# **Developing a plugin framework for spring boot**

by

**Tenson Cai**

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Simanta Mitra, Co-major Professor
Gurpur Prabhu, Co-major Professor
Carl Chang, Committee member

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2021

Copyright © Tenson Cai, 2021. All rights reserved.

## DEDICATION

*Dedicated to my family, Ying Cai, Chen Tang, Kenson Cai, and Emma Cai, and my friends for their love and support through these years.*

# TABLE OF CONTENTS


Page

ACKNOWLEDGMENTS .......... iv

ABSTRACT .......... v

CHAPTER 1. INTRODUCTION .......... 1
- Background .......... 1
- Motivation .......... 1
- Solution .......... 2

CHAPTER 2. REVIEW OF LITERATURE .......... 3

CHAPTER 3. DESIGN AND IMPLEMENTATION .......... 6
- Spring Boot application architecture .......... 6
- High-level plugin architecture .......... 7
- Implementation of the plugin system components .......... 8
  - PluginInterface.java .......... 8
  - PluginManager.java .......... 9
  - MainService.java .......... 9
  - Manifest JSON file .......... 10
  - Event notification system .......... 10
  - Frontend to backend event notification system .......... 13
  - Spring's Application Context .......... 14

CHAPTER 4. RESULTS .......... 15
- Running the application .......... 16

CHAPTER 5. SUMMARY AND FUTURE WORKS .......... 18
- Important lessons learned .......... 18

REFERENCES .......... 21

## ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to all who have supported me throughout this journey. I would like to thank Dr. Simanta Mitra and Dr. Gurpur Prabhu for being my major professors and for their patience, advice, and support in helping me complete this thesis. I would also like to thank my committee member Dr. Carl Chang for his support and advice throughout this journey.

I would also like to thank my family and friends for their constant love and support. Anyone would be blessed to have such family and friends.

Lastly, I would like to thank my colleagues, the department faculty, and staff for making my time at Iowa State University a wonderful experience.

## ABSTRACT

Nowadays, software applications operate on a massive scale in terms of features and the number of customers they serve. As software applications become increasingly complex, it becomes increasingly difficult to modify the source code without the application becoming bloated and disorganized. In response, software engineers are continually designing software architectural patterns and concepts to enhance code organization and productivity. One very popular concept is the plugin system architecture, which allows developers to add features and functionalities to an application without modifying the core application itself. This research delves into different design patterns and components of plugin systems and an implementation of a plugin system for Spring Boot, a popular tool used by Java developers to develop enterprise web applications.

# CHAPTER 1. INTRODUCTION

## Background

One of the most popular frameworks for software engineers is Spring Boot, a Java framework used to develop backend web and enterprise applications. Spring Boot applications serve millions of customers. As these applications become increasingly complex, it becomes increasingly difficult to modify the source code without the application becoming bloated and disorganized. In response, software engineers are continually designing software architectural patterns and concepts to enhance code organization and productivity. One very popular concept is the plugin system architecture, which allows developers to add features and functionalities to an application without modifying the core application itself. This research delves into different design patterns and components of plugin systems and an implementation of a plugin system for Spring Boot, a popular tool used by Java developers to develop enterprise web applications.

## Motivation

Customer requirements for software applications are ever changing, and it is difficult (but still important!) to satisfy different customer needs. To satisfy different customer needs, the traditional way was to develop all of the features into a monolithic application. The advantages of a monolithic application are that it is easier to develop initially and easier to deploy than a plugin system application. However, this one size fits all approach would make the code base disorganized, inflexible, and bloated as some features would never be used by customers. Furthermore, it is virtually impossible to anticipate all feature requirements. Bugs and technical debt would also accumulate over time and break functionalities. Therefore, it is desirable to keep the code base as small and organized as possible.

Instead of developing all features into a monolithic application, a plugin system allows a smaller, core application to attach and detach features through small pieces of software, called plugins. A plugin is a piece of software that adds a new functionality to a host application without modifying the host [1]. The advantages of a plugin system include increased modularity, flexibility, and functionalities. Plugins allow users to customize features to fit their needs. Plugins can also be open sourced to allow other developers to develop additional functionalities. However, a plugin system is harder to design. Managing different plugin versions and conflicts can become tremendously difficult. And since it is open sourced to external developers, the application is less secure. While there are certain disadvantages, plugin systems are still very popular for all the advantages they provide.

**Solution**

In this research, we looked at different plugin frameworks and designs to be used with Spring Boot. We decided to build our own custom plugin framework instead of building upon other frameworks because we wanted to keep it simple, light, and in our control. Our plugin framework integrates with Spring Boot properties like the service layer, Spring data JPA (Java Persistence API) methods, and databases. Building our own system from scratch also enhanced our understanding of the various components and inner workings of a plugin system.

The rest of the report is organized as follows. Chapter 2 goes over the related works about plugin systems. Chapter 3 goes over our design and implementation of our plugin system for Spring Boot. Chapter 4 shows the sample project we did to demonstrate our plugin system. Chapter 5 summarizes the report and talks about future works.

# CHAPTER 2. REVIEW OF LITERATURE

As developers migrated from making monolithic applications to making plugin architectures, two plugin architectures arose, a traditional and a pure plugin architecture [3].

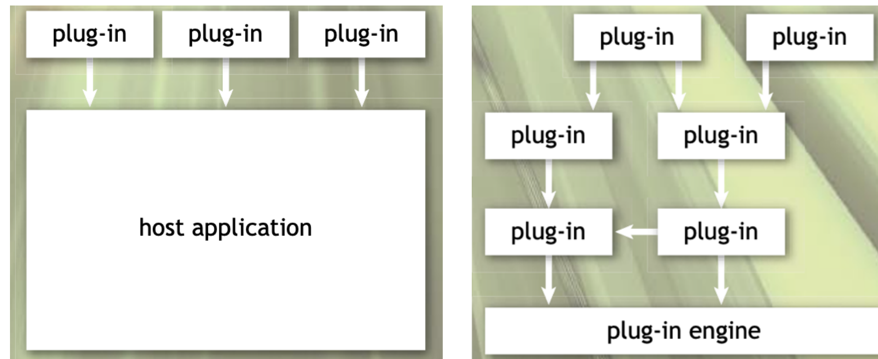


Figure 1. Left: Traditional plugin architecture. Right: Pure plugin architecture

The traditional plugin system involves a host application where the host provides extension points, or hooks, for plugins to use. An extension point is a function that allows a plugin to plug into the host. Examples of traditional systems include Gmail, Figma, Asana, VSCode, and more. The more recent architecture is a pure plugin architecture where every part of the application is a plugin. For example, the Eclipse IDE's UI Workbench is made up of editors, perspectives, views, and actions. Each of these is a plugin that extends off of and communicates with other plugins. In the pure plugin system, the only functionality of the kernel plugin-in engine is to load and start the core plugins. The plugins themselves will become hosts to other plugins by providing their own extension points.

There are two different ways to run plugin systems. It can be run statically, where the system has to be restarted each time a new plugin is added. Or it can be run dynamically, where the system can receive new plugins without having to be restarted. Many plugin systems also

utilize lazy loading (e.g. Eclipse, VSCode, etc.), where plugins are only loaded and ran when they are needed [4, 5]. This reduces the memory footprint and speeds up application start time.

A pattern called the pluggable factories design pattern was introduced by Kharrat and Qadri [1] that allowed plugins to self-register themselves to the core application. This pluggable factory design pattern is a hybrid of the prototype design and abstract factory pattern. The prototype design gives plugins a prototype class to extend off of, while the abstract factory pattern hands the responsibility of self-instantiation to the plugins. The prototype class defines a method, such as create(), that must be implemented by each plugin. The host application would call the create() method for each plugin, and the create() method would return an instance of the plugin. A map registry is used to store the plugin instances. The key of the map is the name of the plugin, and the value of the map is the pointer to one prototype instance of the plugin.

Triglianos and Pautasso [2] designed a JavaScript plugin framework for web-based applications. In their research, plugins were published as NPM (Node Package Manager) modules, a popular registry for many JavaScript modules. On the other hand, Java plugins are usually packaged as Java Archives (Jar) files. Their JavaScript plugin design used proxy objects, implemented by the host application, to expose all endpoints, such as hooks, events, settings, and APIs, to the plugins. Each time the host instantiated a plugin, it passed a new instance of the host-implemented proxy object to the plugin's constructor. In the Java world, the host would implement interfaces instead and inject host objects with the interface types into the plugins' constructors. In their JavaScript plugin framework, the host did not directly call the plugins' methods. Instead, plugins would subscribe to specific hooks and events. The plugins' event listener methods would be triggered when either the necessary events were published or the hooks were activated by the host.

Other than design patterns, we also studied open sourced plugin frameworks used for Spring Boot such as OSGI (Open Service Gateway Initiative), spring-plugin, and Pf4J. OSGI is powerful but also very complex and hard to learn and setup. We decided that OSGI was an overkill for our project. Pf4J is a lighter framework aimed at small to medium projects. We decided that Pf4J was also too complex for our needs. We decided that implementing our own plugin framework was the best way to understand the different parts and inner workings in the system.

Finally, we also studied different implementations of plugin systems in Eclipse, VSCode, and Figma. Eclipse uses the OSGI framework along with interfaces and Java inheritance properties to run plugins [4]. Each Eclipse plugin is packaged as a JAR file. The JAR file contains a manifest file, Java code, read-only files, and other optional resources like images. The manifest file of the plugin provides information about the plugin's name, version, extension points, etc. Eclipse uses dependency injection to inject host objects into plugins to give plugins access to host properties. VSCode and Figma provide a single global object to plugins [5, 6]. Both JavaScript and Java applications use callback functions to trigger event activation functions. In the Java language, callback functions are implemented with interfaces with defined methods and the Java Reflection API is used to activate the functions. A notable difference is that JavaScript applications write their manifest files in JavaScript Object Notation (JSON) syntax, while Eclipse uses XML syntax [4, 5, 6].

# CHAPTER 3. DESIGN AND IMPLEMENTATION

We aimed to build a plugin framework that can integrate with Spring Boot. The plugin framework will work with Spring Boot's service layer and call Spring Data JPA to access data from the databases. First, we describe the typical structure of a Spring Boot application. Then, we describe the high-level architecture of our plugin system integration with Spring Boot. Finally, we describe how the high-level overview is implemented with code.

## Spring Boot application architecture

A typical Spring Boot project follows the model-view-controller pattern. The controllers handle all incoming HTTP requests from the frontend and pass the requests to the service layer. The services execute create, read, update, delete (CRUD) operations on the database using repositories and Spring Data JPA. The entity models are mapped to the database tables where an entity model's instance variables are mapped to the attributes in the corresponding table. Spring Boot makes use of Inversion of Control and Dependency Injection to connect the different parts together.

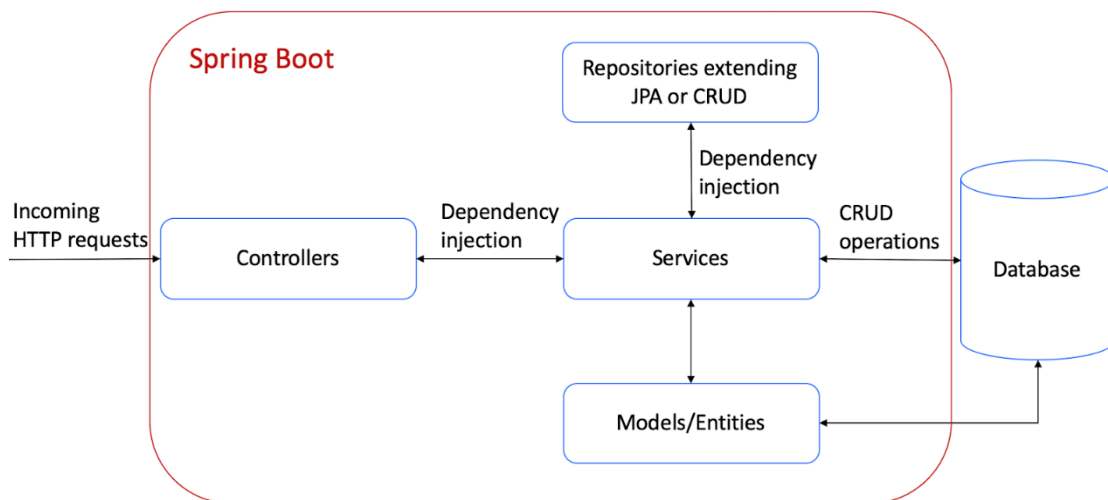


Figure 2. Spring Boot architecture.

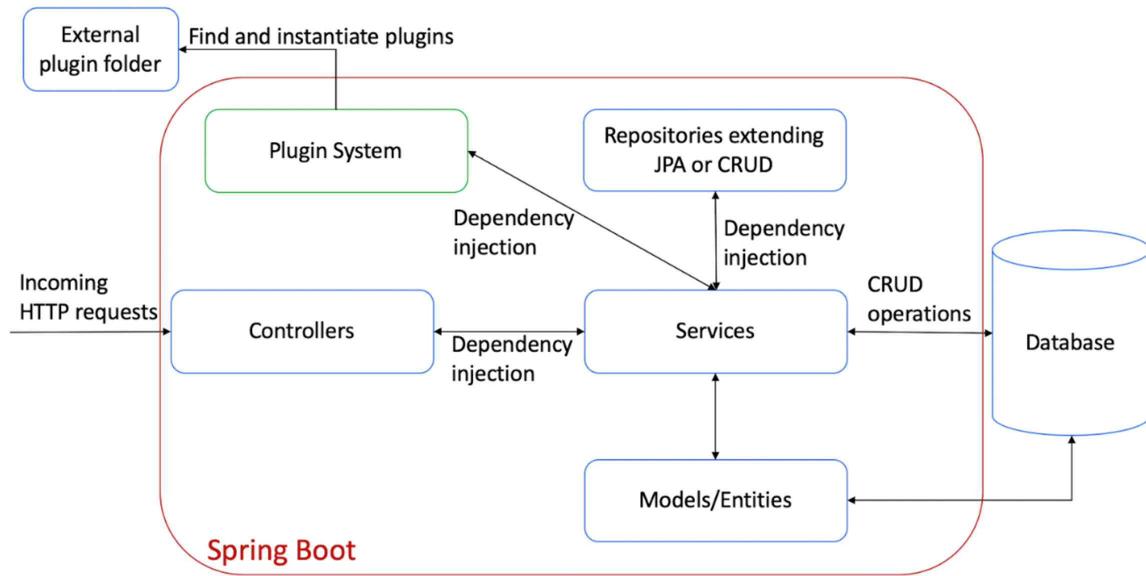


Figure 3. Plugin system integrated with Spring Boot.

Figure 3 shows how our plugin system integrates with Spring Boot. It will read an external plugin folder for plugin Jars and use dependency injection to communicate with the service layer.

**High-level plugin architecture**

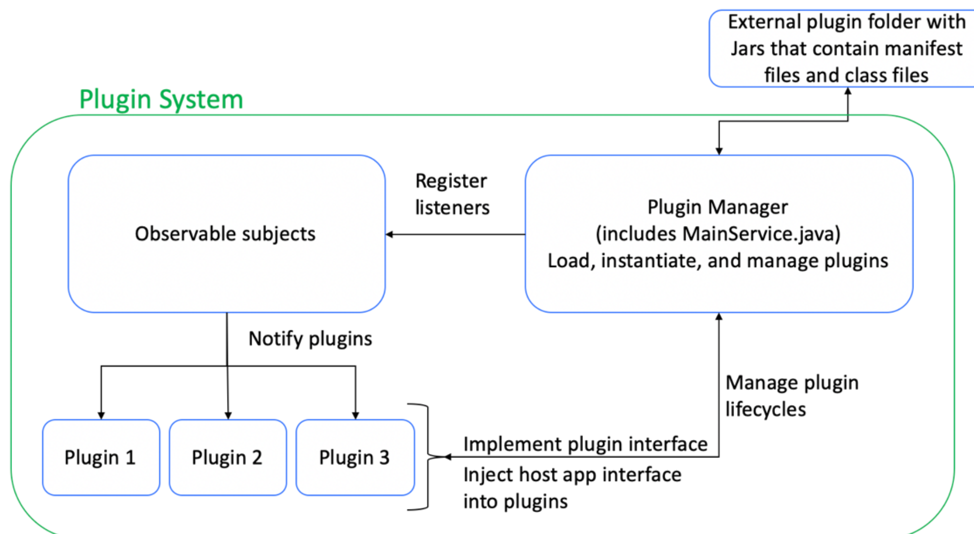


Figure 4. High-level plugin architecture

Each plugin application is compressed into a Jar file and stored in an external plugin folder. Note that plugins are packaged as regular Jar files and not as executable Jars because plugins are not stand-alone applications. Instead, the host will run and monitor the plugins. Each Jar includes a Manifest.json file describing information about the plugin's name, id, description, version, listeners, etc.

The responsibility of the plugin manager is to load, instantiate, and manage plugins. The plugin manager will read the Manifest.json file and register listeners to the correct event publishers.

Plugins need to be notified when certain events occur. Such events could be when data is saved to the database or when certain data is ready for the plugin to consume. The observer pattern and the publisher/subscriber pattern are two popular ways to notify plugins, implemented using interfaces for events and listeners. In order for developers to know which events and listeners a core application can receive, a documentation will have to be created describing the events, listeners, and which methods are called at which times.

In the next section, we explain how each component of the plugin system is implemented.

## Implementation of the plugin system components

### PluginInterface.java

As mentioned before, extension points are functions where plugins can plug into the host application. These extension points are implemented as interfaces. An interface serves as a contract between a host and a plugin. The host Spring Boot application defines a plugin interface for all plugins to implement. The interface defines three methods: setMainApplication(IMainService main), run(), and stop(). It is required for each plugin to implement the plugin interface as this allows the plugin manager to use the Java Reflection API to find the plugin.

**PluginManager.java**

The plugin manager traverses through the external plugin folder using the Java File API. The path to the plugin folder is defined in the manager class by a final variable. When it finds a Jar file, it adds the path of that Jar file to a queue. The paths in the queue is given to the MainService class, who is part of the plugin manager shown in figure 4 and responsible for loading, instantiating, and managing plugins.

**MainService.java**

The mainService class is responsible for loading the Jar files, instantiating plugin objects, and managing plugin lifecycles. To carry out those responsibilities, it uses the Java Reflection API. To load a Jar file, the file path is popped from the queue and a ClassLoader object is created.

```
ClassLoader cl = new URLClassLoader(new URL[] {new URL("jar:file:" + pathToJar +
"!/")}, Thread.currentThread().getContextClassLoader());
```

Then, a JarFile object is created, and the manager loads each file from the JarFile object with classLoader.loadClass(fileClassName). The PluginInterface.class.isAssignableFrom(Class c) API is called for each file and returns a boolean whether or not the class implements PluginInterface.java. If so, the class is instantiated into an object and added to the list of plugins with pluginList.add(c.newInstance()).

In the setMainApplication method, the main app will inject an instance of itself into the plugin for the plugin to access its properties and extension points. The main app has a type of IMainService. This interface defines methods for the plugins to call the main app to do, such as getPostService(). These methods allow the plugins to access the data stored in the database. The run() method will start the plugin and stop() will be called during shutdown.

## Manifest JSON file

A manifest JSON file must be included in each plugin Jar because this provides the main service with important information. JSON is used because it is easier to read than XML and can be manipulated with the Jackson API. The listeners are written in an array of strings. The main service class iterates through the array of listeners and calls the registerListener() method to the correct observable subject.

```
{
  "name": "emailPlugin",
  "pluginId": "email",
  "description": "This is some email plugin.",
  "version": "1",
  "listeners": ["PostListener"]
}
```

Figure 5. Sample manifest JSON file for a plugin.

## Event notification system

Plugins may want to execute certain funtions when a certain state or data changes. Such events could be when a new user is saved or deleted or when a new user comes online. In our implementation, the observer pattern is used for event communication. The observer pattern and the publisher/subscriber pattern are two ways to notify plugins.

The observer pattern is a one-to-many dependency pattern. It contains observables or subjects. These are the objects being observed. It also contains observers or listeners. Each subject maintains a list of listeners that will be notified when the state changes.

In the pub/sub pattern, a publisher publishes events to a specific event queue when the state changes, and a subscriber subscribes to specific event queues. A publisher does not maintain a list of subscribers like in the observer pattern. The only knowledge a publisher has is which event queue to publish to. A subscriber has a separate thread that constantly checks if

there is an event in the queue. The pub/sub pattern is more loosely coupled and more efficient than the observer pattern because it only has to publish once. But it is also harder to implement as it requires multi-threading. We chose to implement the observer pattern for its simplicity, and because we felt that efficiency is not an important requirement at this time.

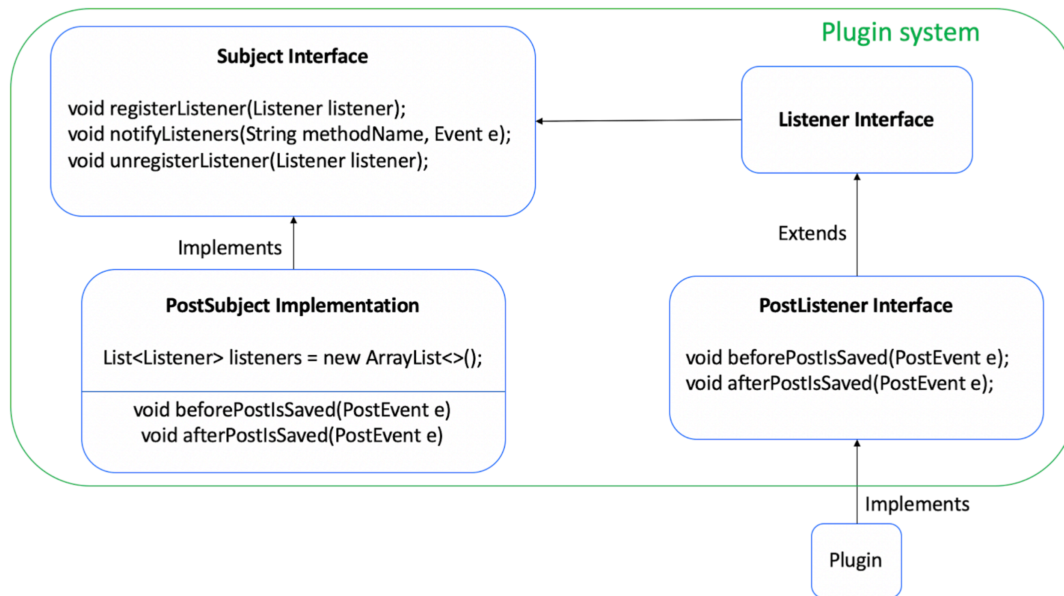


Figure 6. Observer pattern and architecture components.

Figure 6 shows the parts of the notification system. All sub listener interfaces such as PostListener must extend the Listener interface. The Listener interface does not define any methods, but its importance comes from its type. When the PostListener extends Listener, its type becomes a Listener type through Java's inheritance properties. This is important because the Subject interface uses the Listener type in its methods. After extending the Listener interface, the PostListener can define its own unique methods such as beforePostIsSaved and afterPostIsSaved.

Following the listener pattern, all custom subject classes must implement the Subject interface, which defines three methods: registerListener, notifyListener, and unregisterListener. The custom subject class, like the sample PostSubject class, maintains an array list of Listeners.

Registering and unregistering listeners would be adding and removing listeners from the array list. It also has two custom methods: notifyBeforePostIsSaved and notifyAfterPostIsSaved. The subjects are connected to the service classes through dependency injection (autowiring in Spring Boot). When an event occurs, such as after a post is saved to the database, the notify method will be called. The notify method iterates through its list of Listeners and uses the Java Reflection API to call the provided method name to notify each plugin.

```
public class PostSubject implements Subject {
  private List<Listener> listeners = new ArrayList<>();

  @Override
  public void registerListener(Listener listener) {
     listeners.add(listener);
  }

  @Override
  public void notifyListeners(String methodName, Event event) { // uses Java Reflection API…}

  @Override
  public void unregisterListener(Listener listener) {
     listeners.remove(listener);
  }

  public void beforePostIsSaved(PostEvent e) {
     notifyListeners("beforePostIsSaved", e);
  }

  public void afterPostIsSaved(PostEvent e) {
     notifyListeners("afterPostIsSaved", e);
  }
```

Figure 7. Sample code for a custom subject.

## Frontend to backend event notification system

Some plugins may want to be notified when an event occurs on the frontend instead of the database. For example, when a user clicks on a button, a plugin may want to be notified to call an API. In order for the plugin to be notified, it must implement the FrontendEventListener interface that defines a notify method. The notify method accepts a hashmap, which contains any important information the plugin needs. To notify the plugin in the backend, the frontend request needs to specify the plugin id along with any information that the plugin needs, such as text input for example. The FrontendEventService maintains a list of plugins given by the Plugin Manager. The service will search the list for the plugin that has a matching plugin id from the hash map. Once the plugin is found, the plugin's notify method is called through the Reflection API. If the plugin wants to return information to the frontend after processing, the notify method can return an Object for that, which is then passed to the FrontendEvent controller and back to the frontend. Currently, one frontend event can only be matched to one plugin because the frontend event's plugin id is hardcoded into the HTTP request. This is a limitation that can be resolved in future works to allow multiple plugins to subscribe to one frontend event.

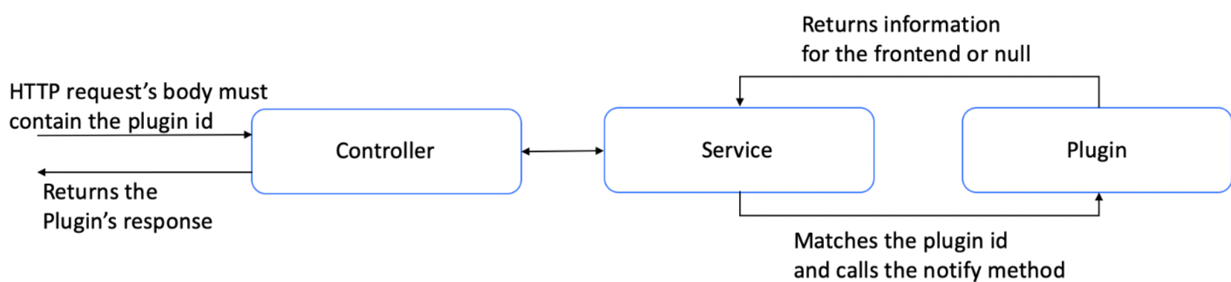


Figure 8. How a plugin is notified from the frontend.

## Spring's Application Context

This class is used to expose the Spring JPA methods from the host to the plugins to allow plugins to query the database. Spring Boot uses dependency of inversion to manage components and the application context is used to retrieve different services, which are components. The MainService is responsible for calling the getBean() method for each service class. These services, such as PostService, is assigned to a variable in the MainService.java class, where it can be exposed for plugins to use. The IMainService interface contains getter methods that will return these service objects to plugins. The services also need to implement an interface because there may be methods that should not be exposed to the public. Only JPA methods that can be exposed to the public should be included in the interface.

```
@Component
public class SpringContext implements ApplicationContextAware {

    private static ApplicationContext context;

    /**
     * Returns the Spring managed bean instance of the given class type if it exists.
     * Returns null otherwise.
     * @param beanClass
     * @return
     */
    public static <T extends Object> T getBean(Class<T> beanClass) {
        return context.getBean(beanClass);
    }

    @Override
    public void setApplicationContext(ApplicationContext context) throws BeansException

        // store ApplicationContext reference to access required beans later on
        SpringContext.context = context;
    }
}
```

Figure 9. Sample code of Spring Context.

# CHAPTER 4. RESULTS

To demonstrate our plugin framework, we developed a website blog application, a popular type of website on the Internet. It was made with React for the frontend and Spring Boot for the backend. The host application's basic functionalities are creating and deleting users, blog posts, and comments. Two plugins were developed to enhance the functionalities of the blog application.

The first plugin is a grammar plugin that demonstrates the frontend to backend communication system. When the user clicks on the "Check grammar" button on the website, this will notify the grammar plugin to call the GrammarBot API. The plugin receives and formats the bot's response into the correct styling and returns it as an Object to the frontend, where the result is displayed.

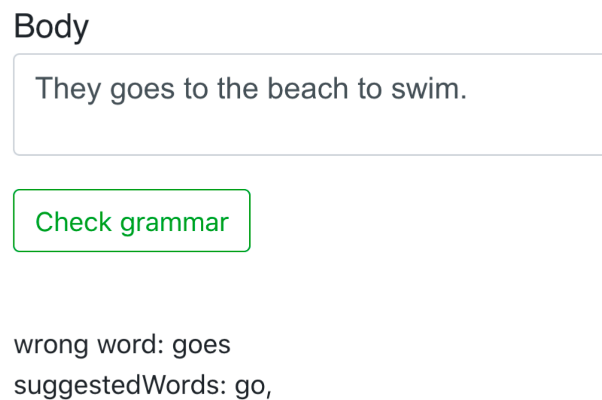


Figure 10. Grammar plugin. The grammar plugin returns suggested words for any incorrect words.

The second plugin is an email plugin that demonstrates the observer pattern. After a new blog post is saved to the database, the afterPostIsSaved method is called in the PostSubject by the PostService class. The PostSubject calls the afterPostIsSaved method for each plugin in its

list. For now, its list only includes the email plugin. The email plugin had implemented the PostListener, so when its afterPostIsSaved method is called, the plugin sends an email to a user.

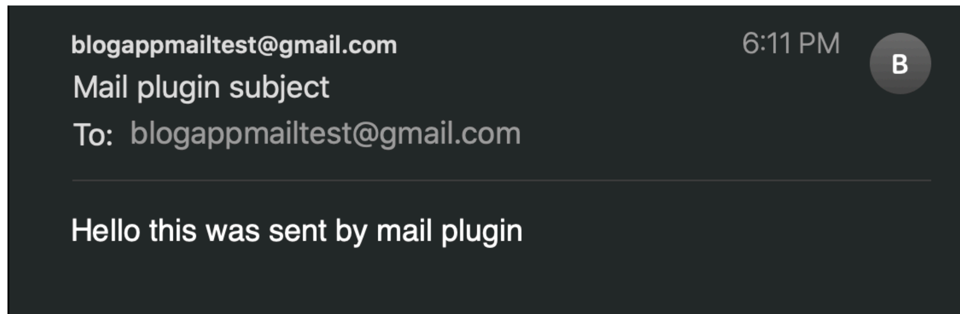


Figure 11. Email plugin. After a new blog post is saved to the database, an email is sent out notifying the user.

**Running the application**

The plugins were packaged as Jar files and placed in an external plugin folder, as shown in figure 12. This plugin folder was outside the project application.

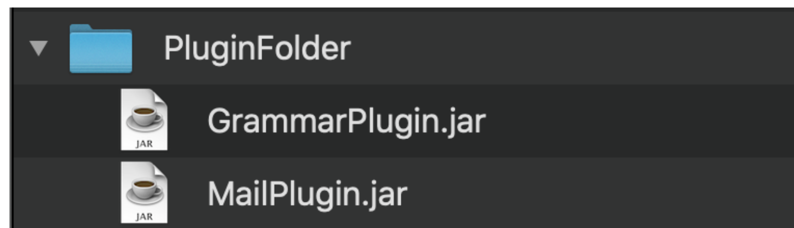


Figure 12. Plugin Jars in the plugin folder.

Maven creates an executable jar from a Spring Boot application. This jar was moved into a folder with a library folder, shown in figure 13. The library folder contains all the dependencies of the two plugins. For example, the email plugin depends on the javax.mail jar.

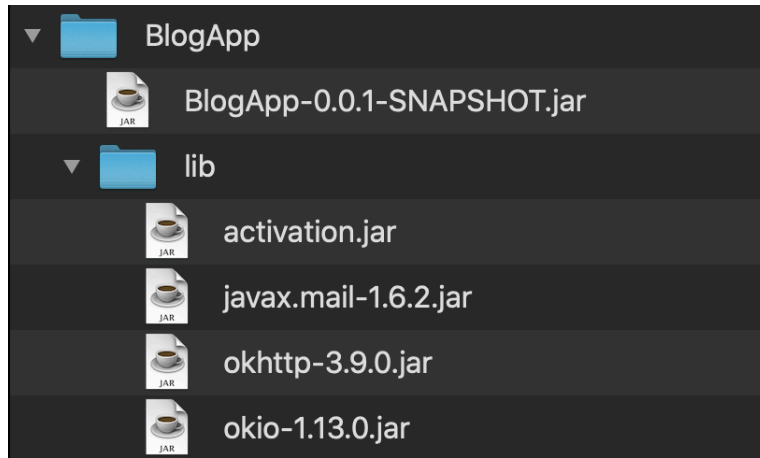


Figure 13. Spring Boot Jar with a library of plugin dependencies.

To run the Spring Boot application, this command is used: **java -Dloader.path="lib/" -jar BlogApp-0.0.1-SNAPSHOT.jar**. This adds all the dependencies from the lib folder into the BlogApp Jar's classpath.

# CHAPTER 5. SUMMARY AND FUTURE WORKS

In this research, we analyzed the different components that make up a plugin system and implemented a system to integrate with Spring Boot. We designed, implemented, and demonstrated a plugin system with Spring Boot that allows users to write and plug in their own code for the features they want.

## Important lessons learned

1. It was important that `Thread.currentThread().getContextClassLoader()` was used in the class loader because a Java application can use different classloaders to load classes. The classes loaded by one classloader are kept apart from classes loaded by a different classloader for security reasons.

2. Java's Reflection API was used to create and run objects.

3. An observable event notification system was created that imitated the design used by Eclipse. We learned why it was important for all listeners to extend off of an empty Listener interface because this gave all sub listeners a type of Listener.

4. A frontend to backend communication system was implemented that matched the event with the correct plugin based on the plugin id. The kind of information required in this HTTP request between frontend and backend needs to be documented for other developers to follow.

5. Spring's Application Context was used to retrieve Spring components in order to expose Spring JPA methods from the host to the plugins.

6. Before using the command **java -Dloader.path="lib/" -jar BlogApp-0.0.1-SNAPSHOT.jar**, make sure the pom file in the Spring Boot application includes the configuration layout zip under the spring-boot-maven-plugin, shown in figure 14.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>  <!-- added -->
        <layout>ZIP</layout>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Figure 14. Modified pom file with ZIP configuration.

For future works, we can implement a frontend plugin system mechanism that can work with the backend plugin system. Developers could develop one plugin with a frontend side and a backend side. The frontend code would be packaged and deposited into a frontend plugin folder, and the backend Jar would be deposited into a backend plugin folder.

Currently, each frontend event request has one hardcoded plugin id which corresponds to one matching plugin in the backend. This limits each frontend event to be matched with only one plugin. To allow multiple plugins to subscribe to one frontend event, plugins in the backend can specify which frontend event id they want to be notified of.

It would be highly beneficial to be able to let plugins define their own Spring beans. This allows plugins to define their own controllers, services, models, and database operations. It would also be beneficial for the plugin framework to load and run plugins dynamically. Currently, the system has to be restarted every time a new plugin is added to the plugin folder.

There also needs to be a way for plugins to define their own extension points and communicate with other plugins. This would allow plugins to become hosts for other plugins. Spring Boot also comes with its own publisher/subscriber implementation to be used within the

Spring framework. However, since plugins cannot yet define their own Spring beans, we had to abandon the idea of using Spring's event notification system.

As stated before, considerable time was spent on loading the plugin classes correctly to use the Java Reflection API. The one-line solution was to include the `Thread.currentThread().getContextClassLoader()` in the class loader object. This solution to the problem was not obvious and took a lot of time searching the internet. Upon observation, many developers ran into the same issue while developing for the OSGI framework. It seems as though even a standard framework like OSGI did not load the classes properly for developers. Everyone seemed to hack their own solutions, and we had to resort to trying every feasible solution we could find. Upon reflection, it would have been much easier if there was a program available to manage the class loaders and communicate between the host app and the plugins. This program could give the host app the correct classes to load or integrate with the Reflection API to help the host app find the correct plugin methods.

Another feature to consider is using websockets to communicate plugin events faster between the frontend and. This would be better for real-time applications. Currently, we use HTTP request.

## REFERENCES

1. D. Kharrat and S. S. Quadri, "Self-registering plug-ins: an architecture for extensible software," *Canadian Conference on Electrical and Computer Engineering, 2005.*, Saskatoon, SK, Canada, 2005, pp. 1324-1327, doi: 10.1109/CCECE.2005.1557221.

2. Triglianos V., Pautasso C. (2015) Asqium: A JavaScript Plugin Framework for Extensible Client and Server-Side Components. In: Cimiano P., Frasincar F., Houben GJ., Schwabe D. (eds) Engineering the Web in the Big Data Era. ICWE 2015. Lecture Notes in Computer Science, vol 9114. Springer, Cham. https://doi.org/10.1007/978-3-319-19890-3_7

3. Birsan, Dorian. On Plug-ins and Extensible Architectures. Association for Computing Machinery, 2005.

4. Rubel, Dan. The Heart of Eclipse. Association for Computing Machinery, 2006.

5. Vscode-docs. https://vscode-docs.readthedocs.io/en/stable/

6. Figma Developers. https://www.figma.com/plugin-docs/intro/