SPRING FRAMEWORK

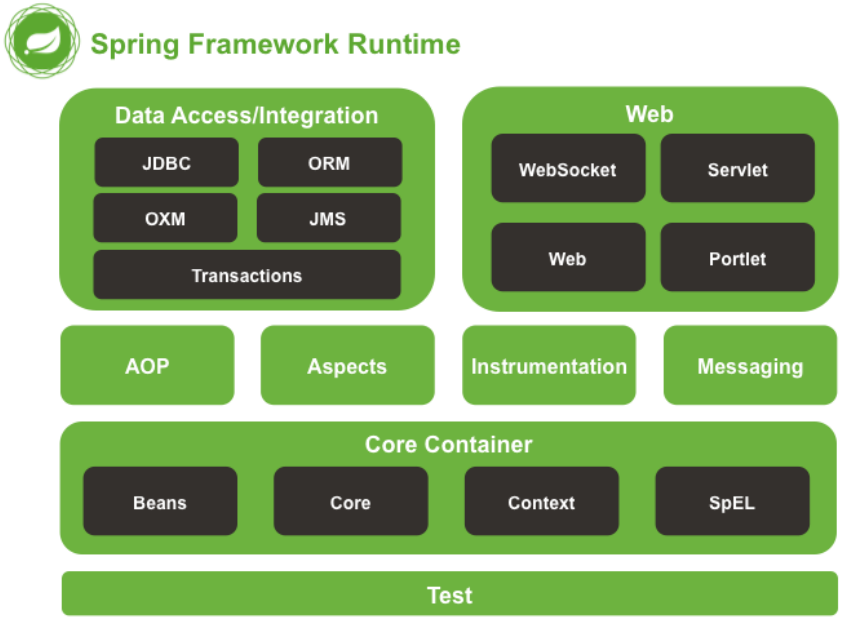
G5: FINAL DELIVERABLES

SUBMITTED BY
JEFFY J POOZHITHARA
MONALI CHANDURKAR
PINKI PATEL

1 CONTENTS

2 INTRODUCTION

2.1 SPRING FRAMEWORK



The spring framework is one of the most popular application development framework for enterprise java. It is an open source Java platform since 2003 that supports all major application servers and JEE standards. Its major features are the Inversion of control and dependency injection control container for the Java platform. It handles the infrastructure details so developers can focus on the application functionality.

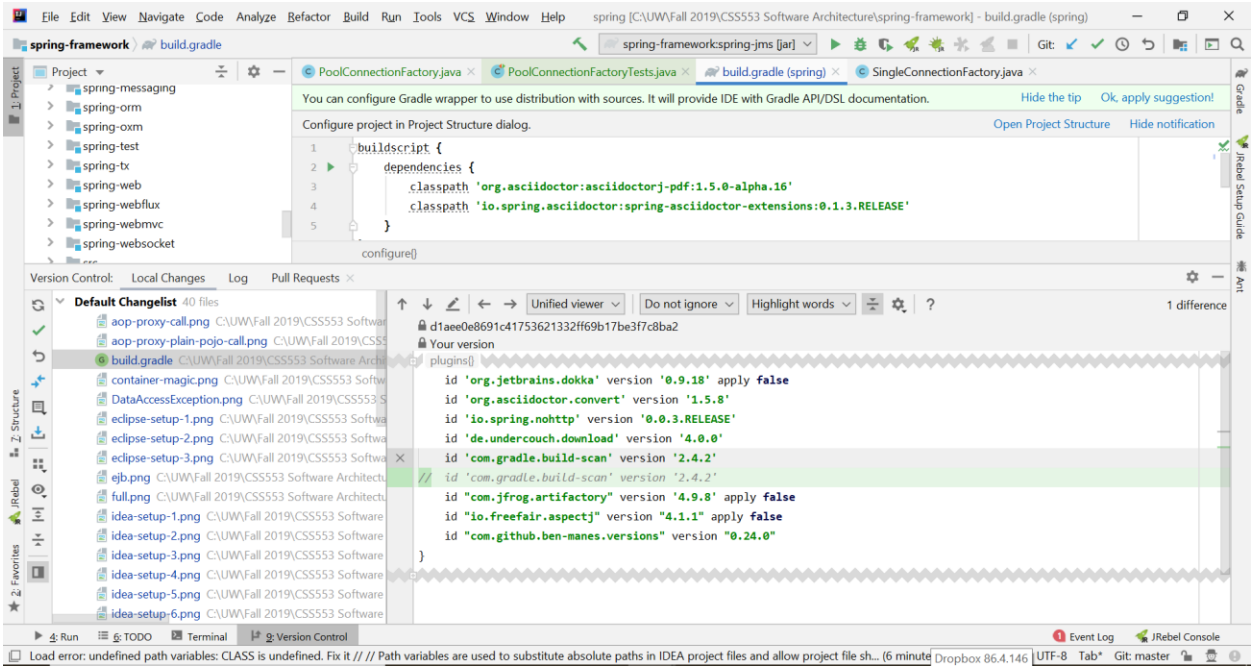
2.2 SOURCE CODE

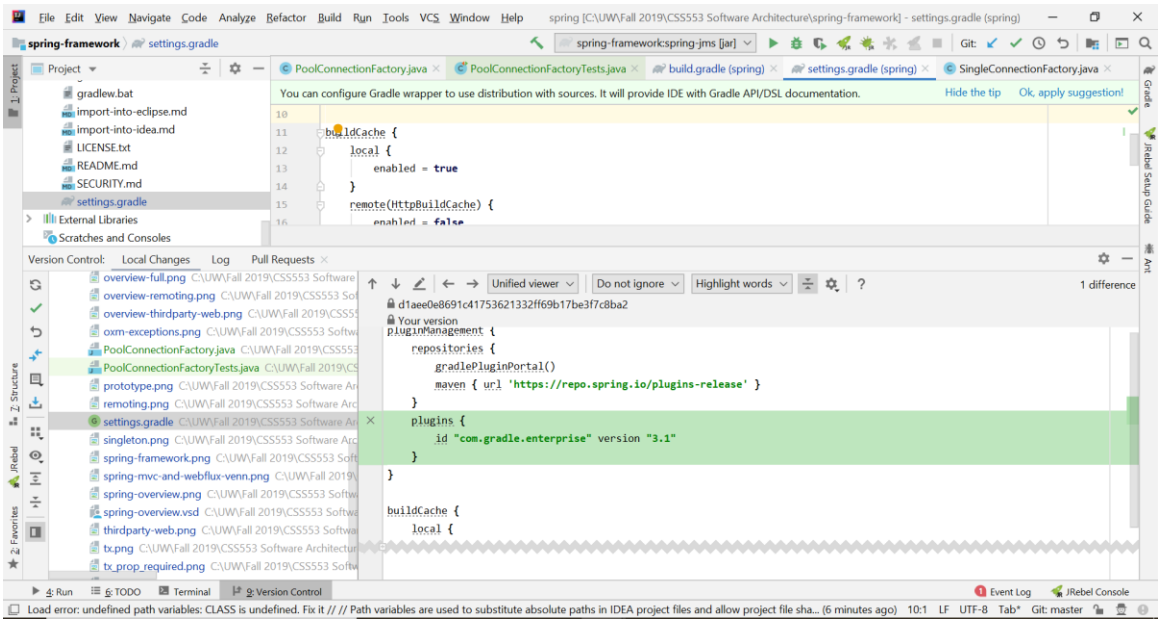
https://github.com/spring-projects/spring-framework

3 SOURCE CODE AND IMPLEMENTATION DETAILS

3.1 PROJECT CONFIGURATION CHANGES

Due to the old version dependency on gradle build-scan plugin, the project configuration files build.gradle and settings.gradle had to be updated to support gradle 6.1+ dependency with the enterprise plugin to be able to build jars. The version control log of the two files showing the changes are as shown below.





3.2 COMPONENTS

The three components we added are the following

1. AnyPoolCachingSessionFactory
2. PoolConnectionFactory
3. DynamicSessionFactory

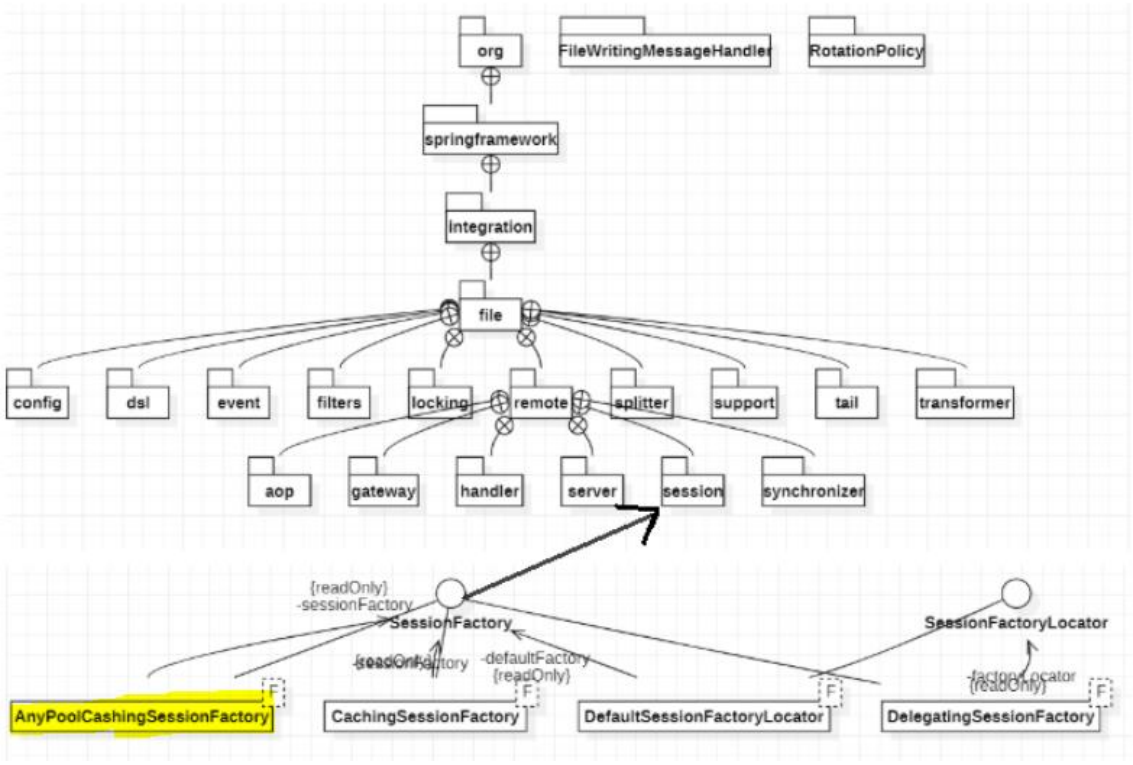
The programming language of our implementation was Java.

3.3 ANYPOOLCACHINGSESSIONFACTORY

AnyPoolCachingSessionFactory is alternative of CachingSessionFactory. Current Implementation of CachingSessionFactory only support one pool which is SimplePool however AnyPoolCachingSessionFactory will supports the multiple pools. CachingSesssionFactory is tightly coupled with SimplePool class while AnyPoolCachingSessionFactory allows extensibility.

3.3.1 Architecture Visualization

The AnyPoolCachingSessionFactory component is added in spring-integration module of spring-framework under the session directory as shown in the architecture visualization below.



3.3.2 Implementation detail

3.3.2.1 Constructors

Constructor
Constructor and Description
AnyPoolCachingSessionFactory(SessionFactory<F> sessionFactory, Pool<Session<F>> pool) Create a AnyPoolCashingSessionFacotry delegating to any pool specified by user

```java
/**
 * A {@link SessionFactory} implementation that caches Sessions for reuse without
 * requiring reconnection each time the Session is retrieved from the factory.
 * This implementation wraps and delegates to a target SessionFactory instance.
 *
 * @author Pinki Patel
 *
 */

public class AnyPoolCachingSessionFactory<F> implements SessionFactory<F>, DisposableBean {
    private static final Log logger = LogFactory.getLog(CachingSessionFactory.class);
    private final SessionFactory<F> sessionFactory;
    private final Pool<Session<F>> pool;
    private final boolean isSharedSessionCapable;
    private volatile long sharedSessionEpoch;

    public AnyPoolCachingSessionFactory(SessionFactory<F> sessionFactory, Pool<Session<F>> pool) {
        Assert.isTrue(!(sessionFactory instanceof DelegatingSessionFactory),
                message: "'sessionFactory' cannot be a 'DelegatingSessionFactory'; cache each delegate instead");
        this.sessionFactory = sessionFactory;
        this.pool = pool;
        this.isSharedSessionCapable = sessionFactory instanceof SharedSessionCapable;
    }
```

3.3.2.2 *Provided Interface*

All Methods	
Modifier and Type	
void	resetCache() Clear the cache of Session; also any in-use session will be closed when returned to the cache.

```java
    // Customize method for AnyPoolCachingSessionFactory class
    public synchronized void resetCache() {
        if (logger.isDebugEnabled()) {
            logger.debug("Cache reset; idle sessions will be removed, in-use sessions will be closed when returned");
        }
        if (this.isSharedSessionCapable && ((SharedSessionCapable) this.sessionFactory).isSharedSession()) {
            ((SharedSessionCapable) this.sessionFactory).resetSharedSession();
        }
        long epoch = System.nanoTime();

        while (epoch == this.sharedSessionEpoch) {
            epoch = System.nanoTime();
        }
        this.sharedSessionEpoch = epoch;
        this.pool.removeAllIdleItems();
    }
```

3.3.2.3 *Required Interface*

Required Interface

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	destroy() Invoked by the containing BeanFactory on destruction of a bean.

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
Session<F>	getSession()

```
@Override
public Session<F> getSession() { return new CachedSession(this.pool.getItem(), this.sharedSessionEpoch); }

@Override
public void destroy() { this.pool.removeAllIdleItems(); }
```

3.3.3 Test Cases

We created JUnit tests as per the Spring-testing framework corresponding to AnyPoolCachingSessionFactory.java class as AnyPoolCachingSessionFactoryTests.java which contained the following five test methods:

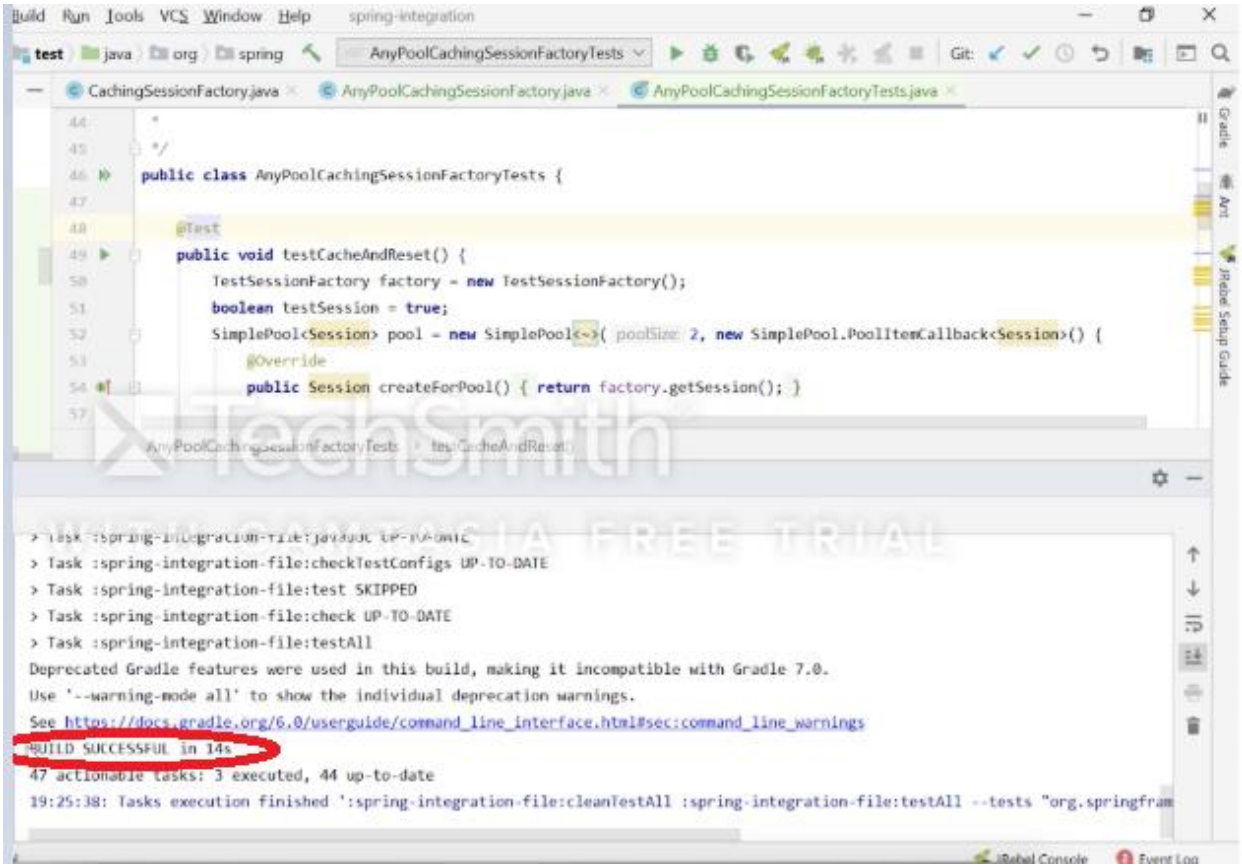
1. testCacheAndReset()
2. testDirtySession()

The tests were ran using Intellji IDE's Junit and Gradle plugin and all two test cases were passed. In the initial attempts the test cases failed at various errors like import order, reg header and some formatting errors since code was not comments as per required by Spring developer. However they were fixed and code was refied such that all cases for AnyPoolCachingSessionFactory passed.

3.3.4 Operational Component Test Results

We have used JUnit test cases as per the Spring-Test framework to test the AnyPoolCachingSessionFactory class. It check that functionality as well as logic of class is correct or not. It also checks for amount of comments in the class. Test class for AnyPoolCachingSessionFactory is annotated by @Test.

The gradle junit test plugin in Intellij IDE were made use of to enable the test so that no additional application server deployment is required to run the test.



Gradle Evaluation Report

The gradle evaluation report of the 2 functions created to check various scenarios

---

AnyPoolCachingSessionFactoryTests

all > org.springframework.integration.file.remote.session > AnyPoolCachingSessionFactoryTests

2	0	0	7.916s
tests	failures	ignored	duration

100% successful

Tests

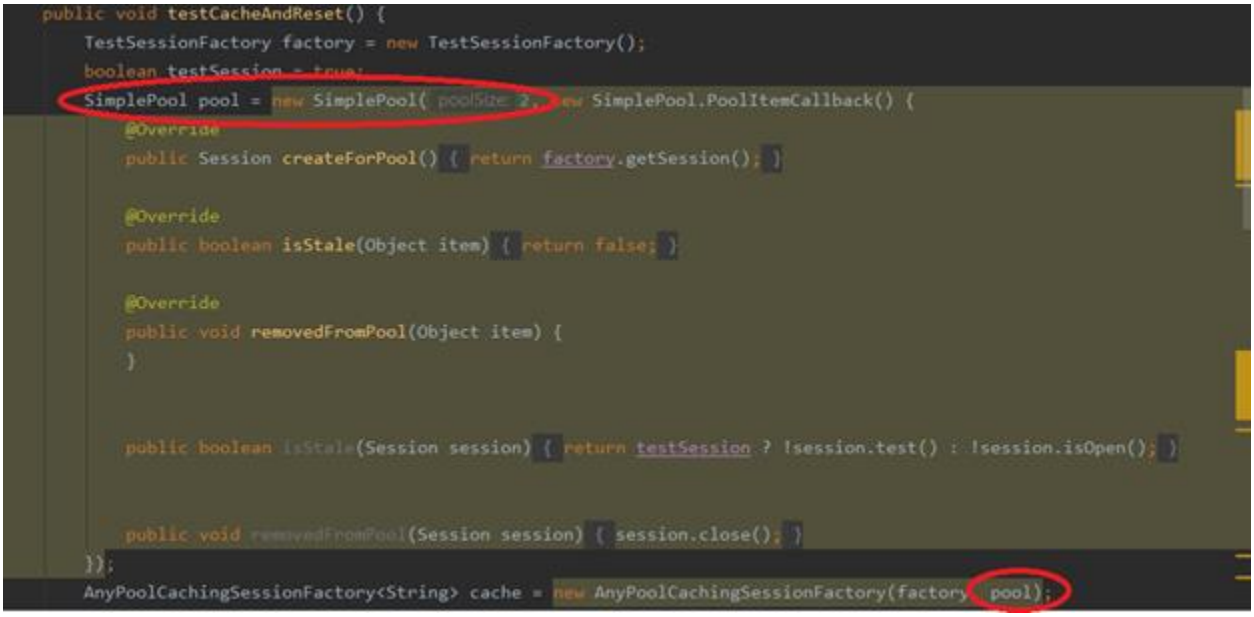
Test	Duration	Result
testCacheAndReset	4.784s	passed
testDirtySession	3.132s	passed

Generated by Gradle 6.0 at 13 Dec, 2019 8:59:00 PM

3.3.5 Component Usage Details

A user might want to use AnyPoolCachingSessionFactory when it implemented session that it wants to reuse without requiring reconnection each time the session is retrieved from the factory. AnyPoolCachingSessionFactory wraps session factory in an instance of AnyPoolCachingSessionFactory where it provides additional properties such as resetCache(). When invoke this method, all idle sessions are immediately closed and in-use sessions are closed when they are returned to the cache.

In order to provide a way to configure other session caching, CachingSessionFactory class constructor used to take size of pool and session factory object. However, our newly built class, AnyPoolCachingSessionFactory, leave the setting up pool size and adjusting the pool size to client. This decoupled the AnyPoolCachingSessionFactory from Pool that it will used to cache the session Factory. As shown in below test case a SimplePool object was created with pool size of 2 which was then passed into AnyPoolCachingSessionFactory object.



3.3.6 Consistency with Existing Code Organization

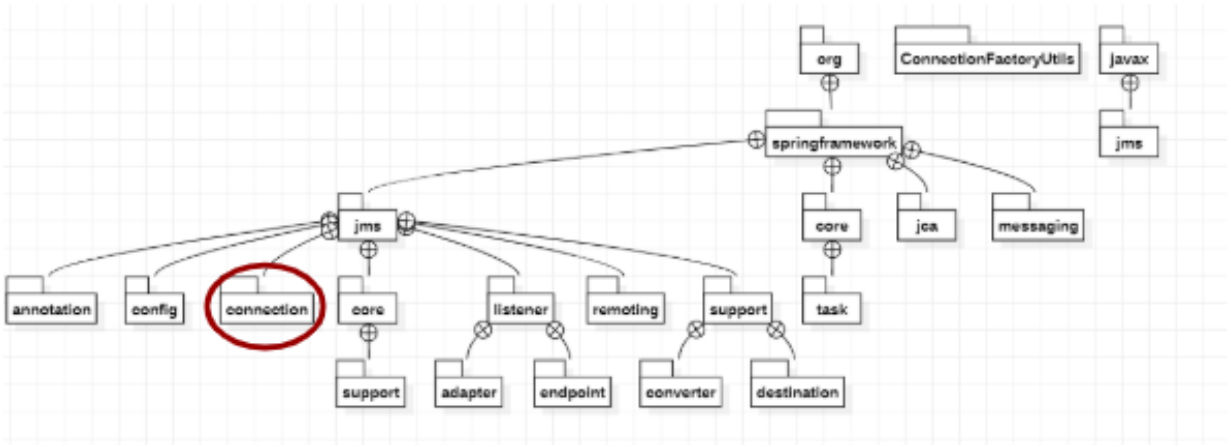
CachingSessionFactory is located under org.springframework.integration.file.remote.session and we have added new class, AnyPoolCachingSessionFactory, in the same folder. As it can be seen in the above diagram. And adding AnyPoolCachingSessionFactory in same folder as CachingSessionFactory does conform to the stated design.

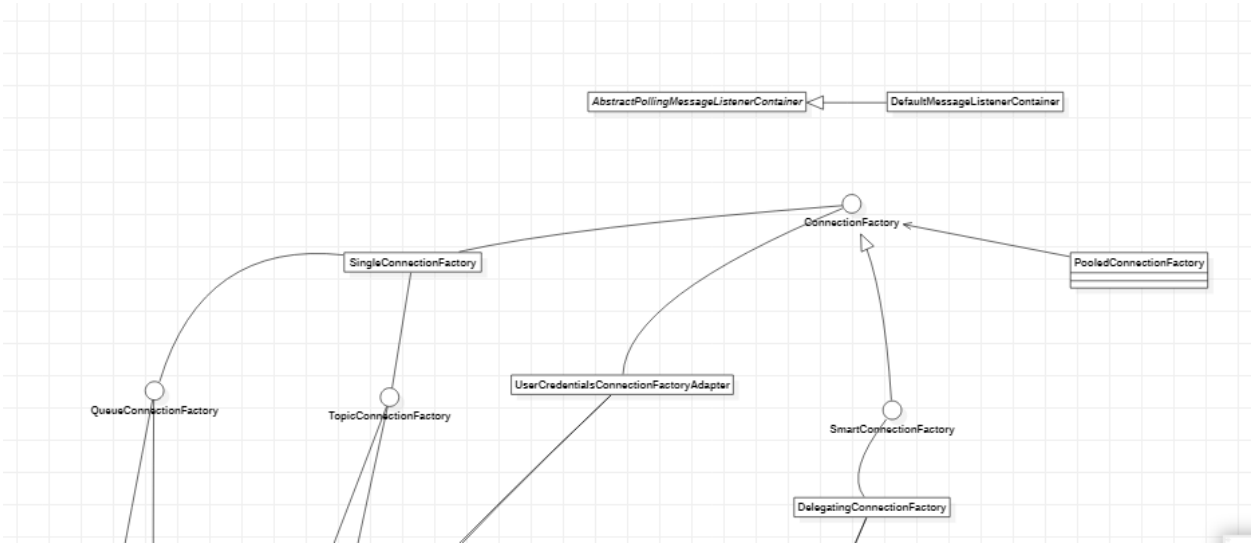
3.4 POOLCONNECTIONFACTORY

PoolConnectionFactory is an alternative implementation of ConnectionFactory interface instead of SingleConnectionFactory and CachingConnectionFactory. SingleConnectionFactory and CachingConnectionFactory are able to maintain only 1 TCP connection under the hood. The class that we have created, PoolConnectionFactory, can pool multiple TCP connection and multiple session/consumer/producer under that. And this will provide better throughput and high load system. SingleConnectionFactory class and CachingConnectionFactory class is located under org.springframework.jms.connection and new class, PoolConnectionFactory, have been added under the same folder as it is visible from the diagram below.

3.4.1 Architecture Visualization

The PoolConnectionFactory component is added in spring-jms module of spring-framework under the connection directory as shown in the architecture visualization below.





3.4.2 Implementation Details

3.4.2.1 Constructors

Consutructors
Constructor and Description
PooledConnectionFactory()
PooledConnectionFactory(List<Connection> connectionPool)

```
/**
 * Create a new PooledConnectionFactory for bean-style usage.
 * @see #setTargetConnectionFactory
 */
public PoolConnectionFactory() {
}

/**
 * Create a new PooledConnectionFactory that always returns the given ConnectionPool.
 * @param targetConnections is the Connection Pool
 */
public PoolConnectionFactory(List<Connection> targetConnections) {
    Assert.notNull(targetConnection, "Target Connection must not be null");
    this.connectionPool = targetConnections;
}
```

3.4.2.2 Required Interface

Required Interface

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
Connection	createConnection() Creates a connection with the default user identity.
Connection	createConnection(String userName, String password) Creates a connection with the specified user identity.
JMSContext	createContext() Creates a JMSContext with the default user identity and an unspecified sessionMode.
JMSContext	createContext(int sessionMode) Creates a JMSContext with the default user identity and the specified session mode.
JMSContext	createContext(String userName, String password) Creates a JMSContext with the specified user identity and an unspecified sessionMode.
JMSContext	createContext(String userName, String password, int sessionMode) Creates a JMSContext with the specified user identity and the specified session mode.

```
    @Override
    public Connection createConnection() throws JMSException {
        return getSharedConnectionProxy(getConnection());
    }

    @Override
    public Connection createConnection(String username, String password) throws JMSException {
        throw new javax.jms.IllegalStateException(
                "SingleConnectionFactory does not support custom username and password");
    }

    @Override
    public JMSContext createContext() { return obtainTargetConnectionFactory().createContext(); }

    @Override
    public JMSContext createContext(String userName, String password) {
        return obtainTargetConnectionFactory().createContext(userName, password);
    }

    @Override
    public JMSContext createContext(String userName, String password, int sessionMode) {
        return obtainTargetConnectionFactory().createContext(userName, password, sessionMode);
    }
```

3.4.2.3 *Provided Interface*

Method Summary	
Modifier and Type	***Method and Description***
protected void	buildFromProperties(Properties props)
protectedConnectionPool	createConnectionPool(javax.jms.Connection connection) Delegate that creates each instance of an ConnectionPool object.
Properties	getProperties() Get the properties from this instance for storing in JNDI
Reference	getReference()
protected javax.jms.Connection	newPooledConnection(ConnectionPool connection)
protected void	populateProperties(Properties props) Called by any superclass that implements a JNDIReferencable or similar that needs to collect the properties of this class for storage etc.
void	setProperties(Properties properties) set the properties for this instance as retrieved from JNDI

```
protected Connection newPooledConnection(ConnectionPool connection) {
    return new PooledConnection(connection);
}

/**
 * Delegate that creates each instance of an ConnectionPool object.  Subclasses can override
 * this method to customize the type of connection pool returned.
 */
protected ConnectionPool createConnectionPool(Connection connection) {
    return new ConnectionPool(connection);
}

/**
 * Called by any superclass that implements a JNDIReferencable or similar that needs to collect
 * the properties of this class for storage etc.
 */
protected void populateProperties(Properties props) {
    props.setProperty("maximumActiveSessionPerConnection", Integer.toString(getMaximumActiveSessionPerConnectio
    props.setProperty("maxConnections", Integer.toString(getMaxConnections()));
    props.setProperty("idleTimeout", Integer.toString(getIdleTimeout()));
    props.setProperty("expiryTimeout", Long.toString(getExpiryTimeout()));
    props.setProperty("timeBetweenExpirationCheckMillis", Long.toString(getTimeBetweenExpirationCheckMillis()))
    props.setProperty("createConnectionOnStartup", Boolean.toString(isCreateConnectionOnStartup()));
    props.setProperty("useAnonymousProducers", Boolean.toString(isUseAnonymousProducers()));
    props.setProperty("blockIfSessionPoolIsFullTimeout", Long.toString(getBlockIfSessionPoolIsFullTimeout()));
    props.setProperty("reconnectOnException", Boolean.toString(isReconnectOnException()));
}
```

3.4.3 Test Cases

We created JUnit tests as per the Spring-testing framework corresponding to PoolConnectionFactory.class as PoolConnectionFactoryTests.java which contained the following five test methods:

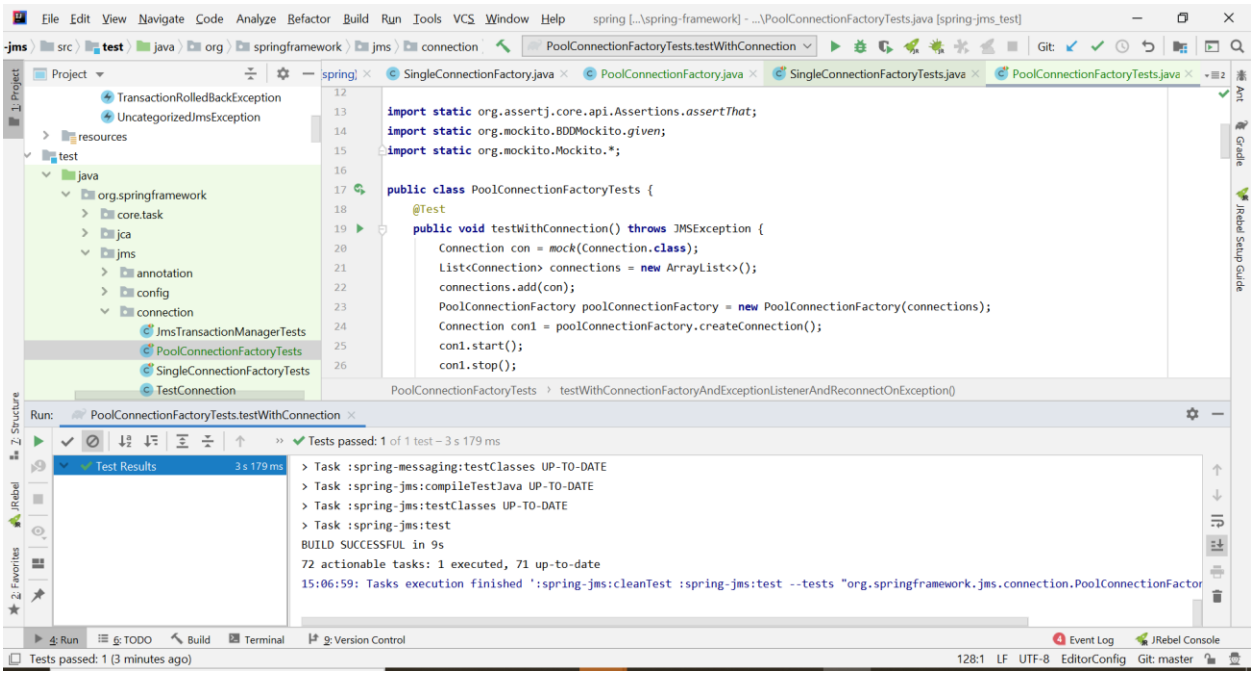
1. testWithConnection()
2. testWithConnectionFactoryAndClientId()
3. testWithConnectionFactoryAndExceptionListener()
4. testWithConnectionFactoryAndExceptionListenerAndReconnectOnException()
5. testWithConnectionFactoryAndReconnectOnException()

The tests were ran using Intellij IDE's JUnit and Gradle plugin and all the 5 test cases were passed. In the initial attempts the test cases failed at various errors like NullPointException and ArrayOutOfBoundException which were occuring at different corner cases which were fixed and code was refined such that all the 5 cases passed.

3.4.4 Operational Component Test Results

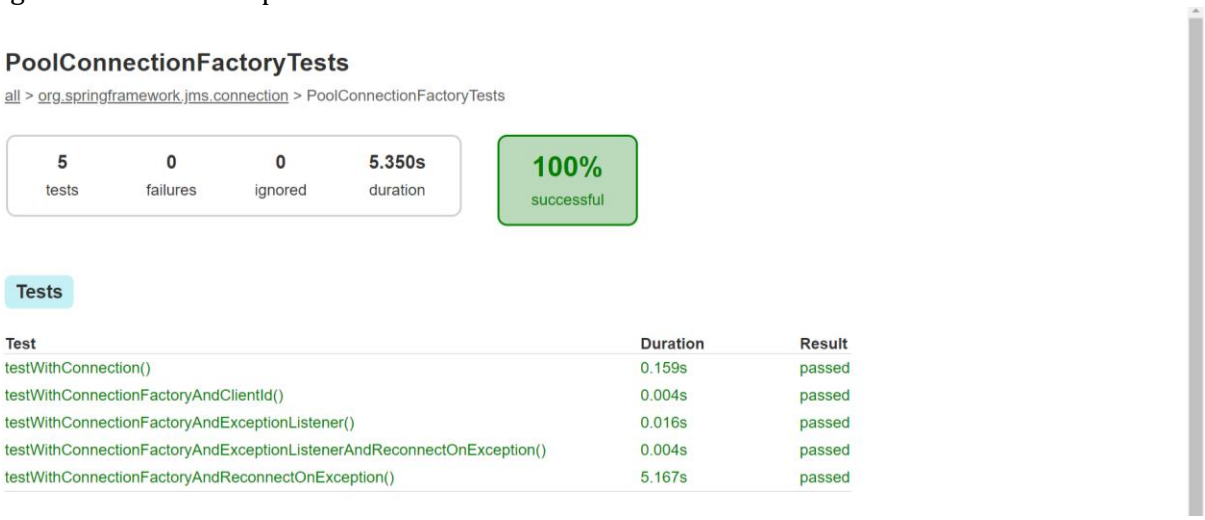
We used JUnit test cases as per the Spring-Test framework to test the components such that not only is the functionality and logic correctness checked but also the inherent rules of coding an open source project related to formatting and documentation are validated.

The gradle junit test plugin in Intellij IDE were made use of to enable the test so that no additional application server deployment is required to run the tests.



3.4.4.1 Gradle Evaluation Report

The gradle evaluation report of the 5 functions created to check various scenarios are as follows



PoolConnectionFactoryTests

all > org.springframework.jms.connection > PoolConnectionFactoryTests

5	0	0	5.350s
tests	failures	ignored	duration

100% successful

Tests

Test	Duration	Result
testWithConnection()	0.159s	passed
testWithConnectionFactoryAndClientId()	0.004s	passed
testWithConnectionFactoryAndExceptionListener()	0.016s	passed
testWithConnectionFactoryAndExceptionListenerAndReconnectOnException()	0.004s	passed
testWithConnectionFactoryAndReconnectOnException()	5.167s	passed

3.4.5 Consistency with Existing Code Organization

The code component and corresponding JUnit test class were added at the same level as the other implementations of ConnectionFactory interface (SingleConnectionFactory, CachingConnectionFactory etc). The package names and documentations were amended to be consistent with the coding convention of the project. The JUnit test case parsed the newly created classes through the checkStyle.xml created that checks for documentation, indentation and formatting of code as well as import statements.

3.4.6 Component Usage Details

The PoolConnectionFactory can be used as shown below:

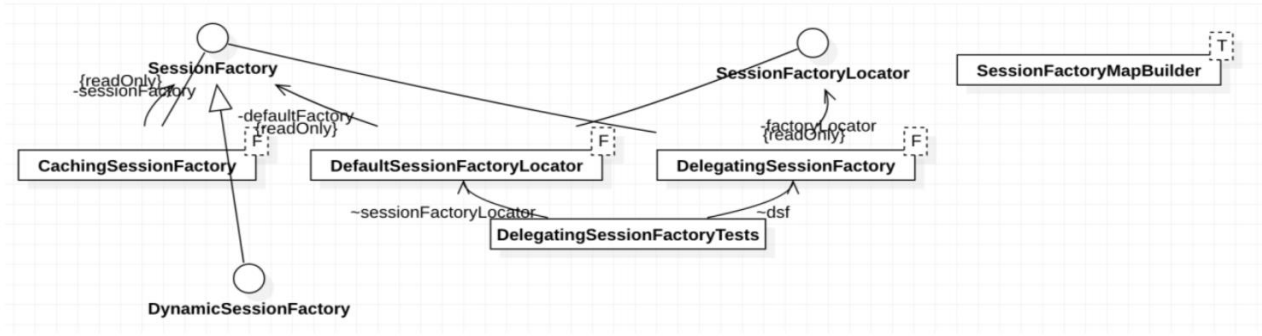
```
public void testWithConnectionFactoryAndReconnectOnException() throws JMSException {
    ConnectionFactory cf = mock(ConnectionFactory.class);
    TestConnection con = new TestConnection();
    given(cf.createConnection()).willReturn(con);

    PoolConnectionFactory poolConnectionFactory = new PoolConnectionFactory(cf);
    poolConnectionFactory.setReconnectOnException(true);
    Connection con1 = poolConnectionFactory.createConnection();
```

3.5 DYNAMICSESSIONFACTORY

DynamicSessionFactory can be used to add multiple SessionFactory objects at runtime. DynamicSessionFactory has an object of DynamicSessionFactoryLocator which contains map to keep track of SessionFactory objects, to add and remove them at run time. Earlier it was planned to be implemented as an interface but it is now implemented as a class to use map to add and remove objects at runtime.

3.5.1 Architecture Visualization

The DynamicSessionFactory implements the SessionFactory interface to get the session of the object.



3.5.2 Implementation Details

3.5.2.1 Constructors

Earlier it was planned to be written as an interface but now it is written as a class to use map to add and remove SessionFactory Objects.

Interface
Interface and Description
Interface DynamicSessionFactory<F>
Factory for acquiring and deleting multiple Session instances on runtime

```
 */
public class DynamicSessionFactory<F> implements SessionFactory<F> {

    private final DynamicSessionFactoryLocator<F> factoryLocator;

    private final ThreadLocal<Object> threadKey = new ThreadLocal<~>();


    public DynamicSessionFactory(DynamicSessionFactoryLocator<F> factoryLocator) {
        Assert.notNull(factoryLocator,  message: "'factoryFactory' cannot be null");
        this.factoryLocator = factoryLocator;
    }
```

3.5.2.2 *Provided Interface*

All Methods	
Modifier and Type	**Method and Description**
Void	addSession(Session<F>) Add a new session to the pool
Void	closeSession(Session<F>) Close a session from the pool
Session<F>	getSession() Get a session from the pool

```
     * @param factory the factory.
     */
    public void addSessionFactory(String key, SessionFactory<F> factory) { this.factories.put(key, factory); }

    /**
     * Remove a session factory.
     * @param key the lookup key.
     * @return the factory, if it was present.
     */
    public SessionFactory<F> removeSessionFactory(Object key) { return this.factories.remove(key); }

    @Override
    public SessionFactory<F> getSessionFactory(Object key) {
        if (key == null) {
            return this.defaultFactory;
        }
        SessionFactory<F> factory = this.factories.get(key);
        return factory != null ? factory : this.defaultFactory;
    }
```

3.5.2.3 *Required Interface*

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
Session<F>	getSession()

```
 */
public class DynamicSessionFactory<F> implements SessionFactory<F> {

    private final DynamicSessionFactoryLocator<F> factoryLocator;

    private final ThreadLocal<Object> threadKey = new ThreadLocal<~>();
```

```
  @Override
  public Session<F> getSession() { return getSession(this.threadKey.get()); }

  public Session<F> getSession(Object key) {
      SessionFactory<F> sessionFactory = this.factoryLocator.getSessionFactory(key);
      Assert.notNull(sessionFactory, message: "No default SessionFactory configured");
      return sessionFactory.getSession();
  }
```

3.5.3 Test Cases

We created JUnit tests as per the Spring-testing framework corresponding to DynamicSessionFactory.class as DynamicSessionFactoryTests.java which contained the following two test methods:

1. public void testDelegates()
2. public void testFlow()

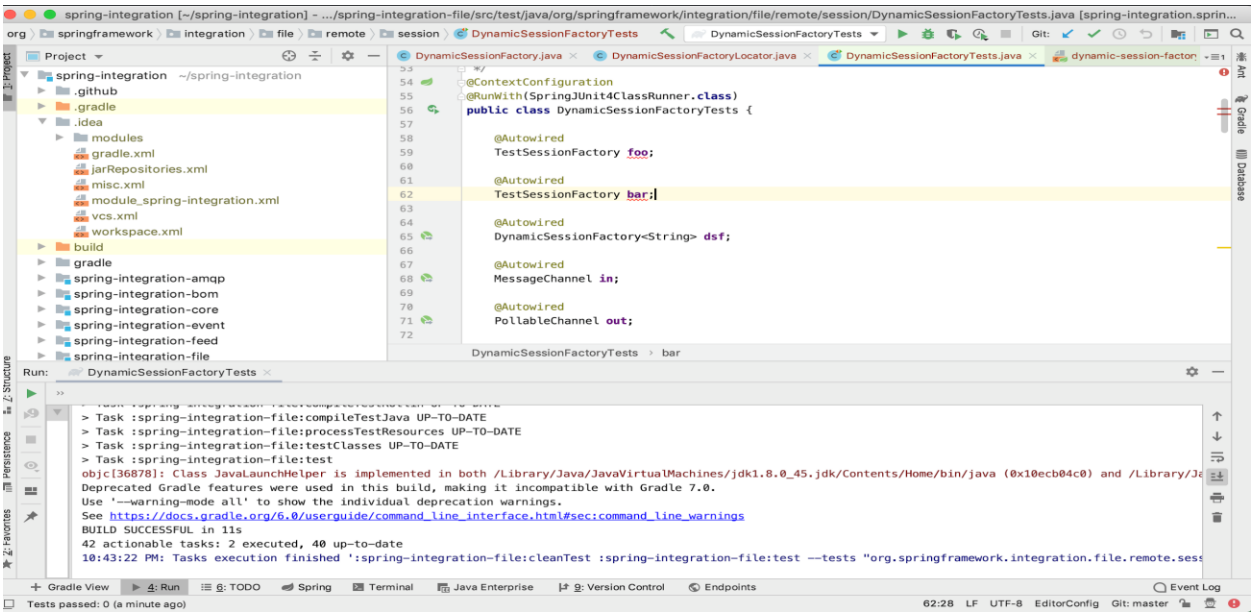
The testDelegates method used two preconfigured sessions to test and created a third session at runtime and was added to the DynamicSessionFactoryLocator at runtime using addSessionFactory method and removed using removeSessionFactory method. The testFlow was used to check the individual session execution. The configurations for the sessions can be added using xml file.

The tests were executed using Intellij IDE's JUnit and Gradle plugin and all the 2 test cases were passed. In the initial attempts the test cases failed at errors like NullPointerException and NoSuchBeanDefinitionException which were experienced because the configuration file was not added in the test class and was fixed after adding the configuration file and test cases were passed.

3.5.4 Operational Component Test Results

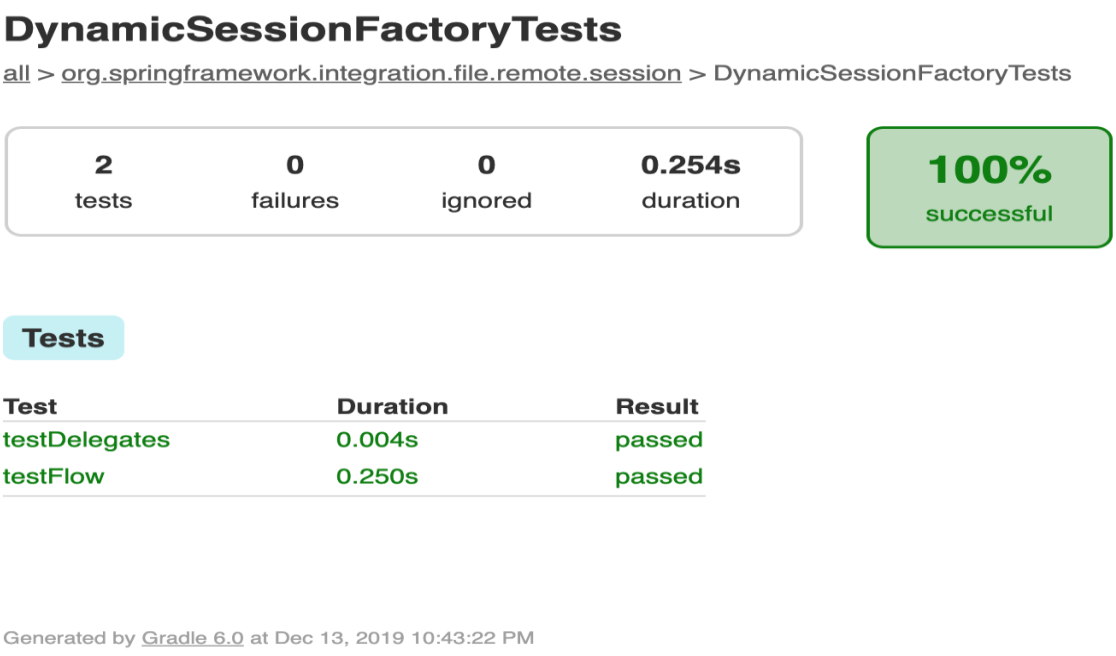
We used JUnit test cases as per the Spring-Test framework to test the components such that not only is the functionality and logic correctness checked but also the inherent rules of coding an open source project related to formatting and documentation are validated.

The gradle junit test plugin in Intellij IDE were made use of to enable the test so that no additional application server deployment is required to run the tests.



3.5.4.1 Gradle Evaluation Report

The gradle evaluation report of the 2 functions created to check various scenarios are as follows



3.5.5 Consistency with Existing Code Organization

The code component and corresponding JUnit test class were added at the same level as the other implementations of SessionFactory interface. The package names and documentations were amended to be consistent with the coding convention of the project. The JUnit test case parsed the newly created classes through the checkStyle.xml created that checks for documentation, indentation and formatting of code as well as import statements.

3.5.6 Component Usage Details

The DynamicConnectionFactory can be used as shown below:

```
@Bean
DynamicSessionFactory<String> dsf() {
    DynamicSessionFactoryLocator<String> sff = sessionFactoryLocator();
    return new DynamicSessionFactory<>(sff);
}

@Bean
public DynamicSessionFactoryLocator<String> sessionFactoryLocator() {
    Map<Object, SessionFactory<String>> factories = new HashMap<>();
    factories.put("foo", foo());
    TestSessionFactory bar = bar();
    factories.put("bar", bar);
    return new DynamicSessionFactoryLocator<>(factories, bar);
}
```

4 Operational Status of Components

Three classes were added/implemented into Spring Integration open source project, AnyPoolCachingSesssionFactory, PoolConnectionFactory and DynamicSessionFactory. All these three classes are working.

Unit tests for all three new classes were written to test all the methods written in the class. All test cases were passed in the unit tests. Below is the screenshots of tests written for each new class that we are planning to add into Spring Integration and Spring JMS.

5 Functional Requirements

The motivation of the three components we created were three issues raised by various users on the spring-framework git issues forum which had a relatively large number of upvotes. The functional requirements requested by the users were the following:

- Program should be able to cache any pool session. This is implemented in AnyPoolCachingSessionFactory class.

- Program should be capable of adding/removing multiple SFTP connections at runtime. This is implemented in DynamicSessionFactory class.
- Program should be able to pool multiple TCP connections and sessions/consumers/producers such that high load systems using the framework have better throughPut. This is implemented in PoolConnectionFactory.

6 NON-FUNCTIONAL REQUIREMENTS

- **Extensibility/Flexibility:** This NFR was implemented in DynamicSessionFactory and AnyPoolCachingSessionFactory. DynamicSessionFactory is able to extend to multiple SFTP connections unlike SessionFactory which would handle only one connection. AnyPoolCachingSessionFactory will any pool not just with Simple pool as it does with CachingSessionFactory because pool will be constructed by client and passed into the constructor of the AnyPoolCachingSessionFactory.
- **Reusability:** This NFR was implemented in PoolConnectionFactory since same class can be used to multiple SFTP connections.
- **Maintainability:** This NFR was implemented in DynamicSessionFactory, AnyPoolCachingSessionFactory, and PoolConnectorFactory. We have implemented new classes instead of changing the existing class in order to meet the backward compatibility. All new classes are loosely coupled and they don't degrade the architecture of the Spring.

7 MAINTENANCE STRATEGY

All our new classes are decoupled therefore modifying one class will not adverse effects on other classes. Also we are not modifying current implemented class which are SessionFactory, SingleConnectionFactory and CachingSessionFactory but adding new one so if there are any project on client which uses these class will not be impacted. Also we have properly document the code so it is easier for developers to understand the written code.

8 USER GUIDE

8.1 SYSTEM REQUIREMENTS

For development and enhancement on the module:

1. Java IDE - Works best with Intellji.
2. Clone Spring Integration and Spring Framework from GitHub (add our new classes).
3. Gradle enterprise plugin required for one of the project dependencies (build-scan). This might take some time to get since you have to request them from gradle enterprise.

For using in an application:

1. Use the jars for spring-integration and spring-jms as a dependency on your project with the following class files added
    a. PoolConnectionFactory.class
    b. AnyPoolCachingSessionFactory.class
    c. DynamicSessionFactory.class
2. Java 1.8+
3. Application server (example: Apache Tomcat v7+)

9 VIDEO TUTORIAL

Three video (.mp4) files are attached with the report

1. Component1_poolConnectionFactory.mp4 (https://drive.google.com/file/d/1M8kloJKrVSnkiRepQzf1DZBJxQQdeffQ/view?usp=sharing)
2. Component2_anypoolCachingSessionFactory.mp4 (https://drive.google.com/file/d/1hGmA2w-MVNWFPpBvGboBaIHpZGx693eq/view?usp=sharing)
3. Component3_dynamicSessionFactory.mp4 (https://drive.google.com/file/d/1HQYkd7XghdzZ4mm24NT6NahxqYpY86_O/view?usp=sharing)

Each video file contains the description of the respective components source code as well as a working demo of the Component in use along with Gradle-JUnit evaluation report for the test cases created for each.

We used camatasia software to create the videos.

10 CODE DELIVERABLES

Due to the large size of the two modules spring-integration and spring-jms, we are uploading only the java class files we have created as part of the project. The actual source code of the individual projects can be downloaded from the official github repository of spring-frameowork.

The path of each component within the projects are as follows:

- java/org/springframework/integration/file/remote/session/AnyPoolCachingSessionFactory.java
- java/org/springframework/integration/file/remote/session/DynamicSessionFactory.java
- java/org/springframework/integration/file/remote/session/DynamicSessionFactoryLocator.java

- java/org/springframework/jms/connection/PoolConnectionFactory.java

Tests

- java/org/springframework/jms/connection/PoolConnectionFactoryTests.java
- java/org/springframework/integration/file/remote/session/AnyPoolCachingSessionFactoryTests.java
- java/org/springframework/integration/file/remote/session/DynamicSessionFactoryTests.java

11 Reflection

11.1 Learning Curve

- We understood the architecture and design of Spring Framework which is an application framework and inversion of control container for the Java platform.
- We also familiarised reverse engineering tools like Star UML.
- Got the confidence to contribute to a popular open source project

11.2 Approach in Future Software Design Projects

- For extending an existing architecture, learn intended architecture before implementation
- Evaluate architecture degradation
- For a design from scratch, finalise principal design decisions and document them with reasons

11.3 Time Investment in Project

The time spent on G4 and G5 is 40 hours.

12 References

1. https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html
2. https://github.com/spring-projects/spring-framework/issues/18163
3. https://objectpartners.com/2017/12/20/dynamic-sftp-connection-factory-for-spring-integration/
4. https://github.com/spring-projects/spring-integration/issues/2771
5. https://github.com/spring-projects/spring-integration/issues/2692