[Development Best Practices](https://wiki.eisgroup.com/display/CRC/EIS+Development+Best+Practices)

# Purpose

This document covers the development best practices specific for EIS Suite.

# EIS Extension and Configuration

## **General tips**

1. Avoid coding whenever you can. Possibly what you are going to implement already exists or possibly it could be achieved using configuration.
2. Prefer implementation in BASE source code. The things that you are going to do could or should be made in BASE? If so, open related EISISSUE with proposal or request.
3. If you have to fix some bug, check if it is reproducible in BASE and submit ticket to BASE if so.
4. If it is something simple or you already know what to do, use Private Source approach.
5. Are you in hurry and could do it by yourself in our customization code? Well, do it by yourself, with related "TODO remove since..." in source left link to JIRA ticket and monitor it.
6. Pay attention on base annotations like Internal, CanInvoke and CanExtend. See PublicAPI for more details.
7. Possibly you found that class is marked as Internal, but it's interface is CanInvoke - this is subject to use delegation.
8. Avoid copy-paste. If you need reuse some logic, think if you can move common part to static method or use delegation/extension. Maybe you could refactor existing logic to be more configurable? Maybe this refactoring could be moved to BASE?
9. Prefer delegation over extension. You are going to extend existing interface and add there some method? Maybe you could introduce a new interface and use delegation to existing one if you need some logic to reuse.
10. If you are going to override UI, pay attention if resource is Overridable, Invokable or Extendable. Don't forget about [EIS UI Extensions Process Guidelines](https://wiki.eisgroup.com/display/CRC/UI+Extensions+Process+Guidelines) and related ef:override tag.
11. Finally read this page [Suite Extension and Customization Options](https://wiki.eisgroup.com/display/CRC/Suite+Extension+and+Customization+Options) - it contains a lot of useful info.
12. In order to avoid copy-paste of base-spring configs use ability of bean definition customization.

## **Individual performance**

### Choose the best time for updating code

1. Update your code before starting to work on the user story.
2. Do not update if your task doesn't need merging others code even if your task depends on it.
3. Update code once you have your task done, and verify if your code works with the latest code.
4. Commit your changes all at once as early as possible.

### Avoid restarting app too frequently

1. Empty Tomcat cache, disable ipb rule cache before staring server
2. Show sql if you want to debug loading data from database
3. Show rule logs if you have problem with rules
4. Use tricks for reloading resource files such as property file and xhtml file
5. Edit java file in debug mode if possible
6. Use "hot deploy" techniques for java files

### Avoid re-deploying your database

1. Mock data in database before you write liquibase changeset (You can use an advantage of transaction capability of some db tools when tested; just revert the changes you manually set)
2. Increase liquibase changeset id to run your newly added changeset
3. Analyze data in the database, not just re-dump a new one.

## **Factory Pattern and Custom CommandFactory Implementation**

### **Task Definition**

To introduce a couple easy ways to instantiate not singleton business service from a singleton customer service.

For demonstration here is a couple simple classes:

**Interface for a business service**

|  |
| --- |
| **public** **interface** AutoService {      String doSomething();  } |

**Implementation**

|  |
| --- |
| **public** **class** AutoServiceAZ **implements** AutoService {      @Override  **public** String doSomething() {  **return** "AZ";      }  } |

**Some methods of the customer service**

|  |
| --- |
| **public** String doSomething(){      AutoService autoService = commonFactory.create();  **return** autoService.doSomething();  }    **public** AutoService createObject(){  **return** commonFactory.create();  } |

### The solution

|  |
| --- |
| **public** **interface** CommonFactory<T>{      T create();  }  And it's implementation    **public** **class** CommonFactoryImpl<T> **implements** CommonFactory<T> {  **private** ObjectFactory<T> factory;        @Override  **public** T create() {  **return** factory.getObject();      }    **public** **void** setFactory(ObjectFactory<T> factory) {  **this**.factory = factory;      }  } |

The main idea of introduced class to isolate business code from Spring related classes like ObjectFactory. We do not want to couple business logic and Spring Framework implementation.

Then our customer service can be implemented like this:

|  |
| --- |
| **public** **class** CommonFactorySingletonService {      /\*\*       \* inject common factory       \*/  **private** CommonFactory<AutoService> commonFactory;    **public** **void** setCommonFactory(CommonFactory<AutoService> commonFactory) {  **this**.commonFactory = commonFactory;      }  ... |

**Junit test**

|  |
| --- |
| @Autowired  **private** SingletonService singleton;    @Test  **public** **void** testFactory(){      AutoService azService1 = singleton.createObject();      AutoService azService2 = singleton.createObject();        assertNotSame(azService1, azService2);      assertEquals("AZ", azService1.doSomething());  } |

**Spring configuration**

|  |
| --- |
| <!-- There is we introduce our business service -->  <bean id="autoAZPrototype" **class**="com.exigen.abc.utils.CommonFactory.AutoServiceAZ" scope="prototype" />    <!-- Instantiate customer service -->  <bean id="singleton" **class**="com.exigen.abc.utils.CommonFactory. CommonFactorySingletonService " scope="singleton" >      <property name="commonFactory"   ref="autoCommonFactory" />  </bean>    <!-- Instantiate factory **for** business service -->  <bean id="autoCommonFactory" **class**="com.exigen.abc.utils.CommonFactory.impl.CommonFactoryImpl">      <property name="factory">          <bean **class**="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean">              <property name="targetBeanName">                  <idref local="autoAZPrototype"/>              </property>          </bean>      </property>  </bean> |

**Pros:**

* Medium complexity configuration Spring framework, 2 interfaces and 3 classes were used

### Lookup

**Cusoter class' implementation**

|  |
| --- |
| **public** **abstract** **class** LookupSingletonService {  **public** String doSomething(){          AutoService autoService = createObject ();  **return** autoService.createObject ();      }    **public** **abstract** AutoService createObject ();  } |

**Spring configuration**

|  |
| --- |
| <bean id="singletonLookup" **class**="com.exigen.abc.utils.CommonFactory.LookupSingletonService" scope="singleton" >      <lookup-method name=" createObject" bean="autoAZPrototype" />  </bean> |

**Pros**:

* Very simple Spring configuration. One interface and two classes were used.

**Cons**:

* The class LookupSingletonService is abstract and can be instantiated by Spring with CGLIB library, but it can have some advantage during testing without Spring.

### Lookup service factory

Implementation of customer class the same as for CommonFactory:

|  |
| --- |
| **public** **class** LocatorSingletonService {  **private** CommonFactory<AutoService> serviceFactory;    **public** **void** setServiceFactory(CommonFactory<AutoService> serviceFactory) {  **this**.serviceFactory = serviceFactory;      }  } |

The difference is only in Spring configuration:

|  |
| --- |
| <bean id="singletonLocator" **class**="com.exigen.abc.utils.CommonFactory.LocatorSingletonService" scope="singleton" >      <property name="serviceFactory" ref="autoServiceFactory" />  </bean>    <bean id="autoServiceFactory"  **class**="org.springframework.beans.factory.config.ServiceLocatorFactoryBean">      <property name="serviceLocatorInterface" value="com.exigen.abc.utils.CommonFactory.api.CommonFactory" />      <property name="serviceMappings">          <props>              <prop key="">autoAZPrototype</prop>          </props>      </property>  </bean> |

**Pros**:

* ServiceLocatorFactoryBean allows to configure injecting different implementations of business service over serviceMappings (out of scope)
* Two interfaces and two classes were used.

**Cons**:

* The most complex Spring configuration.

 You can use any way from above, all of them don't couple with Spring in code and can be tested with and without application context.

## **Recommendations for SimpleDateFormat usage**

Many customer projects heavily use SimpleDateFormat. For example, in one project we create 8200 instances on new business and about 12600 on endorsement.   
As we know SimpleDateFormat is not thread-safe class and we have to use different ways to use it correctly in our web application:

1. Instantiate each time before usage
2. Synchronization on an instance
3. Usage ThreadLocal variable

In the table below you can see time in millisecond consumed by 1 million iterations:

* Formatter – create SimpleDateFormat and call format method
* ThreadLocal formatter – usage ThreadLocal approach to store instances.
* Cashed formatter – usage one instance SimpleDateFormat without synchronization
* Object creation – create one instance of SimpleDateFormat on each iteration.
* Synchronized formatter - usage one instance SimpleDateFormat inside a synchronization block

| **Test** | **Average timing, ms** |
| --- | --- |
| Formatter | 5025 |
| ThreadLocal formatter | 1032 |
| Cashed formatter | 1053 |
| Object creation | 153 |
| Synchronized formatter | 1019 |

All tests were run in one thread and as result we do not see significant overhead on synchronization.

### The Solution

Use ThreadLocal approach. Below you can see example of suggested refactoring approach

**Before refactoring**

|  |
| --- |
| **private** String formatDate(Date effectiveDate) {  **final** String DATE\_FORMAT = "MM/dd/yyyy";      String formattedDate = **null**;      SimpleDateFormat sdf = **new** SimpleDateFormat(DATE\_FORMAT);  **if** (effectiveDate != **null**) {          formattedDate = sdf.format(effectiveDate.getTime());      }  } |

**After refactoring**

|  |
| --- |
| **private** **static** **final** String DATE\_FORMAT = "MM/dd/yyyy";  **private** **static** finalThreadLocal<SimpleDateFormat> dateFormatter = **new** ThreadLocal<SimpleDateFormat>() {      @Override  **protected** SimpleDateFormat initialValue() {  **return** **new** SimpleDateFormat(DATE\_FORMAT);      }  };    **private** String formatDate(Date effectveDate) {  **return** effectiveDate != **null** ? dateFormatter.get().format(effectiveDate.getTime()) : **null**;  } |

## **Recommendations for PF component extension on a Project**

### Case 1: The project implementation doesn't require to add any new attributes to the existing Preconfig component.

The extension of component/building block is not needed, its name remains as Preconfig.

XML configuration for the component is edited without changing XML file name.

### Case 2: The project implementation requires new project-specific attributes addition to existing Preconfig component.

 Component extension should be done:

|  |
| --- |
| <ProjectName><SomeName>Component extending Preconfig<SomeName> |

Component should be added to the project, package should follow Preconfig Component package name, but with Project Name instead ipb in it.

 Example:

* com.exigen.abc.policy.components - for ABC project.

|  |
| --- |
| <ProjectName><SomeName>Entity extending <SomeName>Entity |

should be specified in preconfig-<some-name>-component-beans.xml.

The <ProjectName>-<some-name>-component-beans.xml should be added with Project Component configuration:

|  |
| --- |
| <bean id="<ProjectName>YourNewComponent\_1\_0"      parent="PreconfigYourNewComponent\_1\_0"  **class**="com.exigen.<ProjectName>.policy.components.<ProjectName>YourNewComponent">      <constructor-arg index="0">      <value>com.exigen.<ProjectName>.policy.domain.<ProjectName>YourNewEntity</value>  </constructor-arg>  ...  </bean> |

Preconfig component xml configuration is replaced with project specific component  xml configuration, keeping component reference name in it.

### Сase 3: The project implementation requires new project-specific component.

New component should be created following the [[ProductFactory][Component] Component Creation Steps](https://wiki.eisgroup.com/display/CRC/%5BProductFactory%5D%5BComponent%5D+Component+Creation+Steps) and the [EIS Programming Naming and Coding Conventions](https://wiki.eisgroup.com/display/CRC/EIS+Programming+Naming+and+Coding+Conventions).

Project name prefix is to be used in Class names only if Class overrides/extends some base Class, totally new Class doesn't need Project prefix prefix in it's name.