MICRO-SERVICE SUPPORT BY MODULE ARCHITECTURE APPLICATION OF THE SERVICE PLATFORM FOR OSGI JAVA ADDITIONS

Oleksandr Mamro, Andrii Lagun
Lviv Polytechnic National University, Ukraine, e-mail: a.e.lagun@gmail.com

Abstract

This article analyzes the problems of creating and maintaining the micro-service-oriented architecture. The solution of the OSGI modular architecture and its alternatives was also considered, as well as the strength and weakness, were identified. The practical part is to create an OSGI system for IoT (wireless network access), which uses a sensor system and a data processing system, with a centralized modular system for processing input data from different devices. The modules are broken down into data processing functionality. The complexity of refactoring micro-service architecture using OSGI modules was also investigated.

Keywords

Micro-services, OSGI, Modular architecture

1. Introduction

The speed and ease of software development are very important components of IT in general. An important step will be to simplify the complex relationships of the microservice structure with OSGI. 2

2. Current State of the Problem

In modern application development from all spheres, one of the most common practices is - micro-service architecture - is the separation of functional code into so-called "services" that are responsible for their process, which is easy to reuse. Also, the great advantage of micro-services is the ease of development due to limited functionality - the development of micro-services can be undertaken by a single team in no way intersecting with another service developed by the second team. Communication between these services must be via a well-defined interface such as HTTP. They "communicate" through APIs for which programming language is irrelevant - which is also a great advantage nowadays due to a large number of ready solutions in different programming languages.

Each micro-service can be deployed separately from the others. Therefore, if something changes in one of the services, you can expand those changes without stopping other micro-services that may continue to work. The application will always meet all needs and you can make any changes as often as you need. This is the main difference from a monolithic architecture, where any change requires the deployment of an entire complex system. Therefore, WEB projects are more often developed as separate services that can be scaled and deployed separately, which is very convenient and flexible to develop. A generalized structure of micro-service architecture is shown in Figure 1.

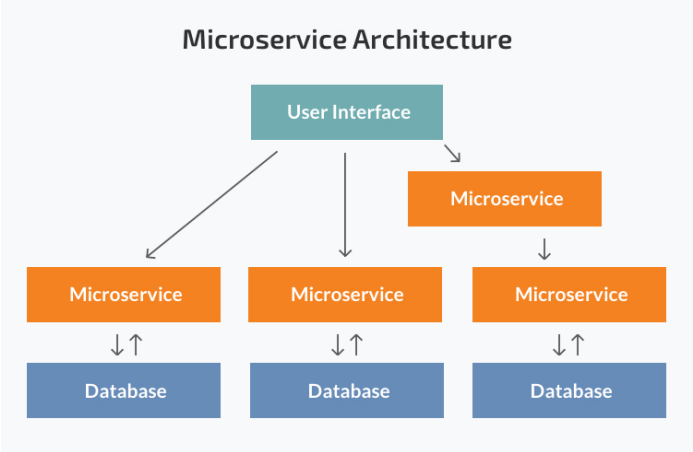

Fig.1. Micro-service architecture structure

There are various disadvantages to micro-service architecture since its formation.

1) Network delays: micro-service architecture requires the atomization of modules and their interaction over the network, unlike the modular ones that interact locally (if they perform several functions). The network is unreliable in

nature. The network may just refuse, it may not work properly, it may suddenly stop skipping any type of messages because the Firewall settings have changed. There are also various causes and types of inaccessibility. Therefore, micro-services may suddenly stop responding, or start responding slower than usual. And every remote call must take this into account; must handle different options of failure correctly, be able to wait, be able to return to normal operation when recovering a contractor.

2) Message formats: the lack of standardization and the need to agree on exchange formats for virtually every pair of interacting micro-services leads to both potential errors and difficulties in debugging;

3) Balance of load and fault tolerance

4) The complexity of operational support - requires competent DevOps engineers, continuous deployment and automatic monitoring. Without all this, you should not use micro-services.

5) Bulky testing of micro-services. In micro-services, each individual service must be started before testing begins when using a monolith; all you have to do is run the application on the server and make sure that the database is connected.

An additional level of complexity is created by the event architecture. And setting up such a system is not a single micro-service, but a system where many streams of multidirectional unordered events are a serious problem. And even the flawlessness of every micro-service in terms of business logic is not a sufficient condition for design. By analogy with sports, "stars" do not guarantee a star team, because the team is more important than "stars", but the coherence of all players. The structure of the system using micro-service architectures is shown in Figure 2. Usage of modular systems creates another problem - the presence of the so-called “spaghetti code”, which can be partially solved by OSGI modules. Spaghetti code is unstructured and difficult to maintain source code that can be caused by several factors, such as changing project requirements, lack of programming style rules, and lack of ability or experience. There is a lot of difficulty in supporting such code and unfortunately, situations where it needs to be done - happen very often. The difficulty is that the time to search for dependencies and further accesses is significantly increasing.

Adding the whole microarchitecture to finding a place to troubleshoot (debugging), which is a very laborious and time-consuming procedure (even if the services are containerized), is added to the dependency search. There are new issues: service discovery, distributed logging, tracing, etc. Managing interface and configuration versions become a major issue.

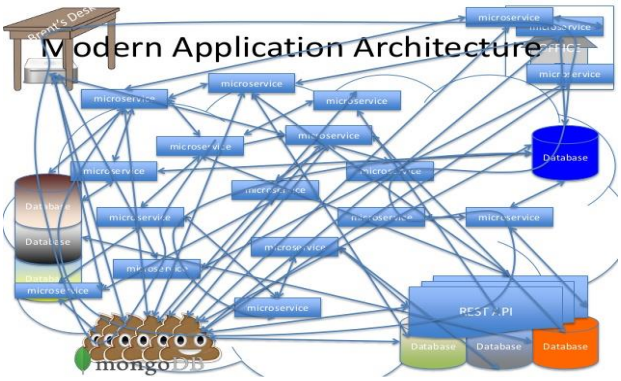

Fig.2. Micro-service architecture structure

Of course, OSGI doesn't solve all problems, but most of them can solve it.

3. Aim of the Work

The purpose of this work is to create a new approach to solve the main and most time-consuming problems of micro-service architecture using a modular architecture.

4. Solution of the OSGI Modular Architecture

OSGI is a tool for creating Java applications from modules. In particular, it can be considered the closest analog of JavaEE, and somehow OSGI containers can execute JavaEE modules (for example, web applications in the form of “War”), and on the other hand, many JavaEE containers contain OSGI inside as a means of implementing modularity for themselves. Thus, JavaEE and OSGI are compatible and complementary products.

An important part of any modular system is the definition of the module itself. In the case of OSGI, the module is called a bundle (Figure 3) and is well known to all developers of the jar archive with some additions. By JavaEE, bands can export and import services that are essentially class methods (ie, the service is the interface or all public class methods).

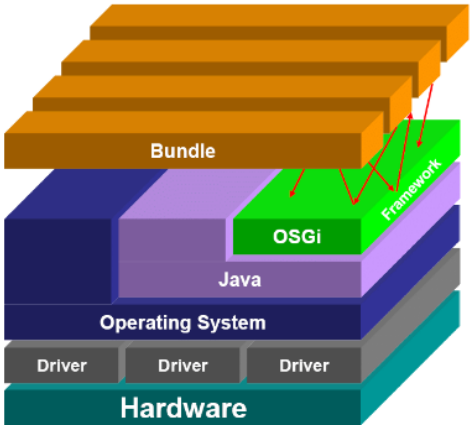

Fig.3. Structure of the modular system

The OSGi runtime uses a Java class load mechanism to implement a container for modular units (packages). Many application servers are implemented using OSGi as the basis. The OSGi package is just a JAR file that contains source code, metadata, and resources. The package may provide various services and components to run OSGi. OSGi runtime allows you to install, start, stop, update, and delete packages without requiring a reboot. The OSGi Core specification is a layered architecture that determines what is supported at runtime. Each layer defines some functionality supported by the runtime program and packages (Figure 4).

Packages can export parts for use by other packages or import packages exported by other packages. This dependency mechanism is called the master protocol and is provided with a modular OSGi layer. Packages can publish services at runtime and use services already published at runtime. This dependency mechanism provides an OSGi service level (Figure 5).

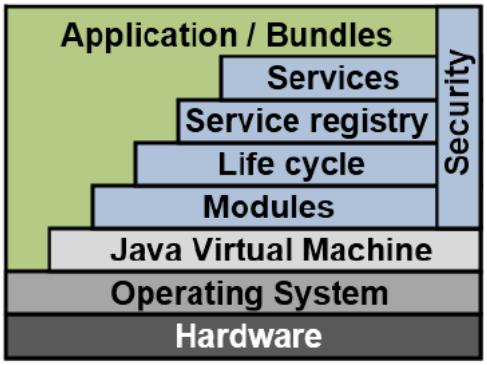

Fig.4 OSGI Core architecture

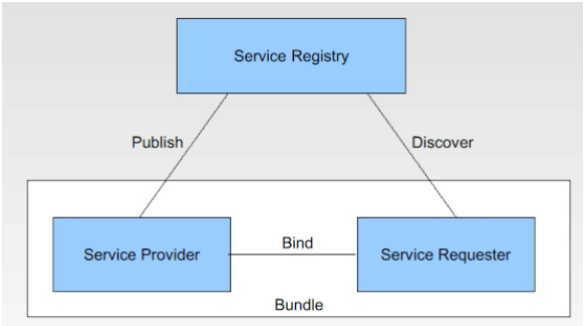

Fig.5. OSGI Service Level Structure

This functionality allows you to implement individual micro-services that will be broken down into separate self-container modules inside, enabling each to evolve over time without combining all micro-services together. Such a system will schematically have the form shown in Figure 6.

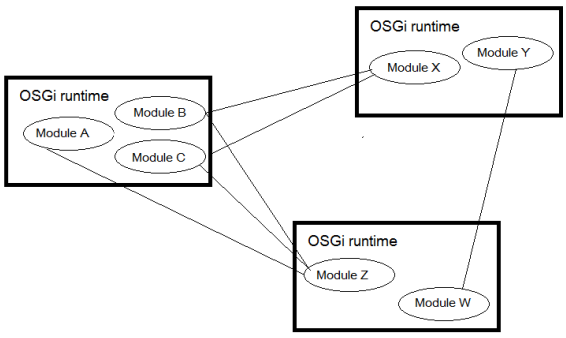

Fig.6. Implementation of micro-services using OSGI scheme

Examples of such complex systems that can be implemented are:
• IoT system (where each device integration is an OSGi micro-service, divided into modules that implement different device functions);
• Complex supply chain (i.e., a system of flight cargo deliveries);
• Complex factory process (i.e. for a car or aircraft factory).
Generally speaking, the concept of micro-services comes down to independently deployed work-oriented units (or closely related groups). They should be able to communicate with lightweight protocols and can be developed, tested and scaled independently.
This means that micro-services directly use system modulation. In this sense, OSGi can be viewed as a micro-service-oriented framework that works in a JVM environment.
Also, OSGI's great advantage is version control. Some Java critical and intermediate applications have stringent technological requirements. Others should support hot deployments in order not to disrupt running services. Still, others should be able to work with different versions of the same package to support legacy external systems.
OSGi platforms are a viable solution to support such requirements. Applications or components that come as deployment packages can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot; Java package management/classes are detailed. Application lifecycle management is implemented through APIs that allow remote management policy downloads. The service registry allows packages to identify new services or delete services and adapt them accordingly. OSGi specifications went beyond the initial focus of gateways and now used in applications ranging from mobile phones to the open-source Eclipse IDE. Other applications include automobiles, industrial automation, building automation, PDAs, computing, entertainment, fleet management, and application servers.
This work uses the Karaf development environment, which is an alternative mechanism for creating a distributed OSGi runtime using the Karaf Cellar runtime.

5. Study of OSGI System for IoT

Remote services provide a way to outsource standard OSGi services to external applications through transport mechanisms such as SOAP or REST web services, Apache Aries Remote Service Admin, r-osgi (Remote Services for OSGi), and more.
There are two well-known tools that provide remote service implementation and remote service administrator specifications - the Apache CXF and the one provided by the ECF (Eclipse Communication Framework).
An OSGi subproject with distributed Apache CXF will be used, which implements a REST provider for the remote Aries administrator. Our application will be deployed in Karaf using Felix OSGi runtime (Karaf also supports Equinox OSGi launch). It should be noted that Karaf offers an alternative mechanism for creating distributed OSGi runtime using Karaf Cellar runtime.
The next step is to go to *karaf.apache.org*, download the distribution, and unpack.
The presented system will be implemented through the creation of a data collection service that communicates with the configuration services by obtaining the configuration data and records the completion status after successful data

collection. Each module will be responsible for different devices (sensors) that collect telemetry data on the environment.
For example, the first "Temperature Module" - will be responsible for the temperature and contain a function that is able to work with temperature sensors. By connecting to the main control module, you can collect device connection data from a configuration file - connect to devices, get current temperature status information, and respond with further logic to different metrics. Because deployment requires two separate Karaf modes, the script will be simulated by running the runtime on different HTTP ports (9991 and 9992).
Thus, by downloading the temperature module to a device with thermosensor control and configuring settings (specifying ports and access address of the device) on the main executable module - it is possible to receive data from the peripheral device into the main computing application, which are different modules.
The developed device is shown in Figure 7, and the result of the computer modeling - in Figure 8.

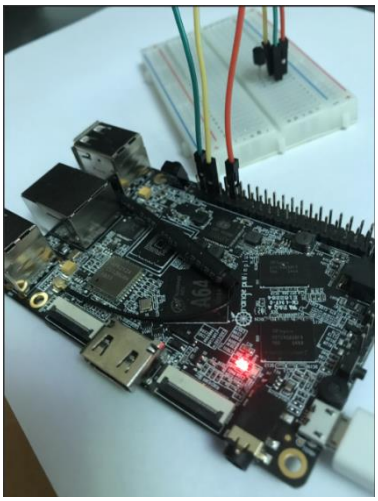

Fig.7. The testing model of the OSGI system for IoT

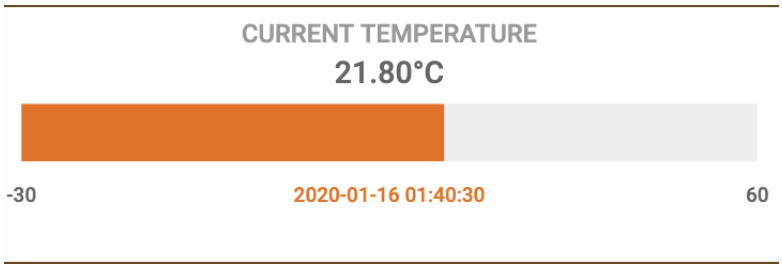

Fig..8 Result of computer modeling
(Data sent to the graphical dashboard using the JSON format and provided API)

6. Conclusions

In this paper, the micro-service architecture and its strong and weak sides are represented. Micro-services are the best choice for complex products that are constantly evolving, despite disadvantages and problems of implementation. It is established that the OSGI system solves the difficulties of testing and support for the confusing structures of complex systems. OSGI is considered on a simple example, which is sufficient to understand the potential of OSGI and the modular structure as a whole. By combining the flexibility of micro-services and the modularity of OSGI, you can achieve a good solution in software development in any field and with any business tasks. If necessary, easy support for the whole structure can be provided. It makes life easier for the designer, the developer and, of course, for the customer. This advantage consists of an easy deployment of lifting modules with a few one-time services and facilitated testing in case of "debugging". While defining the error and testing it by modules, becomes possible to get rid of many services simultaneously.

7. Conflict of Interest

The authors claim that there are no possible financial or other conflicts over the work.

References

[1] Micro-Service Architecture, Medium Corp., 2019. [Online]. Available: https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d.

[2] Building Microservices in OSGi with the Apache Karaf Framework, Exoscale Comp., 2019. [Online]. Available: https://www.exoscale.com/syslog/building-microservices-in-osgi-with-the-apache-karaf-framework/.

[3] Modules Vs. Microservices, O'Reilly Media, Inc., 2017. [Online]. Available: https://www.oreilly.com/radar/modules-vs-microservices/.

[4] The Dynamic Module System for Java, OSGI Alliance, 2020. [Online]. Available: https://www.osgi.org/developer/specifications/

[5] Fowler, Martin. Microservices. 2018. [Online]. Available: https://martinfowler.com/articles/microservices.html

[6] Apache Karaf Microservices article, GitHub, Inc, 2020. [Online]. Available: https://github.com/exoscale-labs/Apache_Karaf_Microservices_article/blob/master/sources/config.api/pom.xml.