JeroMF

A Software Development Framework for Building Distributed Applications Based on Microservices and JeroMQ

Aditi Jain
Computer Science Department
Utah State University
Logan, Utah, U.S.A.
email: aditi.jain@aggiemail.usu.edu

Stephen Clyde
Computer Science Department
Utah State University
Logan, Utah, U.S.A.
email: Stephen.Clyde@usu.edu

Abstract— **This paper describes the design, implementation, and testing of a software development framework, called *JeroMF*, that can help developers create scalable distributed applications based on a microservice architecture and that uses JeroMQ (a native Java implementation of ZeroMQ) for message passing. JeroMF includes an execution framework and extensible components for implementing processes, services, communication channels, messages, communication statistics, and encryption. Applications built with JeroMF do not require a message broker or any other middleware processes. However, they may include an optional Service Registry that can facilitate service discovery and secure communications. The Service Registry itself was implemented with JeroMF and is included as part of the JeroMF distribution. Thorough unit, integration, and system test cases exist for every component of JeroMF. For validation, JeroMF was used to re-design and re-implement a distributed health-care application with 13 separate types of services and very strict security requirements.**

Keywords-Microservices; Distributed Applications; Software Development Frameworks.

I. INTRODUCTION

Microservices are an architectural style for structuring applications around loosely coupled services and for making those services as granular as possible without compromising efficiency [1][2]. Microservices are highly maintainable, testable, independently deployable, and scalable [3]. Also, software engineers can organize them around business capabilities, thereby creating systems with excellent modularity and encapsulation, which can help with dynamic service composition and improve overall reliability, security, and fault tolerance. Microservices can also facilitate the continuous deployment of large, complex applications.

However, without a development framework, an application based on microservices can be hard to construct, test, debug, deploy, and maintain [3][4]. Simply splitting an application into multiple independent services generates more artifacts to manage without necessarily obtaining the desirable properties mentioned above. In fact, a haphazard refactoring of a distributed application into lots of independent services may create more complexity and thereby making maintenance and deployment more difficult.

When building an application based on microservices, developers need to modularize carefully, isolate relatively independent subsets of data together with the functionality for managing that data. Doing so will help reduce coupling and increase cohesion [5][6], and thereby improve reuse, maintainability, extensibility, and even scalability.

Also, when developers use microservices, they need to pay attention to all the typical implementation details for distributed applications, such as a) ensuring consistent implementation of communication protocols, b) ensuring the safety and consistency of transactions, c) achieving the desired amount of reliability despite communication or process failures, and d) guaranteeing the required level of security. Because a microservice-based application may have finer grain and diverse services and more communications than a similar application based on a client-server or service-oriented architectures, these challenges can be daunting and, if poorly handled, can cause the ultimate failure of the application.

This paper describes an open-source software development framework, called *JeroMF*, for creating distributed applications based on microservices efficiently and effectively. Specifically, JeroMF's goal is to make it easier for developers to create secure and reliable distributed applications by providing an execution framework and base components for processes, services, communication channels, messages, and communication statistics. JeroMF uses JeroMQ [7], a native Java port of ZeroMQ [8], as its communication library.

Section II provides some additional background on distributed applications in general, microservices, and JeroMQ. Then, Section III describes a sample application for illustration purposes. This is followed by an overview of JeroMF in Section IV. The full implementations for JeroMF and the sample application are available in public Git repositories. The URLs for these repositories are given later.

To verify JeroMF, we have created executable unit, integration, and system test cases. These test cases provide thorough test coverage using path and input domain partitioning testing techniques (see Section V). To validate JeroMF, we use it to re-design and re-implement a non-trivial

 ISBN: 978-1-61208-750-4

distributed application for the Utah Department of Health. This application, called the Child Health Advanced Records Management (CHARM) system. A brief summary of this case study is also provided in Section V. Finally, Section VI provides a summary and some thoughts about future work.

II. Background and Related Work

A. Distributed Applications

A distributed application is a software system that requires multiple processes to coordinate via network messages to complete its tasks. As such, they have to deal with both inter- and intra-process concurrency, as well as delays dues to message transfer [9]. Also, except for certain kinds of testing, the processes in a distributed application typically run on multiple independent hardware devices and therefore have to deal with the complexities of partial failure due to device or network failure [10]. Many mobile, Web-based, and enterprise applications today are actually distributed applications.

B. Microservices

To date, there is no concrete or widely accepted definition for microservices. Instead, microservices are general understood to be an architectural design concept, where the functionality of a distributed application is modularized into relatively small cohesive services. Each microservice works with its own data, can use other services, and can be implemented, tested, and deployed independent of other microservices [11].

Using microservices to build complex systems is not entirely a new idea. It stems from ideas central to Object-oriented Software Development [12] and that are found in many different types of architectures and design patterns, including Service-oriented Architecture [13], Domain-Driven Design [14], and Bounded Context [15]. Furthermore, they are consistent with software engineering principles, such as the Single Responsibility principle from SOLID [16] and the unified definitions of Abstraction, Modularization, and Encapsulation [17].

Some of the hoped-for benefits of microservices, include independent development, deployment, and scalability [4], as well as reusability, maintainability, and extensibility. Unfortunately, these benefits do not come for free. Developers must apply a wide range of expertise to address challenge inherit to distributed applications and to achieve designs with good modularity. Below is a summary list of some of these challenges identified in [4]:

- Increased complexity due to application features spanning multiple services;
- Increased complexity in setting up unit, integration, and system tests;
- The components or subpart of a real-world system often have poorly defined boundaries and, therefore, mapping them to services is non-trivial;
- Developers need to be expert in analyzing and balancing design decisions;
- Developers are responsible for the entire life cycle of a component (service);
- The complexities of state, when stateless services are not possible; and
- The complexities of communications, especially in achieving certain degrees of reliability and security.

C. Software Development Frameworks

In general, software development frameworks are collections of reusable components that provide execution infrastructures [18] and "inversion of control" [19]. With "inversion of control", developers don't have to write the main control logic directly and can focus on the functionality that is unique to an application [20], and can thus help developers to be more productive. Currently, there are many frameworks for developing distributed applications, such as Grails [21], Angular [22], and Coco [23] to name just a few. However, to our knowledge, none of them supports the creation of distributed applications using microservices and JeroMQ for communications.

D. ZeroMQ and Its Native Java Port, JeroMQ

In 2007, Pieter Hintjens along with Martin Sustrik introduced ZeroMQ as a high-performance, asynchronous, lightweight messaging library for scalable distributed applications [8]. ZeroMQ is fast, simple, and provides easy scalability. Also, it has been ported to over 40 programming languages, including a native implementation for Java, called JeroMQ [7]. Its application programming interface (API) for in-process, inter-process, peer-to-peer, and multicast communications is simple and consistent.

Developers working with ZeroMQ can create distributed application more quickly than with lower-level socket libraries because of its convenient abstractions and simple API. However, ZeroMQ is just a class library and not a development framework. As such, it does not directly provide an execution infrastructure or "inversion of control". Furthermore, it does not directly help developers with the challenges listed above.

III. Sample Application

To illustrate the architecture and use of JeroMF, we use a simple distributed application for managing used cars for multiple dealers (see Figure 1). With this sample application, every used-car dealer would run its own Used-car Server (only one shown in Figure 1) and each Used-car Server would contain a microservice, called Used-car Service. This service would encapsulate the dealer's own used-car data and provide a network-accessible API that would allow remote clients, e.g., the end user interface, to query what cars the dealer currently has in inventory and their prices.

 ISBN: 978-1-61208-750-4

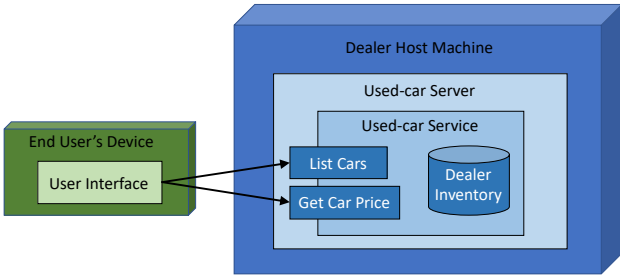

Figure 1. Sample Distributed Application for Tracking Used Cars

This sample application is minimal and only for illustration purposes. It does not contain all of the functionality one would expect in a real used-car application.

IV. OVERVIEW JEROMF

JeroMF is a framework that helps developers manage the complexities identified in Section II.B, so they can build quality distributed applications efficiently and effectively. Specifically, JeroMF aims to make it easy for developers to

1. Setup containers (processes) of services;
2. Manage service configuration parameters;
3. Create custom services that can a) access their own data stores, b) respond to incoming requests, and c) discover and use other services;
4. Define and implement reliable application-level communication protocols;
5. Use secure communications based on either asymmetric or symmetric encryption;
6. Monitor the status of all services in a distributed application;
7. Track service load and communication statistics;
8. Gracefully startup and shutdown services; and
9. Test services and inter-service communications.

A. Architectural Overview

The Unified Modeling Language (UML) Class Diagram [12][24] in Figure 2 shows JeroMF's primary packages with their essential classes and relations. From left to right are the base components for implementing custom processes, application-specific services; and communications. Developers create distributed applications in JeroMF by implementing specializations of these components or by reusing them directly. The following sections describe them in more detail, beginning with the process-related components.

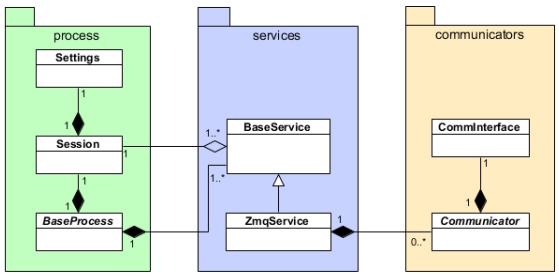

Figure 2. The primary packages in JeroMF with their key classes and relationships.

B. Processes

A process in JeroMF, defined as a specialization of BaseProcess, is an execution container that holds one or more services. If a developer is following a strict microservice architecture, then each JeroMF process will hold exactly one service. However, JeroMF allows a process to hold more than one service, at the developer's discretion, to achieve better execution and deployment efficiencies in certain situations.

A JeroMF process also contains a Session object, which in turn contains a Settings object. The Session object keeps track of the process's name, status, Settings object, JeroMQ context, and encryption keys. The Settings object holds all the configurable settings for a process and its services. Each setting has value that can be changed at runtime through either property files, environment variables, or command-line parameters. The Session object is shared with all the process's services so they can make use of that information.

Figure 3 contains a Class Diagram of used-car application, with the components implemented by the developer in light

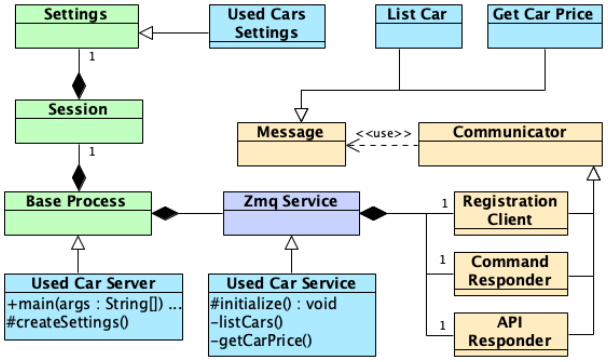

Figure 3. Classes in the Used-car Application, with those implemented by developer shown in light blue.

```
public class UsedCarServer extends BaseProcess {
  public static void main(String[] args){
    UsedCarServer process=new UsedCarServer();
    try {
      process.initialize(args,"server.config");
      UsedCarService service = new
          UsedCarService(instance.getSession(),
          "UsedCarsService");
      process.addService(service);
      process.run();
    }
    catch (Exception e) {e.printStackTrace(); }
    finally { process.cleanup(); }
  }

  @Override
  protected Settings createSettings() {
    return new UsedCarSettings();
  }
}
```

Figure 4. Implementation of the UsedCarServer class

blue. Figure 4 contains the code for UsedCarProcess from the sample application and is typical of most JeroMF processes. When a process starts, the main() routine calls the initialize() method – a Template Method [25] that setups the Session and Settings objects using virtual methods that the developer can override in the specialization. For example, the UsedCarProcess needs a custom Settings, so developer simply implements a specialization of Settings (not shown) and then overrides createSettings() method to return an instance of that specialization. See Figure 3.

After the process is initialized, the main() method instantiates the service that it will contain, adds it to the process, and then calls a run() method to begin execution. The run() method will only return once the process is stopped, which typically occurs after a service receives a Shutdown command or when it determines that the process needs to enter a terminal state. Finally, once the run() method does return, main() method will call cleanup().

C. Services

The BaseService class (see Figure 2) represents a basic microservice with an optional database connection. It has access to the process's Session object, which is provided as a parameter to the constructor. The ZmqService class is a specialization of BaseService class that represents a microservice with communication capabilities based on JeroMQ. As such, it can have zero or more communicators, i.e., instances of the Communicator class, for interacting with clients or other services. Typically, and by default, a ZmqService would include three communicators:

- a registration client that is responsible for listing the service with the Registry (if application uses a Registry), so other processes can find it and for setting up secret keys for symmetric encryption,
- a command responder that listens for general control messages from the Registry or some other control process, and
- an API responder for handling requests from clients.

None of these communicators are required and are only setup if their configuration settings have values in the Settings object.

Although BaseService and ZmqService can be used as-is for instantiating many types of services, they can be further customized through specialization. Like JeroMF processes, services have initialize() and run() methods that follow the Template Method pattern, with the customizable parts encapsulated in virtual methods.

Figure 5 contains a specialization of ZmqService, called UsedCarService, for the used-car application. When a UsedCarService is initialized, which happens when the service is started, it calls its super's (i.e., ZmqService's) initialize() method, which automatically sets up instances of the three types of communicators listed above.

```
public class UsedCarService
        extends ZmqService {

 UsedCarService(Session session, String srvName)
       throws ServiceException {
   super(session, srvName);
 }

 @Override
 protected void initialize()
        throws ServiceException {
  super.initialize();
  apiResponder.addMessageHandler(ListCars.class,
     EncryptionMode.None,
     EncryptionMode.None,
     msg -> listCars());
  apiResponder.addMessageHandler(GetCarPrice.class,
     EncryptionMode.None,
     EncryptionMode.None,
     msg -> getCarPrice(msg));
 }

 private Message listCars(){ … }

 private Message getCarPrice(Message request){ … }
}
```

Figure 5. Code snippet of UsedCarService

ZmqService's initialize() method also calls its super's (i.e., BaseService's) initialize() method, which sets up everything that is needed for working with the database. The actual opening the database connection is deferred until the first time it is used, thereby minimizing initialization time

After calling its super's initialize() method, UsedCarService's initialize() method customizes its API Responder to handle two types of messages, namely ListCars and GetCarPrice, by setting up message handlers for them. A message handler for a type of message defines what kind of encryption to expect for the incoming message and what type of encryption to use for the reply, along with a lambda function for processing incoming messages. In this example, the both lambda function simply call private methods. The private methods (implementations not shown) get a reference to database connection using a protected method inherited from BaseService and then use that connection to retrieve the requested information. They return a reply message or a null, if the desired information could not be retrieved.

D. Communicators

Communicator is an abstract base class for the objects that handle all the communications in JeroMF. A communicator uses JeroMQ, which in turn uses one of three transport-layer communication mechanisms, namely: *Transmission Control Protocol* (TCP), in-process (Inproc), or inter-process communication (IPC) [26]. Each communicator has an end point that defines both the transport-layer communication mechanism and either the local address that the communicator will bind to or the remote end point that it will connect to. The details about a communicator's end point are encapsulated in an instance of CommInterface class. Developers do not need to directly create or access these objects.

JeroMF includes six reusable communicators:

 ISBN: 978-1-61208-750-4

- The *Requester* and *Responder* communicators handle reliable request-reply style communications where the requester initiates all conversations
- The *Active Responder* and *Passive Requester*, which also handle reliable request-reply style communications, but the responder starts by indicating its readiness to receive requests
- The *Command Publisher* and *Command Responder*, which provide for simple but secure one-way message broadcasts.

JeroMF also includes a special type for Requester, called *RegistrationClient*, that registers services with the optional Registry process. This was mentioned above as one of the standard communicators for a ZmqService.

All communicators can send and receive encrypted or unencrypted messages. For encrypted messages, a communicator may use either asymmetric encryption based on a public-private key pair or symmetric encryption based on a shared secret key. For asymmetric encryption where a communicator needs to encrypt or decrypt with a private key, a ZmqService will give the communicator the name of the key pair and the password for opening the private key. It should get these values from the Settings object. For asymmetric encryption where a communicator needs to decrypt or encrypt a message with a public key, it can ask its ZmqService to lookup the public key by name. If the distributed application is using a Registry, then a ZmqService can use the Registry to discover this public key, if it is not already known.

Since communicators send and receive messages, JeroMF provides a base class, called Message, for implementing message structures quickly. Developers simply have to create specializations of this base class and then define appropriate data members with getters and setters.

V. TESTING AND EVALUATION

A. Verification

JeroMF was tested at the unit, integration, and system level with executable test cases using JUnit [27]. For unit testing, we used a combination of path testing [28] and input domain partitioning testing [29] techniques and achieved reasonably good coverage by striving to meet the following criteria:

- Every statement is executed in at least one test case.
- Every possible outcome of each conditional clause is tested in at least one test case.
- Representative examples of each boundary case for every looping construct is executed in at least one test case.
- Every possible exception is thrown in at least one test case.
- Representative examples from each partition element of each input domain for each method is used in at least one test case.

During the unit testing, we discovered that some of the declared exceptions from JeroMQ and other 3rd party libraries are impossible to stimulate in automated test cases. So, our coverage for unit testing is not 100%, but it is very close.

For integration and system testing, we also created executable unit test cases using Junit. However, each of these test cases have to ensure that other services are running and, if not, start them up before executing and shut them down afterwards. To this end, we created some utility components for checking the status of another service, for launching a process that contains that service, and for eventually shutting that process down. These utilities components allow us to create automated integration and system test cases, giving us confidence that the individual components of JeroMF are working together correctly and that the framework as a whole is satisfying its requirements.

B. Validation

Validating JeroMF requires using it to develop real distributed applications. Over the last 20 years, Utah State University has developed a number of distributed applications for the Utah Department of Health, including an information broker, called the *Child Health Advanced Record Management* (CHARM) system [30]. This system allows health-care professionals to view a wide range of health-care data for a given child from multiple data sources, securely and in real-time. To do its job, CHARM must monitor and interact with multiple data sources and data consumers, matcher child records across the data sources, identify special situations about which health-care professionals need to be alerted, and monitor itself.

This distributed application, which has been operating since 2006, seemed like a good candidate to re-design and re-implement using JeroMF. It is complex, requires high levels of security, maintainability, and extensibility. So, as an initial case study, we selected a major portion of this system, called the Sync Facility, and re-built this subsystem using JeroMF.

After refactoring into microservices, the Sync Facility ended up with 16 different types of services, hosted in 13 processes. The refactoring simplified the architectural design of the Sync Facility and improved its ability to be tested and deployed. Though antidotal evidence, the developers also believe that the new Sync Facility will be more maintainable and extensible.

C. Continuous Integration and Deployment

All of JeroMF (i.e., its base components, Registry, and utilities) and the used-car example are contained in the public Git repositories on Bitbucket.org, under the "usucssedevelopment" user [31]–[34]. Specifically, the base repository [31] contains the JeroMF source code and test cases. It compiles to a distribution package that distribution

 ISBN: 978-1-61208-750-4

application will import to use JeroMF. It is configured to use CircleCI [35] for continuous integration and to automatically deploy its distribution package to a Maven repository. The second repository [32] contains the Registry and is itself a program built with JeroMF. The third repository [33] contains some utility components, such as a process launcher, that are used for the integration and system testing of JeroMF but can also help with the deployment and launching of distributed applications, in general. The fourth repository contains the a barebones but functional implementation of used-car example [34].

VI. CONCLUSION

Our initial experience with JeroMF has provided preliminary evidence that it is valuable framework for implementing distributed applications based on microservices and JeroMQ. Its BaseProcess class makes it easy to define new service containers that can run on bare-bones Java platforms, i.e., a platform with no Web servers or application servers. Its BaseService and ZmqService classes make it easy to create custom microservices that can implement diverse and sophisticated functionality. The predefined Communicator and Message classes allow developers to implement common styles of communication and provide excellent starting points for implementing application-specific communication protocols. Also, the Communicator class makes it easy for developers to use either asymmetric or symmetric encryption. Furthermore, the optional Registry process can act like a key store for the public keys of registered services, simplifying key management.

The JeroMF services also have built in monitoring logic that can allow monitoring processes to either actively query the service status or receive periodic updates from services. Services can also track statistics about workloads and message traffic, and then provide that information to monitoring processes for analysis. Finally, the standard Command Responder for a service provides a simple but secure way to shut down or restart services.

Despite its rich set of features, JeroMF is still in its infancy. We envision several important enhancements to JeroMF in the near future. First, we aim to create other specializations of BaseService, like ZmqService, that would support different messaging libraries. For example, we plan to create an HttpService that uses HTTP [36] instead of JeroMQ and that has built-in support for RESTful [37] operations. After that, we plan on implementing and testing extensible services that will act as request proxies and load balancers.

We also plan to conduct several empirical studies and qualitative analyses that will aim to answer questions about its utility, reusability, extensibility, scalability, security, reliability, and maintainability. In preparation for some of these studies, we will track detailed information about software problem reports, time to resolution, induced errors from bug fixes, and more.

Finally, we plan to create more public examples that can help explain how to use JeroMF in build production-quality distribution applications and to serve as testbeds for empirical studies.

We welcome feedback and contributions from developers who would like to use JeroMF to build distributed applications.

REFERENCES

[1] "Microservices," *martinfowler.com*. Available from: https:// martinfowler.com/articles/microservices.html. [retrieved: Sept., 2019].

[2] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study.," presented at the CLOSER, 2018, pp. 221–232.

[3] P. Hauer, "Microservices in a Nutshell. Pros and Cons.," *Phillip Hauer's Blog*. Available from https://phauer.com/ 2015/microservices-nutshell-pros-cons/. [retrieved: Sept., 2019].

[4] D. Kerr, "The Death of Microservice Madness in 2018," *Dave Kerr's Blog*, 12-Jan-2018. Available from: https:// dwmkerr. com/the-death-of-microservice-madness-in-2018/. [retrieved: Sept., 2019].

[5] G. Gui and P. D. Scott, "Coupling and Cohesion Measures for Evaluation of Component Reusability," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, New York, NY, USA, 2006, pp. 18–21.

[6] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using Cohesion and Coupling for Software Remodularization: Is It Enough?," *ACM Trans Softw Eng Methodol*, vol. 25, no. 3, pp. 24:1–24:28, Jun. 2016.

[7] *Pure Java ZeroMQ. Contribute to zeromq/jeromq development by creating an account on GitHub*. The ZeroMQ project, 2019.

[8] "ZeroMQ." Available from: https://en.wikipedia.org/wiki/ ZeroMQ. [retrieved: Sept., 2019]

[9] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5 edition. Boston: Pearson, 2011.

[10] J. Link, "Chapter 11 - Distributed Applications," in *Unit Testing in Java*, J. Link, Ed. San Francisco: Morgan Kaufmann, 2003, pp. 225–240.

[11] E. Wolff, *Microservices: Flexible Software Architecture*, 1 edition. Addison-Wesley Professional, 2016.

[12] G. Booch et al., *Object-Oriented Analysis and Design with Applications*, 3 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2007.

[13] "What Is SOA?," Available from: https://web.archive.org/ web/20160819141303/. [retrieved: Aug., 2019].

[14] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 1 edition. Boston: Addison-Wesley Professional, 2003.

[15] M. Fowler, "BoundedContext," *martinfowler.com*, Available from: https://martinfowler.com/bliki/BoundedContext.html. [retrieved: Sept., 2019].

[16] S. Metz, "SOLID Object-Oriented Design - GORUCO 2009." Available from: http:// https://www.youtube.com/watch?v=v-2yFMzxqwU. [retrieved: Sept., 2019].

[17] S. Clyde and J. E. Lascano, "Unifying Definitions for Modularity, Abstraction, and Encapsulation as a Step Toward Foundational Multi-Paradigm Software Engineering Principles," in *Proceedings of the Twelfth International*

 ISBN: 978-1-61208-750-4

Conference on Software Engineering Advances, Athens, Greece, 2017.

[18] "Library vs. Framework?," *Program Creek*, Available from: https://www.programcreek.com/2011/09/what-is-the-difference-between-a-java-library-and-a-framework/. [retrieved: Sept., 2019].

[19] "Inversion of Control Containers and the Dependency Injection pattern," *martinfowler.com*. Available from: https://martinfowler.com/articles/injection.html. [retrieved: Sept., 2019].

[20] "The Difference Between a Framework and a Library," *Developer News*, 01-Feb-2019. Available from: https://www.freecodecamp.org/news/the-difference-between-a-framework-and-a-library-bd133054023f/. [retrieved: Sept., 2019].

[21] "Grails Framework." Available from: https://grails.org/. [retrieved: Sept., 2019].

[22] "Angular." Available from: https://angular.io/. [retrieved: Sept., 2019].

[23] "Coco: A New Open-Source Blockchain Framework – MontageJS." *MontageJS*. Available from: http://montagejs.org/coco-open-source-blockchain. [retrieved: Sept., 2019].

[24] A. S. Evans, "Reasoning with UML class diagrams," in *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, 1998, pp. 102–113.

[25] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition. Reading, Mass: Addison-Wesley Professional, 1994.

[26] P. Hintjens, *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[27] "JUnit – About." Available from: https://junit.org/junit4/. [retrieved: Sept., 2019].

[28] A. Watson and T. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," National Institute of Standards, NIST Special Publication 500-235, Sep. 1996.

[29] J. Tian, "Input Domain Partitioning and Boundary Testing," in *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*, IEEE, 2005, pp. 127–146.

[30] S. Clyde, *Child-Health Advanced Record Management Systems*. Salt Lake City, Utah, USA: Utah Department of Health, 2006.

[31] S. Clyde and A. Jain, *JeroMF Base Components*. Logan, Utah, USA: Utah State University, 2019. Available from: https://bitbucket.org/usucssedevelopment/base.git. [retrieved: Sept., 2019].

[32] S. Clyde and A. Jain, *JeroMF Registry*. Logan, Utah, USA: Utah State University, 2019. https://bitbucket.org/usucssedevelopment/registry.git. [retrieved: Sept., 2019].

[33] S. Clyde and A. Jain, *JeroMF Utilities*. Logan, Utah, USA: Utah State University, 2019. https://bitbucket.org/usucssedevelopment/utils.git [retrieved: Sept., 2019].

[34] S. Clyde and A. Jain, *JeroMF Used-car Example*. Logan, Utah, USA: Utah State University, 2019. https://bitbucket.org/usucssedevelopment/jeromfexamples-usedcars.git [retrieved: Sept., 2019].

[35] "Continuous Integration and Delivery," *CircleCI*. Available form: https://circleci.com/. [retrieved: Oct., 2019].

[36] J. F. Reschke and R. T. Fielding, "Hypertext Transfer Protocol (HTTP/1.1): Authentication." Available from: https://tools.ietf.org/html/rfc7235. [retrieved: Sept., 2019].

[37] "What is RESTful API? - Definition from WhatIs.com," *SearchMicroservices*. Available from: https://searchmicroservices.techtarget.com/definition/RESTful-API. [retrieved: Sept., 2019].