop>java -jar org.eclipse.osgi_3.4.2.R34x_v20080826-1230.ja
r -console

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.2.R34x_v20080826-1230

osgi>
```

Como podemos ver, neste momento só temos um bundle instalado. O bundle do Equinox. Agora iremos criar o nosso bundle e adicioná-lo do Equinox. Criar um bundle é muito simples. Crie um simples projeto com a seguinte classe:

```java
package br.com.marcelo.helloworld;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        System.out.println("Hello World!!");
    }

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        System.out.println("Goodbye World!!");
    }

}
```

Esta classe é o activator do nosso bundle. O activator é utilizado pelo framework OSGI para iniciar ou parar um bundle. Neste primeiro exemplo, o activator irá apenas imprimir mensagens quando for iniciado e parado. Agora precisamos alterar o MANIFEST do jar para torná-lo um bundle OSGI.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Client Plug-in
Bundle-SymbolicName: br.com.marcelo.helloworld
Bundle-Version: 1.0.0
Bundle-Activator: br.com.marcelo.helloworld.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.osgi.framework;version="1.3.0"
```

Veja que no MANIFEST passamos para o OSGI algumas informações do nosso bundle. Dentre elas o nome do bundle (SymbolicName) e qual a classe Activator. Agora vamos instalar este bundle no Equinox. Gere um jar do projeto e para instalá-lo no Equinox é simples:

install file:<nomeDoBundle>.jar

```
osgi> install file:bundle.jar
Bundle id is 1

osgi> _
```

Para verificar se o bundle foi corretamente instalado, basta executar o comando ss:

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.2.R34x_v20080826-1230
1       INSTALLED   br.com.marcelo.helloworld_1.0.0

osgi> _
```

O bundle está corretamente instalado, agora basta iniciá-lo:

start <idDoBundle>

```
osgi> start 1
Hello World!!

osgi>
```

Para parar o bundle:

stop <idDoBundle>

```
osgi> stop 1
Goodbye World!!

osgi>
```

Agora que já sabemos como criar um bundle, vamos iniciar nosso exemplo. No exemplo, teremos quatro bundles.

- Domínio: Como o próprio nome diz, ele armazena as classes de domínio do nosso exemplo. Teremos duas classes: Message e IMessageSender.
- EnviadorSMS: implementação de IMessageSender que envia mensagens por SMS.
- EnviadorEmail: implementação de IMessageSender que envia mensagens por Email.
- UI: interface gráfica do exemplo

Bundle UI

Iremos começar pelo bundle UI. O activator irá apenas criar o frame para o usuário entrar com a mensagem.

```
package br.com.marcelo.helloworld.ui;

import java.awt.BorderLayout;

public class Activator implements BundleActivator {

    private Message message;
    private JFrame frame;

    /*
     * (non-Javadoc)
     *
     * @see
     * org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext
     * )
     */
    public void start(BundleContext context) throws Exception {
        buildInterface();
    }

    /*
     * (non-Javadoc)
     *
     * @see
     * org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        destroyInterface();
    }

    private void destroyInterface() {
        frame.setVisible(false);
    }
    private void buildInterface() {

        frame = new JFrame("Hello");
        frame.setSize(200, 80);
        frame.getContentPane().setLayout(new BorderLayout());
        final JTextField textField = new JTextField();

        JButton button = new JButton("Enviar");
        button.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent pEvent) {
                message.send(textField.getText());
            }

        });

        frame.getContentPane().add(textField, BorderLayout.NORTH);
        frame.getContentPane().add(button, BorderLayout.SOUTH);
        frame.setVisible(true);
    }
```

Veja que o bundle depende de uma classe chamada Message. Esta classe é nosso domínio, portanto, ela não faz parte deste bundle. Aqui entra outro detalhe do OSGI. A comunicação entre bundles é feita através de serviços. Podemos considerar este modelo como sendo um SOA dentro da VM. O bundle UI irá utilizar serviços do bundle Core. Vamos analisar o MANIFEST do bundle UI.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Ui Plug-in
Bundle-SymbolicName: br.com.marcelo.helloworld.ui
Bundle-Version: 1.0.0
Bundle-Activator: br.com.marcelo.helloworld.ui.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: br.com.marcelo.helloworld.core.service,
 javax.swing,
 org.osgi.framework;version="1.3.0",
 org.osgi.util.tracker;version="1.3.3"
```

Veja a declaração Import-Package. Estamos importando um pacote do bundle core. Neste pacote estão os serviços que nosso domínio está disponibilizando. Importamos também o pacote javax.swing.

Agora precisamos criar o serviço.

Bundle Core

O Bundle Core possui duas classes de domínio. A interface dos enviadores e o domínio Message.

Interface:

```
package br.com.marcelo.helloworld.core.service;

public interface IMessageSender {

    void send(String pMessage);

}
```

Domínio:

```
package br.com.marcelo.helloworld.core.service;

import java.util.ArrayList;

public class Message {

    private List<IMessageSender> services = new ArrayList<IMessageSender>();

    public void addService(IMessageSender pMessageSender) {
        services.add(pMessageSender);
    }

    public void removeService(IMessageSender pMessageSender) {
        services.remove(pMessageSender);
    }

    public void send(String pMessage) {

        for (IMessageSender sender : services) {
            sender.send(pMessage);
        }

    }

}
```

Veja que a classe Message é composta por uma lista de services. Estes services são os enviadores de mensagens que serão utilizados. Veja que o método send apenas itera sobre a lista enviando a mensagem. Até aqui tudo é muito simples. Agora precisamos exportar a classe Message como sendo um serviço do bundle core. O módulo UI irá iteragir diretamente com este serviço para enviar as mensagens.

Primeiro precisamos dizer ao OSGI para exportar este pacote para outros bundles. Veja o MANIFEST:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Helloworld Plug-in
Bundle-SymbolicName: br.com.marcelo.helloworld.core
Bundle-Version: 1.0.0
Bundle-Activator: br.com.marcelo.helloworld.core.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.osgi.framework;version="1.3.0",
 org.osgi.util.tracker;version="1.3.3"
Export-Package: br.com.marcelo.helloworld.core.service
```

Veja a informação Export-Package. Para uma classe ser visível para outro bundle, ela precisa estar dentro de um pacote exportado. No nosso caso, o bundle UI precisa da classe Message, portanto, precisamos exportar o pacote onde a classe está. Lembre-se que o bundle UI importou este pacote.

Para registrar o componente Message como um serviço, teremos que interagir diretamente com a API do OSGI. Quando o bundle core for iniciado, iremos registrar o serviço no contexto do OSGI. O código é simples:

```
package br.com.marcelo.helloworld.core;

import org.osgi.framework.BundleActivator;

public class Activator implements BundleActivator {

    public Message messageService = new Message();

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleC
     */
    public void start(BundleContext context) throws Exception {
        context.registerService(Message.class.getName(), messageService, null);
    }

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleCo
     */
    public void stop(BundleContext context) throws Exception {
        messageService = null;
    }

}
```

O método registerService espera como parâmetros o nome do serviço (por recomendação é o nome da classe), o serviço em si e algumas configurações adicionais.

Agora precisamos alterar o bundle UI para utilizar o serviço Message. No activator do bundle UI, basta fazer o "lookup" do serviço utilizando seu nome (nome da classe) :

```
private Message message;
private JFrame frame;
private ServiceTracker serviceTracker;

/*
 * (non-Javadoc)
 *
 * @see
 * org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext
 * )
 */
public void start(BundleContext context) throws Exception {

    serviceTracker = new ServiceTracker(context, Message.class.getName(), null);
    serviceTracker.open();

    message = (Message) serviceTracker.getService();

    buildInterface();
}

/*
 * (non-Javadoc)
 *
 * @see
 * org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
 */
public void stop(BundleContext context) throws Exception {
    destroyInterface();
    serviceTracker.close();
}
```

Se adicionarmos os dois bundles no Equinox, veremos que os dois bundles estão se comunicando. Agora precisamos criar os bundles que realmente enviam as mensagens.

Bundle Enviador Email e SMS

Os serviços de envio por email e por SMS serão novos serviços de nosso sistema. Portanto, deveremos criar um bundle para cada um. Desta forma poderemos controlá-los separadamente. Por exemplo, podemos parar o serviço de envio por SMS e deixar apenas o de Email sem afetar o funcionamento do sistema. O dois bundles possuem praticamente a mesma estrutura, portanto, irei economizar um pouco de linhas aqui.

O bundle enviador terá apenas uma classe que implementa a interface IMessageSender e a classe Activator. Esta interface está no bundle core, portanto, precisaremos importar o pacote da mesma forma que fizemos no bundle UI.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Sms Plug-in
Bundle-SymbolicName: br.com.marcelo.helloworld.sms
Bundle-Version: 1.0.0
Bundle-Activator: br.com.marcelo.helloworld.sms.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: br.com.marcelo.helloworld.core.service,
 org.osgi.framework;version="1.3.0"
```

A classe enviador apenas implementa a nossa interface:

```
package br.com.marcelo.helloworld.sms;

import br.com.marcelo.helloworld.core.service.IMessageSender;

public class EnviadoMensagemSMS implements IMessageSender{

    @Override
    public void send(String pMessage) {
        System.out.println("Sending by SMS: " + pMessage );
    }

}
```

Enviar por SMS será um serviço do nosso sistema. Portanto, precisamos registrá-lo no contexto OSGI:

```
public class Activator implements BundleActivator {

    private IMessageSender service;

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleC
     */
    public void start(BundleContext context) throws Exception {
        service = new EnviadoMensagemSMS();
        context.registerService(IMessageSender.class.getName(), service, null);
    }

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleCo
     */
    public void stop(BundleContext context) throws Exception {
        service = null;
    }

}
```

O bundle de Email é praticamente o mesmo código. A única diferença é a mensagem no System.out.

Veja que registramos o serviço com o nome da interface. Portanto, teremos dois serviços com o mesmo nome. Sempre que pedimos para o contexto o serviço com o nome da interface, ele irá executar uma lógica de prioridade para retornar apenas uma implementação.

Agora que temos dois serviços de envio de mensagem, precisamos alterar nosso bundle core para utilizá-los. Para isso, utilizaremos um ServiceTrackerCustomizer.

ServiceTrackerCustomizer e ServiceTracker

Como vimos, utilizamos o serviceTracker para fazer o lookup de um serviço. Porém, no caso dos enviadores, precisamos saber quando um novo serviço de enviador está disponível ou quando um enviador for removido. Estas informações são importantes para alimentar a lista de serviços dentro do objeto Message.

Para ter acesso a estas informações, usaremos um ServiceTrackerCustomizer. O código é simples:

```
package br.com.marcelo.helloworld.core;

import org.osgi.framework.BundleContext;

public class MessageSenderServiceTracker implements ServiceTrackerCustomizer{

    private final BundleContext context;
    private final Message message;

    public MessageSenderServiceTracker(BundleContext context, Message pMessage) {
        this.context = context;
        message = pMessage;
    }

    @Override
    public Object addingService(ServiceReference pArg0) {

        IMessageSender sender = (IMessageSender) context.getService(pArg0);
        message.addService(sender);

        System.out.println("tracker: " + sender.getClass().getName());

        return sender;
    }

    @Override
    public void modifiedService(ServiceReference pArg0, Object pArg1) {

    }

    @Override
    public void removedService(ServiceReference pArg0, Object pArg1) {
        IMessageSender env = (IMessageSender) context.getService(pArg0);
        message.removeService(env);

    }

}
```

Basta implementar a interface ServiceTrackerCustomizer e codificar o que você deseja quando um serviço for adicionado, modificado ou removido. Simples!!!!

No nosso caso iremos adicionar ou remover o serviço da lista de serviços do nosso objeto Message. Também tem uma mensagem de "log" para nos ajudar com os testes.

Agora precisamos fazer mais uma pequena alteração no activator do bundle core. Precisamos registrar o nosso ServiceTrackerCustomizer como um listener para os serviços do tipo IMessageSender.

```
public class Activator implements BundleActivator {

    public Message messageService = new Message();
    private ServiceTracker messageSenderServiceTracker;

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        context.registerService(Message.class.getName(), messageService, null);

        MessageSenderServiceTracker myServiceTracker =
                new MessageSenderServiceTracker(context, messageService);

        messageSenderServiceTracker = new ServiceTracker(context,
                                    IMessageSender.class.getName(), myServiceTracker);

        messageSenderServiceTracker.open();
    }

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        messageSenderServiceTracker.close();
        messageService = null;
    }

}
```

Utilizamos o serviceTrackerCustomizer junto com o ServiceTracker. Sempre que um serviço for adicionado, modificado ou removido, nosso componente será chamado.

Testando a aplicação

Agora que já codificamos, vamos testar a aplicação.

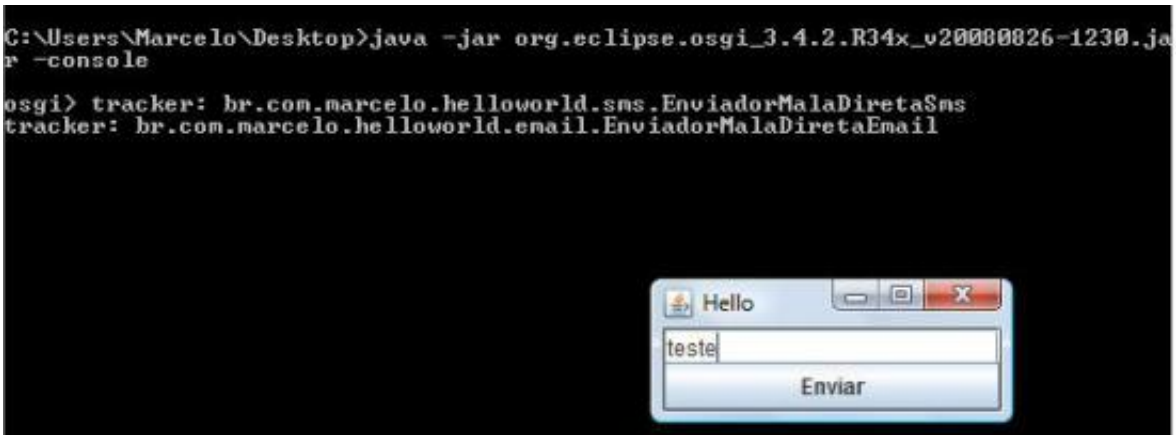
Crie os quatro jars ou faça ou download aqui:

- bundleCore.jar
- blundleUI.jar
- bundleEnviadorEmail.jar
- bundleEnviadorSMS.jar

Instale os quatro bundles no Equinox:

```
Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.2.R34x_v20080826-1230

osgi> install file:bundleUI.jar
Bundle id is 5

osgi> install file:bundleCore.jar
Bundle id is 6

osgi> install file:bundleEnviadorSMS.jar
Bundle id is 7

osgi> install file:bundleEnviadorEmail.jar
Bundle id is 8

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.2.R34x_v20080826-1230
5       INSTALLED   br.com.marcelo.helloworld.ui_1.0.0
6       INSTALLED   br.com.marcelo.helloworld.core_1.0.0
7       INSTALLED   br.com.marcelo.helloworld.sms_1.0.0
8       INSTALLED   br.com.marcelo.helloworld.email_1.0.0

osgi>
```

Inicie os bundles e teste a aplicação.



Veja que as mensagens serão enviadas por email e por SMS. No console do Equinox, pause o serviço de email:

stop <idBundle>

Tente enviar novamente uma mensagem. Como o serviço não está mais disponível, a mensagem foi enviada somente por SMS.

Poder parar módulos da aplicação sem que ela sofra efeitos colaterais é sensacional. Imagine que você descubra um erro crítico no módulo de SMS. Você não precisa tirar toda a aplicação do ar para corrigir este problema. Basta pausar o módulo de SMS. Todo o resto do sistema continuará funcionamento normal. Faça o teste com este pequeno exemplo. Pause e Inicie os serviços. Isto não afetará o Core e muito menos o UI.

Espero ter conseguido explicar um pouco do que é OSGI. Vale ressaltar que tem muito mais detalhes sobre controle de classpath e configuração de bundles que não nos atentamos aqui. Fica como tarefa para quem se interessou dar uma olhada nas outras funcionalidades.

Vale a pena olhar o projeto Spring-DM. O spring facilita muito a configuração de serviços além de propiciar um excelente conteiner de IoC.

3 Comentários »

Feed RSS (Really Simple Syndication) para comentários sobre este post. TrackBack URI (Uniform Resource Identifier)

Em vez de utilizar o (ServiceTrackerCustomizer e ServiceTracker), opine por usar (DS – Declarative Services).

Luís Carlos Moreira da Costa
Eclipse RAP, RCP, eRCP, GMF, OSGI, Spring-DM and Pentaho Developer
Regional Communities/Brazil

http://wiki.eclipse.org/Regional_Communities/Brazil

Comment by Luís Carlos Moreira da Costa— junho 23, 2010 #

Resposta
O que e biosfera?

Comment by Rosiane— outubro 19, 2010 #

Resposta
Qual a diferença entre STC, ST e DS?

Comment by Carlos Alexandro Becker— abril 8, 2011 #

Resposta

Seguir “Marcelo Madeira”