FROGi: Fractal Components Deployment over OSGi

Mikael Desertot[1,3], Humberto Cervantes[2], and Didier Donsez[1]

[1] Laboratoire LSR-IMAG, 220 rue de la Chimie,
Domaine Universitaire, BP 53, 38041, Grenoble, Cedex 9, France
{mikael.desertot, didier.donsez}@imag.fr
[2] Universidad Autonoma Metropolitana-Iztapalapa (UAM-I),
San Rafael Atlixco N 186, Col. Vicentina, C.P. 09340, Iztapalapa. D.F., Mexico
hcm@xanum.uam.mx
[3] Bull SAS,
1 Rue de Provence, 38130, Echirolles, France

Abstract. This paper presents FROGi, a proposal to support continuous deployment activities inside Fractal, a hierarchical component model. FROGi is implemented on top of the OSGi platform. Motivation for this work is twofold. On one hand FROGi provides an extensible component model to OSGi developers and eases bundle providing. FROGi-based bundles are still compatible with legacy OSGi bundles that offer third party services. On the other hand, FROGi benefits from the deployment infrastructure provided by OSGi which simplifies conditioning and packaging of Fractal components. With FROGi, it is possible to automate the assembly of a Fractal component application. Partial or complete deployment is also supported as well as performing continuous deployment and update activities.

1 Introduction

Component-based software engineering (CBSE) is a development methodology that promotes the idea that software can be built through the assembly of reusable software units called components [14]. Components are characterized by the fact that they explicitly define a set of provided functionalities along with dependencies that allow the components to be assembled (i.e. composed). CBSE assumes that component development and component assembly are clearly differentiated activities. Moreover these activities can be performed by different actors. This differentiation implies that delivery and deployment aspects must be taken into account early in the development life-cycle. To support these activities, components are typically packaged in a unit which includes everything that is needed by the component to function, except whatever the component declares as an explicit dependency. Dependencies can be fulfilled either through composition or at deployment time. A component model is also associated to an execution environment which is responsible for controlling several aspects associated to the components at run-time. These aspects include life-cycle

W. Löwe and M. Südholt (Eds.): SC 2006, LNCS 4089, pp. 275–290, 2006.

management and the support of non-functional requirements such as persistence or security.

Currently, many component models exist; the majority of them are targeted toward specific application domains such as the construction of user interfaces or the construction of server-side applications (for example, the Corba Component Model (CCM) and Enterprise Java Beans (EJB)). The Fractal component model, however, aims to be a general-purpose model and to address a wide spectrum of domains [3]. The Fractal specification defines the component model characteristics, and different implementations for this specification exist. One of them is Julia, which is the reference Java-based implementation (`http://fractal.objectweb.org`). An important particularity of the Fractal component model is that it supports hierarchical composition, where a composition itself can be seen as a component that can be used in other compositions. Another particularity of this model is that it is extensible; this characteristic allows this model to be independent from a particular application domain. Although the Fractal specification defines clearly the characteristics of Fractal components, it does not cover deployment aspects which, as previously mentioned, need to be taken into account early in the development lifecycle. This paper presents FROGi [6] which is an extension of the Fractal component model that supports deployment features and dynamic service-orientation. FROGi introduces the concept of a deployment unit which is not covered in the original Fractal specification. Furthermore, FROGi deployment units address the problem of deployment at both the component and the composition level, necessary to support Fractal's hierarchical model. FROGi also addresses the issue of supporting continuous deployment activities, which represent the fact that deployment activities, which include installation, activation, update and un-installation of components occur continually. Supporting continuous deployment is facilitated by introducing concepts from Service Orientation [2,7] into the component model.

FROGi implements these concepts by combining the Julia reference implementation of the Fractal component Model and the OSGi services platform (`http://www.osgi.org`). FROGi simplifies Fractal-based application deployment and also allows these applications to support continuous deployment activities. This paper describes the concepts and the implementation of FROGi and discusses some issues related to its realization. It is structured in the following way. Section 2 presents the Fractal component model and its reference implementation Julia. Section 3 presents FROGi concepts. It describes how a Fractal application is delivered as a set of deployment units. Section 4 discusses implementation details, including OSGi. Section 5 presents related work and finally section 6 provides a conclusion and gives some perspectives to this work.

2 The Fractal Component Model

This section discusses the principles behind the Fractal component model and its reference implementation Julia.

2.1 Fractal

The Fractal component model is intended as a general-purpose component model. Fractal components are defined as entities that provide and require functional interfaces and that can be composed hierarchically. Fractal components can also provide or require various named instances of an interface of a same type (similar to CCM's facets). To support multiple application domains, Fractal components allow an undefined number of control interfaces to be implemented by the components. Control interfaces are used at run-time for various purposes. The Fractal specification defines several control interfaces which cover aspects such as life-cycle control (LifeCycleController LC), the management of connections between components (BindingController BC), the configuration of the component attributes (AttributeController AC) and the management of composite contents (ContentController CC). Furthermore, different instances of a Fractal component can be created from a factory associated with a particular component type. Figure 1 illustrates an example of a composite component containing two component instances. These instances, which represent a client and a server, are bound together and an interface provided by the client instance is exported outside the composition. Additionally, the two instances and the composite provide several control interfaces.

The Fractal specification defines a standard API that allows component types to be defined programmatically. The API also allows component instances to be created, configured and connected.

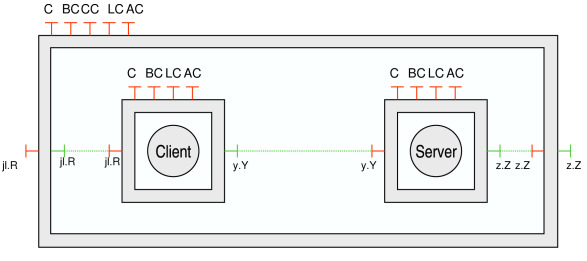

Fig. 1. Graphical representation of a Fractal composite

2.2 Julia

Julia (`http://fractal.objectweb.org`) is the Java-based reference implementation of the Fractal framework which implements the Fractal API. Julia aims to simplify the construction of Fractal applications through the generation of support classes, which allow standard Java classes to adhere to the Fractal component model. A developer using Julia who wishes to create a Fractal component must only provide code associated to application logic (the code that implements the functional interfaces or *component implementation*). Julia generates a set of classes which include implementations of control interfaces as well as interceptors between functional interfaces and the component implementation. Support

classes are generated either in a static or in a dynamic way through mixin and byte code injection techniques. It must be noted that Julia is not the only Fractal implementation; other implementations of Fractal are also available for other languages and frameworks such as C, C#, Smalltalk, JavaScript, etc.

2.3 Construction of Fractal Applications Using Julia

A Fractal application is typically built from a set of classes implementing the application logic contained in the components, one or more coordination classes, as well as a primary class (bootstrap) responsible for performing the application startup. Coordination classes interact with the Fractal framework to create the different component types, component instances and instance connections required by the application. Coordination logic can be written either programmatically or declaratively using the Fractal Architecture Description Language (ADL). It must be noted that the ADL only allows static compositions between component instances to be described; as a consequence, dynamic changes must be programmed explicitly in the application code.

3 FROGi

As previously described, the Fractal component model intends to be general and allow many types of applications to be constructed, either distributed or not. Construction of applications using this model is beneficial for several reasons. First, Fractal is an extensible model; it allows the developer to extend it by providing additional control interfaces and by extending its ADL as well. Fractal is also flexible since it permits to dynamically adapt the binding configuration between the components (although this has to be done programmatically through the API). Finally its hierarchical model provides a way to build coarse components by composing finer components. Fractal enables also the management of the non-functional aspects of components. Despite these advantages, Fractal still has some limitations. The first concerns component deployment since nothing is specified in Fractal regarding this aspect. Although this issue has been addressed by some recent work, proposed solutions are limited because they do not support component unloading when components are not used anymore (see related work section). The second limitation concerns component packaging. As deployment is not currently addressed, no deployment unit has been specified. As a result, a Fractal application is delivered a set of classes. Although these classes can be packaged together in a JAR file (Java ARchive), the components themselves cannot be delivered independently. The last limitation concerns the versioning of the components constituting the application. It is not currently possible to support multiple versions of components running simultaneously as classes or package versioning is not supported, although this is more a limitation of standard Java.

On the other hand, Service-Oriented Architectures (SOA) [2,7] are built following a different model. Services are similar to components in the sense that they are composed to build applications. Services, however, are specifically designed to be shared at runtime. Services are usually discovered using a service registry before being used in a composition. Web services are the most common incarnation of SOA, however, other frameworks which are based on the SOA principles also exist. OSGi is one of them and is presented later in this article. OSGi was initially designed to build applications running inside home gateways; this kind of environment is typically shared by several providers and must run continuously.

The construction of a component-based application or a service-based application requires different concerns to be addressed. The main difference is that in component models, bindings are static and explicitly described (naming) whereas in service architectures bindings are dynamic as services are referenced in a registry (trading) and can appear or disappear at runtime. Moreover, components tend to be fine-grained assembly units. It is possible to create a considerable amount of component instances inside an application. For instance, a large number of components can be deployed in an application server. On the other side, services. are usually designed to be coarse-grained entities. A reason for this is that in service orientation the program must deal with the inherent dynamism. Therefore the lookup and adaptation required to support dynamic service availability tend to be resource consuming activities which are too costly for fine grained components.

FROGi introduces an approach where concepts from component orientation and service orientation are mixed. FROGi components (either single components or compositions) are used to provide services. This approach allows applications to be constructed as hierarchical compositions where bindings are dynamic. Dynamic binding is supported through the introduction of service orientation concepts. Furthermore, the introduction of support for dynamic binding also allows dynamic deployment activities to be performed.

FROGi is built by introducing the OSGi service platform into Fractal. FROGi intends to illustrate that OSGi can be used to deploy applications build using different component models and furthermore to be able to make these applications interact. This interoperability can occur, for example, between a Fractal component and another like an EJB. The authors have already demonstrated in [8] the dynamic deployment of J2EE applications and technical services on Java EE application servers running on the top of OSGi platform. In this case, FROGi offers a deployment container that takes in charge bindings between components using inversion of control [9], in a similar way to PicoContainer [11]. Finally, FROGi also allows Fractal-based applications to benefit from all the legacy services already offered by the OSGi platform. For instance, Comanche HTTP, a web server implemented with Fractal, can use the Log service specified in OSGi.

4 FROGi Implementation

This section discusses the implementation of FROGi by describing the OSGi framework upon which FROGi is built. It also discusses how components are packaged, an ADL that is used for deployment and finally a generation chain.

4.1 The OSGi Framework

The Open Services Gateway Initiative (OSGi) Alliance [16] is an independent, non-profit corporation working to define and promote open specifications originally intended for the delivery of managed services to networked environments, such as homes and automobiles. These specifications include the definition of the OSGi Services Platform, which consists of two pieces: the OSGi framework and a set of standard service definitions. The OSGi framework is a Java-based deployment and execution environment for components. The OSGi framework was originally conceived to be used inside restricted environments, such as set-top boxes. The OSGi framework can however be used in other domains, as for example, an infrastructure to support underlying release 3.0 of the Eclipse IDE and of the Eclipse RCP.

The OSGi framework supports uninterrupted deployment of components that are delivered inside of *bundles*. The framework also provides a service registry that allows the components to interact following a service-oriented approach. In OSGi, each bundle is used to deploy a single component that results in a unique instance at run time (singleton). The continuous deployment activities supported by the framework include bundle installation, activation, deactivation, update and de-installation of the bundles. The framework ensures that deployment dependencies at the bundle level are satisfied before allowing the bundle to be activated. Bundle activation results in the creation of the component instance deployed inside the bundle.

Physically, a bundle is packaged a jar file that contains binary code as well as resources needed by the component. The jar file manifest file provides meta-information that describes the bundle's dependencies and the name of an activation class. This class is instantiated by the framework upon bundle activation. The bundle's dependencies are divided between deployment-time and run-time dependencies. Deployment-time dependencies are code dependencies described as packages that are exported and imported by the bundles. Run-time dependencies describe the services that are provided or required by the component that is deployed inside the bundle.

Component instances can publish or discover services provided by other component instances at run-time. In OSGi, a service is published from a service interface, a reference toward the component implementing the service and a set of properties. Those properties, defined as keys and values, allow clients to differentiate two equivalent service offers (i.e. two services with the same interfaces). Moreover, the registry allows constraint searches to be made using filters based on the properties following LDAP syntax. Because service publication or departure can occur at anytime, the service registry supports a notification mechanism

that allows service clients to be aware of a particular service arrival or departure events. In OSGi application assembly occurs at execution time as a result of the interaction between components and the service registry.

4.2 Component Packaging

In FROGi, a Fractal application is packaged as one or more bundles. It is important to notice that inside a single bundle, FROGi components are bound together following the standard Fractal approach. However, when components are delivered in separate bundles, components become service providers and binding is performed using the service-oriented interaction pattern which is facilitated by the OSGi platform.

Because a Fractal application is built as a hierarchical composition, FROGi supports independent packaging of primitive components as well as composites. As a consequence, it is possible to perform independent delivery as well as independent update of the components. The example of figure 2 presents the application from figure 1 packaged as a set of bundles. In this example, each component is delivered in a different bundle: B0 for the composite, B1 for the client and B2 for server.

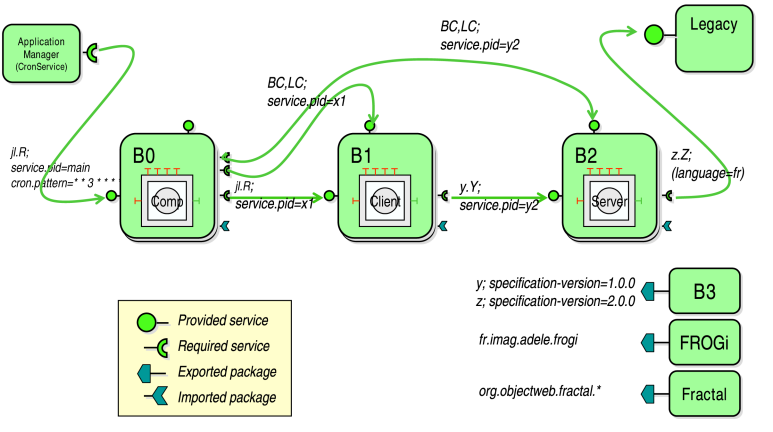

Fig. 2. Fractal application packaged as a set of OSGi bundles

It is important to notice that once published, service interfaces become stable contracts which evolve slowly while their implementations can evolve more frequently. As a consequence, service interfaces used for the binding between components should be delivered in separate bundles (for example bundle B3 in the figure 2 contains the interfaces implemented by the components in the other bundles). The bundles that implement interfaces have a deployment-time dependency towards the bundle that contains them. The independent delivery of service interfaces allows implementation bundles to be updated without impact

on the other bundles. If services interfaces were delivered with their implementation, a bundle update would lead to stopping and restarting (i.e. refresh) of the bundles that depend on those services interfaces. This situation can be problematic when applications run in non-stop environments. In FROGi, the Fractal API as well as the Julia runtime are themselves delivered inside a bundle (fractal.jar); this bundle exports packages that must be imported by bundles containing Fractal components.

FROGi uses standard OSGi mechanisms for managing deployment activities of a bundle-based Fractal application. During bundle installation, the OSGi framework resolves in an automatic way deployment dependencies corresponding to packages containing service interfaces as well as the Fractal API. When those dependencies are resolved, the bundle can be activated. Activation of a FROGi bundle results in the instantiation and activation of an object from a generic class, FrogiBundleActivator, contained in each FROGi bundle. This class is responsible for configuring Julia execution environment (notably by specifying that the classloader to use is the bundle one). It then instantiates a primary class (i.e. BootStrap) that is responsible for creating the component(s) instance(s) delivered by the bundle.

4.3 Component Runtime

This section describes the runtime environment associated to FROGi.

Controller Publication. Once a FROGi component instance (i.e., Fractal components located at the bundle root) is created, its control interfaces are published in the OSGi service registry. The publication of those interfaces allows a third party bundle (its encapsulating composite or an administration bundle) to control the component instance's lifecycle. Management can, however, also be performed externally, for example using a JMX Agent [10].

Instance Binding. Trading associated with the service oriented approach is used in FROGi to support binding of component instances that are delivered in different bundles. The use of trading allows flexible bindings to be created. A binding can be performed, for instance, with regard to any instance providing a particular service (i.e. org.osgi.service.log.LogService). Furthermore, services are characterized by a set of registration properties (such as "'language=en"' or "'cron.pattern=***3***"' in figure 2). Trading also allows 'static' bindings to be created. In that case a service request must contain the unique instance identifier (i.e. the property service.pid) towards which the binding must be created.

Life-cycle and Binding Management. FROGi proposes two policies to manage the life-cycle and binding of the components: a composite-driven policy and an autonomic policy. The instance life-cycle of a root component can be managed either by its composite (delivered in another bundle), either by itself in an autonomous way. Life-cycle management by the composite requires the instance control interfaces to be published as services in OSGi service registry. Each

service is identified by the service.pid properties. This properties identifies the instance that provides the service in a unique and persistent way. The composite creates bindings between instances through the BindingController services they expose. Once those binding are created, the composite activates the instances with the help of their LifeCycleController services.

The alternative to this policy is to consider the bundle as an autonomous life cycle management unit of the instance with regard to its composite. This policy is inspired from the Service Binder (see section 6.2). The instance is started as soon as mandatory services dependencies are available in the registry. This last policy is used for connecting components to legacy bundles that are devoid of life cycle and binding controllers.

Dynamic reconfiguration. Whatever policy is used, it is necessary to support dynamic reconfiguration when the framework notifies that new components are introduced or removed from the environment. If an arriving component is required by another one, the binding must be performed. If a component leaves, the components that depend on it must check in the registry that the mandatory services they depend on are still available. In the case of the autonomous policy, provided services are systematically unregistered of OSGi registry at component stopping time. They are registered again (still with the same service.pid attribute) during the component instance restart.

Application Activation. A Fractal application is a component/composite that can be activated from one of its functional interfaces by Fractal support classes such as org.objectweb.fractal.adl.Launcher. In OSGi, the application concept doesn't really exist: the application is built as a set of bundles that create connections as they are installed or removed from the framework. Bundles can, however, be classified into two categories: support bundles (i.e., which provide services), and coordination bundles (which may not provide services but use services provided by other bundles). Coordination bundles are closer to the concept of an application, however, these bundles may themselves provide services to other bundles and become part of a bigger application. In FROGi, an application manager is responsible for activating the Fractal application deployed on OSGi. This can be for instance a Cron Service calling the run() method of a component or an administrator command executed on the terminal console.

4.4 Extensions to the Fractal ADL

Fractal provides an Architecture Description Language (ADL) that allows component assemblies to be described. As previously mentioned, this ADL is extensible. FROGi extends this ADL to take into account the deployment aspects of the components, i.e. Packaging them within bundles.

The extended ADL is specified as shown in figure 3. This example presents the Fractal ADL description used to obtain the FROGi packaging of figure 2 for the application depicted in figure 1. The `<bundle>` sub-element of the `<component>` and `<definition>` elements define how components are packaged inside the

```
<definition name="HelloWorld">
  <bundle name="B0"/>
  <interface name="main" role="server" signature="java.lang.Runnable">
    <property name="cron.pattern" value="** * 3 * * *"
        type="java.lang.String"/>
  </interface>
  <component name="client">
    <bundle name="B1"/>
    <interface name="x1" role="server" signature="java.lang.Runnable"/>
    <interface name="cy2" role="client" signature="y.Y"
      version="1.0.0" bundle="B3"/>
    <content class="ClientImpl"/>
  </component>
  <component name="server">
    <bundle name="B2"/>
    <interface name="y2" role="server" signature="y.Y"
      version="1.0.0" bundle="B3"/>
    <interface name="cz3" role="client" signature="z.Z"
      cardinality="collection" contingency="optional"
      version="2.0.0" bundle="B3"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.x1" server="client.x1"/>
  <binding client="client.cy2" server="server.y2"/>
  <binding client="server.z3" server="this.cz3"/>
  <binding client="this.z3" serverfilter="(language=fr)"/>
</definition>
```

Fig. 3. The Extended Fractal ADL

bundle identified by the `name` attribute. The `version` attribute specifies the overall implementation version. It corresponds to the bundle's `Bundle-Version` manifest attribute in OSGi. All the elements that are declared after a `bundle` element are packaged together in the same bundle and this occurs until another `<bundle>` element is encountered.

The `bundle` attribute under the `<interface>` element indicates that the interface must be packaged inside another bundle whose name is specified by the name value. If the bundle attribute value is an empty string, the interface is not packaged by FROGi: it is already available in another bundle, generally a legacy bundle. By default, if nothing is specified, service interfaces are packaged in the same bundle as their component implementation. The `version` attribute of the `<interface>` element declares the package specification version (i.e. contract) of the interface. The default version value is 0.0.0.

The sub-element `<property>` of the `<interface>` element defines some properties that are associated to the service interface and which are used when the interface is published in the service registry. Those properties are used for service trading and and to provide information to application managers.

The sub-element `<binding>` of the `<component>` and `<description>` elements is used simultaneously to create standard Fractal bindings between instances created in the same bundle, bindings between instances created into separate bundles and bindings between instances and legacy OSGi services. The `server` attribute can be substituted by a `filterserver` attribute whose value is a LDAP expression that the requiring service must match to perform the binding. This attribute is not available for standard Fractal bindings (i.e. intra-bundle). We can notice that the `serverfilter="(service.pid=server.y2)"` attribute is equivalent to `server="server.y2"`.

4.5 Generation and Deployment Chain

The extended ADL presented in the previous section allows packaging tasks to be automated using a generation and deployment chain. Once Fractal components are packaged inside bundles, the facilities provided by the OSGi platform are used to perform their deployment.

The first step in the chain is concerned with bundle generation. This activity is performed by the FROGi packager (left of figure 4). The packager parses the ADL and packages interfaces and implementations following the ADL descriptor. The packager tries to separate interfaces from implementations since this is essential to support dynamic component updates.

The deployment is managed by another tool dedicated to OSGi deployment (right of figure 4). This tool manages OSGi gateways distributed over several nodes. It reads deployment files that are produced by the FROGi packager (xml files). These files contain both the localizations of the generated bundles and the gateway on which they must be deployed. The description also contains the dependencies between the bundles. The tool deploys the FROGi bundles and, if necessary, depending of the state of the targeted OSGi platform, it also deploys required OSGi legacy bundles. Those bundles, and possibly their dependencies, are made available from bundle repositories (such as the Oscar Bundle Repository, Oscar (`http://oscar.objectweb.org`) being the open source OSGi implementation we are using for demonstration purposes).

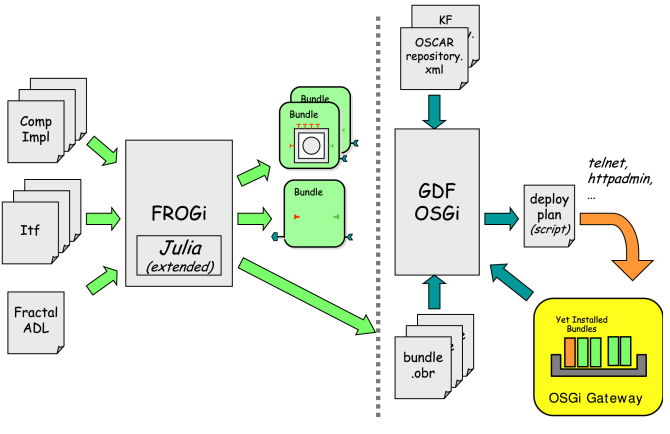

Fig. 4. Generation and deployment chain

4.6 Security

Service oriented architectures and service deployment require security aspects to be taken into account. In the context of FROGi, it is necessary to ensure that an architecture that is deployed using the ADL functions properly after installation. The components that interact with legacy OSGi services must be able to trust

them. This concern is exacerbated by the fact that the OSGi environment is designed to be operated by different actors, and a FROGi-based application may coexist with unsafe bundles from a different provider.

FROGi currently relies on the mechanisms provided by the OSGi framework to handle security. These mechanisms allow bundles to be signed so that other bundles can verify their origin. This offers an initial level of security. The second level occurs at the service level. OSGi provides a mechanism that allows services to be traded according to security policies. Furthermore, those policies can be updated dynamically. Security mechanisms at the service level are adequate for FROGi because they bring additional capabilities to the component model. Finally, It must be noted that Fractal does not support these concepts (which is understandable as it targets mono-operated applications).

5 Experimentation

This section presents an experimentation which compares the creation of an HTTP server using a "'standard"' approach versus a FROGi-based approach. The experimentation is inspired from the comanche HTTP server discussed in the Fractal tutorial.

5.1 Using Standard Fractal

Figure 5 depicts a minimal HTTP server. This server is assembled as a composite component that is responsible of receiving, analyzing and dispatching requests (to simplify, only the external composite is shown, not the contained components). This component requires a Log component and one or more handlers towards which the requests will be directed. Before a call arrives to a handler, the request may go through different filters that are capable of adapting the requests or that can be used as probes for example to collect information. To realize this example in standard Fractal, all the needed components are described in the ADL along with their bindings.

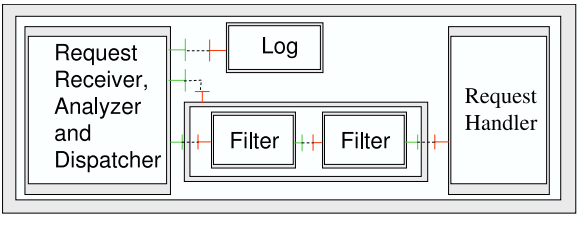

Fig. 5. A minimal HTTP server with Fractal

Once deployed and during execution, it is still possible to adapt the bindings between the components. For example it is possible to disconnect the Log component if we do not want to trace the requests anymore. It is also possible

to adapt the filter chain between the requests manager and the request handler by connecting or disconnecting filters. This adaptation is taken in charge by the requests analyzer and dispatcher. What is not possible, however, is to add dynamically a new filter that was not previously described in the ADL. This is simply because the implementation classes of this filter are not deployed with the original application. The same problem occurs if a filter needs to be updated, for example for performance reasons. It is possible to disconnect the filter properly but no mechanism is available to perform an update of the filter's implementation and maintain the coherency.

5.2 Using FROGi

The construction of the same example using FROGi illustrates three key points: the capability of using legacy OSGi services, of dynamically deploying new components and of updating components without restarting the application.

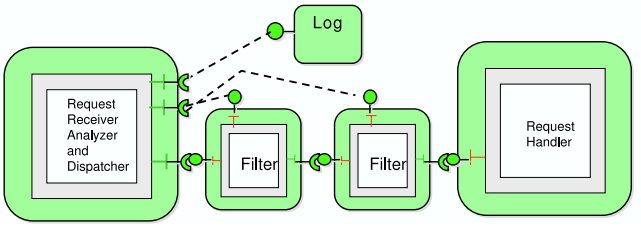

Fig. 6. A minimal HTTP server with FROGi

Figure 6 depicts how the HTTP server application is assembled and deployed using FROGi components. The capability of using legacy OSGi services is illustrated by replacing the previous Fractal log component with the Log service defined in the OSGi specification. Deployment concerns are now addressed since the components are packaged into different bundles which are later managed by OSGi framework. In this example, the filter components are packaged and delivered in different bundles. As a result new filters can be deployed easily. Using the trading mechanisms, the Dispacher is able to select, among the set of filters, the ones that it requires to create the filter chain. Updating a component is also possible and is supported by the OSGi update mechanism. First of all, bindings with the corresponding component are relieved. Then the update mechanism manages the download, replacement and reactivation of the component embedded in the bundle. During this period, Fractal's interception capabilities are used to hold the calls towards the components until it they are reactivated. It is interesting to notice that as soon as a component is not used anymore, it is possible to uninstall it and completely free the resources it was using. F This simple example shows that the FROGi's features introduce important benefits into the standard Fractal model.

6 Related Works

This section presents different related works concerning the OSGi use as an infrastructure for deploying components as well as Fractal components packaging.

6.1 Beanome

Using OSGi as a component deployment infrastructure is explored in the Beanome component model [4]. In Beanome, OSGi bundles are used to deploy COM-like components. Moreover, the OSGi service registry is used to publish components factories when the bundle is activated. A benefit of registering component factories as services is that factories can be located based on the functionalities of the components they create and not only from a unique identifier as in COM. Beanome, however, does not provide support for dynamic changes.

6.2 Gravity

The Gravity project [5] explores the creation of applications with autonomous adaptation capabilities towards component availability. Gravity introduces a service oriented component model in which trading is used at run time to bind component instances as well as to maintain compositions despite components arrival and departure. In Gravity, an execution environment entity, called the Service Binder, is in charge of adapting component instances and compositions with respect to dynamic changes. Gravity is built as a layer on top of OSGi, and the Service Binder is deployed as a bundle inside the service platform. A drawback of Gravity is that is uses a particular component model that is nevertheless not far from Fractal. Many of the ideas introduced in the Service Binder have been recently added to the OSGi specification's 4th release under the name of Service Component Runtime (Declarative Services). This component is also the subject of the JSR 291 (Dynamic Component Support for Java SE) submitted to the JCP by several members of the OSGi Alliance.

6.3 Fractal Packages and Deployment Activities

Some discussions on the Fractal mailing list mention the definition of a packaging mechanism for Fractal components and some work has been realized concerning this mechanism. The proposals that have been made also rely on OSGi but only for packaging purposes (packaging units are .FAR)[1]. An XML manifest that contains the metadata is added to the archive. Deployment is supported but it is impossible to update components at runtime. This proposal does, however, not consider the existence of an infrastructure to perform continuous deployment activities. This issue is addressed in another work [12]. This proposal uses a layer that supports the creation of Java classloaders to bring additional components to an application at runtime. This work does, however, not support component uninstall.

6.4 JSR277

Packaging an application is one of the most recurrent problems to facilitate deployment. JSR277 (Java Module System `http://www.jcp.org/en/jsr/detail?id=277`) aims to specify an unified packaging model for all Java software for J2SE 1.7 (2007). JSR277 intends to overtake JNLP, J2EE EAR, OSGi R4 packaging formats. It will be based on the JAR file format and the Manifest will be augmented by explicit versioned package dependencies. In fact, the chapter "Module Layer" of the recent OSGi R4 specifications already covers all of JSR 277 requirements. Moreover, JSR277 does not address the OSGi service layer which enables to build dynamic service-oriented architectures of Java applications as SCR, JSR 291 or FROGi. If this JSR is integrated in Java, FROGi would already be compliant at the packaging level with future Java versions.

7 Conclusions and Perspectives

This paper has presented FROGi, a proposition that is based on the introduction of some characteristics of the OSGi service platform in the Fractal component model. With FROGi, a Fractal application is packaged inside one or more OSGi bundles; this allows the components to be delivered and deployed individually and continuously. Moreover, binding between components instances can be realized either through the 'standard' Fractal connexion technique, either by the publication of functional interfaces in the services registry and the use of OSGi proper trading technique. In addition, FROGi proposes Fractal ADL extensions to automate packaging and deployment. It must be pointed out that FROGi, as well as the different works described in the fifth section, show that OSGi is an ideal platform to perform component deployment, application update and code versioning. Nevertheless, some points have not been considered in the work realized until now:

Multiple instances creation mechanism: Fractal supports the creation of a variable number of component instances. The work presented here focuses on a singleton based approach. A way to resolve this, still being compatible with the OSGi environment, is to publish components factories through services (similar to the approach followed by Beanome and described in 6.1).
Architecture introspection: as we assume that different kinds of components can be deployed and bound on OSGi, it is desirable to expose the architecture of the application independently of the technologies we are using. An example of such architecture viewer is Fractal Explorer but it only manages pure Fratal applications.

Finally, as it was mentioned in the second part, there is currently not a clear vision of the difference between component models and service oriented architectures. Most of the time, these approaches are considered either as orthogonal aspects, either as similar approaches. We have already cited some tracks on the subject and this is the focus of our current research. For instance we are currently working on the interoperability we can have between Fractal and EJB

components model inside an application server and on component deployment on heterogeneous platforms [13].

References

1. Abdellatif, T., Kornas, J. And Stephani, J-B.: J2EE Packaging, Deployment and Reconfiguration Using a General Component Model. Proceedings of Component Deployment, CD, Grenoble 2005
2. Bieber, G., Carpenter, J.: Introduction to Service-Oriented Programming. OpenWings whitepaper, Septembre 2001, `http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf`
3. Bruneton, E., Coupaye, T. and Stefani, J.B.: The Fractal Composition Framework Version 2.0-3. Object Web Consortium, July 2004.
4. Cervantes, H. and Hall, R.S.: Beanome, A Component Model for the OSGi Framework. Proceedings of the workshop Software Infrastructures for Component Based Applications on Consumer Devices, Lausanne, 2002
5. Cervantes, H. and Hall, R.S.: Automating Service Dependency Management in a Service-Oriented Component Model. Proceedings of CBSE 6, Portland, USA, 2003
6. Cervantes, H., Desertot, M. And Donsez, B.: FROGi: Dploiement de composents Fractal sur OSGi. Proceedings of Decor'04, CoRR, Grenoble 2004
7. Cervantes, H. and Hall R. S.: Chapter I: Service Oriented Concepts and Technologies. In the book "Service-Oriented Software System Engineering: Challenges and Practices" (ISBN 1-59140-426-6) edited by Zoran Stojanovic and Ajantha Dahanayake, Idea Group Publishing, 2005.
8. Desertot, M., Escoffier, C. And Donsez, D.: Autonomic administration of J2EE Edge Servers. Proceedings of the International Worshop of Middleware for Grid Computing (MGC), Grenoble, 2005
9. Fowler, M.: Inversion of Control and the Dependency Injection Pattern. Online Document, 2004. `http://martinfowler.com/articles/injection.html`
10. Frnot, S. And Stefan D.: Instrumentation de plate formes de services ouvertes Getion JMX sur OSGi. Ubimob, Nice, 2004
11. Hammant, P., Hellesoy, A., and Tirsen, J.: PicoContainer: a lightweight embeddable container. `http://www.picocontainer.org`
12. Kornas, J., Leclercq, M., Quema, V. And Stephani, J-B.: Support pour la reconfiguration d'implantation dans les applications a composants Java. Proceedings of Decor'04, CoRR, Grenoble 2004
13. Marin, C. And Desertot, M.: SensorBean: A Component Platform for Sensor-Based Services. Proceedings of the International Worshop of Middleware for Pervasive and Ad-Hoc Compouting (MPAC), Grenoble, 2005
14. Szyperski, C.: Component software: beyond object-oriented programming. ACM Press/Addison-Wesley Publishing Co., 1998.