# Research Methodology

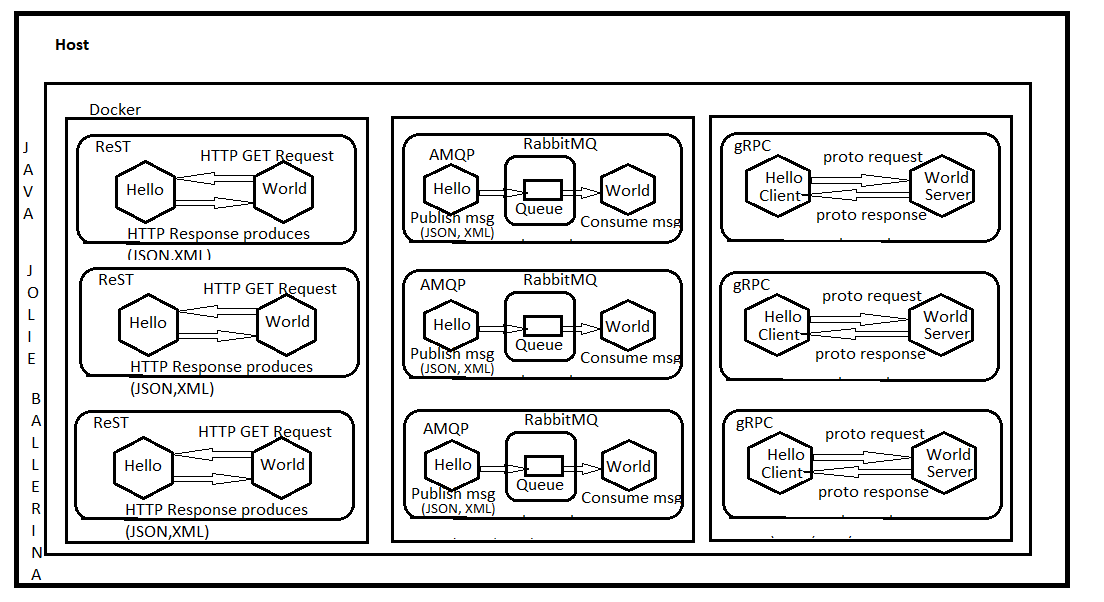
This section discusses the methodology used in this research. There are different variables involved in the research that helps to understand the property of the language and measure it. This section will also cover the coding, deployment and the execution stage of the experiment.

## Experiment

This research uses the experiment design approach to answer the research question. According to Jackson (2014), the experiment design approach “allows a researcher not only to describe and predict but also to establish a cause and effect relationship through manipulation of a variable and control of the situation.” The main foundation of the experimental approach is that the researcher gains control of the environment as much as possible. This is to identify if there is a cause and effect relationship between the variables being studied. In the experimental process, certain factors are selected and deliberately varied in a controlled manner to understand their effects on the response of interest (Wang & Wan, 2009). These variables are called independent variables. The variables that are measured are the dependent variables. These variables are derived from the critical factors for a programming language and integration and these variables will be used to measure the programming language in my research.

This research analyses three different languages being Java, which is widely used for the development of microservices [(Glen, 2018)](https://www.zotero.org/google-docs/?broken=N1mp2I); Jolie a language especially designed for the distributed system to support service oriented architecture [(Montesi et al., 2014)](https://www.zotero.org/google-docs/?broken=Hduwmr); and Ballerina a newly developed language for microservices [(WSO2 Inc, n.d.)](https://www.zotero.org/google-docs/?broken=rs9CbD). The experiment carried out for this research will answer the research question “What programming language is the best for the integration of the microservices?”. This experiment will address what programming language is the best by analysing, empirically, the programs written in that language [(Nanz & Furia, 2015)](https://www.zotero.org/google-docs/?broken=josahc). As this experiment is performed in a controlled fashion, it will help to understand the properties of the programming languages and more reliable data is obtained as the outcome of the controlled experiment [(Nanz et al., 2013)](https://www.zotero.org/google-docs/?broken=S3VQmG).

To understand the properties of the programming language for internal integration between the microservices, the experiment is done on two services communicating with each other. These services are simple services called hello service and world service. World service communicates with hello service using different technology and data types. Figure given below illustrates the complete experiment.



### Control Variable

To understand the impact of the integration on these programming languages, my research uses the controlled experiment methodology. The control is achieved at two different stages of microservices development namely coding and deployment. Also, it is achieved at the time of execution of the microservices.

For the coding of the microservices, the programs for all three languages are developed by myself. Control at the time of coding is achieved by writing the programs in all the language as the replica of each other in terms of functionality. Initial coding is done for microservices in Java programming language. Programs written in Java are as per the tutorial guide given by the official website of different integration technology. For microservices built-in ReST, this research uses (JAX-RS) API given by Oracle [(*Java EE APIs*, n.d.)](https://www.zotero.org/google-docs/?7t9QLe), for microservices using gRPC, this research follows the tutorial guide for Java by gRPC official website [(*Basics Tutorial*, n.d.)](https://www.zotero.org/google-docs/?DtRPNS) and for AMQP the microservices are built using the official website of RabbitMQ that uses AMQP protocol [(*RabbitMQ Tutorial - “Hello World!” — RabbitMQ*, n.d.)](https://www.zotero.org/google-docs/?i3yDY3). These microservices built using Java programs serve as the baseline for the microservices in the other two languages. The microservices written in the other two languages are the same in functionality as Java microservice. There are different sets of libraries that each language uses to achieve a given functionality, and these libraries and functions are used as per the official documentation of that language. The libraries used in Ballerina code, to achieve the functionality as that of Java microservices are as per the documentation of Ballerina for each integration technology [(Inc, n.d.)](https://www.zotero.org/google-docs/?VGvOv4). Similarly, the libraries used in Jolie are as per Jolie [(*Jolie/Examples*, 2016/2020)](https://www.zotero.org/google-docs/?nNIxkg). Thus, enforcing the high control at the development stage of the coding.

For the deployment of the microservices, this research uses Docker containers. “A docker container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another” [(*What Is a Container?*, n.d.)](https://www.zotero.org/google-docs/?wJLHXB). Microservices and the containers are a natural pair that helps the former to achieve higher levels of modularity, code reuse, reproducibility and fine-grained scaling of a distributed system [(Stubbs et al., 2015)](https://www.zotero.org/google-docs/?broken=RZ07Hm). Container technology, and Docker in particular, are having a profound impact on microservices. Thus, making Docker the first choice for the research to run the microservices. The control for the deployment stage is achieved by deploying all the services in the common same docker environment.

As the integration of microservices means the communication calls between the services across the network. And with the network comes the response time delay and latency. Network failures are inevitable and given the nature of the network, there will always be outliers [(Newman, 2015)](https://www.zotero.org/google-docs/?SlTidL). It is important to achieve control of the network. To avoid this, the research uses docker compose, a tool that is used to define and run multiple docker applications [(*Overview of Docker Compose*, 2020)](https://www.zotero.org/google-docs/?jXPlJB). Docker compose sets up a single network for the apps. Each container for a service joins the default network and is both reachable by the containers on that network [(*Networking in Compose*, 2020)](https://www.zotero.org/google-docs/?JVlQaa). This will ensure that every service is in the reach and the delay in the response will be minimal.

Finally, the control at the time of execution is already achieved as all the services are deployed in the same environment. The additional configuration is done by allocating the same memory for each service inside the containers.

### Independent Variables

Independent variables help to understand the impact of the integration on the programming language. In the experimental process, factors like integration technology and the integration dataare selected and deliberately varied in a controlled manner to understand their effects on the languages[(Wang & Wan, 2009)](https://www.zotero.org/google-docs/?broken=n628y1). The research focuses on the integration of the microservices, thus the effect of integration can be determined by testing the language for different integration technologies and integration data. The research will talk in depth about each of the independent variables.

#### Integration Technology

As the microservices development approach follows smart endpoint and dumb pipeline philosophy, the best strategy for building microservices from the integration standpoint is to use technology that is language agnostic and simple to use that adheres to the philosophy of microservices development[(Fowler, 2010; Newman, 2015)](https://www.zotero.org/google-docs/?broken=fqVnM4). According to [Fowler (2010)](https://www.zotero.org/google-docs/?broken=ocddrN) request-response enables to achieve the integration in the simplest possible way by making use of principles like ReST which is the simple protocol to use for web services. Second approach according to him is the messaging via queues over the lightweight messaging, as simple messaging implementations such as RabbitMQ provide a reliable asynchronous fabric for communication. Thus the smart still lives at the endpoint that are producing and consuming the messages rather than the infrastructure chosen which is typically dumb.

[Schwarz and Riehle (2020)](https://www.zotero.org/google-docs/?broken=XJ9RRb) studied eight different research that had citations of over 50 - 950 generated a comparative model for different integration technology for the internal integration of microservices. This model suggested the technologies that are best fit, namely ReST, RPC, and Messaging. My research uses the findings of the former research and uses the same technologies suggested, to test against each language selected for comparison. Each technology is discussed in detail in the below literature.

##### Representational State Transfer (ReST)

ReST is the set of principles for the request response based service development and is very popular in the web service development community [(Li & Chou, 2011)](https://www.zotero.org/google-docs/?broken=e3GEGB). According to [Richards (2015)](https://www.zotero.org/google-docs/?broken=85KH8v), microservices architectures rely on REST as their primary remote-access protocol. Others like [Newman (2015)](https://www.zotero.org/google-docs/?broken=DmpSRT) and [Fowler (2010)](https://www.zotero.org/google-docs/?broken=yVURfp) also recommend ReST over HTTP for service to service interactions in the microservices architecture. One of the key things for this research is the use of the technology that is loosely coupled with the programming language, as that also serves as an important factor for choosing any language for the development of microservices from the integration viewpoint. ReST is totally language agnostic and thus testing the language for ReST is important.

##### Remote Procedure Call (RPC)

RPC is the request response protocol where the request is made by the client to execute on the remote system. The client can make this remote call directly as if it was made from the local object. There are different implementations of RPC, but the one that this research uses considers two factors, being technology agnostic and the implementation recommended for microservices architecture, one of them is gRPC [(Gan et al., 2019; *GRPC*, n.d.)](https://www.zotero.org/google-docs/?broken=se2MHp). gRPC is an open source framework for remote procedure calls developed by Google. gRPC uses protocol buffers as the service definition language and the format to exchange the data.

##### Messaging

Messaging follows publisher subscriber design protocol, where the publisher service sends the message to a queue or a topic and the subscriber service consumes the message from the ques after the publish event is done. Like RPC, there are different implementations and standards for efficient messaging. However, this research makes use of the latest protocol available for messaging being Advanced Messaging Queue Protocol (AMQP) and the implementation used for AMQP is RabbitMQ as it is one of the open source frameworks available, and is also suggested in the official website of AMQP [(*Products and Success Stories | AMQP*, n.d.)](https://www.zotero.org/google-docs/?broken=Z3FqOF)

#### Integration Data

There are different formats to represent the data, namely JSON, XML, CSV, binaries and so, that can be exchanged between the microservices [(Newman, 2015)](https://www.zotero.org/google-docs/?iYNkoA). This research will use the popular or recommended format(s) for a given integration scenario of the experiment.

##### JSON (JavaScript Object Notation)

JSON is a lightweight data-interchange textual format [(*JSON*, n.d.)](https://www.zotero.org/google-docs/?wyLTsO). It is completely language independent and is a collection of name-value pairs. The format that has grown its popularity with time is JSON because of its simplicity that makes it easier to consume [(Newman, 2015)](https://www.zotero.org/google-docs/?bItYs6). This research will use JSON as one of the integrations of data types for the ReST and messaging integration type, to test the languages.

##### XML (Extensible Markup Language)

XML is a markup language for documents containing the structured information [(*A Technical Introduction to XML*, n.d.)](https://www.zotero.org/google-docs/?UlrYGe). Markup language is a mechanism that is used to identify the structures in a document. Like JSON, XML is completely language independent. Unlike HTML (HyperText Markup Language), XML allows defining the tags and the structural relationships between them. It is the second popular language used to represent the data and is used for integration [(Newman, 2015)](https://www.zotero.org/google-docs/?YuabKr). This research will use XML as the second integration data type for ReST and messaging integration type to test the languages.

##### Protocol buffers

Protocol buffer is the data represented in the form of messages and stored in the file with extension proto [(*Protocol Buffers*, n.d.)](https://www.zotero.org/google-docs/?hbFPbS). Like XML, protocol buffers allow us to define the structured data but they are smaller, faster and simpler than XML. Protocol buffers are also language-neutral and they are the recommended data format for gRPC integration type.

### Dependent Variable

Dependent variables are the factors that help to measure the language. The list of dependent variables is derived from different factors that include the factors critical for the language and the linguistic side of integration. The factors important for language is derived from similar research that compares multiple programming languages. One such research is conducted by Nanz & Furia (2015) that compares 20 different programming languages and the variable used by them is used in my research. These variables include verbosity of the source code, size of the executables and the execution time of the code (Nanz & Furia, 2015). The other variables for the integration are as per Guidi et al theory (2017) he has listed the variables important from the language point of view for the integration of the services. According to him, Interfaces or Services should be the first-class citizen of the language. And besides Interfaces, Ports and Workflows are also important for microservices. In the following section, I will talk in detail about each variable used.

#### Verbosity

Term Verbose means explaining in words more than needed. According to the developer community StackOverFlow, majority of the developers supported that a good code should be easy to comprehend at a glance (Why Is Verbosity Bad for a Programming Language?, n.d.). Also, according to Gupta (2004), excess verbosity can cause the programmers at the beginner level to miss the conceptual learning in order to get a grasp of the huge code. When a programming language is less verbose it also means that it is capable to do a task in less number of lines which also saves the development time with respect to writing code (Krishna et al., 2017). Considering all these factors, the line of code for each program will be measured and the languages will be evaluated on the numbers obtained from the experiment.

#### Size

Size of the executable has a great impact on the deployment of the code. Reduced size of executable has several benefits like faster deployment, smaller disk size, small server space, reduced cost of hardware and easy management of code (Cepa, 2005). The factors that influence the size of the code is the use of third-party libraries or the external frameworks to support a development. However, the same can be avoided if the language development environment itself provides those libraries. (How Ballerina  Is Different From Other Programming Languages - DZone Integration, n.d.). In this study, the executable built for every language will be evaluated against the size of the executables for every integration scenario.

#### Execution Time

It is a well-known fact that faster applications are preferred over slow performing applications. Execution time forms an important parameter to measure the performance of any application. A new technology is always assessed on its performance and execution time is one of the parameters to measure the performance (Pongnumkul et al., 2017). In this research, the execution time for every program will be recorded and compared for all the languages.

#### Debug

Debugging is useful in various stages of software development like design where the error made in writing the code is identified, secondly, it is also helpful in the later stages of testing. Debugging is also helpful in the error diagnostics in the production environment (Cheng et al., 2017).

#### Interfaces and Port

Microservices are deployed as the “black boxes” whose implementation details are hidden to support modular programming. The details of the functionality of the services are provided by the interfaces. Interfaces describe the set of operations that can be remotely invoked. Thus, interfaces become the first-class citizen of the microservices. Thus, it is important for a microservices language to provide the set-theoretical operators like union and intersection that can work with the interfaces (Guidi et al., 2017).

Besides Interfaces, ports are important as a communication port describes how the services are made available to the network. Each service may be equipped with many ports thus ports should be separate from the implementation of the service. Hence, the language for microservices should provide the capability to separate the port from the implementation of the service (Guidi et al., 2017).

This research will identify what language treats Interfaces and Ports as the first-class citizen. This can be seen from the artifact built in all three languages.

#### Graphical View Support

It is a common practice to draw a sequence diagram for the complex integration scenarios (Weerawarana et al., 2018). Also, there are different workflow languages like BPEL being used by industry experts to simplify the complex integration scenarios (Guidi et al., 2017). Thus, it is helpful for the microservices language to provide visual support or workflow development options to support the development of microservices. This research will test the three languages for its capability to support visual designing. This can be seen from the artifact built in all three languages.

### Experiment Stages

Before coding began in the first stage of the experiment, different repository options to store the code in the version control system like Git were considered. All the code for this experiment is pushed in the master branch of my Git account with public access. The link to access my code is - <https://github.com/ushakotian/Ballerina_Java_Jolie-/tree/master/workspace>

#### Coding

This section will include my approach to coding for each language. I will talk about the development setup, the IDE I used, all the libraries that I used for coding and their version. I will also talk about the blockers I faced while coding each language and the support available. I will cover dependent variables like verbosity, debug, interfaces and ports as the first class citizen and my strategy to measure them in the coding stage of the experiment. Finally, I will touch base on my experience of coding each language. The approach for this experiment was to first write the coding in Java for every integration scenario, and then do the same in the other two languages. The following section will cover how the coding was done in each integration scenario.

###### Java

**Installation**

The first step was to set up the prerequisites as per the tutorial guide. This included Java Enterprise Edition 8 Software Development Kit (Java EE 8 SDK), Net beans Integrated Development Environment (NetBeans IDE), Glassfish Server Open Source Edition that is embedded in the NetBeans IDE and Apache Maven [(*Required Software*, n.d.)](https://www.zotero.org/google-docs/?zE9fzh).

###### Ballerina

**Installation**

The first step to set up the development environment for ballerina is to install the latest version of Ballerina. The latest version in today’s date is ballerina 1.2.8 <https://ballerina.io/downloads/>. I downloaded the Ballerina installer for windows. The next step was to set up the Visual Studio Code, IDE to write Ballerina programs <https://ballerina.io/learn/setting-up-visual-studio-code/>. I followed the above to set up the VS Code, by installing the VS Code for Windows and then installing the VS Code extension for Ballerina.

###### Jolie

**Installation**

The first step to set up the development environment for Jolie is to install Jolie in the system. The latest version of Jolie as of today is 1.9.1. I followed the official documentation to install jolie <https://www.jolie-lang.org/downloads.html>. The IDE recommended for Jolie is Visual studio Code, after installing VS Code the next step is to install the plugin for Jolie.

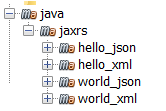
##### ReST

###### Java

I followed the latest tutorial guide given by Oracle to develop the ReST service in Java. The latest version of Java Enterprise Edition today is Java EE 8.

**Project structure**

I created four maven services projects- hello\_json, hello\_xml, world\_json and world\_xml for the experiment. These service projects are children of a project called jaxrs and jaxrs is the child of java. Below is the structure of the project.



All these projects have a common groupid. The dependency for javaee-api-8.0.jar is included in the pom.xml of the java project. The project downloads the jar that has the implementation for jaxrs, required for building ReST. Below is the declaration, packaging and dependency configuration added in the parent pom file -

<groupId>org.glassfish.javaeetutorial</groupId>

<artifactId>javaeetutorial</artifactId>

<version>8.1-SNAPSHOT</version>

<packaging>pom</packaging>

<dependencies>

<dependency>

<groupId>javax</groupId>

<artifactId>javaee-api</artifactId>

<version>8</version>

<scope>provided</scope>

</dependency>

</dependencies>

Relationship between each project was defined in the pom.xml of the individual project by configuring the parent information in the pom file in below the given format.

**Parent configuration in jaxrs pom file**

<parent>

<groupId>org.glassfish.javaeetutorial</groupId>

<artifactId>javaeetutorial</artifactId>

<version>8.1-SNAPSHOT</version>

</parent>

**Parent configuration in the pom file of each service project**

<parent>

<groupId>org.glassfish.javaeetutorial</groupId>

<artifactId>jaxrs</artifactId>

<version>8.1-SNAPSHOT</version>

</parent>

I created a maven project with the following folder structure



After the setup was done, the next step for the experiment was to write the code. I had to write four ReST services in all, two hello services producing output “Hello” in the format json and xml, while the other two world services consuming the respective hello service for json and xml and producing the final output with “hello” and “world” together in json or xml format.

**hello\_json service**

I created a class called Hello.java inside the package javaeetutorial.hello. This class has a httpmethod called get with the uri path “hello”.This http get method produces the media type called application/json. The get method creates a java String object called message that has the json data {“message1”:”hello”}

**hello\_xml service**

I created a class called Hello.java inside the package javaeetutorial.hello. This class has a httpmethod called get with the uri path “hello”. This http get method produces the media type called application/xml. The get method creates a java String object called message that has the json data “<message1>hello</message1>”

**world\_json service**

I created a class called World.java inside the package javaeetutorial.world. This class has a http method called get with the uri path “world”. The http get method produces the media type application/json. The get method makes the call to the hello\_json service and reads the value in the response returned by the hello\_json service for the key “message1” and creates a java String object called “finalmessage” by appending this to the string “world”. To convert the string value to the json object in java, the library used for json in java is as given in the official website of json[(*JSON*, n.d.)](https://www.zotero.org/google-docs/?9SHz3x). The dependency for this jar is added in the pom.xml of world\_json service with below configuration.

<dependency>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

<version>20200518</version>

</dependency>

The final message will look like this {"finalmessage": "hello world"}

**world\_xml service**

I created a class called World.java inside the package javaeetutorial.world. This class has a http method called get with the uri path “world”. The http get method produces the media type application/xml. The get method makes the call to the hello\_xml service and reads the value in the response returned by the hello\_xml service for the key “message1” and creates a java String object called “finalmessage” by appending this to the string “world”. To convert the string value to the xmlobject in java, the library provided for xml in java is as given in the official website of Java [(*Java EE*, n.d.)](https://www.zotero.org/google-docs/?RsoA13). The dependency for this jar is added in the pom.xml of world\_json service with below configuration.

<dependency>

<groupId>javax.xml.parsers</groupId>

<artifactId>jaxp-api</artifactId>

<version>1.4.5</version>

</dependency>

The final message will look like this <finalmessage>"hello world</finalmessage>

**Challenges**

I had one issue at the time of installation and building the code in the local environment. The first issue was with the NetBeans IDE installation, despite the correct installation of JDK on my system, NetBeans was unable to point to it and complained of not having the right JDK in the error message. This was resolved by explicitly specifying the JDK location while running the NetBeans installer with the command “ --javahome <path-to-jdk>” . Found the solution for this problem in the FAQ section given in the wiki of the NetBeans [(*FaqInstallJavahome - NetBeans Wiki*, n.d.)](https://www.zotero.org/google-docs/?IIm47X).

I faced an issue while building the code from the netbeans IDE and after following the instructions given in one of the StackOverflow blogs , this issue was resolved. <https://stackoverflow.com/questions/58411279/java-with-maven-wouldnt-build-cannot-run-program-cmd-malformed-argument-has>. Second issue I faced was while running the services in the embedded glassfish server. The issue was with the maven again not being able to point to the correct glassfish server. This was resolved by changing the configuration of the maven to point to the correct server. And lastly, there was an issue in the maven bat called maven.bat file being missing. This was also resolved by following the instruction in one of the StackOverflow blogs.

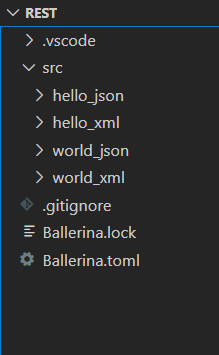
<https://stackoverflow.com/questions/32706000/error-in-maven-build-mvn-bat-not-recognized>.

###### Ballerina

I referred to the Ballerina by example which is an incremental way of learning to code in Ballerina, that is provided by Ballerina itself.

**Project Structure**

For this experiment, I created a single ballerina project called REST by running the ballerina command to create a project. I run the command “ballerina new REST” inside the VS Code built-in command line terminal. After the project was created, I ran the command to add the module for four different services in the REST project. Below snippet is the project snippet for my experiment.



Unlike Java, all the dependency libraries required to write the ReST services are not required to be downloaded or mentioned in the configuration file like pom.xml. Ballerina has a centralised repository where it maintains all the libraries required for different modules and can be available by writing an import statement in the .bal files.

**hello\_json service**

I created a module for the hello\_json service by running the command “ballerina add hello\_json”. Ballerina created the folder with the name hello\_json and by default created the main.bal file. I renamed this file to hello.bal to follow the naming convention for my experiment.

In the hello.bal, I created a service with the name “hello” and the base-path “hello”. This service had a http method called get and the uri-path was “hello”. The get method produces the content type called application/json. The message created in the get method is a ballerina json object with data as {message1: "hello"}. The only import statement required was “ballerina/http”.

**hello\_xml service**

I created a module for the hello\_xml service by running the command “ballerina add hello\_xml”. Ballerina created the folder with the name hello\_xml and by default created the main.bal file. I renamed this file to helloxml.bal to follow the naming convention for my experiment.

In the helloxml.bal, I created a service with the name “helloxml” and the base-path “hello”. This service had a http method called get and the uri-path was “hello”. The get method produces the content type called application/xml. The message created in the get method is a ballerina xml object with data as <message1>hello</message1>. The only import statement required was “ballerina/http”.

**world\_json service**

I created a module for the world\_json service by running the command “ballerina add world\_json”. Ballerina created the folder with the name world\_json and by default created the main.bal file. I renamed this file to world.bal to follow the naming convention for my experiment.

In the world.bal, I created a service with the name “world” and the base-path “world”. This service had a http method called get and the uri-path was “world”. The get method produces the content type called application/xml with the uri path “world”. The http get method produces the media type application/json. The get method makes the call to the hello\_json service and reads the value in the response returned by the hello\_json service for the key “message1”. As ballerina provides the inbuilt support for json data type, there is no additional conversion of message required. The value from the response is simply fetched by using the json inbuilt functionality as given in the tutorial <https://ballerina.io/learn/by-example/json-objects.html>. The final message created in the get method is a ballerina json object with data as {finalmessage:”hello world"}. The only import statement required was “ballerina/http”.

**world\_xml service**

I created a module for the world\_xml service by running the command “ballerina add world\_xml”. Ballerina created the folder with the name world\_xml and by default created the main.bal file. I renamed this file to worldxml.bal to follow the naming convention for my experiment.

In the worldxml.bal, I created a service with the name “world” and the base-path “world”. This service had a http method called get and the uri-path was “world”. The get method produces the content type called application/xml with the uri path “world”. The http get method produces the media type application/xml. The get method makes the call to the hello\_xml service and reads the value in the response returned by the hello\_xml service for the key “message1”. As ballerina provides the inbuilt support for xml data type, there is no additional conversion of message required. The value from the response is simply fetched by using the xml inbuilt functionality as given in the tutorial <https://ballerina.io/learn/by-example/xml-access.html>. The final message created in the get method is a ballerina xml object with data as <finalmessage>

hello world</finalmessage>. There are two import statements used here “ballerina/http” and ballerina/lang.'xml.

###### Jolie

**Installation**

The first step to set up the development environment for Jolie is to install Jolie in the system. The latest version of Jolie as of today is 1.9.1. I followed the official documentation to install jolie <https://www.jolie-lang.org/downloads.html>. The IDE recommended for Jolie is Visual studio Code, after installing VS Code the next step is to install the plugin for Jolie.

**Project Structure**

Unlike Java and Ballerina, there is no project structure for Jolie. A service in jolie can be defined in a single file with extension .ol. I created four files each defining the respective service for hello-json, hello-xml, world-json,world-xml. Like Ballerina, Jolie also does not need a dependency jar to be downloaded or mentioned separately in the config files.

**hello\_json service**

I created a jolie file called Hello.ol to write the code for hello\_json service. In jolie, the service is defined as the interface. The hello-json service is defined as the interface with name HelloAPI. The service has one http method called get that produces the content type of application/json. The get method creates the object of type “message”. The type message has a property called message1 that is of type string. In the get method, the value of message1 of type message is set to “hello”. Unlike Java and Ballerina, there is no single import statement in the code.

**hello\_xml service**

I created a jolie file called HelloXML.ol to write the code for hello\_xml service. The hello-xml service is defined as the interface with name HelloAPI. The service has one http method called get that produces the content type of application/xml. The get method creates the object of type “message”. The type message has a property called .message1 that is of type string. In the get method, the value of .message1 of type message is set to “hello”. Unlike Java and Ballerina, there is no single import statement in the code.

**world\_json service**

I created a jolie file called World.ol to write the code for world\_json service. The world-json service is defined as the interface with the name WorldAPI. The service has one http method called get that produces the content type of application/json. The get method calls the hello-json service in jolie object of type message. We get the value of key “message1”, we create the object of type finalmessage with the property of name finalmessage and type string. The value assigned to the property is “hello world”. There are no import statements.

**world\_xml service**

I created a jolie file called WorldXML.ol to write the code for world\_xml service. The world-xml service is defined as the interface with the name WorldAPI. The service has one http method called get that produces the content type of application/xml. The get method calls the hello-xml service in jolie object of type message. We get the value of key “.message1”, we create the object of type finalmessage with the property of name finalmessage and type string. The value assigned to the property is “hello world”. There are no import statements.

**Challenges**

There was one issue in installing Jolie in the local environment with the default setup of the Jolie installer. This was resolved by changing the launcher location of the installer to the binary location.

##### AMQP

For testing the languages for the second integration type that is messaging using the AMQP protocol, I wrote a hello service as a ReST service that produces the “hello” message in json and xml data format. This message is then published to the RabbitMQ queue which is then consumed by the world service. World service which is a simple runnable java application constantly listens to the same RabbitMQ queue where the hello service publishes the message. The world service then appends the message “world” to the message received and prints the final message - “hello world” in json and xml format.

###### Java

For writing the program in java for AMQP, I followed the tutorial given by RabbitMQ [(*RabbitMQ Tutorial - “Hello World!” — RabbitMQ*, n.d.)](https://www.zotero.org/google-docs/?r451Jv). However, the tutorial was to code a simple runnable java application. To code the example for the given scenario of my experiment, I had to make the changes accordingly. As the first service that is hello is the ReST service, I referred to the same tutorial to build the ReST service as in the previous example.

**Installation**

The prerequisites for this scenario is RabbitMQ installed on the local system. For RabbitMQ installation I downloaded the docker image of RabbitMQ. The other prerequisites are the same as that for the ReST service for java(<https://registry.hub.docker.com/_/rabbitmq/>).

**Project Structure**

I created four maven projects - hello, hello\_xml, world and world\_xml. The glassfish server and the ReST api dependency was resolved by the jaxrs project in a similar way I did for the earlier ReST scenario for java. To add the dependency for RabbitMQ in all the projects, the pom contained the following configuration-

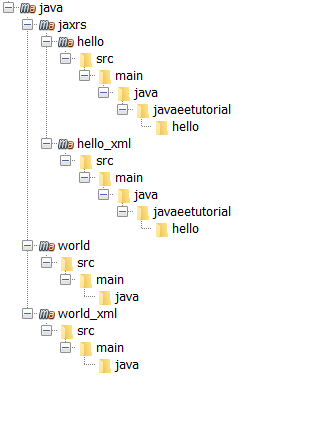
<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

<version>5.7.1</version>

</dependency>



**hello\_json service**

I created a class called Hello.java inside the package javaeetutorial.hello. This class has a httpmethod called get with the uri path “hello”.This http get method sends the message to the RabbitMQ broker in the format json. The get method creates a java String object called message that has the json data {“message1”:”hello”}. The connection is established with the RabbitMQ broker when the get method is called, and this message is published to the queue whose name is “hello-json”.

**hello\_xml service**

I created a class called Hello.java inside the package javaeetutorial.hello. This class has a httpmethod called get with the uri path “hello”. This http get method sends the message in the format xml. The get method creates a java String object called message that has the json data “<message1>hello</message1>”. The connection is established with the RabbitMQ broker when the get method is called, and this message is published to the queue whose name is “hello-xml”.

**world\_json service**

I created a class called World.java that has a main method that connects to the RabbitMQ broker. This method consumes messages from queue “hello-json” when the queue receives the message. After the message is received by the method, I am reading the message and converting it to the json object and getting the value for the key “message1”. To convert the string value to the json object in java, the library used for json in java is as given in the official website of json[(*JSON*, n.d.)](https://www.zotero.org/google-docs/?8ff6iL). The dependency for this jar is added in the pom.xml of world\_json service with below configuration.

<dependency>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

<version>20200518</version>

</dependency>

The final message will be print like this “finalmessage = hello world".

**world\_xml service**

I created a class called World.java that has a main method that connects to the RabbitMQ broker. This method consumes messages from queue “hello-json” when the queue receives the message. After the message is received by the method, I am reading the message and converting it to the json object and getting the value for the key “message1”. To convert the string value to the xmlobject in java, the library provided for xml in java is as given in the official website of Java [(*Java EE*, n.d.)](https://www.zotero.org/google-docs/?kKlcWJ). The dependency for this jar is added in the pom.xml of world\_json service with below configuration.

<dependency>

<groupId>javax.xml.parsers</groupId>

<artifactId>jaxp-api</artifactId>

<version>1.4.5</version>

</dependency>

The final message will be print like this “finalmessage = hello world".

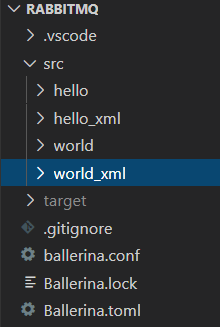
###### Ballerina

**Installation**

t

**Project Structure**

For this experiment I created a single ballerina project called RABBITMQ by running the ballerina command to create a project. I run the command “ballerina new RABBITMQ” inside the VS Code built in command line terminal. After the project was created, I ran the command to add the module for four different services in the RABBITMQ project. Below snippet is the project snippet for my experiment.



Unlike Java, all the dependency libraries required to write the Messaging services are not required to be downloaded or mentioned in the configuration file like pom.xml. Ballerina has a centralised repository where it maintains all the libraries required for different modules and can be available by writing an import statement in the .bal files.

**hello\_json service**

I created a module for the hello\_json service and named it hello. I did this by running the command “ballerina add hello”. Ballerina created the folder with the name hello and by default created the main.bal file. I renamed this file to hello.bal to follow the naming convention for my experiment.

In the hello.bal, I created a service with the name “hello” and the base-path “hello”. This service had a http method called get and the uri-path was “hello”. This http get method sends the message to the RabbitMQ broker in the format json. The get method creates a java String object called message that has the json data {“message1”:”hello”}. The connection is established with the RabbitMQ broker when the get method is called, and this message is published to the queue whose name is “hello-json”.

The import statements required are “ballerina/http” and “ballerina/rabbitmq”.

**hello\_xml service**

I created a module for the hello\_xml service by running the command “ballerina add hello\_xml”. Ballerina created the folder with the name hello\_xml and by default created the main.bal file. I renamed this file to helloxml.bal to follow the naming convention for my experiment.

In the helloxml.bal, I created a service with the name “helloxml” and the base-path “hello”. This service had a http method called get and the uri-path was “hello”. This http get method sends the message to the RabbitMQ broker in the format xml. The get method creates a java String object called message that has the xml data <message1>hello</message1>. The connection is established with the RabbitMQ broker when the get method is called, and this message is published to the queue whose name is “hello-xml”.

The import statements required are “ballerina/http” and “ballerina/rabbitmq”.

**world\_json service**

I created a module for the world\_json service and named it world. I did this by running the command “ballerina add world”. Ballerina created the folder with the name world and by default created the main.bal file. I renamed this file to world.bal to follow the naming convention for my experiment.

In the world.bal, I created a service with the name “world”. This service has a listener that connects to the RabbitMQ broker and listens to the queue called “hello-json”. Upon receiving a message on the queue, the function called OnMessage is invoked. In this method, the value of key “message1”. As ballerina provides the inbuilt support for json data type, there is no additional conversion of message required. The value from the response is simply fetched by using the json inbuilt functionality as given in the tutorial <https://ballerina.io/learn/by-example/json-objects.html>. The final message will be print like this “finalmessage = hello world".

The only import statement required was “ballerina/rabbitmq”.

**world\_xml service**

I created a module for the world\_xml service by running the command “ballerina add world\_xml”. Ballerina created the folder with the name “world\_xml” and by default created the main.bal file. I renamed this file to world.bal to follow the naming convention for my experiment.

In the world.bal, I created a service with the name “world\_xml”. This service has a listener that connects to the RabbitMQ broker and listens to the queue called “hello-xml”. Upon receiving a message on the queue, the function called OnMessage is invoked. In this method, the value of key “message1”. As ballerina provides the inbuilt support for xml data type, there is no additional conversion of message required. The value from the response is simply fetched by using the xml inbuilt functionality as given in the tutorial <https://ballerina.io/learn/by-example/xml-access.html>. The final message will be print like this “finalmessage = hello world".

The only import statement required was “ballerina/rabbitmq”.

##### gRPC

###### Java

To test the gRPC scenario, my experiment has two services: hello and world. Like the other two scenarios, the world service sends the hello request but with the message “world” to the hello service. Unlike the other two scenarios where the final message was generated by the world service, in this example, the final message is generated by the hello service with hello appended and sent back to world service.

**Installation**

The prerequisite for this scenario is java8.

**Project Structure**

I created two maven projects following the tutorial guide for grpc-java(<https://www.grpc.io/docs/languages/java/quickstart/> ) namely hello and world. The dependency for both the project are added in the pom.xml using below configuration -

<dependency>

<groupId>io.grpc</groupId>

<artifactId>grpc-netty</artifactId>

<version>1.7.0</version>

</dependency>

<dependency>

<groupId>io.grpc</groupId>

<artifactId>grpc-protobuf</artifactId>

<version>1.7.0</version>

</dependency>

<dependency>

<groupId>io.grpc</groupId>

<artifactId>grpc-stub</artifactId>

<version>1.7.0</version>

</dependency>

**hello service**

I created a class called HelloService, a simple java class with the main method. The main method starts up the grpc service application for hello. This service has the hello method that is available for other services for use. This method accepts HelloRequest and sends back HelloResponse. The hello method prefix “hello” to the message received in the HelloRequest.

**world service**

I created a class called World, a simple java class with the main method. The method makes the request to the hello service by creating the object of HelloRequest with the message “world”. After getting the HelloResponse back from the hello service, the response is printed on the console.

###### Ballerina

**Installation**

T

**Project Structure**

For this experiment, I created a single ballerina project called grpc by running the ballerina command to create a project. I run the command “ballerina new grpc” inside the VS Code built-in command line terminal. After the project was created, I ran the command to add the module for four different services in the grpcsproject. Below snippet is the project snippet for my experiment.

**hello service**

I generated the code required for the hello service using the command - ballerina grpc --input hello.proto --mode service --output helloservice. .This command generated the helloservice folder and created the ballerina file “hello\_sample\_service.bal” that contains the definition of HelloRequest, HelloResponse provided in the hello.proto. I copied this file inside the hello module.

I created the new ballerina file called “hello.bal” that defined the implementation for the hello method of the hello service. Inside the method hello, I create the final message from the HelloRequest received with the prefix “hello” and send it back as the HelloResponse to the caller.

**world service**

I generated the code required for the world service using the command - ballerina grpc --input hello.proto --mode client --output worldclient.This command generated the worldclient folder and created the ballerina file “hello\_pb.bal” that contains the definition of the client methods to call the grpc service along with the definition of the HelloRequest and HelloResponse provided in the hello.proto. I copied this file inside the world module.

I created the new ballerina file called “world.bal”. This file has a single main method that connects to the helloservice using the definition of the client method that will call the hello method of the helloservice.

#### Deployment

Environment system specifications -

For deployment, I am using Docker 2.3.0.4. After deploying docker, I have built the docker image of the services for every integration scenario. To start the services I am using docker-compose to run every service in the same network to avoid the latency due to the network.

For building the applications in java, I am using maven to build the application jar. Ballerina and Jolie are built using the respective ballerina and jolie built tool.

The docker image for glassfish server is used from Dockerhub with tag “latest” <https://hub.docker.com/_/glassfish>.

The docker image for rabbitmq is pulled from dockerhub. Image of Rabbitmq with tag “rabbitmq:3” is used for the experiment.

The image for Java is java:8-jdk, Ballerina is ballerina:1.2.3 and Jolie is jolielang/jolie

#### Execution

I have used postman to invoke the request for ReST. Powershell script for invoking gRPC and AMQP script.

# References

[Fowler, M. (2010). *Richardson Maturity Model*. Martinfowler.Com. https://martinfowler.com/articles/richardsonMaturityModel.html](https://www.zotero.org/google-docs/?broken=zEowuK)

[Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., & Jackson, B. (2019). An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 3–18.](https://www.zotero.org/google-docs/?broken=EiKxG0)

[Glen, A. (2018, July 26). *[DZone Research] Microservices Priorities and Trends*. Dzone.Com. https://dzone.com/articles/dzone-research-microservices-priorities-and-trends](https://www.zotero.org/google-docs/?broken=KBYy0h)

[*GRPC*. (n.d.). GRPC. Retrieved September 22, 2020, from https://grpc.io/](https://www.zotero.org/google-docs/?broken=ATchU3)

[Jackson, S. L. (2014). *Research methods: A modular approach*. Cengage Learning.](https://www.zotero.org/google-docs/?broken=wFb9qa)

[Li, L., & Chou, W. (2011). Design and Describe REST API without Violating REST: A Petri Net Based Approach. *2011 IEEE International Conference on Web Services*, 508–515. https://doi.org/10.1109/ICWS.2011.54](https://www.zotero.org/google-docs/?broken=ZIdv4Q)

[Montesi, F., Guidi, C., & Zavattaro, G. (2014). Service-oriented programming with Jolie. In *Web Services Foundations* (pp. 81–107). Springer.](https://www.zotero.org/google-docs/?broken=z3G4vi)

[Nanz, S., & Furia, C. A. (2015). A comparative study of programming languages in rosetta code. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, *1*, 778–788.](https://www.zotero.org/google-docs/?broken=oU0BnX)

[Nanz, S., West, S., & Silveira, K. S. D. (2013). *K.: Benchmarking usability and performance of multicore languages*.](https://www.zotero.org/google-docs/?broken=IkpdEG)

[Newman, S. (2015). *Building microservices*. O’Reilly Media, Inc. https://learning.oreilly.com/library/view/building-microservices/9781491950340/titlepage01.html](https://www.zotero.org/google-docs/?broken=Alesh5)

[*Products and Success Stories | AMQP*. (n.d.). Retrieved September 22, 2020, from https://www.amqp.org/product/realworld](https://www.zotero.org/google-docs/?broken=zmeTwH)

[Richards, M. (2015). *Microservices vs. Service-oriented architecture*.](https://www.zotero.org/google-docs/?broken=gsHRPP)

[Schwarz, G.-D., & Riehle, D. (2020). What Microservices Can Learn From Enterprise Information Integration. *HICSS*, 1–10.](https://www.zotero.org/google-docs/?broken=JQkYwT)

[Stubbs, J., Moreira, W., & Dooley, R. (2015). Distributed systems of microservices using docker and serfnode. *2015 7th International Workshop on Science Gateways*, 34–39.](https://www.zotero.org/google-docs/?broken=olZnIn)

[Wang, J., & Wan, W. (2009). Experimental design methods for fermentative hydrogen production: A review. *International Journal of Hydrogen Energy*, *34*, 235–244.](https://www.zotero.org/google-docs/?broken=VmmXvI)

[WSO2 Inc. (n.d.). *Ballerina.io*. Retrieved September 6, 2019, from](https://www.zotero.org/google-docs/?broken=0Skz6U) <https://ballerina.io/>

<https://www.docker.com/resources/what-container>

[*A Technical Introduction to XML*. (n.d.). Retrieved September 30, 2020, from https://www.xml.com/pub/a/98/10/guide0.html](https://www.zotero.org/google-docs/?H3ojeh)

[*Basics tutorial*. (n.d.). GRPC. Retrieved September 29, 2020, from https://grpc.io/docs/languages/java/basics/](https://www.zotero.org/google-docs/?H3ojeh)

[*FaqInstallJavahome—NetBeans Wiki*. (n.d.). Retrieved October 3, 2020, from http://wiki.netbeans.org/FaqInstallJavahome](https://www.zotero.org/google-docs/?H3ojeh)

[Inc, W. (n.d.). *Ballerina—Ballerina by Example*. Retrieved July 17, 2020, from https://ballerina.io/learn/by-example/](https://www.zotero.org/google-docs/?H3ojeh)

[*Java EE*. (n.d.). Retrieved October 3, 2020, from https://javaee.github.io/](https://www.zotero.org/google-docs/?H3ojeh)

[*Java EE APIs*. (n.d.). Retrieved September 29, 2020, from https://javaee.github.io/tutorial/overview008.html#GIRBT](https://www.zotero.org/google-docs/?H3ojeh)

[*Jolie/examples*. (2020). [Jolie]. The Jolie Programming Language. https://github.com/jolie/examples (Original work published 2016)](https://www.zotero.org/google-docs/?H3ojeh)

[*JSON*. (n.d.). Retrieved September 30, 2020, from https://www.json.org/json-en.html](https://www.zotero.org/google-docs/?H3ojeh)

[*Networking in Compose*. (2020, September 26). Docker Documentation. https://docs.docker.com/compose/networking/](https://www.zotero.org/google-docs/?H3ojeh)

[Newman, S. (2015). *Building microservices: Designing fine-grained systems* (First Edition). O’Reilly Media.](https://www.zotero.org/google-docs/?H3ojeh)

[*Overview of Docker Compose*. (2020, September 26). Docker Documentation. https://docs.docker.com/compose/](https://www.zotero.org/google-docs/?H3ojeh)

[*Protocol Buffers*. (n.d.). Google Developers. Retrieved September 30, 2020, from https://developers.google.com/protocol-buffers](https://www.zotero.org/google-docs/?H3ojeh)

[*RabbitMQ tutorial—"Hello World!"—RabbitMQ*. (n.d.). Retrieved September 29, 2020, from https://www.rabbitmq.com/tutorials/tutorial-one-java.html](https://www.zotero.org/google-docs/?H3ojeh)

[*Required Software*. (n.d.). Retrieved October 3, 2020, from https://javaee.github.io/tutorial/usingexamples001.html#GEXBA](https://www.zotero.org/google-docs/?H3ojeh)

[*What is a Container? | App Containerization | Docker*. (n.d.). Retrieved September 29, 2020, from https://www.docker.com/resources/what-container](https://www.zotero.org/google-docs/?H3ojeh)