International Journal of Core Engineering & Management

Volume-6, Issue-06, 2019 ISSN No: 2348-9510

A GUIDE TO IMPLEMENTING OBSERVABILITY IN SALESFORCE

Chirag Amrutlal Pethad
PetSmart.com, LLC, Stores and Services
Phoenix, Arizona, USA
ChiragPethad@live.com
ChiragPethad@gmail.com

Abstract

Observability [2][3][6] refers to the ability to infer the internal state of a system based on its external outputs. In the context of Salesforce, observability encompasses monitoring and gaining insights into various aspects of the platform's performance, data flow, user interactions, and integrations. Unlike traditional monitoring, observability emphasizes understanding rather than just tracking metrics, enabling proactive management and rapid issue resolution. This document discusses the implementation of observability in Salesforce, highlighting the importance of Observability, its key components, some of the challenges in implementing Observability, Step by Step guide to implementing the framework components and provides pseudocodes for the implementations, emphasizing application reliability, maintainability and performance.

Keywords: Logging, Observability, Performance, Debugging, Auditing, Monitoring, Analysis, Framework, Events, Traces, Maintainability, Reliability, Monitoring

I. INTRODUCTION

Observability [2][3][6] refers to the ability to understand and monitor the internal state of the system by observing its outputs, such as logs, metrics, and traces. This is crucial for diagnosing issues, improving performance, and ensuring the reliability of any application. As organizations increasingly rely on Salesforce for their CRM and business operations, ensuring observability becomes essential for detecting issues, understanding system behavior, and improving overall performance of Salesforce applications. Salesforce Apex is a powerful programming language designed specifically for building custom applications on the Salesforce platform. As with any robust development environment, effective logging is crucial for debugging, monitoring, and maintaining applications. A well-implemented and robust logger framework in Apex can significantly enhance the development process by providing detailed insights into application behavior, facilitating quicker issue resolution, and improving overall system reliability. It also helps developers track the execution flow, capture errors, and gain insights into application behavior. This white paper explores the challenges, key components, principles, benefits, and best practices for implementing an effective Observability strategy and logger framework in Salesforce Apex to enhance application reliability, maintainability and performance. This guide provides step by step instructions and pseudocode examples to implement all the logic required for a robust framework.

International Journal of Core Engineering & Management

Volume-6, Issue-06, 2019 ISSN No: 2348-9510

II. KEY COMPONENTS OF OBSERVABILITY

1. Monitoring and Metrics[3][6]

Monitoring and measuring performance indicators such as API response times, data throughput, and system resource utilization.

- Apex Exception Emails: Salesforce automatically sends emails to developers when unhandled exceptions occur in Apex code. These are critical for catching runtime errors.
- Governor Limits Monitoring: Salesforce imposes limits on resource usage to ensure that no single tenant can monopolize shared resources. Monitoring governor limits helps prevent performance degradation and ensures compliance with Salesforce's multi-tenant architecture.
- Event Monitoring: Provides detailed metrics about user interactions, such as API usage, page views, and performance issues. This is particularly useful for understanding user behavior and diagnosing performance bottlenecks.

2. Logging[6][7]

Recording detailed information about transactions, errors, user activities, and system events to facilitate troubleshooting and auditing.

- Debug Logs: Capture detailed events, including database operations (DML), system events, and Apex code execution. Debug logs are essential for tracing issues in real-time.
- System Logs: Provide information about system-wide events, such as login attempts, errors, and system changes. These are useful for auditing and security monitoring.

3. Tracing[6]

Capturing end-to-end transaction paths to analyze dependencies, bottlenecks, and performance issues across different Salesforce components and integrations.

- Apex Profiling: Helps in understanding the execution flow of Apex code, including SOQL queries, DML statements, and method calls. Profiling is crucial for performance tuning and debugging.
- Transaction Logs: Track transactions across different components, including integrations, workflows, and triggers. This is important for tracing the flow of data and understanding the sequence of events that led to a particular state.

4. Real Time Alerts and Events[4]

Notifying and reacting to significant occurrences within Salesforce, such as platform events, API calls, or external integrations.

- Health Check [4]: Monitors the security health of your Salesforce instance by evaluating the settings against Salesforce's recommended baseline. Any deviations trigger alerts, helping to maintain a secure environment.
- Custom Alerts [4]: Salesforce allows you to set up custom alerts for specific conditions, such as high CPU usage, long-running queries, or failed integrations. These alerts help in proactively managing system health.

5. Analytics and Reporting

Gather insight into the systems health and activity to ensure system stability and compliance.

- Salesforce Reports and Dashboards: Provide insights into system performance, user activity, and data trends. Custom reports can be created to monitor specific metrics related to observability.
- Event Monitoring Analytics: This feature offers detailed insights into how Salesforce is being used, helping to identify anomalies, optimize performance, and ensure compliance.

6. External Integrations[7]

Aggregate logs from all the integrated applications to get a comprehensive view of the entire system.

- External Monitoring Tools: Salesforce can be integrated with external observability tools like New Relic, Splunk, or Datadog. These tools provide advanced monitoring, log aggregation, and alerting capabilities, enabling a more comprehensive observability strategy.
- Salesforce Shield: Provides additional logging and monitoring features for compliance and auditing, including Event Monitoring, Field Audit Trail, and Platform Encryption.

7. Error Tracking and Handling[4]

Track the system and events in real time and handle the errors to prevent system outages and failures.

- Apex Error Handling: Implementing robust error handling mechanisms in Apex code ensures that errors are caught and managed appropriately, preventing system failures and data corruption.
- Platform Events: Salesforce Platform Events can be used to log and monitor events in real-time, which can be critical for maintaining observability in event-driven architectures.

8. Performance Monitoring

Test the system regularly to ensure no introduction of buggy code can bring down the system.

- Apex Test Execution: Regularly running Apex tests helps ensure that code changes do not introduce performance regressions or new issues.
- Page Performance Analysis: Salesforce provides tools to analyze the performance of Visualforce pages and Lightning components, helping to identify and resolve performance bottlenecks.

III. BENEFITS OF OBSERVABILITY

Observability in Salesforce provides a comprehensive understanding of the system's internal state, enabling organizations to monitor, diagnose, and optimize their Salesforce environments effectively. Logging is a fundamental aspect of software development that involves recording various types of events and information during the execution of a program. Some of the key benefits include:

1. Improved System Reliability[3]

- Proactive Issue Detection: With robust observability, you can detect and address issues before they impact users. Monitoring tools alert you to potential problems, such as reaching governor limits or performance bottlenecks, allowing you to take corrective action proactively.
- Faster Incident Response: Detailed logs and metrics provide the necessary information to quickly diagnose and resolve issues. This reduces downtime and minimizes the impact on business operations.

International Journal of Core Engineering & Management

Volume-6, Issue-06, 2019 ISSN No: 2348-9510

2. **Enhanced Performance Optimization[6]**
- Identifying Bottlenecks: Observability helps identify performance bottlenecks, such as slow-running queries, inefficient code, or resource contention. By pinpointing these issues, you can optimize the performance of your Salesforce instance.
- Optimized Resource Utilization: By monitoring resource usage (like CPU, memory, and database operations), you can ensure that your Salesforce environment is operating efficiently and within the prescribed limits.

3. **Better User Experience[3]**
- Monitoring User Behavior: Event monitoring and analytics provide insights into how users interact with Salesforce. Understanding user behavior helps in identifying areas where the user experience can be improved, leading to increased user satisfaction.
- Minimized Disruptions: Continuous monitoring and proactive issue management minimize disruptions, ensuring a smooth and consistent user experience.

4. **Enhanced Security and Compliance**
- Real-Time Security Monitoring: Observability allows for real-time monitoring of security events, such as unauthorized access attempts or changes to sensitive data. This helps in maintaining the security posture of your Salesforce environment.
- Compliance Reporting: Detailed logs and audit trails are crucial for compliance with industry regulations and internal policies. Salesforce observability features like Event Monitoring and Salesforce Shield provide the necessary tools to meet compliance requirements.

5. **Increased Developer Productivity[3]**
- Easier Debugging: Detailed logs, error tracking, and profiling tools make it easier for developers to debug issues, reducing the time spent on troubleshooting and fixing bugs.
- Continuous Improvement: By continuously monitoring and analyzing system performance, developers can iteratively improve the quality and performance of their code, leading to more stable and efficient applications.

6. **Scalability and Flexibility**
- Scalable Architecture: Observability enables you to monitor and manage the scalability of your Salesforce applications. By understanding how your system behaves under different loads, you can scale your architecture to meet growing demands effectively.
- Flexible Integrations: Salesforce's observability can be extended with external monitoring tools, allowing for more sophisticated analytics, alerting, and visualization. This flexibility ensures that your observability strategy can grow and adapt to changing needs.

7. **System Transparency**
- Holistic View of System Health: Observability provides a comprehensive view of your Salesforce environment, encompassing all components, from the application layer to the underlying infrastructure. This holistic visibility ensures that no aspect of the system is overlooked.

- Correlation of Events: By correlating different events (like API calls, user actions, and system errors), observability tools help you understand the cause-and-effect relationships within your Salesforce environment. This leads to more accurate diagnoses and targeted solutions.

8. **Enhances Decision Making with Data**

- Informed Decision-Making: Observability data can be used to make informed decisions about system changes, upgrades, or new feature implementations. By understanding the current state of the system, you can make decisions that improve performance and reliability.
- Strategic Insights: The insights gained from monitoring user interactions, system performance, and security events can inform strategic initiatives, such as improving customer engagement or optimizing operational efficiency.

IV. CHALLENAGES OF IMPLEMENTING OBSERVABILITY

1. Complexity

Salesforce environments can be complex with customizations, integrations, and large datasets, making it challenging to capture and analyze relevant data comprehensively.

2. Data Volume

Handling large volumes of logs, metrics, and traces requires robust storage and processing capabilities.

3. Cost

Implementing comprehensive observability may require investment in tools, infrastructure, and expertise. Some of the tools most widely used:

- Salesforce Event Monitoring: Provides detailed logs of user activity, API usage, and system events for compliance and operational visibility. However, Event Monitoring is an Add-On product which can be purchased on Salesforce platform.
- Detailed Debug Logs: Built-in Salesforce feature for capturing detailed logs of transaction execution, helping diagnose issues in custom code and configurations. Debug logs work well for development purpose. However, it is not effective and feasible to capture Debug logs in real time for production workloads.
- Third-Party Monitoring Tools: Platforms like Splunk, Datadog, and New Relic offer integrations with Salesforce for comprehensive logging, monitoring, and analytics capabilities. These third-party tools need expertise as well as licensing to integrate with Salesforce.
- Open-Source Tools: There are open-source tools available like RFLIB, Nebula Logger, or Tools / Plugins like Grafana; However, it needs expertise in implementing these tools and solutions. Additionally, you need you own infrastructure to configure and host these solutions.

V. NEED FOR A LOGGER FRAMEWORK

To mitigate the challenges and risks mentioned in this paper, a robust Logging Framework is required. It helps serves several purposes:

A. Error Tracking: Capture and Log exception to facilitate troubleshooting.
B. Execution Flow Monitoring: Record inputs, outputs, and significant events to understand application behavior.
C. Audit and Compliance: Maintain logs for audit and compliance requirements.
D. Performance Monitoring: Track execution times to identify performance bottlenecks.
E. Standardization: Standardize log messages and formatting.
F. Easy Retrieval: Capturing all the logs consistently in a single place allows easily retrieval and querying.

VI. KEY FEATURES OF A LOGGER FRAMEWORK

An effective logger framework in Salesforce Apex should include the following features:

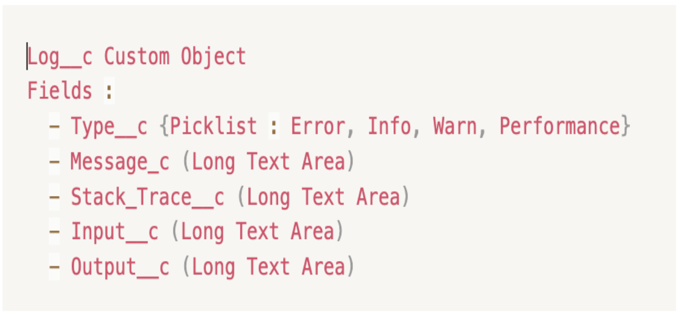
A. Log Levels: Support for different log levels (e.g., PERFORMANCE, INFO, WARN, ERROR) to categorize the importance and type of log messages.
B. Configurable Logging: Ability to enable or disable logging dynamically based on different criteria such as environment, or specific conditions.
C. Persistent Storage: Mechanism for storing log messages in a persistent manner, such as in custom objects, to facilitate long-term analysis and auditing.
D. Contextual Information: Inclusion of contextual information (e.g., user details, execution context, Inputs, Outputs, etc.) to provide more meaningful and actionable logs.Standardization: Standardize log messages and formatting.
E. Performance Considerations: Efficient logging that minimizes performance overhead and avoids impacting the application's responsiveness.

VII. IMPLEMENTATION OF LOGGING FRAMEWORK IN APEX

The first step in implementing a logger framework is its design. A typical logger framework in Apex consists of the following core components:

1. Log Object[11]

We start by defining a custom object in Salesforce to store log records persistently.



```
Log__c Custom Object
Fields :
  - Type__c {Picklist : Error, Info, Warn, Performance}
  - Message_c (Long Text Area)
  - Stack_Trace__c (Long Text Area)
  - Input__c (Long Text Area)
  - Output__c (Long Text Area)
```

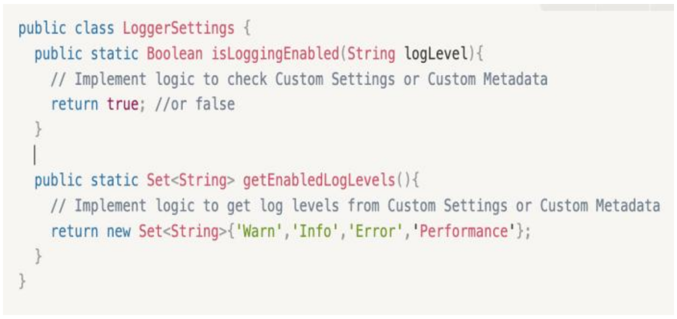
Figure 1: Log Object Metadata

International Journal of Core Engineering & Management

Volume-6, Issue-06, 2019 ISSN No: 2348-9510

2. Configuration Settings

We create Custom settings or Custom metadata to manage logging configurations dynamically (Like a Feature Flag - ON/OFF switch, Switch for Log Levels, etc.). And we create an Apex Class that will be responsible to get the logger settings.

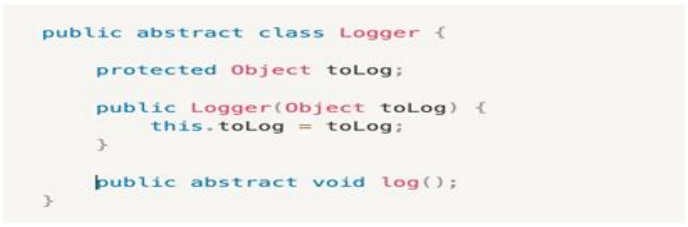


```
public class LoggerSettings {
  public static Boolean isLoggingEnabled(String logLevel){
    // Implement logic to check Custom Settings or Custom Metadata
    return true; //or false
  }
  |
  public static Set<String> getEnabledLogLevels(){
    // Implement logic to get log levels from Custom Settings or Custom Metadata
    return new Set<String>{'Warn','Info','Error','Performance'};
  }
}
```

Figure 2: Utility Class for getting the Configuration of Log Levels

3. **Base Class for Loggers[8]**

Next, We create an abstract base class for all the different types of Loggers like WARN Logger, Info Logger, Error Logger and Performance Logger. This constructor of this class takes in an Object or a message that needs to be logged.

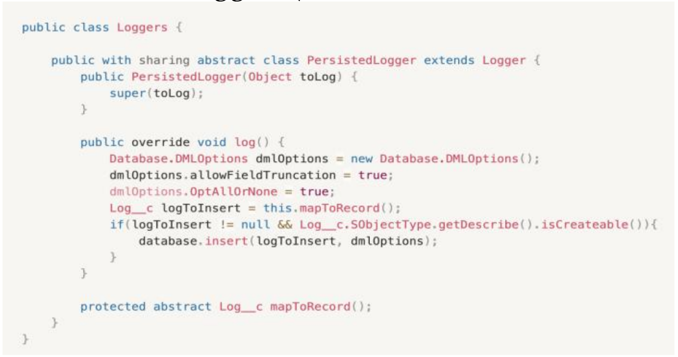


```
public abstract class Logger {

    protected Object toLog;

    public Logger(Object toLog) {
        this.toLog = toLog;
    }

    public abstract void log();
}
```

Figure 3: Abstract Base Class for Logger

4. Loggers Implementation

Next we create a Loggers class with an Inner Abstract Class [8] that extends the capability of Logger Class by implementing a generic implementation of the log method to insert log records into the Log object. And it declares a method that all the advanced loggers (Warn, Info, Error and Performance) will implement.

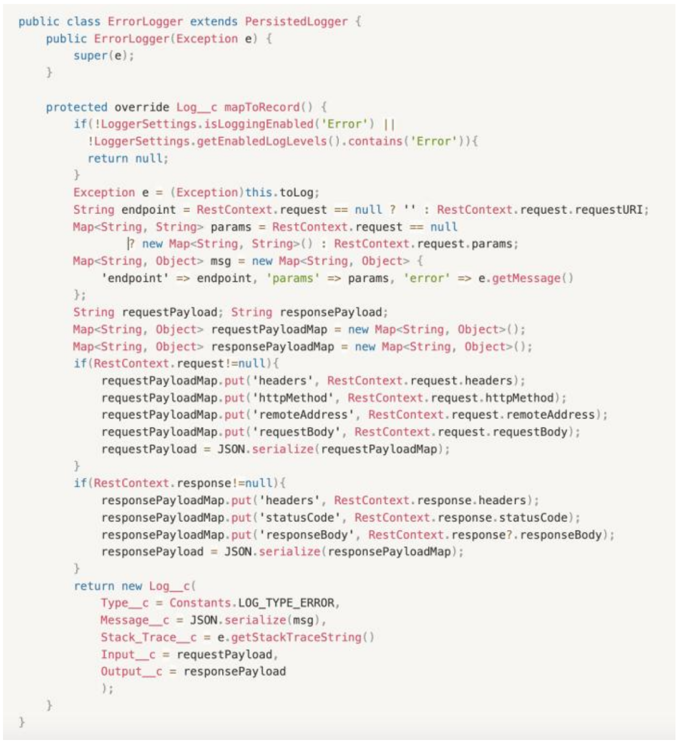


```
public class Loggers {

    public with sharing abstract class PersistedLogger extends Logger {
        public PersistedLogger(Object toLog) {
            super(toLog);
        }

        public override void log() {
            Database.DMLOptions dmlOptions = new Database.DMLOptions();
            dmlOptions.allowFieldTruncation = true;
            dmlOptions.OptAllOrNone = true;
            Log__c logToInsert = this.mapToRecord();
            if(logToInsert != null && Log__c.SObjectType.getDescribe().isCreateable()){
                database.insert(logToInsert, dmlOptions);
            }
        }

        protected abstract Log__c mapToRecord();
    }
}
```

Figure 4: Default Implementation of Logger Class to persist log records in the Log custom object.

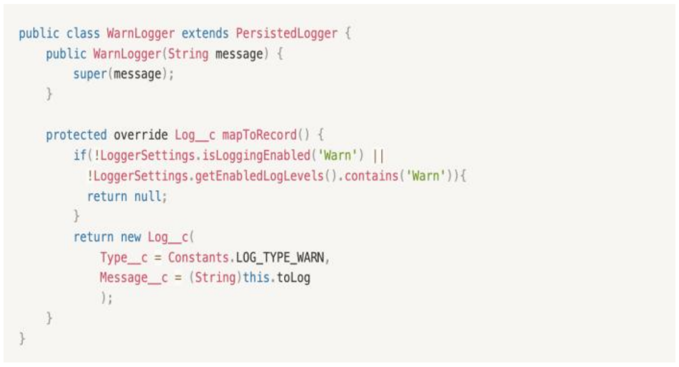
5. Advanced Loggers Implementation

We provide the implementation for all the types of Advanced Loggers. We will implement them as Inner classes[8] of the Loggers class. Fist we implement Error Logger. This class handles logging the error conditions as well request and response from the REST Context as well.
Error Logger creates a log record using type defined as Error and details about the error and REST Context information if available in the current transaction.



```
public class ErrorLogger extends PersistedLogger {
    public ErrorLogger(Exception e) {
        super(e);
    }

    protected override Log__c mapToRecord() {
        if(!LoggerSettings.isLoggingEnabled('Error') ||
          !LoggerSettings.getEnabledLogLevels().contains('Error')){
          return null;
        }
        Exception e = (Exception)this.toLog;
        String endpoint = RestContext.request == null ? '' : RestContext.request.requestURI;
        Map<String, String> params = RestContext.request == null
                |? new Map<String, String>() : RestContext.request.params;
        Map<String, Object> msg = new Map<String, Object> {
            'endpoint' => endpoint, 'params' => params, 'error' => e.getMessage()
        };
        String requestPayload; String responsePayload;
        Map<String, Object> requestPayloadMap = new Map<String, Object>();
        Map<String, Object> responsePayloadMap = new Map<String, Object>();
        if(RestContext.request!=null){
            requestPayloadMap.put('headers', RestContext.request.headers);
            requestPayloadMap.put('httpMethod', RestContext.request.httpMethod);
            requestPayloadMap.put('remoteAddress', RestContext.request.remoteAddress);
            requestPayloadMap.put('requestBody', RestContext.request.requestBody);
            requestPayload = JSON.serialize(requestPayloadMap);
        }
        if(RestContext.response!=null){
            responsePayloadMap.put('headers', RestContext.response.headers);
            responsePayloadMap.put('statusCode', RestContext.response.statusCode);
            responsePayloadMap.put('responseBody', RestContext.response?.responseBody);
            responsePayload = JSON.serialize(responsePayloadMap);
        }
        return new Log__c(
            Type__c = Constants.LOG_TYPE_ERROR,
            Message__c = JSON.serialize(msg),
            Stack_Trace__c = e.getStackTraceString()
            Input__c = requestPayload,
            Output__c = responsePayload
            );
    }
}
```

Figure 6: Performance Logger Class Implementation
We continue implementing Info (Figure 8) and Warn Loggers (Figure 7).



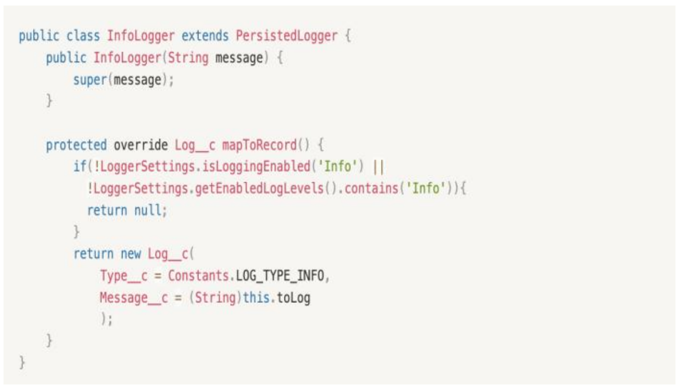
```
public class WarnLogger extends PersistedLogger {
    public WarnLogger(String message) {
        super(message);
    }

    protected override Log__c mapToRecord() {
        if(!LoggerSettings.isLoggingEnabled('Warn') ||
          !LoggerSettings.getEnabledLogLevels().contains('Warn')){
          return null;
        }
        return new Log__c(
            Type__c = Constants.LOG_TYPE_WARN,
            Message__c = (String)this.toLog
            );
    }
}
```

Figure 7: Warning Logger Class Implementation

iJCEM
International Journal of Core Engineering & Management

International Journal of Core Engineering & Management

Volume-6, Issue-06, 2019 ISSN No: 2348-9510

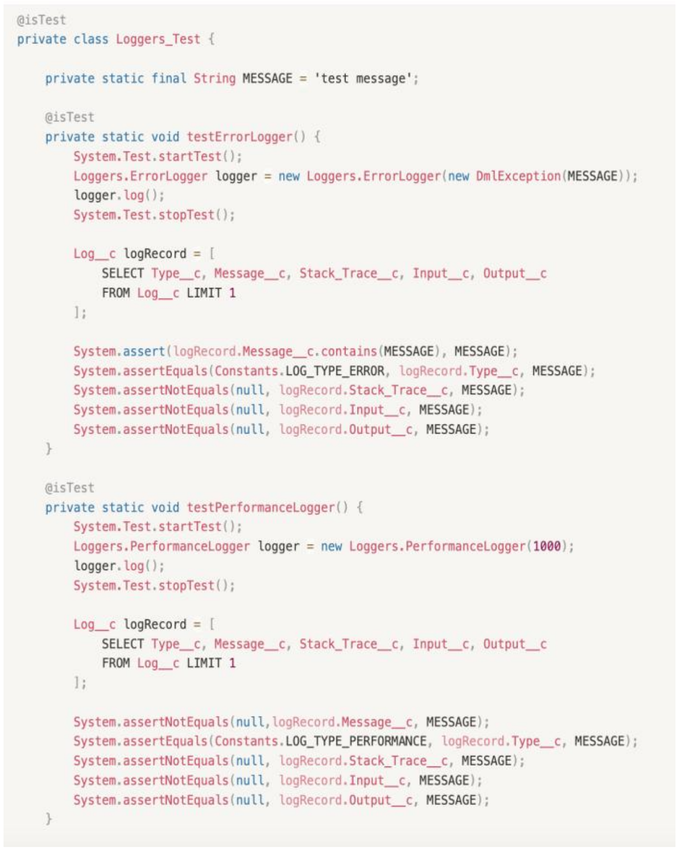


```
public class InfoLogger extends PersistedLogger {
    public InfoLogger(String message) {
        super(message);
    }

    protected override Log__c mapToRecord() {
        if(!LoggerSettings.isLoggingEnabled('Info') ||
          !LoggerSettings.getEnabledLogLevels().contains('Info')){
          return null;
        }
        return new Log__c(
            Type__c = Constants.LOG_TYPE_INFO,
            Message__c = (String)this.toLog
            );
    }
}
```

Figure 8: Information / Info Logger Class Implementation

6. Apex Test Loggers

Next we demonstrate how to write Apex test[10] to validate the functionality of these Advanced Loggers.



```
@isTest
private class Loggers_Test {

    private static final String MESSAGE = 'test message';

    @isTest
    private static void testErrorLogger() {
        System.Test.startTest();
        Loggers.ErrorLogger logger = new Loggers.ErrorLogger(new DmlException(MESSAGE));
        logger.log();
        System.Test.stopTest();

        Log__c logRecord = [
            SELECT Type__c, Message__c, Stack_Trace__c, Input__c, Output__c
            FROM Log__c LIMIT 1
        ];

        System.assert(logRecord.Message__c.contains(MESSAGE), MESSAGE);
        System.assertEquals(Constants.LOG_TYPE_ERROR, logRecord.Type__c, MESSAGE);
        System.assertNotEquals(null, logRecord.Stack_Trace__c, MESSAGE);
        System.assertNotEquals(null, logRecord.Input__c, MESSAGE);
        System.assertNotEquals(null, logRecord.Output__c, MESSAGE);
    }

    @isTest
    private static void testPerformanceLogger() {
        System.Test.startTest();
        Loggers.PerformanceLogger logger = new Loggers.PerformanceLogger(1000);
        logger.log();
        System.Test.stopTest();

        Log__c logRecord = [
            SELECT Type__c, Message__c, Stack_Trace__c, Input__c, Output__c
            FROM Log__c LIMIT 1
        ];

        System.assertNotEquals(null,logRecord.Message__c, MESSAGE);
        System.assertEquals(Constants.LOG_TYPE_PERFORMANCE, logRecord.Type__c, MESSAGE);
        System.assertNotEquals(null, logRecord.Stack_Trace__c, MESSAGE);
        System.assertNotEquals(null, logRecord.Input__c, MESSAGE);
        System.assertNotEquals(null, logRecord.Output__c, MESSAGE);
    }
```

Figure 9: Apex Text class to test the framework

VIII. BEST PRACTICES FOR LOGGING

1. Performance Considerations: Ensure logging does not significantly impact code performance. Avoid logging excessively within loops or frequently called methods.

2. Log only Meaningful Information [7]: Ensure that log messages are clear, concise, and provide enough context to understand the event or error. Be selective about what you log to avoid cluttering the logs with too much information. Focus on key actions, such as significant data changes, errors, and decision points in your code.

3. Set Appropriate Log Levels[7]: Adjust the debug log levels (e.g., ERROR, WARN, INFO, DEBUG) based on the situation. For production environments, use more restrictive levels like ERROR or WARN to minimize performance impact, while in development or troubleshooting, INFO or DEBUG can provide more detailed insights.

4. Secure Sensitive Information[7]: Avoid logging sensitive information such as passwords, personal data, or confidential business information to protect privacy and comply with regulations.

5. Use Logging for Monitoring and Alerts: Set up monitoring and alerting based on log entries to proactively address issues and maintain system health.

6. Establish Retention policy and Clean Up Logs: Periodically review and clean up log entries to manage storage and ensure the logging framework remains efficient. Implement automated processes to cleanup logs regularly.

IX. CONCLUSION

- Observability is indispensable for maintaining optimal performance, reliability, and user satisfaction in Salesforce environments.
- Incorporating observability practices not only mitigates risks but also enhances the overall efficiency and effectiveness of Salesforce implementations, thereby driving business success in an increasingly competitive landscape.
- Implementing a robust logger framework in Salesforce is essential for effective debugging, monitoring, and auditing.
- By leveraging the features and best practices outlined in this white paper, developers can create a robust logging solution that enhances their ability to manage and optimize Salesforce applications.
- With a well-implemented logger framework, businesses can improve system reliability, expedite issue resolution, and drive better decision-making based on comprehensive insights into application behavior.
- Adopting best practices such as performance considerations, configurable logging levels, secure logging, and comprehensive testing further strengthens the reliability and efficiency of the logger framework.

International Journal of Core Engineering & Management

Volume-6, Issue-06, 2019 ISSN No: 2348-9510

REFERENCE

1. Log Management Best Practices - https://newrelic.com/resources/white-papers/log-management-best-practices
2. Observability - https://engineering.salesforce.com/what-is-observability-d175eb6cd2e4/
3. Observability - https://newrelic.com/blog/best-practices/what-is-observability
4. Observability Design Patterns - https://engineering.salesforce.com/5-design-patterns-for-building-observable-services-d56e7a330419/
5. Observability in Salesforce - https://www.apexhours.com/observability-framework-in-salesforce/
6. https://www.influxdata.com/what-is-observability/
7. Log Management Best Practices - https://newrelic.com/resources/white-papers/log-management-best-practices
8. Apex Developer Guide - https://developer.salesforce.com/docs/atlas.enus.apexcode.meta/apexcode/apex_classes.htm?q=Apex%20Class
9. Custom Metadata Types - https://help.salesforce.com/s/articleView?id=sf.custommetadatatypes_overview.htm&type=5
10. Apex Testing - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_testing.htm
11. Objects in Salesforce - https://help.salesforce.com/s/articleView?id=sf.dev_objectcreate_task_parent.htm&type=5
12. https://www.tandfonline.com/doi/abs/10.1080/08853134.1996.10754049
13. https://journals.sagepub.com/doi/abs/10.1177/0886368715581959
14. https://www.emerald.com/insight/content/doi/10.1108/08858629710188036/full/html
15. https://www.tandfonline.com/doi/abs/10.1080/07408170490487777
16. https://pubsonline.informs.org/doi/abs/10.1287/mnsc.2013.1809
17. https://pubsonline.informs.org/doi/abs/10.1287/msom.1120.0424