Leaf Framework v1.0

PREFACE

Leaf Framework is a annotation driven framework that aims to simplify java development and increase the productivity for the model layer issues of the MVC (Model View Controller) based java applications. To simplify development Leaf Framework provides declarative programming environment using annotations. With declarative programming developers describe desired result of the program without explicitly writing codes that need to be carried out to achieve the results. You can still use imperative development with Leaf Framework. You can also use declarative and imperative development together.

Leaf Framework is a lightweight model layer framework that can be easily used in distributed and collocated application architectures. Leaf Framework does not have any dependency to jsp/servlet technology but it can be started from servlet environment using support library.

CONTENTS

What is Leaf Framework

Leaf Framework is a model layer framework whose objective is to provide services to developers for the following issues;

POJO (Plain Old Java Object) Based Programming

Pojo beans are simple java classes that do not have to implement or extend any class to use services provided by framework. Your application classes do not have to extend or implement any framework class to use Leaf Framework services. Required services are injected into your classes by Leaf Framework according to your descriptions. As a result, pojo classes are decoupled from framework classes. By decoupling of applications from frameworks upgrading to a new version or switching to different frameworks becomes easier and less risky.

Pojo also make testing easier, which simplifies and accelerates developments. Your business logic will be easier to understand because of it will not contain framework code.

Program to Interface

Program to interface is an important principle of reusable object-oriented design. The principle is “Program to an interface, not to an implementation”. Using the program to interface principle you can achieve the flexibility in choosing different implementations at run-time as many times via the interface. Leaf Framework provides full support for program to interface by allowing declarative definitions on your application interface classes.

Declarative Programming

Declarative programming is opposite to imperative programming that you explicitly write codes to achieve desired results. With declarative programming, developers describe desired result of the program without explicitly writing codes that need to be carried out to achieve the results.

Leaf Framework provides declarative services for most of provided services such as configuration management, persistency, transaction management, caching etc. You still have imperative development option with Leaf Framework.

Declarative development allows using framework services without explicitly calling them. To achieve desired result of descriptions required code will be injected and called by Leaf Framework. With declarative development you don't have to know which components provide services and write codes to use them. Your applications will not contain framework related codes and will be more clear, easy to understand and will have less dependency to framework.

Strongly Typed Configuration

Leaf Framework is annotation driven framework and uses java annotations for configuration management. You don't have to use any XML file for application configuration. Annotations are strongly typed and thus your application will have the advantage of compile-time checks. Leaf Framework also provides programmatic configuration. You can override configurations defined by annotations using programmatic configuration. You can configure your applications using declaratively or imperatively.

Multi Source Configuration Support

You can divide your application into independent logical modules (sub applications) by using module support of Leaf Framework. You can also specify startup order for each module. When you divide your application into several sub applications and each sub application may contain its own configuration and can be packaged as separate jar files. At run-time Leaf Framework collects and combines all of the configuration information from jar files.

Zero Configuration

To use Leaf Framework you don't have to make any configuration for framework.

Design By Exception

Most of framework services work with predefined values and you only have to specify parameters for exceptional cases.

Design By Contract

You can develop your applications using program to interface approach. With the program to interface approach in Leaf Framework, you have to define declarative definitions on your interfaces classes only. Normally there is a

contract between an interface and it's implementations that provide same methods defined in interface class. With Leaf Framework this contract also valid for declarative definitions (annotations) used in interface classes will be valid for all implementation classes. For example, if you mark an interface class as singleton that means all implementation classes also will be singleton. If you mark a method of interface as cached, same method in all implementation classes will be handled as cached by Leaf Framework according the contract between interface and its implementations.

Bean Factory

Leaf Framework provides strongly typed annotation driven bean factory for program to interface support. Bean factory manage the lifecycles and inject required services for your beans in your applications.

Bean factory controls instantiating, initializing, refreshing of your beans. To configure life cycle (instantiating, initializing and refreshing) of your beans you can use annotations like Bean, Initialize, and Refresh. Bean factory will be managing your beans according to your definitions.

Bean factory also inject required services for your beans such as caching, transaction management etc.

You can also programmatically configure your beans, programmatically configuring of your beans will be detailed in the module section in this tutorial. Leaf Framework bean factory will work according to configurations done with declaratively and programmatically. With programmatic configuration you can override your declarative configurations. Programmatic configuration is more flexible and enables you to change your configuration information at run time whenever you want.

Leaf Framework bean factory has cyclic dependency detection feature. If cyclic dependency happens during creating of your beans, an error message will be given by Leaf Framework.

Declarative and Imperative Persistent Services

Leaf Framework provides imperative and declarative persistence services for the data access issues. It atomically manages database resources and developers do not have to deal with resource management such as how to get

and close a connection etc. This makes your applications more robust and less error prone.

First release of Leaf Framework only provides declarative persistent development for calling stored procedures. To call stored procedures you don't have to write any data access code (implementation code). It is necessary define an interface and set descriptions required to call stored procedure. All required implementation will be performed by Leaf Framework.

Declarative and Imperative Transaction Management

When you use imperative persistent services you can programmatically start a new transaction, all of the database operations will be executed in newly created transaction. Using Leaf Framework you can programmatically start and close transactions. Leaf Framework also provides declarative transaction management by using of transactional annotations in your classes. During the run of your application whenever transactional annotations are detected a new transaction will be created or existing transaction will be used by framework according the transaction definition. Programmatic and declarative transactions can work together.

Leaf Framework has global and local transaction management support. Scope of local transaction is limited with class that starts the transaction. Global transactions can span multiple classes. Scope of global transaction is limited with the call graph of function that starts the global transaction.

Caching

Leaf Framework provides declarative caching of your beans using annotations. You can cache any bean in your application without any restriction. You can also easily specify expiration time for any cached data. You don't have to write any code to cache your application data; it is enough to describe which beans will be cached using annotations.

Extendable and Customizable Framework

Leaf Framework is a framework that uses its own technology. For example it uses its bean factory, module concept etc. to configure and startup itself. You can customize and extend any feature of framework using the same services provided for applications.

Initializing Leaf Framework

Before using Leaf Framework you have to initialize Leaf Framework using support library available in Leaf Framework.

Initializing Leaf Framework for Web Based Applications

For the web based applications, add the code into `web.xml` file given in Setup 1.

Setup1 Web.xml code segment for starting leaf framework For Web Based Applications

```
<listener>

<listener-class> com.leaf.framework.support.web.LeafContextListener

</listener-class>

</listener>
```

Initializing Leaf Framework for Struts Applications

If you are using struts 1.x, add the code into plug-in section of `struts-config.xml` file given in Setup2. You can also use `Leaf Framework ContextListener` to initialize struts based applications.

Setup2 Struts-Config.xml file code segment for starting leaf framework from struts

```
<plug-in className="com.leaf.framework.support.struts.LeafStrutsPlugIn"/>
```

SETTING SCAN JARS FOR MULTI-SOURCE CONFIGURATION SUPPORT

For the web based applications by default all configuration information inside the `/WEB-INF/classes` folder and Leaf Framework jar file will be scanned without any configuration.

If you have multiple sources, you can give the names of jar files contained in `/WEB-INF/lib` folder using `com.leaf.framework.SCAN_JARS` parameter in web.xml file.

Put the following statement into web.xml file, all jar files contained in `com.leaf.framework.SCAN_JARS` parameter will also be scanned.

SETUP3 WEB.XML CODE SEGMENT FOR SETTING THE com.leaf.framework.SCAN_JARS PARAMETER

```
<context-param>

<param-name>com.leaf.framework.SCAN_JARS</param-name>

<param-value>accounts.jar,funds.jar,cards.jar</param-value>

</context-param>
```

Warning: Scanning of your classes performed on class files available in file system. For this reason,scanned classes will not be loaded into JVM to prevent exhausting system resources. `com.leaf.framework.SCAN_JARS` parameter is available to improve startup time of applications. If you don't have multiple source available in WEB-INF/lib folder, you don't have to configure this parameter. For a JEE6 based sample web application scanning of all jar files in class path takes approximately 30 seconds on a normal development pc. Scanning of WEB-INF/classes and jar files specified in `com.leaf.framework.SCAN_JARS` parameter takes lower than 1 second on the samedevelopment pc for the sample application. As a result, If you have multiple sources you should use `com.leaf.framework.SCAN_JARS` parameter to decrease startup time of your applications.

Configuring Your Beans

Leaf Framework provides declarative and imperative configuration of your pojo beans. After you successfully configured your beans you can use `BeanHelper` class to get instances of your pojo beans.

To declaratively configure your beans, you can use `@Bean` annotation. `Bean` annotation has properties to declaratively manage the configuration of your beans. With `Bean` annotation you can specify implementation classes for your interface classes and whether your beans will be singleton or not. You can also specify order and module information for your beans, this information is required to manage life cycles of your beans.

To manage lifecycles of your beans you can use `@Initialize and @Refresh` annotations. `@Initialize` annotation is used to mark initialization method for your beans. Methods marked with `@Initialize` annotation will be called just after the creation of your beans. To periodically refresh your beans you can use `@Refresh` annotation. Methods marked with `@Refresh` annotation will periodically called by Leaf Framework according to properties specified in `@Refresh annotation.`

You can also configure your beans programmatically. Programmatic configuration of your beans has precedence over declarative configuration. You can easily override the declarative definitions during the startup or at runtime with programmatic configuration option. Programmatically configuration of your beans will be detailed in defining modules section of this tutorial.

Declarative Bean Configuration

`Bean` annotation is used to specify implementation classes of your interfaces. You can also specify whether created instances will be singleton and the module information will be containing your beans.

By default all instances will be singleton and will belong to the default Module. You can specify initialization order of your beans that belong to a module. You can also specify initialization order for each module in your application.

TABLE1 ATTRIBUTES OF @BEAN ANNOTATION

Attribute	Type	Description
implementedBy	Class	Class information implementing your interface class. Valid for Bean annotation used in interface classes only.
implementedByRemote	String	Literal class information for implementation class. Valid for Bean annotation used in interface classes only.
providedBy	Class<? extends ImplementationProvider>	If your interface has more than one implementation you can use providedBy to get different implementations. Valid for Bean annotation used in interface classes only.
providedByRemote	String	Literal class information for provided by implementation class. Valid for Bean annotation used in interface classes only.
singleton	Boolean	Default: true
order	Int	Default: 1000 Use order to set initialization order of bean inside a module.
module	Class<? extends Module>	Default: DefaultModule.class

Using @Bean Annotation

You can use `Bean` annotation before the class definition section of your interface classes. If you use `Bean` annotation with concrete classes only the `singleton`, `order` and `module` properties will be regarded.

Using implementedBy and implementedByRemote Property

You can use `implementedBy` and `implementedByRemote` properties of bean annotation to specify implementation classes for interface classes.

SAMPLE1 USING BEAN ANNOTATION TO SPECIFY IMPLEMENTATION CLASS USING IMPLEMENTEDBY PROPERTY FOR THE TESTINTERFACE

```
@Bean(implementedBy=TestImpl.class)

//you can also specify bean as a literal value

//@Bean(implementedByRemote="com.testapp.TestImpl") public interface TestInterface {
public String testMethod(int testValue);

}

//implementation class for TestInterface

public class TestImpl implements TestInterface{ public String testMethod(int testValue) {
return "Method value is " + testValue;

}

}
```

The important points in the Sample 1 are;

`@Bean(implementedBy=TestImpl.class)`

Bean annotation is used to specify implementation class for the `TestInterface`. Implementation class for the `TestInterface` interface is specified by `implementedBy` property of `Bean` annotation. Implementation class for the `TestInterface` is `TestImpl`class.

`@Bean(implementedByRemote="com.testapp.TestImpl")`

You can also use `implementedByRemote` property to specify implementation classes as literal value. In the case of strong dependency is not preferred, you can use `implementedByRemote` property to loosely bind the interface classes to implementation classes.

Using providedBy and providedByRemote Property

If your interface classes have several implementations you can use `providedBy` property to return different implementations.

To get different implementation for your interface classes, first you have to create a class that implements `com.baselib.bean.ImplementationProvider` interface and put your logic into `getImplClass method of created class like below.`

Interface1 Implementation Provider interface definition

```
public interface ImplementationProvider <T> { public Class<? extends T> getImplClass();
}
```

Sample2 Returning different implementations for the TestInterface interface class

```
public class TestImplProvider implements ImplementationProvider <TestInterface>
{
  public Class<? extends TestInterface> getImplClass() {
     if (ConfigHelper.getConfigProperty("appId") == 1)

      // return implementation 1 return TestImpl1.class;
     else if (ConfigHelper.getConfigProperty("appId") == 2)

     // return implementation 2 return TestImpl2.class; return null;
     }

}
```

After creating provider class you can use bean annotation to bind your interfaces to implementation providers. Leaf Framework bean factory will create instance of class returned by the implementation provider specified in `providedBy property.`

SAMPLE3 USING PROVIDEDBY PROPERTY TO GET DIFFERENT IMPLEMENTATIONS OF TESTINTERFACE CLASS

```
@Bean(providedBy = TestImplProvider.class)

// you can also specify provided by information as a literal value

//@Bean(implementedByRemote="com.testapp.TestImplProvider") public interface TestInterface {
public String testMethod(int testValue);

}
```

The important points for the Sample3are;

You can give implementation class as strongly typed or loosely coupled. For the strongly typed configuration you can use `providedBy` property. For the loosely coupled configuration you can use `providedByRemote` property.

Using Singleton and Order Property

By default all beans will be singleton if you didn't assign false to singleton property inside the bean annotation definition. If you specify false for the singleton a new instance will be created and returned for each request.

You can also set order property if you want to order initialization of your beans. For the

SAMPLE4 `TestImplA` will be created before `TestImplB` because of it has lower order value. To initialize your beans you can use `Initialize` annotation. `Initialize` annotation will be detailed later. `Order` property has effect to the beans inside the same module. `Module` concept also will be detailed later.

SAMPLE4 SPECIFYING SINGLETON AND ORDER PROPERTY

```
@Bean(implementedBy= TestImplA.class, singleton = false, order = 10)
public interface TestInterfaceA {
	public String testMethod(int testValue);
}

@Bean(implementedBy= TestImplB.class, singleton = false, order = 20)
public interface TestInterfaceB {
	public String testMethod(int testValue);
}
```

Using Module Property

Module definition is optional and by default all beans will belong to Leaf Framework default module. During the startup of applications all modules will also be started up according their order preferences. For example if you have two modules named as `ModuleA` with order 10 and `ModuleB` with order 20, Leaf Framework first will initialize beans belong to `ModuleA` and then after finishing initialization `ModuleA`, `moduleB` will be created. You can also programatically configure your beans during module started.

Programmatically Configuring Your Beans

You can programmatically configure your beans during application startup. To programmatically configure your beans you need `ModuleType` beans. Beans using `ModuleType` annotation will be started up by Leaf Framework.

Defining Modules Using @ModuleType Annotation

You can define modules to programmatically configure your beans by using `ModuleType` annotation. For each defined module you can also specify initialization order. Modules will be processed according their order value by Leaf Framework during application startup.

Following code segment shows programmatic way of configuring your beans using module support. You can also configure your beans declaratively using `Bean` annotation. After starting each module lifecycle operations for each bean inside the current module will also be started.

`Module` beans have to implement `com.baselib.moduletype.Module` interface.

Interface2 Required Interface Definition for ModuleTypeBeans

```
public interface Module {

public void bindBeans(Binder binder);

}
```

`bindBeans` method of `Module` interface called during the initialization modules during startup for the module type beans.

SAMPLE5 DEFINING MODULE FOR PROGRAMATIC CONFIGURATION

```
@ModuleType(order = ModuleConstants.DATASOURCE_MODULE_ORDER)
public class DataSourceModule implements Module {
        public void bindBeans(Binder binder) {
        // programatically bind implementation..
                binder.bind(JndiService.class).to(DefaultJndiService.class).
                bind(DataSourceService.class).to(DefaultDataSourceService.class);
        }

}
```

For the Sample 5, bindBeans method of DataSourceModule will be called by Leaf Framework during the application startup. Order property of ModuleType annotation specifies startup order of module. Modules will be started according to value in order property.

JndiService interface bind to DefaultJndiService implementation class.

After defining your modules you can use this module information in module property of bean annotation like below,

SAMPLE6 SETTING MODULE INFORMATION FOR BEANS

```
// module specify JndiService interface belongs to DataSourceModule module
@Bean(implementedBy=DefaultJndiService.class, module=DataSourceModule.class)
public interface JndiService {
        public DataSource[] getDatasources();
}
```

During the application startup when the `DataSourceModule` is started up life cycle events (initialization) for the `JndiInterface class will be processed.`

Getting Instances of Beans

BeanHelper

`BeanHelper` class is a factory class that creates and decorates instances of classes according to configuration definitions. You can configure beans declaratively using annotation and programmatically using `moduleType` beans. Programmatic definitions have preference over declarative definitions. Before getting instances of beans using `BeanHelper`, you have to configure your beans.

`BeanHelper` bean has two methods to get instances of your beans. You can pass interface and concrete classes to get instances for your classes.

ClassDefinition1 Bean Helper Abstract Class Definition

```
public abstract class BeanHelper {

public static <T> T getBean(Class<T> clazz) ; public static Object getBean(String beanName);
}
```

You have to use `BeanHelper` to utilize framework services for your pojo beans. `BeanHelper` is not only a factory bean also inject framework services into your beans.

If you are getting instances by passing interface class as parameters to `BeanHelper`, you have to use declarative definitions (annotations used in your classes) only in your interface classes. Declarative definitions used in implementation class will have no any effect for instances created using interface classes.

If you are getting instances by passing concrete classes as parameters to `BeanHelper`, you have to use declarative definitions in your concrete classes. But some properties of declarative definitions will be ignored. For example still you can use `Bean` annotation with concrete classes but some properties of `Bean` annotation such as `implementedBy` ,`providedBy` etc will be ignored while singleton property will be regarded for concrete classes.

For the program to interface approach and to develop more flexible applications you should get instances by passing interface classes.

Using BeanHelper

SAMPLE7 USINGBEANHELPER TOGETINSTANCES

```
// if you pass your interface and concrete class as parameter , you don't have to cast
TestInterface test = BeanHelper.getBean(TestInterface.class);

// if you pass a literal value, you have to cast
TestInterface test = (TestInterface) BeanHelper.getBean("myapp.sample.TestInterface");

// you can pass concrete classes to get instances
TestInterface test = BeanHelper.getBean(TestImplementation.class);
```

`getBean` method of `BeanHelper` is used to create instances for your beans. `BeanHelper` first check programmatic configuration information for requested bean to create instance and to inject required services. If there is no programmatic configuration information for requested bean, `BeanHelper` looks for the declarative configuration information for the requested bean.

You can define your beans as `Transactional` or `Cached` etc. using annotations inyour applications. `BeanHelper` will create interceptors for created instances for each service provided by framework. For example if you define a bean as transactional and cached `BeanHelper` will return a proxy object that contains `Transactional` and `Cached` interceptor beans and target object. Method calls on your target bean will be intercepted by these interceptors.

During the initialization of your beans if there is cyclic dependency between your classes it will be automatically detected and an error message will be given.

Managing The LifeCycle of Beans

To manage lifecycle of your beans you can use `Initialize and Refresh annotations.`

Warning:`Initialize, Refresh` etc annotations are method level annotations you can only use this annotations before method declarations. If you are getting instances from interface definitions these annotations have to be used in methods of interface definitions and annotations used inside the implementation classes will have no effect and will be ignored.

Initializing Beans

@Initialize Annotation

You can use Initialize annotation before methods inside your interface and concrete classes. If you set `loadOnStartup` property to true, these beans will be automatically initialized during application startup. Otherwise initialization will happen just after creating first instance at runtime.

Table2 Attributes of @InitializeAnnotation

Attribute	Type	Required	Description
loadOnStartup	boolean		Default: false Controls whether bean will be initialized during application startup.

If `loadOnStartup` property is true and bean defined as singleton it will be initialized during application startup. `LoadOnStartup` property meaningful only for singleton beans and non singleton beans will not be initialized during startup even if `loadOnStartup` is true. For the non-singleton beans If `loadOnStartup` property is true, initialization will be happen just after creating instance.

Methods using the `@Initialize (com.baselib.bean.lifecycle.Initialize)` annotation must have the following signature:

```
public void <Arbitrary method name>() throws java.lang.Exception
```

Warning: Methods not having the above signature will be ignored and will not be called during the creation of your beans.

Using @Initialize Annotation

Sample8 Setting Initialization Information For Your Beans

```
@Bean(implementedBy= TestImpl.class, singleton = true, order = 10)
public interface TestInterface {
        // Initialize annotation mark the init method for initialization at startup
        @Initialize(loadOnStartup = true)
        public void init();
}
```

Because of `loadOnStartup` property value is true, new instance of `TestInterface` will be created and init method will be called during application startup.

Refreshing Beans

@Refresh Annotation

You can use `Refresh` annotation before methods inside your interface classes. If you set `refreshRateInMs` property, your beans will periodically be refreshed, according to frequency specified in `refreshRateInMs` property.

Table3 Attributes of@RefreshAnnotation

Attribute	Type	Required	Description
refreshRateInMs	Long	Yes	Refresh rate inMilliseconds
refreshInterceptor	Class<? extendsRefreshInterceptor>		Default null Conditionally refreshing yourbeans

Methods using the `Refresh (com.baselib.bean.lifecycle.Refresh) annotation must have the following signature:`

```
public void <Arbitrary method name>() throws java.lang.Exception
```

Warning: Methods not having the above signature will be ignored and will not be called periodically during the life cycle of your beans.

Warning: Beans using the `Refresh annotation must be defined as singleton. Non singleton beans` will not get refresh service and `Refresh annotation will be ignored.`

Using @Refresh Annotation

SAMPLE9 SETTING REFRESH FREQUENCY USING REFRESHRATEINMS PROPERTY

```
@Bean(implementedBy= TestImpl.class, singleton = true, order = 10)
public interface TestInterface {
	@Refresh(refreshRateInMs = 60000)
	public void freshData();
}
```

Because of `refreshRateInMs property set to 60000 miliseconds, freshData method will be called in every 60 seconds.`

Controlling Refresh Frequency with refreshInterceptor Property

By default all methods using `Refresh` annotation will be called according to the frequency specified in `refreshRateInMs` property. If you want to control refresh frequency of refreshed beans you can use `refreshInterceptor` property of `Refresh` annotation. You can assign an interface or concrete class that implements `RefreshInterceptor` interface for the `refreshInterceptor property.`

`RefreshInterceptor` interface class has the signature in Interface 3. If you set `refreshInterceptor` property, Leaf Framework first will be call the `isRefreshNeeded` method of `RefreshInterceptor` interface to decide refreshing is needed. If `isRefreshNeeded` method returns true then bean method using the `Refresh` annotation will be called. If `isRefreshNeeded`

method returns false then method using the `Refresh` annotation will not be called.

INTERFACE3 REFRESH INTERCEPTOR INTERFACE SPECIFICATION

```
public interface RefreshInterceptor {
	public boolean isRefreshNeeded();
}
```

The following example demonstrates how you can control refresh frequency for your beans using `refreshInterceptor` property of `Refresh` annotation. `RefreshInterceptor` property of `Refresh` annotation is set to `TimeCheckInterceptorImpl` class. `TimeCheckInterceptorImpl` class implements `RefreshInterceptor` interface. Whenever a refreshing happens according the frequency specified in refreshRateInMs property, `isRefreshNeeded` method of `TimeCheckInterceptorImpl` class will be called. If `isRefreshNeeded` methods returns true then `freshData method will be called.`

SAMPLE10 SETTING RESFRESH INTERCEPTOR FOR CONDITIONALLY REFRESHING

```
@Bean(implementedBy= TestImpl.class, singleton = true, order = 10)
public interface TestInterface {
	@Refresh(refreshRateInMs = 60000, refreshInterceptor=TimeCheckInterceptorImpl.class)
	public void freshData();
}
```

Sample11 Sample Refresh Interceptor Interface and Implementation

```
public class TimeCheckInterceptorImpl implements RefreshInterceptor {
	public boolean isRefreshNeeded() {
		Calendar cal = new GregorianCalendar ();


		int hour24 = cal.get (Calendar.HOUR_OF_DAY);     // 0..23

			// refreshing will not happen after 18:00
		if (hour24 > 18)
			return false;
		else
			return true;







	}

}
```

Normally `FreshData` method will be called every 60 seconds because of `refreshRateInMs` property value is 60000 ms. For every 60 seconds before calling the freshData method `isRefreshNeeded` method of `TimeCheckInterceptorImpl` classed will be called. After 18:00 pm `isRefreshNeeded` method will return false and `freshData` method will not be called.

Persistent Management

Imperative Persistent Management

Calling Queries

Before calling queries (Select, Insert, Update, Delete) first you have to get instance of `QueryCaller`.

```
QueryCaller query = PersistentManager.getQueryCaller();
```

Setting SQL Statements

To execute query you have to set sql statement using `setSql` method of `QueryCaller` like below.

```
query.setSql("SELECT col1, col2,col3 from testTable");
```

```
query.setSql("update testable set col1 = „testValue‟ where col1 ="oldValue"");
```

Setting Data Source Name

To set used data source use `setDataSourceName` method like below,

```
query.setDataSourceName("jdbc/TEST");
```

Also you can set default data source for all data access beans by using default data source property. Setting default data source will be detailed in configuration parameters section in this tutorial.

Setting Parameters for Executing Queries

You can use `SetParam` method of `QueryCaller` to set input parameters for your queries.

```
                    //setting string parameter caller.setParam("input value");
                    //setting integer parameter caller.setParam(23);
                    //setting double parameter

                    caller.setParam(23.5);
```

Getting Query results

You can get results of your queries using the methods of `ResultSet` interface of Leaf Framework in Interface 4.

INTERFACE4 COM.Leaf.Framework.JDBC.RESULTSET.RESULTSET INTERFACE

```
public interface ResultSet {

        public abstract BigDecimal getBigDecimal(int loc) throws SQLException;

        public abstract BigDecimal getBigDecimal(String name) throws SQLException;

        public abstract double getDouble(int loc) throws SQLException;

        public abstract double getDouble(String name) throws SQLException;

        public abstract int getInt(int loc) throws SQLException ;

        public abstract int getInt(String name) throws SQLException ;

        public abstract long getLong(int loc) throws SQLException;

        public abstract long getLong(String name) throws SQLException;

        public abstract String getLongString(int loc) throws SQLException, IOException;

        public abstract String getLongString(String name) throws SQLException, IOException;

        public abstract String getString(int loc) throws SQLException;

        public abstract String getString(String name) throws SQLException;

        public abstract Timestamp getTimeStamp(int loc) throws SQLException;

        public abstract Timestamp getTimeStamp(String name) throws SQLException;

        public boolean next() throws SQLException;

}
```

SAMPLE12 SAMPLE CODE FOR EXECUTING QUERIES

```
public class PersonelDaoImpl implements PersonelDao {

public static final String SQL_FETCH_PERSONEL = "SELECT ID, NAME, AGE, SALARY" +
                              "FROM PERSONEL WHERE DEPARTMENTID=? AND VALID='Y' ";
        public List<Personel> fetchPersonelList(int departmentId) throws Exception {
                List<Personel> perList = new ArrayList<Personel>();
                QueryCaller cal = (QueryCaller) PersistentManager.getQueryCaller ();
                cal.setDataSourceName (“jdbc/PRODUCTION”);
                cal.setSql (SQL_FETCH_PERSONEL);
                cal.setParam (departmentId);
                ResultSet rs = cal.executeQuery();
                while (rs.next()) {
                        Personel pers = new Personel();
                        pers.setId(rs.getInt("ID")); // or rs.getInt("1");
                        pers.setName(rs.getString ("NAME"));
                        pers.setAge(rs.getInt ("AGE"));
                        pers.setSalary (rs.getDouble("SALARY"));
                        perList.add(pers);
                }

                return perList;
        }

}
```

The important points in the Sample 12 are:

```
QueryCaller query = PersistentManager.getQueryCaller();
```

To execute SQL queries it is necessary to get new instance of `QueryCaller` object. This is done by calling `PersistentManager.getQueryCaller().`

```
cal.setDataSourceName (“jdbc/PRODUCTION”);
```

```
cal.setSql (SQL_FETCH_PERSONEL);
```

Method `setSql` stores the SQL query to execute. This query can contain dynamic parameters. In that case, the parameters are replaced with placeholders (?) in the query.

```
cal.setParam (departmentId);
```

For each dynamic parameter in the query, call method `setParam` to set the parameter value. The order must correspond to the order of appearance of the parameters in the SQL request.

```
ResultSet rs = cal.executeQuery();
```

Method `executeQuery` is used to execute a query that fetches data from database.

```
pers.setId(rs.getInt("ID"));
```

Calling Stored Procedures

Before calling stored procedures first you have to get instance of `ProcedureCaller` interface by calling the `getProcedureCaller` method `PersistentManager` class.

```
ProcedureCaller proCaller = PersistentManager.getProcedureCaller();
```

Setting Stored Procedure Name

To set called stored procedure name call `setProcedureName` method of `ProcedureCaller` like below.

```
proCaller.setProcedureName("nameOfProcedure");

proCaller.setProcedureName("nameOfPackage.nameOfProcedure");
```

To call stored functions use `setFunctionName` method of `ProcedureCaller` like below.

```
proCaller.setFunctionName("nameOfFunction");

proCaller.setFunctionName("nameOfPackage.nameOfFunction");
```

Setting Data Source Name

To set data source name that will be used to execute queries call `setDataSourceName` method like below,

```
prodecureCaller.setDataSourceName(“jdbc/TEST”);
```

You can also configure default data source information for all data access objects using the `ConfigType` beans. `ConfigType beans will be detailed in configuration beans section of this tutorial.`

Setting Parameters for Stored Procedure Calls

You can use methods of `CallableParameterAware` and `OutParamAware` interfaces to set parameters for stored procedures.

Most of the methods of `CallableParameterAware` interface returns `OutParamAware` interface to mark input parameters also as output parameters. Type of input parameters will be same with the type parameters.

INTERFACE5 CALLABLE PARAMETER AWARE INTERFACE FOR PARAMETER SETTING

```
public interface CallableParameterAware {
        public OutParamAware setNull(int type);
        public OutParamAware setParam(Object value);
        public void setParams(List<Object> params);
        public void setParams(Object[] params);
        public OutParamAware setParam( int value);
        public OutParamAware setParam(long value);
        public OutParamAware setParam( float value);
        public OutParamAware setParam(double value);
        public abstract void registerOutParam(final int typeofOutParameter) ;
        public abstract void registerOutParam(final int typeofOutParameter, String typeName)
        public void setParams(ProcedureParamHolder params);
}
```

INTERFACE6 OUTPARAMAWARE INTERFACE FOR OUT PARAMETER SETTING

```
public interface OutParamAware {

public void registerAsOutParam(int typeofOutParameter);

public void registerAsOutParam(int typeofOutParameter, String typeName);

}
```

`CallableParameterAware` and `OutParamAware` interfaces implemented by `ProcedureCaller` interface and can be used to set input and output parameters for stored procedure calls. `OutParamAware` interface is used mark in parameters as inout parameters.

Sample 13 demostrates how you can set input and output parameters for stored procedure calls.

SAMPLE13 SETTING PARAMETERS FOR STORED PROCEDURE CALL

```
//setting a parameter as in and out parameter, you can register in params as out params
// using registerAsOutParam method

proCaller.setParam("input value").registerAsOutParam(Types.VARCHAR);



//setting only in parameter
proCaller.setParam(23);


//setting only out parameter
proCaller.registerOutParam (Types.INTEGER);


//setting null input parameter
proCaller.setNull(Types.VARCHAR);
```

`proCaller.setParam("input value").registerAsOutParam(Types.VARCHAR);`

setParam method set the first parameter as input parameter with String type, registerAsOutParam method also mark first input parameter as out parameter.

`proCaller.setParam(23);`

setParam method set the second parameter as input parameter with int type.

```
proCaller.registerOutParam (Types.INTEGER);
```

`registerOutParam` method set third parameter as out parameter with integer type.

Getting Out Results for Stored Procedures

You can get the results of stored procedure calls using the methods of `ProcedureResultSet` interface of Leaf Framework specified in Interface 7.

INTERFACE7COM.Leaf Framework.JDBC.RESULTSET.PROCEDURERESULTSETINTERFACE

```
public interface ProcedureResultSet {

        public BigDecimal getBigDecimal(int loc) throws SQLException;

        public double getDouble(int loc) throws SQLException;

        public int getInt(int loc) throws SQLException ;

        public long getLong(int loc) throws SQLException;

        public String getLongString(int loc) throws SQLException, IOException;

        public String getString(int loc) throws SQLException;

        public Timestamp getTimeStamp(int loc) throws SQLException;

        public String getClob(int loc) throws SQLException;

        public Map<Integer, Object> getReturnedObjects();

        public Object getObject(int loc);

}
```

Samples from Sample 14 to Sample 18 show how you can call stored procedures and get results programmatically.

SAMPLE14 SAMPLESTOREDFUNCTION

```
    FUNCTION testFunction

    (pValue IN VARCHAR2) RETURN CLOB IS
           wResult CLOB; BEGIN
           wResult := 'Huge Data coming from test function for' || pValue; RETURN wResult;
    END;
```

`TestFunction` function takes only one input parameter and returns `clob` value.

SAMPLE15 SAMPLE STORED PROCEDURE

```
                        PROCEDURE testProcedure (pValue IN VARCHAR2)
                        IS BEGIN
                        --Procedure code

                        END;
```

`TestProcedure` procedure takes only one input and returns no value.

Sample16 Sample Data Access BeanInterface To Call Stored Procedures

```
// bean annotation sets implementation class, note that all clases are singleton by default

@Bean(implementedBy= TestDaoImpl.class)
public interface TestDao {

        public String callTestFunction(String value);

        public void callTestProcedure(String value);

}
```

`Bean` annotation is used to bind `TestDao` interface to `TestDaoImpl` implementation class.

`TestDao` interface class has two methods to call stored procedures.

```
public String callTestFunction(String value);
```

`calltestFunction` method is used call database function, takes only single input parameter for database function and returns string value.

```
public void callTestProcedure(String value);
```

`calltestProcedure` method is used call database procedure, takes only single input parameter for database procedure and returns no value.

Implementation class for the `TestDao` class is specified using `Bean` annotation. Implementation class for the `TestDao` interface is `TestDaoImpl` class and specified by `implementedBy` property of `Bean` annotation.

SAMPLE17 DATAACCESS BEAN IMPLEMENTATION FOR CALLING STORED PROCEDURES

```
public class TestDaoImpl implements TestDao{

public void callTestProcedure(String value){

       //get instance of ProcedureCaller

      ProcedureCaller pro = PersistentManager.getProcedureCaller();

       //set datasource

      pro.setDataSourceName(“jdbc/TEST”);

       //set procedure name to call

      pro.setProcedureName("testProcedure");

       //set input parameter for procedure

      pro.setParam(value);

       //call procedure

      pro.executeUpdate();

    }

public String callTestFunction(String value) {

        //get instance of procedureCaller

       ProcedureCaller pro = PersistentManager.getProcedureCaller();

        //set datasource

       pro.setDataSourceName(“jdbc/TEST”);

        //set procedure name to call

       pro.setFunctionName("testFunction");

        //register function return value as CLOB

       pro.registerOutParam(Types.CLOB);

        //set input parameter for function

       pro.setParam(value);

        //call function

       int count = pro.executeUpdate();

        //get CLOB result value

       return pro.getClob(1);

    }

    }
```

You can get instance of `TestDao using BeanHelper`,

SAMPLE18 CALLING DATA ACCESS CLASS THAT CALLS THE STORED PROCEDURES

```
//get instance for TestDao

TestDao testDao = BeanHelper.getBean(TestDao.class);



//call function

String result1 = testDao.callTestFunction(“input value”);


// call procedure testDao.callTestProcedure(“input value”);
```

DECLARATIVE PERSISTENT MANAGEMENT

For the current release, Leaf Framework only provides declarative persistent development for calling stored procedures. To call stored procedures developers don’t have to write any code. It is enough to define an interface with some annotations. Declarative development services for the queries are planned for the next release of Leaf Framework.

Declarative Persistent Annotations

Using @SQLCaller Interface Annotation

`SQLCaller Interface` annotation has to be used with interface classes and this annotation mark interface class as zero implemented that means you don’t have to implement that interface to call stored procedures. All the implementation logic to achieve desired result of stored procedure calls will be performed by data access broker of Leaf Framework.

TABLE4 ATTRIBUTES OF @SQLCALLER INTERFACE

Attribute	Type	Description
prefix	String	Default: null To specify default prefix for the package calls you can use prefix property. Prefix property will be added to all stored procedure calls inside the described interface class.

Using @Procedure Annotation to Call Stored Procedures

You can use `Procedure` annotation to call stored procedures declaratively without writing implementation code.

TABLE5 ATTRIBUTES OF @PROCEDURE ANNOTATION

Attribute	Type	Description
outParams	@SQLParamOut[]	Default: null You can specify out parameters for stored procedure calls using outParams property. Example: If second parameter in procedure definition is out and return character, you can define out parameter such as; @SQLParamOut(paramIndex = 2, returnType = String.class)
Name	String	Default: null If name property is null then called stored procedure will be exactly same with the method name marked with procedure annotation. You can give name of called stored procedure using name property.
prefix	String	Default: null

Attribute	Type	Description
jndiName	String	Default: null
procedureType	ProcedureType	Default: ProcedureType.PROCEDURE Type of stored procedure.
invocationInterceptor	Class<? extends SQLInterceptor>	Default: null

Defining Input and Output Parameters for Stored Procedures

By default all method arguments will be accepted as input parameters for stored procedure calls. If method arguments also are out parameters you have mark these arguments as out parameters using `SQLParamInOut annotation.`

Setting INOUT Parameters Using SQLParamInOut Annotation

TABLE6 ATTRIBUTES OF@SQLPARAMINOUT ANNOTATION

Attribute	Type	Required	Description
name	String		Stored procedure parameter name. Optional for current release.
returnType	Class	Yes	Return type of parameter.

Setting OUT Parameter Using SQLParamOut Annotation

To get out parameters from stored procedures you can use `SQLParamOut annotation inside the procedure` annotation. For the first version of Leaf Framework, it could automatically assign out parameter of stored procedure call as method return value if stored procedure returns only one out parameter. You can declarative calling of stored procedures that returns multiple out parameters but assignment of out parameters to complex java method return types will be supported in next release.

SAMPLE19 INTERFACE CLASS TO CALL STORED PROCEDURES

```
// SQLCallerInterface mark this interface as zero implemented bean
@SQLCallerInterface
public interface TestDao {

        @Procedure(name="testFunction" , jndiName = "jdbc/TEST",
                  procedureType = ProcedureType.FUNCTION )
        public String callTestFunction(String value);

        @Procedure(name="testProcedure" , jndiName = "jdbc/TEST" )
        public String callTestProcedure(String value);
}
```

Important points for the Sample 19 are;

`SQLCallerInterface`

`SQLCallerInterface` annotation is used to mark interface classes as zero implemented beans that implementation class is not available and must be processed by Leaf Framework. For the current release of Leaf Framework to call stored procedure you don't have to write implementation code. Data access broker layer in Leaf Framework will be handle required implementation to call stored procedures. For the next release of Leaf Framework zero implemented beans would be valid for the callingqueries.

```
@Procedure(name="testFunction" , jndiName = "jdbc/TEST", procedureType =
ProcedureType.FUNCTION )
    public String callTestFunction(String value);
```

`Name` property specifies the name of called stored procedure. `jndiName` specifies the data source that contains the called stored procedure. `ProcedureType` property specifies type of called procedure. `CallTestFunction` has only one parameter and this parameter will be accepted as input parameter for function call. By default all method parameters will be accepted

as input parameters for stored procedure call. You can also specify input parameters as out parameter using `SQLParamInOut` annotation before method arguments.

You can get instance of zero implemented interface class by using `BeanHelper` class such below,

```
TestDao  testDao = BeanHelper.getBean(TestDao.class);

String result = testDao.callTestFunction("testvalue");
```

Calling `getBean` on zero implemented interface classes returns a procedure broker proxy instance that handles the request for stored procedure call. Procedure proxy will call the stored procedure and will assign the return value of stored procedure into `result variable`.

Transaction Management

You can control transactions programmatically or declaratively with Leaf Framework. For the declarative transaction management Leaf Framework provides annotations. You can use `Transactional` annotation to declaratively control your transactions.

Imperative Transaction Management

Programmatically you can only start local transaction that scope of transaction is limited with class in which transaction is started.

Starting Local Transactions Programmatically

`QueryCaller` and `ProcedureCaller` have following methods for local transaction management. You can use `QueryCaller` and `ProcedureCaller` to start new local transactions if there is no already been opened a global transaction. If a global transaction has already been opened a new local transaction will not be started and will be using existing transaction.

Table7 Programmatically Managing Transactions

beginTransaction	Opens a new transaction. This method can be called if a local or global transaction has already been opened and has not been committed or roll backed using the any instance of QueryCaller or ProcedureCaller. This method does not start a local transaction if a global transaction already has been opened.
commitTransaction	Commits the locally opened transaction. This method must be called on a QueryCaller or ProcedureCaller while there is an open transaction. This method will not do anything if there is a globally opened transaction.
rollbackTransaction	Cancel the current open local transaction. This method can only be called on a QueryCaller or ProcedureCaller while has an open transaction. This method will not do anything if there is a globally opened transaction.
reset	Reset the SQL request and parameters previously set on the QueryCaller or ProcedureCaller instance. It can be called.

```
inside the same transaction to reuse the same QueryCaller
instance for making multiple requests.
```

SAMPLE20 PROGRAMMATICALLY MANAGING TRANSACTIONS

```
public void testTransaction() {
        QueryCaller caller = null;
        try {
                caller = PersistenManager.getQueryCaller();
                caller.beginTransaction();
                caller.setQuery("insert into test_table values (?,?)");
                caller.addParam("test key1");
                caller.addParam("test value1"); caller.executeUpdate();

                // reset previous request and parameters for next call
                caller.reset();

                caller.setQuery("insert into test_table values (?,?)");
                caller.addParam("test key2");
                caller.addParam("test value2"); caller.executeUpdate();
                caller.commitTransaction();
        } catch (Exception e) {
                caller.rollbackTransaction();
        }
}
```

The transaction must always be called in a `try/catch block. If no error occurs, the transaction must be committed at the end of the try` block. If an error happens, the transaction must be rolled back in the `catch block.`

Warning: If you are executing several queries using the same instance of `QueryCaller` *or* `ProcedureCaller`*, it is necessary to call there* `set` *method to reinitialize the SQL query and parameters.*

Warning: If a transaction is opened, it is your responsibility to commit or rollback the transaction. Otherwise, resources may be used indefinitely or errors may occur. To ensure to always commit or rollback a transaction, you must always open the transaction in a `try/catch` *block; commit the transaction at the end of the* `try` *block, and rollback the transaction in the* `catch` *block in the case of error.*

Declarative Transaction Management

Starting Global and Local Transactions

You can start global or local transactions with using @Transactional annotation.

Using @Transactional Annotation to Manage Transactions

Table8 Attributes ofTransactionalAnnotation

Attribute	Type	Required	Description
requiresNew	boolean		Default: false If value is true means always a new transaction will be started. If value is false that means if there is a global transaction already has been started, a new transaction will not be created and global transaction will be used. If there already has not been started a global transaction, a new transaction will be created and will be global transaction.

All transactions created with transactional annotation will be global transaction inside the scope of used method. You can use `Transactional annotation before class and method`

declarations. If you are using `Transactional` annotation before class declarations, all methods in these classes will be transactional.

If a method marked as transactional using `Transactional` annotation, transaction will be automatically started just before method call and will be finished just after returned from called method.

For the Sample 21 `insertPersonel` method marked as transactional using `Transactional` annotation. A transaction will be started on call of `insertPersonel` method and will be finished just after returned from `insertPersonel` method. All operations performed inside the `insertPersonel` method will be using same transaction if none of them are marked with `Transactional` annotation with `requiresNew` property value is true.

SAMPLE21 USING TRANSACTIONAL ANNOTATION WITH THE METHODS OF INTERFACE CLASSES

```
public interface PersonelService {

	// transactional annotation mark insertPersonel method as transactional
	@Transactional
	public void insertPersonel(PersonelContext context);

}
```

In Sample 21 `Transactional` annotation marks the `insertPersonel` method as transactional. `InsertPersonel` methods in all classes that implements `PersonelService` interface will be handled as transactional.

SAMPLE22 PERSONEL SERVICE IMPLEMENTATION THAT IMPLEMENTS PERSONEL SERVICE INTERFACE

```
public class PersonelServiceImpl implements PersonelService {
	public void interpersonal(PersonelContext context) {
		personelDao.insert(context);
		adresDao.insert(context);
		contactDao.insert(context);
		adminDao.inform(context);
	}
}
```

For the Sample 22 if we mark `inform` method of `AdminDao` class with `Transactional` annotation with `requiresNew` property is true, inform method will be processed in a separate transaction. Actually, it does not matter wherever the inform method is used , it will be processed in a separate transaction because its `requiresNew` property value is true. `RequiresNew` property always causes to open a new local transaction.

SAMPLE23 USING TRANSACTION ANNOTATION WITH REQUIRES NEW PROPERTY

```
public interface AdminDao {

// this method is transactional and opens a new transaction
// even if a global transaction already has been opened.

        @Transactional(requiresNew = true)
        public void inform();

}
```

`Transactional` annotation with `requiresNew` property is true always open a new transaction even if a global transaction already has been opened.

Warning: If you are using declarative transaction management, scope of a transaction is limited with method and its call graph. For this reason, methods with the same level will create separate transactions. If you want to call same level of methods in a class in a single transaction you have to create a upper level method that call same level of sub methods.

Caching

You can declaratively cache your application data using`Cache`annotation. You can also specify expiration time for each cached data. There is a background process that will automatically removes expired cached data from cache.

Using @Cache Annotation To Cache Data

Table9 Attributes of @Cache Annotation

Attribute	Type	Required	Description
expirationTimeInHours	long		Default: 0 You can specify expiration hours for the cached items using expirationTimeInHours property
expirationTimeInMinutes	long		Default: 0 You can specify expiration minutes for the cached items using expirationTimeInMinutes property
expirationTimeInSecs	long		Default: 0 You can specify expiration seconds for the cached items using expirationTimeInSecs property

`Cache` annotation has three properties to specify expiration time for the cached data. By default all properties is 0 that means cached data will be kept in cache system without any time restriction. If you want to specify any time restriction for cached data you can use any property of cached annotation. You can use `cache` annotation at method level or class level. If you mark interface class as cached that means all methods are also cached with time restriction specified by class level cache annotation. Method level cache definition has precedence over class level cache annotation.

Warning: If you are using interface class in `BeanHelper` to get implementation class, you have to use `cache` annotation only with interface classes. `Cache` annotations used in implementation classes will have no effect.

SAMPLE24 DEFINING CACHE ANNOTATION

```
// By default all method results will be stay in cache for 10 minutes
@Cache(expirationTimeInMinutes = 10)
public interface Test {

        // result of below method will be stay in 10 min in cache because of
        // class level cache declaration
        public String cachedMethodForTenMinutes();

        // after first call of testMethod result data will be cached and data
        // will be returned from cache for the all subsequent calls

        @Cache
        public String testForCacheForever();

        // result data will be stay in cache for every 1 hours 10 minutes and 5 seconds
        @Cache(expirationTimeInHours = 1, expirationTimeInMinutes = 10,
                expirationTimeInSecs =5)
        public String testForTimeRestriction();

}
```

You can also use `Cache` annotation for methods that takes parameters. For the methods that takes parameters data will be cached for each unique parameter values. For example for the following class declaration, result for the each personnel will be cached for 10 minutes.

SAMPLE25 CACHING METHODS THAT TAKES PARAMETERS

```
public interface PersonelManager {

// result of below method will be stay in 10 min in cache
//because of class level cache declaration

        @Cache(expirationTimeInMinutes = 10)
        public Personel getPersonel (int personelId);

}
```

```
// get instance of personel manager

PersonelManager perMan = BeanHelper.getBean(PersonelManager.class);
```

```
// it will get personel information from database system
perMan.getPersonel (100);


// it will get personel information from cache system
perMan.getPersonel (100);


// it will get personel information from cache system
perMan.getPersonel (100);


// it will get personel information from database system
perMan.getPersonel (195);

// it will get personel information from cache system
perMan.getPersonel (195);
```

SCHEDULING JOBS USING WATCHDOGS BEANS

DEFINING WATCHDOGS

You can specify scheduled jobs for batch processing using `WatchDog Type` annotation.

Using @WatchDog Type Annotation

TABLE10 ATTRIBUTES OF @WATCHDOG TYPE ANNOTATION

Attribute	Type	Required	Description
name	String	Yes	Name of scheduled job. Name is required for the logging purposes.
refreshRateInMs	long	Yes	Run frequency for scheduled job
Order	long		Run order of job

To define watchdogs your classes has to implement `WatchDog` interface and has to use

`WatchDog Type` annotation to define watchdog properties.

Your scheduled jobs have to implement `Watchdog` interface to be periodically called by Leaf Framework.

INTERFACE8 WATCHDOG INTERFACE DEFINITION FOR PERIODICALLY RUNNING JOBS

S

```
// your watchdogs has to implement WatchDog interface
public interface WatchDog {
        public void refresh();
}
```

Sample26 Defining WatchDogs on Your BEans Using WatchDog Type Annotation

```
//define name and run frequency for watchdog.. run frequency is 30 secs

@WatchDogType(name = "Notify Manager Watch Dog", refreshRateInMs = 30000 )

//You can use implementation class using bean annotation

@Bean(implementedBy=  DefaultNotifyWatchDog.class)
public interface NotifyWatchDog extends WatchDog {

}
```

For the Sample 26 `implementedBy` property of `Bean` annotation specifies the implementation class containing the method that will be periodically called.

Sample27 WatchDog Implementation Bean that Will be Periodically Called

```
public class DefaultNotifyWatchDog implements NotifyWatchDog
{

        // notify method will be called every 30 secs

        public void notify() {
                // implement notify code here..

        }



}
```

Configuring Applications using Configuration Beans

You can specify configuration parameters using `@ConfigType`annotation. You can configure your beans and specify configuration parameters for your applications.

Defining Configuration Beans

Using @ConfigType Annotation

Table11 Attributes of @ConfigType Annotation

Attribute	Type	Required	Description
order	long		Execution order for configuration bean

To configure your application you can use `ConfigType` annotation in your beans. Configuration beans have to implement `com.baselib.config.Config` interface and must have `ConfigType` annotation. During the application startup all configuration beans will be called according their order preference. You can also specify order for each configuration bean in your application. In your configuration beans you can fetch configuration data from file or database based system.

Interface9 Config Interface Definition for Configuring Application Parameters

```
public interface Config {

	public void config(ConfigManager configManager);

}
```

```
	// ConfigType annotation marks this class as configuration bean class

	@ConfigType( order = 20 )
	public class AccountConfig implements Config  {

		public void config(ConfigManager configManager) {

			// visible accounts types
			configManager.setConfigProperty("visibleAccountTypes","C,P,S");

			// set property key and value
			configManager.setConfigProperty("key","value");

		}

	}
```

`PersonelConfig` will be called first because its order property value is lower than `AccountConfig` configuration beans.

```
// ConfigType annotation marks this class as configuration bean class
@ConfigType( order = 10 )
public class PersonelConfig implements Config  {

	//config method will automatically called by seahorse framework during the startup
	public void config(ConfigManager configManager) {

		// set default jndi name for all jdbc operations..
		configManager.setDefaultJndiName("jdbc/DEFAULT");

		// set property key and value
		configManager.setConfigProperty("testKey", "testValue");

	}

}
```

Accessing Configuration Parameters

You can use`ConfigHelper` class to access configured parameter values.

```
// this will print Database to console..

System.out.println("Log Destination is " +
ConfigHelper.getConfigManager().getConfigProperty("logDestination"));
```

You can also use `BeanHelper` to get configuration parameter values such below,

```
BeanHelper.getBean(ConfigManager.class).getConfigProperty("logDestination"));
```

Putting All Together

//TODO : a complete example for Leaf Framework

APPENDIX

INSTALLING Leaf Framework

To use Leaf Framework first you should use below depency packages in your pom.xml.

```
<dependency>

	<groupId>commons-beanutils</groupId>
	<artifactId>commons-beanutils</artifactId>
</dependency>
<dependency>
	<groupId>org.apache.struts</groupId>
	<artifactId>struts-core</artifactId>
</dependency>
<dependency>
	<groupId>javax.servlet</groupId>
	<artifactId>servlet-api</artifactId>
	<scope>provided</scope>
</dependency>
<dependency>
	<groupId>org.scannotation</groupId>
	<artifactId>scannotation</artifactId>
</dependency>
```