A Feature Flag Framework for Toggling Features in Warehouse Management Systems

Gautham Ram Rajendiran

gautham.rajendiran@icloud.com

Abstract
Feature flagging is a very useful technique for evaluating and optimizing feature performance in software systems. In software systems with complex features, introducing a new feature by using the control and treatment mechanism will include multiple parameters that determine whether or not a particular treatment should be applied to the request. This paper explores the design of a methodology to pass in feature flags in the context of a Warehouse Management System to enable dynamic feature toggles based on the context of the request being processed. The framework uses modular strategies for control and treatment evaluation, enabling effective experimentation in complex system architectures.

Keywords: Feature Flag, Warehouse Management System, Software Engineering

Introduction

In today's modern world, the ability to deliver quality software and new functionalities over the period of a few days or hours represents the key factor for staying at the top of the list. Feature toggles, also known as feature flags or feature switches have emerged as a new tool to enable agile software development and continuous delivery as seen in [1].

Feature flagging is a technique that allows software teams to control the activation of certain features without affecting existing functionality deployed in production systems. In modern systems, especially those with complex and diverse use cases, the decision to enable or disable features often depends on several contextual factors, such as user attributes, system state, or specific business requirements. For example, the new feature to be implemented could only be released to clients in the Europe region prior to expanding globally. This would reduce the blast radius of errors that may be introduced by the new feature, while also providing a mechanism to turn off the feature during initial stages of release into the production system [2].

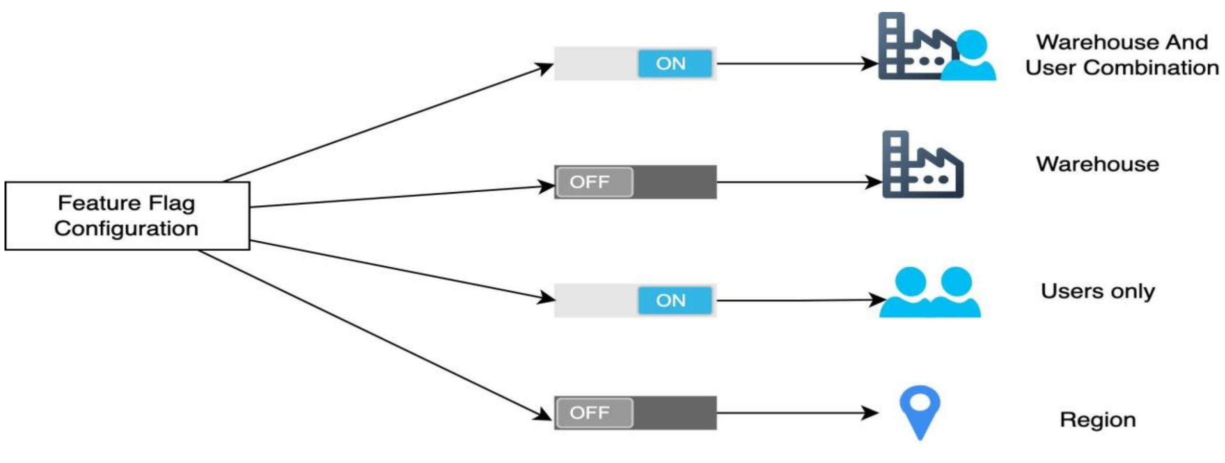

Fig 1: Use case

In this paper we explore the design and implementation of a feature flagging mechanism that can augment requests that pass into the system with control and treatment data. By using this system, features can be enabled based on the request attributes. This is particularly useful in the context of Warehouse Management Systems that are designed through an event driven microservice architecture [4]. In such systems, enabling or disabling features depend on several parameters passed through the request payload such as geographic location, warehouse name, warehouse type, user and activity type to name a few (shown in Fig 1). The framework presented in this paper can also double as a A/B testing mechanism in order to run new features in production parallelly with existing features to compare performance.

High Level Architecture

Warehouse Management Systems (WMS) typically receive events as a JSON payload which has all metadata associated with the event. The event originates from the WMS client which is received by a message queue and forwarded to downstream services. In the architecture presented in Fig2, the Treatment Service which can also be a Forward Proxy [5] to the WMS backend takes the responsibility assigning treatment and control attributes in the request. It does so by using a configuration store that contains static configuration which will be evaluated against the request to determine whether or not a certain treatment group can be applied to the request. The components used to read configuration, evaluate requests based on the configuration and assign treatment and control groups are all managed through a set of classes designed with the Mediator design pattern [6]. The following sections describe the implementation of this pattern and the individual classes involved in the form of a Low Level Design [7].

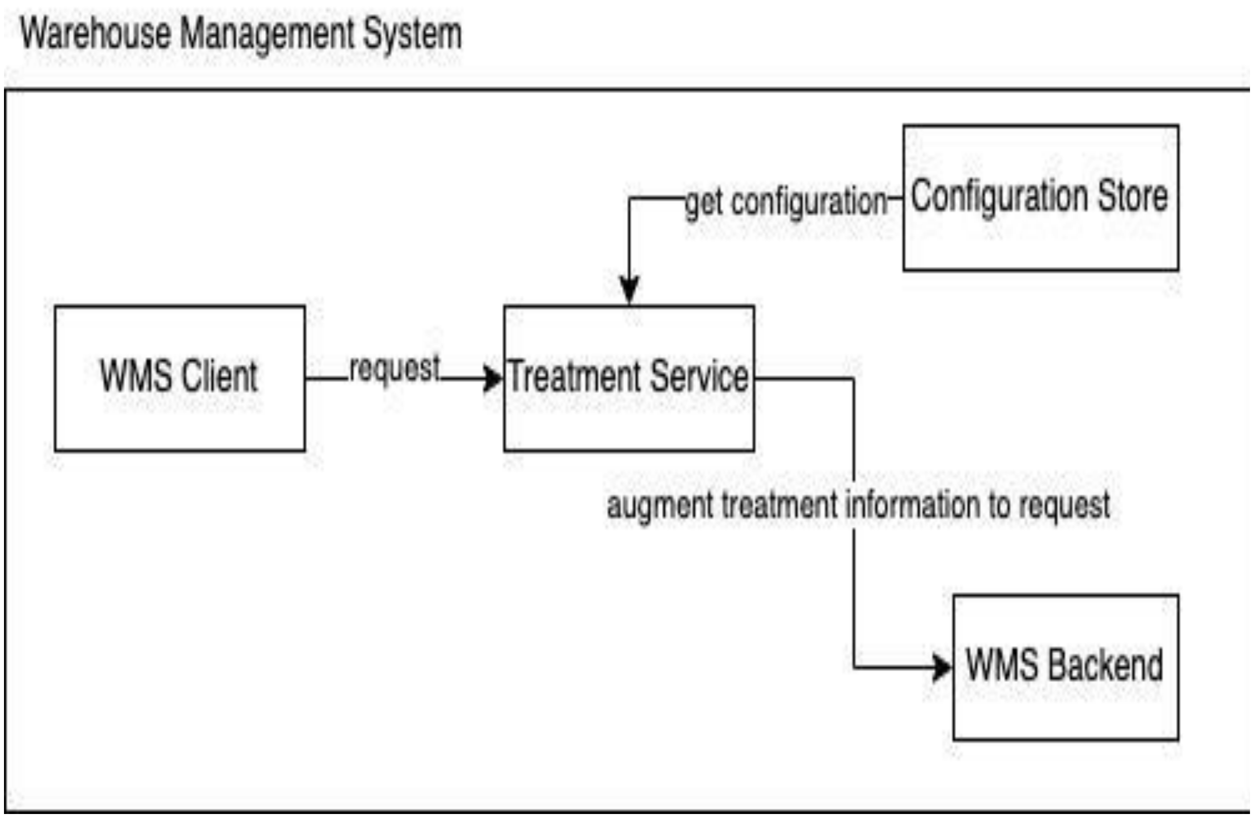

Fig2: High level architecture

Low Level Design

All the components shown in Fig 3 are part of the Treatment Service that intercepts requests from the WMS client prior to forwarding it to the WMS backend service(s). The components shown below can be abstracted into its SDK [8] and shared across multiple such forward proxy services.

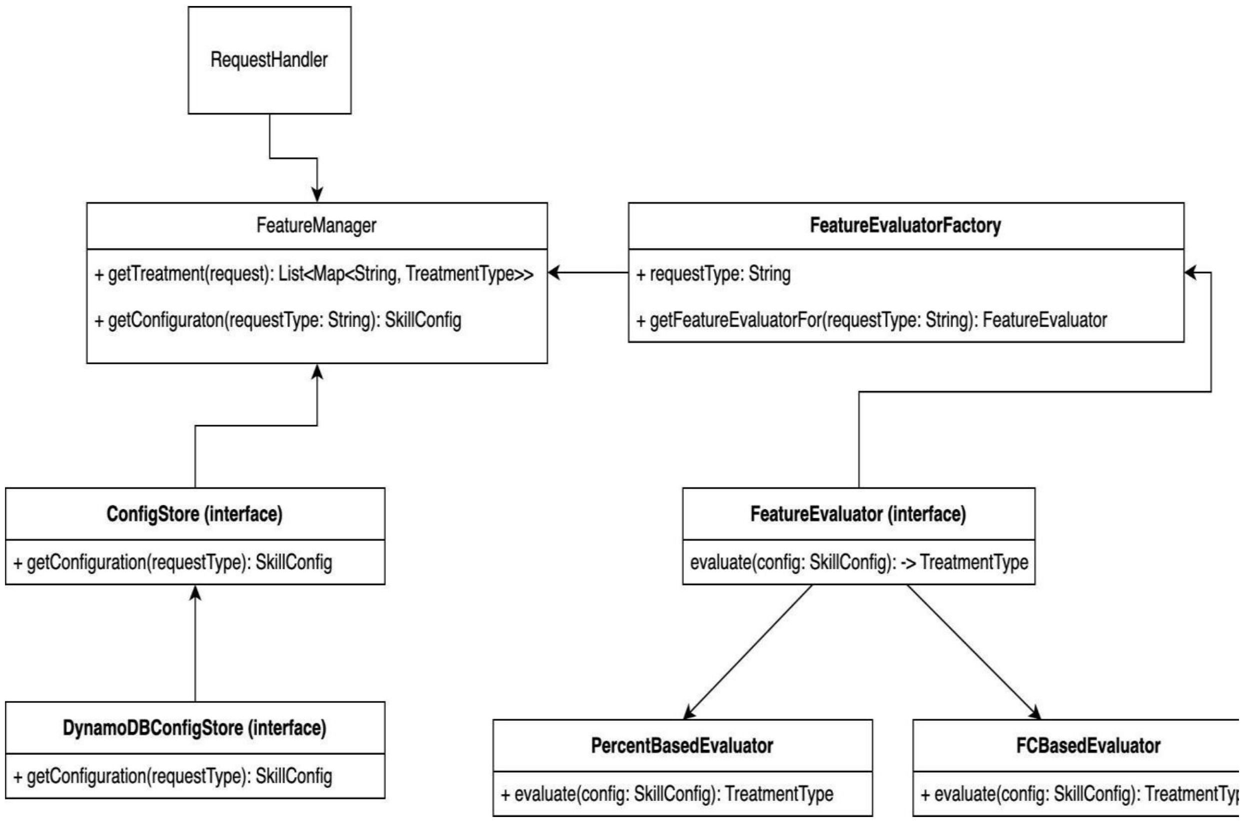
RequestHandler
FeatureManager
+ getTreatment(request): List<Map<String, TreatmentType>>
+ getConfiguraton(requestType: String): SkillConfig
FeatureEvaluatorFactory
+ requestType: String
+ getFeatureEvaluatorFor(requestType: String): FeatureEvaluator
ConfigStore (interface)
+ getConfiguration(requestType): SkillConfig
FeatureEvaluator (interface)
evaluate(config: SkillConfig): -> TreatmentType
DynamoDBConfigStore (interface)
+ getConfiguration(requestType): SkillConfig
PercentBasedEvaluator
+ evaluate(config: SkillConfig): TreatmentType
FCBasedEvaluator
+ evaluate(config: SkillConfig): TreatmentTy

Fig3: Components for feature evaluation

Request Handler

The request handler is a logical component that can be anything that receives events from the warehouse. This can be a consumer for pull based or long polling systems or an API endpoint handler. In Fig2 this role is played by the Treatment Service. In practice it can be any kind of component that acts as a forward proxy to the backend systems. This component gets the list of treatments that need to be applied for the request and their values as shown in the snippet below.

```
Map<String, TreatmentType> treatments = featureManager.getTreatments(request)
```

The list of treatments return looks something like this

```
{
	‘FEATURE_1’: ‘T’,
	‘FEATURE_2’: ‘C’
}
```

The map object above is included into the body of the request before forwarding the request downstream.

Skill Configuration

The skill configuration is a serializable object that can be stored in any kind of data store in the form of a serialized JSON [9] . The configuration object contains attributes that can be matched with request attributes to determine treatment and control groups. It can also have a random percentage based control or treatment evaluation, in which case there will be no attribute matching and will apply to any request. It can also be a combination of both.

```
public class SkillConfiguration:
	String schemaVersion = ‘v1’;
```

```
public class FCBasedSkillConfiguration extends SkillConfiguration:
        List<String, Boolean> warehouseNameToTreatment;

public class PercentBasedSkillConfiguration extends SkillConfiguration:
        Float threshold;
```

A sample of how this looks like is shown below

```
{
        warehouseNameToTreatment: {
                "WAREHOUSE_1": true,
                "WAREHOUSE_2": false
        }
}
```

Feature Manager

The design follows a Mediator Design pattern in which this component acts as the mediator. It owns the responsibility of fetching the skill configuration for a certain request type and subsequently uses the feature evaluator to calculate treatment type for the list of configurations that apply to the request type.

```
public Map<String, TreatmentType> getTreatments(request) {
        Map<String, SkillConfig> configurations = configurationStore.getConfigurationFor(request
                .getRequestType());
        If (Collections.isEmpty(configurations)) {
                return Collections.EMPTY_MAP;
        }
        Map<String, TreatmentType> treatmentsMap = new HashMap<>();
        for (Map.Entry<String, SkillConfig> config : configurations) {
                Feature Evaluator evaluator = featureEvaluatorFactory.getEvaluatorFor(
                        new Pair(config.key(), config.value()));
                TreatmentType treatmentType = evaluator.evaluate(request, config);
                treatmentsMap.put(config.getKey(), treatmentType);
        }

        return treatmentsMap
```

The object returned by the configuration store has a unique feature ID as the key and a list of SkillConfig objects that are attributes which determine whether a given feature can be activated for a certain request type. This is designed in such a way so that there can be multiple skill configurations that need to be matched with the request attributes in order to determine whether or not a treatment type can be applied to the request.

When a request arrives, the feature manager uses a unique attribute of the request which is the request type in this case, to pull out a list of skill configurations defined for that request type from the configuration store.

```
{
        "LABEL_PROCESSING_REQUEST": {
                "'FC_BASED_SKILL_CONFIGURATION'": {
                    . . .
                }
        }
}
```

If the incoming request has a request type of LABEL_PROCESSING_REQUEST and the warehouse identifier has a configuration defined, then it will be used to determine treatment type based on the configuration stored.

Configuration Store

The configuration store is an interface that exposes the method getConfiguration. Implementation specifics will vary depending on the type of store used. For example it can be a NoSQL [10] store like dynamo DB [11] or a blob store like AWS S3 [12].

```
public Map<String, Map<String, SkillConfig>> getConfiguration(requestType)
```

The motivation behind dedicating a separate component is to provide the flexibility of implementing any kind of data store for the configurations and decouple serialization and deserialization [13] from the application logic.

Feature Evaluator Factory

The feature evaluator factory follows a factory design pattern to fetch the evaluator implementation depending on the configuration object provided. This standardizes the evaluator interface and decouples evaluation logic which makes it flexible enough to add as many evaluator implementations as needed.

```
getEvaluatorFor(Pair<String, SkillConfig> configurationMetadata) {
        if (configMap.containsKey(configurationMetadata.getLeft())) {
                return configurationMap.get(configurationMetadata.getRight())
        }
        throw new ConfigurationDoesNotExistError(configurationMetadata.getLeft())
}
```

The feature evaluator factory maintains a map of configuration keys to the corresponding evaluator implementation that owns the logic to properly parse the content of SkillConfig and subsequently produce evaluation results.

Feature Evaluator

The feature evaluator component owns the logic of consuming the request and corresponding skill configurations, and determining whether or not to apply the treatment for the request. This implementation can range from randomly deciding to determining based on the context of the request.

```
public interface FeatureEvaluator {
        TreatmentType evaluate(Request request, Optional<SkillConfig> configurations);
}
```

Shown below is an implementation of a feature evaluator that randomly samples requests for treatment and control based on a threshold defined in the configuration.

```
public class RandomizedFeatureEvaluator implements FeatureEvaluator {
        TreatmentType evaluate(Request request, PercentBasedSkillConfiguration skillConfig) {
                if (generateRandomNumber() < skillConfig.getThreshold()) {
                        return TreatmentType.T;
                }
                return TreatmentType.C;
        }
}
```

In the above sections, we explored implementation specifics of the context based feature flagging system. Following sections discuss the benefits of having such a system in place, specifically when used in a WMS.

Results

Advantages of usage in Warehouse Management Systems

Warehouse management systems often need to expose features based on various factors like geography, warehouse, logged in user and the application skill being used. The presence of such a context aware and dynamic feature toggling mechanism provides the following performance benefits:

Streamline the Release Process

When releasing new features on the WMS client, the randomized feature toggling mechanism can be used to send a very low percentage of requests to be processed by the feature branch of code. Appropriate logging and metrics can be used for errors if any. Since the configuration parameters can be changed dynamically, the percentage can be increased or decreased based on the performance of the feature. This can also be applied at a warehouse and user level in order to target the feature only for a particular user working at a certain warehouse, and gradually increase the percentage based on feedback from the user. This opens up the possibility of a User Acceptance Testing phase prior to fully releasing the feature.

A/B testing

A/B testing is a method for comparing two versions of an application to see which performs better. By using the architecture shown above, the request handler can choose to send only a certain number or percentage of requests to the new feature after the code change has been deployed to production. This allows the flexibility of deploying code changes without having to worry about production impact while also enabling the new feature to run in a “shadow” mode to assess the performance of the application with and without the new feature. For example, consider a feature where a machine learning model is currently automating a decision in the warehouse, and a new model needs to be deployed. By using the feature flag mechanism described in this feature, the new model can be deployed in parallel with the old feature while flexibly configuring when and how the new model needs to be activated. This enables the new model to run in parallel to the old model while allowing us to compare performance differences between the two.

Performance

While there are immediate benefits that come from the ability to deploy features seamlessly and efficiently, there can be certain latency issues caused when the request handler frequently fetches skill configuration from the configurations store for each request. This can be avoided by using a caching mechanism that locally caches the configuration most recently fetched. A distributed cache like Redis can be used to monitor and update the cache in order to keep it fresh. Well tested strategies [2] can be implemented to avoid issues like cache-miss or cache freshness in order to mitigate latency issues that may arise due to this.

Conclusion

This paper presents a modular system design that can be used to implement a dynamic and context aware feature toggling system tailored to be used in a warehouse management system. It elaborates on individual components using a low level class design, which can be coupled with concepts like Dependency Injection [14] to make it even more flexible and modular. Implementation of this system has been proven to significantly augment the agile release process in warehouse management systems by unblocking code deployment and providing levers to modify the way a new feature is enabled in production. The concepts presented in this paper can also be extended to other applications that have similar use-cases where the application uses an event driven microservice architecture and the features exposed vary depending on multiple factors.

References

1. N. Michas, Feature Flags: How to Decouple Code from Features, [online] Available: https://medium.com/swlh/feature-flags-how-to-decouple-code-from-features-b36b59792e0c.
2. Gulgundi, M.S., 2022. CONTROLLING DEGRADATION AND BLAST RADIUS. SOUTH ASIAN ACADEMIC RESEARCH JOURNALS (www. saarj. com), p.98.
3. A. A. Shvidkiy, A. A. Savelieva and A. A. Zarubin, "Caching Methods Analysis for Improving Distributed Storage Systems Performance," 2021 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO, Kaliningrad, Russia, 2021, pp. 1-5, doi: 10.1109/SYNCHROINFO51390.2021.9488328.
4. Michelson, B.M., 2006. Event-driven architecture overview. Patricia Seybold Group, 2(12), pp.10-1571.
5. "Forward Proxy - Radware Cyberpedia," Radware. [Online]. Available: https://www.radware.com/cyberpedia/application-delivery/forward-proxy/.
6. Menjiba, F.A.A., Arias, H.R.P. and Villarroel, M.J.L., 2011. Software component development based on the mediator pattern design: The interactive graphic organizer case. IEEE Latin America Transactions, 9(7), pp.1105-1111.
7. Reeves, J.W., 1992. What is software design. C++ Journal, 2(2), pp.14-12.
8. "What is an SDK?" Amazon Web Services. [Online]. Available: https://aws.amazon.com/what-is/sdk/.
9. Bourhis, P., Reutter, J.L., Suárez, F. and Vrgoč, D., 2017, May. JSON: data model, query languages and schema specification. In Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems (pp. 123-135).
10. Strauch, C., Sites, U.L.S. and Kriha, W., 2011. NoSQL databases. Lecture Notes, Stuttgart Media University, 20(24), p.79.
11. Sivasubramanian, S., 2012, May. Amazon dynamoDB: a seamlessly scalable non-relational database service. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (pp. 729-730).
12. Murty, J., 2008. Programming amazon web services: S3, EC2, SQS, FPS, and SimpleDB. " O'Reilly Media, Inc.".
13. Tauro, C.J., Ganesan, N., Mishra, S.R. and Bhagwat, A., 2012. Object Serialization: A Study of Techniques of Implementing Binary Serialization in C++, Java & .NET. Intl J of Computer Applications, 45, pp.25-29.
14. Yang, H.Y., Tempero, E. and Melton, H., 2008, March. An empirical study into use of dependency injection in java. In 19th Australian Conference on Software Engineering (aswec 2008) (pp. 239-247). IEEE.