



# OSGi e os benefícios de uma Arquitetura Modular

Entenda a arquitetura deste framework que vem ganhando espaço no mundoj e aprenda a criar componentes modulares com ele.



**Leonardo Fernandes**

(leofernandesmo@gmail.com): é formado em Tecnologia da Informação pelo CEFET-AL, trabalha com Java desde 2005, possui as certificações SCJP e SCWCD, co-fundador do Grupo de Engenharia de Software de Alagoas e do Grupo de Usuários Python de Alagoas, entusiasta e participante das comunidades de Python e Android. Atualmente é bolsista na FAPEAL/UGTI. Mantém um blog em <http://jroller.com/leofernandesmo>.

*Apesar de o conceito de Modularidade não ser novo no mundo da computação, parece que não aprendemos com a história e repetidos erros acontecem. Java não possui um suporte explícito para construção de sistemas modulares e à medida que as aplicações vão crescendo e ficando mais complexas torna-se mais difícil mantê-las. A OSGi™ Service Platform introduz mecanismos que suprem essa carência através de um framework construído sobre a plataforma Java.*

## ❖ O problema da modularidade em Java

A plataforma Java desde a sua criação tem apresentado um grande sucesso tanto no meio acadêmico como no mundo dos negócios. Mesmo com sua frase "Write once, run anywhere", tão criticada por muitos, Java vem sendo usada para desenvolver aplicações que rodam de pequenos dispositivos móveis até massivas soluções distribuídas. Desse modo, tem sido apontado um crescimento contínuo na adoção da plataforma nos últimos 10 anos. Mas infelizmente nem tudo é alegria e à medida que as aplicações foram crescendo um problema já conhecido de grandes projetos de engenharia apareceu: a complexidade. Como exemplo, imaginem um Boeing 747-400 que possui aproximadamente 6 milhões de partes. É humanamente impossível uma única pessoa conhecer completamente esta complexa máquina. O mesmo acontece com grandes sistemas de software hoje em dia, e a melhor maneira de resolver esse problema é quebrá-lo em pedaços menores, ou módulos. Todavia, a plataforma Java não possui, até a versão atual, um suporte adequado para construção de sistemas modulares além dos conceitos provenientes da orientação a objetos (como encapsulamento de dados). Felizmente, Java foi muito bem lapidado com relação a sua flexibilidade, o que permitiu a construção de um poderoso sistema de módulos sobre a sua plataforma: O OSGi.

## ❖ O que é OSGi?

A resposta mais simples para essa pergunta é que OSGi é uma camada

sobre a plataforma Java, para dar-lhe suporte à construção de sistemas modulares. Originalmente o termo significava "Open Service Gateway initiative", mas com o passar dos anos o nome ficou obsoleto já que a ideia inicial de servir apenas como um "Home Gateway" ficou muito restrita comparada com a proposta atual. Quem coordena o andamento da plataforma é a OSGi Alliance, um consórcio mundial de empresas fundado em 1999, o qual vem trabalhando para especificar uma infraestrutura que permita a distribuição e interoperabilidade de aplicações e serviços baseados em componentes, que possam ser gerenciados remotamente e que atuem nos mais diferentes mercados. De sistemas embarcadas para casas (ou automóveis) inteligentes, passando por dispositivos móveis, até aplicações distribuídas. A parte central da especificação é o framework que define um modelo para gerenciar o ciclo de vida da aplicação, o registro de serviços e um ambiente de execução. A aliança provê uma especificação, suíte de testes e certificação com o propósito de garantir um valioso ecossistema entre estes diferentes domínios.

Devido a este grande número de mercados possíveis, o conselho administrativo da OSGi Alliance criou os Working Groups, que são subdivisões responsáveis pelo desenvolvimento técnico da OSGi Service Platform de acordo com a área de atuação do participante. Dentre os Working Groups estão: Core Platform Expert Group (CPEG), Vehicle Expert Group (VEG), Mobile Expert Group (MEG), Enterprise Expert Group (EEG) e Residential Expert Group (REG). Para mais detalhes da área de atuação e acompanhamento do trabalho de cada grupo, visite a página indicada nas referências.



Entre alguns dos membros da aliança estão: Nokia, Motorola, Siemens, Ericsson, Oracle, SAP, SpringSource entre outros (veja o link sobre os Working Groups e seus membros nas referências).

A ideia de modularidade não é nova no mundo da Tecnologia da Informação, é anterior aos anos 70 e entre algumas propriedades de um sistema modular, pode-se citar:

- **Autocontido:** um módulo é um conjunto lógico, ele pode ser incluído, retirado, instalado ou desinstalado como uma unidade autônoma. Normalmente essas partes não podem trabalhar sozinhas, mas são de grande valia para o todo (a aplicação).
- **Alta coesão:** um módulo não deve fazer coisas alheias à sua função, deve-se manter uma finalidade lógica e cumprir bem essa finalidade.
- **Baixo acoplamento:** um módulo não deve se preocupar com a implementação interna de outros módulos que interagem com ele. O baixo acoplamento permite alterar um módulo, sem a necessidade de atualizar os outros.

Para dar um bom suporte às três propriedades, é muito importante que os módulos tenham uma interface bem definida para interação com outros módulos, criando fronteiras lógicas e evitando acesso a detalhes da implementação interna.

## ❖ JARs (tradicionais) não são módulos

A unidade padrão para distribuição de aplicações em Java é o arquivo JAR, como documentado pela especificação. Além disso, eles são muito usados como bibliotecas do sistema. Construir uma aplicação em Java requer a utilização de vários JARs e o JDK sozinho não possui boas ferramentas para gerenciá-los de maneira amigável. Apesar de poderem ser movidos e reutilizados em diversas aplicações e saber que construir sistemas modulares é mais uma questão de processo, arquitetura e disciplina da equipe, seria muito mais fácil se houvesse um suporte para isso. Ferramentas como o Maven e o Ivy possuem características que ajudam neste sentido, principalmente no que diz respeito ao gerenciamento de dependências, mas ambas diferem um pouco da proposta do OSGi, pois não trabalham em tempo de execução.

Grças à falta de informação explícita, desenvolvedores podem, por exemplo, utilizar detalhes da implementação interna de uma biblioteca (acessando qualquer membro declarado como público) e assim quebrar facilmente seu código quando uma mudança nessa implementação acontecer. Outro exemplo acontece quando se utiliza uma biblioteca de

terceiro e esta necessita de outras libs para funcionar. Caso a biblioteca não tenha uma boa documentação informando quais libs são necessárias para sua correta utilização fica muito complicado descobrir suas dependências.

## ❖ Casos de sucesso

Como já citado, são diversos os campos de atuação do OSGi, assim como os casos de sucesso na utilização da plataforma. No campo Automotivo, por exemplo, o BMW Research usa o serviço certificado da ProSyst para integração dos componentes do carro, assim como sua relação com ambientes externos ao veículo em projetos como o Ertico GST. O Eclipse, a partir da versão 3, começou a usar com sucesso o OSGi para sua conhecida plataforma de plugins. Os principais servidores JEE do mercado, como Glassfish, Websphere, Weblogic, Jonas e SpringSource dm Server, também usam o framework em seu core.

## ❖ Implementações

Como a OSGi Alliance define apenas uma especificação. Várias são as implementações do framework disponíveis no mercado. Abaixo são listados os mais populares.

- **Equinox** – Uma das implementações mais utilizadas ganhou notoriedade por gerenciar o famoso sistema de plugins do Eclipse. Presente também em produtos da IBM, como o Lotus Notes e o Websphere Application Server. Está sob licença EPL (Eclipse Public License) e já implementa totalmente a versão 4.1 da especificação.
- **Knopifish** – Bem madura e popular implementação da versão 4.1, talvez devido as excelentes ferramentas para build. É mantida pela empresa sueca Makewave que foi uma das fundadoras da OSGi Alliance. Possui licença BSD para sua versão gratuita, mas possui também uma versão comercial.
- **Felix** – Mantida pela Apache Software Foundation e inicialmente chamada de Oscar, esta implementação é utilizada pelo Glassfish e Jonas. Disponível sob licença Apache 2.0 e implementando a versão 4.1 do OSGi, esta implementação será utilizada para os exemplos deste artigo.
- **Prosynt** – Baseado no Eclipse Equinox com algumas características adicionais, esta implementação é certificada OSGi R4. Apesar de não ser tão popular na comunidade, uma característica marcante é a grande quantidade de clientes, como: BMW, Ford, Bombardier, Cisco, Philips, entre outros.

- **Concierge** – Uma implementação bem compacta e otimizada da versão 3 do OSGi. O que faz dele particularmente apto para plataformas com recursos limitados, como aplicações móveis e embarcadas. Utiliza uma licença BSD.

## ❧ O framework

O framework é o centro da especificação desta plataforma de serviços. Ele fornece um ambiente seguro para distribuição e gerenciamento de componentes, neste contexto chamados de bundles.

Bundles são arquivos JAR simples disponibilizados em um contêiner OSGi, que podem ser remotamente instalados, inicializados, atualizados, finalizados ou desinstalados dinamicamente, em tempo de execução e sem a necessidade de reiniciar todo o sistema. Deve-se imaginar um bundle como uma caixa-preta que pode expor seus pacotes para serem acessados externamente. Uma vez iniciado o componente, seus serviços são disponibilizados a outros bundles, permitindo assim a utilização de suas funcionalidades.

A única diferença entre um bundle JAR e um arquivo JAR tradicional é uma pequena quantidade de metadados adicionado no arquivo META-INF/manifest.mf. Então, caso deseje-se usar um bundle JAR fora de um contêiner OSGi, não haverá nenhum problema.

A especificação define cerca de 20 metadados. Dentre eles, pode-se citar:

- O nome do Bundle. Deve-se colocar um nome simbólico único que é usado pelo contêiner para determinar uma identidade única.
- A versão do Bundle.
- A lista de imports e exports. Que informa ao framework quais classes o bundle precisa e quais classes (interfaces) o bundle deixa externamente visíveis a outros módulos.
- O Activator. Um tipo de classe "main" que é chamada quando o bundle é iniciado ou finalizado.
- Informações como Vendor, indicação de Copyright, Contato etc.

Uma importante vantagem deste tipo de abordagem no processo de desenvolvimento, em relação ao modelo tradicional (monolítico), é uma aproximação maior ao modelo incremental, já que no caso de alteração num determinado bundle, tanto o build como o deploy, podem ser feitos isoladamente do resto da aplicação.

Com relação a uma parte da sua arquitetura, o framework é conceitualmente dividido nas seguintes camadas: (ver figura 1 para a visão geral das camadas)

- **Module Layer** – Define o conceito de módulo do OSGi, como um bundle pode importar e exportar código. Também provê a base para funcionalidade do classload.
- **Lifecycle Layer** – Define como os bundles são dinamicamente instalados e gerenciados no framework. Mais adiante será explicado melhor como funciona o ciclo de vida de um bundle.
- **Service Layer** – Promove um modelo de desenvolvimento flexível que incorpora o conceito de SOA (Service-Oriented Architecture), publish-find-bind. Responsável por conectar os bundles de maneira dinâmica.

Para quem já desenvolve aplicações seguindo um dos mais recomendados princípios da Orientação a Objetos, que é programar para interface e não para implementação, então este modelo de desenvolvimento do OSGi parecerá bem familiar.

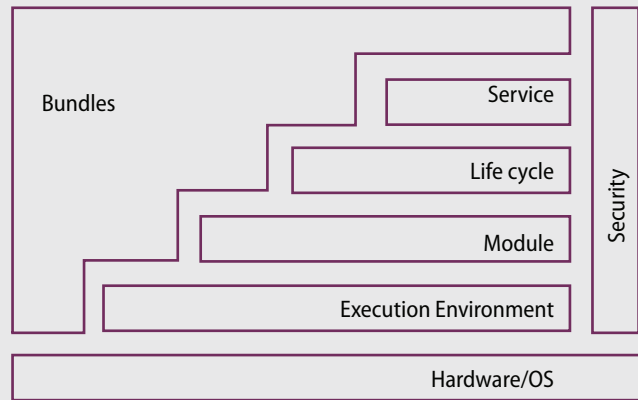


Figura 1. Visão geral das camadas do OSGi.

## Entendendo ciclo de vida do bundle

Como já foi citado, o bundle possui uma camada chamada Lifecycle Layer, a qual provê uma API para controlar a segurança e o ciclo de vida das operações dos bundles. Na figura 2 pode-se ver os estados que um bundle pode passar durante sua vida dentro de um contêiner.

O exemplo que será apresentado neste artigo mostrará como é fácil gerenciar os estados dos bundles através de comandos num prompt. As funções de cada estado são muito bem descritas na especificação da plataforma. Um resumo dos estados é apresentado a seguir:

- **INSTALLED** – O bundle foi instalado com sucesso;
- **RESOLVED** – Avalia se o bundle está pronto para ser iniciado/finalizado validando coisas como a versão do Java, se os imported packages estão disponíveis etc.;
- **STARTING** – Este é um estado de transição, pelo qual passa o ciclo, porém o bundle não pára sobre ele. Neste estado, o método BundleActivator.start() é invocado;
- **ACTIVE** – O bundle foi validado com sucesso e está pronto para ser utilizado;
- **STOPPING** – Outro estado de transição, assim como o STARTING. Neste estado, o método BundleActivator.stop() é chamado;
- **UNINSTALLED** – O bundle é desinstalado e não pode passar para nenhum outro estado.

Quando um bundle é instalado, ele fica armazenado em um meio persistente do framework e continua lá até ser explicitamente desinstalado.

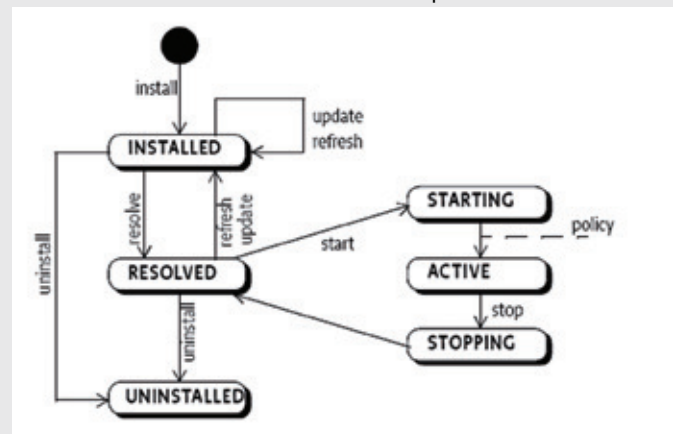


Figura 2. Estados do bundle dentro de um contêiner OSGi.

## Controle operacional

Todo controle do ciclo de vida dos módulos, assim como outras funcionalidades, pode ser feito através de consoles de administração (gráficos ou via linha de comando) que variam de acordo com cada implementação. Através desses consoles, é possível instalar ou desinstalar qualquer bundle dinamicamente sem precisar parar ou reiniciar a aplicação. Ele provê informações dos serviços ativos, mostra os cabeçalhos com os metadados, a lista de imports e exports de cada módulo e outras informações. Alguns desses serviços que o contêiner fornece podem ser feitos programaticamente, dentro do componente sendo criado. Um dos exemplos apresentados neste artigo mostrará como interagir com o framework ativando bundles dinamicamente, bastando colocar o arquivo JAR dentro de uma pasta no sistema.

## Trabalhando orientado a serviços

O modelo de serviços é conhecido pelo processo “publish-find-bind”, que dentro do framework é aplicado na Service Layer. Esta camada define um modelo dinâmico e colaborativo muito integrado com a Life Cycle Layer. Também conhecido como “SOA in a VM”, um serviço, neste contexto, poderia ser caracterizado pelo conjunto:

- a) **Especificação** – uma interface Java onde seriam definidos os métodos públicos;
- b) **Implementação** – uma classe Java que implemente os métodos da interface;
- c) O registro disso no Framework Service Registry para disponibilizá-lo a outros módulos.
- d) **Os clientes** – bundles que consomem o serviço.

Nos exemplos apresentados neste artigo, ficará claro como aplicar o conceito de serviços dentro do OSGi.

## Versionamento

Versionar componente é uma parte muito importante do OSGi. É fato que poucos softwares hoje em dia permanecem inalterados após sua primeira release e com o passar do tempo alterações feitas nas versões novas podem acarretar em incompatibilidade com versões anteriores. Para resolver este problema, o OSGi especifica o conceito de versionamento, em que cada bundle pode ser versionado com um consistente esquema de numeração. Ele usa três segmentos de números (major, minor e micro), mais um segmento alfanumérico (qualifier), no qual qualquer um dos segmentos podem ser omitidos. Ex. “1.2.3.beta\_1” ou “1.2”. Este número de versão pode representar tanto o bundle (atributo Bundle-Version do MANIFEST.MF) como um pacote (atributo Export-Package Ex. Export-Package: org.leofernandesmo.myproject;version=”2.0.0”). Gerenciar esta tarefa manualmente pode ficar inviável, então é altamente recomendado usar softwares de apoio (ver quadro BND Tool ). Agora digamos que nossa aplicação precise utilizar o pacote versionado de um bundle, mas apenas a versão 2.0 é compatível. Então, ao especificar o atributo Import-Package no arquivo MANIFEST, coloca-se o parâmetro da versão desejada. Por exemplo:

```
Import-Package: myapp.utilities;version="[1.0.0,1.5.1]" # Colocar duas versões entre colchetes
```

# separados por vírgula indica um intervalo

# válido para a versão solicitada

Para sistemas que possuem um grande volume releases, um bom mecanismo seria usar a data escrita de traz para frente no segmento de alfanumérico, como, por exemplo: 2.0.1.2009-07-05.

## Os primeiros passos

Para o exemplo mostrado neste artigo, foi escolhida a implementação Apache Felix mantida pela Apache Software Foundation, contudo os códigos apresentados aqui podem rodar em qualquer outra implementação OSGi Release 4 (com exceção de alguns comandos próprios deste contêiner). Foi utilizado o Eclipse 3.4 como IDE para desenvolvimento deste código.

## Instalar o framework

A última versão lançada até a data de escrita deste artigo é a 1.8.0 e pode-se obtê-la no site do projeto (ver referências). Basta fazer download dos binários em zip (ou tar.gz). Após baixar o arquivo, descompacte-o em uma pasta de sua preferência. Para fins de explicação, serão utilizados apenas os binários do projeto, mas caso queira baixar o código-fonte, ele é bastante útil quando usado com a IDE, já que esta permite consultar a API usando o assistente.

Após descompactar o arquivo, vá até a pasta-raiz e digite:

```
java -jar bin\felix.jar
```

Você verá algo parecido com a figura 3.

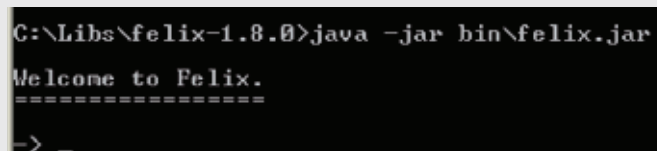


Figura 3. Imagem do prompt do Felix 1.8.0.

A melhor maneira de interagir com o Felix é via Shell. Digitando “help” no prompt, aparecerá uma lista de comandos disponíveis. Um comando particularmente útil e frequentemente usado é o “ps”, que imprime a coleção de bundles instalados e seus estados.

## Configurar o eclipse

Antes de criar qualquer projeto, será definida uma “User Library” para o Felix, o que ajudará a referenciá-lo de qualquer projeto. Para fazer isso, abra o Eclipse e acesse Window->Preferences->Java->Build Path->User Libraries, como mostra a figura 4. Clique no botão “New”. Coloque o nome “Felix” e depois adicione o arquivo bin/Felix.jar que está no diretório onde foi descompactado o framework. Caso tenha baixado o código-fonte, selecione “Source attachment”, clique no botão Edit e depois selecione o arquivo/pasta com o código-fonte.

## Criar um projeto

Crie um novo projeto Java no Eclipse com o nome “myFirstBundle”. Antes de finalizar o assistente, clique na aba “Libraries” e no botão “Add Libra-



ry”, selecione a opção “User Library” e marque o checkbox da biblioteca “Felix” que foi criada no item anterior.

Crie pacotes com seu domínio (para os exemplos deste artigo foi definido o pacote “org.leofernandesmo”) e uma classe Java com nome “MyActivator”, implementando a interface BundleActivator. Em seguida, escreva os métodos herdados como no código da Listagem 1. Esta interface é um tipo de main para o contêiner. Como foi explicado anteriormente, todo bundle precisa de um arquivo MANIFEST.MF, responsável por prover informações sobre ele ao contêiner. A Listagem 2 mostra um exemplo válido deste arquivo (os comentários mostrados são de caráter explicativo e não fazem parte do código).

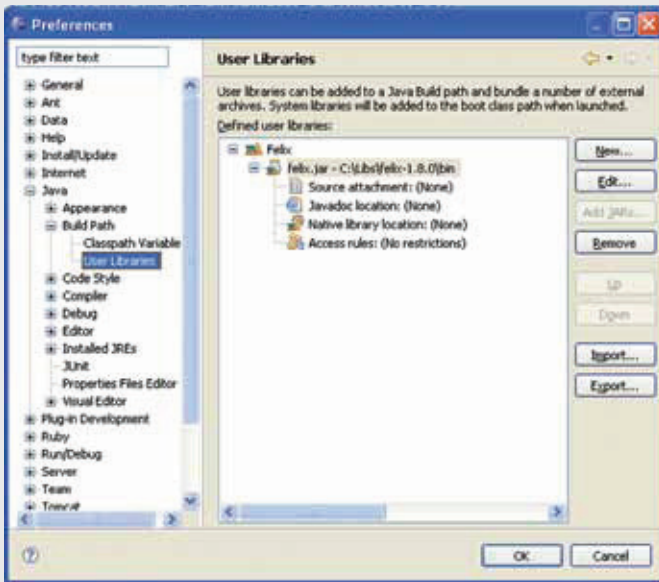


Figura 4. Configurando uma User Library no Eclipse para ser utilizado em vários projetos.

#### Listagem 1. Implementação do MyActivator.

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

public class MyActivator implements BundleActivator {

    public void start(BundleContext context) throws Exception {
        System.out.println("Iniciou!!");
    }

    public void stop(BundleContext context) throws Exception {
        System.out.println("Finalizou!!");
    }
}
```

#### Listagem 2. Implementação do arquivo MANIFEST.MF.

```
Manifest-Version: 1
Bundle-Name: MyActivator
Bundle-SymbolicName: org.leofernandesmo.MyFirstBundle #nome do bundle
deve ser único
Bundle-Version: 1.0.0
Bundle-Activator: org.leofernandesmo.MyActivator #classe "main" que
implementa BundleActivator
Import-Package: org.osgi.framework #pacotes utilizados por este exemplo
```

Um atributo importante do arquivo MANIFEST é o “Import-Package”. Muita gente acha que todos os pacotes existentes no Java Runtime Environment (JRE) não precisam ser importados, o que é um erro. Apenas pacotes abaixo de “java.\*” são automaticamente importados. Classes da API do Swing, por exemplo, fazem parte dos pacotes de extensão (javax.swing.\*) e precisam ser importadas. Isso foi feito para habilitar bundles a rodarem sobre implementações diferentes do JRE que possuam sua própria estrutura de pacotes. Agora basta empacotar o componente num arquivo JAR. Voltando ao prompt do Felix para instalar o JAR criado, digite o comando “install file:/<caminho do arquivo jar>/arquivo.jar”. Vai aparecer um código (ID) que será usado para controlar o ciclo de vida do bundle. Caso queira visualizar os headers do arquivo MANIFEST, digite “headers <ID>” e a lista de atributos será impressa na tela. O próximo passo é iniciar o serviço. Digite “start” seguido do ID do seu bundle e pronto, você verá a frase “Iniciou!!” presente no método “start()” do MyActivator.

Seu primeiro componente está pronto, apesar de não fazer nada demais.

### Interagir com o framework

Dando uma olhada no exemplo anterior, pode-se notar que quando o framework chama os métodos start e stop do Activator, um objeto do tipo BundleContext é passado para ele. Este objeto é o principal canal de comunicação caso a aplicação desenvolvida deseje interagir com o framework. E entre alguns serviços dele estão:

- obter uma lista de todos bundles instalados;
- instalar novos bundles programaticamente (e automaticamente);
- registrar listeners que informam quando o estado de qualquer bundle, serviço, ou mesmo o próprio framework, é alterado.

Agora imagine que uma grande equipe desenvolve vários bundles, e instalar esses componentes da maneira apresentada anteriormente pode levar muito tempo. Como todo programador que se preze é louco por automação, para solucionar isso vamos escrever um programa que periodicamente verifique o conteúdo de um diretório no filesystem e sempre que um novo arquivo com extensão “jar” aparecer, ele deve instalá-lo como um bundle OSGi. Tendo cuidado para não instalar módulos que já estejam presentes no framework.

Para implementar este programa, pode-se utilizar o projeto já criado no exemplo anterior. Vamos criar uma nova classe dentro do mesmo pacote do Activator chamada BundleManager. Esta classe será responsável por instalar e iniciar os bundles que forem colocados num diretório. A Listagem 3 apresenta como este arquivo ficará.



Listagem 3. Classe BundleManager. Responsável por instalar todos os arquivos JAR colocados num diretório do filesystem.

```
public class BundleManager implements Runnable {
    //Diretorio que guardará todos os bundles criados pela equipe.
    private static final String BUNDLE_FOLDER = "C:/MyBundles";
    //Intervalo de 5 segundos para a Thread verificar se há algum módulo novo.
    private static final long INTERVAL = 5000;
    //O contexto é o canal de comunicação entre a aplicação e o framework
    private BundleContext context;

    public BundleManager(BundleContext _context) {
        super();
        this.context = _context;
    }
    //Método para iniciar a Thread que gerencia os bundles.
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                Thread.sleep(INTERVAL);
                //Busca todos os bundles novos..
                Set<String> bundles = findNewBundles();
                for (String path : bundles) {
                    installBundle(path); //instala cada bundle passando seu caminho absoluto.
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BundleException e) {
            e.printStackTrace();
        }
    }
    //Método que instala um bundle de acordo com o caminho do arquivo JAR
    private void installBundle(String path) throws BundleException {
        //Usando o contexto para instalar um arquivo.
        Bundle bundle = this.context.installBundle("file:" + path);
        System.out.println("New Bundle: " + bundle.getId() + "->" +
            bundle.getSymbolicName());
        //Iniciando o Bundle...
        bundle.start();
    }
    //Método que compara os arquivos do diretório com os bundles presentes
    // no framework.
    //listando apenas bundles novos.
    private Set<String> findNewBundles() {
        Set<String> newBundles = new HashSet<String>();
        for (String file : getJarFiles()) {
            boolean bundleExist = false;
            for (String name : getBundles()) {
                if (file.equalsIgnoreCase(name)) {
                    bundleExist = true;
                    continue;
                }
            }
            if (!bundleExist) {
                newBundles.add(BUNDLE_FOLDER + "/" + file);
            }
        }
        return newBundles;
    }
    //Método que retorna todos os arquivos JAR numa determinada pasta
    private List<String> getJarFiles() {
        List<String> files = new ArrayList<String>();
        File folder = new File(BUNDLE_FOLDER);

        for (File file : folder.listFiles()) {
            if (file.getName().endsWith(".jar")) {
                files.add(file.getName());
            }
        }
    }
}
```

```
    }
}
return files;
}
//Método que retorna os bundles presentes no framework
private List<String> getBundles() {
    List<String> bundles = new ArrayList<String>();
    for (Bundle bundle : this.context.getBundles()) {
        if (bundle.getLocation().contains("/")) {
            String name = bundle.getLocation()
                .substring(bundle.getLocation()
                    .lastIndexOf("/") + 1);
            bundles.add(name);
        }
    }
    return bundles;
}
}
```

Agora, deve-se alterar a classe MyActivator para finalizar este exemplo. A alteração é feita apenas nos métodos start e stop, que têm como objetivo inicializar e interromper a Thread do BundleManager.

```
private Thread manager;
public void start(BundleContext context) throws Exception {
    this.manager = new Thread(new BundleManager(context));
    this.manager.start();
}
public void stop(BundleContext context) throws Exception {
    this.manager.interrupt();
}
```

O OSGi define um repositório que guarda uma referência do local onde o módulo foi originalmente instalado no filesystem. Então, após empacotar a aplicação em um arquivo JAR, basta digitar o comando "update <ID>", colocando o mesmo ID gerado anteriormente, e o Felix irá "reinstalar" a partir desta referência. O exemplo anterior foi desenvolvido para ser usado apenas em contêineres OSGi, mas uma recomendação interessante para aqueles que pretendem projetar módulos é isolar o uso da service layer API no menor número de classes possíveis, diminuindo assim o acoplamento com o framework. Fazer isso facilita a migração do componente para um ambiente não-OSGi.

## Criar um serviço

No terceiro exemplo do artigo, será apresentado como criar um serviço através de um bundle.

A ideia é criar um serviço que faça uma cópia profunda de qualquer objeto passado como parâmetro e retorne esta cópia para o cliente que o chamou.

Crie um novo projeto seguindo o exemplo anterior, defina dois pacotes dentro do projeto, um para armazenar sua interface (API) e outro que guardará a implementação (para os exemplos deste artigo foram definidos os pacotes "org.leofernandesmo.copyObjectService.api" para interface e "org.leofernandesmo.copyObjectService" para implementação). Definir pacotes diferentes para a API e implementação é considerado uma boa prática, porque evita que pessoas desinformadas acessem deta-

lhes da sua implementação. O primeiro passo quando se decide criar um serviço é projetar bem API, lembre-se que essa será sua fronteira pública para que outros componentes (bundles ou não) acessem seu serviço. Crie uma interface chamada “CopyObject” com um método “createCopyOf”, como na Listagem 4, e uma classe que implemente esta interface chamada “CopyObjectImpl”, como na Listagem 5.

Listagem 4. Interface CopyObject. Apenas esta interface ficará disponível para o cliente deste serviço.

```
public interface CopyObject {
    public ObjectInputStream createCopyOf(Object object);
}
```

Listagem 5. Classe CopyObjectImpl, que implementa a interface CopyObject fornecendo uma cópia profunda de qualquer objeto passado como parâmetro.

```
public class CopyObjectImpl implements CopyObject{
    public ObjectInputStream createCopyOf(Object obj){
        ObjectOutputStream output = null;
        ObjectInputStream input = null;
        try{
            ByteArrayOutputStream byteOutput = new ByteArrayOutputStream();
            output = new ObjectOutputStream(byteOutput);
            output.writeObject(obj);
            output.flush();
            byte[] byteArray = byteOutput.toByteArray();
            ByteArrayInputStream byteInput = new ByteArrayInputStream(byteArray);
            input = new ObjectInputStream(byteInput);
            return input;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        } finally {
            try{
                output.close();
                input.close();
            } catch (Exception e) {
                e.printStackTrace();
                return null;
            }
        }
    }
}
```

## Saber mais

Entenda como funciona uma cópia profunda de um Objeto na Sessão Jogo Rápido da Revista Mundoj Nº. 23.

Em OSGi, serviços são publicados em um service registry dentro do contêiner e são identificados pelas interfaces que possuem. Para registrar o nosso serviço, criamos uma classe a qual chamaremos de “CopyObjectPublisher”, como na Listagem 6, e colocaremos no pacote junto com “CopyObjectImpl”.

Esta classe é um bundle activator, similar à que foi criada no exemplo anterior. Este Activator possui outra função: ele utiliza o “BundleContext” passado como parâmetro pelo framework no método start() para regis-

Listagem 6. Activator responsável por iniciar o bundle, que também faz o papel de Publisher para registrar o serviço.

```
public class CopyObjectPublisher implements BundleActivator{
    //Responsável por gerenciar o serviço criado.
    private ServiceRegistration registry;

    public void start(BundleContext context) throws Exception {
        System.out.println("Iniciou Servidor!!!");
        //Registrando o serviço através do objeto context passado como parâmetro.
        //Utiliza o nome da própria classe para facilitar depois busca pelo cliente e
        //uma instância da implementação do serviço.
        this.registry = context.registerService(CopyObject.class.getName(),
            new CopyObjectImpl(), null);
    }

    public void stop(BundleContext context) throws Exception {
        System.out.println("Parou Servidor!!!");
        //Tirando o registro do serviço do service registry.
        this.registry.unregister();
    }
}
```

trar o serviço que criamos e colocar seu retorno num atributo do tipo “ServiceRegistration”. Essa variável é importante para cancelar o registro do serviço quando o bundle finalizar. A última coisa que se precisa criar é o arquivo MANIFEST.MF, que não terá muita diferença do exemplo anterior. Como pode ser visto na Listagem 7.

Listagem 7. Implementação do arquivo MANIFEST.MF. Os nomes dos pacotes podem variar de acordo com sua organização.

```
Manifest-Version: 1
Bundle-Name: CopyObjectService
Bundle-SymbolicName: org.leofernandesmo.CopyObjectService
Bundle-Version: 1.0.0
Bundle-Activator: org.leofernandesmo.copyObjectService.
CopyObjectPublisher
Import-Package: org.osgi.framework
Export-Package: org.leofernandesmo.copyObjectService.api
```

Se fosse possível representar o processo de Exportar e Importar pacotes em uma imagem, a figura 5 definiria bem isso.

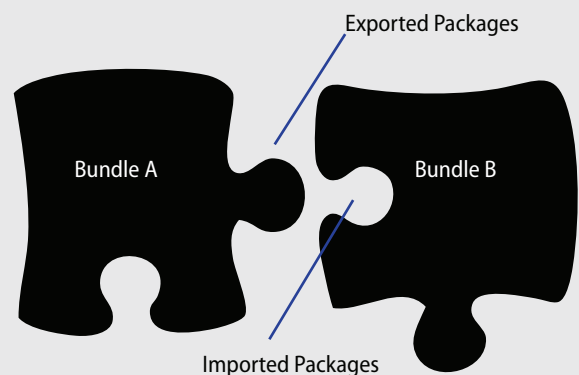


Figura 5. Representação do processo de importar e exportar pacotes do OSGi.

Agora o serviço está pronto para ser compilado e empacotado dentro de um arquivo JAR. Assim como no exemplo anterior, para instalar o componente no contêiner basta digitar o comando "install file:" seguido do comando "start". Quando o service bundle é iniciado, o Felix irá invocar o método start() do CopyObjectPublisher (Activator) e consequentemente publicar o serviço no service registry. Para comprovar que o serviço está publicado, digite o comando "services <ID>" no console do Felix. Similar à figura 6.

```
-> services 66
CopyObjectService (66) provides:
objectClass = org.leofernandesmo.copyObjectService.api.CopyObject
service.id = 26
->
```

Figura 6. Imagem do prompt mostrando os serviços disponíveis no bundle com comando "services" do Felix.

Para consumir o serviço, vamos criar outro projeto, com um nome de sua preferência e criar uma classe chamada Funcionario para que o serviço faça uma cópia do seu estado. Esta classe possuirá apenas três atributos para facilitar a demonstração. A Listagem 8 mostra como ficará esta classe. Será necessária a criação de um Activator como exigência do framework, o mesmo será aproveitado para ter código que localizará o serviço no service registry e fará o bind para utilização pelo cliente do serviço. A Listagem 9 apresenta esta classe. Por último, a criação do arquivo MANIFEST.MF, que pouco diferencia dos apresentados até aqui. A única diferença seria no atributo import-package, que inclui o pacote da interface do serviço criado. A Listagem 10 mostra como o arquivo parecerá.

Listagem 8. Classe Funcionário, utilizada para testar o serviço CopyObjectService.

```
public class Funcionario implements Serializable {
    private static final long serialVersionUID = 1L;
    private int id;
    private String rg;
    private String nome;

    public Funcionario(int id, String rg, String nome) {
        super();
        this.id = id;
        this.rg = rg;
        this.nome = nome;
    }

    public void imprimeDados() {
        System.out.println("ID: " + this.id);
        System.out.println("RG: " + this.rg);
        System.out.println("Nome: " + this.nome);
    }
}

//Getters e Setters suprimidos...
```

Como já foi mostrado, é necessário empacotar o componente em um arquivo JAR e usar os comandos "install file:" e "start" do framework para que o cliente fique ativo e utilize o serviço. Se tudo estiver correto, um resultado similar à figura 7 aparecerá no console do Felix.

Listagem 9. Classe Activator, responsável por localizar o serviço para ser utilizado pelo componente.

```
public class Activator implements BundleActivator {
    public void start(BundleContext context) throws Exception {
        //Busca o service registrado..
        CopyObject copy = getCopyObjectService(context);

        //Cria uma entidade funcionario, que sera copiada pelo serviço.
        Funcionario original = new Funcionario(1, "001001001-01", "Jose da Silva");
        System.out.println("Dados do Funcionario Original: ");
        original.imprimeDados(); //imprime os dados do objeto Funcionario original

        //Utiliza o método da interface para fazer uma cópia do objeto passado no
        //parâmetro.
        ObjectInputStream input = copy.createCopyOf(original);
        Funcionario copia = (Funcionario)input.readObject();
        System.out.println("Dados do Funcionario Copiado: ");
        copia.imprimeDados(); //imprime os dados do objeto Funcionario copiado..
    }

    public void stop(BundleContext context) throws Exception {
        //---
    }

    private CopyObject getCopyObjectService(BundleContext context) {
        //Utiliza o BundleContext para buscar uma referência do serviço através do
        //nome registrado no Service Registry.
        ServiceReference ref =
            context.getServiceReference(CopyObject.class.getName());
        //Utiliza a referência para pegar o serviço e fazer cast para a interface.
        CopyObject cp = (CopyObject)context.getService(ref);
        return cp;
    }
}
```

Listagem 10. Implementação do arquivo MANIFEST.MF do cliente do serviço. Os nomes dos pacotes e das classes podem variar de acordo com sua organização.

```
Manifest-Version: 1.0
Bundle-Name: CopyObjectClient
Bundle-SymbolicName: org.leofernandesmo.CopyObjectClient
Bundle-Version: 1.0.0
Import-Package: org.leofernandesmo.copyObjectService.api, org.osgi.
framework
Bundle-Activator: org.leofernandesmo.copyObjectClient.Activator
```

```
Dados do Funcionario Original:
ID: 1
RG: 001001001-01
Nome: Jose da Silva
Dados do Funcionario Copiado:
ID: 1
RG: 001001001-01
Nome: Jose da Silva
```

Figura 7. Resultado da utilização do CopyObjectService pelo bundle cliente do serviço.



Após este exemplo, é comum imaginar: “E se o módulo cliente tivesse sido disponibilizado primeiro?”. Nunca deve-se assumir que o serviço pode ser obtido durante a inicialização (isso foi feito apenas como exemplo). Bundles podem iniciar em diferentes ordens e é muito difícil conseguir controle sobre isto. A especificação define os objetos ServiceTracker e ServiceEvents, para monitorar e dar ao desenvolvedor a possibilidade de escolher o que fazer caso o serviço esteja disponível ou não. Para observar um exemplo de utilização do ServiceTracker, veja página citada nas referências.

Alguns erros comuns ocorrem devido à geração manual do MANIFEST.MF e do arquivo JAR. Existem ferramentas que automatizam esse processo. Uma das mais populares é apresentada no quadro BND Tool.

## OSGi e o Projeto Jigsaw

Uma das grandes reclamações da comunidade Java é a falta de abertura do JCP (Java Community Process) para uma maior adesão aos produtos desenvolvidos pela comunidade e com OSGi não é diferente. Outros esforços têm sido feitos em direção à modularização da plataforma Java. As seguintes JSRs (Java Specification Request) foram criadas nesta tentativa:

- **JSR 277** – Também conhecida como “Java™ Module System”, possui uma proposta parecida com o que OSGi chega a citar na sua documentação, porém afirma claramente que as especificações dele são inadequadas para resolver este problema;
- **JSR 291** – “Dynamic Component Support for Java™ SE” foi proposta pela IBM em 2006, esta JSR define um modelo de componentes dinâmicos para o Java SE, assim como foi feito para o Java ME pela JSR 232. Basicamente segue o modelo apresentado pela lifecycle layer do OSGi. Desta vez, houve participação de Peter Kriens (OSGi co-fundador e atual diretor);
- **JSR 294** – “Improved Modularity Support in Java™ Programming Language”. A ideia é estender a linguagem com novos construtores (chamados de superpackages) que permita uma organização hierárquica mais modular. Mudanças na JVM serão necessárias para atender a essa especificação.

Atualmente (até a data de escrita do artigo), a JSR 277 está inativa, a 291 está finalizada e a 294 está em progresso. Mas é importante não se confundir com as propostas. A JSR 294 não define um sistema de módulos, ela apenas especifica mudanças na linguagem e na máquina virtual para que eles sejam construídos sobre esta JSR. E hoje existem dois sistemas de módulos para a plataforma Java – Jigsaw e OSGi.

Jigsaw é um projeto do Open JDK que possui duas intenções: modularizar o JDK internamente e permitir aos desenvolvedores modularizar suas aplicações (assim como o OSGi). Mark Reynolds, engenheiro sênior do projeto, caracteriza o JDK atual como massivo e monolítico. Quanto maior fica sua implementação, mais complexo para manter. Sendo assim, a melhor solução seria dividi-la em conjuntos separados, bem especificados e interdependentes, ou seja, em módulos. Apesar de os proponentes atuais do projeto (Mark Reynolds e Alex Buckley) planejarem trabalhar diretamente com a OSGi Alliance, não há garantia da interoperabilidade entre eles. Este tipo de problema notadamente mais político do que tecnológico é ruim para a comunidade.

## BND Tool




Também conhecida como BuNDle Tool, esta excelente ferramenta desenvolvida por Peter Kriens (fundador e diretor da OSGi Alliance) para construção de bundles OSGi utiliza um arquivo descritor (.bnd) que analisa as classes Java disponíveis e seus imports para gerar os arquivos MANIFEST e JAR. O interessante é que BND Toll pode ser usada via linha de comando, através do Ant, com tarefas dedicadas ou integradas ao Eclipse como um plugin. Para mais informação, veja a página do projeto nas referências.

## Considerações finais

De aplicações embarcadas a sistemas distribuídos todos requerem, ou pelo menos se beneficiariam da modularidade e o framework OSGi, que inicialmente foi pensado para atuar como gateway residencial, agora encontrou uma grande audiência e tem sido usado para distribuição de aplicações em quase todos ambientes que consistem de múltiplos componentes trabalhando juntos. O fácil gerenciamento remoto, o ambiente bem definido e a boa adoção da comunidade (principalmente pelos App Servers) faz deste framework um interessante candidato para resolver diversos problemas de deployment da plataforma Java.

Este artigo mostrou a importância de a plataforma Java possuir um melhor suporte a modularidade, e apresentou OSGi™ Service Platform como uma ótima alternativa.

A intenção deste artigo foi servir apenas como uma introdução ao OSGi. Então, caso o leitor tenha se interessado pelo assunto, recomendo começar pela página da OSGi Alliance (ver nas referências). Já existem também bons livros sobre ele e diversos artigos e posts na internet. 

### Referências

#### Sites:

- [1] - [http://www.boeing.com/commercial/747family/pf/pf\\_facts.html](http://www.boeing.com/commercial/747family/pf/pf_facts.html) – Informações sobre um Boeing 747-400.
- [2] - <http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.htm> – Especificação do Arquivo JAR.
- [3] - <http://www.osgi.org/WG/HomePage> – Página da OSGi Alliance.
- [4] - <http://www.osgi.org/About/Members> – Expert Groups.
- [5] - [http://www.ertico.com/en/activities/safety/gst\\_website.htm](http://www.ertico.com/en/activities/safety/gst_website.htm) – Projeto Ertico GST da BMW.
- [6] - <http://felix.apache.org/site/index.html> – Página do Felix Framework.
- [7] - <http://www.aqute.biz/Code/Bnd> – Página do aplicativo BND.
- [8] - <http://en.wikipedia.org/wiki/OSGi> – Wikipedia definição.
- [9] - <http://felix.apache.org/site/apache-felix-tutorial-example-5.html> – Utilização do ServiceTracker para monitorar serviços disponíveis.

#### Livros:

- [10] - Neil Bartlett. *OSGi in Practice*. Disponível para download sob licença Creative Commons.
- [11] - Hall, R. S., Pauls, K., McCulloch, S., *OSGi In Action*. Manning Publications, 2008.

