SPRING FRAMEWORK

IDENTIFYING DESIGN AND ARCHITECTURE PATTERNS AND STYLES

SUBMITTED BY

MONALI CHANDURKAR
PINKI PATEL
JEFFY J POOZHITHARA

1 CONTENTS

2 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.

1.1.1 SPONSORING ORGANIZATIONS

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

OTHER INFORMATION:

The first version was written by Rod Johnson, who released the framework with the publication of his book Expert One-on-One J2EE Design and Development in October 2002. The framework was first released under the Apache 2.0 license in June 2003. The releases are in every 2 to 6 years with latest release in 2019.

3 ARCHITECTURE / PRINCIPAL DESIGN DECISIONS

The principal design decisions (PDD) found in the project's documentation are as follows

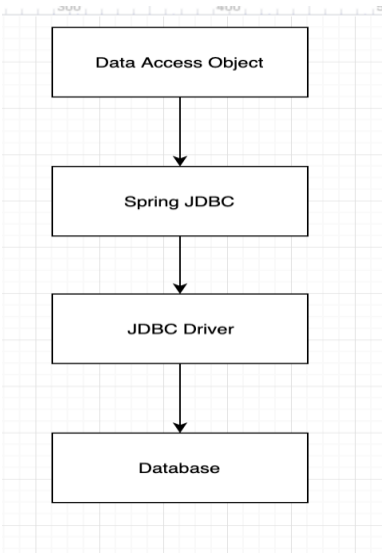
1. **Inversion of Control (IoC)** - In its broadest sense, this means that the framework handles responsibilities on behalf of the components that are managed within its context. The components themselves are simplified, because they are relieved of those responsibilities. For example, dependency injection relieves the components of the responsibility of locating or creating their dependencies.
2. **Provide choice at every level** - Spring lets you defer design decisions as late as possible. For example, you can switch persistence providers through configuration without changing your code. The same is true for many other infrastructure concerns and integration with third-party APIs.
3. **Accommodate diverse perspectives** - Spring embraces flexibility and is not opinionated about how things should be done. It supports a wide range of application needs with different perspectives.
4. **Maintain strong backward compatibility** - Spring's evolution has been carefully managed to force a few breaking changes between versions. Spring supports a carefully chosen range of JDK versions and third-party libraries to facilitate maintenance of applications and libraries that depend on Spring.

5. **Care about API design** - The Spring team puts a lot of thought and time into making APIs that are intuitive and that hold up across many versions and many years.
6. **Set high standards for code quality** - The Spring Framework puts a strong emphasis on meaningful, current, and accurate javadoc. It is one of very few projects that can claim clean code structure with no circular dependencies between packages.
7. **Don't Repeat Yourself (DRY)** - There should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When you use the @AspectJ style, this information is encapsulated in a single module: the aspect.
8. **Facilitate asynchronous, message-driven behavior within a Spring-based application** - Spring-integration supports message-driven architectures where inversion of control applies to runtime concerns, such as when certain business logic should run and where the response should be sent. It supports routing and transformation of messages so that different transports and different data formats can be integrated without impacting testability. In other words, the messaging and integration concerns are handled by the framework.
9. **Components should be loosely coupled for modularity and testability** - Dependency injection relieves the components of the responsibility of locating or creating their dependencies. Likewise, aspect-oriented programming relieves business components of generic cross-cutting concerns by modularizing them into reusable aspects. In each case, the end result is a system that is easier to test, understand, maintain, and extend
10. **Separation of Concerns** - framework should enforce separation of concerns between business logic and integration logic. Business components are further isolated from the infrastructure, and developers are relieved of complex integration responsibilities.
11. **Promote reuse and portability** - Extension points should be abstract in nature (but within well-defined boundaries) to promote reuse and portability.

4 ARCHITECTURE STYLES

4.1 LAYERED STYLE

The overall architecture of Spring framework as well as various sub packages including Spring Jdbc project follows a layered architecture style where each layer is only connected to its immediate neighbours.



4.2 RULE BASED

The most common form of architecture used in expert and other types of knowledge-based systems is the rule-based systems. This type of system uses knowledge encoded in the form of production rules i.e. if-then rules. The rule has a conditional part on the left-hand side and a conclusion or action part on the right-hand side. The rule to be applied for a specific scenario is determined by an inference engine. Spring framework uses Rule-based style majorly in the following two modules with support for both declarative and programmatic rule definitions:

1. Spring test
2. Spring-tx (transaction management)

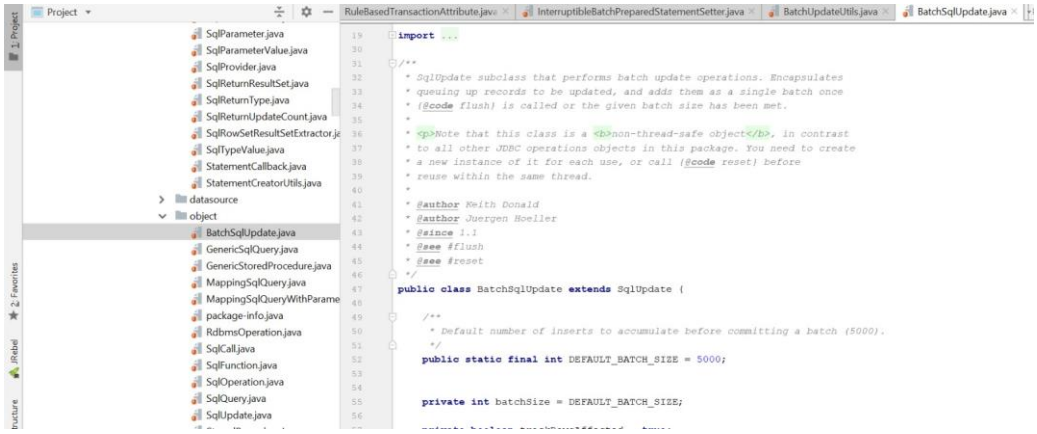
```
package org.springframework.transaction.interceptor;

import ...

/**
 * TransactionAttribute implementation that works out whether a given exception
 * should cause transaction rollback by applying a number of rollback rules,
 * both positive and negative. If no rules are relevant to the exception, it
 * behaves like DefaultTransactionAttribute (rolling back on runtime exceptions).
 *
 * <p>{@link TransactionAttributeEditor} creates objects of this class.
 *
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @since 09.04.2003
 * @see TransactionAttributeEditor
 */
@SuppressWarnings("serial")
public class RuleBasedTransactionAttribute extends DefaultTransactionAttribute implements Serializable {

    /** Prefix for rollback-on-exception rules in description strings. */
    public static final String PREFIX_ROLLBACK_RULE = "-";
```

4.3 BATCH SEQUENTIAL

Batch sequential is a classical data processing model, in which a data transformation subsystem can initiate its process only after its previous subsystem is completely through –

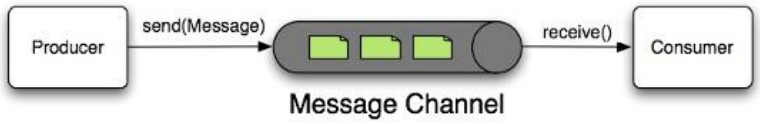
1. The flow of data carries a batch of data as a whole from one subsystem to another.
2. The communications between the modules are conducted through temporary intermediate files which can be removed by successive subsystems.

Spring-jdbc provide improved performance by creating batches of multiple calls to the same prepared statement. By grouping updates into batches you limit the number of round trips to the database. The JdbcTemplate batch processing is implemented using two methods of a special interface, BatchPreparedStatementSetter, and passing that in as the second parameter in batchUpdate method call. The getBatchSize method is used to get the size of the current batch.



4.4 PIPE AND FILTER

The Message Channel of Spring-integration module represents the “pipe” of a pipes-and-filters architecture. Producers send messages to a channel, and consumers receive messages from a channel. The message channel therefore decouples the messaging components and also provides a convenient point for interception and monitoring of messages. A Message Endpoint represents the “filter” of a pipes-and-filters architecture. The endpoint’s primary role is to connect application code to the messaging framework and to do so in a non-invasive manner.



4.5 POINT TO POINT

A message channel that follow point-to-point semantics, no more than one consumer can receive each message sent to the channel. The DirectChannel has point-to-point semantics. It implements the SubscribableChannel interface instead of the PollableChannel interface, so it dispatches messages directly to a subscriber. As a point-to-point channel, it sends each Message to a single subscribed MessageHandler.

Constructor Summary

Constructors

Constructor and Description
`DirectChannel()` Create a channel with default `RoundRobinLoadBalancingStrategy`
`DirectChannel(LoadBalancingStrategy loadBalancingStrategy)` Create a DirectChannel with a `LoadBalancingStrategy`.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`protected UnicastingDispatcher`	`getDispatcher()`
`protected void`	`onInit()` Subclasses may implement this for initialization logic.
`void`	`setFailover(boolean failover)` Specify whether the channel's dispatcher should have failover enabled.
`void`	`setMaxSubscribers(int maxSubscribers)` Specify the maximum number of subscribers supported by the channel's dispatcher.

4.6 PUBLISH-SUBSCRIBE

A message channel that follow Publish-subscribe semantics attempt to broadcast each message to all subscribers on the channel. The PublishSubscribeChannel implementation broadcasts any Message sent to it to all of its subscribed handlers. This is most often used for sending event messages, whose primary role is notification.

Constructor Summary

Constructors

Constructor and Description
`PublishSubscribeChannel()` Create a PublishSubscribeChannel that will invoke the handlers in the message sender's thread.
`PublishSubscribeChannel(Executor executor)` Create a PublishSubscribeChannel that will use an `Executor` to invoke the handlers.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`String`	`getComponentType()` Subclasses may implement this method to provide component type information.
`protected BroadcastingDispatcher`	`getDispatcher()`
`void`	`onInit()` Callback method for initialization.
`void`	`setApplySequence(boolean applySequence)` Specify whether to apply the sequence number and size headers to the messages prior to invoking the subscribed handlers.
`void`	`setErrorHandler(ErrorHandler errorHandler)` Provide an `ErrorHandler` strategy for handling Exceptions that occur downstream from this channel.
`void`	`setIgnoreFailures(boolean ignoreFailures)` Specify whether failures for one or more of the handlers should be ignored.
`void`	`setMinSubscribers(int minSubscribers)`

5 DESIGN PATTERNS

5.1 FACTORY

This pattern allows the initialization of object through a public static method, called the factory method. The Spring framework uses the factory design pattern for the creation of the object of beans by using Spring BeanFactory Container approach. It is the simplest container present in the spring framework which provides the basic support for DI (Dependency Injection).

Source: org.springframework.beans.factory.BeanFactory

```
/**
 * The root interface for accessing a Spring bean container.
 * This is the basic client view of a bean container;
 * further interfaces such as {@link ListableBeanFactory} and
 * {@link org.springframework.beans.factory.config.ConfigurableBeanFactory}
 * are available for specific purposes.
 *
 * <p>This interface is implemented by objects that hold a number of bean definitions,
 * each uniquely identified by a String name. Depending on the bean definition,
 * the factory will return either an independent instance of a contained object
 * (the Prototype design pattern), or a single shared instance (a superior
 * alternative to the Singleton design pattern, in which the instance is a
 * singleton in the scope of the factory). Which type of instance will be returned
 * depends on the bean factory configuration: the API is the same. Since Spring
 * 2.0, further scopes are available depending on the concrete application
 * context (e.g. "request" and "session" scopes in a web environment).
 *

public interface BeanFactory {

        /**
         * Used to dereference a {@link FactoryBean} instance and distinguish it from
         * beans <i>created</i> by the FactoryBean. For example, if the bean named
         * {@code myJndiObject} is a FactoryBean, getting {@code &myJndiObject}
         * will return the factory, not the instance returned by the factory.
         */
        String FACTORY_BEAN_PREFIX = "&";


        /**
         * Return an instance, which may be shared or independent, of the specified bean.
         * <p>This method allows a Spring BeanFactory to be used as a replacement for the
         * Singleton or Prototype design pattern. Callers may retain references to
         * returned objects in the case of Singleton beans.
         * <p>Translates aliases back to the corresponding canonical bean name.
         * Will ask the parent factory if the bean cannot be found in this factory instance.
         * @param name the name of the bean to retrieve
         * @return an instance of the bean
         * @throws NoSuchBeanDefinitionException if there is no bean with the specified name
         * @throws BeansException if the bean could not be obtained
         */
        Object getBean(String name) throws BeansException;
```

5.2 FRONT CONTROLLER

The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. Spring uses this by

providing "DispatcherServlet" to ensure an incoming request gets dispatched to your controllers.

Source: org.springframework.web.servlet.DispatcherServlet

```
/**
 * Central dispatcher for HTTP request handlers/controllers, e.g. for web UI controllers
 * or HTTP-based remote service exporters. Dispatches to registered handlers for processing
 * a web request, providing convenient mapping and exception handling facilities.
 *
 * <p>This servlet is very flexible: It can be used with just about any workflow, with the
 * installation of the appropriate adapter classes. It offers the following functionality
 * that distinguishes it from other request-driven web MVC frameworks:
 *
 * <ul>
 * <li>It is based around a JavaBeans configuration mechanism.
 *
 * <li>It can use any {@link HandlerMapping} implementation - pre-built or provided as part
 * of an application - to control the routing of requests to handler objects. Default is
 * {@link org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping} and
 * {@link org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping}.
 * HandlerMapping objects can be defined as beans in the servlet's application context,
 * implementing the HandlerMapping interface, overriding the default HandlerMapping if
 * present. HandlerMappings can be given any bean name (they are tested by type).
 *
```

```
/**
 * Return the HandlerExecutionChain for this request.
 * <p>Tries all handler mappings in order.
 * @param request current HTTP request
 * @return the HandlerExecutionChain, or {@code null} if no handler could be found
 */
@Nullable
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
        if (this.handlerMappings != null) {
                for (HandlerMapping mapping : this.handlerMappings) {
                        HandlerExecutionChain handler = mapping.getHandler(request);
                        if (handler != null) {
                                return handler;
                        }
                }
        }
        return null;
}
```

5.3 BUILDER

Builder pattern aims to separate the construction of a complex object from its representation so that the same construction process can create different representations. It is used to construct a complex object step by step and the final step will return the object. Spring provides programmatic means of constructing BeanDefinitions using the builder pattern through Class "BeanDefinitionBuilder".

Source: org.springframework.beans.factory.support.BeanDefinitionBuilder

```
/**
 * Programmatic means of constructing
 * {@link org.springframework.beans.factory.config.BeanDefinition BeanDefinitions}
 * using the builder pattern. Intended primarily for use when implementing Spring 2.0
 * {@link org.springframework.beans.factory.xml.NamespaceHandler NamespaceHandlers}.
 *
 * @author Rod Johnson
 * @author Rob Harrop
 * @author Juergen Hoeller
 * @since 2.0
 */
public final class BeanDefinitionBuilder {

        /**
         * Create a new {@code BeanDefinitionBuilder} used to construct a {@link
GenericBeanDefinition}.
         */
        public static BeanDefinitionBuilder genericBeanDefinition() {
                return new BeanDefinitionBuilder(new GenericBeanDefinition());
        }
```

```
public final class BeanDefinitionBuilder {

	/**
	 * Create a new {@code BeanDefinitionBuilder} used to construct a {@link
GenericBeanDefinition}.
	 */
	public static BeanDefinitionBuilder genericBeanDefinition() {
		return new BeanDefinitionBuilder(new GenericBeanDefinition());
	}

	/**
	 * Create a new {@code BeanDefinitionBuilder} used to construct a {@link
GenericBeanDefinition}.
	 * @param beanClassName the class name for the bean that the definition is being created for
	 */
	public static BeanDefinitionBuilder genericBeanDefinition(String beanClassName) {
		BeanDefinitionBuilder builder = new BeanDefinitionBuilder(new GenericBeanDefinition
());
		builder.beanDefinition.setBeanClassName(beanClassName);
		return builder;
	}
```

5.4 PROXY

Proxy pattern is used heavily in AOP, and remoting. A good example of proxy design pattern is ProxyFactoryBean class. This factory constructs AOP proxy based on Spring beans. The proxy provides a surrogate or placeholder for another object to control access to it.

Source: org.springframework.aop.framework.ProxyFactoryBean

```
 * <p>Creates a JDK proxy when proxy interfaces are given, and a CGLIB proxy for the
 * actual target class if not. Note that the latter will only work if the target class
 * does not have final methods, as a dynamic subclass will be created at runtime.
 *
 * <p>It's possible to cast a proxy obtained from this factory to {@link Advised},
 * or to obtain the ProxyFactoryBean reference and programmatically manipulate it.
 * This won't work for existing prototype references, which are independent. However,
 * it will work for prototypes subsequently obtained from the factory. Changes to
 * interception will work immediately on singletons (including existing references).
 * However, to change interfaces or target it's necessary to obtain a new instance
 * from the factory. This means that singleton instances obtained from the factory
 * do not have the same object identity. However, they do have the same interceptors
 * and target, and changing any reference will change all objects.


/**
 * Set the ClassLoader to generate the proxy class in.
 * <p>Default is the bean ClassLoader, i.e. the ClassLoader used by the
 * containing BeanFactory for loading all bean classes. This can be
 * overridden here for specific proxies.
 */
public void setProxyClassLoader(@Nullable ClassLoader classLoader) {
	this.proxyClassLoader = classLoader;
	this.classLoaderConfigured = (classLoader != null);
}
```

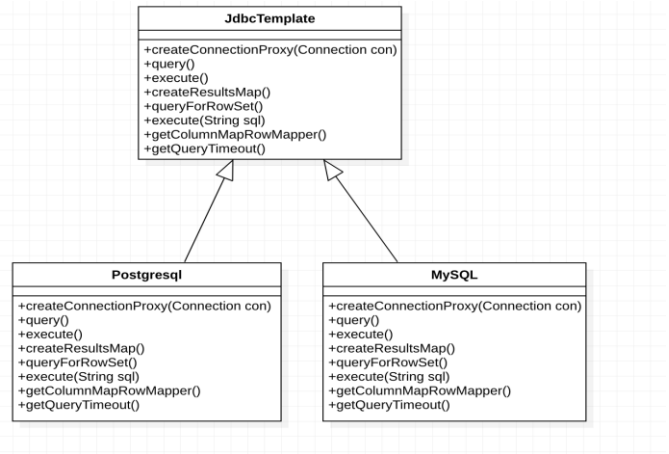
5.5 TEMPLATE

Template pattern defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior. In spring framework, it is extensively used in JDBC module to built JDBC template.

Source: org.springframework.jdbc.core.JdbcTemplate

```
/**
 * <b>This is the central class in the JDBC core package.</b>
 * It simplifies the use of JDBC and helps to avoid common errors.
 * It executes core JDBC workflow, leaving application code to provide SQL
 * and extract results. This class executes SQL queries or updates, initiating
 * iteration over ResultSets and catching JDBC exceptions and translating
 * them to the generic, more informative exception hierarchy defined in the
 * {@code org.springframework.dao} package.
 *
 * <p>Code using this class need only implement callback interfaces, giving
 * them a clearly defined contract. The {@link PreparedStatementCreator} callback
 * interface creates a prepared statement given a Connection, providing SQL and
 * any necessary parameters. The {@link ResultSetExtractor} interface extracts
 * values from a ResultSet. See also {@link PreparedStatementSetter} and
 * {@link RowMapper} for two popular alternative callback interfaces.
 *
```

```
/**
 * Construct a new JdbcTemplate, given a DataSource to obtain connections from.
 * <p>Note: Depending on the "lazyInit" flag, initialization of the exception translator
 * will be triggered.
 * @param dataSource the JDBC DataSource to obtain connections from
 * @param lazyInit whether to lazily initialize the SQLExceptionTranslator
 */
public JdbcTemplate(DataSource dataSource, boolean lazyInit) {
        setDataSource(dataSource);
        setLazyInit(lazyInit);
        afterPropertiesSet();
}
```



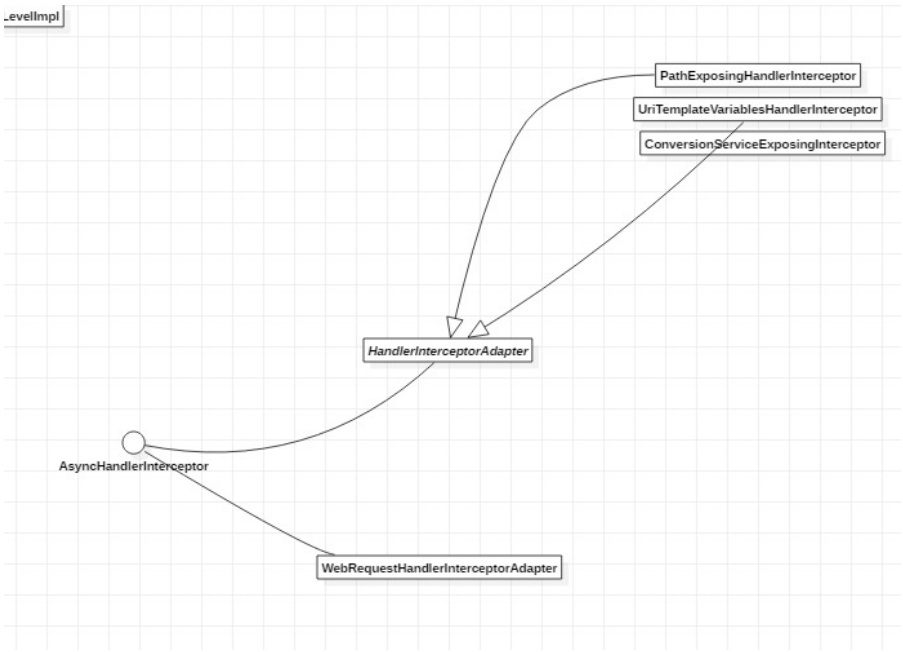
5.6 SINGLETON

Singleton design pattern ensures that there will exist only the single instance of the object in the memory that could provide services. In the spring framework, the Singleton is the default scope and the IOC container creates exactly one instance of the object per spring IOC container. Spring container will store this single instance in a cache of singleton beans, and all following requests and references for that named bean will get the cached object as return bean. It is recommended to use the singleton scope for stateless beans.

```
public class MethodInvokingFactoryBean extends MethodInvokingBean implements FactoryBean<Object> {

        private boolean singleton = true;

        private boolean initialized = false;

        /** Method call result in the singleton case. */
        @Nullable
        private Object singletonObject;


        /**
         * Set if a singleton should be created, or a new object on each
         * {@link #getObject()} request otherwise. Default is "true".
         */
        public void setSingleton(boolean singleton) {
                this.singleton = singleton;
        }

        @Override
        public void afterPropertiesSet() throws Exception {
                prepare();
                if (this.singleton) {
                        this.initialized = true;
                        this.singletonObject = invokeWithTargetException();
                }
        }


/**
 * Returns the same value each time if the singleton property is set
 * to "true", otherwise returns the value returned from invoking the
 * specified method on the fly.
 */
@Override
@Nullable
public Object getObject() throws Exception {
        if (this.singleton) {
                if (!this.initialized) {
                        throw new FactoryBeanNotInitializedException();
                }
                // Singleton: return shared object.
                return this.singletonObject;
        }
        else {
                // Prototype: new object on each call.
                return invokeWithTargetException();
        }
}
```

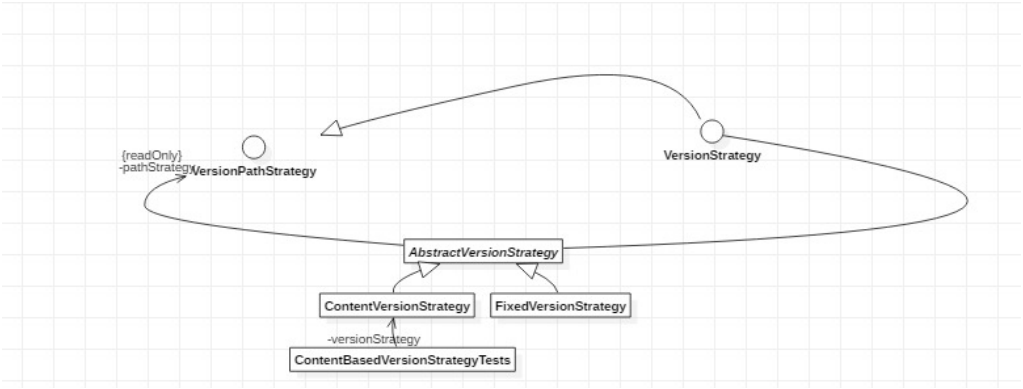
5.7 ADAPTER PATTERN

The adapter pattern is a software design pattern that allows the interface of an existing class to be used as another interface. This HandlerInterceptorAdapter is an abstract adapter class for the interface, for simplified implementation of pre-only/post-only interceptors.



5.8 STRATEGY

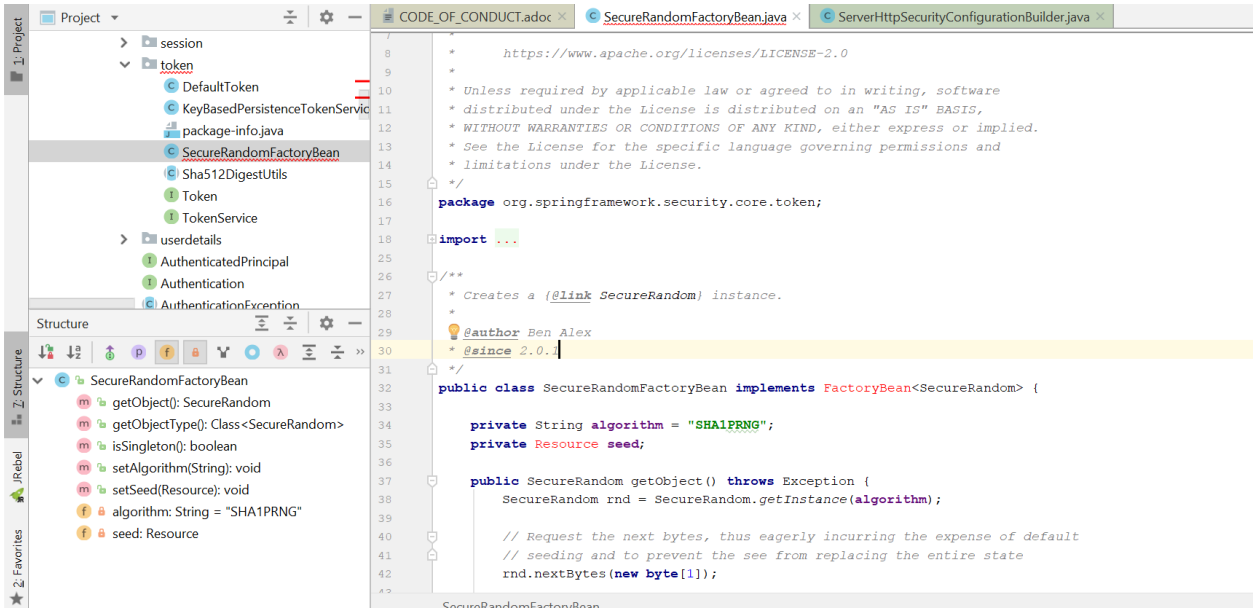
The strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use. The AbstractVersionStrategy allows for selecting the version strategies between calculating Hex MD5 hashes from the content of the resource and appends it to the file name or on a fixed version applied as a request path prefix, e.g. reduced SHA, version name, release date, etc. The FixedVersionStrategy is useful for example when ContentVersionStrategy cannot be used such as when using javascript module loaders which are incharge of loading the Javascript resources and need to know their relative paths.



6 SECURE DESIGN PATTERNS

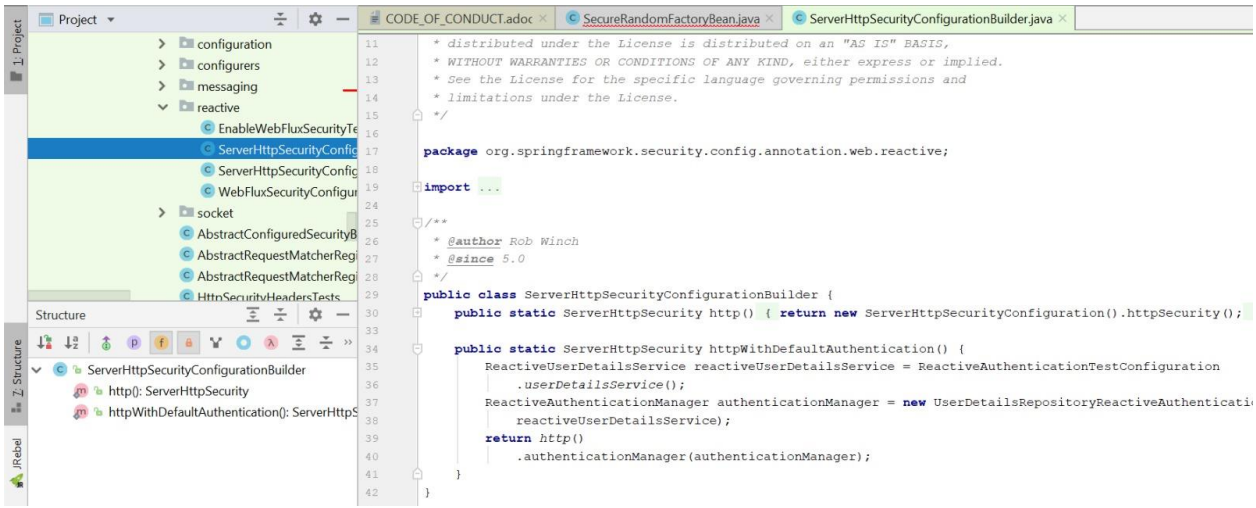
6.1 SECURE FACTORY

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.



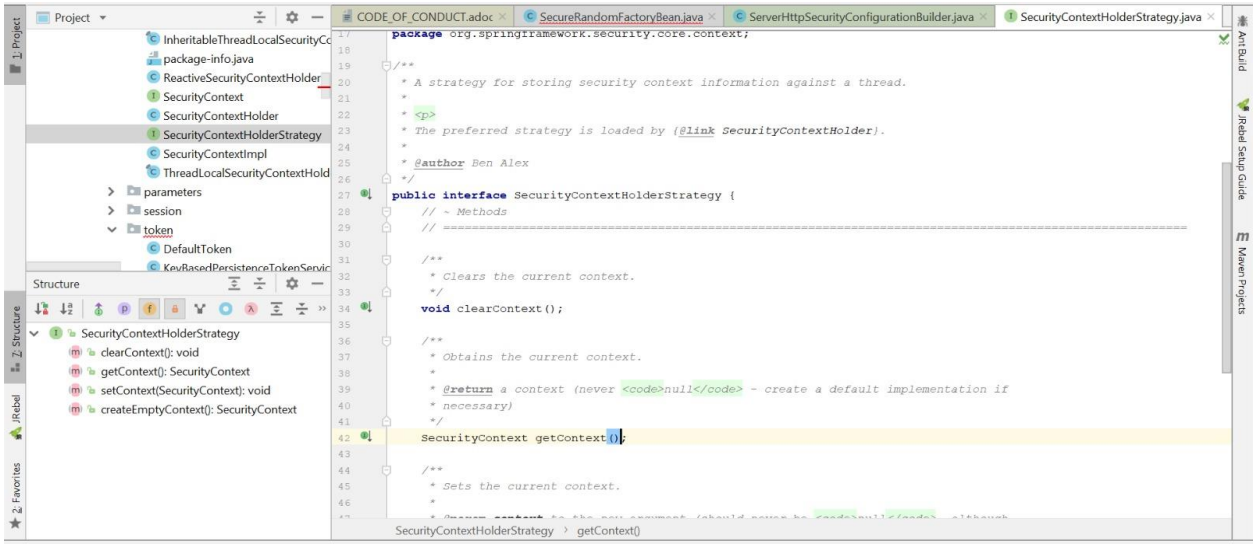
6.2 SECURE BUILDER

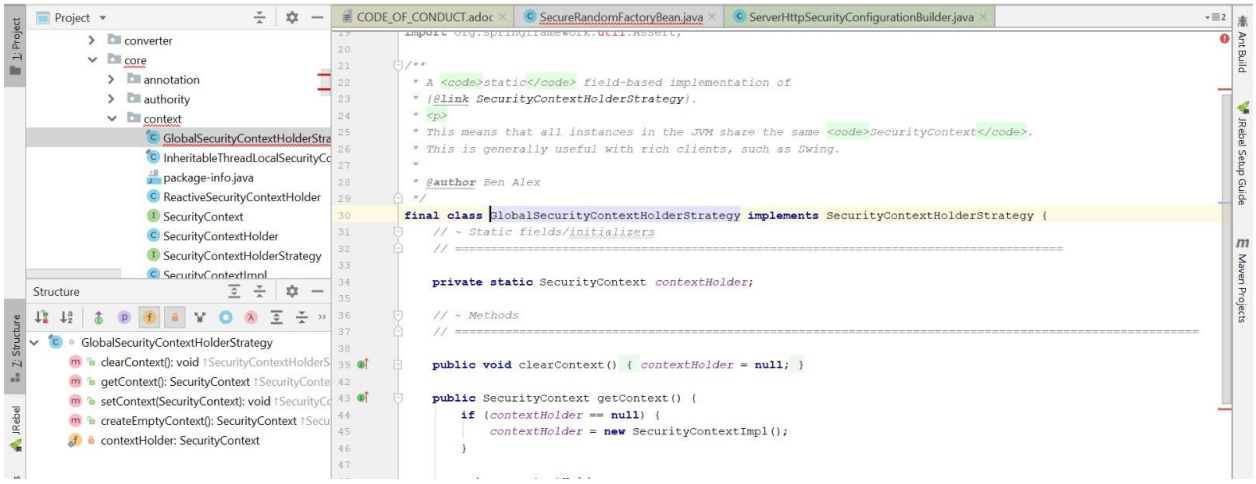
The intent of the Secure Builder Factory secure design pattern is to separate the security dependent rules involved in creating a complex object from the basic steps involved in actually creating the object. ServerHttpSecurityConfigurationBuilder is the class that aligns with secure builder design pattern in spring framework.



6.3 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.





7 DETAILED DESIGN

7.1 ASPECT ORIENTED PROGRAMMING (AOP)

One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework. Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. There are various common good examples of aspects like logging, auditing, declarative transactions, security, caching, etc.

Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. Spring AOP module provides interceptors to intercept an application. For example, when a method is executed, you can add extra functionality before or after the method execution.

Type	Name	Terms & Description
Component	Aspect	This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.
Component	Join point	This represents a point in your application where you can plug-in the AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.
Connector	Advice	This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework.
Component	Pointcut	This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples.
Connector	Introduction	An introduction allows you to add new methods or attributes to the existing classes.
Component	Target object	The object being advised by one or more aspects. This object will always be a proxied object, also referred to as the advised object.
	Weaving	Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

7.2 COMPONENTS

The Components of the specific Design patterns discussed in Section Design Patterns are the following:

7.2.1 BeanFactory

org.springframework.beans.factory

7.2.1.1 Purpose

The root interface for accessing a Spring bean container. This is the basic client view of a bean container. This interface is implemented by objects that hold a number of bean definitions, each uniquely identified by a String name. Depending on the bean definition, the factory will return either an independent instance of a contained object (the Prototype design pattern), or a single shared instance (a superior alternative to the Singleton design pattern, in which the instance is a singleton in the scope of the factory). Which type of instance will be returned depends on the bean factory configuration.

7.2.1.2 Required Interface

Initialization of bean in user program.

7.2.1.3 Provided Interface

Modifier and Type	Method and Description
`boolean`	`containsBean(String name)` Does this bean factory contain a bean definition or externally registered singleton instance with the given name?
`String[]`	`getAliases(String name)` Return the aliases for the given bean name, if any.
`<T> T`	`getBean(Class<T> requiredType)` Return the bean instance that uniquely matches the given object type, if any.
`<T> T`	`getBean(Class<T> requiredType, Object... args)` Return an instance, which may be shared or independent, of the specified bean.
`Object`	`getBean(String name)` Return an instance, which may be shared or independent, of the specified bean.
`<T> T`	`getBean(String name, Class<T> requiredType)` Return an instance, which may be shared or independent, of the specified bean.
`Object`	`getBean(String name, Object... args)` Return an instance, which may be shared or independent, of the specified bean.
`<T> ObjectProvider<T>`	`getBeanProvider(Class<T> requiredType)` Return a provider for the specified bean, allowing for lazy on-demand retrieval of instances, including availability and uniqueness options.
`<T> ObjectProvider<T>`	`getBeanProvider(ResolvableType requiredType)` Return a provider for the specified bean, allowing for lazy on-demand retrieval of instances, including availability and uniqueness options.
`Class<?>`	`getType(String name)` Determine the type of the bean with the given name.
`Class<?>`	`getType(String name, boolean allowFactoryBeanInit)` Determine the type of the bean with the given name.
`boolean`	`isPrototype(String name)` Is this bean a prototype? That is, will `getBean(java.lang.String)` always return independent instances?
`boolean`	`isSingleton(String name)` Is this bean a shared singleton? That is, will `getBean(java.lang.String)` always return the same instance?
`boolean`	`isTypeMatch(String name, Class<?> typeToMatch)` Check whether the bean with the given name matches the specified type.

7.2.2 DispatcherServlet

org.springframework.web.servlet.DispatcherServlet

7.2.2.1 Purpose

Central dispatcher for HTTP request handlers/controllers, e.g. for web UI controllers or HTTP-based remote service exporters. Dispatches to registered handlers for processing a web request, providing convenient mapping and exception handling facilities.This servlet is very flexible: It can be used with just about any workflow, with the installation of the appropriate adapter classes.

7.2.2.2 Required Interface

HttpServletBean component is required for DispatcherServlet because DispatcherServlet is subclass of HttpServeltBean class. HttpServletBean class is handy superclass for any type of servlet. Type conversion of config parameters is automatic, with the corresponding setter

method getting invoked with the converted value. It is also possible for subclasses to specify required properties. Parameters without matching bean property setter will simply be ignored.

Modifier and Type	Method and Description
protected void	**addRequiredProperty**(String property) Subclasses can invoke this method to specify that this property (which must match a JavaBean property they expose) is mandatory, and must be supplied as a config parameter.
protected ConfigurableEnvironment	**createEnvironment**() Create and return a new StandardServletEnvironment.
ConfigurableEnvironment	**getEnvironment**() Return the Environment associated with this servlet.
String	**getServletName**() Overridden method that simply returns null when no ServletConfig set yet.
void	**init**() Map config parameters onto bean properties of this servlet, and invoke subclass initialization.
protected void	**initBeanWrapper**(BeanWrapper bw) Initialize the BeanWrapper for this HttpServletBean, possibly with custom editors.
protected void	**initServletBean**() Subclasses may override this to perform custom initialization.
void	**setEnvironment**(Environment environment) Set the Environment that this servlet runs in.

7.2.2.3 Provided Interface

Modifier and Type	Method and Description
protected LocaleContext	**buildLocaleContext**(HttpServletRequest request) Build a LocaleContext for the given request, exposing the request's primary locale as current locale.
protected HttpServletRequest	**checkMultipart**(HttpServletRequest request) Convert the request into a multipart request, and make multipart resolver available.
protected void	**cleanupMultipart**(HttpServletRequest request) Clean up any resources used by the given multipart request (if any).
protected Object	**createDefaultStrategy**(ApplicationContext context, Class<?> clazz) Create a default strategy.
protected void	**doDispatch**(HttpServletRequest request, HttpServletResponse response) Process the actual dispatching to the handler.
protected void	**doService**(HttpServletRequest request, HttpServletResponse response) Exposes the DispatcherServlet-specific request attributes and delegates to **doDispatch**(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) for the actual dispatching.
protected <T> List<T>	**getDefaultStrategies**(ApplicationContext context, Class<T> strategyInterface) Create a List of default strategy objects for the given strategy interface.
protected <T> T	**getDefaultStrategy**(ApplicationContext context, Class<T> strategyInterface) Return the default strategy object for the given strategy interface.
protected String	**getDefaultViewName**(HttpServletRequest request) Translate the supplied request into a default view name.
protected HandlerExecutionChain	**getHandler**(HttpServletRequest request) Return the HandlerExecutionChain for this request.
protected HandlerAdapter	**getHandlerAdapter**(Object handler) Return the HandlerAdapter for this handler object.
List<HandlerMapping>	**getHandlerMappings**() Return the configured HandlerMapping beans that were detected by type in the WebApplicationContext or initialized based on the default set of strategies from DispatcherServlet.properties.
MultipartResolver	**getMultipartResolver**() Obtain this servlet's MultipartResolver, if any.
ThemeSource	**getThemeSource**() Return this servlet's ThemeSource, if any; else return null.
protected void	**initStrategies**(ApplicationContext context) Initialize the strategy objects that this servlet uses.
protected void	**noHandlerFound**(HttpServletRequest request, HttpServletResponse response) No handler found -> set appropriate HTTP response status.
protected void	**onRefresh**(ApplicationContext context) This implementation calls **initStrategies**(org.springframework.context.ApplicationContext).
protected ModelAndView	**processHandlerException**(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) Determine an error ModelAndView via the registered HandlerExceptionResolvers.
protected void	**render**(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) Render the given ModelAndView.
protected View	**resolveViewName**(String viewName, Map<String,Object> model, Locale locale, HttpServletRequest request) Resolve the given view name into a View object (to be rendered).
void	**setCleanupAfterInclude**(boolean cleanupAfterInclude) Set whether to perform cleanup of request attributes after an include request, that is, whether to reset the original state of all request attributes after the DispatcherServlet has processed within an include request.
void	**setDetectAllHandlerAdapters**(boolean detectAllHandlerAdapters) Set whether to detect all HandlerAdapter beans in this servlet's context.
void	**setDetectAllHandlerExceptionResolvers**(boolean detectAllHandlerExceptionResolvers) Set whether to detect all HandlerExceptionResolver beans in this servlet's context.

7.2.3 JdbcTemplate

org.springframework.jdbc.core.JdbcTemplate

7.2.3.1 Purpose

It simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving application code to provide SQL and extract results. This class executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative exception hierarchy defined in the org.springframework.dao package.

7.2.3.2 Required Interface

JdbcAccessor is the base class for JdbcTemplate and other JDBC-accessing DAO helpers, defining common properties such as DataSource and exception translator. Below methods are required interface for JDBCTemplate component.

Modifier and Type	Method and Description
void	**afterPropertiesSet**() Eagerly initialize the exception translator, if demanded, creating a default one for the specified DataSource if none set.
DataSource	**getDataSource**() Return the DataSource used by this template.
SQLExceptionTranslator	**getExceptionTranslator**() Return the exception translator for this instance.
boolean	**isLazyInit**() Return whether to lazily initialize the SQLExceptionTranslator for this accessor.
protected DataSource	**obtainDataSource**() Obtain the DataSource for actual use.
void	**setDatabaseProductName**(String dbName) Specify the database product name for the DataSource that this accessor uses.
void	**setDataSource**(DataSource dataSource) Set the JDBC DataSource to obtain connections from.
void	**setExceptionTranslator**(SQLExceptionTranslator exceptionTranslator) Set the exception translator for this instance.
void	**setLazyInit**(boolean lazyInit) Set whether to lazily initialize the SQLExceptionTranslator for this accessor, on first encounter of an SQLException.

7.2.3.3 Provided Interface

Modifier and Type	Method and Description
protected void	**applyStatementSettings**(Statement stmt) Prepare the given JDBC Statement (or PreparedStatement or CallableStatement), applying statement settings such as fetch size, max rows, and query timeout.
int[]	**batchUpdate**(String... sql) Issue multiple SQL updates on a single JDBC Statement using batching.
int[]	**batchUpdate**(String sql, BatchPreparedStatementSetter pss) Issue multiple update statements on a single PreparedStatement, using batch updates and a BatchPreparedStatementSetter to set values.
<T> int[][]	**batchUpdate**(String sql, Collection<T> batchArgs, int batchSize, ParameterizedPreparedStatementSetter<T> pss) Execute multiple batches using the supplied SQL statement with the collect of supplied arguments.
int[]	**batchUpdate**(String sql, List<Object[]> batchArgs) Execute a batch using the supplied SQL statement with the batch of supplied arguments.
int[]	**batchUpdate**(String sql, List<Object[]> batchArgs, int[] argTypes) Execute a batch using the supplied SQL statement with the batch of supplied arguments.
Map<String,Object>	**call**(CallableStatementCreator csc, List<SqlParameter> declaredParameters) Execute an SQL call using a CallableStatementCreator to provide SQL and any required parameters.
protected Connection	**createConnectionProxy**(Connection con) Create a close-suppressing proxy for the given JDBC Connection.
protected Map<String,Object>	**createResultsMap**() Create a Map instance to be used as the results map.
<T> T	**execute**(CallableStatementCreator csc, CallableStatementCallback<T> action) Execute a JDBC data access operation, implemented as callback action working on a JDBC CallableStatement.
boolean	**isResultsMapCaseInsensitive**() Return whether execution of a CallableStatement will return the results in a Map that uses case insensitive names for the parameters.
boolean	**isSkipResultsProcessing**() Return whether results processing should be skipped.
boolean	**isSkipUndeclaredResults**() Return whether undeclared results should be skipped.
protected PreparedStatementSetter	**newArgPreparedStatementSetter**(Object[] args) Create a new arg-based PreparedStatementSetter using the args passed in.
protected PreparedStatementSetter	**newArgTypePreparedStatementSetter**(Object[] args, int[] argTypes) Create a new arg-type-based PreparedStatementSetter using the args and types passed in.
protected Map<String,Object>	**processResultSet**(ResultSet rs, ResultSetSupportingSqlParameter param) Process the given ResultSet from a stored procedure.
<T> T	**query**(PreparedStatementCreator psc, PreparedStatementSetter pss, ResultSetExtractor<T> rse) Query using a prepared statement, allowing for a PreparedStatementCreator and a PreparedStatementSetter.

void	**setQueryTimeout**(int queryTimeout) Set the query timeout for statements that this JdbcTemplate executes.
void	**setResultsMapCaseInsensitive**(boolean resultsMapCaseInsensitive) Set whether execution of a CallableStatement will return the results in a Map that uses case insensitive names for the parameters.
void	**setSkipResultsProcessing**(boolean skipResultsProcessing) Set whether results processing should be skipped.
void	**setSkipUndeclaredResults**(boolean skipUndeclaredResults) Set whether undeclared results should be skipped.
protected DataAccessException	**translateException**(String task, String sql, SQLException ex) Translate the given SQLException into a generic DataAccessException.
int	**update**(PreparedStatementCreator psc) Issue a single SQL update operation (such as an insert, update or delete statement) using a PreparedStatementCreator to provide SQL and any required parameters.

7.2.4 HandlerInterceptorAdapter

Org.springframework.web.servlet.handler.HandlerInterceptorAdapter

7.2.4.1 Purpose

Abstract adapter class for the AsyncHandlerInterceptor interface, for simplified implementation of pre-only/post-only interceptors.

7.2.4.2 Required Interface

AsyncHandlerInterceptor is implemented by HandlerInterceptor to do work when an async request times out or completes with a network error. For such cases the Servlet container does not dispatch and therefore the postHandle and afterCompletion methods will not be invoked. Instead, interceptors can register to track an asynchronous request through the registerCallbackInterceptor and registerDeferredResultInterceptor methods on WebAsyncManager. This can be done proactively on every request from preHandle regardless of whether async request processing will start.

Modifier and Type	Method and Description
default void	**afterConcurrentHandlingStarted**(HttpServletRequest request, HttpServletResponse response, Object handler) Called instead of postHandle and afterCompletion when the handler is being executed concurrently.

7.2.4.3 Provided Interface

Modifier and Type	Method and Description
void	**afterCompletion**(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) This implementation is empty.
void	**afterConcurrentHandlingStarted**(HttpServletRequest request, HttpServletResponse response, Object handler) This implementation is empty.
void	**postHandle**(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) This implementation is empty.
boolean	**preHandle**(HttpServletRequest request, HttpServletResponse response, Object handler) This implementation always returns true.

7.2.5 VersionStrategy

Org.springframework.web.servlet.resource

All Known Implementing Classes are AbstractVersionStrategy, ContentVersionStrategy, FixedVersionStrategy

7.2.5.1 Purpose

An extension of VersionPathStrategy that adds a method to determine the actual version of a Resource.

7.2.5.2 Required Interface

VersionPathStrategy is implemented by VersionStrategy for extracting and embedding a resource version in its URL path.

Modifier and Type	Method and Description
String	addVersion(String requestPath, String version) Add a version to the given request path.
String	extractVersion(String requestPath) Extract the resource version from the request path.
String	removeVersion(String requestPath, String version) Remove the version from the request path.

7.2.5.3 Provided Interface

Modifier and Type	Method and Description
String	getResourceVersion(Resource resource) Determine the version for the given resource.

7.2.6 MethodInvokingFactoryBean

Org.springframework.beans.factory.config

7.2.6.1 Purpose

FactoryBean which returns a value which is the result of a static or instance method invocation. For most use cases it is better to just use the container's built-in factory method support for the same purpose, since that is smarter at converting arguments. This factory bean is still useful though when you need to call a method which doesn't return any value (for example, a static class method to force some sort of initialization to happen). This use case is not supported by factory methods, since a return value is needed to obtain the bean instance.All Implemented Interfaces are Aware, BeanClassLoaderAware, BeanFactoryAware, FactoryBean<Object>, InitializingBean

Constructor is public MethodInvokingFactoryBean()

7.2.6.2 Required Interface

FactoryBean<T>

Interface to be implemented by objects used within a BeanFactory that are themselves factories for individual objects. If a bean implements this interface, it is used as a factory for an object to expose, not directly as a bean instance that will be exposed itself.

Modifier and Type	Method and Description
T	getObject() Return an instance (possibly shared or independent) of the object managed by this factory.
Class<?>	getObjectType() Return the type of object that this FactoryBean creates, or null if not known in advance.
default boolean	isSingleton() Is the object managed by this factory a singleton? That is, will getObject() always return the same object (a reference that can be cached)?

7.2.6.3 Provided Interface

Modifier and Type	Method and Description
void	afterPropertiesSet() Invoked by the containing BeanFactory after it has set all bean properties and satisfied BeanFactoryAware, ApplicationContextAware etc.
Object	getObject() Returns the same value each time if the singleton property is set to "true", otherwise returns the value returned from invoking the specified method on the fly.
Class<?>	getObjectType() Return the type of object that this FactoryBean creates, or null if not known in advance.
boolean	isSingleton() Is the object managed by this factory a singleton? That is, will FactoryBean.getObject() always return the same object (a reference that can be cached)?
void	setSingleton(boolean singleton) Set if a singleton should be created, or a new object on each getObject() request otherwise.

7.2.7 SecureRandomFactoryBean

Org.springframework.security.core.token

7.2.7.1 Purpose

Creates a SecureRandom instance.

7.2.7.2 Required Interface

FactoryBean<T>

Interface to be implemented by objects used within a BeanFactory which are themselves factories. If a bean implements this interface, it is used as a factory for an object to expose, not directly as a bean instance that will be exposed itself.

Modifier and Type	Method and Description
`T`	`getObject()` Return an instance (possibly shared or independent) of the object managed by this factory.
`Class<?>`	`getObjectType()` Return the type of object that this FactoryBean creates, or `null` if not known in advance.
`default boolean`	`isSingleton()` Is the object managed by this factory a singleton? That is, will `getObject()` always return the same object (a reference that can be cached)?

7.2.7.3 Provided Interface

Modifier and Type	Method and Description
`java.security.SecureRandom`	`getObject()`
`java.lang.Class<java.security.SecureRandom>`	`getObjectType()`
`boolean`	`isSingleton()`
`void`	`setAlgorithm(java.lang.String algorithm)` Allows the Pseudo Random Number Generator (PRNG) algorithm to be nominated.
`void`	`setSeed(org.springframework.core.io.Resource seed)` Allows the user to specify a resource which will act as a seed for the `SecureRandom` instance.

7.2.8 SecurityContextHolderStrategy

Org.springframework.security.core.context

7.2.8.1 Purpose

A strategy for storing security context information against a thread.The preferred strategy is loaded by SecurityContextHolder.

7.2.8.2 Required Interface

No required interface

7.2.8.3 Provided Interface

Modifier and Type	Method and Description
`void`	`clearContext()` Clears the current context.
`SecurityContext`	`createEmptyContext()` Creates a new, empty context implementation, for use by `SecurityContextRepository` implementations, when creating a new context for the first time.
`SecurityContext`	`getContext()` Obtains the current context.
`void`	`setContext(SecurityContext context)` Sets the current context.

7.2.9 GlobalSecurityContextHolderStrategy

org.springframework.security.context

7.2.10 Purpose

A static field-based implementation of SecurityContextHolderStrategy.This means that all instances in the JVM share the same SecurityContext. This is generally useful with rich clients, such as Swing.

7.2.11 Required Interface

SecurityContextHolderStrategy is a strategy for storing security context information against a thread.The preferred strategy is loaded by SecurityContextHolder

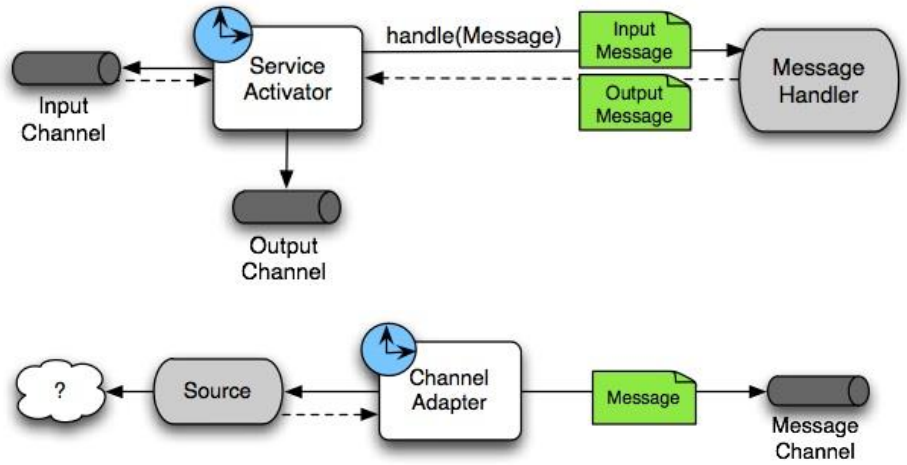
Modifier and Type	Method and Description
void	`clearContext()` Clears the current context.
SecurityContext	`createEmptyContext()` Creates a new, empty context implementation, for use by `SecurityContextRepository` implementations, when creating a new context for the first time.
SecurityContext	`getContext()` Obtains the current context.
void	`setContext(SecurityContext context)` Sets the current context.

7.2.12 Provided Interface

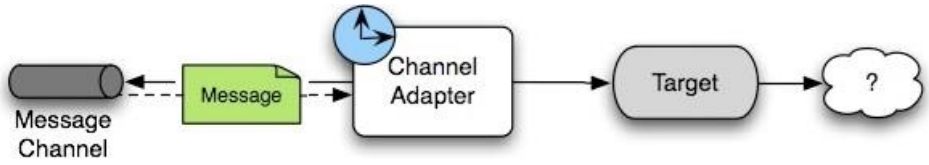
void	**clearContext**() Clears the current context.
SecurityContext	**getContext**() Obtains the current context.
void	**setContext**(SecurityContext context) Sets the current context.

7.3 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:



An inbound channel adapter endpoint connects a source system to a MessageChannel



An outbound channel adapter endpoint connects a MessageChannel to a target system

8 ARCHITECTURAL STYLES ALIGNMENT WITH PRINCIPAL DESIGN DECISIONS

1. The separation of concerns decision is supported by the layered architecture where each layer performs a different function and the different security design patterns which isolate the security concerns for the framework.
2. Spring-integration extends the Spring programming model into the messaging domain and builds upon Spring's existing enterprise integration support to provide an even higher level of abstraction.
3. Aspect-oriented programming relieves business components of generic cross-cutting concerns by modularizing them into reusable aspects. In each case, the end result is a system that is easier to test, understand, maintain, and extend.
4. The strategy design pattern used in the Spring framework aligns with the decision of providing choice at every level with the class AbstractVersionStrategy that allows for selecting the version strategies between calculating Hex MD5 hashes from the content of the resource or on a fixed version applied as a request path prefix, e.g. reduced SHA, version name, release date, etc.
5. The singleton design pattern used in the Spring framework aligns with Do Not Repeat Yourself decision mentioned in the documentation where Singleton is the default scope and the IOC container creates exactly one instance of the object per spring IOC container.
6. Spring-integration supports message-driven architectures where inversion of control applies to runtime concerns, such as when certain business logic should run and where the response should be sent. It supports routing and transformation of messages so that different transports and different data formats can be integrated without impacting testability. In other words, the messaging and integration concerns are handled by the framework.

9 REFLECTION

9.1 ARCHITECTURE RECOVERY MECHANISM

9.1.1 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.

9.1.2 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.

9.2 FUTURE IMPROVEMENTS TO THE APPROACH

Text based search for design patterns is tedious and time consuming. We hope to explore advanced methods suitable for automated pattern mining like topic modelling based traceability mechanism.

10 REFERENCES

10.1 DOCUMENTATION

1. https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#overall-architecture
2. https://docs.spring.io/spring-data/jdbc/docs/current/reference/html/#
3. https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/dao.html
4. https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/mvc.html
5. https://docs.spring.io/spring/docs/4.3.23.RELEASE/spring-framework-reference/htmlsingle/
6. https://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch13s07.html
7. https://docs.spring.io/spring/docs/3.0.0.RC2/spring-framework-reference/html/ch12s04.html
8. https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html
9. https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/transaction.html
10. https://docs.spring.io/spring/docs/2.0.x/reference/transaction.html
11. https://docs.spring.io/spring/docs/2.5.x/reference/aop.html
12. https://docs.spring.io/spring-integration/reference/html/overview.html
13. https://docs.spring.io/spring-integration/reference/html/overview.html#overview-components-channel

10.2 SOURCE CODE

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