SPRING FRAMEWORK

SECURITY PATTERNS AND IMPLEMENTATION PLAN

SUBMITTED BY

MONALI CHANDURKAR

PINKI PATEL

JEFFY J POOZHITHARA

1 CONTENTS

1 INTRODUCTION

The Spring Framework is an application framework and inversion of control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE (Enterprise Edition) platform.

The project was created with the following goals:

1. To provide convenience to developers to develop enterprise-class applications using POJO(Plain Old Java Object).
2. It is created in a modular fashion so that the developer can use the required classes ignore the rest.
3. To make testing easy by adding environment-dependent code into this framework.
4. To provide API to translate technology-specific exceptions into consistent, unchecked exceptions.

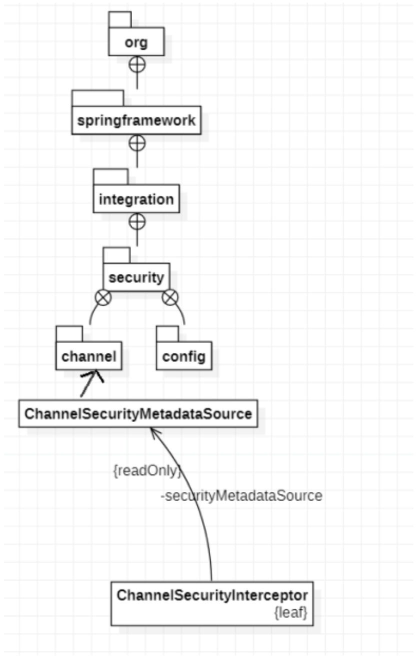
1.1 SPONSORING ORGANIZATIONS

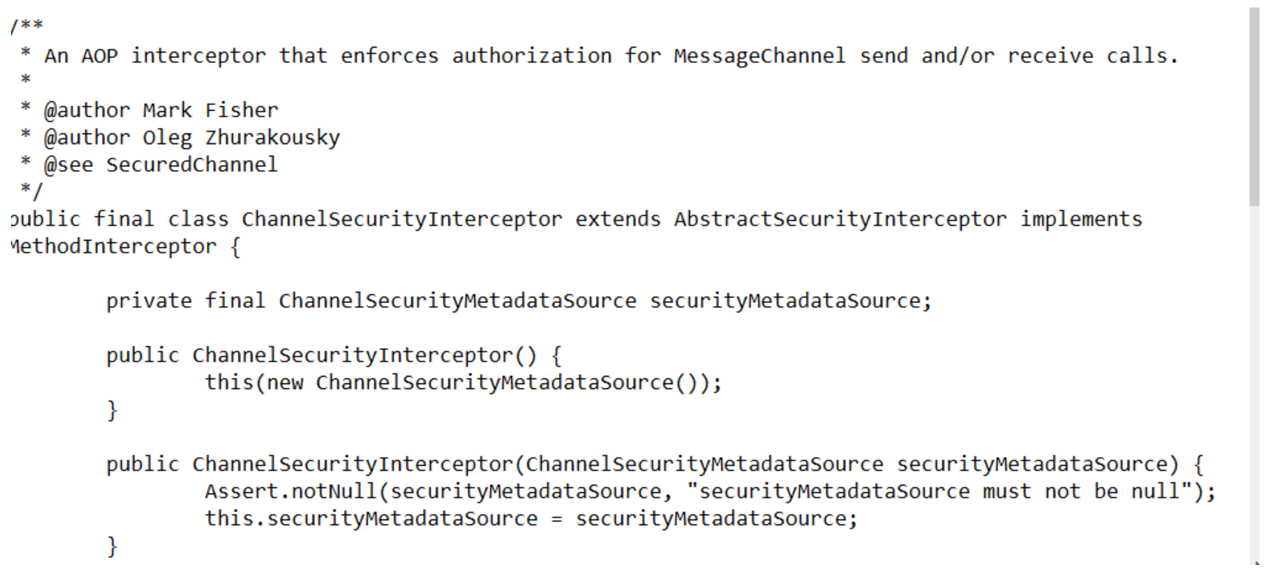
Pivotal Software, Inc is the sponsoring organization, but it also invites broad participation from the Java community.

2 ARCHITECTURE DIAGRAM WITH SECURITY PATTERN

2.1 SECURE INTERCEPTOR

ChannelSecurityInterceptor which will intercept all send and receive calls on a channel and decide if that call can be executed or denied. And it uses intercept design pattern to do so.

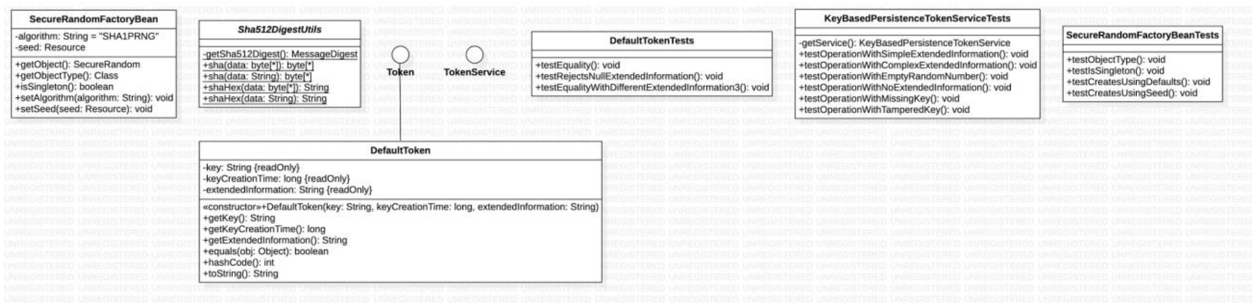


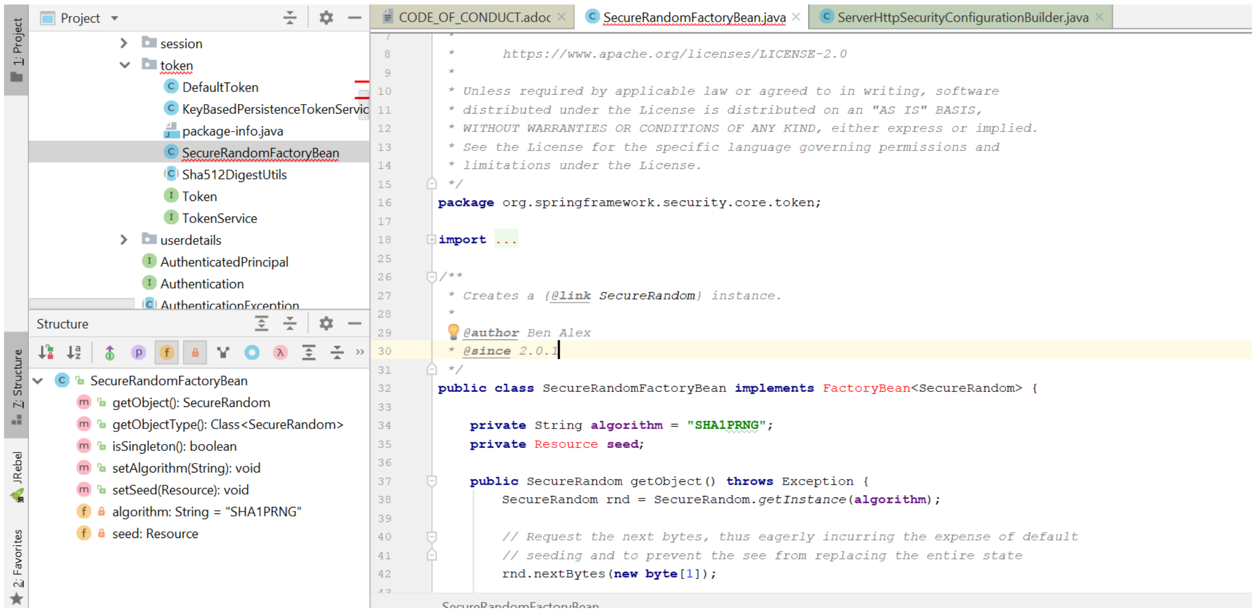


```
/**
 * An AOP interceptor that enforces authorization for MessageChannel send and/or receive calls.
 *
 * @author Mark Fisher
 * @author Oleg Zhurakousky
 * @see SecuredChannel
 */
public final class ChannelSecurityInterceptor extends AbstractSecurityInterceptor implements
MethodInterceptor {

        private final ChannelSecurityMetadataSource securityMetadataSource;

        public ChannelSecurityInterceptor() {
                this(new ChannelSecurityMetadataSource());
        }

        public ChannelSecurityInterceptor(ChannelSecurityMetadataSource securityMetadataSource) {
                Assert.notNull(securityMetadataSource, "securityMetadataSource must not be null");
                this.securityMetadataSource = securityMetadataSource;
        }
```

2.2 SECURE FACTORY

The intent of the Secure Factory design pattern is to separate the security dependent logic involved in creating or selecting an object from the basic functionality of the created or selected object. Class SecureRandomFactoryBean is responsible for providing the security in spring framework.

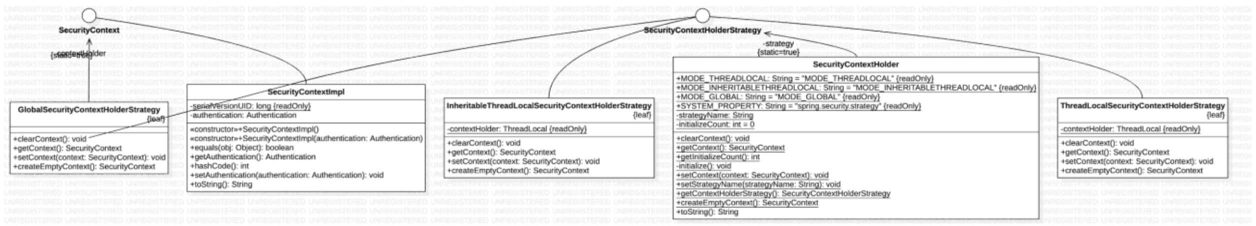

SecureRandomFactoryBean
-algorithm: String = "SHA1PRNG"
-seed: Resource
+getObject(): SecureRandom
+getObjectType(): Class
+isSingleton(): boolean
+setAlgorithm(algorithm: String): void
+setSeed(seed: Resource): void
Sha512DigestUtils
-getSha512Digest(): MessageDigest
+sha(data: byte[*]): byte[*]
+sha(data: String): byte[*]
+shaHex(data: byte[*]): String
+shaHex(data: String): String
Token
TokenService
DefaultTokenTests
+testEquality(): void
+testRejectsNullExtendedInformation(): void
+testEqualityWithDifferentExtendedInformation3(): void
KeyBasedPersistenceTokenServiceTests
-getService(): KeyBasedPersistenceTokenService
+testOperationWithSimpleExtendedInformation(): void
+testOperationWithComplexExtendedInformation(): void
+testOperationWithEmptyRandomNumber(): void
+testOperationWithNoExtendedInformation(): void
+testOperationWithMissingKey(): void
+testOperationWithTamperedKey(): void
SecureRandomFactoryBeanTests
+testObjectType(): void
+testIsSingleton(): void
+testCreatesUsingDefaults(): void
+testCreatesUsingSeed(): void
DefaultToken
-key: String {readOnly}
-keyCreationTime: long {readOnly}
-extendedInformation: String {readOnly}
«constructor»+DefaultToken(key: String, keyCreationTime: long, extendedInformation: String)
+getKey(): String
+getKeyCreationTime(): long
+getExtendedInformation(): String
+equals(obj: Object): boolean
+hashCode(): int
+toString(): String

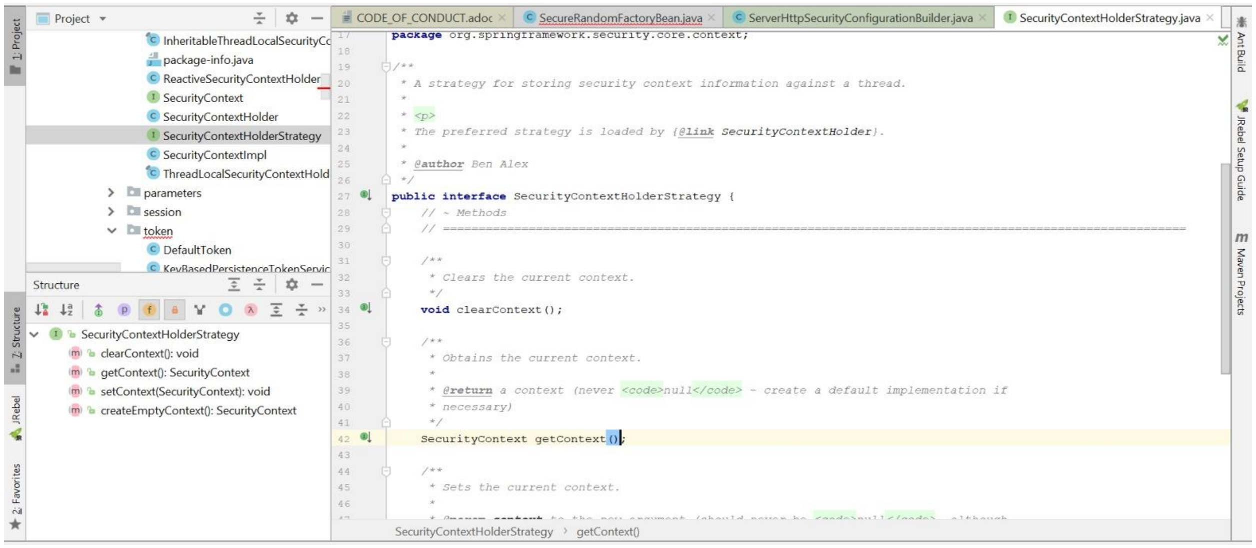


```
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.springframework.security.core.token;

import ...

/**
 * Creates a {@link SecureRandom} instance.
 *
 * @author Ben Alex
 * @since 2.0.1
 */
public class SecureRandomFactoryBean implements FactoryBean<SecureRandom> {

    private String algorithm = "SHA1PRNG";
    private Resource seed;

    public SecureRandom getObject() throws Exception {
        SecureRandom rnd = SecureRandom.getInstance(algorithm);

        // Request the next bytes, thus eagerly incurring the expense of default
        // seeding and to prevent the see from replacing the entire state
        rnd.nextBytes(new byte[1]);
```

2.3 SECURE STRATEGY

The intent of the Secure Strategy pattern is to provide an easy to use and modify method for selecting the appropriate strategy object for per- forming a task based on the security credentials of a user or environment. Class SecurityContextHolderStrategy and GlobalSecurityContextHolderStrategy are responsible for secure strategy design pattern in spring framework.


SecurityContext
-contextHolder
{static=true}
GlobalSecurityContextHolderStrategy
{leaf}
+clearContext(): void
+getContext(): SecurityContext
+setContext(context: SecurityContext): void
+createEmptyContext(): SecurityContext
SecurityContextImpl
-serialVersionUID: long {readOnly}
-authentication: Authentication
«constructor»+SecurityContextImpl()
«constructor»+SecurityContextImpl(authentication: Authentication)
+equals(obj: Object): boolean
+getAuthentication(): Authentication
+hashCode(): int
+setAuthentication(authentication: Authentication): void
+toString(): String
InheritableThreadLocalSecurityContextHolderStrategy
{leaf}
-contextHolder: ThreadLocal {readOnly}
+clearContext(): void
+getContext(): SecurityContext
+setContext(context: SecurityContext): void
+createEmptyContext(): SecurityContext
SecurityContextHolderStrategy
-strategy
{static=true}
SecurityContextHolder
+MODE_THREADLOCAL: String = "MODE_THREADLOCAL" {readOnly}
+MODE_INHERITABLETHREADLOCAL: String = "MODE_INHERITABLETHREADLOCAL" {readOnly}
+MODE_GLOBAL: String = "MODE_GLOBAL" {readOnly}
+SYSTEM_PROPERTY: String = "spring.security.strategy" {readOnly}
-strategyName: String
-initializeCount: int = 0
+clearContext(): void
+getContext(): SecurityContext
+getInitializeCount(): int
-initialize(): void
+setContext(context: SecurityContext): void
+setStrategyName(strategyName: String): void
+getContextHolderStrategy(): SecurityContextHolderStrategy
+createEmptyContext(): SecurityContext
+toString(): String
ThreadLocalSecurityContextHolderStrategy
{leaf}
-contextHolder: ThreadLocal {readOnly}
+clearContext(): void
+getContext(): SecurityContext
+setContext(context: SecurityContext): void
+createEmptyContext(): SecurityContext


Project
InheritableThreadLocalSecurityCc
package-info.java
ReactiveSecurityContextHolder
SecurityContext
SecurityContextHolder
SecurityContextHolderStrategy
SecurityContextImpl
ThreadLocalSecurityContextHold
parameters
session
token
DefaultToken
KeyBasedPersistenceTokenServic
Structure
SecurityContextHolderStrategy
clearContext(): void
getContext(): SecurityContext
setContext(SecurityContext): void
createEmptyContext(): SecurityContext
CODE_OF_CONDUCT.adoc
SecureRandomFactoryBean.java
ServerHttpSecurityConfigurationBuilder.java
SecurityContextHolderStrategy.java
Ant Build
JRebel Setup Guide
Maven Projects
1: Project
7: Structure
JRebel
2: Favorites
SecurityContextHolderStrategy › getContext()

```java
package org.springframework.security.core.context;

/**
 * A strategy for storing security context information against a thread.
 *
 * <p>
 * The preferred strategy is loaded by {@link SecurityContextHolder}.
 *
 * @author Ben Alex
 */
public interface SecurityContextHolderStrategy {
    // ~ Methods
    // ========================================================================================================

    /**
     * Clears the current context.
     */
    void clearContext();

    /**
     * Obtains the current context.
     *
     * @return a context (never <code>null</code> - create a default implementation if
     * necessary)
     */
    SecurityContext getContext();

    /**
     * Sets the current context.
     *
```

3 VULNERABILITIES IDENTIFIED IN THE PROJECT

The following are the vulnerabilities found in Spring Framework

- **XSS**

    1. [**CVE-2014-1904**] Cross-site scripting (XSS) vulnerability in web/servlet/tags/form/FormTag.java in Spring MVC in Spring Framework 3.0.0 before 3.2.8 and 4.0.0 before 4.0.2 allows remote attackers to inject arbitrary web script or HTML via the requested URI in a default action.

- **DoS CSRF**

    1. [**CVE-2014-0054**] The Jaxb2RootElementHttpMessageConverter in Spring MVC in Spring Framework before 3.2.8 and 4.0.0 before 4.0.2 does not disable external entity resolution, which allows remote attackers to read arbitrary files, cause a denial of service, and conduct CSRF attacks via crafted XML, aka an XML External Entity (XXE) issue. This vulnerability exists because of an incomplete fix for CVE-2013-4152, CVE-2013-7315, and CVE-2013-6429.(These are mentioned in next points)

    2. [**CVE-2013-7315**] The Spring MVC in Spring Framework before 3.2.4 and 4.0.0.M1 through 4.0.0.M2 does not disable external entity resolution for the StAX XMLInputFactory, which allows context-dependent attackers to read arbitrary files, cause a denial of service, and conduct CSRF attacks via crafted XML with JAXB, aka an XML External Entity (XXE) issue, and a different vulnerability than CVE-2013-4152. This issue was SPLIT from CVE-2013-4152 due to different affected versions.

    3. [**CVE-2013-4152**] The Spring OXM wrapper in Spring Framework before 3.2.4 and 4.0.0.M1, when using the JAXB marshaller, does not disable entity resolution, which allows context-dependent attackers to read arbitrary files, cause a denial of service, and conduct CSRF attacks via an XML external entity declaration in conjunction with an entity reference in a (1)

DOMSource, (2) StAXSource, (3) SAXSource, or (4) StreamSource, aka an XML External Entity (XXE) issue.

- **+Info**

  1. [**CVE-2011-2730**] VMware SpringSource Spring Framework before 2.5.6.SEC03, 2.5.7.SR023, and 3.x before 3.0.6, when a container supports Expression Language (EL), evaluates EL expressions in tags twice, which allows remote attackers to obtain sensitive information via a (1) name attribute in a (a) spring:hasBindErrors tag; (2) path attribute in a (b) spring:bind or (c) spring:nestedpath tag; (3) arguments, (4) code, (5) text, (6) var, (7) scope, or (8) message attribute in a (d) spring:message or (e) spring:theme tag; or (9) var, (10) scope, or (11) value attribute in a (f) spring:transform tag, aka "Expression Language Injection."

- **Exec Code**

  1. [**CVE-2010-1622**] SpringSource Spring Framework 2.5.x before 2.5.6.SEC02, 2.5.7 before 2.5.7.SR01, and 3.0.x before 3.0.3 allows remote attackers to execute arbitrary code via an HTTP request containing class.classLoader.URLs[0]=jar: followed by a URL of a crafted .jar file.

4 SUGGESTED SECURITY PATTERNS TO TACKLE VULNERABILITIES

- For XSS[**CVE-2014-1904**] in Spring, **Distrustful Decomposition** which is the Architectural-Level pattern can mitigate the vulnerability by isolating security vulnerabilities to a small subset of a system such that compromising a single component of the system does not lead to the entire system being compromised. The attacker will only have the functionality and data of the single compromised component at their disposal for malicious activity, not the functionality and data of the entire application.

- For DoS CSRF[**CVE-2014-0054**], the **Secure Visitor** pattern can be used which allows nodes to lock themselves against being read by a visitor unless the visi-tor supplies the proper credentials to unlock the node. The Secure Visitor is defined so that the only way to access a locked node is with a visitor, helping to prevent unauthorized access to nodes in the data structure.

- For DoS CSRF[**CVE-2013-7315**], the **Secure Chain of Responsibility** pattern can be applied which decouples the logic that determines user/environment-trust dependent functionality from the portion of the application requesting the functionality, simplifying the logic that determines user/environment-trust dependent functionality, and making it relatively easy to dynamically change the user/environment-trust dependent functionality.

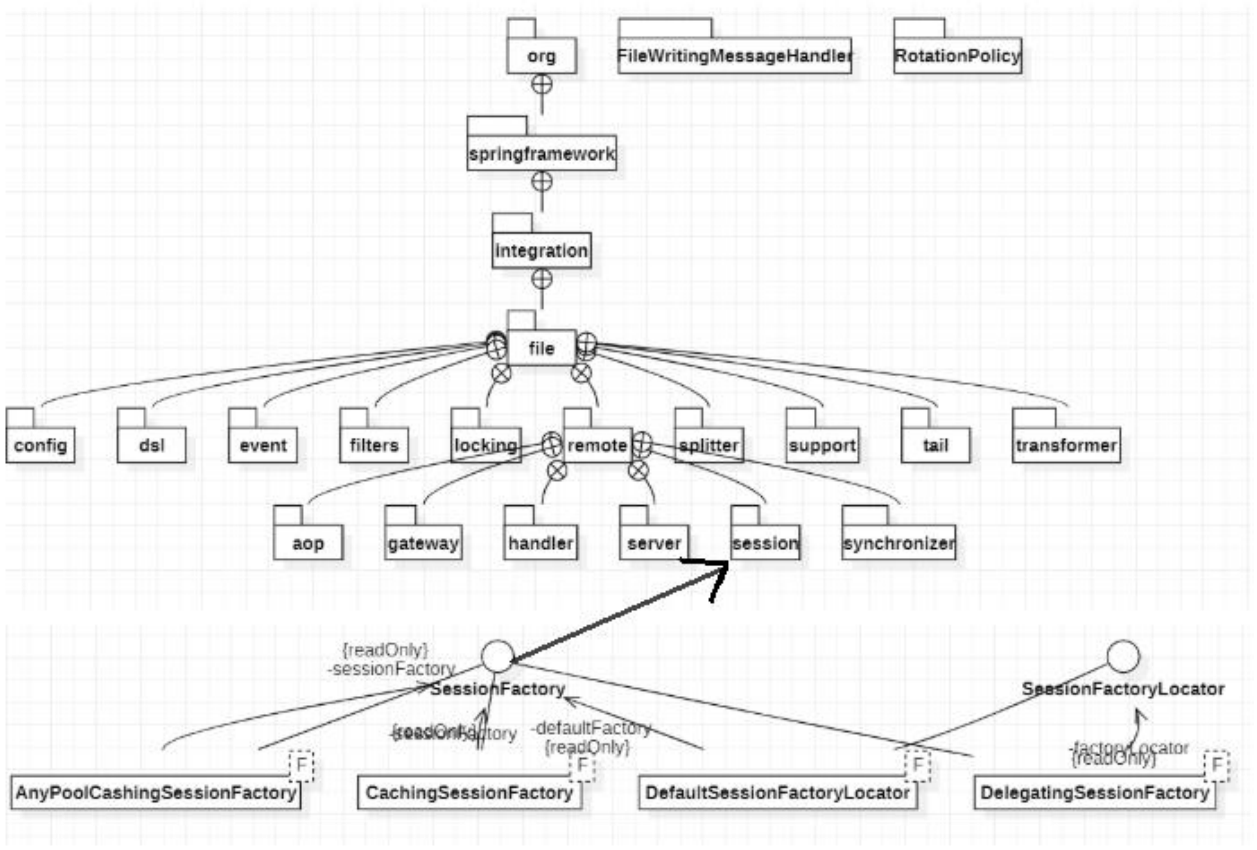
5 IMPLEMENTATION PLAN

5.1 ANYPOOLCASHINGSESSIONFACTORY CLASS

Current implementation of CashingSessionFactory (CSF) is tightly coupled to SimplePool class. Problem resides in two methods that CSF exposes which are coupled to SimplePool, namely public void setPoolSize(int poolSize) and public void setSessionWaitTimeout(long sessionWaitTimeout). Also, CSF uses SImplePool and it is hardcoded which means that CSF can only use SimplePool and will not be able to support different types of pool.

Therefore, we are planning to implement AnyPoolCashingSessionFactory class which will provide loose coupling of components between SessionFactory and Pool. And it can also extended to other pool implementation.

5.1.1 Redrawing architecture diagram with AnyPoolCashingSessionFactory class



5.1.2 Explanation for how the change does not degrade existing architecture

Current Implementation of CSF violates one of the principal design decisions of the Spring Framework, which is to achieve loosely coupled components by implementing modularity. In order to achieve loose coupling, we will implement AnyPoolCashingSessionFactory class, which will only have methods to manage session in the pool and which can extend to any pool implementation unlike current implementation of CSF. Methods such as setPoolSize will not be part of AnyPoolCashingSessionFactory class since they are more pool specific than Session.

AnyPoolCashingSession Factory will provide loose coupling components and extensibility and therefore it will enhancement to current architecture and not degradation.

5.1.3 Provided Interface

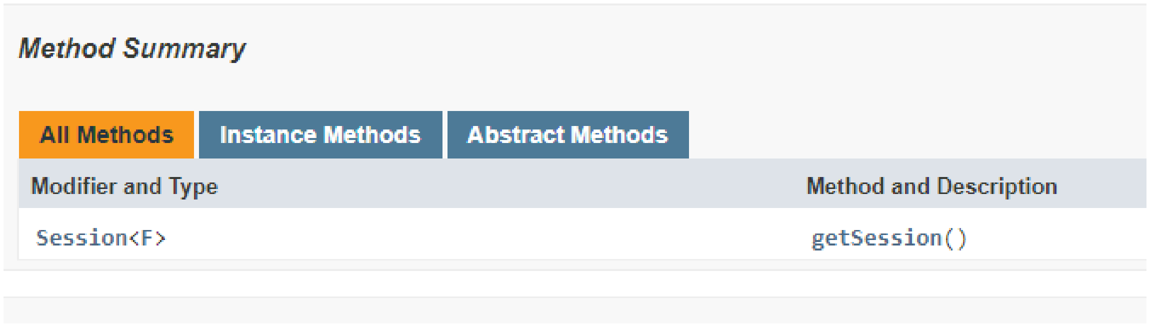
Constructor
Constructor and Description
AnyPoolCachingSessionFactory(SessionFactory<F> sessionFactory, Pool<Session<F>> pool) Create a AnyPoolCashingSessionFacotry delegating to any pool specified by user

All Methods	
Modifier and Type	Method and Description
Void	destroy() Remove (close) any unused session in the pool.
Session<F>	getSession() Get a session from the pool (or block if none available).
void	resetCache() Clear the cache of session; also any in-use session will be closed when returned to the cache.

5.1.4 Required Interface

Interface SessionFactory<F>

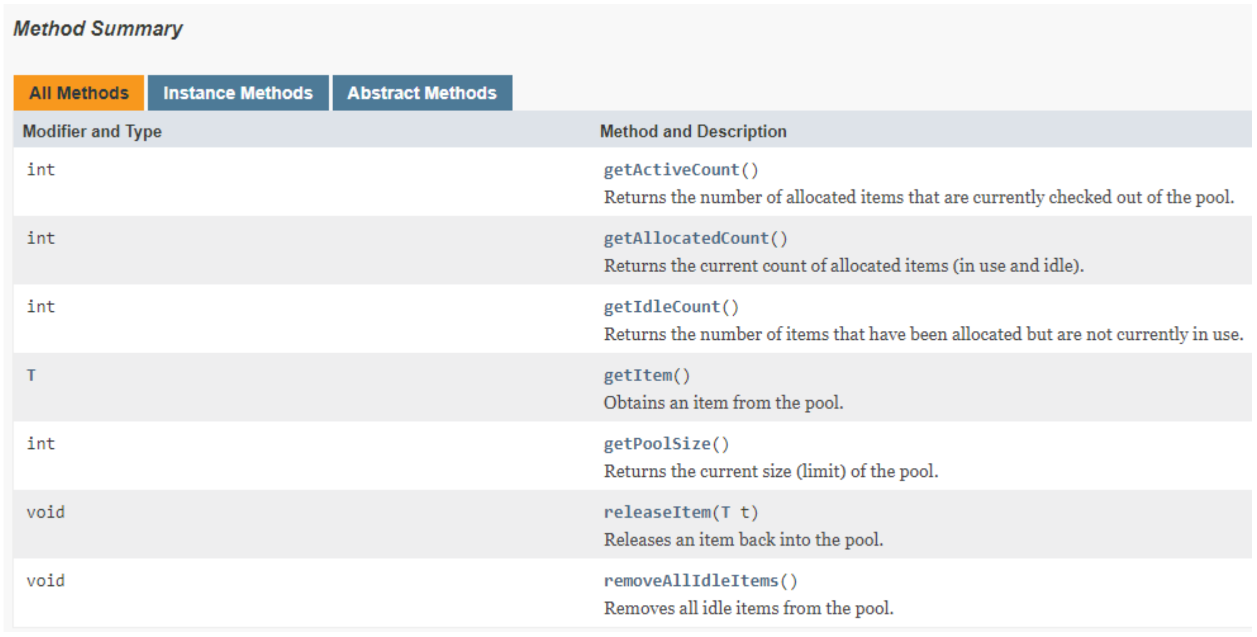
Source: org.springframework.integration.file.remote.session


Method Summary

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

Interface Pool<T>

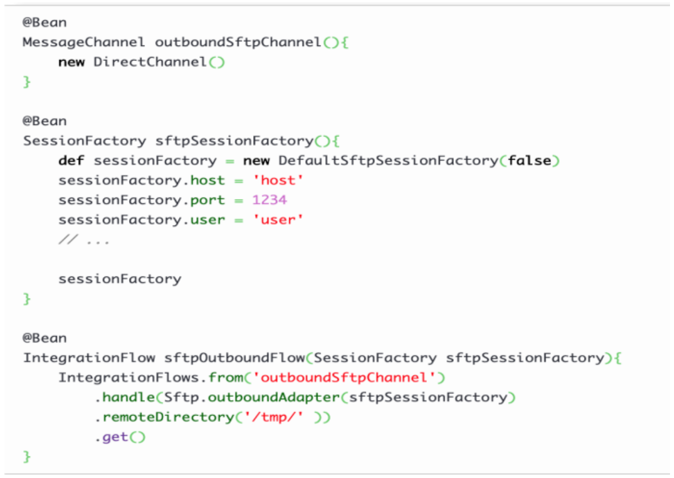
Source: org.springframework.integration.util



Modifier and Type	Method and Description
int	getActiveCount() Returns the number of allocated items that are currently checked out of the pool.
int	getAllocatedCount() Returns the current count of allocated items (in use and idle).
int	getIdleCount() Returns the number of items that have been allocated but are not currently in use.
T	getItem() Obtains an item from the pool.
int	getPoolSize() Returns the current size (limit) of the pool.
void	releaseItem(T t) Releases an item back into the pool.
void	removeAllIdleItems() Removes all idle items from the pool.

5.2 DynamicSessionFactory interface

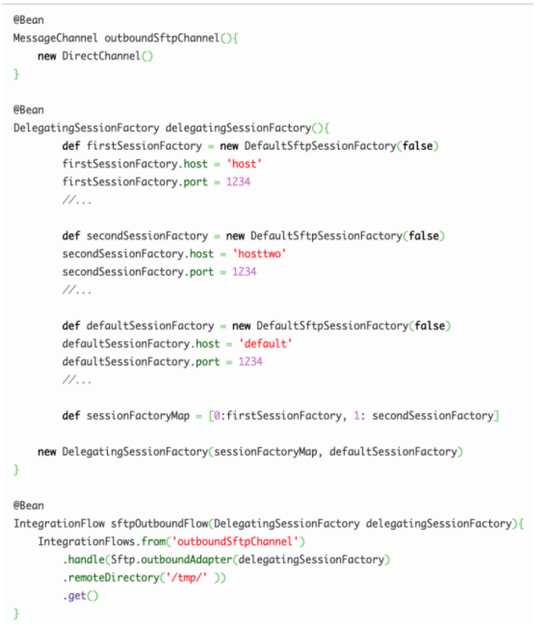
In a simple outbound SFTP integration, configuration for Spring Integration is brief and to the point.The MessageChanel that the SFTP adapter will listen to, the SessionFactory containing the SFTP details, and the IntegrationFlow that defines the path of the message to be processed needs to be defined. Following is an example of a simple Integration flow for a single, static SFTP outbound connection:



```
@Bean
MessageChannel outboundSftpChannel(){
    new DirectChannel()
}

@Bean
SessionFactory sftpSessionFactory(){
    def sessionFactory = new DefaultSftpSessionFactory(false)
    sessionFactory.host = 'host'
    sessionFactory.port = 1234
    sessionFactory.user = 'user'
    // ...

    sessionFactory
}

@Bean
IntegrationFlow sftpOutboundFlow(SessionFactory sftpSessionFactory){
    IntegrationFlows.from('outboundSftpChannel')
        .handle(Sftp.outboundAdapter(sftpSessionFactory)
        .remoteDirectory('/tmp/' ))
        .get()
}
```

This configuration is suitable for a single SFTP connection.DelegatingSessionFactory class can be used if multiple connections are required, each with a unique host/username.

The DelegatingSessionFactory contains a SessionFactoryLocator that finds the correct SessionFactory based on a ThreadKey that is set when the message is being written to the MessageChanel. This means several SFTP connections can be used or read them from configuration, place them in their own session factories and have the proper SessionFactory create an SFTP connection as the message flows through the defined pipeline. This can be done with few modifications to previous configuration the DelegatingSessionFactory can be used.

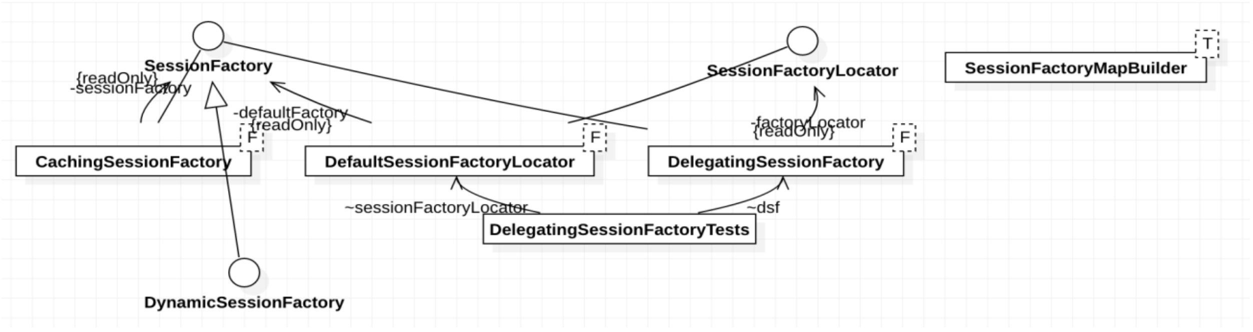


```
@Bean
MessageChannel outboundSftpChannel(){
    new DirectChannel()
}

@Bean
DelegatingSessionFactory delegatingSessionFactory(){
        def firstSessionFactory = new DefaultSftpSessionFactory(false)
        firstSessionFactory.host = 'host'
        firstSessionFactory.port = 1234
        //...

        def secondSessionFactory = new DefaultSftpSessionFactory(false)
        secondSessionFactory.host = 'hosttwo'
        secondSessionFactory.port = 1234
        //...

        def defaultSessionFactory = new DefaultSftpSessionFactory(false)
        defaultSessionFactory.host = 'default'
        defaultSessionFactory.port = 1234
        //...

        def sessionFactoryMap = [0:firstSessionFactory, 1: secondSessionFactory]

    new DelegatingSessionFactory(sessionFactoryMap, defaultSessionFactory)
}

@Bean
IntegrationFlow sftpOutboundFlow(DelegatingSessionFactory delegatingSessionFactory){
    IntegrationFlows.from('outboundSftpChannel')
        .handle(Sftp.outboundAdapter(delegatingSessionFactory)
        .remoteDirectory('/tmp/' ))
        .get()
}
```

With this configuration, 3 different connection factories for my SFTP endpoints are specified. As long as the appropriate Thread key is specified, the proper SFTP connection will be initiated and single SFTP adapter can now handle multiple endpoints. This is appropriate for any process that has a long and stable life, but it does not fit if the process that feeds the outboundSftpChannel is more dynamic. Also a solution is needed for a business use case to be able to add/change/remove SFTP connections at runtime.

The dynamic-ftp sample app is a reasonable solution, but also adding a DynamicSessionFactory interface and implementations that can handle the situation with less complexity for the user.

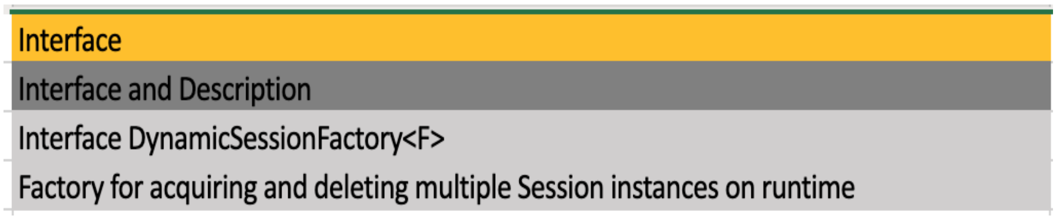
5.2.1 Redrawing architecture diagram with DynamicSessionFactory Interface



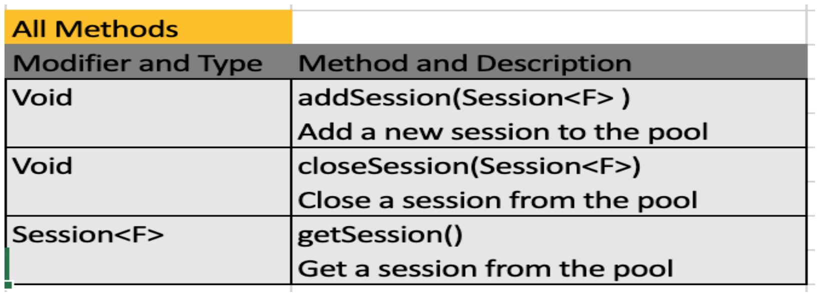
5.2.2 Explanation for how the change does not degrade existing architecture

Addition of DynamicSessionFactory interface and implementations that can handle the situation with less complexity for the user will not degrade the architecture style as this interface will be used only in situations where multiple connections are required, each with a unique host/username. This interface does not connect with any other class or interface except SessionFactory. Hence architecture style will be preserved in this case.

5.2.3 Provided Interface



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

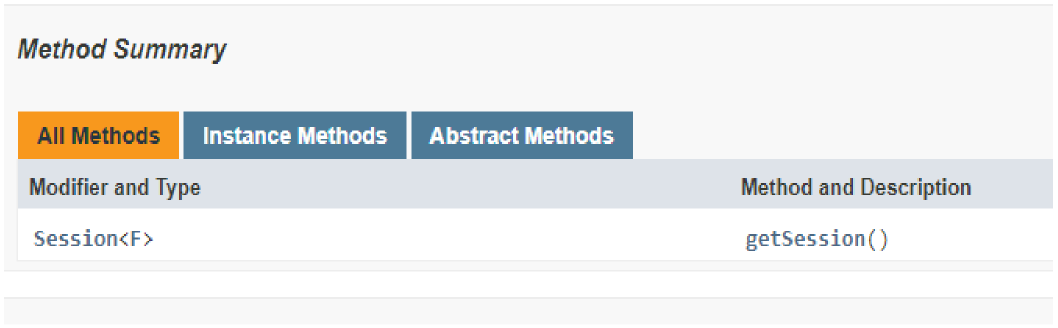


All Methods	
Modifier and Type	Method and Description
Void	addSession(Session<F>) Add a new session to the pool
Void	closeSession(Session<F>) Close a session from the pool
Session<F>	getSession() Get a session from the pool

5.2.4 Required Interface

Interface SessionFactory<F>

Source: org.springframework.integration.file.remote.session



Method Summary

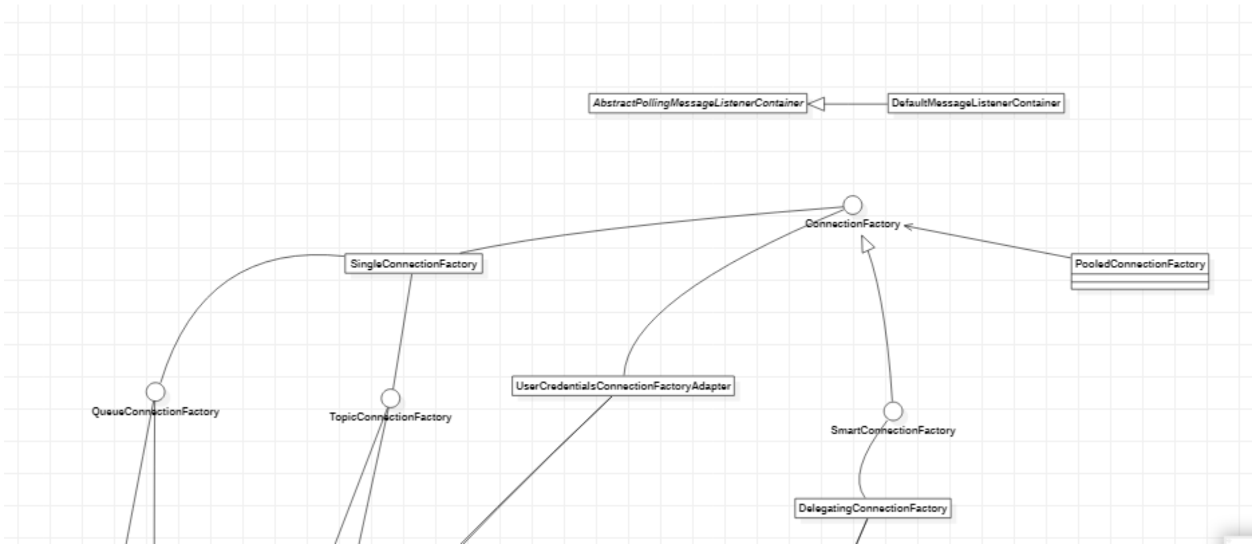
Modifier and Type	Method and Description
Session<F>	getSession()

5.3 POOLEDCONNECTIONFACTORY FOR JMS

Spring provides classes for jms connection optimisation, however both SingleConnectionFactory and CachingConnectionFactory able to maintain only 1 TCP connection under the hood, Spring should have a PooledConnectionFactory which can pool multiple TCP connection and multiple session/consumer/producers under that, which can be better for throughPut and high load system.

In applications that use no third party tool which provides PooledConnectionFactory, all the connections should be managed at the application side or directly use DefaultMessageListenerContainer directly if not using an Application server. The solution is to have PooledConnectionfactory which can have many parallel connections in a controlled manner.

5.3.1 Architecture Diagram After Incorporating the change


AbstractPollingMessageListenerContainer
DefaultMessageListenerContainer
ConnectionFactory
SingleConnectionFactory
PooledConnectionFactory
UserCredentialsConnectionFactoryAdapter
QueueConnectionFactory
TopicConnectionFactory
SmartConnectionFactory
DelegatingConnectionFactory

5.3.2 Explanation for how the change does not degrade existing architecture

For performance many JMS applications that are not being deployed on EE servers choose to have only org.springframework.jms.listener.DefaultMessageListenerContainer which is lightweight. In such scenarios they also expect a PooledConnectionFactory which can give a better throughput. Also, addition of this components resonates with the following principal design decisions

1. Provide choice at every level
2. Separation of Concerns
3. Promote reuse and portability

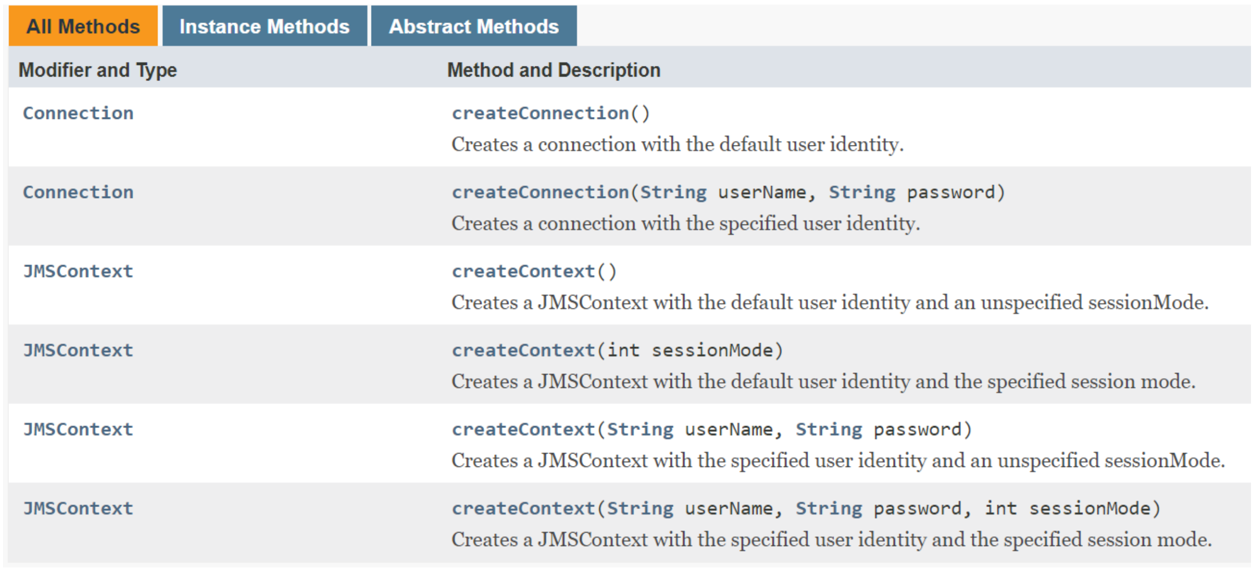
5.3.3 Provided Interface

Consutructors
Constructor and Description
PooledConnectionFactory()
PooledConnectionFactory(String brokerURL)

Method Summary	
Modifier and Type	***Method and Description***
protected Connection	getConnection() Obtain an initialized shared Connection.
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

protected Session	createSession(Connection con, Integer mode) Create a default Session for this ConnectionFactory, adapting to JMS 1.0.2 style queue/topic mode if necessary.
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

5.3.4 Required Interface

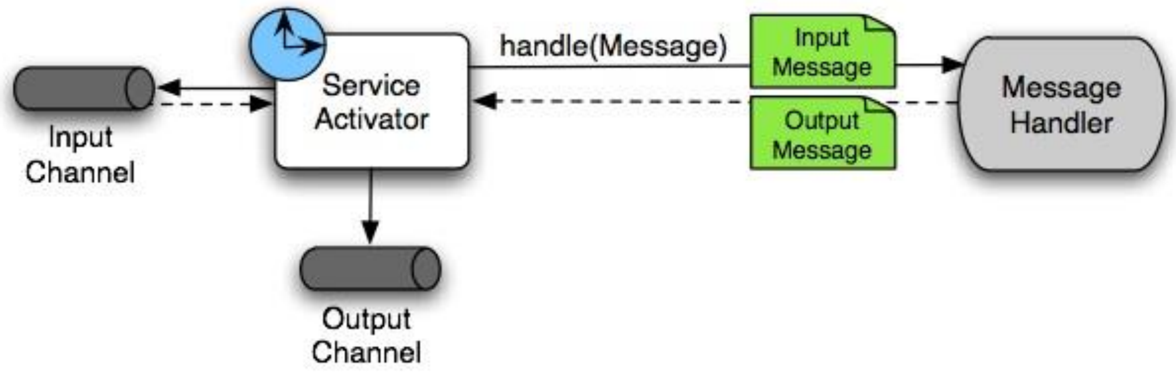


All Methods | Instance Methods | Abstract Methods

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.

6 CONNECTORS

Spring framework generally follows AOP framework, an extension of the OOP style and hence connectors are generally method invocations and procedure calls. A few modules however, follow implicit invocation styles like Pub-Sub. The connectors in such scenarios are Message Handlers and Message Channels that handle the notifications. For instance, the implementation in Spring-integration and Spring-messaging modules are as follows:



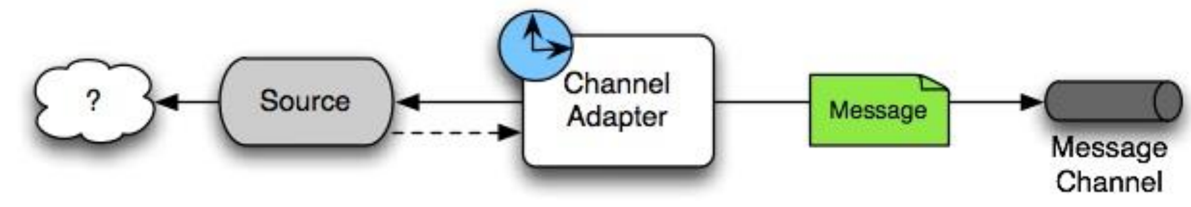


Figure 1: An inbound channel adapter endpoint connects a source system to a MessageChannel

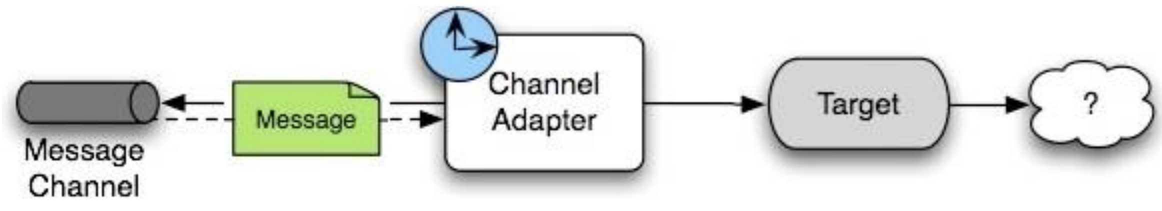


Figure 6. An outbound channel adapter endpoint connects a MessageChannel to a target system

7 REFLECTION

7.1 ARCHITECTURE RECOVERY MECHANISM

7.1.1 UML Diagram Generation

StarUML software was used to parse source code of the project to generate the Package Structure and Type Hierarchy diagrams. Package Structure diagram was used to identify Architecture Styles and Type Hierarchy diagram was used to identify Design Patterns.

7.1.2 Code Search Based on Textual Similarity

Another approach used was to conduct code search based on textual similarity. Keywords like "Builder", "Factory", "Strategy" etc were used to find if there are Classes that has similar names which was then cross checked with the generated UML diagrams.

7.2 HOW WOULD YOU IMPROVE YOUR DESIGN RECOVERY PROCESS IN THE FUTURE

Design Recovery process in the future can be improved by studying the package structure of the project and reading the documentation of each and every component and connector and generating architectural diagrams.

7.3 APPROXIMATELY TIME INVESTMENT IN THE ASSIGNMENT

The time spent on G3 is 40 hours.

8 REFERENCES

8.1 DOCUMENTATION

1. https://spring.io/guides/topicals/spring-security-architecture/
2. https://www.cvedetails.com/vulnerability-list/vendor_id-9664/product_id-17274/Springsource-Spring-Framework.html(Vulnerabilities)

8.2 ISSUES

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