

# Raising Web Service Updates Efficiency with Dynamic Technologies

Valery Abu-Eid

Spb. Electro technical University

vladrin@dynamicjava.org

## Abstract

*This paper presents a solution which uses OSGi to allow the development of Dynamic Web Service Applications. Taking into account the update efficiency of Dynamic Applications in general, this solution was developed to increase Web Service Updates efficiency.*

## 1. Introduction

Like any other type of application, Web Service Applications need to be updated, these updates can be Web Service technology related, e.g., Web Service Contracts, Security parameters, Message Handlers, etc. or related to the functionality provided by the Web Service, e.g., Domain Logic, Data Access Logic, etc. In order to Perform Web Service Updates developers and administrators use update solutions provided by Application Servers (like Sun's Glassfish and Microsoft IIS) or Web Service Engines (like JAX-WS RI, Apache Axis2, Apache CXF, ASP .Net, etc.). Taking into account that application update at runtime is merely a feature of Application Servers and Web Service Engines, it's no surprise that certain insufficiencies in the update solutions provided by them affect Web Service's availability during updates and introduce some other problems that will be discussed later in this paper.

Dynamic technologies (like OSGi) on the other hand are all about components coming and going at runtime and such technologies would be for nothing if each time a component is updated the application gets affected in any way. The only limitation of OSGi from Web Services perspective is that it offers no specific solutions that meet the needs of Web Service Applications.

Dynamic-WS is a solution which brings dynamicity to Web Service Applications using OSGi. Dynamic-WS should allow updating Web Services efficiently by not only keeping them available, but alive also (in the

sense that they process clients requests while being updated).

This paper presents Dynamic-WS and compares it to current Web Service Update solutions. The remainder of this paper is organized as follows: Section 2 observes current update solutions. Section 3 provides a brief introduction to OSGi. Section 4 presents Dynamic-WS. Dynamic-WS will be evaluated and compared to other solutions in Section 5 before I conclude in Section 6.

## 2. Current Update Approaches

Below are the two mostly used Web Service Update approaches.

### 2.1. Web Application Hot Updates

This approach can be used if the Application Server supports Web Application hot updates (updates to the Web Application at runtime). ASP .Net Web Applications running on IIS and Java Web Applications running on Glassfish Application Server follow in this category.

Web Application Servers that support hot updates can be divided into two categories depending on how they deal with client requests during Web Application updates:

- Servers that reject the requests: They return HTTP Status code 405 (Service Unavailable) for requests to the application during updates.

- Servers that put the requests on hold: They put the requests to the application during updates on hold and process them when the update is finished.

Glassfish follows in the first category and ASP .NET Web Applications on IIS in the second.

It's worth to note, that if the amount of the requests to an application during an update exceeds the maximum number of concurrent requests that the application can handle, then the rest of the requests will

be rejected. Basically, this problem occurs when an application that is under heavy load being updated – Actually, not too heavy, Servlet Containers are usually configured by default to have the maximum handled parallel requests in the range of 150-250, so if an update takes 5 seconds to an application that receives 60 requests per second then many requests will be rejected.

## 2.2. Web Service Archive Hot Update

A Web Service Archive is a ZIP file that contains Web Service Implementation classes and a Web Services descriptor file. It's important to note that a Web Service Archive can't contain class library files – In other words it can't contain other modules within it. As such, Web Service archives usually define the Web Service contract, handlers, features and Web Service Engine specific configurations.

In Apache Axis2, any requests made to a Web Service whose Web Service Archive is being updated are processed by the current version of the Web Service, when the update is finished, new requests will be processed by the new version. Basically, in this technology the Web Service is not only available (receives requests) but also alive (processes them).

Web Service Archives are not a typical way to deploy Web Services since they are a Web Service Engine specific capability. For instance, of the most used Java Web Service Engines Apache Axis2 is the only one that supports Web Service Archives, unlike Apache CXF and The Reference Implementation of JAX-WS (JAX-WS RI).

## 3. OSGi

OSGi is a dynamic module system for Java that provides Java applications with:

- Modularity: Applications are composed from modules.
- Dependencies: Modules can depend on each other in a declarative way.
- Visibility: Modules contents can be shared with other modules.
- Versioning: Both modules and their contents have versions.

The rest of this section will discuss the main concepts of OSGi.

### 3.1. Bundles

The Bundle is the unit of modularization for OSGi. It defines classes and other resources (images, text files, etc.), and is capable of sharing them with other

bundles. It's also capable of registering services that can be consumed by other bundles.

Bundles live in OSGi environments. They can be installed, updated and uninstalled at runtime.

In the context of Web Services, if we develop a Web Service Application then we would separate it into bundles that interact with each other. We might place all contract related entities (be they Java classes or descriptor files) in a separate bundle, so if we would need to update the contract of the Web Service, the only bundle that we will have to update is the contracts bundle and nothing else - Which is of course much more effective than updating the whole application each time we need to update a component of the Web Service Application. Other bundles might contain business logic, data access logic, common interfaces, configurations, handlers, etc.

### 3.2. Class Sharing

As mentioned before bundles can share classes they contain with other bundles. For a bundle to share its classes it needs to export them, and for another bundle to use them it needs to import them. Bundle's exports and imports are declared in its metadata.

Class sharing is useful if the bundle needs to use classes provided by other bundles or if it refers to a specific version of the class, but if the functionality which is consumed by the bundle is dynamic then class sharing is not a suitable solution (for a lot of reasons that are not the subject of this paper) and for similar cases OSGi Services are the best option.

### 3.3. Services

Any bundle running in an OSGi environment can register a Service that implements a certain interface. The registered service can be consumed by other bundles.

Services are dynamic in nature, they can be registered and unregistered at runtime. An OSGi environment can contain multiple services that implement the same interface and it's up to the bundle to consume them all or just one of them. Services also may have properties so they can be filtered by consumer bundles.

If bundles of an OSGi Application need to share functionalities in a dynamic fashion then services are the best option.

An example of services would be a Data Access service that provides Data Storage capabilities to a relational database. First we would register a service that implements a Data Access Object interface which

works with a relational database. If we would like to change data storage to an objects database or a files database, then all we need to do is to register a service that provides the new functionality and implements the same interface with a rank higher than the rank of the previous one, then all service consumers will use the new one – Actually the implementation of the example is a little simplified although it follows the same principle.

### 3.4. OSGi Framework

OSGi Framework provides the OSGi Environment where bundles could be run. Currently there are three popular Frameworks: Eclipse Equinox, Apache Felix and Knopflerfish – All of them are open source. OSGi Frameworks can be run separately or embedded into other applications. To use OSGi capabilities within a typical Servlet Container or Application Server (like Tomcat or Glassfish), the only viable option is to embed the framework inside the Application Server.

## 4. Dynamic-WS

Dynamic-WS is an open source project that integrates OSGi with current Web Service Engines to:

1- Allow developers to deploy Web Services as OSGi bundles.

2- Make Web Service Engines support the dynamicity of Web Services running in OSGi environments.

Next section discusses the barriers that prevent achieving these goals with current technologies.

### 4.1. Barriers of using OSGi with Web Services

The main problem of current Web Service Engines is that they either register application's Web Services at initialization time, as such Web Services can't be registered and unregistered later on (like JAX-WS RI and Apache CXF) or that they don't provide users with an API that allows them to register and unregister Web Services at runtime (like Apache Axis2).

Another problem is the fact that in order to simplify the process of registering Web Services to users, all they

need to do should be declaring the Web Services they want to register in a descriptor file contained in the bundle – The same way they do with current Web Service Engines but with one slight difference in that descriptor files should be contained in bundles. When a bundle is activated the Web Services declared in it should be registered and when it's deactivated the Web Services should be deregistered. To achieve this goal a bundle listener that scans bundles for Web Service Descriptors must be implemented.

Dynamic-WS provides services that process Web Service Descriptor files contained in bundles and interacts with Web Service Engine's internals to register and deregister Web Services at runtime.

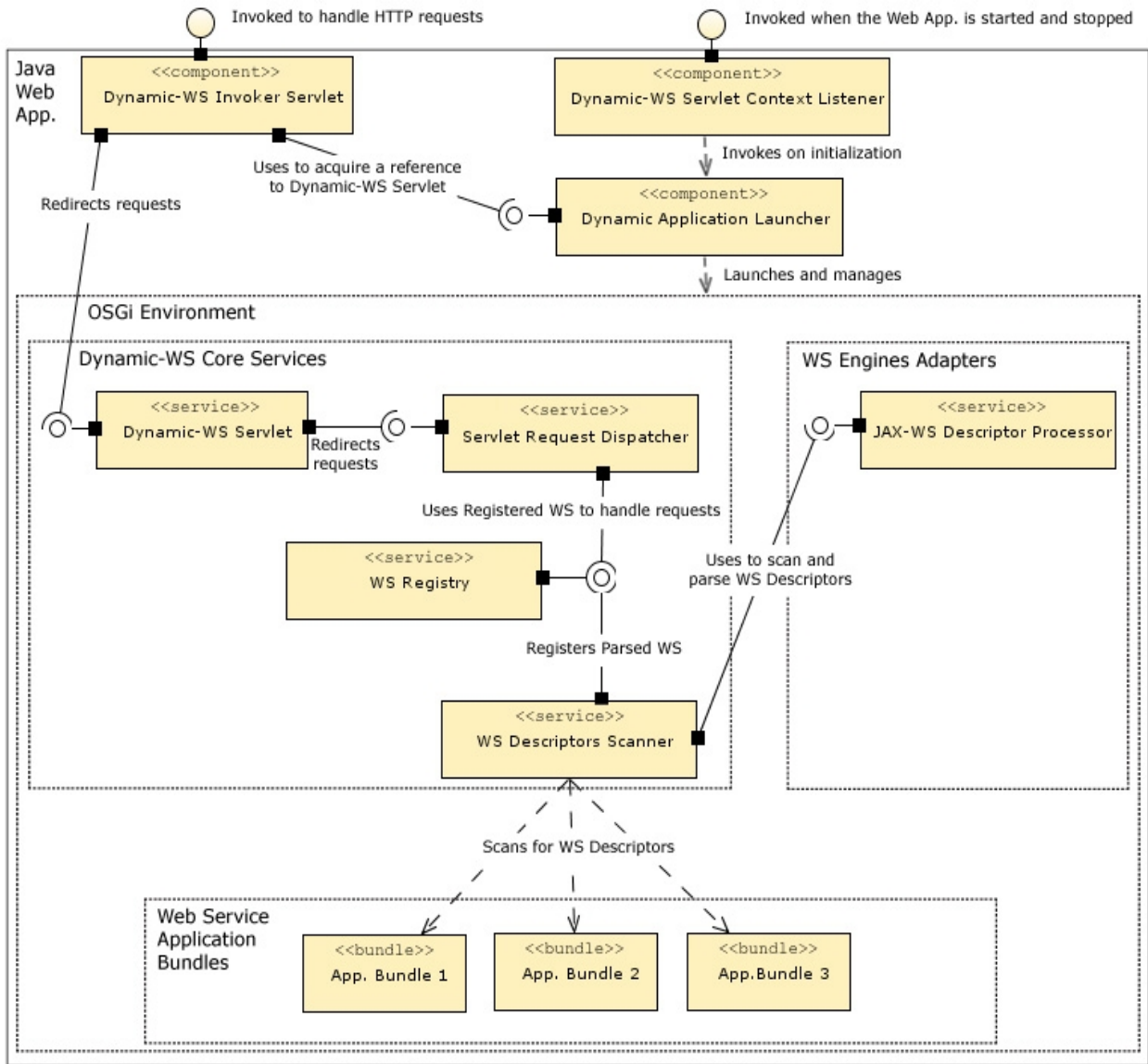
### 4.2. Dynamic-WS Architecture

The detailed architecture of Dynamic-WS used in a Java Web Application is presented in Diagram 1. Below is a description of each element:

- Dynamic-WS Servlet Context Listener: Is a Servlet Context Listener that that will be called by the Servlet Container when the Web Application is started and stopped. It starts the Dynamic Application Launcher when the Web Application is started and stops it when the application is stopped. It saves a reference to the objects provided by DA-Launcher in the Servlet Context so Servlets in the Web Application can interact with services in the OSGi Environment.

- Dynamic Application Launcher (DA-Launcher): Is a module that simplifies launching OSGi applications and managing bundles. It's used here to make the process of launching the OSGi Environment and running Dynamic-WS bundles transparent to the user. It also provides the users with easy means to install bundles by allowing them to simply copy a bundle archive to a certain directory if they want to install a bundle, and remove it from there if he wants to uninstall it.

- Dynamic-WS Invoker Servlet: This Servlet acquires a reference to the OSGi Service Dynamic-WS Servlet and redirects to it all the requests. All the requests processed by Web Service provided by Dynamic-WS are redirected from this Servlet.



**Diagram 1. The Architecture of Dynamic-WS used in a Web Application**

- Dynamic-WS Servlet: Is an OSGi Service that implements the Servlet interface, to which all the requests are redirected from Dynamic-WS Invoker Servlet. All it does is redirecting requests to the Servlet Request Dispatcher service. The reason why there are two Servlets, is because on one hand we need to register a Servlet in the Servlet Container so we would be able to have requests to the application, on the other, Servlets that are loaded by the Servlet Container are not dynamic (can't be updated at runtime), and since the solution is all about dynamicity we need to have the solution itself (Dynamic-WS) to be dynamic so users

can update it with a newer version at runtime without affecting application's availability or performance in a single way. That's why there are two Servlets, one is which all it does is redirecting requests to the second one and the second is dynamic which has the actual logic in it and is updatable.

- Servlet Request Dispatcher: Is an OSGi Service which chooses a Web Service from the registry based on request's URL and hands it the request.

- Web Service Registry: Is an OSGi Service where all the Web Services are registered. It provides the Web Services to other services and notifies them about any updates.

- Web Service Descriptor Scanner: Is an OSGi Service which scans bundles for Web Service Descriptor files and processes them with matching Web Service Descriptor Processors to generate Web Services out of them and then registering them in the Web Service Registry.

- Web Service Descriptor Processor: Is an OSGi Service which generates Web Services from Web Service Descriptor files using Web Service Engine's internals. The Processor gives a hint to the Scanner about the Descriptor files it can process to generate Web Services. Every supported Web Service Engine has at least one Web Service Descriptor Processor that uses its internals to generate Web Services. For instance, the JAX-WS RI Web Service Engine has one Processor which processes a Descriptor file named "sun-jaxws.xml" in the "/META-INF" directory of the bundle and uses Engine's internals to generate a Web Service from it.

It's worth to note the fact that all the internal functionalities are implemented as OSGi Services. This was done to make the solution highly dynamic. Any Service of Dynamic-WS can be replaced by a Service that implements the same interface and has a higher rank. Taking into account that the Dynamic Application Launcher is used, updating Dynamic-WS at runtime without affecting the availability or the performance of running Web Services is turned into a matter of copying and pasting newer module files.

## 5. Evaluation

The evaluation consists of two parts. In the first part, I will highlight the differences between Dynamic-WS and the other update solutions. In the second, I will compare the behavior of a Web Service Application that uses Dynamic-WS as the update technology and another one which I update by performing hot updates to the Web Service Application.

### 5.1. Update Aspects and Characteristics Comparison

Table 1 shows the aspects and characteristics of update solutions provided by Dynamic-WS and other technologies. Although data in Table 1 speaks for itself, I would like to highlight the main advantage of Updating Service Bundles in OSGi using Dynamic-WS which is that it keeps Web Services alive during updates (like Axis2 does) while not constraining the supported Update Components (like IIS and Glassfish does).

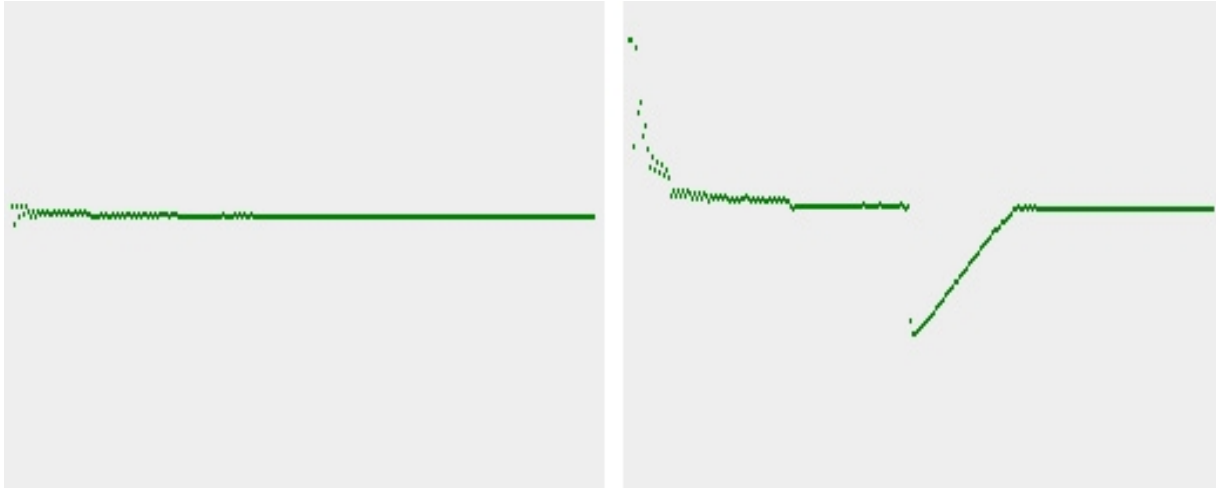
### 5.2. Comparison of Web Service Applications Behavior during Updates

Having two versions of a Web Service, one a Java version runs in Apache Tomcat and the other a .Net version runs in IIS, using JMeter I send each Web Service 25 requests per second for 10 seconds. Around the 5th second, I update a module that defines the service of each application. Figure 1 was generated by JMeter and it demonstrates the changes to throughput of Web Services during update, X axis reflects time change and Y axis relative change of Web Service's throughput.

The green curve shows relative changes to throughput and doesn't provide concrete numbers. Observing the curve of the .Net application running in IIS, we can tell that the stable period of the throughput - Where the curve were almost perpendicular to the Y axis - is the period where the curve where receiving a constant amount of requests (25 per second) without having any external effects, but when the module file was updated at around the 5th second, the throughput decreased heavily then began recovering until 6th second where it came back to normal. On the contrary, nothing to be noted in the OSGi version of the application that uses Dynamic-WS since the curve is almost perpendicular to the Y axis all the way and there is nothing special in the diapason between 5th and 6th second.

Technology	Method	Components	Design Constraints	Runtime support	WS Available	WS Alive
Glassfish	Web App. Hot Update	All	No Constraints	Yes	No	No
ASP .Net	Web App. Hot Update	All	No Constraints	Yes	Yes	No
Axis2	WS Archive Hot Update	Contract, Features, Handlers	Updatable components should be placed in the WS Archive	Yes	Yes	Yes
Dynamic-WS	Services Bundle Update	All	Bundles should communicate through OSGi Services	Yes	Yes	Yes

Table 1. Update Solutions Aspects and Characteristics Table



**Figure 1. Throughput of the tested Web Services**  
**(a) Throughput of the OSGi Version**                      **(b) Throughput of the .Net version**

I choose ASP .Net on IIS in favor of Glassfish because during updates, Glassfish would have returned error responses. So I would have had to include another curve that shows the errors. I preferred to choose the simplest way to prove my point of view.

As for Axis2, since Axis2 doesn't allow Web Service Archives to contain other modules, such test couldn't be performed on Axis2 because the updated module (Date Retrieving module) is separate from the one that contains the contracts (Greeter Service module). Though, if the updated module was the one that contains Web Service contracts, the throughput would have looked similar to the throughput of the OSGi application. But in real life it will be rarely the case that business logic is contained with Web Service contracts in the same module.

## 6. Conclusion

Unfortunately, very few or none researches were done on the subject of Web Service Updates – At least that's what seemed to the author. This might be due the fact that the inefficiency in the update process at the software level is usually solved by administrative means – Like handling requests by a reserve server during updates. The research observed the most common update approaches and discussed their main insufficiencies – Which might be due the high level of complexity of software that provides efficient update capabilities, the fact that leads Web Service Engines to use more modest techniques for updating Web Services. On the other hand, OSGi is a system whose sole focus is dynamic applications, it provides

everything a developer needs to develop highly dynamic and modular applications. It might seem more convenient to delegate the task of handling Web Service Updates to OSGi while keeping Web Service Engines focus on problems related to the Web Services domain. This paper tries to take the first step in this direction by providing a solution that integrates OSGi with current Web Service Engines to allow the development of Dynamic Web Service Applications.

Having a Web Service Update efficient solution at the software level not only can reduce the resources required to maintain highly available Web Services, but also reduce the barriers of implementing innovative approaches that involve Web Service Updates such as automated versioning of Web Services, simple ways to rolling back Web Service updates, etc.

## 7. References

- [1] OSGi Specification. <http://www.osgi.org/Specifications/>.
- [2] JAX-WS JSR. <http://jcp.org/en/jsr/detail?id=224>.
- [3] JAX-WS Reference Implementation. <https://jax-ws.dev.java.net/>.
- [4] Apache Axis2. <http://ws.apache.org/axis2/>.
- [5] Apache CXF. <http://cxf.apache.org/>.
- [6] Glassfish. <https://glassfish.dev.java.net/>.
- [7] Eclipse Equinox. <http://www.eclipse.org/equinox/>.
- [8] Apache Felix. <http://felix.apache.org/>.
- [9] Dynamic-WS Project. <http://www.dynamicjava.org/projects/dynamic-ws/>.
- [10] DA-Launcher Project. <http://www.dynamicjava.org/projects/da-launcher/>.