Eclipse Scout
Technical Guide

Scout Team

Version 7.1

Table of Contents

Introduction

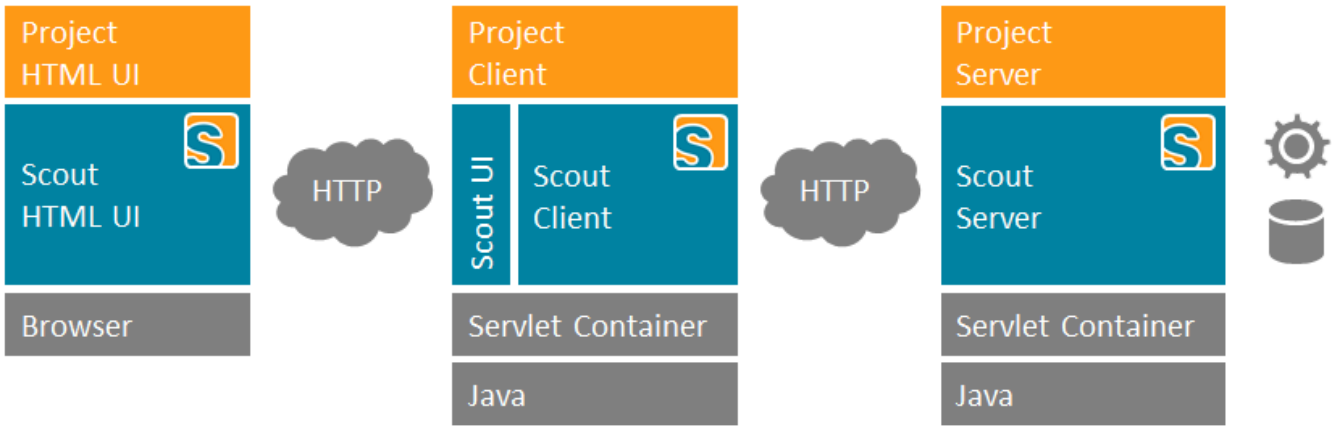
This technical guide documents the Scout architecture and describes important concepts used in Scout.

This document is not complete. Contributions are welcome!
If you like to help, please create a pull request. Thanks!

Repository:
https://github.com/BSI-Business-Systems-Integration-AG/org.eclipse.scout.docs

Chapter 1. Overview

Scout is a framework for creating modern business applications. Such applications are typically separated into multiple tiers where each tier is responsible for a specific part of the application like presenting information to the user or processing business logic and persisting data. Scout solves these requirements by providing a separation of such tiers out of the box.



A typical Scout application consists of the following parts:

- A server layer responsible for persisting data on a database and possibly providing and consuming webservices. The scout server layer provides utilities to simplify the most common tasks.
- A client layer responsible for handling the java UI code. It consists of a model represented by plain java classes as well as services and utilities to implement behaviour associeted with client code. The scout client layer provides utilities to simplify the most common tasks. For simplicity, the client model is processed in a single threaded way to avoid synchronization. Callbacks, e.g. for validating a field or calling services when opening a form run inside a model job.
- A UI layer responsible for rendering the client model in the browser. Since the scout UI layer already provides javascript/html/css code for many common UIs, the project specific code in this layer is typically quite small. Examples are specific css styling or a new custom input field for special purposes.

Server and client both run in a servlet container, such as Apache Tomcat. They are usually deployed as separate war files in order to be able to scale them differently. However, it is also possible to create a single war file.

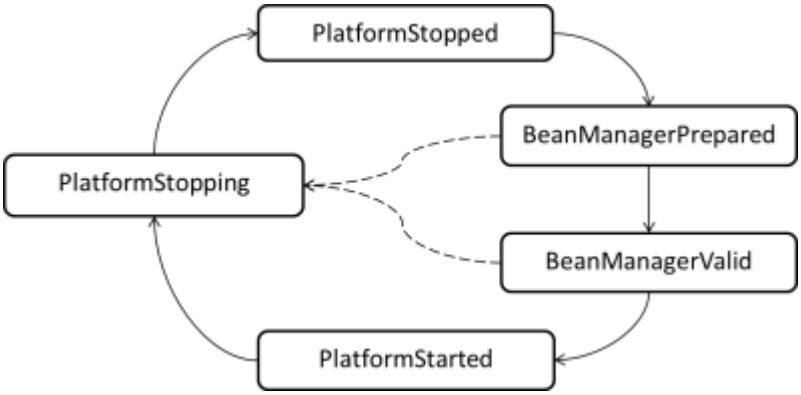
Chapter 2. Scout Platform

Scout contains a platform which provides basic functionality required by many software applications. The following list gives some examples for which tasks the platform is responsible for:

- Application Lifecycle Management
- Object Instance Management (Bean Management)
- Configuration Management
- Application Inventory

2.1. Application Lifecycle

The lifecycle of a Scout application is controlled by implementations of `org.eclipse.scout.rt.platform.IPlatform`. This interface contains methods to start and stop the application and to retrieve the Bean Manager associated with this application. The class `org.eclipse.scout.rt.platform.Platform` provides access to the current platform instance. On first access the platform is automatically created and started.



During its startup, the platform transitions through several states. Depending on the state of the platform some components may already be initialized and ready to use while others are not available yet.

See enum `org.eclipse.scout.rt.platform.IPlatform.State` for a description of each state and what may be used in a certain state.

2.1.1. Platform Listener

To participate in the application startup or shutdown a platform listener can be created. For this a class implementing `org.eclipse.scout.rt.platform.IPlatformListener` must be created. The listener is automatically a bean and must therefore not be registered anywhere. See Section 2.3 to learn more about bean management in Scout and how the listener becomes a bean. As soon as the state of the platform changes the listener will be notified.

Listing 1. A listener that will do some work as soon as the platform has been started.

```
public class MyListener implements IPlatformListener {
  @Override
  public void stateChanged(PlatformEvent event) {
    if (event.getState() == State.PlatformStarted) {
      // do some work as soon as the platform has been started completely
    }
  }
}
```

As platform listeners may run as part of the startup or shutdown not the full Scout platform may be available. Depending on the state some tasks cannot be performed or some platform models are not available yet!

2.2. Class Inventory

Scout applications use an inventory containing the classes available together with some meta data about them. This allows finding classes available on the classpath by certain criteria:

- All subclasses of a certain base class (also known as type hierarchy)
- All classes having a specific annotation.

This class inventory can be accessed as described in listing Listing 2.

Listing 2. Access the Scout class inventory.

```
IClassInventory classInventory = ClassInventory.get();

// get all classes below IService
Set<IClassInfo> services = classInventory.getAllKnownSubClasses(IService.class);

// get all classes having a Bean annotation (directly on them self).
Set<IClassInfo> classesHavingBeanAnnot = classInventory.getKnownAnnotatedTypes(Bean
.class);
```

2.2.1. scout.xml

In its static initializer, the `ClassInventory` collects classes in projects containing a resource called META-INF/scout.xml.

Scanning all classes would be unnecessarily slow and consume too much memory. The file `scout.xml` is just an empty xml file. Scout itself also includes `scout.xml` files in all its projects.

The format XML was chosen to allow adding exclusions in large projects, but this feature is not implemented right now.

It is recommended to add an emtpy `scout.xml` file into the META-INF folder of your projects, such that the classes are available in the 'ClassInventory'.

Scout uses Jandex [1: https://github.com/wildfly/jandex] to build the class inventory. The meta data to find classes can be pre-computed during build time into an index file describing the contents of the jar file. See the jandex project for details.

2.3. Bean Manager

The Scout bean manager is a dynamic registry for beans. Beans are normal Java classes usually having some meta data describing the characteristics of the class.

The bean manager can be changed at any time. This means beans can be registered or unregistered while the application is running. For this the bean manager contains methods to register and unregister beans. Furthermore methods to retrieve beans are provided.

The next sections describe how beans are registered, the different meta data of beans, how instances are created, how they can be retrieved and finally how the bean decoration works.

2.3.1. Bean registration

Usually beans are registered during application startup. The application startup can be intercepted using platform listeners as described in Section 2.1.1.

Listing 3. A listener that registers a bean (direct class or with meta data).

```
public class RegisterBeansListener implements IPlatformListener {
  @Override
  public void stateChanged(PlatformEvent event) {
    if (event.getState() == State.BeanManagerPrepared) {
      // register the class directly
      BEANS.getBeanManager().registerClass(BeanSingletonClass.class);

      // Or register with meta information
      BeanMetaData beanData = new BeanMetaData(BeanClass.class).withApplicationScoped
(true);
      BEANS.getBeanManager().registerBean(beanData);
    }
  }
}
```

There is also a predefined bean registration built into the Scout runtime. This automatically registers all classes having an `org.eclipse.scout.rt.platform.@Bean` annotation. Therefore it is usually sufficient to only annotate a class with `@Bean` to have it available in the bean manager as shown in listing Listing 4.

Listing 4. An normal bean

```
@Bean
public class BeanClass {
}
```

As the `@Bean` annotation is an `java.lang.annotation.@Inherited` annotation, this automatically registers all child classes too. This means that also interfaces may be `@Bean` annotated making all implementations automatically available in the bean manager! Furthermore other annotations may be `@Bean` annotated making all classes holding these annotations automatically to beans as well.

If you inherit a `@Bean` annotation from one of you super types but don't want to be automatically registered into the bean manger you can use the `org.eclipse.scout.rt.platform.@IgnoreBean` annotation. Those classes will then be skipped.

@TunnelToServer

There is a built in annotation `org.eclipse.scout.rt.shared.@TunnelToServer`. Interfaces marked with this annotation are called on the server. The server itself ignores this annotation.

To achieve this a bean is registered on client side for each of those interfaces. Because the platform cannot directly create an instance for these beans a specific producer is registered which creates a proxy that delegates the call to the server. Please note that this annotation is not inherited. Therefore if an interface extends a tunnel-to-server interface and the new methods of this interface should be called on the server as well the new child interface has to repeat the annotation!

The proxy is created only once for a specific interface bean.

2.3.2. Bean Scopes

The most important meta data of a bean is the scope. It describes how many instances of a bean can exist in a single application. There are two different possibilities:

- Unlimited instances: Each bean retrieval results in a new instance of the bean. This is the default.
- Only one instance: There can only be one instance by Scout platform. From an application point of view this can be seen as singleton. The instance is created on first use and each subsequent retrieval of the bean results in this same cached instance.

As like all bean meta data this characteristic can be provided in two different ways:

1. With a Java annotation on the bean class as shown in the listing Listing 5.
2. With bean meta data as shown in listing Listing 3.

Listing 5. An application scoped bean using annotations

```
@ApplicationScoped
public class BeanSingletonClass {
}
```

So the Java annotation `org.eclipse.scout.rt.platform.@ApplicationScoped` describes a bean having singleton characteristics.

Also `@ApplicationScoped` is an `@Inherited` annotation. Therefore all child classes automatically inherit this characteristic like with the `@Bean` annotation.

2.3.3. Bean Creation

It is not only possible to influence the number of instances to be created (see Section 2.3.2), but also to create beans eagerly, execute methods after creation (like constructors) or to delegate the bean creation completely. These topics are described in the next sections.

Eager Beans

By default beans are created on each request. An exception are the beans marked to be application scoped (as shown in Section 2.3.2). Those beans are only created on first request (lazy). This means if a bean is never requested while the application is running, there will never be an instance of this class.

But sometimes it is necessary to create beans already at the application startup (eager). This can be done by marking the bean as `org.eclipse.scout.rt.platform.@CreateImmediately`. All classes holding this annotation must also be marked as `@ApplicationScoped`! These beans will then be created as part of the application startup.

Constructors

Beans must have empty constructors so that the bean manager can create instances. But furthermore it is possible to mark methods with the `javax.annotation.@PostConstruct` annotation. Those methods must have no parameters and will be called after instances have been created.

When querying the bean manager for an application scoped bean, it will always return the same instance. However, the constructor of an application scoped bean may run more than once, whereas a method annotated with @PostConstruct in an application scoped been is guaranteed to run exactly once.

2.3.4. Bean Retrieval

To retrieve a bean the class `org.eclipse.scout.rt.platform.BEANS` should be used. This class provides (amongst others) the following methods:

Listing 6. How to get beans.

```
BeanSingletonClass bean = BEANS.get(BeanSingletonClass.class);
BeanClass beanOrNull = BEANS.opt(BeanClass.class);
```

- The `get()` method throws an exception if there is not a single bean result. So if no bean can be found or if multiple equivalent bean candidates are available this method fails!
- The `opt()` method requires a single or no bean result. It fails if multiple equivalent bean candidates are available and returns `null` if no one can be found.
- The `all()` method returns all beans in the correct order. The list may also contain no beans at all.

There are now two more annotations that have an effect on which beans are returned if multiple beans match a certain class. Consider the following example bean hierarchy:

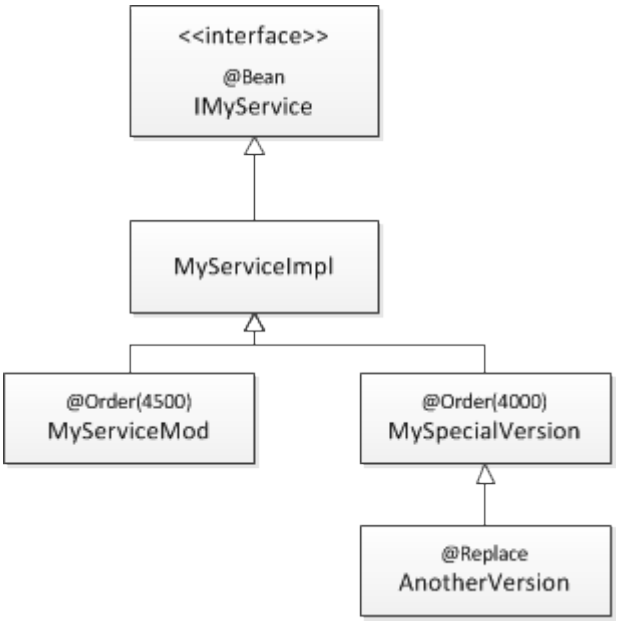


Figure 1. A sample bean hierarchy.

In this situation 4 bean candidates are available: MyServiceImpl, MyServiceMod, MySpecialVersion and AnotherVersion. But which one is returned by `BEANS.get(IMyService.class)`? Or by `BEANS.get(MySpecialVersion.class)`? This can be influenced with the `org.eclipse.scout.rt.platform.@Order` and `org.eclipse.scout.rt.platform.@Replace` annotations. The next sections describe the idea behind these annotations and gives some examples.

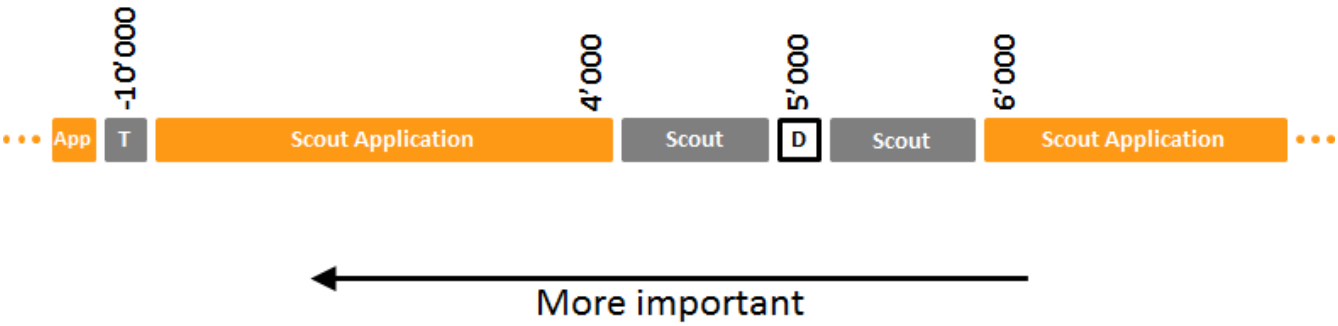
@Order

This annotation works exactly the same as in the Scout user interface where it brings classes into an order. It allows to assign a `double` value to a class. All beans of a certain type are sorted according to this value in ascending order. This means a low order value is equivalent with a low position in a list (come first).

Please note that the `@Order` annotation is not inherited so that each bean must declare its own value where it fits in.

The `@Order` annotation value may be inherited in case it replaces. See the next section for details.

If a bean does not declare an order value, the default of `5000` is used. Scout itself uses orders from `4001` to `5999`. So for user applications the value `4000` and below can be used to declare more important beans. For testing bean mocks the value `-10'000` can be used which then usually comes before each normal Scout or application bean.



@Replace

The `@Replace` annotation can be set to beans having another bean as super class. This means that the original bean (the super class) is no longer available in the Scout bean manager and only the new child class is returned.

If the replacing bean (the child class) has no own `@Order` annotation defined but the replaced bean (the super class) has an `@Order` value, this order is inherited to the child. This is the only special case in which the `@Order` annotation value is inherited!

2.3.5. Examples

The next examples use the bean situation as shown in figure Figure 1. In this situation the bean manager actually contains 3 beans:

1. `AnotherVersion` with `@Order` of 4000. This bean has no own order and would therefore get the default order of 5000. But because it is replacing another bean it inherits its order.
2. `MyServiceMod` with `@Order` of 4500. This bean declares its own order.
3. `MyServiceImpl` with `@Order` of 5000. This bean gets the default order of 5000 because it does not declare an order.

The bean `MySpecialVersion` is not part of the bean manager because it has been replaced by `AnotherVersion`.

- `BEANS.get(IMyService.class)`: Returns `AnotherVersion` instance. The result cannot be an exact match because the requested type is an interface. Therefore of all candidates there is one single candidate with lowest order (comes first).
- `BEANS.get(MyServiceImpl.class)`: Returns `MyServiceImpl` because there is an exact match available.
- `BEANS.get(MySpecialVersion.class)`: Returns `AnotherVersion`. The result cannot be an exact match

because there is no exact bean with this class in the bean manager (`MySpecialVersion` has been replaced). Therefore only `AnotherVersion` remains as candidate in the hierarchy below `MySpecialVersion`.

- `BEANS.get(MyServiceMod.class)`: Returns `MyServiceMod` because there is no other candidate.
- `BEANS.all(IMyService.class)`: Returns a list with all beans sorted by `@Order`. This results in: `AnotherVersion`, `MyServiceMod`, `MyServiceImpl`.

If `MyServiceMod` would have no `@Order` annotation, there would be two bean candidates available with the same default order of 5000: `MyServiceImpl` and `MyServiceMod`. In this case a call to `BEANS.get(IMyService.class)` would fail because there are several equivalent candidates. Equivalent candidates means they have the same `@Order` value and the system cannot decide which one is the right one.

2.3.6. Bean Decoration

Bean decorations allow to wrap interfaces with a proxy to intercept each method call to the interface of a bean and apply some custom logic. For this a `IBeanDecorationFactory` has to be implemented. This is one single factory instance for the entire application. It decides which decorators are created for a bean request. The factory is asked for decorators on every bean retrieval. This allows to write bean decoration factories depending on dynamic conditions.

As bean decoration factories are beans themselves, it is sufficient to create an implementation of `org.eclipse.scout.rt.platform.IBeanDecorationFactory` and to ensure this implementation is used (see Section 2.3.4). This factory receives the bean to be decorated and the originally requested bean class to decide which decorators it should create. In case no decoration is required the factory may return `null`. Then the original bean is used without decorations.

Decorations are only supported if the class obtained by the bean manager (e.g. by using `BEANS.get()`) is an interface!

It is best practice to mark all annotations that are interpreted in the bean decoration factory with the annotation `org.eclipse.scout.rt.platform.@BeanInvocationHint`. However this annotation has no effect at runtime and is only for documentation reasons.

The sample in listing Listing 7 wraps each call to the server with a profiler decorator that measures how long a server call takes.

Listing 7. Bean decoration example.

```
@Replace
public class ProfilerDecorationFactory extends SimpleBeanDecorationFactory {
  @Override
  public <T> IBeanDecorator<T> decorate(IBean<T> bean, Class<? extends T> queryType) {
    return new BackendCallProfilerDecorator<>(super.decorate(bean, queryType));
  }
}

public class BackendCallProfilerDecorator<T> implements IBeanDecorator<T> {

  private final IBeanDecorator<T> m_inner;

  public BackendCallProfilerDecorator(IBeanDecorator<T> inner) {
    m_inner = inner;
  }

  @Override
  public Object invoke(IBeanInvocationContext<T> context) {
    final String className;
    if (context.getTargetObject() == null) {
      className = context.getTargetMethod().getDeclaringClass().getSimpleName();
    }
    else {
      className = context.getTargetObject().getClass().getSimpleName();
    }

    String timerName = className + '.' + context.getTargetMethod().getName();
    TuningUtility.startTimer();
    try {
      if (m_inner != null) {
        // delegate to the next decorator in the chain
        return m_inner.invoke(context);
      }
      // forward to real bean
      return context.proceed();
    }
    finally {
      TuningUtility.stopTimer(timerName);
    }
  }
}
```

2.3.7. Destroy Beans

Application scoped beans can declare methods annotated with `javax.annotation.@PreDestroy`. These methods will be called when the Scout platform is stopping. The methods may have any visibility modifier but must not be static and must not declare any parameters. If such a pre-destroy method throws an exception, the platform will continue to call all other pre-destroy methods (even methods

on the same bean).

Please note that pre-destroy methods are only called for application-scoped beans that already have created their instance.

Pre-destroy methods inherited from super classes are always called after the ones from the class itself. Methods that are overridden are only called on the leaf class. Private methods are always called (because they cannot be overridden). The order in which multiple methods in the same declaring class are called is undefined.

2.4. Configuration Management

Applications usually require some kind of configuration mechanism to use the same binaries in a different environment or situation. Scout applications provide a configuration mechanism using properties files [2: https://en.wikipedia.org/wiki/.properties].

For each property a class cares about default values and value validation. These classes share the `org.eclipse.scout.rt.platform.config.IConfigProperty` interface and are normal application scoped beans providing access to a specific configuration value as shown in listing Listing 8. If the property class is an inner class it has to be defined as a static class with the `static` modifier.

Listing 8. A configuration property of type Long.

```
import org.eclipse.scout.rt.platform.config.AbstractLongConfigProperty;

/**
 * Property of data type {@link Long} with key 'my.custom.timeout' and default value
'3600L'.
 */
public class MyCustomTimeoutProperty extends AbstractLongConfigProperty {

  @Override
  public String getKey() {
    return "my.custom.timeout"; ①
  }

  @Override
  protected Long getDefaultValue() {
    return 3600L; ②
  }
}
```

① key

② default value

To read the configured value you can use the `CONFIG` class as demonstrated in Listing 9.

Listing 9. Read the configured value in your code.

```
Long value = CONFIG.getPropertyValue(MyCustomTimeoutProperty.class);
```

The given property key is searched in the following environments:

1. In the system properties (`java.lang.System.getProperty(String)`).
2. In the properties file. The properties file can be
   a. a file on the local filesystem where the system property with key `config.properties` holds an absolute URL to the file or
   b. a file on the classpath with path `/config.properties` (recommended).
3. In the environment variables of the system (`java.lang.System.getenv(String)`).

Supported formats are simple key-value pairs, list values and map values. For more details about the format please refer to the JavaDoc of the `org.eclipse.scout.rt.platform.config.PropertiesHelper` class.

A properties file may import other config files from the classpath or any other absolute URL. This is done using the special key `import`. It can be a single value or a list:

- `import[0]=classpath:myConfigs/other.properties`
- `import[1]=file:/C:/path/to/my/settings.properties`
- `import[2]=file:${catalina.base}/conf/db_connection.properties`

2.4.1. Additional examples

Because the property classes are managed by the bean manager, you can use all the mechanisms to change the behavior (`@Replace` in particular).

Listing 10 demonstrates how you can use the replace annotation to change the existing `ApplicationNameProperty` class. The value is no longer fetched via the config mechanism, because the `getValue(String)` method is overriden. In this case a fixed value is returned.

Listing 10. Property class providing a constant value.

```
import org.eclipse.scout.rt.platform.IgnoreBean;
import org.eclipse.scout.rt.platform.Replace;
import
org.eclipse.scout.rt.platform.config.PlatformConfigProperties.ApplicationNameProperty;

@Replace
public class ApplicationNameConstant extends ApplicationNameProperty {
  @Override
  protected String readFromSource(String namespace) {
    return "Contacts Application";
  }
}
```

The next example presented in Listing 11 uses the same idea. In this case, the `getKey()` method is overriden to read the value from an other key as demonstrated is the Listing 12.

Listing 11. Property class reading the value from an other key.

```
import org.eclipse.scout.rt.platform.IgnoreBean;
import org.eclipse.scout.rt.platform.Replace;
import
org.eclipse.scout.rt.platform.config.PlatformConfigProperties.ApplicationNameProperty;

@Replace
public class ApplicationNamePropertyRedirection extends ApplicationNameProperty {

  @Override
  public String getKey() {
    return "myproject.application.name";
  }
}
```

Listing 12. Read the configured value in your code.

```
### Redirected Application Config
myproject.application.name=My Project Application
```

2.5. Testing

TODO [7.0] mvi

Chapter 3. Client Model

3.1. Desktop

3.1.1. Desktop Bench Layout

The Desktop Layout can be configured using the `IDesktop.setBenchLayoutData` method. This property is observed and might be changed during the applications lifecycle. The desktop consists out of 9 view stacks (see Figure 2). Each form can be assigned to a single view stack using the property DisplayViewId (`IForm.getConfiguredDisplayViewId`). If multiple forms are assigned to the same view stack the views will be displayed as tabs where the top form is visible and the corresponding tab selected.

Tabs are only visible if the form does have a title, subtitle or an image.

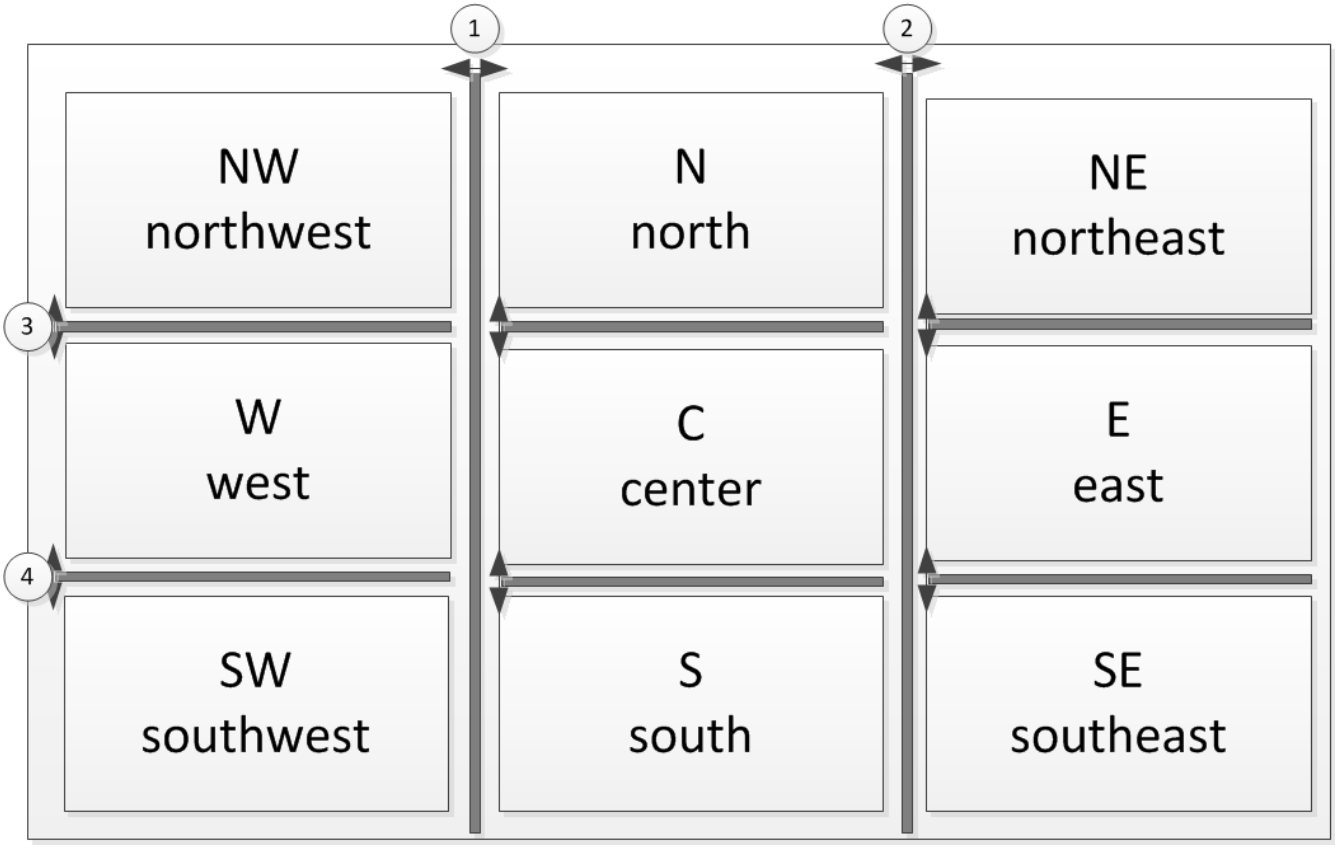


Figure 2. Desktop Bench overview

The east, center and west columns are separated with splitters which can be moved according to the layout data properties. Each column is split into a north, center and south part. Within a column the north, center and south parts can not differ in their width.

The modifications (splitter movements) are cached when a cache key (`BenchLayoutData.withCacheKey`) is set. In case the cache key is null the layout starts always with the initial values.

An example of a bench layout data configuration with a fixed north (N) view stack and an south (S) view stack with an minimal size. See

`org.eclipse.scout.rt.client.ui.desktop.bench.layout.FlexboxLayoutData` API for the documentation of the properties.

```
    desktop.setBenchLayoutData( ①
        new BenchLayoutData()
            .withCacheKey("a-cache-key") ②
            .withCenter( ③
                new BenchColumnData()
                    .withNorth(new FlexboxLayoutData().withGrow(0).withShrink(0)
.withInitial(280).withRelative(false)) ④
                    .withCenter(new FlexboxLayoutData()) ⑤
                    .withSouth(new FlexboxLayoutData().withShrink(0).withInitial(-1))
)); ⑥
  }
}
```

① set the `BenchLayoutData` to the desktop.

② set a cache key to store the layout modifications (dragging splitters) to the session store. Aware the settings are stored to the browsers session store they are not transfered over different browsers nor systems.

③ configure the center column (N, C, S).

④ The north part is fixed in size so the splitter between north (N) and center © view stack is disabled. The size is fixed to 280 pixel.

⑤ Use default for the center © view stack.

⑥ The south part is using the UI height as initial size and is growable but not shrinkable.

3.2. Outlines

3.3. Pages

3.4. Forms

3.5. Form Fields

3.6. Actions

3.7. Multiple Dimensions Support

Several components support multiple dimensions for visibility or enabled flags. This means the component is only visible or enabled if all dimensions are set to true. This gives developers the flexibility to e.g. use a dimension for granting and one for the business logic.

A total of 8 dimensions are available for a certain component type and attribute. This means you

e.g. have a total of 8 dimensions for Form Field visibility in your application. And 8 dimensions for enabled-states of Actions. So the dimensions are not consumed by component instance but by component type. This means you have to be careful in defining new dimensions as all components of the same type share these dimensions.

Some of these dimensions are already used internally. Refer to the implementation and JavaDoc of the component for details about how many dimensions are available for custom use.

```
menu.setEnabled(false); ①
menu.setEnabledGranted(false); ②
menu.setVisible(false, IDimensions.VISIBLE_CUSTOM); ③
formField.setVisible(true, false, true, "MyCustomDimension"); ④
formField2.setVisible(true, true, true); ⑤

formField3.isEnabled(IDimensions.ENABLED_CUSTOM); ⑥
formField3.isEnabled(IDimensions.ENABLED); ⑦
formField3.isEnabled(); ⑧
formField3.isEnabledIncludingParents(); ⑨
```

① Disables the menu using the internal default dimension

② Disables the menu using the internal granted dimension

③ Hides the menu with a third custom dimension

④ Form Fields also support the propagation of new values to children and parents. This sets the custom dimension of this field and all of its children to true.

⑤ This sets the internal default enabled dimension of this field and all of its parents and children to true.

⑥ Checks if the custom dimension is set to true

⑦ Checks if the internal default dimension is set to true

⑧ Checks if all dimensions of formField2 are true

⑨ Checks if all dimensions of formField2 and all dimensions of all parent Form Fields are enabled.

In the example above the instance 'formField3' uses 4 dimensions for the enabled attribute: ENABLED_CUSTOM because it is explicitly used and the 3 dimensions that are used internally (ENABLED, ENABLED_GRANTED, ENABLED_SLAVE). Even though the instance 'formField2' makes no use of the custom dimension it is consumed for this instance as well because the dimensions do not exist by instance but by attribute (as explained above).

Chapter 4. Texts

The `TEXTS` class is a convenience class to access the default Text Provider Service used for the localization of the texts in the user interface.

Listing 13. Text lookup

```
TEXTS.get("persons");
```

Its also possible to use some parameters:

Listing 14. Text lookup

```
String name = "Bob";
int age = 13;

TEXTS.get("NameWithAge", name, age);
```

In this case, some placeholders for the parameters are needed in the translated text:

Listing 15. Text lookup

```
 NameWithAge={0} is {1} years old;
```

4.1. Text properties files

Scout uses the java.util.ResourceBundle mechanism for native language support. So whatever language files you have in your `<project-prefix>.shared/resources/texts/*.properties` are taken as translation base.

Example setup:

- <project-prefix>.shared/resources/texts/Texts.properties
- <project-prefix>.shared/resources/texts/Texts_fr.properties

If your application starts with the `-vmargs -Duser.language=fr` or `eclipse.exe -nl=fr` the translations in Texts_fr.properties are considered. In case of any other user language the translations in Texts.properties are considered.

It is possible to edit these files in the Eclipse Scout SDK with the NLS Editor.

4.2. Text Provider Service

Text Provider Services are services responsible to provide localization for texts in the user interface. A typical application contains a such service contributed by the Shared Project.

- implements: `ITextProviderService`

- extends: `AbstractDynamicNlsTextProviderService` (default, translations are stored in properties files)

Using Text Provider Services developers can decide to store the translations in a custom container like a database or XML files. Furthermore using TextProviderServices it is very easy to overwrite any translated text in the application (also texts used in Scout itself) using the service ranking.

The mechanism is aligned with the icon retrieval which is also managed using Icon Provider Services.

4.2.1. Localization using .properties files

By default the internationalization mechanism relies on .properties files using a reference implementation of the TextProviderServices:

Service extending the `AbstractDynamicNlsTextProviderService` class.

A Text Provider Service working with the default implementation need to define where the properties files are located. This is realized by overriding the getter getDynamicNlsBaseName(). Here an example:

Listing 16. Text lookup

```
@Override
protected String getDynamicNlsBaseName() {
  return "resources.texts.Texts";
}
```

If configured like this, it means that the .properties files will be located in the same plug-in at the location:

- `/resources/texts/Texts.properties` (default)
- `/resources/texts/Texts_fr.properties` (french)
- `/resources/texts/Texts_de.properties` (german)
- ⋯ (additional languages)

If you decide to store your translated texts in .properties files, you migth want to use the NLS Editor to edit them.

You need to respect the format defined by the Java Properties class. In particular the encoding of a .properties file is ISO-8859-1 (also known as Latin-1). All non-Latin-1 characters must be encoded. Examples:

```
'à' => "\u00E0"
'ç' => "\u00E7"
'ß' => "\u00DF"
```

The encoding is the "Unicode escape characters": \uHHHH where HHHH is a hexadecimal id of the

character in the Unicode character table. Read more on the .properties File on wikipedia.

Chapter 5. Icons

A lot of Scout widgts support icons. For instance a menu item can show an icon next to the menu text. Icons in Scout can be either a bitmap image (GIF, PNG, JPEG, etc.) or a character from an icon-font. An example for an icon-font is the *scoutIcons.ttf* which comes shipped with Scout.

It's a good practice to define the available icons in your application in a class that defines each icon as a constant. Create a class `Icons` in the shared module of your project. These constants should be references, when you set the *IconId* property in your code.

For bitmap images you simply specify the filename of the image file *without* the file extension. Place all your icon files in the resource folder of your *client* module. Assuming your project name is "org.scout.hello", the correct location to store icon files would be:

```
org.scout.hello.client/              # Client project directory
  src/main/resources/                # Resources directory
    org/scout/hello/client/icons/    # Path to icons
      application_logo_large.png
      person.png
      ...
```

Listing 17. Icons.java

```
// Bitmap image (references icons/application_logo_large.png)
public static final String ApplicationLogo = "application_logo_large";

// Character from icon-font scoutIcons.woff (default)
public static final String Calendar ="font:\uE003";

// Character from a custom icon-font
public static final String Phone ="font:awesomeIcons \uF095";
```

Listing 18. Usage of iconId in a Scout widget

```
@Override
protected String getConfiguredIconId(){
  return Icons.Calendar;
}
```

5.1. Using a custom icon font

You can use your own icon font. The required file format for an icon font is *.woff*. For the following examples we assume the name of your font file is *awesomeIcons.woff*. The following steps are required:

Place the font file in the WebContent/res directory of your *html.ui* module. This makes it available for http requests on the URL `http://[base]/res/awesomeIcons.woff`.

Create a CSS/LESS definition, to reference the icon font in stylesheets. Make sure the definition is added to the LESS module of your project.

Listing 19. The CSS/LESS font definition should look like this:

```
@font-face {
  font-family: awesomeIcons;
  font-weight: normal;
  src: url('awesomeIcons.woff') format('woff');
}

/* Overrides definitions in fonts.css > .font-icon
* Use iconId 'font:awesomeIcons [character]' in Scout model.
* See icons.js and usage of this class to see how iconId is used.
*/
.font-awesomeIcons {
  font-family: awesomeIcons, @defaultFont;
}
```

To check if your CSS definition is correct, you should download the CSS file directly via URL and check if the CSS file contains the required font definition. Assuming your LESS macro is named *hello-scout-macro.less* the URL is: `http://[base]/res/hello-scout.css`

When you request resources from the */res* folder via http, Scout will find resources from parent modules too. Thus the *scoutIcons.woff* is always available in a Scout project. However, you must avoid naming conflicts, since at runtime all files exist on the same classpath.

5.2. How to create a custom icon font

Here's what we do to create and maintain our own icon font *scoutIcons.woff*. There may be other methods to achieve the same.

To create and modify our icon font we use the online application IcoMoon. IcoMoon allows you to assemble a set of icons from various sources (e.g. FontAwesome or custom SVG graphics) and create a font file from that set.

You can export/import your icon set from and to IcoMoon, and you should store the files exported from IcoMoon in a SCM system like GIT. IcoMoon stores all important data in the file *selection.json*. Make sure you also store the raw SVG graphics you've uploaded to IcoMoon in your SCM, in case you have to change a single icon later.

To edit the icon font in IcoMoon follow these steps:

- Import *selection.json* in IcoMoon, click on the "Import Icons" button.
- With the *Select* tool (arrow) you select the icons you want to add to your set. You can also add one or more characters from other icon fonts like FontAwesome by choosing *Add Icons From Library...*

- Your can import your custom SVG graphics with *Import to Set*, which you find in the hamburger menu on the icon set. The SVG graphic should have the same size as the other icons in the set and must use only a single color, black. The background must be transparent. Hint: the filename of the SVG graphic should contain the unicode of the character in the font in order to simplify maintencance. Only use unicodes from the *Private Use Area* from U+E000 to U+F8FF.
- When you're happy with your icon set, you hit the *Generate Font* button in the footer in IcoMoon. On the following page you can set the unicode of each icon/character. Click on the prefences button (cog icon), to set the name of your icon font (e.g. scoutIcons). Finally click on *Download* and you receive a ZIP file which contains the new *selection.json*, and font files like .ttf and .woff.
- When you've added new unicodes to the icon font, you should also update *Icons.java* and add constants for the new characters. When you're using *Scout JS* you should also update *icons.js* and *icons.less*.
- **Important!** don't forget to check in the new *selection.js* to your SCM.

5.2.1. Tools

- Windows tool *Character Map*: first you must install your custom TrueType Font .ttf in Windows. Simply double-click on the .ttf file and choose *Install*. After that you can start *Character Map* and browse through the font.
- The ZIP archive from IcoMoon contains a file *demo.html*. This file shows a preview of your icon font. Works in Chrome, but we had trouble viewing the font with Firefox.
- This tool from Wikipedia also creates a preview for an icon font: Vorlage:Private-Use-Area-Test. Icon font must be installed first.

Chapter 6. Lookup Call

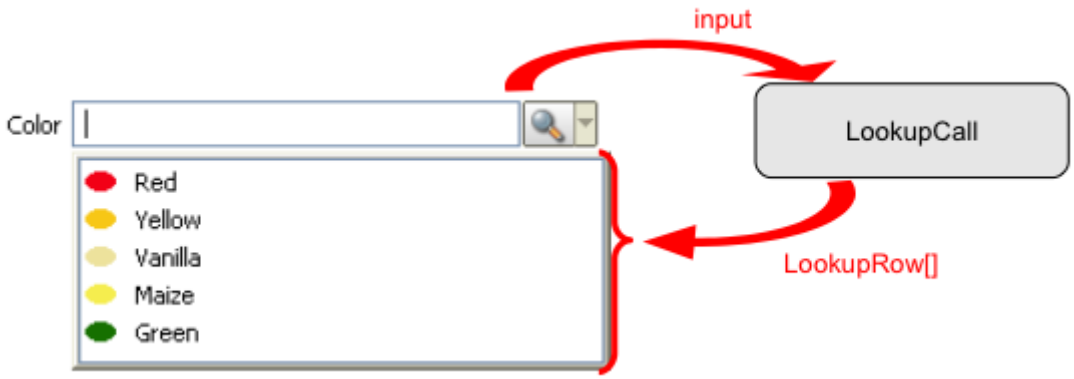
Lookup calls are mainly used by SmartFields and SmartColumns to look up single or multiple LookupRows.

Class: `LookupCall`

6.1. Description

The Lookup call mechanism is used to lookup up a set of key-text pairs. Whereas the key can be of any Java type the text must be of the type String. Each entry in this set is called LookupRow. In addition to the key and the text a LookupRow can also define and icon, font, colors and a tooltiptext.

This schema explains the role of a LookupCall in a SmartField:



6.2. Input

Lookup calls provide different method to compute the set of LookupRows :

- **getDataByKey():** Retrieves a single lookup row for a specific key value. Used by SmartFields and SmatColumns to get the display text for a given key value.
- **getDataByText():** Retrieve multiple lookup rows which match a certain String. Used by SmartFields when the user starts to enter some text in the field.
- **getDataByAll():** Retrieves all available lookup rows. Used by SmartFields when the user clicks on the browse icon.
- **getDataByRec():** This can only be used for hierarchical lookup calls. It retrieves all available sub-tree lookup rows for a given parent.

6.3. Members

The Lookup call contains attributes (accessible with getter and setter) that can be used to compute the list of lookups rows. Out of the box you have:

- key: contains the key value when the lookup is queried by key.

- text: contains the text input in case of a text lookup (typically this is the text entered by the user smart field).
- all: contains the browse hint in case of a lookup by all (typically when a user click on the button to see all proposal in a smart field).
- rec: contains the key of the parent entry, in when the children of a node are loaded.
- master: contains the value of the master field (if a master field is associated to the field using the lookup call).

It is possible to add you own additional attributes, for example validityFrom, validityTo as date parameter. Just add them as field with getter and setter:

```
public class LanguageLookupCall extends LookupCall<String> {
  // other stuff like serialVersionUID, Lookup Service definition...

  private static final long serialVersionUID = 1L;

  private Date m_validityFrom;
  private Date m_validityTo;

  @Override
  protected Class<? extends ILookupService<String>> getConfiguredService() {
    return ILanguageLookupService.class;
  }

  public Date getValidityFrom() {
    return m_validityFrom;
  }

  public void setValidityFrom(Date validityFrom) {
    this.m_validityFrom = validityFrom;
  }

  public Date getValidityTo() {
    return m_validityTo;
  }

  public void setValidityTo(Date validityTo) {
    this.m_validityTo = validityTo;
  }
}
```

In this case, you might want to set your properties bevor the lookupcall query is sent. This can be done with the PrepareLookup event of the SmartField or the ListBox:

```
    @Override
    protected void execPrepareLookup(ILookupCall<String> call) {
      LanguageLookupCall c = (LanguageLookupCall) call;
      c.setValidityFrom(DateUtility.parse("2012-02-26", "yyyy-mm-dd"));
      c.setValidityTo(DateUtility.parse("2013-02-27", "yyyy-mm-dd"));
    }
```

If you follow this pattern, you will consume the values in the server, by casting the call:

```
  @Override
  public List<? extends ILookupRow<String>> getDataByAll(ILookupCall<String> call) {
    LanguageLookupCall c = (LanguageLookupCall) call;
    Date validityFrom = c.getValidityFrom();
    Date validityTo = c.getValidityTo();

    List<? extends ILookupRow<String>> result = new ArrayList<>();
    //compute result: corresponding lookup rows (depending on validityFrom and
validityTo).
    return result;
  }
```

6.4. Type of lookup calls

6.4.1. With a Lookup Service

Delegation to the Lookup Service on server side.

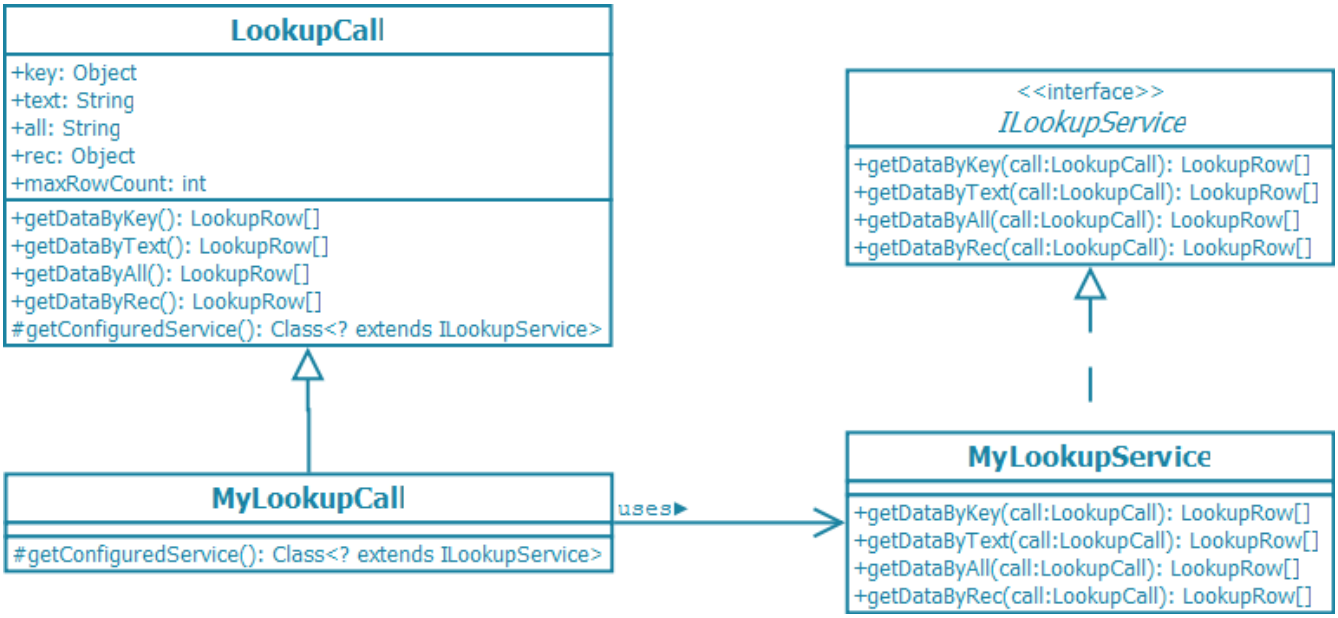
They are not necessarily restricted to a fix number of records. Instead they should be favoured if the set of records is rather large.

6.4.2. Directy

Principe of the Local Lookup Calls

An example of this approach is when a SmartField or a SmartColumn is configured to be use with a CodeType. A `CodeLookupCall` is instantiated for the CodeType. It creates the LookupRows corresponding to the codes in the CodeType.

6.4.3. Overview



6.4.4. Properties

Defined with `getConfiguredXxxxxx()` methods.

- Service: Defines which service is used to retrieve lookup rows
- MasterRequired: Defines whether a master value must be set in order to query for multiple lookup rows

6.4.5. Code examples

Using a LookupCall in a SmartField:

```
@Override
protected Class<? extends ILookupCall<String>> getConfiguredLookupCall() {
  return LanguageLookupCall.class;
}
```

Accessing a LookupRow directly:

It is possible to access a LookupRow direclty. In this example the input is a key (thisKey) and the method getDataByKey() is used. Before accessing the text, we ensure that a LookupRow has been retrived.

```
//Execute the LookupCall (using DataByKey)
LookupCall<String> call = new LanguageLookupCall();
call.setKey(thisKey);
List<? extends ILookupRow<String>> rows = call.getDataByKey();

//Get the text (with a null check)
String text = null;
if (rows != null && !rows.isEmpty()) {
  text = rows.get(0).getText();
}
```

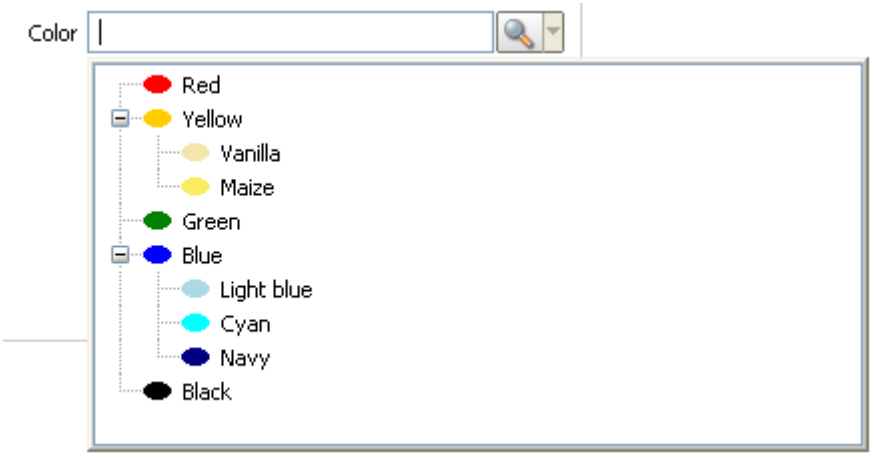
Chapter 7. Code Type

A CodeType is a structure to represent a tree key-code association. They are used in SmartField and SmartColumn.

- implements: `ICodeType<T>`
- extends: `AbstractCodeType<T>`

7.1. Description

CodeTypes are used in SmartField to let the user choose between a finite list of values. The value stored by the field corresponds to the key of the selected code.



A CodeType can be seen as a tree of Codes. Each code associates to the key (the Id) other properties: among others a Text and an IconId.

In order to have the same resolving mechanism (getting the display text of a key), CodeTypes are also used in SmartColumns. To choose multiple values in the list, the fields ListBox (flat CodeType) and TreeBox (hierarchical CodeType) can be used.

7.1.1. Organisation of the codes

The codes are organized in a tree. Therefore a CodeType can have one or more child codes at the root level, and each code can have other child codes. In a lot of cases a list of codes (meaning a tree containing only leaves at the first level) is sufficient to cover most of the need.

Child codes are ordered in their parent code. This is realized with the order annotation.

7.1.2. Type of the key

The type of the key is defined by its generic parameter `<T>`. It is very common to use a type from the `java.lang.*` package (like `Integer` or `String`), but any Java Object is suitable. It must:

- implement `Serializable`
- have correctly implemented `equals()` and `hashCode()` functions

- be present in the server and the client

There is no obligation to have the same type for the Id between the codes of a CodeType (meaning the same generic type parameter for the codes inner-class). However it is a good practice to have the same type between the codes of a CodeType, because the Id are used as value of SmartFields. Therefore the generic parameter describing the type of value of a SmartField must be compatible with the type of the codes contained in the CodeType.

7.2. Using a CodeType

7.2.1. SmartField or SmartColumn

CodeType in a SmartField (or SmartColumn).

```
public class YesOrNoSmartField extends AbstractSmartField<Boolean> {

  // other configuration of properties.

  @Override
  protected Class<? extends ICodeType<?, Boolean>> getConfiguredCodeType() {
    return YesOrNoCodeType.class;
  }
}
```

If the SmartField (or SmartColumn) works with a CodeType, a specific LookupCall is instantiated to get the LookupRows based on the Codes contained in a CodeType.

7.2.2. Accessing a code directly

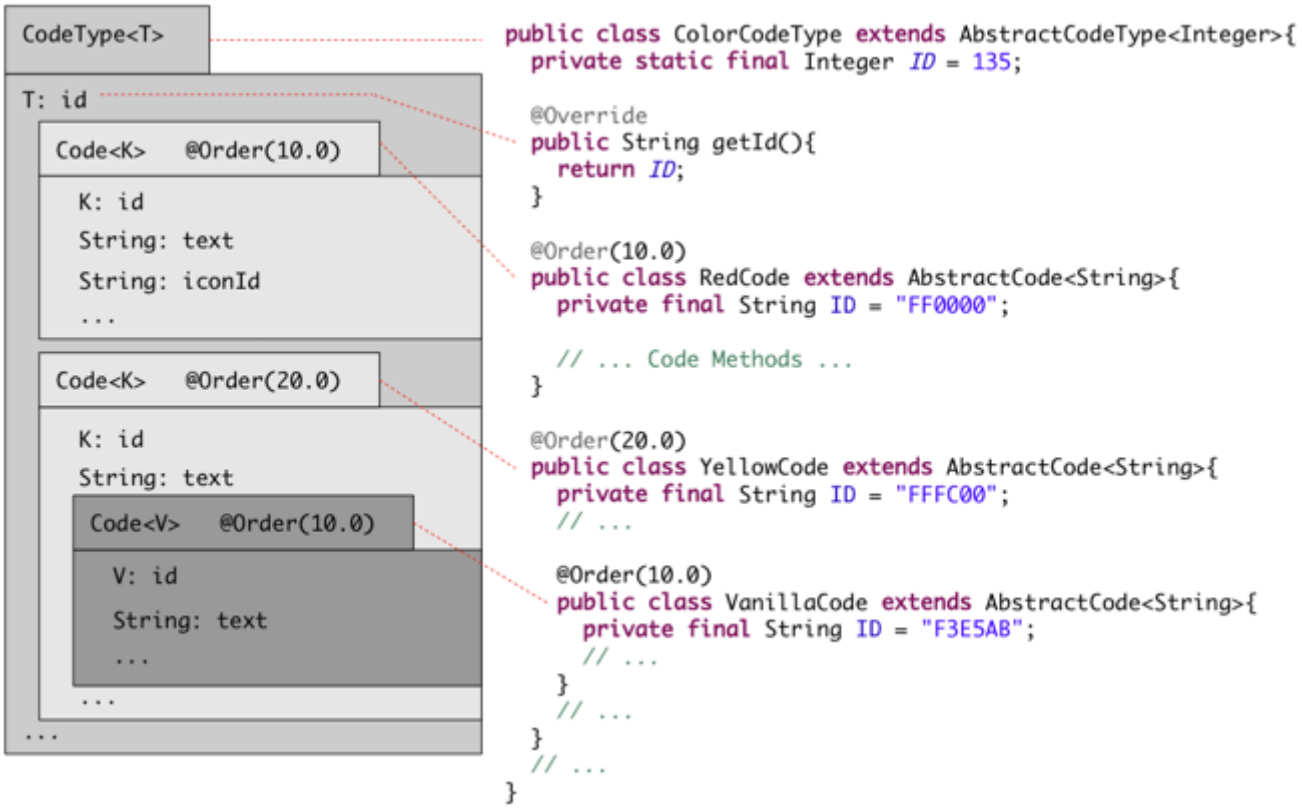
Scout-runtime will handle the instantiation and the caching of CodeTypes.

This function returns the text corresponding to the key using a CodeType:

```
  public String getCodeText(boolean key) {
    ICode c = BEANS.get(YesOrNoCodeType.class).getCode(key);
    if (c != null) {
      return c.getText();
    }
    return null;
  }
```

7.3. Static CodeType

7.3.1. Java Code and structure



The common way to define a CodeType is to extend AbstractCodeType. Each code is an inner-class extending AbstractCode. Like usual the properties of Codes and CodeTypes can be set using the getConfiguredXxxxxx() methods.

See the Java Code of a simple `YesOrNoCodeType` having just two codes:

- `YesOrNoCodeType.YesCode`
- `YesOrNoCodeType.NoCode`

7.3.2. With the SDK

The SDK provides some help to generate CodeTypes and Codes. Use File → New → Scout → Scout Code Type to generate a new code.

7.4. Dynamic CodeType

Code types are not necessarily hardcoded. It is possible to implement other mechanisms to load a CodeType dynamically.

The description of the Codes can come from a database or from an XML files. If you want to do so, you just need to implement the method corresponding to the event LoadCodes.

It is possible to use the static and the dynamic approach together. In this case, if there is a conflict (2 codes for the same id) the event OverwriteCode is triggered.

Note for advanced users:

Each CodeType is instantiated for

- each language
- each partition

Note: A drawback is that the CodeType class is not aware of the language and the partition it is instantiated for. Only the `CodeTypeStore` that manages the CodeType instances knows for which language and which partition they have been instantiated.

Chapter 8. Working with exceptions

Exceptions can be logged via SLF4J Logger, or given to exception handler for centralized, consistent exception handling, or translated into other exceptions. Scout provides some few exceptions/errors, which are used by the framework.

8.1. Scout Throwables

All scout throwables are unchecked and typically implementing the `IThrowableWithContextInfo` interface, which provides functionality for associating context information with the occurred error.

Most scout throwables are runtime exceptions, and typically inherit from PlatformException. See Section 8.1.1 for more information.

Some scout throwables are instances of `java.lang.Error` by extending PlatformError. Those errors usually provide functionality to interrupt Jobs, for example when a user is canceling a long running operation.
Note: **PlatformErrors should never be catched by business logic!** See Section 8.1.2 for more information.

8.1.1. Scout Runtime Exceptions

PlatformException

Base runtime exception of the Scout platform, which allows for message formatting anchors and context information to be associated.

There is a single constructor which accepts the exception's message, and optionally a variable number of arguments. Typically, a potential cause is given as its argument. The message allows further the use of formatting anchors in the form of {} pairs. The respective formatting arguments are provided via the constructor's *varArg* parameter. If the last argument is of the type `Throwable` and not referenced as formatting anchor in the message, that `Throwable` is used as the exception's cause. Internally, SLF4J `MessageFormatter` is used to provide substitution functionality. Hence, The format is the very same as if using SLF4j Logger.

Further, `PlatformException` allows to associate context information, which are available in Log4j *diagnostic context map* (MDC) upon logging the exception.

Listing 20. PlatformException examples

```
Exception cause = new Exception();

// Create a PlatformException with a message
new PlatformException("Failed to persist data");

// Create a PlatformException with a message and cause
new PlatformException("Failed to persist data", cause);

// Create a PlatformException with a message with formatting anchors
new PlatformException("Failed to persist data [entity={}, id={}]", "person", 123);

// Create a PlatformException with a message containing formatting anchors and a cause
new PlatformException("Failed to persist data [entity={}, id={}]", "person", 123,
cause);

// Create a PlatformException with context information associated
new PlatformException("Failed to persist data", cause)
    .withContextInfo("entity", "person")
    .withContextInfo("id", 123);
```

ProcessingException

Represents a PlatformException and is thrown in case of a processing failure, and which can be associated with an exception error code and severity.

VetoException

Represents a ProcessingException with VETO character. If thrown server-side, exceptions of this type are transported to the client and typically visualized in the form of a message box.

AssertionException

Represents a PlatformException and indicates an assertion error about the application's assumptions about expected values.

TransactionRequiredException

Represents a PlatformException and is thrown if a ServerRunContext requires a transaction to be available.

8.1.2. Scout Runtime Errors

Runtime Errors are used to indicate an error, that shouldn't be catched/treated by business logic and therefore bubble up to the appropriate exception handler in the scout framework. Because those errors are handled by the framework internals, they should never be catched on the server (Services etc.) nor on the client side (Pages, Forms, etc.).

All Scout Runtime Errors extend PlatformError.

PlatformError

Like PlatformException, PlatformErrors implement `IThrowableWithContextInfo` for associating context information with the occurred error. See PlatformException for usage and example code.

ThreadInterruptedError

Represents a PlatformError and indicates that a thread was interrupted while waiting for some condition to become true, e.g. while waiting for a job to complete. Unlike `java.lang.InterruptedException`, the thread's interrupted status is not cleared when catching this exception.

FutureCancelledError

Represents a PlatformError and indicates that the result of a job cannot be retrieved, or the IFuture's completion not be awaited because the job was cancelled.

TimedOutError

Represents a PlatformError and indicates that the maximal wait time elapsed while waiting for some condition to become true, e.g. while waiting a job to complete.

8.2. Exception handling

An exception handler is the central point for exception handling. It provides a single method 'handle' which accepts a `Throwable`, and which never throws an exception. It is implemented as a bean, meaning managed by the bean manager to allow easy replacement, e.g. to use a different handler when running client or server side. By default, a ProcessingException is logged according to its severity, a VetoException, ThreadInterruptedError or FutureCancelledError logged in *DEBUG* level, and any other exception logged as an *ERROR*. If running client side, exceptions are additionally visualized and showed to the user.

8.3. Exception translation

Exception translators are used to translate an exception into another exception.

Also, they unwrap the cause of wrapper exceptions, like `UndeclaredThrowableException`, or `InvocationTargetException`, or `ExecutionException`. If the exception is of the type `Error`, it is normally not translated, but re-thrown instead. That is because an `Error` indicates a serious problem due to an abnormal condition.

8.3.1. DefaultExceptionTranslator

Use this translator to work with checked exceptions and runtime exceptions, but not with `Throwable`.

If given an `Exception`, or a `RuntimeException`, or if being a subclass thereof, that exception is returned as given. Otherwise, a PlatformException is returned which wraps the given Throwable.

8.3.2. DefaultRuntimeExceptionTranslator

Use this translator to work with runtime exceptions. When working with RunContext or IFuture, some methods optionally accept a translator. If not specified, this translator is used by default.

If given a `RuntimeException`, it is returned as given. For a checked exception, a PlatformException is returned which wraps the given checked exception.

8.3.3. PlatformExceptionTranslator

Use this translator to work with PlatformExceptions.

If given a PlatformException, it is returned as given. For all other exceptions (checked or unchecked), a PlatformException is returned which wraps the given exception.

Typically, this translator is used if you require to add some context information via `IThrowableWithContextInfo.withContextInfo(String, Object, Object)`.

Listing 21. PlatformException examples

```
try {
  // do something
}
catch (Exception e) {
  throw BEANS.get(PlatformExceptionTranslator.class).translate(e)
      .withContextInfo("cid", "12345")
      .withContextInfo("user", Subject.getSubject(AccessController.getContext()))
      .withContextInfo("job", IFuture.CURRENT.get());
}
```

8.3.4. NullExceptionTranslator

Use this translator to work with `Throwable` as given.

Also, if given a wrapped exception like `UndeclaredThrowableException`, `InvocationTargetException` or `ExecutionException`, that exception is returned as given without unwrapping its cause.

For instance, this translator can be used if working with the Job API, e.g. to distinguish between a FutureCancelledError thrown by the job's runnable, or because the job was effectively cancelled.

8.4. Exception Logging

Scout framework logs via SLF4J (Simple Logging Facade for Java). It serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time.

SLF4J allows the use of formatting anchors in the form of {} pairs in the message which will be replaced by the respective argument. If the last argument is of the type Throwable and not referenced as formatting anchor in the message, that `Throwable` is used as the exception.

Listing 22. Logging examples

```
Exception e = new Exception();

Logger logger = LoggerFactory.getLogger(getClass());

// Log a message
logger.error("Failed to persist data");

// Log a message with exception
logger.error("Failed to persist data", e);

// Log a message with formatting anchors
logger.error("Failed to persist data [entity={}, id={}]", "person", 123);

// Log a message and exception with a message containing formatting anchors
logger.error("Failed to persist data [entity={}, id={}]", "person", 123, e);
```

Chapter 9. JobManager

Scout provides a job manager based on Java Executors framework to run tasks in parallel, and on Quartz Trigger API to support for schedule plans and to compute firing times. A task (aka job) can be scheduled to commence execution either immediately upon being scheduled, or delayed some time in the future. A job can be single executing, or recurring based on some schedule plan. The job manager itself is implemented as an application scoped bean, meaning that it is a singleton which exists once in the web application.

9.1. Functionality

- immediate, delayed or timed execution
- single (one-shot) or repetitive execution (based on Quartz schedule plans)
- listen for job lifecycle events
- wait for job completion
- job cancellation
- limitation of the maximal concurrently level among jobs
- `RunContext` based execution
- configurable thread pool size (core pool size, max pool size)
- association of job execution hints to select jobs (e.g. to cancel or await job's completion)
- named jobs and threads to ease debugging

9.2. Job

A job is defined as some work to be executed asynchronously and is associated with a `JobInput` to describe how to run that work. The work is given to the job manager in the form of a `Runnable` or `Callable`. The only difference is, that a `Runnable` represents a 'fire-and-forget' action, meaning that the submitter of the job does not expect the job to return a result. On the other hand, a `Callable` returns the computation's result, which the submitter can await for. Of course, a runnable's completion can also be waited for.

Listing 23. Work that does not return a result

```
public class Work implements IRunnable {

  @Override
  public void run() throws Exception {
    // do some work
  }
}
```

Listing 24. Work that returns a computation result

```
public class WorkWithResult implements Callable<String> {

  @Override
  public String call() throws Exception {
    // do some work
    return "result";
  }
}
```

Upon scheduling a job, the job manager returns a `IFuture` to interact with the job, e.g. to cancel its execution, or to await its completion. The job itself can also access its `IFuture`, namely via `IFuture.CURRENT()` ThreadLocal.

Listing 25. Accessing the Future from within the job

```
public class Job implements IRunnable {

  @Override
  public void run() throws Exception {
    IFuture<?> myFuture = IFuture.CURRENT.get();
  }
}
```

9.3. Scheduling a Job

The job manager provides two scheduling methods, which only differ in the work they accept for execution (callable or runnable).

```
IFuture<Void> schedule(IRunnable runnable, JobInput input); ①

<RESULT> IFuture<RESULT> schedule(Callable<RESULT> callable, JobInput input); ②
```

① Use to schedule a runnable which does not return a result to the submitter

② Use to schedule a callable which does return a result to the submitter

The second and mandatory argument to be provided is the `JobInput`, which tells the job manager how to run the job. Learn more about JobInput.

The following snippet illustrates how a job is actually scheduled.

Listing 26. Schedule a job

```
IJobManager jobManager = BEANS.get(IJobManager.class); ①

②
jobManager.schedule(() -> {
  // do something
}, BEANS.get(JobInput.class)); ③
```

① Obtain the job manager via bean manager (application scoped bean)

② Provide the work to be executed (either runnable or callable)

③ Provide the `JobInput` to instrument job execution

This looks a little bit clumsy, which is why Scout provides you with the `Jobs` class to simplify dealing with the job manager, and to support you in the creation of job related artifacts like `JobInput`, filter builders and more. Most importantly, it allows to schedule jobs in a shorter and more readable form.

Listing 27. Schedule a job via Jobs helper class

```
Jobs.schedule(() -> {
  // do something
}, Jobs.newInput());
```

9.4. JobInput

The job input tells the job manager how to run the job. It further names the job to ease debugging, declares in which context to run the job, and how to deal with unhandled exceptions. The job input itself is a bean, useful if adding some additional features to the job manager. The API of `JobInput` supports for method chaining for reduced and more solid code.

Listing 28. Schedule a job and control execution via JobInput

```
Jobs.schedule(() -> {
  // do something
}, Jobs.newInput()
    .withName("job name") ①
    .withRunContext(ClientRunContexts.copyCurrent()) ②
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withStartIn(10, TimeUnit.SECONDS) ③
        .withSchedule(FixedDelayScheduleBuilder.repeatForever(5, TimeUnit.SECONDS)))
④
    .withExceptionHandling(new ExceptionHandler() { ⑤

      @Override
      public void handle(Throwable t) {
        System.err.println(t);
      }
    }, true));
```

This snippet instructs the job manager to run the job as following:

① Give the job a name.

② Run the job in the current calling context, meaning in the very same context as the submitter is running when giving this job to the job manager. By copying the current context, the job will also be cancelled upon cancellation of the current `RunContext`.

③ Commence execution in 10 seconds (delayed execution).

④ Execute the job repeatedly, with a delay of 5 seconds between the termination of one and the commencement of the next execution. Also, repeat the job infinitely, until being cancelled.

⑤ Print any uncaught exception to the error console, and do not propagate the exception to the submitter, nor cancel the job upon an uncaught exception.

The following paragraphs describe the functionality of `JobInput` in more detail.

9.4.1. JobInput.withName

To optionally specify the name of the job, which is used to name the worker thread (only in development environment) and for logging purpose. Optionally, *formatting anchors* in the form of {} pairs can be used in the name, which will be replaced by the respective argument.

```
Jobs.newInput()
    .withName("Sending emails [from={}, to={}]", "frank", "john@eclipse.org,
jack@eclipse.org");
```

9.4.2. JobInput.withRunContext

To optionally specify the `RunContext` to be installed during job execution. The `RunMonitor` associated with the `RunContext` will be used as the job's monitor, meaning that cancellation requests to the job

future or the context's monitor are equivalent. If no context is given, the job manager ensures a monitor to be installed, so that executing code can always query its cancellation status via `RunMonitor.CURRENT.get().isCancelled()`.

9.4.3. JobInput.withExecutionTrigger

To optionally set the trigger to define the schedule upon which the job will commence execution. If not set, the job will commence execution immediately after being scheduled, and will execute exactly once.

The trigger mechanism is provided by Quartz Scheduler, meaning that you can profit from the powerful Quartz schedule capabilities.

For more information, see http://www.quartz-scheduler.org.

Use the static factory method `Jobs.newExecutionTrigger()` to get an instance:

```
// Schedules a delayed single executing job
Jobs.newInput()
    .withName("job")
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withStartIn(10, TimeUnit.SECONDS));

// Schedules a repeatedly running job at a fixed rate (every hour), which ends in 24
hours
Jobs.newInput()
    .withName("job")
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withEndIn(1, TimeUnit.DAYS)
        .withSchedule(SimpleScheduleBuilder.repeatHourlyForever()));

// Schedules a job which runs at 10:15am every Monday, Tuesday, Wednesday, Thursday
and Friday
Jobs.newInput()
    .withName("job")
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withSchedule(CronScheduleBuilder.cronSchedule("0 15 10 ? * MON-FRI")));
```

Learn more about ExecutionTrigger.

9.4.4. JobInput.withExecutionSemaphore

To optionally control the maximal concurrently level among jobs assigned to the same semaphore.

With a semaphore in place, this job only commences execution, once a permit is free or gets available. If free, the job commences execution immediately at the next reasonable opportunity, unless no worker thread is available.

A semaphore initialized to one allows to run jobs in a mutually exclusive manner, and a semaphore

initialized to zero to run no job at all. The number of total permits available can be changed at any time, which allows to adapt the maximal concurrency level to some dynamic criteria like time of day or system load. However, a semaphore can be sealed, meaning that the number of permits cannot be changed anymore, and any attempts will be rejected.

A new semaphore instance can be obtained via `Jobs` class.

```
IExecutionSemaphore semaphore = Jobs.newExecutionSemaphore(5); ①

for (int i = 0; i < 100; i++) {
  ②
  Jobs.schedule(() -> {
    // doing something
  }, Jobs.newInput()
      .withName("job-{}", i)
      .withExecutionSemaphore(semaphore)); ③
}
```

① Create a new `ExecutionSemaphore` via `Jobs` class. The semaphore is initialized with 5 permits, meaning that at any given time, there are no more than 5 jobs running concurrently.

② Schedule 100 jobs in a row.

③ Set the semaphore to limit the maximal concurrency level to 5 jobs.

Learn more about ExecutionSemaphore.

9.4.5. JobInput.withExecutionHint

To associate the job with an execution hint. An execution hint is simply a marker to mark a job, and can be evaluated by filters to select jobs, e.g. to listen to job lifecycle events of some particular jobs, or to wait for some particular jobs to complete, or to cancel some particular jobs. A job may have multiple hints associated. Further, hints can be registered directly on the future via `IFuture.addExecutionHint(hint)`, or removed via `IFuture.removeExecutionHint(hint)`.

9.4.6. JobInput.withExceptionHandling

To control how to deal with uncaught exceptions. By default, an uncaught exception is handled by `ExceptionHandler` bean and then propagated to the submitter, unless the submitter is not waiting for the job to complete via `IFuture.awaitDoneAndGet()`.

This method expects two arguments: an optional exception handler, and a boolean flag indicating whether to swallow exceptions. 'Swallow' is independent of the specified exception handler, and indicates whether an exception should be propagated to the submitter, or swallowed otherwise.

If running a repetitive job with swallowing set to `true`, the job will continue its repetitive execution upon an uncaught exception. If set to false, the execution would exit.

9.4.7. JobInput.withThreadName

To set the thread name of the worker thread that will execute the job.

9.4.8. JobInput.withExpirationTime

To set the maximal expiration time upon which the job must commence execution. If elapsed, the job is cancelled and does not commence execution. By default, a job never expires.

For a job that executes once, the expiration is evaluated just before it commences execution. For a job with a repeating schedule, it is evaluated before every single execution.

In contrast, the trigger's end time specifies the time at which the trigger will no longer fire. However, if fired, the job may not be executed immediately at this time, which depends on whether having to compete for an execution permit first. So the end time may already have elapsed once commencing execution. In contrast, the expiration time is evaluated just before starting execution.

9.5. IFuture

A future represents the result of an asynchronous computation, and is returned by the job manager upon scheduling a job. The future provides functionality to await for the job to complete, or to get its computation result or exception, or to cancel its execution, and more.

Learn more about job cancellation in Section 9.9.
Learn more about listening for job lifecycle events in Section 9.10.
Learn more about awaiting the job's completion in Section 9.11.

9.6. Job states

Upon scheduling a job, the job transitions different states. The current state of a job can be queried from its associated IFuture.

state	description
SCHEDULED	Indicates that a job was given to the job manager for execution.
REJECTED	Indicates that a job was rejected for execution. This might happen if the job manager has been shutdown, or if no more worker threads are available.
PENDING	Indicates that a job's execution is pending, either because scheduled with a delay, or because of being a repetitive job while waiting for the commencement of the next execution.
RUNNING	Indicates that a job is running.
DONE	Indicates that a job finished execution, either normally or because it was cancelled. Use `IFuture.isCancelled()` to check for cancellation.
WAITING_FOR_PERMIT	Indicates that a semaphore aware job is competing for a permit to become available.
WAITING_FOR_BLOCKING_CONDITION	Indicates that a job is blocked by a blocking condition, and is waiting for it to fall.

The state 'done' does not necessarily imply that the job already finished execution. That is because a job also enters 'done' state upon cancellation, but may still continue execution.

9.7. Future filter

A future filter is a filter which can be passed to various methods of the job manager to select some futures. The filter must implement `IFilter` interface, and has a single method to accept futures of interest.

Listing 29. Example of a future filter

```
public class FutureFilter implements Predicate<IFuture<?>> {

  @Override
  public boolean test(IFuture<?> future) {
    // Accept or reject the future
    return false;
  }
}
```

Scout provides you with `FutureFilterBuilder` class to ease building filters which match multiple criteria joined by logical 'AND' operation.

Listing 30. Usage of FutureFilterBuilder

```
Predicate<IFuture<?>> filter = Jobs.newFutureFilterBuilder() ①
    .andMatchExecutionHint("computation") ②
    .andMatchNotState(JobState.PENDING) ③
    .andAreSingleExecuting() ④
    .andMatchNotFuture(IFuture.CURRENT.get()) ⑤
    .andMatchRunContext(ClientRunContext.class) ⑥
    .andMatch(new SessionFutureFilter(ISession.CURRENT.get())) ⑦
    .toFilter(); ⑧
```

① Returns an instance of the future filter builder

② Specifies to match only futures associated with execution hint 'computation'

③ Specifies to match only jobs not in state pending

④ Specifies to match only single executing jobs, meaning no recurring jobs

⑤ Specifies to exclude the current future (if any)

⑥ Specifies to match only jobs running on behalf of a `ClientRunContext`

⑦ Specifies to match only jobs of the current session

⑧ Builds the filters to get a Filter instance

Fore more information, refer to the JavaDoc of `FutureFilterBuilder`.

9.8. Event filter

A job event filter is a filter which can be given to job manager to subscribe for job lifecycle events. The filter must implement `IFilter` interface, and has a single method to accept events of interest.

Listing 31. Example of an event filter

```
public class EventFilter implements Predicate<JobEvent> {

  @Override
  public boolean test(JobEvent event) {
    // Accept or reject the event
    return false;
  }
}
```

Scout provides you with `JobEventFilterBuilder` class to ease building filters which match multiple criteria joined by logical 'AND' operation.

Listing 32. Usage of JobEventFilterBuilder

```
Predicate<JobEvent> filter = Jobs.newEventFilterBuilder() ①
    .andMatchEventType(JobEventType.JOB_STATE_CHANGED) ②
    .andMatchState(JobState.RUNNING) ③
    .andMatch(new SessionJobEventFilter(ISession.CURRENT.get())) ④
    .andMatchExecutionHint("computation") ⑤
    .toFilter(); ⑥
```

① Returns an instance of the job event filter builder

② Specifies to match all events representing a job state change

③ Specifies to match only events for jobs which transitioned into running state

④ Specifies to match only events for jobs of the current session

⑤ Specifies to match only events for jobs which are associated with the execution hint 'computation'

⑥ Builds the filters to get a Filter instance

Fore more information, refer to the JavaDoc of `JobEventFilterBuilder`.

9.9. Job cancellation

A job can be cancelled in two ways, either directly via its IFuture, or via job manager. Both expect you to provide a boolean flag indicating whether to interrupt the executing working thread. Upon cancellation, the job immediately enters 'done' state. Learn more about Section 9.6. If cancelling via job manager, a future filter must be given to select the jobs to be cancelled. Learn more about Section 9.7

The cancellation attempt will be ignored if the job has already completed or was cancelled. If not

running yet, the job will never run. If the job has already started, then the *interruptIfRunning* parameter determines whether the thread executing the job should be interrupted in an attempt to stop the job.

In the following some examples:

Listing 33. Cancel a job via its future

```
// Schedule a job
IFuture<?> future = Jobs.schedule(new Work(), Jobs.newInput());

// Cancel the job via its future
future.cancel(false);
```

Listing 34. Cancel multiple jobs via job manager

```
Jobs.getJobManager().cancel(Jobs.newFutureFilterBuilder()
    .andMatchFuture(future1, future2, future3)
    .toFilter(), false);
```

Listing 35. Cancel multiple jobs which match a specific execution hint and the current session

```
Jobs.getJobManager().cancel(Jobs.newFutureFilterBuilder()
    .andMatchExecutionHint("computation")
    .andMatch(new SessionFutureFilter(ISession.CURRENT.get()))
    .toFilter(), false);
```

A job can query its current cancellation status via `RunMonitor.CURRENT.get().isCancelled()`. If doing some long running operations, it is recommended for the job to regularly check for cancellation.

A job which is scheduled to run on a copy of the submitting `RunContext`, it gets also cancelled once the RunMonitor of that context gets cancelled.

9.10. Subscribe for job lifecycle events

Sometimes it is useful to register for some job lifecycle events. The following event types can be subscribed for:

state	description
JOB_STATE_CHANGED	Signals that a job transitioned to a new JobState, e.g. form JobState.SCHEDULED to JobState.RUNNING.
JOB_EXECUTION_HINT_ADDED	Signals that an execution hint was added to a job.
JOB_EXECUTION_HINT_REMOVED	Signals that an execution hint was removed from a job.

state	description
JOB_MANAGER_SHUTDOWN	Signals that the job manager was shutdown.

The listener is registered via job manager as following:

Listing 36. Subscribe for global job events

```
Jobs.getJobManager().addListener(Jobs.newEventFilterBuilder() ①
    .andMatchEventType(JobEventType.JOB_STATE_CHANGED)
    .andMatchState(JobState.RUNNING)
    .andMatch(new SessionJobEventFilter(ISession.CURRENT.get()))
    .toFilter(), event -> {
      IFuture<?> future = event.getData().getFuture(); ②
      System.out.println("Job commences execution: " + future.getJobInput().getName()
);
    });
```

① Subscribe for all events related to jobs just about to commence execution, and which belong to the current session

② Get the future this event was fired for

If interested in only events of a single future, the listener can be registered directly on the future.

Listing 37. Subscribe for local job events

```
future.addListener(Jobs.newEventFilterBuilder()
    .andMatchEventType(JobEventType.JOB_STATE_CHANGED)
    .andMatchState(JobState.RUNNING)
    .toFilter(), event -> System.out.println("Job commences execution"));
```

9.11. Awaiting job completion

A job's completion can be either awaited on its IFuture, or via job manager - the first optionally allows to consume the job's computation result, whereas the second allows multiple futures to be awaited for.

9.11.1. Difference between 'done' and 'finished' state

When awaiting futures, the definition of 'done' and 'finished' state should be understood - 'done' means that the future completed either normally, or was cancelled. But, if cancelled while running, the job may still continue its execution, whereas a job which not commenced execution yet, will never do so. The latter typically applies for jobs scheduled with a delay. However, 'finished' state differs from 'done' state insofar as a cancelled, currently running job enters 'finished' state only upon its actual completion. Otherwise, if not cancelled, or cancelled before executing, it is equivalent to 'done' state. In most situations, it is sufficient to await for the future's done state, especially because a cancelled job cannot return a result to the submitter anyway.

9.11.2. Awaiting a single future's 'done' state

Besides of some overloaded methods, IFuture basically provides two methods to wait for a future to enter 'done' state, namely `awaitDone` and `awaitDoneAndGet`, with the difference that the latter additionally returns the job's result or exception. If the future is already done, those methods will return immediately. For both methods, there exists an overloaded version to wait for at most a given time, which once elapsed results in a `TimedOutError` thrown.

Further, `awaitDoneAndGet` allows to specify an IExceptionTranslator to control exception translation. By default, DefaultRuntimeExceptionTranslator is used, meaning that a `RuntimeException` is propagated as it is, whereas a checked exception would be wrapped into a PlatformException. If you require checked exceptions to be thrown as they are, use DefaultExceptionTranslator instead, or even NullExceptionTranslator to work with the raw `ExecutionException` as being thrown by Java Executor framework.

Listing 38. Examples of how to await done state on a future

```
IFuture<String> future = Jobs.schedule(() -> {
  // doing something
  return "computation result";
}, Jobs.newInput());

// Wait until done without consuming the result
future.awaitDone(); ①
future.awaitDone(10, TimeUnit.SECONDS); ②

// Wait until done and consume the result
String result = future.awaitDoneAndGet(); ③
result = future.awaitDoneAndGet(10, TimeUnit.SECONDS); ④

// Wait until done, consume the result, and use a specific exception translator
result = future.awaitDoneAndGet(DefaultExceptionTranslator.class); ⑤
result = future.awaitDoneAndGet(10, TimeUnit.SECONDS, DefaultExceptionTranslator.
class); ⑥
```

① Waits if necessary for the job to complete, or until cancelled. This method does not throw an exception if cancelled or the computation failed, but throws ThreadInterruptedError if the current thread was interrupted while waiting.

② Waits if necessary for at most 10 seconds for the job to complete, or until cancelled, or the timeout elapses. This method does not throw an exception if cancelled, or the computation failed, but throws TimedOutError if waiting timeout elapsed, or throws ThreadInterruptedError if the current thread was interrupted while waiting.

③ Waits if necessary for the job to complete, and then returns its result, if available, or throws its exception according to DefaultRuntimeExceptionTranslator, or throws FutureCancelledError if cancelled, or throws ThreadInterruptedError if the current thread was interrupted while waiting.

④ Waits if necessary for at most 10 seconds for the job to complete, and then returns its result, if available, or throws its exception according to DefaultRuntimeExceptionTranslator, or throws

FutureCancelledError if cancelled, or throws TimedOutError if waiting timeout elapsed, or throws ThreadInterruptedError if the current thread was interrupted while waiting.

⑤ Waits if necessary for the job to complete, and then returns its result, if available, or throws its exception according to the given DefaultExceptionTranslator, or throws FutureCancelledError if cancelled, or throws ThreadInterruptedError if the current thread was interrupted while waiting.

⑥ Waits if necessary for at most the given time for the job to complete, and then returns its result, if available, or throws its exception according to the given DefaultExceptionTranslator, or throws FutureCancelledError if cancelled, or throws TimedOutError if waiting timeout elapsed, or throws ThreadInterruptedError if the current thread was interrupted while waiting.

It is further possible to await asynchronously on a future to enter done state by registering a callback via `whenDone` method. The advantage over registering a listener is that the callback is invoked even if the future already entered done state upon registration.

Listing 39. Example of when-done callback

```
future.whenDone(event -> {
  // invoked upon entering done state.
}, ClientRunContexts.copyCurrent());
```

Because invoked in another thread, this method optionally accepts a RunContext to be applied when being invoked.

9.11.3. Awaiting a single future's 'finished' state

Use the method `awaitFinished` to wait for the job to finish, meaning that the job either completed normally or by an exception, or that it will never commence execution due to a premature cancellation. To learn more about the difference between 'done' and 'finished' state, click here. Please note that this method does not return the job's result, because by Java Future definition, a cancelled job cannot provide a result.

Listing 40. Examples of how to await finished state on a future

```
IFuture<String> future = Jobs.schedule(() -> {
  // doing something
  return "computation result";
}, Jobs.newInput());

// Wait until finished
future.awaitFinished(10, TimeUnit.SECONDS);
```

9.11.4. Awaiting multiple future's 'done' state

Job Manager allows to await for multiple futures at once. The filter to be provided limits the futures to await for. This method requires you to provide a maximal time to wait.

Filters can be plugged by using logical filters like `AndFilter` or `OrFilter`, or negated by enclosing a

filter in `NotFilter`. Also see Section 9.7 to create a filter to match multiple criteria joined by logical 'AND' operation.

Listing 41. Examples of how to await done state of multiple futures

```
// Wait for some futures
Jobs.getJobManager().awaitDone(Jobs.newFutureFilterBuilder() ①
    .andMatchFuture(future1, future2, future3)
    .toFilter(), 1, TimeUnit.MINUTES);

// Wait for all futures marked as 'reporting' jobs of the current session
Jobs.getJobManager().awaitDone(Jobs.newFutureFilterBuilder() ②
    .andMatchExecutionHint("reporting")
    .andMatch(new SessionFutureFilter(ISession.CURRENT.get()))
    .toFilter(), 1, TimeUnit.MINUTES);
```

① Waits if necessary for at most 1 minute for all three futures to complete, or until cancelled, or the timeout elapses.

② Waits if necessary for at most 1 minute until all jobs marked as 'reporting' jobs of the current session complete, or until cancelled, or the timeout elapses.

9.11.5. Awaiting multiple future's 'finished' state

Use the method `awaitFinished` to wait for multiple jobs to finish, meaning that the jobs either completed normally or by an exception, or that they will never commence execution due to a premature cancellation. To learn more about the difference between 'done' and 'finished' state, click here.

Listing 42. Examples of how to await finish state of multiple futures

```
// Wait for some futures
Jobs.getJobManager().awaitFinished(Jobs.newFutureFilterBuilder() ①
    .andMatchFuture(future1, future2, future3)
    .toFilter(), 1, TimeUnit.MINUTES);

// Wait for all futures marked as 'reporting' jobs of the current session
Jobs.getJobManager().awaitFinished(Jobs.newFutureFilterBuilder() ②
    .andMatchExecutionHint("reporting")
    .andMatch(new SessionFutureFilter(ISession.CURRENT.get()))
    .toFilter(), 1, TimeUnit.MINUTES);
```

① Waits if necessary for at most 1 minute for all three futures to finish, or until cancelled, or the timeout elapses.

② Waits if necessary for at most 1 minute until all jobs marked as 'reporting' jobs of the current session finish, or until cancelled, or the timeout elapses.

9.12. Uncaught job exceptions

If a job throws an exception, that exception is handled by `ExceptionHandler`, and propagated to the submitter. However, the exception is only propagated if having a waiting submitter. Also, an uncaught exception causes repetitive jobs to terminate.

This default behavior as described can be changed via JobInput.withExceptionHandling(..).

9.13. Blocking condition

A blocking condition allows a thread to wait for a condition to become *true*. That is similar to the Java Object's 'wait/notify' mechanism, but with some additional functionality regarding semaphore aware jobs. If a semaphore aware job enters a blocking condition, it releases ownership of the permit, which allows another job of that same semaphore to commence execution. Upon the condition becomes *true*, the job then must compete for a permit anew.

A condition can be used across multiple threads to wait for the same condition. Also, a condition is reusable upon invalidation. And finally, a condition can be used even if not running within a job.

A blocking condition is often used by model jobs to wait for something to happen, but to allow another model job to run while waiting. A typical use case would be to wait for a MessageBox to be closed.

9.13.1. Example of a blocking condition

You are running in a semaphore aware job and require to do some long running operation. During that time you do not require to be the permit owner. A simple but wrong approach would be the following:

```
// Schedule a long running operation.
IFuture<?> future = Jobs.schedule(new LongRunningOperation(), Jobs.newInput());

// Wait until done.
future.awaitDone();
```

The problem with this approach is, that you still are the permit owner while waiting, meaning that you possibly prevent other jobs from running. Instead, you could use a blocking condition for that to achieve:

```
// Create a blocking condition.
final IBlockingCondition operationCompleted = Jobs.newBlockingCondition(true);

// Schedule a long running operation.
IFuture<Void> future = Jobs.schedule(new LongRunningOperation(), Jobs.newInput());

// Register done callback to unblock the condition.
future.whenDone(event -> {
  // Let the waiting job re-acquire a permit and continue execution.
  operationCompleted.setBlocking(false);
}, null);

// Wait until done. Thereby, the permit of the current job is released for the time
while waiting.
operationCompleted.waitForUninterruptibly();
```

9.14. ExecutionSemaphore

Represents a fair counting semaphore used in Job API to control the maximal number of jobs running concurrently.

Jobs which are assigned to the same semaphore run concurrently until they reach the maximal concurrency level defined for that semaphore. Subsequent tasks then wait in the queue until a permit becomes available.

A semaphore initialized to one allows to run jobs in a mutually exclusive manner, and a semaphore initialized to zero to run no job at all. The number of total permits available can be changed at any time, which allows to adapt the maximal concurrency level to some dynamic criteria like time of day or system load. However, once calling seal(), the number of permits cannot be changed anymore, and any attempts will result in an AssertionException. By default, a semaphore is unbounded.

9.15. ExecutionTrigger

Component that defines the schedule upon which a job will commence execution.

A trigger can be as simple as a 'one-shot' execution at some specific point in time in the future, or represent a schedule which executes a job on a repeatedly basis. The latter can be configured to run infinitely, or to end at a specific point in time. It is further possible to define rather complex triggers, like to execute a job every second Friday at noon, but with the exclusion of all the business's holidays.

See the various schedule builders provided by Quartz Scheduler: `SimpleScheduleBuilder`, `CronScheduleBuilder`, `CalendarIntervalScheduleBuilder`, `DailyTimeIntervalScheduleBuilder`. The most powerful builder is `CronScheduleBuilder`. Cron is a UNIX tool with powerful and proven scheduling capabilities. For more information, see http://www.quartz-scheduler.org.

Additionally, Scout provides you with `FixedDelayScheduleBuilder` to run a job with a fixed delay

between the termination of one execution and the commencement of the next execution.

Use the static factory method 'Jobs.newExecutionTrigger()' to get an instance.

9.15.1. Misfiring

Regardless of the schedule used, job manager guarantees no concurrent execution of the same job. That may happen, if using a repeatedly schedule with the job not terminated its last execution yet, but the schedule's trigger would like to fire for the next execution already. Such a situation is called a misfiring. The action to be taken upon a misfiring is configurable via the schedule's misfiring policy. A policy can be to run the job immediately upon termination of the previous execution, or to just ignore that missed firing. See the JavaDoc of the schedule for more information.

9.16. Stopping the platform

Upon stopping the platform, the job manager will also be shutdown. If having a IPlatformListener to perform some cleanup work, and which requires the job manager to be still functional, that listener must be annotated with an `@Order` less than `IJobManager.DESTROY_ORDER`, which is 5'900. If not specifying an `@Order` explicitly, the listener will have the default order of *5*, meaning being invoked before job manager shutdown anyway.

9.17. ModelJobs

Model jobs exist client side only, and are used to interact with the Scout client model to read and write model values in a serial manner per session. That enables no synchronization to be used when interacting with the model.

By definition, a model job requires to be run on behalf of a ClientRunContext with a `IClientSession` set, and must have the session's model job semaphore set as its ExecutionSemaphore. That causes all such jobs to be run in sequence in the model thread. At any given time, there is only one model thread active per client session.

The class `ModelJobs` is a helper class, and is for convenience purpose to facilitate the creation of model job related artifacts, and to schedule model jobs.

Listing 43. Running work in model thread

```
①
ModelJobs.schedule(() -> {
  // doing something in model thread
}, ModelJobs.newInput(ClientRunContexts.copyCurrent()) ②
    .withName("Doing something in model thread"));
```

① Schedules the work to be executed in the model thread

② Creates the `JobInput` to become a model job, meaning with the session's model job semaphore set

For model jobs, it is also allowed to run according to a Quartz schedule plan, or to be executed with a delay. Then the model permit is acquired just before each execution, and not upon being

scheduled.

Furthermore, the class `ModelJobs` provides some useful static methods:

```
// Returns true if the current thread represents the model thread for the current
client session. At any given time, there is only one model thread active per client
session.
ModelJobs.isModelThread();

// Returns true if the given Future belongs to a model job.
ModelJobs.isModelJob(IFuture.CURRENT.get());

// Returns a builder to create a filter for future objects representing a model job.
ModelJobs.newFutureFilterBuilder();

// Returns a builder to create a filter for JobEvent objects originating from model
jobs.
ModelJobs.newEventFilterBuilder();

// Instructs the job manager that the current model job is willing to temporarily
yield its current model job permit. It is rarely appropriate to use this method. It
may be useful for debugging or testing purposes.
ModelJobs.yield();
```

9.18. Configuration

Job manager can be configured with the following config properties.

property	default value	description
scout.jobmanager.corePoolSize	25	The number of threads to keep in the pool, even if they are idle
scout.jobmanager.prestartCoreThreads	true if running in productive environment, else false	Specifies whether all threads of the core-pool should be started upon job manager startup, so that they are idle waiting for work.
scout.jobmanager.maximumPoolSize	infinite	The maximal number of threads to be created once the core-pool-size is exceeded.
scout.jobmanager.keepAliveTime	60	The time limit (in seconds) for which threads, which are created upon exceeding the 'core-pool-size' limit, may remain idle before being terminated.
scout.jobmanager.allowCoreThreadTimeOut	false	Specifies whether threads of the core-pool should be terminated after being idle for longer than 'keepAliveTime'.

9.19. Extending job manager

Job manager is implemented as an application scoped bean, and which can be replaced. To do so, create a class which extends `JobManager`, and annotate it with `@Replace` annotation. Most likely, you like to use the EE container's `ThreadPoolExecutor`, or to contribute some behavior to the callable chain which finally executes the job.

To change the executor, overwrite `createExecutor` method and return the executor of your choice. But do not forget to register a rejection handler to reject futures upon rejection. Also, overwrite `shutdownExecutor` to not shutdown the container's executor.

To contribute some behavior to the callable chain, overwrite the method `interceptCallableChain` and contribute your decorator or interceptor. Refer to the method's JavaDoc for more information.

9.20. Scheduling examples

This sections contains some common scheduling examples.

Listing 44. Schedule a one-shot job

```
Jobs.schedule(() -> {
  // doing something
}, Jobs.newInput()
    .withName("Running once")
    .withRunContext(ClientRunContexts.copyCurrent()));
```

Listing 45. Schedule a job with a delay

```
Jobs.schedule(() -> {
  // doing something
}, Jobs.newInput()
    .withName("Running in 10 seconds")
    .withRunContext(ClientRunContexts.copyCurrent())
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withStartIn(10, TimeUnit.SECONDS))); // delay of 10 seconds
```

Listing 46. Schedule a repetitive job at a fixed rate

```
Jobs.schedule(() -> {
  // doing something
}, Jobs.newInput()
    .withName("Running every minute")
    .withRunContext(ClientRunContexts.copyCurrent())
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withStartIn(1, TimeUnit.MINUTES) ①
        .withSchedule(SimpleScheduleBuilder.simpleSchedule() ②
            .withIntervalInMinutes(1) ③
            .repeatForever()))); ④
```

① Configure to fire in 1 minute for the first time

② Use Quartz simple schedule to achieve fixed-rate execution

③ Repetitively fire every minute

④ Repeat forever

Listing 47. Schedule a repetitive job which runs 60 times at every minute

```
Jobs.schedule(() -> {
  // doing something
}, Jobs.newInput()
    .withName("Running every minute for total 60 times")
    .withRunContext(ClientRunContexts.copyCurrent())
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withStartIn(1, TimeUnit.MINUTES) ①
        .withSchedule(SimpleScheduleBuilder.simpleSchedule() ②
            .withIntervalInMinutes(1) ③
            .withRepeatCount(59)))); ④
```

① Configure to fire in 1 minute for the first time

② Use Quartz simple schedule to achieve fixed-rate execution

③ Repetitively fire every minute

④ Repeat 59 times, plus the initial execution

Listing 48. Schedule a repetitive job at a fixed delay

```
Jobs.schedule(() -> {
  // doing something
}, Jobs.newInput()
    .withName("Running forever with a delay of 1 minute between the termination of the
previous and the next execution")
    .withRunContext(ClientRunContexts.copyCurrent())
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withStartIn(1, TimeUnit.MINUTES) ①
        .withSchedule(FixedDelayScheduleBuilder.repeatForever(1, TimeUnit.MINUTES))));
②
```

① Configure to fire in 1 minute for the first time

② Use fixed delay schedule

Listing 49. Schedule a repetitive job which runs 60 times, but waits 1 minute between the termination of the previous and the commencement of the next execution

```
Jobs.schedule(() -> {
  // doing something
}, Jobs.newInput()
    .withName("Running 60 times with a delay of 1 minute between the termination of
the previous and the next execution")
    .withRunContext(ClientRunContexts.copyCurrent())
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withStartIn(1, TimeUnit.MINUTES) ①
        .withSchedule(FixedDelayScheduleBuilder.repeatForTotalCount(60, 1, TimeUnit
.MINUTES)))); ②
```

① Configure to fire in 1 minute for the first time

② Use fixed delay schedule

Listing 50. Running at 10:15am every Monday, Tuesday, Wednesday, Thursday and Friday

```
Jobs.schedule(() -> {
  // doing something
}, Jobs.newInput()
    .withName("Running at 10:15am every Monday, Tuesday, Wednesday, Thursday and
Friday")
    .withRunContext(ClientRunContexts.copyCurrent())
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withSchedule(CronScheduleBuilder.cronSchedule("0 15 10 ? * MON-FRI")))); ①
```

① Cron format: [second] [minute] [hour] [day_of_month] [month] [day_of_week] [year]?

Listing 51. Running every minute starting at 14:00 and ending at 14:05, every day

```
Jobs.schedule(() -> {
  // doing something
}, Jobs.newInput()
    .withName("Running every minute starting at 14:00 and ending at 14:05, every day")
    .withRunContext(ClientRunContexts.copyCurrent())
    .withExecutionTrigger(Jobs.newExecutionTrigger()
        .withSchedule(CronScheduleBuilder.cronSchedule("0 0-5 14 * * ?")))); ①
```

① Cron format: [second] [minute] [hour] [day_of_month] [month] [day_of_week] [year]?

Listing 52. Limit the maximal concurrency level among jobs

```
IExecutionSemaphore semaphore = Jobs.newExecutionSemaphore(5); ①

for (int i = 0; i < 100; i++) {
  Jobs.schedule(() -> {
    // doing something
  }, Jobs.newInput()
      .withName("job-{}", i)
      .withExecutionSemaphore(semaphore)); ②
}
```

① Create the execution semaphore initialized with 5 permits

② Set the execution semaphore to the job subject for limited concurrency

Listing 53. Cancel all jobs of the current session

```
Jobs.getJobManager().cancel(Jobs.newFutureFilterBuilder()
    .andMatch(new SessionFutureFilter(ISession.CURRENT.get()))
    .toFilter(), true);
```

Listing 54. Query for cancellation

```
public class CancellableWork implements IRunnable {

  @Override
  public void run() throws Exception {

    // do first chunk of operations

    if (RunMonitor.CURRENT.get().isCancelled()) {
      return;
    }

    // do next chunk of operations

    if (RunMonitor.CURRENT.get().isCancelled()) {
      return;
    }

    // do next chunk of operations
  }
}
```

Listing 55. Release current semaphore permit while executing

```
// Create a blocking condition.
final IBlockingCondition operationCompleted = Jobs.newBlockingCondition(true);

// Schedule a long running operation.
IFuture<Void> future = Jobs.schedule(new LongRunningOperation(), Jobs.newInput());

// Register done callback to unblock the condition.
future.whenDone(event -> {
  // Let the waiting job re-acquire a permit and continue execution.
  operationCompleted.setBlocking(false);
}, null);

// Wait until done. Thereby, the permit of the current job is released for the time
while waiting.
operationCompleted.waitForUninterruptibly();
```

Chapter 10. RunContext

Mostly, code is run on behalf of some semantic context, for example as a particular `Subject` and with some context related `ThreadLocals` set, e.g. the user's `session` and its `Locale`. Scout provides you with different `RunContexts`, such as `ClientRunContext` or `ServerRunContext`. They all share some common characteristics like `Subject`, `Locale` and `RunMonitor`, but also provide some additional functionality like transaction boundaries if using `ServerRunContext`. Also, a `RunContext` facilitates propagation of state among different threads. In order to ease readability, the 'setter-methods' of the `RunContext` support method chaining.

All a `RunContext` does is to provide some setter methods to construct the context, and a `run` and `call` method to run an action on behalf of that context. Thereby, the only difference among those two methods is their argument. Whereas `run` takes a `IRunnable` instance, `call` takes a `Callable` to additionally return a result to the caller. The action is run in the current thread, meaning that the caller is blocked until completion.

By default, a `RunContext` is associated with a `RunMonitor`, and the monitor's cancellation status can be queried via `RunMonitor.CURRENT.get().isCancelled()`. The monitor allows for hard cancellation, meaning that the executing thread is interrupted upon cancellation. For instance if waiting on an interruptible construct like `Object.wait()` or `IFuture.awaitDone()`, the waiting thread returns with an interruption exception.

10.1. Factory methods to create a RunContext

Typically, a `RunContext` is created from a respective factory like `RunContexts` to create a `RunContext`, or `ServerRunContexts` to create a `ServerRunContext`, or `ClientRunContexts` to create a `ClientRunContext`. Internally, the `BeanManager` is asked to provide a new instance of the `RunContext`, which allows you to replace the default implementation of a `RunContext` in an easy way. The factories declare two factory methods: `empty()` and `copyCurrent()`. Whereas `empty()` provides you an empty `RunContext`, `copyCurrent()` takes a snapshot of the current calling context and initializes the `RunContext` accordingly. That is useful if only some few values are to be changed, or, if using `ServerRunContext`, to run the code on behalf of a new transaction.

The following Listing 57 illustrates the creation of an empty `RunContext` initialized with a particular `Subject` and `Locale`.

Listing 56. Creation of an empty `RunContext`

```
Subject subject = new Subject(); ①
subject.getPrincipals().add(new SimplePrincipal("john"));
subject.setReadOnly();

②
RunContexts.empty()
    .withSubject(subject)
    .withLocale(Locale.US)
    .run(() -> {
      // run some code ③
      System.out.println(NlsLocale.CURRENT.get()); // > Locale.US
      System.out.println(Subject.getSubject(AccessController.getContext())); // > john
    });
```

① create the `Subject` to do some work on behalf

② Create and initialize the `RunContext`

③ This code is run on behalf of the `RunContext`

The following Listing 57 illustrates the creation of a 'snapshot' of the current calling `RunContext` with another `Locale` set.

Listing 57. Create a copy of the current calling `RunContext`

```
RunContexts.copyCurrent()
    .withLocale(Locale.US)
    .run(() -> {
      // run some code
    });
```

An important difference is related to the `RunMonitor`. By using the `copyCurrent()` factory method, the context's monitor is additionally registered as child monitor of the monitor of the current calling context. That way, a cancellation request to the calling context is propagated down to this context as well. Of course, that behavior can be overwritten by providing another monitor yourself.

10.2. Properties of a RunContext

The following properties are declared on a `RunContext` and are inherited by `ServerRunContext` and `ClientRunContext`.

property	description	accessibility
runMonitor	Monitor to query the cancellation status of the context. * must not be `null` * is automatically set if creating the context by its factory * is automatically registered as child monitor if creating the context by `copyCurrent()` factory method	RunMonitor.CURRENT.get()

property	description	accessibility
subject	Subject to run the code on behalf	Subject.getSubject(AccessController.getContext())
locale	Locale to be bound to the Locale `ThreadLocal`	NlsLocale.CURRENT.get()
propertyMap	Properties to be bound to the Property `ThreadLocal`	PropertyMap.CURRENT.get()

10.3. Properties of a ServerRunContext

A `ServerRunContext` controls propagation of server-side state and sets the transaction boundaries, and is a specialization of `RunContext`.

property	description	accessibility
session	Session to be bound to Session `ThreadLocal`	ISession.CURRENT.get()
transactionScope	To control transaction boundaries. By default, a new transaction is started, and committed or rolled back upon completion. * Use `TransactionScope.REQUIRES_NEW` to run the code in a new transaction (by default). * Use `TransactionScope.REQUIRED` to only start a new transaction if not running in a transaction yet. * Use `TransactionScope.MANDATORY` to enforce that the caller is already running in a transaction. Otherwise, a `TransactionRequiredException` is thrown.	ITransaction.CURRENT.get()
transaction	Sets the transaction to be used to run the runnable. Has only an effect, if transaction scope is set to TransactionScope.REQUIRED or TransactionScope.MANDATORY. Normally, this property should not be set manually.	ITransaction.CURRENT.get()

property	description	accessibility
clientNotificationCollector	To associate the context with the given ClientNotificationCollector, meaning that any code running on behalf of this context has that collector set in ClientNotificationCollector.CURRENT thread-local. That collector is used to collect all transactional client notifications, which are to be published upon successful commit of the associated transaction, and which are addressed to the client node which triggered processing (see withClientNodeId(String)). That way, transactional client notifications are not published immediately upon successful commit, but included in the client's response instead (piggyback). Typically, that collector is set by ServiceTunnelServlet for the processing of a service request.	ClientNotificationCollector.CURRENT.get()
clientNodeId	Associates this context with the given 'client node ID', meaning that any code running on behalf of this context has that id set in IClientNodeId.CURRENT thread-local. Every client node (that is every UI server node) has its unique 'node ID' which is included with every 'client-server' request, and is mainly used to publish client notifications. If transactional client notifications are issued by code running on behalf of this context, those will not be published to that client node, but included in the request's response instead (piggyback). However, transactional notifications are only sent to clients upon successful commit of the transaction. Typically, this node ID is set by ServiceTunnelServlet for the processing of a service request.	IClientNodeId.CURRENT.get()

10.4. Properties of a ClientRunContext

A `ClientRunContext` controls propagation of client-side state, and is a specialization of `RunContext`.

property	description	accessibility
session	Session to be bound to Session `ThreadLocal`	ISession.CURRENT.get()
form	Associates this context with the given `IForm`, meaning that any code running on behalf of this context has that `IForm` set in `IForm.CURRENT` thread-local. That information is mainly used to determine the current calling model context, e.g. when opening a message-box to associate it with the proper `IDisplayParent`. Typically, that information is set by the UI facade when dispatching a request from UI, or when constructing UI model elements.	IForm.CURRENT.get()

property	description	accessibility
outline	Associates this context with the given `IOutline`, meaning that any code running on behalf of this context has that `IOutline` set in `IOutline.CURRENT` thread-local. That information is mainly used to determine the current calling model context, e.g. when opening a message-box to associate it with the proper `IDisplayParent`. Typically, that information is set by the UI facade when dispatching a request from UI, or when constructing UI model elements.	IOutline.CURRENT.get()
desktop	Associates this context with the given `IDesktop`, meaning that any code running on behalf of this context has that `IDesktop` set in `IDesktop.CURRENT thread-local`. That information is mainly used to determine the current calling model context, e.g. when opening a message-box to associate it with the proper `IDisplayParent`. Typically, that information is set by the UI facade when dispatching a request from UI, or when constructing UI model elements.	IDesktop.CURRENT.get()

Chapter 11. RunMonitor

A RunMonitor allows the registration of `ICancellable` objects, which are cancelled upon cancellation of this monitor. A RunMonitor is associated with every RunContext and IFuture, meaning that executing code can always query its current cancellation status via `RunMonitor.CURRENT.get().isCancelled()`.

A `RunMonitor` itself is also of the type `ICancellable`, meaning that it can be registered within another monitor as well. That way, a monitor hierarchy can be created with support of nested cancellation. That is exactly what is done when creating a copy of the current calling context, namely that the new monitor is registered as `ICancellable` within the monitor of the current calling context. Cancellation only works top-down, and not bottom up, meaning that a parent monitor is not cancelled once a child monitor is cancelled.

When registering a `ICancellable` and this monitor is already cancelled, the `ICancellable` is cancelled immediately.

Furthermore, a job's Future is linked with the job's `RunMonitor`, meaning that cancellation requests targeted to the `Future` are also propagated to the `RunMonitor`, and vice versa.

The following Figure 3 illustrates the `RunMonitor` and its associations.

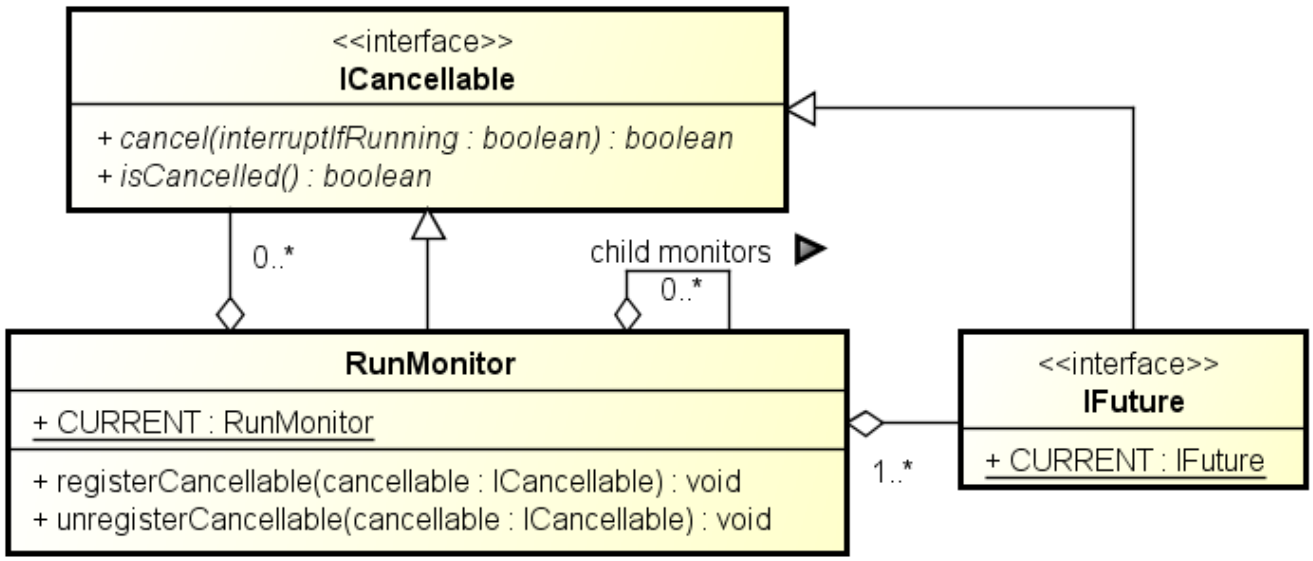


Figure 3. RunMonitor and its associations

Chapter 12. Client Notifications

In a scout application, typically, the scout client requests some data from the scout server. Sometimes, however, the communication needs go the other way: The scout server needs to inform the scout client about something. With client notifications it is possible to do so.

Figure 4. Client Notifications

12.1. Examples

Example scenarios for client notifications are:

- some data shared by client and server has changed (e.g. a cache on the client is no longer up-to-date, or a shared variable has changed)
- a new incoming phone call is available for a specific client and should be shown in the GUI
- a user wants to send a message to another user

Scout itself uses client notifications to synchronize code type and permission caches and session shared variables.

12.2. Data Flow

A client notification message is just a serializable object. It is published on the server and can be addressed either to all client nodes or only to a specific session or user. On the UI server side, handlers can be used to react upon incoming notifications.

Client notification handlers may change the state of the client model. In case of visible changes in the UI, these changes are automatically reflected in the UI.

In case of multiple server nodes, the client notifications are synchronized using cluster notifications to ensure that all UI servers receive the notifications.

12.3. Push Technology

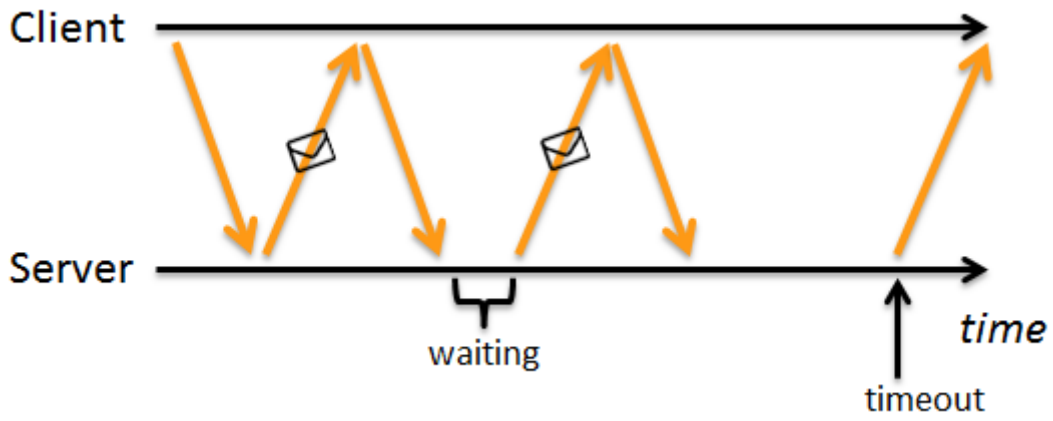


Figure 5. Long Polling

Client notifications are implemented using `long polling` as described below, because long polling works reliably in most corporate networks with proxy servers between server and client as well as with security policies that do not allow server push.

With long polling, the client requests notifications from the server repeatedly. If no new notifications are available on the server, instead of sending an empty response, the server holds the request open and waits until new notifications are available or a timeout is reached.

In addition to the long polling mechanism, pending client notifications are also transferred to the client along with the response of regular client requests.

12.4. Components

A client notification can be published on the server using the `ClientNotificationRegistry`. Publishing can be done either in a non-transactional or transactional way (only processed, when the transaction is committed).

The UI Server either receives the notifications via the `ClientNotificationPoller` or in case of transactional notifications together with the response of a regular service request. The notification is then dispatched to the corresponding handler.

When a client notifications is published on the server, it is automatically synchronized with the other server nodes (by default).

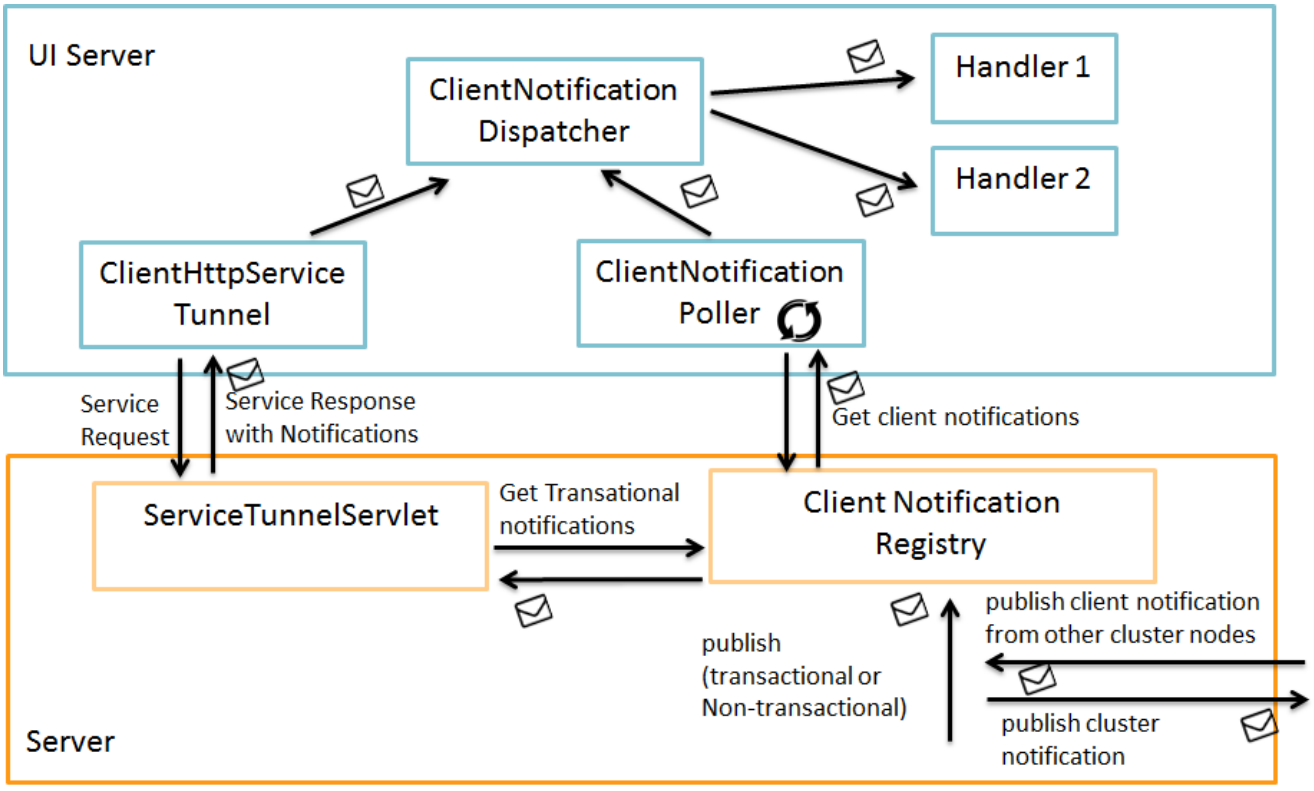


Figure 6. Client Notification Big Picture

12.4.1. Multiple Server Nodes

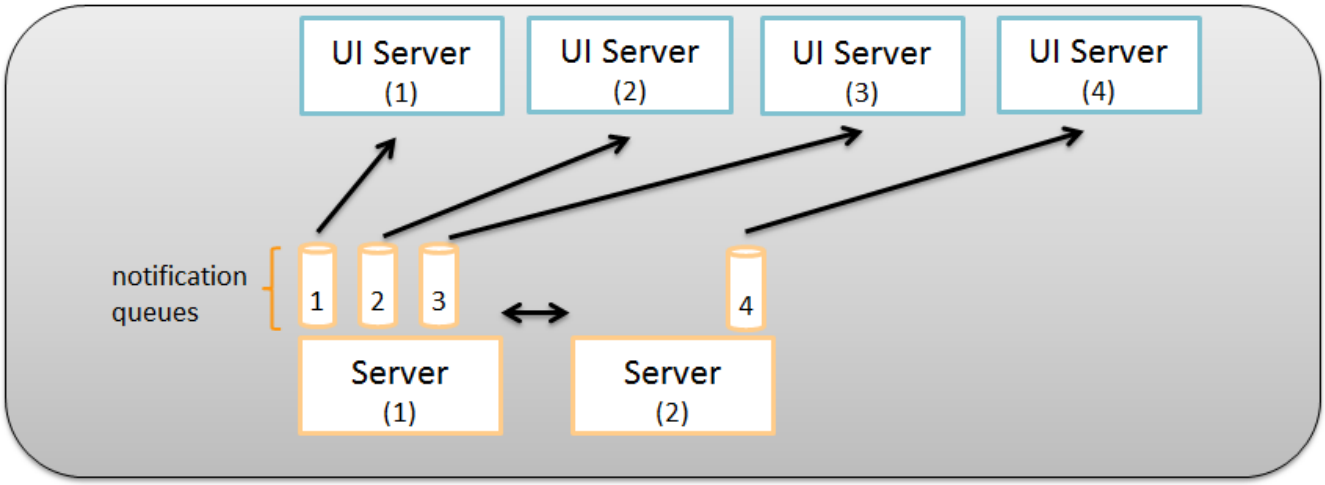


Figure 7. Client Notification Multiple Server Nodes

In order to deal with multiple ui-server nodes, the server holds a single notifications queue per ui-server node.

In this queues only the relevant notifications need to be kept: If a client notification is addressed to a session or user, that does not exist on a ui-server node, it is not added to the queue.

Sessions and corresponding users are registered on the server upon creation (and de-registered after destruction).

12.5. Publishing

Listing 58. Publishing Client Notifications

```
BEANS.get(ClientNotificationRegistry.class).putForUser("admin", new
PersonTableChangedNotification());
```

There are several options to choose from when publishing a new client notification:

12.5.1. ClientNotificationAddress

The `ClientNotificationAddress` determines which how the client notification needs to be dispatched and handled. A client notification can be addressed to

- all nodes
- all sessions
- one or more specific session
- one or more specific user

12.5.2. Transactional vs. Non-transactional

Client notifications can be published in a transactional or non-transactional way.

- Transactional means that the client notifications are only published once the transaction is committed. If the transaction fails, client notifications are disregarded.
- Non-transactional means that client notifications are published immediately without considering any transactions.

12.5.3. Distributing to all Cluster Nodes

Generally, it makes sense to distribute the client notifications automatically to all other server cluster nodes (if available). This is achieved using `ClusterNotifications`. It is however also possible to publish client notifications without cluster distribution. E.g. in case of client notifications already received from other cluster nodes.

12.5.4. Coalescing Notifications

It is possible that a service generates a lot of client notifications that are obsolete once a newer notification is created. In this case a coalescer can be created to reduce the notifications:

Listing 59. Client Notification Coalescer

```
public class BookmarkNotificationCoalescer implements ICoalescer
<BookmarkChangedClientNotification> {

  @Override
  public List<BookmarkChangedClientNotification> coalesce(List
<BookmarkChangedClientNotification> notifications) {
    // reduce to one
    return CollectionUtility.arrayList(CollectionUtility.firstElement(notifications));
  }
}
```

12.6. Handling

The `ClientNotificationDispatcher` is responsible for dispatching the client notifications to the correct handler.

12.6.1. Creating a Client Notification Handler

To create a new client notification handler for a specific client notification, all you need to do is creating a class implementing `org.eclipse.scout.rt.shared.notification.INotificationHandler<T>`, where `T` is the type (or subtype) of the notification to handle.

The new handler does not need to be registered anywhere. It is available via jandex class inventory.

Listing 60. Notification Handler for `MessageNotifications`

```
public class MessageNotificationHandler implements INotificationHandler
<MessageNotification> {

  @Override
  public void handleNotification(final MessageNotification notification) {
```

12.6.2. Handling Notifications Temporarily

Sometimes it is necessary to start and stop handling notification dynamically, (e.g. when a form is opened) in this case `AbstractObservableNotificationHandler` can be used to add and remove listeners.

12.6.3. Asynchronous Dispatching

Dispatching is always done asynchronously. However, in case of transactional notifications, a service call blocks until all transactional notifications returned with the service response are handled.

This behavior was implemented to simplify for example the usage of shared caches:

Listing 61. Blocking until notification handling completed

```
CodeService cs = BEANS.get(CodeService.class);
cs.reloadCodeType(UiThemeCodeType.class);

//client-side reload triggered by client notifications is finished
List<? extends ICode<String>> reloadedCodes = cs.getCodeType(UiThemeCodeType.class)
.getCodes();
```

In the example above, it is guaranteed, that the codetype is up-to-date as soon as reloadCodeType is finished.

12.6.4. Updating Scout Model

Notification handlers are never called from a scout model thread. If the scout model needs to be updated when handling notifications, a model job needs to be created for that task.

Listing 62. Notification Handler Creating Model Job

```
@Override
public void handleNotification(final MessageNotification notification) {
  ModelJobs.schedule(() -> {
    UserNodePage userPage = getUserNodePage();
    String buddy = notification.getSenderName();

    if (userPage != null) {
      ChatForm form = userPage.getChatForm(buddy);
      if (form != null) {
        form.getHistoryField().addMessage(false, buddy, form.getUserName(), new Date(
), notification.getMessage());
      }
    }
  }, ModelJobs.newInput(ClientRunContexts.copyCurrent()));
}
```

Make sure to always run updates to the scout models in a model job (forms, pages, ...): Use ModelJobs.schedule(...) where necessary in notification handlers.

Chapter 13. Extensibility

Required version: The API described here requires Scout version 4.2 or newer.

13.1. Overview

Since December 2014 and Scout 4.2 or newer a new extensibility concept is available for Scout. This article explains the new features and gives some examples how to use them.

When working with large business applications it is often required to split the application into several modules. Some of those modules may be very basic and can be reused in multiple applications. For those it makes sense to provide them as binary library. But what if you have created great templates for your applications but in one special case you want to include one more column in a table or want to execute some other code when a pre-defined context menu is pressed? You cannot just modify the code because it is a general library used everywhere. This is where the new extensibility concept helps.

To achieve this two new elements have been introduced:

- Extension Classes: Contains modifications for a target class. Modifications can be new elements or changed behavior of existing elements.
- Extension Registry: Service holding all Extensions that should be active in the application.

The Scout extensibility concept offers three basic possibilites to extend existing components:

- **Extensions** Changing behavior of a class
- **Contributions** Add new elements to a class
- **Moves** Move existing elements within a class

The following chapers will introduce this concepts and present some examples.

13.2. Extensions

Extensions contain modifications to a target class. This target class must be extensible. All elements that implement `org.eclipse.scout.rt.shared.extension.IExtensibleObject` are extensible. And for all extensible elements there exists a corresponding abstract extension class.

Examples:

- `AbstractStringField` is extensible. Therefore there is a class `AbstractStringFieldExtension`.
- `AbstractCodeType` is extensible. Therefore there is a class `AbstractCodeTypeExtension`.

Target classes can be all that are instanceof those extensible elements. This means an `AbstractStringFieldExtension` can be applied to `AbstractStringField` and all child classes.

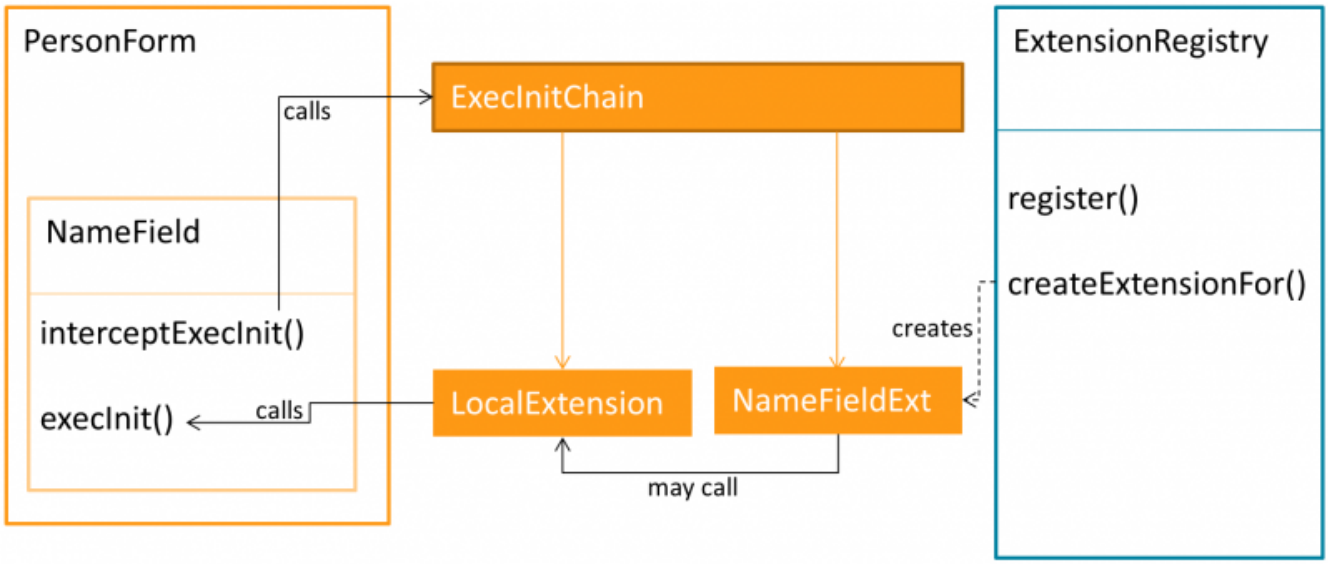
Extensions contain methods for all Scout Operations (see Exec Methods). Those methods have the same signature except that they have one more input parameter. This method allows you to

intercept the given Scout Operation and execute your own code even though the declaring class exists in a binary library. It is then your decision if you call the original code or completely replace it. To achieve this the Chain Pattern is used: All extensions for a target class are called as part of a chain. The order is given by the order in which the extensions are registered. And the original method of the Scout element is an extension as well.

Extensions to specific types of elements are prepared as abstract classes:

- AbstractGroupBoxExtension
- AbstractImageFieldExtension

The following image visualizes the extension chain used to intercept the default behavior of a component:



13.2.1. Extending a StringField example

The following example changes the initial value of a StringField called `NameField`:

Listing 63. Extension for NameField

```
public class NameFieldExtension extends AbstractStringFieldExtension<NameField> {

  public NameFieldExtension(NameField owner) {
    super(owner);
  }

  @Override
  public void execInitField(FormFieldInitFieldChain chain) {
    chain.execInitField(); // call the original exec init. whatever it may do.
    getOwner().setValue("FirstName LastName"); // overwrite the initial value of the
name field
  }
}
```

Note: The type parameter of the extension (e.g. `NameField`) denotes the element which is extended.

The extension needs to be registeres when starting the application:

Listing 64. Register extension for NameField

```
Jobs.schedule(() -> BEANS.get(IExtensionRegistry.class).register(NameFieldExtension
.class), Jobs.newInput()
    .withRunContext(ClientRunContexts.copyCurrent())
    .withName("register extension"));
```

13.3. Contributions

The section before explained how to modify the behavior of existing Scout elements. This section will describe how to contribute new elements into existing containers.

This is done by using the same mechanism as before. It is required to create an Extension too. But instead of overwriting any Scout Operation we directly define the new elements within the Extension. A lot of new elements can be added this way: Fields, Menus, Columns, Codes, ...

Some new elements may also require a new DTO (FormData, TablePageData, TableData) to be filled with data from the server. The corresponding DTO for the extension is automatically created when using the SDK 4.2 or newer and having the `@Data` annotation specified on your extension. As soon as the DTO extension has been registered in the `IExtensionRegistry` service it is automatically created when the target DTO is created and will also be imported and exported automatically!

The following example adds two new fields for salary and birthday to a `PersonForm`. Please note the `@Data` annotation which describes where the DTO for this extension should be created.

Listing 65. Extension for PersonForm

```
/**
 * Extension for the MainBox of the PersonForm
 */
@Data(PersonFormMainBoxExtensionData.class)
public class PersonFormMainBoxExtension extends AbstractGroupBoxExtension<MainBox> {

  public PersonFormMainBoxExtension(MainBox ownerBox) {
    super(ownerBox);
  }

  @Order(2000)
  public class SalaryField extends AbstractBigDecimalField {
  }

  @Order(3000)
  public class BirthdayField extends AbstractDateField {
  }
}
```

The extension data must be registered manually in the job like in the example before:

Listing 66. Register extension for PersonForm

```
BEANS.get(IExtensionRegistry.class).register(PersonFormMainBoxExtension.class);
```

Then the SDK automatically creates the extension DTO which could look as follows. Please note: The DTO is generated automatically but you have to register the generated DTO manually!

Listing 67. Extension Data for PersonForm

```
/**
 * <b>NOTE:</b><br>
 * This class is auto generated by the Scout SDK. No manual modifications recommended.
 */
@Extends(PersonFormData.class)
@Generated(value = "org.eclipse.scout.docs.snippets.person.PersonFormMainBoxExtension
", comments = "This class is auto generated by the Scout SDK. No manual modifications
recommended.")
public class PersonFormMainBoxExtensionData extends AbstractFormFieldData {

  private static final long serialVersionUID = 1L;

  public Birthday getBirthday() {
    return getFieldByClass(Birthday.class);
  }

  public Salary getSalary() {
    return getFieldByClass(Salary.class);
  }

  public static class Birthday extends AbstractValueFieldData<Date> {

    private static final long serialVersionUID = 1L;
  }

  public static class Salary extends AbstractValueFieldData<BigDecimal> {

    private static final long serialVersionUID = 1L;
  }
}
```

You can also access the values of the DTO extension as follows:

Listing 68. Access extended fields

```
// create a normal FormData
// contributions are added/imported/exported automatically
PersonFormData data = new PersonFormData();

// access the data of an extension
PersonFormMainBoxExtensionData c = data.getContribution(
PersonFormMainBoxExtensionData.class);
c.getSalary().setValue(new BigDecimal("200.0"));
```

13.3.1. Extending a form and a handler

Extending a AbstractForm and one (or more) of its AbstractFormHandlers that can be achieved as follows:

Listing 69. Extension for PersonForm

```
public class PersonFormExtension extends AbstractFormExtension<PersonForm> {

  public PersonFormExtension(PersonForm ownerForm) {
    super(ownerForm);
  }

  @Override
  public void execInitForm(FormInitFormChain chain) throws ProcessingException {
    chain.execInitForm();
    // Example logic: Access the form, disable field
    getOwner().getNameField().setEnabled(false, true, true);
  }

  public void testMethod() {
    MessageBoxes.create().withHeader("Extension method test").withBody("A method from
the form extension was called").show();
  }

  public static class NewFormHandlerExtension extends AbstractFormHandlerExtension
<NewHandler> {

    public NewFormHandlerExtension(NewHandler owner) {
      super(owner);
    }

    @Override
    public void execPostLoad(FormHandlerPostLoadChain chain) throws
ProcessingException {
      chain.execPostLoad();
      // Example logic: Show a message box after load
      MessageBoxes.create().withHeader("Extension test").withBody("If you can read
this, the extension works correctly").show();

      // Access element from the outer extension.
      PersonFormExtension extension = ((AbstractForm) getOwner().getForm())
.getExtension(PersonFormExtension.class);
      extension.testMethod();
    }
  }
}
```

There are a few things to note about this example:

- It is only necessary to register the outer form extension, not the inner handler extension as well.
- The inner handler extension must be `static`, otherwise an Exception will occur when the extended form is being started!
- You can access the element you are extending by calling `getOwner()`.

- Since you cannot access elements from your form extension directly from the inner handler extension (because it is static), you will need to retrieve the form extension via the `getExtension(Class<T extends IExtension<?>>)` method on the extended object, as done here to retrieve the form extension from the form handler extension.

13.4. Move elements

You can also move existing Scout elements to other positions. For this you have to register a move command in the IExtensionRegistry. As with all extension registration it is added to the extension registration Job in your Activator class:

Listing 70. Move NameField to LastBox

```
BEANS.get(IExtensionRegistry.class).registerMove(NameField.class, 20d, LastBox.class);
```

13.5. Migration

The new extensibility concept is added on top of all existing extension possibilities like injection or sub-classing. Therefore it works together with the current mechanisms. But for some use cases (like modifying template classes) it offers a lot of benefits. Therefore no migration is necessary. The concepts do exist alongside each others.

However there is one impact: Because the Scout Operation methods are now part of a call chain they may no longer be invoked directly. So any call to e.g. `execValidateValue()` is no longer allowed because this would exclude the extensions for this call. The Scout SDK marks such calls with error markers in the Eclipse Problems view. If really required the corresponding intercept-Method can be used. So instead directly calling `myField.execChangedValue` you may call `myField.interceptChangedValue()`.

Chapter 14. Mobile Support

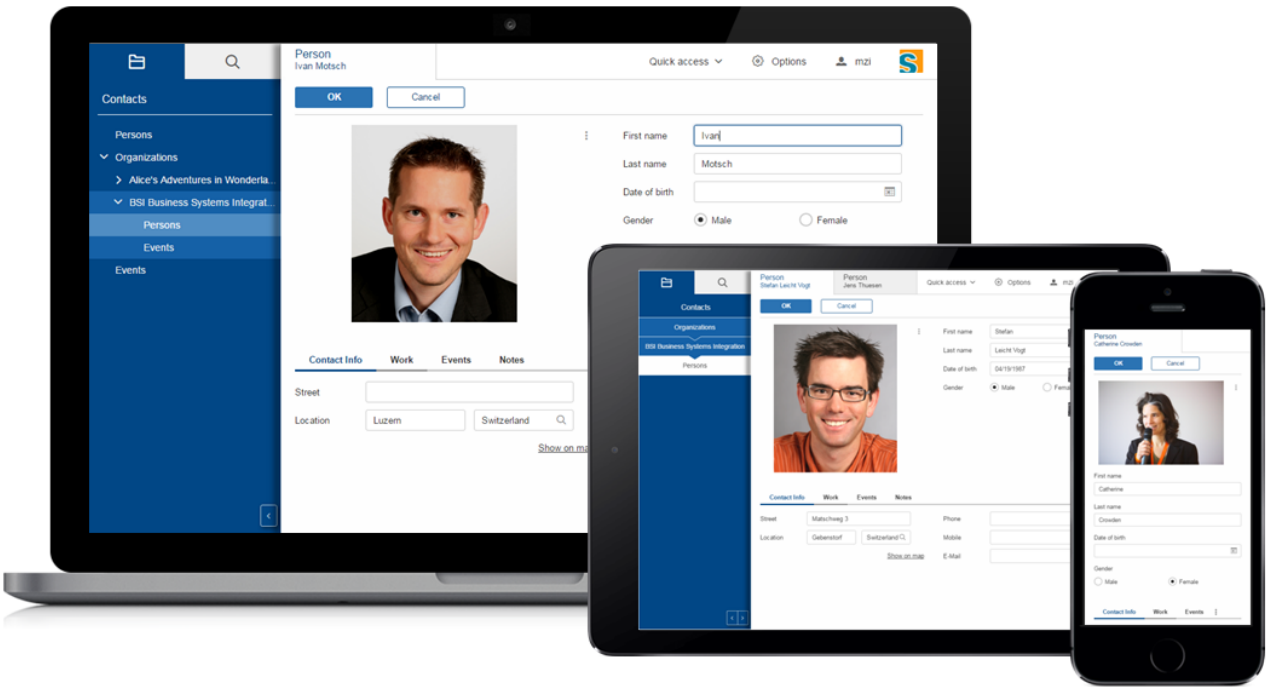


Figure 8. Scout apps run on desktops, tablets and mobile phones

Scout applications are mobile capable, meaning that they can be used on portable touch devices like smart phones and tablets. This capability is based on 2 main parts:

- Responsive and Touch Capable Widgets
- Device Transformation

14.1. Responsive and Touch Capable Widgets

Responsive design in context of a web application means that the design reacts to screen size changes. A Scout application does not use responsive design for the whole page, but many widgets itself may change the appearance when they don't fit into screen.

One example is the menu bar that stacks all menus which don't fit into an ellipsis menu.

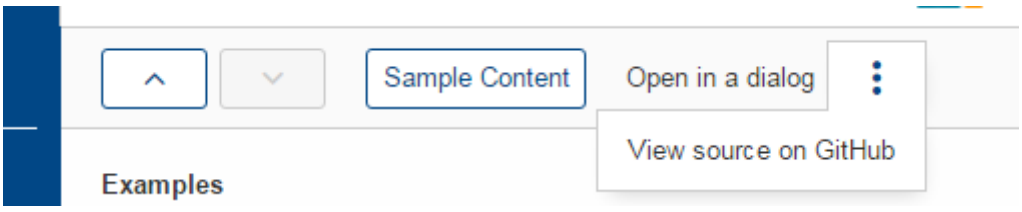


Figure 9. Responsive menu bar

Beside being responsive, the widgets may deal with touch devices as well. This means they are big enough to be used with the finger. And they don't need a mouse, especially the right mouse button.

One example is the tooltip of a form field which is reflected by an info icon on the right side of the field. Instead of hovering over the field the user can press that info icon to bring up the tooltip. This approach not only provides an indicator where tooltips are available, it also works for mouse and

touch based devices.

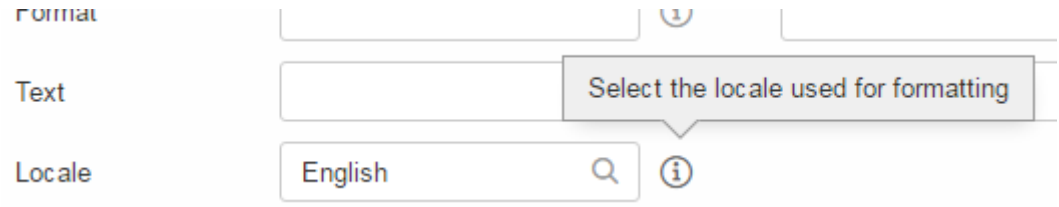


Figure 10. Touch friendly widgets

14.2. Device Transformation

The second part of the mobile support is called device transformation. Transformation means the adaptation of model properties for different devices. Example: Setting the property labelPosition from 'left' to 'top' in order to move the label to the top.

Such transformations are done on the UI server by so called device transformers. Currently 3 device transformers are available:

- Mobile Device Transformer
- Tablet Device Transformer
- Touch Device Transformer

Which transformer is active depends on the used user agent. The mobile transformer is active if the Scout app is used on a smart phone, the tablet one is active if it is used from a tablet, and the touch transformer is active in both cases. And may also be active if a desktop device supports touch. The order in which these transformers are processed is defined using the `@Order` annotation which is possible because they are regular Scout beans. This also means you can add your own transformer if you need custom transformations.

The transformations are mainly limited to the adjustment of properties, although some properties have a bigger effect than others. The property `displayStyle` of the desktop for example controls the look of the desktop and setting it to `COMPACT` rearranges the desktop in a mobile friendly way.

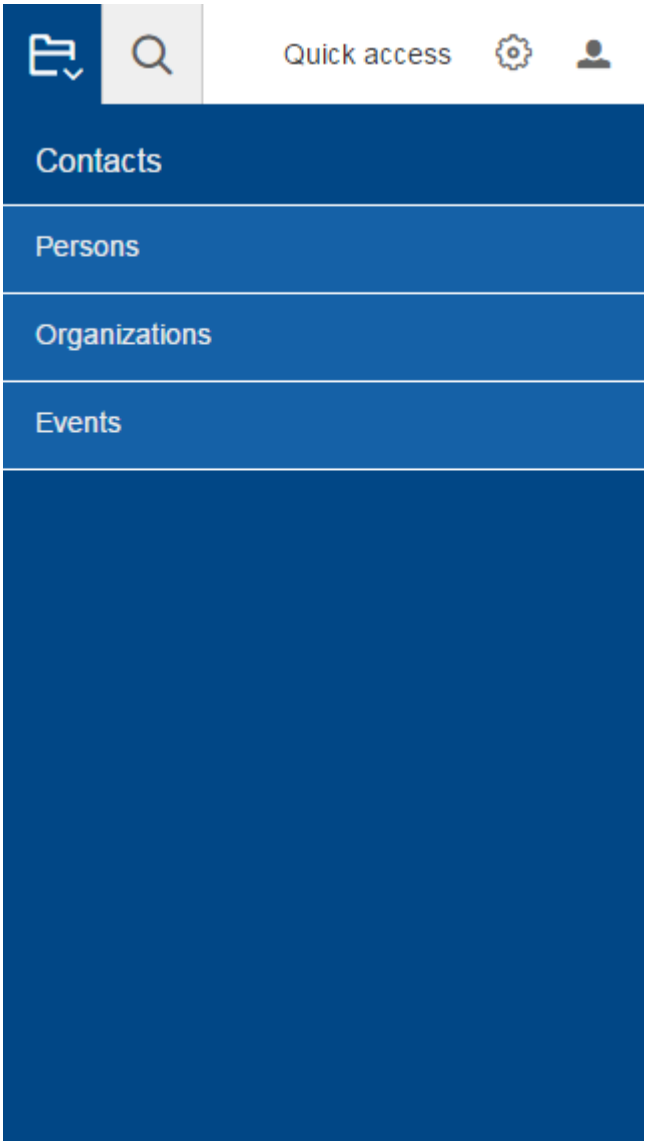


Figure 11. Desktop with displayStyle set to 'compact'

All the transformations are triggered by extensions to components like form fields or the desktop. These extensions are registered by `DeviceTransformationPlatformListener`. If you don't want any of these transformers to be active you could simply replace this listener and do nothing.

14.3. Adapt specific Components

The device transformers take care of global transformations which should be applied for most of the components. If you need to adapt a specific component you can do it at the component itself. Let's say you want to hide a field if the application is running on a smart phone, you could do the following.

```
@Order(20)
public class MyField extends AbstractStringField {

  @Override
  protected void execInitField() {
    if (UserAgentUtility.isMobileDevice()) {
      setVisibleGranted(false);
    }
  }
}
```

14.4. User Agent

The class `UserAgent` is essential for the mobile support. It stores information about the running device like the used browser or OS. The user agent is available on the UI server as well as on the backend server and can be accessed using the static method `UserAgent.get()`.

The class `UserAgentUtility` provides some useful helper methods to check which type of device is running, like if it's a mobile phone, a tablet, or a desktop device.

14.5. Best Practices

When creating a Scout application which should run on touch devices as well, the following tipps may help you.

1. Focus on the essential. Even though most of the application should run fine on a mobile device, some parts may not make sense. Identify those parts and make them invisible using `setVisibleGranted(false)`. The advantage of using setVisibleGranted over setVisible is that the model of the invisible components won't be sent to the client at all, which might increase the performance a little. But remember: The users nowadays might expect every functionality to be available even on a mobile phone, so don't take them away too much.
2. Limit the usage of custom HTML. Custom HTML cannot be automatically transformed, so you need to do it by yourself. Example: You created a table with several columns using HTML. On a small screen this table will be too large, so you have to make sure that your table is responsive, or provide other HTML code when running on a mobile device.
3. Don't use too large values for gridH. GridH actually is the minimum grid height, so if you set gridH to 10 the field will always be at least 10 logical grid rows height. This may be too big on a mobile device.
4. Use appropriate values for table column width. Tables are displayed the same way on a mobile phone as on the desktop device, if the content is not fully visible the user can scroll. If you have tables with `autoResizeColumns` set to true, you should make sure that the column widths are set properly. Just check how the table looks on a small screen and adjust the values accordingly.
5. Know the difference between small screens and touch capable. If you do checks against different device types, you should be aware that a touch device is not necessarily a small device. That means `UserAgentUtility.isTouchDevice()` may be true on a laptop as well, so use it with

care.

6. If you use filler fields for layouting purpose, make sure you use the official `IPlaceholderField`. Such filler fields normally waste space on a one column layout, so the mobile transformer will make them invisible.

Chapter 15. Security

15.1. Default HTTP Response Headers

All Scout HTTP servlets delegate to a central authority to append HTTP response headers. This is the bean `HttpServletControl`. It enables developers to control which headers that should be added to the HTTP response for each servlet and request.

The next sections describe the headers that are added to any response by default. Beside these also the following headers may be of interest for an end user application (consider adding them to your application if possible):

- Public Key Pinning
- Strict Transport Security (HSTS)
- X-Content-Type-Options

Please note that not all headers are supported in all user agents!

15.1.1. X-Frame-Options

The X-Frame-Options HTTP response header [3: https://developer.mozilla.org/en-US/docs/Web/HTTP/X-Frame-Options] can be used to indicate whether or not a user agent should be allowed to render a page in a `<frame>`, `<iframe>` or `<object>`. Sites can use this to avoid clickjacking [4: https://en.wikipedia.org/wiki/Clickjacking] attacks, by ensuring that their content is not embedded into other sites. The X-Frame-Options header is described in RFC 7034 [5: https://tools.ietf.org/html/rfc7034].

In Scout this header is set to `SAMEORIGIN` which allows the page to be displayed in a frame on the same origin (scheme, host and port) as the page itself only.

15.1.2. X-XSS-Protection

This header enables the XSS [6: https://en.wikipedia.org/wiki/Cross-site_scripting] filter built into most recent user agents. It's usually enabled by default anyway, so the role of this header is to re-enable the filter for the website if it was disabled by the user. The X-XSS-Protection header is described in controlling-the-xss-filter.

In Scout this header is configured to enable XSS protections and instructs the user-agent to block a page from loading if reflected XSS is detected.

15.1.3. Content Security Policy

Content Security Policy is a HTTP response header that helps you reduce XSS risks on modern user agents by declaring what dynamic resources are allowed to load [7: http://content-security-policy.com/]. The CSP header is described in Level 1 and Level 2. There is also a working draft for a Level 3.

Scout makes use of Level 1 (and one directive from Level 2) and sets by default the following settings:

- JavaScript [8: https://en.wikipedia.org/wiki/JavaScript]: Only accepts JavaScript resources from the same origin (same scheme, host and port). Inline JavaScript is allowed and unsafe dynamic code evaluation (like `eval(string)`, `setTimeout(string)`, `setInterval(string)`, `new Function(string)`) is allowed as well.
- Stylesheets (CSS) [9: https://en.wikipedia.org/wiki/Cascading_Style_Sheets]: Only accepts Stylesheet resources from the same origin (same scheme, host and port). Inline style attributes are allowed.
- Frames [10: https://en.wikipedia.org/wiki/Framing_(World_Wide_Web)]: All sources are allowed because the iframes created by the Scout BrowserField run in the sandbox mode and therefore handle the security policy on their own.
- All other types (Image, `WebSocket` [11: https://en.wikipedia.org/wiki/WebSocket], `EventSource` [12: https://developer.mozilla.org/en-US/docs/Web/API/EventSource], AJAX calls [13: https://en.wikipedia.org/wiki/Ajax_%28programming%29], fonts, `<object>` [14: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/object], `<embed>` [15: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/embed], `<applet>` [16: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/applet], `<audio>` [17: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/audio] and `<video>` [18: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video]) only allow resources from the same origin (same scheme, host and port).

If a resource is blocked because it violates the CSP a report is created and logged on server side using level `warning`. This is done in the class `ContentSecurityPolicyReportHandler`. This enables admins to monitor the application and to react if a CSP violation is detected.

15.2. Session Cookie (JSESSIONID Cookie) Configuration Validation

The UiServlet checks if the session cookie is configured safely. The validation is only performed on first access to the `UiServlet`. There is no automatic validation on the backend server side or on any custom servlets!

If the validation fails, a corresponding error message is logged to the server and an exception is thrown making the `UiServlet` inaccessible. Because of security reasons the exception shown to the user includes no details about the error. These can only be seen on the server side log.

15.2.1. HttpOnly

First the existence of the `HttpOnly` flag is checked. The servlet container will then add this flag to the `Set-Cookie` HTTP response header. If the user agent supports this flag, the cookie cannot be accessed through a client side script. As a result even if a cross-site scripting (XSS) flaw exists and a user accidentally accesses a link that exploits this flaw, the user agent will not reveal the cookie to a third party. For a list of user agents supporting this feature please refer to OWASP.

It is recommended to always enable this flag.

Since Java Servlet 3.0 specification this property can be set in the configuration in the deployment descriptor `WEB-INF/web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">
  ...
  <session-config>
    ...
    <cookie-config>
      <http-only>true</http-only> ①
      ...
    </cookie-config>
    ...
  </session-config>
  ...
</web-app>
```

① The HttpOnly flag activated

15.2.2. Secure

Second the existence of the `Secure` flag is checked. The servlet container will then add this flag to the `Set-Cookie` HTTP response header. The purpose of the secure flag is to prevent cookies from being observed by unauthorized parties due to the transmission of a the cookie in clear text. Therefore setting this flag will prevent the user agent from transmitting the session id over an unencrypted channel.

Since Java Servlet 3.0 specification this property can be set in the configuration in the deployment descriptor `WEB-INF/web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">
  ...
  <session-config>
    ...
    <cookie-config>
      <secure>true</secure> ①
      ...
    </cookie-config>
    ...
  </session-config>
  ...
</web-app>
```

① The Secure flag activated

This of course only makes sense if the application is exposed to the end user using an encrypted channel like HTTPS (which is strongly recommended).

Unfortunately for the UI server it is not possible to detect if an application uses a secured channel. Consider the following example: The servlet container is protected by a reverse proxy. The communication between the user agent and the proxy is encrypted while the channel between the proxy and the servlet container is not. In this scenario the container cannot know that from a user agent point of view the channel is secured.

Because of this the validation assumes that the channel from the user agent to the entering node is secured and by default checks for the `Secure` flag. In case this assumption is not true and an unencrypted channel must be used this validation step can be disabled by setting the following property in the `config.propertis` file:

```
scout.auth.cookie.session.validate.secure=false
```

This skips the `Secure` flag check completely. In this scenario (not using https) it is also required to remove the secure tag from the cookie config in the `WEB-INF/web.xml`.

15.3. Secure Output

This chapter describes how HTML Output can be handled in a secure way.

Scout applications often display potentially dangerous data, e.g. user input or data from other systems. Encoding this input in such a way, that it can not be executed, prevents security vulnerabilities like cross-site scripting.

15.3.1. Encoding by Default

By default, all input in the Scout model is encoded. Examples are values/labels in value fields, cells in tables, message in message box. The reason behind this default choice is that developers do not have to think about output encoding in the standard case and are therefore less likely to forget output encoding and introduce a security vulnerability.

Example: In the following label field, the HTML tag is encoded as `<b>bold text</b>`:

bold text

```
public class LabelField extends AbstractLabelField {
  @Override
  protected void execInitField() {
    setValue("...<b>Bold text</b>...");
  }
```

15.3.2. Html Enabled

Sometimes developers may want to use HTML in the Scout model.

Examples are

- Simple styling of dynamic content, such as addresses or texts in message boxes
- Text containing application-internal or external links
- Html or XML content received from other systems, such as e-mails or html pages

Html input should only partially be encoded or not at all.

To disable the encoding of the whole value, the property `HtmlEnabled` can be used:

```
public class NoEncodingLabelField extends AbstractLabelField {
  @Override
  protected boolean getConfiguredHtmlEnabled() {
    return true;
  }
```

```
@Override
protected void execInitField() {
  setValue("...<b>Bold text</b>...");
}
```

There are several ways to implement the use cases above. Some typical implementations are described in the following sections.

CSS Class and Other Model Properties

Often using HTML in value fields or table cells is not necessary for styling. Very basic styling can be done for example by setting the CSS class.

HTML Builder

For creating simple HTML files or fragments with encoded user input, the class `org.eclipse.scout.rt.platform.html.HTML` can be used. It is also easily possible to create application internal and external link with this approach.

Styling in the UI-Layer

For more complex HTML, using `IBeanField` in the scout model and implementing the styling in the UI-Layer is often the preferred way. Links are possible as well.

Manual Encoding

It is also possible to encode any String manually using `StringUtility.htmlEncode(String)`. `org.eclipse.scout.rt.platform.html.HTML` uses this method internally for encoding. However, using `HTML` is recommended, where possible, because it is more concise and leads to less errors.

Using a White-List Filter

If HTML or XML from external sources or more complex HTML are used in the Scout model, using a white-list filter might be the best way to avoid security bugs. Libraries, such as JSoup provide such a white-list filter. Scout currently does not include any services or utilities for using white-list filters, because the configuration and usage is very use-case-specific and would therefore not add much benefit.

Chapter 16. Webservices with JAX-WS

The Java API for XML-Based Web Services (JAX-WS) is a Java programming language API for creating web services. JAX-WS is one of the Java XML programming APIs, and is part of the Java EE platform.

Scout facilitates working with webservices, supports you in the generation of artifacts, and provides the following functionality:

16.1. Functionality

- ready to go Maven profile for easy webservice stub and artifact generation
- full JAX-WS 2.2 compliance
- JAX-WS implementor independence
- provides an up front port type *EntryPoint* to enforce for authentication, and to run web requests in a RunContext
- adds cancellation support for on-going webservice requests
- provides a port cache for webservice consumers
- allows to participate in 2PC protocol for webservice consumers
- allows to provide 'init parameters' to handlers

16.2. JAX-WS implementor and deployment

16.2.1. JAX-WS version and implementor

The JAX-WS Scout integration provides a thin layer on top of JAX-WS implementors to facilitate working with webservices. It depends on the JAX-WS 2.2.x API as specified in JSR 224. It is implementor neutral, and was tested with with the following implementations:

- *JAX-WS RI* (reference implementation) as shipped with Java 7 and Java 8
- *JAX-WS METRO* (2.2.10)
- *Apache CXF* (3.1.3)

The integration does not require you to bundle the JAX-WS implementor with your application, which is a prerequisite for running in an EE container.

16.2.2. Running JAX-WS in a servlet container

A servlet container like Apache Tomcat typically does not ship with a JAX-WS implementor. As the actual implementor, you can either use *JAX-WS RI* as shipped with the JRE, or provide a separate implementor like *JAX-WS METRO* or *Apache CXF* in the form of a Maven dependency. However, *JAX-WS RI* does not provide a servlet based entry point, because the Servlet API is not part of the Java SE specification.

When publishing webservices, it therefore is easiest to ship with a separate implementor: Declare a respective Maven dependency in your webbapp project - that is the Maven module typically containing the application's *web.xml*.

16.2.3. Running JAX-WS in a EE container

When running in an EE container, the container typically ships with a JAX-WS implementor. It is highly recommended to use that implementor, primarily to avoid classloading issues, and to further profit from the container's monitoring and authentication facility. Refer to the containers documentation for more information.

16.2.4. Configure JAX-WS implementor

JAX-WS Scout integration is prepared to run with different implementors. Unfortunately, some implementors do not implement the JSR exactly, or some important functionality is missing in the JSR. To address this fact without loosing implementor independence, the delegate bean JaxWsImplementorSpecifics exists.

As of now, Scout ships with three such implementor specific classes, which are activated via *config.properties* by setting the property *jaxws.implementor* with its fully qualified class name. By default, *JAX-WS METRO* implementor is installed.

For instance, support for *Apache CXF* implementor is activated as following:

```
jaxws.implementor=org.eclipse.scout.rt.server.jaxws.implementor.JaxWsCxfSpecifics
```

class	description
JaxWsRISpecifics	implementor specifics for *JAX-WS Reference Implementation (RI)* as contained in JRE
JaxWsMetroSpecifics	implementor specifics for *JAX-WS METRO* implementation
JaxWsCxfSpecifics	implementor specifics for *Apache JAX-WS CXF* implementation

Of course, other implementors can be used as well. For that to work, install your own *JaxWsImplementorSpecifics* class, and reference its fully qualified name in *config.properties*.

JaxWsImplementorSpecifics

This class encapsulates functionality that is defined in JAX-WS JSR 224, but may diverge among JAX-WS implementors. As of now, the following points are addressed:

- missing support in JSR to set socket connect and read timeout;
- proprietary 'property' to set response code in *Apache CXF*;
- when working with *Apache CXF*, response header must be set directly onto Servlet Response, and not via `MessageContext`;
- when working with *JAX-WS METRO* or *JAX-WS RI*, the handler's return value is ignored in one-way communication; instead, the chain must be exited by throwing a webservice exception;

Learn more about how to configure a JAX-WS implementor: Section 16.2.4

Configure JAX-WS Maven dependency in pom.xml

The effective dependency to the JAX-WS implementor is to be specified in the pom.xml of the webapp module (not the server module). That allows for running with a different implementor depending on the environment, e.g. to provide the implementor yourself when starting the application from within your IDE in Jetty, or to use the container's implementor when deploying to an EE enabled application server. Even if providing the very same implementor for all environments yourself, it is good practice to do the configuration in the webapp module.

A generally applicable configuration cannot be given, because the effective configuration depends on the implementor you choose, and whether it is already shipped with the application server you use. However, if *JAX-WS RI* is sufficient, you do not have to specify an implementor at all because already contained in JRE.

If running in an EE application server, refer to the containers documentation for more information.

Listing 71 provides sample configuration for shipping with *JAX_WS METRO* and Listing 72 does the same for *Apache CXF*

Listing 71. Maven dependency for JAX-WS METRO

```
<!-- JAX-WS METRO not bundled with JRE -->
<dependency>
  <groupId>com.sun.xml.ws</groupId>
  <artifactId>jaxws-rt</artifactId>
  <version>...</version>
</dependency>
```

Listing 72. Maven dependency for Apache CXF

```
<!-- JAX-WS Apache CXF -->
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-frontend-jaxws</artifactId>
  <version>...</version>
</dependency>
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http</artifactId>
  <version>...</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>...</version>
</dependency>
```

Configure JAX-WS servlet in web.xml

This section describes the configuration of the entry point Servlet to publish webservices. If working with webservice consumers only, no configuration is required.

Similar to the pom.xml as described in Section 16.2.4.2, the *web.xml* differs from implementor to implementor, and whether the implementor is already shipped with the application server. Nevertheless, the following Listing 73 show a sample configuration for *JAX-WS METRO* and Listing 74 for *Apache CXF*.

Listing 73. web.xml for JAX-WS METRO Servlet

```
<!-- JAX-WS METRO not bundled with JRE -->
<context-param>
  <param-name>com.sun.xml.ws.server.http.publishStatusPage</param-name>
  <param-value>true</param-value>
</context-param>
<context-param>
  <param-name>com.sun.xml.ws.server.http.publishWSDL</param-name>
  <param-value>true</param-value>
</context-param>
<listener>
  <listener-class>
com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listener-class>
</listener>
<servlet>
  <servlet-name>jaxws</servlet-name>
  <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>jaxws</servlet-name>
  <url-pattern>/jaxws/*</url-pattern> ①
</servlet-mapping>
```

① the base URL where to publish the webservice endpoints

Listing 74. web.xml for Apache CXF Servlet

```
<!-- JAX-WS Apache CXF -->
<servlet>
  <display-name>CXF Servlet</display-name>
  <servlet-name>jaxws</servlet-name>
  <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
  <init-param>
    <param-name>config-location</param-name>
    <param-value>/WEB-INF/cxf-jaxws.xml</param-value> ①
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>jaxws</servlet-name>
  <url-pattern>/jaxws/*</url-pattern> ②
</servlet-mapping>
```

① *Apache CXF* specific configuration file for endpoints to be published. See Section 16.5.8.2 for more information.

② the base URL where to publish the webservice endpoints

But, if running in an EE container, it is most likely that a Servlet configuration must not be configured, because the endpoints are discovered by the application server, or registered in a vendor specific way. Refer to the containers documentation for more information.

Some application servers like Oracle WebLogic Server (WLS) allow the port types to be registered as a Servlet in web.xml. However, this is vendor specific, and works despite the fact that port type does not implement 'javax.servlet.Servlet'.

Do not forget to exclude the webservice's Servlet URL pattern from authentication filter.

16.3. Modularization

Scout JAX-WS integration does not prescribe how to organize your webservices in terms of Maven modules. You could either put all your webservices directly into the server module, or create a separate *jaxws* module containing all webservices, or even create a separate *jaxws* module for each webservice. Most often, the second approach of a single, separate *jaxws* module, which the server module depends on, is chosen.

This is mainly because of the following benefits:

- annotation processing must not be enabled for the entire server module
- one module to build all webservice artifacts at once
- easier to work with shared element types among webservices

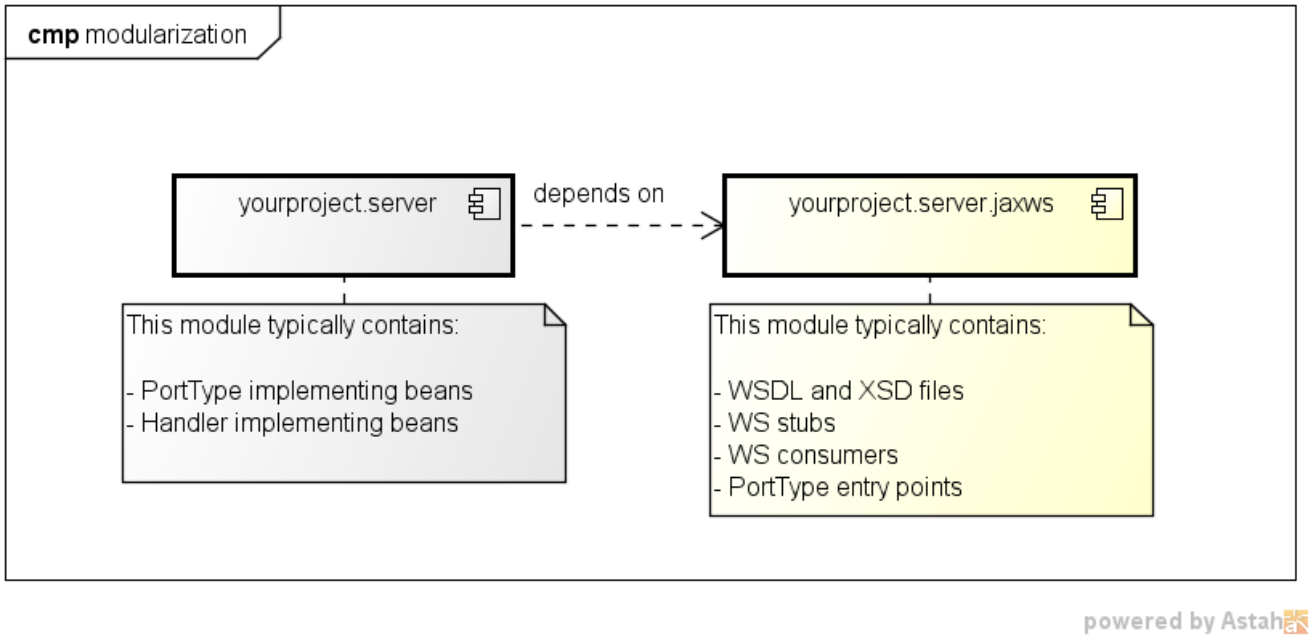


Figure 12. typical modularization

It is important to note, that the *server* depends on the *jaxws* module, and not vice versa. The *jaxws* module is primarily of technical nature, meaning that it knows how to generate its WS artifacts, and also contains those. However, implementing port type beans and even implementing handler beans are typically put into the server module to the access service and database layer. On the other hand, WS clients may be put into *jaxws* module, because they rarely contain any project specific business logic.

You may ask yourself, how the *jaxws* module can access the implementing port type and handlers located in the *server* module. That works because of the indirection via bean manager, and because there is a flat classpath at runtime.

See WebServiceEntryPoint for more information.

16.4. Build webservice stubs and artifacts

16.4.1. Configure webservice stub generation via wsimport

The Maven plugin 'org.codehaus.mojo:jaxws-maven-plugin' with the goal 'wsimport' is used to generate a webservice stub from a WSDL file and its referenced XSD schema files. If your Maven module inherits from the Scout module 'maven_rt_plugin_config-master', the 'jaxws' profile is available, which activates automatically upon the presence of a 'WEB-INF/wsdl' folder. Instead of inheriting from that module, you can alternatively copy the 'jaxws' profile into your projects parent POM module.

This profile is for convenience purpose, and provides a ready-to-go configuration to generate webservice stubs and webservice provider artifacts. It configures the 'jaxws-maven-plugin' to look for WSDL and XSD files in the folder 'src/main/resources/WEB-INF/wsdl', and for binding files in the folder '/src/main/resources/WEB-INF/binding'. Upon generation, the stub will be put into the folder 'target/generated-sources/wsimport'.

The profiles requires the Scout runtime version to be specified, and which is used to refer to

`org.eclipse.scout.jaxws.apt` module to generate webservice provider artifacts. However, this version is typically defined in pom.xml of the parent module, because also used to refer to other Scout runtime artifacts.

Listing 75. Scout version defined as Maven property

```
<properties>
  <org.eclipse.scout.rt.version>5.2.0-SNAPSHOT</org.eclipse.scout.rt.version>
</properties>
```

If your project design envisions a separate JAR module per WSDL, you simply have to set the property 'jaxws.wsdl.file' with the name of your WSDL file in the module's pom.xml (example in Listing 76).

Listing 76. wsimport configuration in pom.xml if working with a single WSDL file per JAR module

```
<properties>
  <jaxws.wsdl.file>YourWebService.wsdl</jaxws.wsdl.file> ①
</properties>
```

① name of the wsdl file

Otherwise, if having multiple WSDL files in your JAR module, some little more configuration is required, namely a respective execution section per WSDL file. Thereby, the 'id' of the execution section must be unique. Scout 'jaxws' profile already provides one such section, which is used to generate the stub for a single WSDL file (see such configuration in Listing 76), and names it 'wsimport-1'. It is simplest to name the subsequent execution sections 'wsimport-2', 'wsimport-3', and so on.
For each execution section, you must configure its unique *id*, the *goal* 'wsimport', and in the configuration section the respective *wsdlLocation* and *wsdlFile*. For 'wsimport' to work, *wsdlLocation* is not required. However, that location will be referenced in generated artifacts to set the wsdl location via `@WebService` and `@WebServiceClient`. The complete configuration is presented in Listing 77.

If you decide to configure multiple WSDL files in your POM as described in Listing 77, the configuration defined in the parent POM (`maven_rt_plugin_config-master`) and expecting a configuration as presented in Listing 76 needs to be overridden, therefore one of your execution id needs to be `wsimport-1`.

Listing 77. wsimport configuration in pom.xml if working with multiple WSDL files per JAR module

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jaxws-maven-plugin</artifactId>
      <executions>
        <!-- YourFirstWebService.wsdl -->
        <execution> ①
          <!-- DO NOT CHANGE THE ID: 'wsimport-1';
               it overrides an execution defined in the parent pom -->
          <id>wsimport-1</id> ②
          <goals>
            <goal>wsimport</goal> ③
          </goals>
          <configuration>
            <wsdlLocation>WEB-INF/wsdl/YourFirstWebService.wsdl</wsdlLocation> ④
            <wsdlFiles>
              <wsdlFile>YourFirstWebService.wsdl</wsdlFile> ⑤
            </wsdlFiles>
          </configuration>
        </execution>

        <!-- YourSecondWebService.wsdl -->
        <execution> ⑥
          <id>wsimport-2</id>
          <goals>
            <goal>wsimport</goal>
          </goals>
          <configuration>
            <wsdlLocation>WEB-INF/wsdl/YourSecondWebService.wsdl</wsdlLocation>
            <wsdlFiles>
              <wsdlFile>YourSecondWebService.wsdl</wsdlFile>
            </wsdlFiles>
          </configuration>
        </execution>

        ...

      </executions>
    </plugin>
  </plugins>
</build>
```

① declare an execution section for each WSDL file

② give the section a unique id (wsimport-1, wsimport-2, wsimport-3, ...)

③ specify the goal 'wsimport' to build the webservice stub

④ specify the project relative path to the WSDL file

⑤ specify the relative path to the WSDL file (relative to 'WEB-INF/wsdl')

⑥ declare an execution section for the next WSDL file

Further, you can overwrite any configuration as defined by 'jaxws-maven-plugin'. See http://www.mojohaus.org/jaxws-maven-plugin/ for supported configuration properties.

Also, it is good practice to create a separate folder for each WSDL file, which also contains all its referenced XSD schemas. Then, do not forget to change the properties *wsdlLocation* and *wsdlFile* accordingly.

16.4.2. Customize WSDL components and XSD schema elements via binding files

By default, all XML files contained in folder 'WEB-INF/binding' are used as binding files. But, most often, you will have a global binding file, which applies to all your WSDL files, and some custom binding files different per WSDL file and XSD schema files. See how to explicitly configure binding files in Listing 78.

Listing 78. explicit configuration of binding files

```
<!-- YourFirstWebService.wsdl -->
<execution>
  ...
  <configuration>
    ...
    <bindingFiles>
      <bindingFile>global-bindings.xml</bindingFile> ①
      <bindingFile>your-first-webservice-ws-bindings.xml</bindingFile> ②
      <bindingFile>your-first-webservice-xs-bindings.xml</bindingFile> ③
    </bindingFiles>
  </configuration>
</execution>

<!-- YourSecondWebService.wsdl -->
<execution>
  ...
  <configuration>
    ...
    <bindingFiles>
      <bindingFile>global-bindings.xml</bindingFile> ①
      <bindingFile>your-second-webservice-ws-bindings.xml</bindingFile> ②
      <bindingFile>your-second-webservice-xs-bindings.xml</bindingFile> ③
    </bindingFiles>
  </configuration>
</execution>
```

① global binding file which applies to all XSD schema elements. See Listing 79 for an example.

② custom binding file to customize the webservice's WSDL components in the namespace *http://java.sun.com/xml/ns/jaxws*. See Listing 80 for an example.

③ custom binding file to customize the webservice's XSD schema elements in the namespace *http://java.sun.com/xml/ns/jaxb*. See Listing 81 for an example.

With binding files in place, you can customize almost every WSDL component and XSD element that can be mapped to Java, such as the service endpoint interface class, packages, method name, parameter name, exception class, etc.

The global binding file typically contains some customization for common data types like *java.util.Date* or *java.util.Calendar*, whereas the custom binding files are specific for a WSDL or XSD schema. See Section 16.7.

Listing 79. example of global binding file in the namespace http://java.sun.com/xml/ns/jaxb

```
<bindings version="2.0"
  xmlns="http://java.sun.com/xml/ns/jaxb"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc">
  <globalBindings>
    <xjc:javaType
      name="java.util.Date"
      xmlType="xsd:date"
      adapter="org.eclipse.scout.rt.server.jaxws.adapter.UtcDateAdapter" />
    <xjc:javaType
      name="java.util.Date"
      xmlType="xsd:time"
      adapter="org.eclipse.scout.rt.server.jaxws.adapter.UtcTimeAdapter" />
    <xjc:javaType
      name="java.util.Date"
      xmlType="xsd:dateTime"
      adapter="org.eclipse.scout.rt.server.jaxws.adapter.UtcDateTimeAdapter" />
  </globalBindings>
</bindings>
```

By default, generated artifacts are put into the package corresponding to the element's namespace. Sometimes, you like to control the package names, but you want to do that on a per-namespace basis, and not put all the artifacts of a webservice into the very same package. That is mainly to omit collisions, and to have artifacts shared among webservices not duplicated.

Two separate binding files are required to customize WSDL components and XSD schema elements. That is because WSDL component customization is to be done in 'jaxws' namespace http://java.sun.com/xml/ns/jaxws, whereas XSD schema element customization in 'jaxb' namespace http://java.sun.com/xml/ns/jaxb.

Listing 80. example of jaxws component customization in the namespace http://java.sun.com/xml/ns/jaxws

```
<!-- binding to customize webservice components
(xmlns=http://java.sun.com/xml/ns/jaxws) -->
<bindings xmlns="http://java.sun.com/xml/ns/jaxws"> ①
    <package name="org.eclipse.ws.yourfirstwebservice"/> ②
</bindings>
```

① customization via jaxws namespace: http://java.sun.com/xml/ns/jaxws

② instructs to put all webservice components (port type, service) into package *org.eclipse.ws.yourfirstwebservice*

Listing 81. example of xsd schema element customization in the namespace http://java.sun.com/xml/ns/jaxb

```
<!-- binding to customize xsd schema elements (xmlns=http://java.sun.com/xml/ns/jaxb)
-->
<bindings xmlns="http://java.sun.com/xml/ns/jaxb" version="2.1"> ①
  <!-- namespace http://eclipse.org/public/services/ws/soap  -->
  <bindings scd="x-schema::tns" xmlns:tns="http://eclipse.org/public/services/ws/soap
">
    <schemaBindings>
      <package name="org.eclipse.ws.yourfirstwebservice" /> ②
    </schemaBindings>
  </bindings>

  <!-- namespace http://eclipse.org/public/services/ws/common/soap -->
  <bindings scd="x-schema::tns" xmlns:tns=
"http://eclipse.org/public/services/ws/common/soap">
    <schemaBindings>
      <package name="org.eclipse.ws.common" /> ③
    </schemaBindings>
  </bindings>
</bindings>
```

① customization via jaxb namespace: http://java.sun.com/xml/ns/jaxb

② instructs to put all XSD schema elements in namespace *http://eclipse.org/public/services/ws/soap* into package *org.eclipse.ws.yourfirstwebservice*

③ instructs to put all XSD schema elements in namespace *http://eclipse.org/public/services/ws/common/soap* into package *org.eclipse.ws.common*

wsimport allows to directly configure the package name for files to be generated (packageName). However, this is discouraged, because all artifacts are put into the very same package. Use package customization on a per-namespace basis instead.

For shared webservice artifacts, you can also use XJC binding compiler to generate those artifacts in advance, and then provide the resulting episode binding file (META-INF/sun-jaxb.episode) to *wsimport*. See http://www.mojohaus.org/jaxb2-maven-plugin/Documentation/v2.2/xjc-mojo.html for more information.

16.4.3. Annotation Processing Tool (APT)

Annotation Processing (APT) is a tool which can be enabled to fire for annotated types during compilation. In JAX-WS Scout integration, it is used as a trigger to generate webservice port type implementations. Such an auto-generated port type implementation is called an entry point. It is to be published as the webservice's endpoint, and acts as an interceptor for webservice requests. It optionally enforces for authentication, and makes the request to be executed in a RunContext. Then, it handles the web request to the effectively implementing port type bean for actual processing.

The entry point generated simplifies the actual port type implementation by removing lot of glue code to be written by hand otherwise. Of course, this entry point is just for convenience purpose, and it is up to you to make use of this artifact.

When using 'jaxws' Scout Maven profile, annotation processing is enabled for that module by default. But, an entry point for a webservice port type will only be generated if enabled for that port type, meaning that a class annotated with WebServiceEntryPoint pointing to that very endpoint interface is found in this module. Anyway, for a sole webservice consumer, it makes no sense to generate an entry point at all.

Enable Annotation Processing Tool (APT) in Eclipse IDE

In Eclipse IDE, the workspace build ignores annotation processing as configured in pom.xml. Instead, it must be enabled separately with the following files. Nevertheless, to simply run Maven build with annotation support from within Eclipse IDE, those files are not required.

file	description
.settings/org.eclipse.jdt.core.prefs	Enables APT for this module via the property *org.eclipse.jdt.core.compiler.processAnnotations=enabled*
.settings/org.eclipse.jdt.apt.core.prefs	Enables APT for this module via the property *org.eclipse.jdt.apt.aptEnabled=true*
.factorypath	Specifies the annotation processor to be used (JaxWsAnnotationProcessor) and dependent artifacts

16.4.4. Build webservice stubs and APT artifacts from console

Simply run *mvn clean compile* on the project. If you are experiencing some problems, run with -X debug flag to get a more detailed error message.

16.4.5. Build webservice stubs and APT artifacts from within Eclipse IDE

In the Eclipse IDE, there are three ways to generate webservice stubs and APT artifacts.

1. the implicit way on behalf of the workspace build and m2e integration (automatically, but sometimes not reliable)
2. the explicit but potentially slow way by doing a 'Update Maven Project' with 'clean projects' checked (Alt+F5)
3. the explicit and faster way by running a Maven build for that project. Thereto, right-click on the

project or pom.xml, then select the menu 'Run As | Maven build...', then choose 'clean compile' as its goal and check 'Resolve workspace artifacts', and finally click 'Run'. Afterwards, do not forget to refresh the project by pressing F5.

If the webservice stub(s) or APT artifacts are not generated (anew or at all), delete the target folder manually, and continue according to procedure number three. A possible reason might be the presence of 'target\jaxws\wsartifact-hash'. Then, for each webservice, a 'hash file' is computed by 'wsimport', so that regeneration only occurs upon a change of WSDL or XSD files.

16.4.6. Exclude derived resources from version control

Stub and APT artifacts are derived resources, and should be excluded from version control. When working with Eclipse IDE, this is done automatically by eGit, because it adds derived resources to *.gitignore* (if configured to do so).

16.4.7. JaxWsAnnotationProcessor

`JaxWsAnnotationProcessor` is an annotation processor provided by Scout JAX-WS integration to generate an entry point for an endpoint interface during compilation. The instructions how to generate the entry point is given via a Java class or Java interface annotated with WebServiceEntryPoint annotation.

16.5. Provide a webservice

In this chapter, you will learn how to publish a webservice provider via an entry point.

16.5.1. The concept of an Entry Point

An entry point implements the endpoint interface (or port type interface), and is published as the webservice endpoint for that endpoint interface. The entry point itself is auto generated by JaxWsAnnotationProcessor during compile time, based on instructions as given by the respective class/interface annotated with `WebServiceEntryPoint` annotation. The entry point is responsible to enforce authentication and to run the web request in a `RunContext`. In turn, the request is propagated to the bean implementing the endpoint interface.

Figure 13 illustrates the endpoint's class hierarchy and the message flow for a web request.

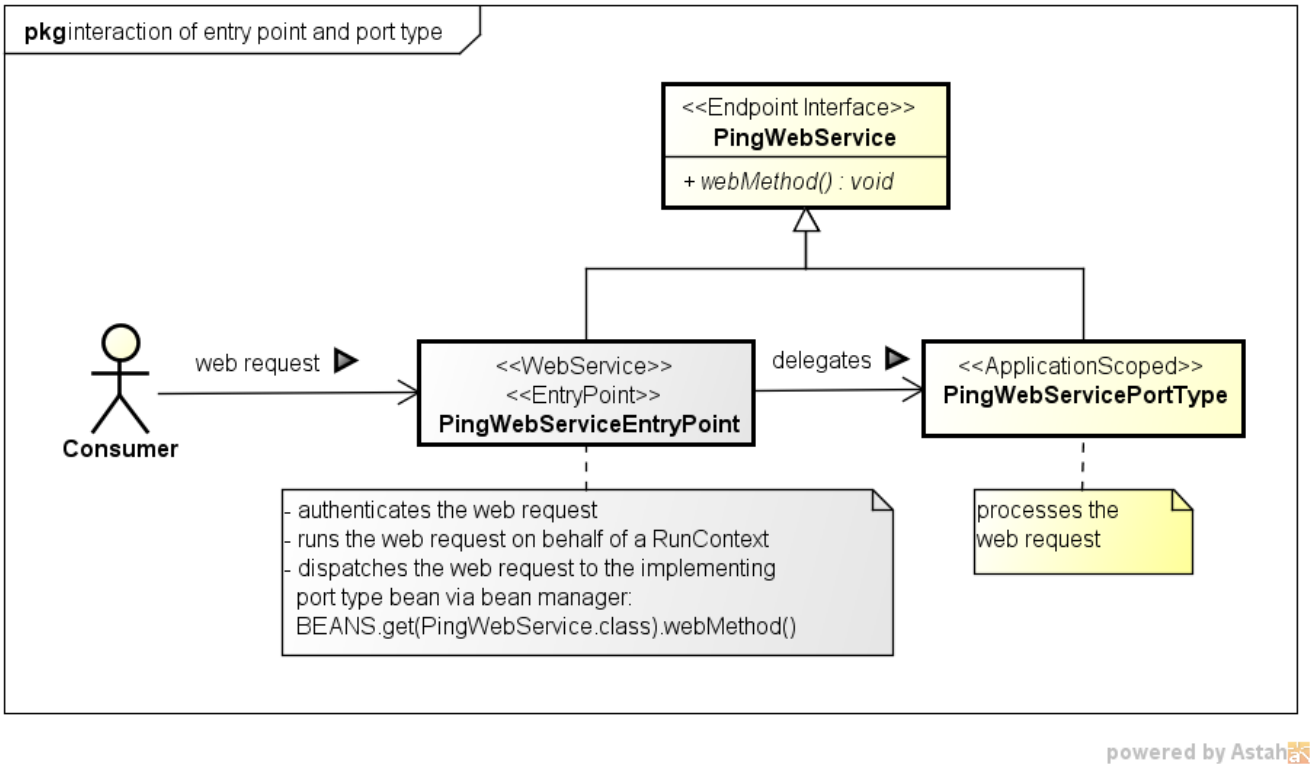


Figure 13. Interaction of entry point and port type

As you can see, both, entry point and port type implement the endpoint interface. But it is the entry point which is actually installed as the webservice endpoint, and which receives web requests. However, the webservice itself is implemented in the implementing bean, which typically is located in *server* module. See Section 16.3 for more information. Upon a web request, the entry point simply intercepts the web request, and then invokes the web method on the implementing bean for further processing.

See an example of an implementing port type bean, which is invoked by entry point.

Do not forget to annotate the implementing bean with `ApplicationScoped` annotation in order to be found by bean manager.

16.5.2. Generate an Entry Point as an endpoint interface

This section describes the steps required to generate an entry point. For demonstration purposes, a simple ping webservice is used, which provides a single method 'ping' to accept and return a `String` object.

See the WSDL file of ping webservice: Section 16.8.1
See the endpoint interface of ping webservice: Section 16.8.2

To generate an entry point for the webservice's endpoint interface, create an interface as following in your jaxws project.

```
@WebServiceEntryPoint(endpointInterface = PingWebServicePortType.class) ②
interface PingWebServiceEntryPointDefinition { ①
}
```

① Create an interface or class to act as an anchor for the `WebServiceEntryPoint` annotation. This class or interface has no special meaning, except that it declares the annotation to be interpreted by annotation processor.

② Reference the endpoint interface for which an entry point should be generated for. Typically, the endpoint interface is generated by 'wsimport' and is annotated with `WebService` annotation.

It is important to understand, that the interface `PingWebServiceEntryPointDefinition` solely acts as the anchor for the `WebServiceEntryPoint` annotation. This class or interface has no special meaning, except that it declares the annotation to be interpreted by annotation processor. Typically, this class is called *Entry Point Definition*.

If running `mvn clean compile`, an entry point is generated for that endpoint interface. See the entry point as generated for ping webservice: Section 16.8.3

If you should experience some problems in the entry point generation, refer to Build webservice stubs and APT artifacts from within Eclipse IDE, or Build webservice stubs and APT artifacts from console.

16.5.3. Instrument the Entry Point generation

This section gives an overview on how to configure the entry point to be generated.

attribute	description
endpointInterface (*mandatory*)	Specifies the endpoint interface for which to generate an entry point for. An endpoint interface defines the service's abstract webservice contract, and is also known as port type interface. Also, the endpoint interface is annotated with WebService annotation.
entryPointName	Specifies the class name of the entry point generated. If not set, the name is like the name of the endpoint interface suffixed with EntryPoint.
entryPointPackage	Specifies the package name of the entry point generated. If not set, the package name is the same as of the element declaring this WebServiceEntryPoint annotation.
serviceName	Specifies the service name as declared in the WSDL file, and must be set if publishing the webservice via auto discovery in an EE container. Both, 'serviceName' and 'portName' uniquely identify a webservice endpoint to be published. See for valid service names in the WSDL: `<wsdl:service name="SERVICE_NAME">⋯</wsdl:service>`
portName	Specifies the name of the port as declared in the WSDL file, and must be set if publishing the webservice via auto discovery in an EE container. Both, 'serviceName' and 'portName' uniquely identify a webservice endpoint to be published. See for valid port names in the WSDL: `<wsdl:service name="⋯"><wsdl:port name="PORT_NAME" binding="⋯"/></wsdl:service>`
wsdlLocation	Specifies the location of the WSDL document. If not set, the location is derived from WebServiceClient annotation which is typically initialized with the 'wsdlLocation' as provided to 'wsimport'.

attribute	description
authentication	Specifies the authentication mechanism to be installed, and in which `RunContext` to run authenticated requests. By default, authentication is *disabled*. If *enabled*, an `AuthenticationHandler` is generated and registered in the handler chain as very first handler. However, the position of that handler can be changed via *order* field on `Authentication` annotation. See Section 16.5.3.1 for more information.
handlerChain	Specifies the handlers to be installed. The order of the handlers is as declared. A handler is looked up as a bean, and must implement `javax.xml.ws.handler.Handler` interface. See Section 16.5.5 for more information.

Besides the instructions which can be set via `WebServiceEntryPoint` annotation, it is further possible to contribute other annotations to the entry point. Simply declare the annotation of your choice as a sibling annotation to `WebServiceEntryPoint` annotation. In turn, this annotation will be added to the entry point as well. This may be useful to enable some vendor specific features, or e.g. to enable *MTOM* to efficiently send binary data to a client.
That also applies for `WebService` annotation to overwrite values as declared in the WSDL file.

Further, you can also provide your own *handler chain binding file*. However, *handlers* and *authentication* as declared via `WebServiceEntryPoint` annotation are ignored then.

Handlers registered via *handlerChain* must be beans, meaning either annotated with *@Bean* or *@ApplicationScoped*.

The binding to the concrete endpoint is done via 'endpointInterface' attribute. If a WSDL declares multiple services, create a separate entry point definition for each service to be published.

Annotate the *Entry Point Definition* class with `IgnoreWebServiceEntryPoint' to not generate an entry point for that definition. This is primarily used while developing an entry point, or for documenting purpose.

Some fields require you to provide a Java class. Such fields are mostly of the annotation type `Clazz`, which accepts either the concrete `Class`, or its 'fully qualified name'. Use the latter if the class is not visible from within `jaxws` module.
However, if ever possible specify a `Class`. Because most classes are looked up via bean manager, this can be achieved with an interface located in 'jaxws' module, but with an implementation in 'server' module.

Configure Authentication

The field 'authentication' on `WebServiceEntryPoint` configures what authentication mechanism to install on the webservice endpoint, and in which `RunContext` to run authenticated webservice requests. It consists of the IAuthenticationMethod to challenge the client to provide credentials, and the ICredentialVerifier to verify request's credentials against a data source.

By default, authentication is *disabled*. If *enabled*, an `AuthenticationHandler` is generated and registered in the handler chain as very first handler. The position can be changed via *order* field on `Authentication` annotation.

The following properties can be set.

method (*mandatory*)	Specifies the authentication method to be used to challenge the client to provide credentials. By default, `NullAuthenticationMethod` is used to disable authentication. See IAuthenticationMethod for more information.
verifier	Specifies against which data source credentials are to be verified. By default, `ForbiddenCredentialVerifier` is used to reject any webservice request. See ICredentialVerifier for more information.
order	Specifies the position where to register the authentication handler in the handler chain. By default, it is registered as the very first handler.
principalProducer	Indicates the principal producer to use to create principals to represent authenticated users. By default, SimplePrincipalProducer is used.
runContextProducer	Indicates which RunContext to use to run authenticated webservice requests. By default, `ServerRunContextProducer` is used, which is based on a session cache, and enforces to run in a new transaction.

If using container based authentication (authentication enforced by the application server), use ContainerBasedAuthenticationMethod as authentication method, and do not configure a credential verifier.

16.5.4. Example of an Entry Point definition

Listing 82. Example configuration for an entry point definition

```
@WebServiceEntryPoint(
    endpointInterface = PingWebServicePortType.class, ①
    entryPointName = "PingWebServiceEntryPoint",
    entryPointPackage = "org.eclipse.scout.docs.ws.ping",
    serviceName = "PingWebService",
    portName = "PingWebServicePort",
    handlerChain = {②
        @Handler(@Clazz(CorrelationIdHandler.class)), ③
        @Handler(value = @Clazz(IPAddressFilter.class), initParams = { ④
            @InitParam(key = "rangeFrom", value = "192.200.0.0"),
            @InitParam(key = "rangeTo", value = "192.255.0.0")}),
        @Handler(@Clazz(LogHandler.class)), ⑤
    },
    authentication = @Authentication( ⑥
        order = 2, ⑦
        method = @Clazz(BasicAuthenticationMethod.class), ⑧
        verifier = @Clazz(ConfigFileCredentialVerifier.class))) ⑨
@MTOM ⑩
```

① References the endpoint interface for which to generate an entry point for.

② Declares the handlers to be installed on that entry point. The order is as declared.

③ Registers the 'CorrelationIdHandler' as the first handler to set a correlation ID onto the current message context. See Section 16.5.6 for more information about state propagation.

④ Registers the 'IpAddressFilter' as the second handler to filter for IP addresses. Also, this handler is parameterized with 'init params' to configure the valid IP range.

⑤ Registers the `LogHandler` as the third handler to log SOAP messages.

⑥ Configures the webservice's authentication.

⑦ Configures the 'AuthHandler' to be put at position 2 (0-based), meaning in between of `IpAddressFilter` and `LogHandler`. By default, `AuthHandler` would be the very first handler in the handler chain.

⑧ Configures to use *BASIC AUTH* as authentication method.

⑨ Configures to verify user's credentials against 'config.properties' file.

⑩ Specification of an `MTOM` annotation to be added to the entry point.

This configuration generates the following artifacts:

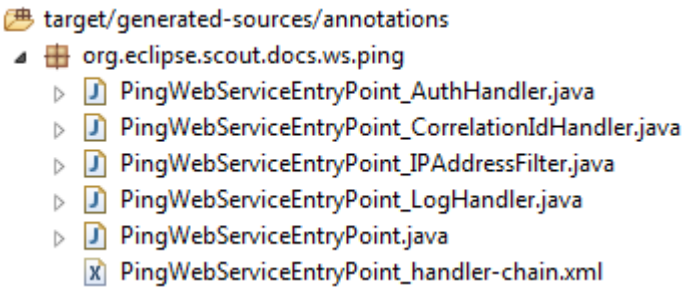


Figure 14. generated artifacts

All artifacts are generated into the package 'org.eclipse.scout.docs.ws.ping', as specified by the definition. The entry point itself is generated into 'PingWebServiceEntryPoint.java'. Further, for each handler, a respective handler delegate is generated. That allows handlers to be looked up via bean manager, and to run the handlers on behalf of a `RunContext`. Also, an `AuthHandler` is generated to authenticate web requests as configured.

The handler-chain XML file generated looks as following. As specified, the authentication handler is installed as the third handler.

Listing 83. PingWebServiceEntryPoint_handler-chain.xml

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <!-- Executed as 4. handler-->
    <handler>
      <handler-class>org.eclipse.scout.docs.ws.ping
.PingWebServiceEntryPoint_LogHandler</handler-class>
    </handler>
  </handler-chain>
  <handler-chain>
    <!-- Executed as 3. handler-->
    <handler>
      <handler-class>org.eclipse.scout.docs.ws.ping
.PingWebServiceEntryPoint_AuthHandler</handler-class>
    </handler>
  </handler-chain>
  <handler-chain>
    <!-- Executed as 2. handler-->
    <handler>
      <handler-class>org.eclipse.scout.docs.ws.ping
.PingWebServiceEntryPoint_IPAddressFilter</handler-class>
    </handler>
  </handler-chain>
  <handler-chain>
    <!-- Executed as 1. handler-->
    <handler>
      <handler-class>org.eclipse.scout.docs.ws.ping
.PingWebServiceEntryPoint_CorrelationIdHandler</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

The following listing shows the beginning of the entry point generated. As you can see, the handler-chain XML file is referenced via `HandlerChain` annotation, and the `MTOM` annotation was added as well.

Listing 84. PingWebServiceEntryPoint.java

```
@WebService(name = "PingWebServicePortType",
    targetNamespace = "http://scout.eclipse.org/docs/ws/PingWebService/",
    endpointInterface =
"org.eclipse.scout.docs.snippets.JaxWsSnippet.PingWebServicePortType",
    serviceName = "PingWebService",
    portName = "PingWebServicePort")
@MTOM
@HandlerChain(file = "PingWebServiceEntryPoint_handler-chain.xml")
public class PingWebServiceEntryPoint implements PingWebServicePortType {
```

16.5.5. Configure JAX-WS Handlers

See listing for an example of how to configure JAX-WS handlers.

JAX-WS handlers are configured directly on the entry point definition via the array field `handlerChain`. In turn, *JaxWsAnnotationProcessor* generates a 'handler XML file' with the handler's order preserved, and which is registered in entry point via annotation `handlerChain`.

A handler can be initialized with static 'init parameters', which will be injected into the handler instance. For the injection to work, declare a member of the type `Map` in the handler class, and annotate it with `javax.annotation.Resource` annotation.

Because handlers are looked up via bean manager, a handler must be annotated with `ApplicationScoped` annotation.

If a handler requires to be run in a `RunContext`, annotate the handler with `RunWithRunContext` annotation, and optionally specify a `RunContextProducer`. If the web request is authenticated upon entering the handler, the `RunContext` is run on behalf of the authenticated user. Otherwise, if not authenticated yet, it is invoked with the Subject as configured in `jaxws.provider.user.handler` config property.

Listing 85. Example of a JAX-WS Handler

```
@ApplicationScoped ①
@RunWithRunContext ②
public class IPAddressFilter implements SOAPHandler<SOAPMessageContext> {

  @Resource
  private Map<String, String> m_initParams; ③

  @Override
  public boolean handleMessage(SOAPMessageContext context) {
    String rangeForm = m_initParams.get("rangeFrom"); ④
    String rangeTo = m_initParams.get("rangeTo");
    // ...
    return true;
  }

  @Override
  public boolean handleFault(SOAPMessageContext context) {
    return true;
  }

  @Override
  public Set<QName> getHeaders() {
    return Collections.emptySet();
  }

  @Override
  public void close(MessageContext context) {
  }
}
```

① Annotate the Handler with `ApplicationScoped` annotation, so it can be looked up via bean manager

② Optionally annotate the Handler with `RunWithRunContext` annotation, so the handler is invoked in a `RunContext`

③ Declare a `Map` member annotated with `Resource` annotation to make injection of 'init parameters' work

④ Access injected 'init parameters'

16.5.6. Propagate state among Handlers and port type

Sometimes it is useful to share state among handlers, and even with the port type. This can be done via `javax.xml.ws.handler.MessageContext`. By default, a property put onto message context is only available in the handler chain. To make it available to the port type as well, set its scope to 'APPLICATION' accordingly.

The following listings gives an example of how to propagate state among handlers and port type.

Listing 86. This handler puts the correlation ID onto message context to be accessible by subsequent handlers and the port type.

```
@ApplicationScoped
public class CorrelationIdHandler implements SOAPHandler<SOAPMessageContext> {

  @Override
  public boolean handleMessage(SOAPMessageContext context) {
    context.put("cid", UUID.randomUUID().toString()); ①
    context.setScope("cid", Scope.APPLICATION); ②
    return true;
  }

  @Override
  public boolean handleFault(SOAPMessageContext context) {
    return true;
  }

  @Override
  public Set<QName> getHeaders() {
    return Collections.emptySet();
  }

  @Override
  public void close(MessageContext context) {
  }
}
```

① Put the 'correlation ID' onto message context.

② Set scope to *APPLICATION* to be accessible in port type. By default, the scope if *HANDLER* only.

Listing 87. This handler accesses the 'correlation ID' as set by the previous handler.

```
@ApplicationScoped
public class CorrelationIdLogger implements SOAPHandler<SOAPMessageContext> {

  @Override
  public boolean handleMessage(SOAPMessageContext context) {
    String correlationId = (String) context.get("cid"); ①
    // ...
    return true;
  }

  @Override
  public boolean handleFault(SOAPMessageContext context) {
    return true;
  }

  @Override
  public void close(MessageContext context) {
  }

  @Override
  public Set<QName> getHeaders() {
    return Collections.emptySet();
  }
}
```

① Get the 'correlation ID' from message context.

Listing 88. This port type accesses the 'correlation ID' as set by the previous handler.

```
@ApplicationScoped
public class CorrelationIdPortType implements PingWebServicePortType {

  @Override
  public String ping(String ping) {
    MessageContext currentMsgCtx = IWebServiceContext.CURRENT.get().getMessageContext
(); ①
    String correlationId = (String) currentMsgCtx.get("cid"); ②
    // ...
    return ping;
  }
}
```

① Get the current message context via thread local `IWebServiceContext`

② Get the 'correlation ID' from message context.

16.5.7. JAX-WS Correlation ID Propagation

Scout's JAX-WS integration already provides complete support for reading a correlation ID from the

HTTP header named `X-Scout-Correlation-Id` of the incoming web service request and propagates it to the RunContext that executes the actual service operation. A new correlation ID is created if the HTTP header is empty or missing.

The CorrelationIdHandler example above just illustrates the capabilities of a SOAPHandler.

You have to implement your own handler if the consumer provides a correlation ID in another header parameter or as part of the request's payload.

Listing 89. Add Scout's WsProviderCorrelationIdHandler to the handler chain

```
@WebServiceEntryPoint(
    endpointInterface = PingWebServicePortType.class,
    entryPointName = "PingWebServiceEntryPoint",
    entryPointPackage = "org.eclipse.scout.docs.ws.ping2",
    serviceName = "PingWebService",
    portName = "PingWebServicePort",
    handlerChain = {
        @Handler(@Clazz(WsProviderCorrelationIdHandler.class)), ①
        @Handler(@Clazz(LogHandler.class)),
    },
    authentication = @Authentication(
        method = @Clazz(BasicAuthenticationMethod.class),
        verifier = @Clazz(ConfigFileCredentialVerifier.class)))
```

① Add the correlation ID handler **at the beginning** of the handler chain to ensure that all handlers can use its value (especially the `LogHandler` has to be added after the correlation ID handler).

16.5.8. Registration of webservice endpoints

The registration of webservice endpoints depends on the implementor you use, and whether you are running in an EE container with webservice auto discovery enabled.

When running in an EE container, webservice providers are typically found by their presence. In order to be found, such webservice providers must be annotated with `WebService` annotation, and must have the coordinates 'serviceName' and 'portName' set. Still, most application servers allow for manual registration as well. E.g. if using Oracle WebLogic Server (WLS), endpoints to be published can be registered directly in 'web.xml' as a Servlet. However, this is vendor specific. Refer to the container's documentation for more information.

If not running in an EE container, the registration is implementor specific. In the following, an example for *JAX-WS METRO* and *Apache CXF* is given.

JAX-WS METRO

During startup, *JAX-WS METRO* looks for the file '/WEB-INF/sun-jaxws.xml', which contains the endpoint definitions.

Listing 90. WEB-INF/sun-jaxws.xml

```
<jws:endpoints xmlns:jws="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">

  <!-- PingWebService -->
  <jws:endpoint
      name="PingService"
      implementation="org.eclipse.scout.docs.ws.ping.PingWebServiceEntryPoint"
      service="{http://scout.eclipse.org/docs/ws/PingWebService/}PingWebService"
      port="{http://scout.eclipse.org/docs/ws/PingWebService/}PingWebServiceSOAP"
      url-pattern="/jaxws/PingWebService"/>
</jws:endpoints>
```

Apache CXF

During startup, *Apache CXF* looks for the config file as specified in 'web.xml' via 'config-location'. See Listing 74 for more information.

Listing 91. WEB-INF/cxf-jaxws.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xsi:schemaLocation=" http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
  <import resource="classpath:META-INF/cxf/cxf.xml" />

  <!-- PingWebService -->
  <jaxws:endpoint id="PingWebService"
    implementor="org.eclipse.scout.docs.ws.ping.PingWebServiceEntryPoint"
    address="/PingWebService" />
</beans>
```

As the webservice endpoint, specify the fully qualified name to the entry point, and not to the implementing port type.

Depending on the implementor, a HTML page may be provided to see all webservices published.
For *JAX-WS METRO*, enter the URL to a concrete webservice, e.g. http://localhost:8080/jaxws/PingWebService.
For *Apache CXF*, enter the base URL where the webservices are published, e.g. http://localhost:8080/jaxws.

16.6. Consume a webservice

Communication with a webservice endpoint is done based on the webservice's port generated by 'wsimport'. Learn more how to generate a webservice stub from a WSDL file.

To interact with a webservice endpoint, create a concrete 'WebServiceClient' class which extends from `AbstractWebServiceClient`, and specify the endpoint's coordinates ('service' and 'port') via its bounded type parameters.

Listing 92. Example of a WS-Client

```
public class PingWebServiceClient extends AbstractWebServiceClient<PingWebService,
PingWebServicePortType> { ①
}
```

① Specify 'service' and 'port' via bounded type parameters

A WS-Client can be configured with some default values like the endpoint URL, credentials, timeouts and more. However, the configuration can also be set or overwritten later when creating the `InvocationContext`.

See also Section 16.6.7.

Listing 93. Example of a WS-Client configuration

```
public class PingWebServiceClient1 extends AbstractWebServiceClient<PingWebService,
PingWebServicePortType> {

  @Override
  protected Class<? extends IConfigProperty<String>> getConfiguredEndpointUrlProperty
() {
    return JaxWsPingEndpointUrlProperty.class; ①
  }

  @Override
  protected Class<? extends IConfigProperty<String>> getConfiguredUsernameProperty() {
    return JaxWsPingUsernameProperty.class; ②
  }

  @Override
  protected Class<? extends IConfigProperty<String>> getConfiguredPasswordProperty() {
    return JaxWsPingPasswordProperty.class; ②
  }

  @Override
  protected Class<? extends IConfigProperty<Integer>>
getConfiguredConnectTimeoutProperty() {
    return JaxWsPingConnectTimeoutProperty.class; ③
  }

  @Override
  protected Class<? extends IConfigProperty<Integer>>
getConfiguredReadTimeoutProperty() {
    return JaxWsPingReadTimeoutProperty.class; ③
  }
}
```

① Specifies the endpoint URL

② Specifies credentials

③ Specifies timeouts

16.6.1. Invoke a webservice

A webservice operation is invoked on behalf of an invocation context, which is associated with a dedicated port, and which specifies the data to be included in the web request. Upon a webservice call, the invocation context should be discarded.

Listing 94. Example of a webservice call

```
PingWebServicePortType port = BEANS.get(PingWebServiceClient.class)
.newInvocationContext().getPort(); ①

port.ping("Hello world"); ②
```

① Obtain a new invocation context and port via WS-Client

② Invoke the webservice operation

Invoking `newInvocationContext()` returns a new context and port instance. The context returned inherits all properties as configured for the WS-Client (endpoint URL, credentials, timeouts, ...), but which can be overwritten for the scope of this context.

The following listing illustrates how to set/overwrite properties.

Listing 95. Configure invocation context with data to be included in the web request

```
final InvocationContext<PingWebServicePortType> context = BEANS.get
(PingWebServiceClient.class).newInvocationContext();

PingWebServicePortType port = context
    .withUsername("test-user") ①
    .withPassword("secret")
    .withConnectTimeout(10, TimeUnit.SECONDS) ②
    .withoutReadTimeout() ③
    .withHttpRequestHeader("X-ENV", "integration") ④
    .getPort();

port.ping("Hello world"); ⑤
```

① Set the credentials

② Change the connect timeout to 10s

③ Unset the read timeout

④ Add a HTTP request header

⑤ Invoke the webservice operation

The WS-Client provides port instances via a preemptive port cache. This cache improves performance because port creation may be an expensive operation due to WSDL/schema validation. The cache is based on a 'corePoolSize', meaning that that number of ports is created on a preemptively basis. If more ports than that number are required, they are created on demand, and additionally added to the cache until expired, which is useful at a high load.

The JAX-WS specification does not specify thread safety of a port instance. Therefore, a port should not be used concurrently among threads. Further, JAX-WS API does not support to reset the Port's request and response context, which is why a port should only be used for a single webservice call.

16.6.2. Cancel a webservice request

The WS-Client supports for cancellation of webservice requests. Internally, every web request is run in another thread, which the calling thread waits for to complete. Upon cancellation, that other thread is interrupted, and the calling thread released with a `WebServiceRequestCancelledException`. However, depending on the JAX-WS implementor, the web request may still be running, because JAX-WS API does not support the cancellation of a web request.

16.6.3. Get information about the last web request

The invocation context allows you to access HTTP status code and HTTP headers of the last web request.

```
final InvocationContext<PingWebServicePortType> context = BEANS.get
(PingWebServiceClient.class).newInvocationContext();

String pingResult = context.getPort().ping("Hello world");

// Get HTTP status code
int httpStatusCode = context.getHttpStatusCode();

// Get HTTP response header
List<String> httpResponseHeader = context.getHttpResponseHeader("X-CUSTOM-HEADER");
```

16.6.4. Propagate state to Handlers

An invocation context can be associated with request context properties, which are propagated to handlers and JAX-WS implementor.

```
BEANS.get(PingWebServiceClient.class).newInvocationContext()
    .withRequestContextProperty("cid", UUID.randomUUID().toString()) ①
    .getPort().ping("Hello world"); ②
```

① Propagate the correlation ID

② Invoke the web operation

Learn more how to access context properties from within a handler in Listing 87.

16.6.5. Install handlers and provide credentials for authentication

To install a handler, overwrite `execInstallHandlers` and add the handler to the given List. The handlers are invoked in the order as added to the handler-chain. By default, there is no handler installed.

The method `execInstallHandlers` is invoked upon preemptive creation of the port. Consequently, you cannot do any assumption about the calling thread.

If a handler requires to run in another `RunContext` than the calling context, annotate it with

`RunWithRunContext` annotation, e.g. to start a new transaction to log into database.

If the endpoint requires to authenticate requests, an authentication handler is typically added to the list, e.g. `BasicAuthenticationHandler` for 'Basic authentication', or `WsseUsernameTokenAuthenticationHandler` for 'Message Level WS-Security authentication', or some other handler to provide credentials.

```
public class PingWebServiceClient2 extends AbstractWebServiceClient<PingWebService,
PingWebServicePortType> {

  @Override
  protected void execInstallHandlers(List<javax.xml.ws.handler.Handler<?>>
handlerChain) {
    handlerChain.add(new BasicAuthenticationHandler());
    handlerChain.add(BEANS.get(LogHandler.class));
  }
}
```

The credentials as provided via `InvocationContext` can be accessed via request context with the property `InvocationContext.PROP_USERNAME` and `InvocationContext.PROP_PASSWORD`.

16.6.6. JAX-WS Client Correlation ID Propagation

The current context's correlation ID can be forwarded to the consumed web service. Scout provides a handler that sets the `X-Scout-Correlation-Id` HTTP header on the outgoing request.

```
public class PingWebServiceClient3 extends AbstractWebServiceClient<PingWebService,
PingWebServicePortType> {

  @Override
  protected void execInstallHandlers(List<javax.xml.ws.handler.Handler<?>>
handlerChain) {
    handlerChain.add(new BasicAuthenticationHandler());
    handlerChain.add(BEANS.get(LogHandler.class));
    handlerChain.add(BEANS.get(WsConsumerCorrelationIdHandler.class)); ①
  }
}
```

① The handler can be at any position in the handler chain.

16.6.7. Default configuration of WS-Clients

The following properties can be set globally for all WS-Clients. However, a WS-Client can overwrite any of this values.

property	description	default value
jaxws.consumer.portCache.enabled	To indicate whether to use a preemptive port cache for WS-Clients. Depending on the implementor used, cached ports may increase performance, because port creation is an expensive operation due to WSDL and schema validation. The cache is based on a 'corePoolSize', meaning that that number of ports is created on a preemptive basis. If more ports than that number is required, they are are created on demand and also added to the cache until expired, which is useful at a high load.	true
jaxws.consumer.portCache.corePoolSize	Number of ports to be preemptively cached to speed up webservice calls.	10
jaxws.consumer.portCache.ttl	Maximum time in seconds to retain ports in the cache if the 'corePoolSize' is exceeded. That typically occurs at high load, or if 'corePoolSize' is undersized.	15 minutes
jaxws.consumer.connectTimeout	Connect timeout in milliseconds to abort a webservice request, if establishment of the HTTP connection takes longer than this timeout. A timeout of null means an infinite timeout.	infinite
jaxws.consumer.readTimeout	Read timeout in milliseconds to abort a webservice request, if it takes longer than this timeout for data to be available for read. A timeout of null means an infinite timeout.	infinite

16.7. XML adapters to work with java.util.Date and java.util.Calendar

Scout ships with some XML adapters to not have to work with `XMLGregorianCalendar`, but with `java.util.Date` instead.
It is recommended to configure your global binding file accordingly. See Listing 79 for an example.

See the adapter's JavaDoc for more detailed information.

Table 1. UTC Date adapters

adapter	description
UtcDateAdapter	Use this adapter to work with UTC *xsd:dates*. A UTC date is also known as 'zulu' date, and has 'GMT+-00:00'. Unlike `UtcDateTimeAdapter`, this adapter truncates hours, minutes, seconds and milliseconds.
UtcTimeAdapter	Use this adapter to work with UTC *xsd:times*. A UTC time is also known as 'zulu' time, and has 'GMT+-00:00'. Unlike `UtcDateTimeAdapter`, this adapter sets year, month and day to the epoch, which is defined as 1970-01-01 in UTC.

adapter	description
UtcDateTimeAdapter	Use this adapter to work with UTC *xsd:dateTimes*. A UTC time is also known as 'zulu' time, and has 'GMT+-00:00'. This adapter converts xsd:dateTime into UTC milliseconds, by respecting the timezone as provided. If the timezone is missing, the date is interpreted as UTC-time, and not local to the default JVM timezone. To convert a `Date` into *xsd:dateTime*, the date's milliseconds are used as UTC milliseconds from the epoch, and are formatted as 'zulu' time.

Table 2. Calendar adapters

adapter	description
CalendarDateAdapter	Use this adapter to work with `Calendar` *xsd:dates* without loosing timezone information. Unlike CalendarDateTimeAdapter, this adapter truncates hours, minutes, seconds and milliseconds.
CalendarTimeAdapter	Use this adapter to work with `Calendar` *xsd:times* without loosing timezone information. Unlike CalendarDateTimeAdapter, this adapter sets year, month and day to the epoch, which is defined as 1970-01-01 in UTC.
CalendarDateTimeAdapter	Adapter to convert a *xsd:dateTime* to a `Calendar` and vice versa. For both directions, the timezone information is not lost. Use this adapter if you expect to work with dates from various timezones without loosing the local time. If the UTC (Zulu-time) is sufficient, use `UtcDateTimeAdapter` instead.

Table 3. Default timezone Date adapters

adapter	description
DefaultTimezoneDateAdapter	Use this adapter to work with *xsd:dates* in the default timezone of the Java Virtual Machine. Depending on the JVM installation, the timezone may differ: 'GMT+-XX:XX'. Unlike `DefaultTimezoneDateTimeAdapter`, this adapter truncates hours, minutes, seconds and milliseconds. Whenever possible, use `UtcDateAdapter` or `CalendarDateAdapter` instead.
DefaultTimezoneTimeAdapter	Use this adapter to work with *xsd:times* in the default timezone of the Java Virtual Machine. Depending on the JVM installation, the timezone may differ: 'GMT+-XX:XX'. Unlike DefaultTimezoneDateTimeAdapter, this adapter sets year, month and day to the epoch, which is defined as 1970-01-01 in UTC. Whenever possible, use `UtcTimeAdapter` or `CalendarTimeAdapter` instead.

adapter	description
DefaultTimezoneDateTimeAdapter	Use this adapter to work with *xsd:dateTimes* in the default timezone of the Java Virtual Machine. Depending on the JVM installation, the timezone may differ: 'GMT+-XX:XX'. Whenever possible, use `UtcDateTimeAdapter` or `CalendarDateTimeAdapter` instead.

16.8. JAX-WS Appendix

16.8.1. PingWebService.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions name="PingWebService"
  targetNamespace="http://scout.eclipse.org/docs/ws/PingWebService/"
  xmlns:tns="http://scout.eclipse.org/docs/ws/PingWebService/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <xsd:schema targetNamespace="http://scout.eclipse.org/docs/ws/PingWebService/">
      <xsd:element name="pingRequest" type="xsd:string"/>
      <xsd:element name="pingResponse" type="xsd:string"/>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="pingRequest">
    <wsdl:part element="tns:pingRequest" name="ping" />
  </wsdl:message>
  <wsdl:message name="pingResponse">
    <wsdl:part element="tns:pingResponse" name="parameters" />
  </wsdl:message>
  <wsdl:portType name="PingWebServicePortType">
    <wsdl:operation name="ping">
      <wsdl:input message="tns:pingRequest" />
      <wsdl:output message="tns:pingResponse" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="PingWebServiceSOAP" type="tns:PingWebServicePortType">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="ping">
      <soap:operation soapAction="
http://scout.eclipse.org/docs/ws/PingWebService/ping" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="PingWebService">
    <wsdl:port binding="tns:PingWebServiceSOAP" name="PingWebServiceSOAP">
      <soap:address location="http://scout.eclipse.org/docs/ws/PingWebService/" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

16.8.2. PingWebServicePortType.java

```
@FunctionalInterface
@WebService(name = "PingWebServicePortType", targetNamespace =
"http://scout.eclipse.org/docs/ws/PingWebService/")
@SOAPBinding(parameterStyle = ParameterStyle.BARE)
public interface PingWebServicePortType {

  @WebMethod(action = "http://scout.eclipse.org/docs/ws/PingWebService/ping")
  @WebResult(name = "pingResponse", targetNamespace =
"http://scout.eclipse.org/docs/ws/PingWebService/", partName = "parameters")
  String ping(@WebParam(name = "pingRequest", targetNamespace =
"http://scout.eclipse.org/docs/ws/PingWebService/", partName = "ping") String ping);
}
```

16.8.3. PingWebServicePortTypeEntryPoint.java

```
@Generated(value = "org.eclipse.scout.jaxws.apt.JaxWsAnnotationProcessor", date =
"2016-01-25T14:22:58:583+0100", comments = "EntryPoint to run webservice requests on
behalf of a RunContext")
@WebService(name = "PingWebServicePortType", targetNamespace =
"http://scout.eclipse.org/docs/ws/PingWebService/", endpointInterface =
"org.eclipse.scout.docs.ws.pingwebservice.PingWebServicePortType")
public class PingWebServicePortTypeEntryPoint implements org.eclipse.scout.docs.ws
.pingwebservice.PingWebServicePortType {

  @Resource
  protected WebServiceContext m_webServiceContext;

  @Override
  public String ping(final String ping) {
    try {
      return lookupRunContext().call(new Callable<String>() {
        @Override
        public final String call() throws Exception {
          return BEANS.get(PingWebServicePortType.class).ping(ping);
        }
      }, DefaultExceptionTranslator.class);
    }
    catch (Exception e) {
      throw handleUndeclaredFault(e);
    }
  }

  protected RuntimeException handleUndeclaredFault(final Exception e) {
    throw BEANS.get(JaxWsUndeclaredExceptionTranslator.class).translate(e);
  }

  protected RunContext lookupRunContext() {
    return BEANS.get(JaxWsRunContextLookup.class).lookup(m_webServiceContext);
  }
}
```

16.8.4. PingWebServicePortTypeBean.java

```
@ApplicationScoped
public class PingWebServicePortTypeBean implements PingWebServicePortType {

  @Override
  public String ping(String ping) {
    return "ping: " + ping;
  }
}
```

16.8.5. .settings/org.eclipse.jdt.core.prefs file to enable APT in Eclipse IDE

```
...
org.eclipse.jdt.core.compiler.processAnnotations=enabled
...
```

16.8.6. .settings/org.eclipse.jdt.apt.core.prefs file to enable APT in Eclipse IDE

```
org.eclipse.jdt.apt.aptEnabled=true
org.eclipse.jdt.apt.genSrcDir=target/generated-sources/annotations
org.eclipse.jdt.apt.processorOptions/consoleLog=true
org.eclipse.jdt.apt.reconcileEnabled=true
```

16.8.7. .factorypath file to enable APT in Eclipse IDE

```
<!-- Replace 'XXX-VERSION-XXX' by the respective Scout RT version -->
<factorypath>
    <factorypathentry kind="VARJAR" id=
"M2_REPO/org/eclipse/scout/rt/org.eclipse.scout.jaxws.apt/XXX-VERSION-
XXX/org.eclipse.scout.jaxws.apt-XXX-VERSION-XXX.jar" enabled="true" runInBatchMode=
"false"/>
    <factorypathentry kind="VARJAR" id=
"M2_REPO/com/unquietcode/tools/jcodemodel/codemodel/1.0.3/codemodel-1.0.3.jar"
enabled="true" runInBatchMode="false"/>
    <factorypathentry kind="VARJAR" id=
"M2_REPO/org/eclipse/scout/rt/org.eclipse.scout.rt.platform/XXX-VERSION-
XXX/org.eclipse.scout.rt.platform-XXX-VERSION-XXX.jar" enabled="true" runInBatchMode=
"false"/>
    <factorypathentry kind="VARJAR" id=
"M2_REPO/org/eclipse/scout/rt/org.eclipse.scout.rt.server.jaxws/XXX-VERSION-
XXX/org.eclipse.scout.rt.server.jaxws-XXX-VERSION-XXX.jar" enabled="true"
runInBatchMode="false"/>
    <factorypathentry kind="VARJAR" id="M2_REPO/javax/servlet/javax.servlet-
api/3.1.0/javax.servlet-api-3.1.0.jar" enabled="true" runInBatchMode="false"/>
    <factorypathentry kind="VARJAR" id="M2_REPO/org/slf4j/slf4j-api/1.7.12/slf4j-api-
1.7.12.jar" enabled="true" runInBatchMode="false"/>
</factorypath>
```

16.8.8. Authentication Method

The authentication method specifies the protocol to challenge the webservice client to provide credentials.
Scout provides an implementation for *BASIC* and *WSSE_UsernameToken*. You can implement your own authentication method by implementing `IAuthenticationMethod` interface.

BasicAuthenticationMethod

Authentication method to apply *Basic Access* Authentication. This requires requests to provide a valid user name and password to access content. User's credentials are transported in HTTP headers. Basic authentication also works across firewalls and proxy servers.

However, the disadvantage of Basic authentication is that it transmits unencrypted base64-encoded passwords across the network. Therefore, you only should use this authentication when you know that the connection between the client and the server is secure. The connection should be established either over a dedicated line or by using Secure Sockets Layer (SSL) encryption and Transport Layer Security (TLS).

WsseUsernameTokenMethod

Authentication method to apply *Message Level WS-Security with UsernameToken* Authentication. This requires requests to provide a valid user name and password to access content. User's credentials are included in SOAP message headers.

However, the disadvantage of WSSE UsernameToken Authentication is that it transmits unencrypted passwords across the network. Therefore, you only should use this authentication when you know that the connection between the client and the server is secure. The connection should be established either over a dedicated line or by using Secure Sockets Layer (SSL) encryption and Transport Layer Security (TLS).

ContainerBasedAuthenticationMethod

Use this authentication method when using container based authentication, meaning that webservice requests are authenticated by the application server, or a Servlet filter.

16.8.9. Credential Verifier

Verifies user's credentials against a data source like *database*, *config.properties*, *Active Directory*, or others.
Scout provides an implementation for verification against users in *config.properties*. You can implement your own verifier by implementing `ICredentialVerifier` interface.

If you require to run in a specific `RunContext` like a transaction for user's verification, annotate the verifier with `RunWithRunContext` annotation, and specify a `RunContextProducer` accordingly.

ConfigFileCredentialVerifier

Credential verifier against credentials configured in *config.properties* file.

By default, this verifier expects the passwords in 'config.properties' to be a hash produced with SHA-512 algorithm. To support you in password hash generation, `ConfigFileCredentialVerifier` provides a static Java main method.

Credentials are loaded from property `scout.auth.credentials`. Multiple credentials are separated with the semicolon, username and password with the colon. If using hashed passwords (by default),

the password's salt and hash are separated with the dot.

To work with plaintext passwords, set the property `scout.auth.credentials.plaintext` to *true*.

Example of hashed passwords: `scott:SALT.PASSWORD-HASH;jack:SALT.PASSWORD-HASH;john:SALT.PASSWORD-HASH` Example of plaintext passwords: `scott:*;jack:;john:*`

Chapter 17. HTML UI

17.1. Browser Support

The Scout HTML UI requires a web browser with modern built-in technologies: HTML 5, CSS 3, JavaScript (ECMAScript 5). Scout does its best to support all browsers widely in use today by making use of vendor-specific prefixes, polyfills or other workarounds. However, some older or obscure browsers are not supported deliberately, simply because they are lacking basic capabilities or the required effort would be beyond reason.

Here is a non-exhaustive list of supported browsers:

Desktop

- Mozilla Firefox >= 35
- Google Chrome >= 36
- Microsoft Internet Explorer >= 9 (preferrably 10 or later, see limitations listed below)
- Microsoft Edge >= 12
- Apple Safari >= 7

Mobile

(Due to the nature of mobile operating systems, it is hard to specify exact versions of supported browsers. Usually, the screen size and the device speed are the limiting factors.)

- iOS >= 8
- Android >= 5
- Windows Mobile >= 10

Table 4. Known Limitations

Affected System	Description
Internet Explorer 9	**Drag and drop** of files from disk to the file chooser is not possible. IE9 is missing the required API (and it cannot be added with a polyfill). You can also only **upload one file at a time**, because the File API is missing as well.
Internet Explorer 9	**On-field labels** are not visible in IE9 because it lacks support for the "placeholder" attribute.
Internet Explorer 9	Some **loading animations** may appear static, because IE9 does not support CSS3 animations.
Internet Explorer 9	Tables and trees with many entries tend to degrade **performance** in IE9.

Affected System	Description
Internet Explorer	If the browser is configured to enable the so-called **"protected mode"**, the state of a popup window cannot be determined correctly. This is noticeable when a `AbstractBrowserField` has the property *"show in external window"* set to `true`. Even though the popup window is still open, the method `execExternalWindowStateChanged()` is called immediately, telling you the window was closed (because IE reports so). There is no workaround for this problem, apart from disabling the "protected mode".
Internet Explorer and Edge	**Performance in popup windows** (e.g. opening a form with `DISPLAY_HINT_POPUP_WINDOW`) is very poor. We have filed a bug with the folks at Microsoft in 2015, but unfortunately the issue is still unresolved. To prevent slow forms in IE, they should be using a different display hint. Alternatively, users can use a different browser.

Chapter 18. Scout JS

A classic Scout application has a client model written in Java, and a UI which is rendered using JavaScript. With this approach you can write your client code using a mature and type safe language. Unfortunately you cannot develop applications which have to run offline because the UI and the Java model need to be synchronized.

With Scout JS this will change. You will be able to create applications running without a UI server because there won't be a Java model anymore. The client code will be written using JavaScript and executed directly in the browser.

You still don't have to care about how the model will be rendered. There isn't a strict separation of the UI and the model anymore, but that doesn't mean you have to write HTML and CSS. You can of course, if you want to. But typically you will be using the Scout widgets you already know.

Scout JS is used in classic Scout applications as well. This means if you understand the concepts of Scout JS, writing custom widgets will be a lot easier.

18.1. Widget

A widget is a component which may be rendered. It may be simple like a label, or more complex like a tree or table. A form is a widget and a form field, too. A widget contains the model, which represents the state of the widget. In a Scout Classic application, that model will be sent from the UI server to the browser and the Scout UI will use that model to create the widget. In a Scout JS app, the model may be provided using JSON or directly with JavaScript.

18.1.1. Lifecycle

Every widget has a lifecycle. After a widget is instantiated, it has to be initialized using `init`. If you want to display it, you have to call the `render` method. If you want to remove it from the DOM, call the `remove` method. Removing a widget is not the same as destroying it. You can still use it, you can for example change some properties and then render it again. If you really don't need it anymore, call the `destroy` method.

So you see the widget actually has 3 important states:

- initialized
- rendered
- destroyed

The big advantage of this concept is that the model of the widget may be changed any time, even if the widget is not rendered. This means you can prepare a widget like a form, prepare all its child widgets like the form fields, and then render them at once. If you want to hide the form, just remove it. It won't be displayed anymore, but you can still modify it, like changing the label of a field or adding rows to a table. The next time it is rendered the changes will be reflected. If you do such a modification when it is rendered, it will be reflected immediately.

Destroying a widget means it will detach itself from the parent and destroy all children. If you have

attached listeners to other widgets at initialization time, now is the time to detach them. After a widget is destroyed it cannot be used anymore. Every attempt will result in a `Widget is destroyed` error.

18.1.2. Creating a Widget

A widget may be created using the constructor function or `scout.create`. Best practice is to always use `scout.create` which gives you two benefits:

1. You don't have to call `init` by yourself.
2. The widget may be extended (see Section 18.2 for details).

The following example creates a `StringField`.

Listing 96. Creating a string field

```
var field = scout.create('StringField', {
  parent: groupBox,
  label: 'hello',
  value: 'world'
};
```

The first parameter is the object type, which typically is the name of the constructor function preceded by the name space. `StringField` belongs to the `scout` name space which is the default and may be omitted. If the string field belonged to another name space called `mynamespace`, you would have to write the following:

Listing 97. Creating a field considering the name space

```
scout.create('mynamespace.StringField', {})
```

The second parameter of scout.create is the model. The model actually is the specification for your widget. In case of the `StringField` you can specify the label, the max length, whether it is enabled and visible and more. If you don't specify them, the defaults are used. The only needed property is the `parent`. To see what the defaults are, have a look at the source code of the widget constructor.

Every widget needs a parent. The parent is responsible to render (and remove) its children. In the example above, the parent is a group box. This group box has a property called `fields`. If the group box is rendered, it will render its fields too.

You don't need a group box to render the string field, you could render it directly onto the desktop. But if you want to use a form, you need a group box and create the form, group box and the field. Doing this programmatically is time consuming, that is why we suggest to use the declarative JSON based approach.

18.1.3. Creating a Widget using JSON

Using the JSON based approach makes it easier to specify multiple widgets in a structured way. The following example defines a form with a group box and a string field.

Listing 98. A form model defined using JSON

```
{
  "id": "example.MyForm",
  "type": "model",
  "title": "My first form!",
  "rootGroupBox": {
    "id": "MainBox",
    "objectType": "GroupBox",
    "borderDecoration": "empty",
    "fields": [
      {
        "id": "MyStringField",
        "objectType": "StringField",
        "label": "hello",
        "value": "world"
      }
    ]
  }
}
```

This description of the form is put in a separate file called `MyForm.json`. Typically you would create a file called `MyForm.js` as well, which contains the logic to interact with the fields. But since we just want to open the form it is not necessary. Instead you can use the following code to create the form:

```
scout.create('Form', scout.models.getModel('example.MyForm', parent));
```

When the form is shown it will look like this:

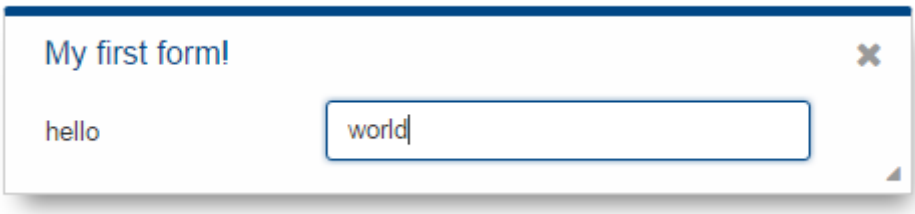


Figure 15. A form described using JSON

18.1.4. Properties

As seen before, every widget has a model representing its state. This model is written onto the widget at the time it is being instantiated. The properties of that model are now available as properties of the widget. So in order to access such a property, just call `widget.yourProperty`. If you want to modify the property, just call `widget.setYourProperty(value)`.

Listing 99. Accessing and modifying a property

```
var field = scout.create('scout.StringField', {
  parent: scout.sessions[0].desktop,
  labelVisible: false
});
console.log(field.labelVisible); // prints false

field.setLabelVisible(true);
console.log(field.labelVisible); // prints true
```

It is important to always use the setter to modify a property, because calling it does not just change the value. Instead it will call the method `setProperty(propertyName, value)` which does the following:

1. It will check if the property has changed at all. If the value is still the same, nothing happens. To compare the values `scout.objects.equals` is used, which uses `===` to compare and if that returns false, uses the equals methods of the given objects, if available.
2. If the values are not equal, the model is updated using the method `_setProperty` (notice the _). Beside setting the value it also notifies every listener about the property change. So if another widget is interested in that property it may attach a listener and will be informed on every property change (see also the Events for details).
3. In order to reflect the property change in the UI, the `_render` method is called, if available. The name of this method depends on the property name, it always starts with `_render` and ends with the property name. Example: `_renderLabelVisible`. If the widget does not implement such a method, nothing happens.

It is worth to mention that the behavior of step 2 may be influenced by the widget. If the widget provides a method called `_setPropertyName` (e.g. `_setLabelVisible`, notice the _), that method will be called instead of `_setProperty`. This may be useful if something other should be done beside setting the property. If that is the case, that new function is responsible to call `_setProperty` by itself in order to set the property and inform the listeners. That method may also be called by the `_init` method to make sure the additional code is also executed during initialization (calling the public setter in `_init` would not have any effect due to the equals check at the beginning).

18.1.5. Widget Properties

A widget property is a special kind of a property which references another widget.

Defining a property as widget property has the benefit that the widget is created automatically. Lets take the group box as an example. A group box has a widget property called fields. The fields are widgets, namely form fields. If I create a group box, I may specify its fields directly:

Listing 100. Creating the string field automatically using a widget property

```
var groupBox = scout.create('GroupBox', {
  parent: scout.sessions[0].desktop,
  label: 'My Group Box',
  fields: [{
    objectType: 'StringField',
    label: 'My String Field'
  }]
});
// check if the string field was created as well
console.log(groupBox.fields[0] instanceof scout.StringField);
```

In the above example the group box is created using `scout.create`. After creating the group box you can access the property fields and you will notice that the string field was created as well, even though `scout.create` has not been called explicitly for the string field. This is because the property `fields` is defined as widget property. During the initialization of the group box it sets the property `fields` and because the value is not a widget yet (resp. the elements in the array), `scout.create` will be called.

This will also happen if you use a setter of a widget property. You can either call the setter with a previously created widget, or just pass the model and the widget will be created automatically.

In addition to creating widgets, calling such a setter will also make sure that obsolete widgets are destroyed. This means if the widget was created using the setter, it will be destroyed when the setter is called with another widget which replaces the previous one. If the widget was created before calling the setter, meaning the `owner` is another widget, it won't be destroyed.

So if a property is defined as widget property, calling a setter will do the following:

1. It checks if the property has changed at all (same as for regular properties).
2. If the values are not equal, `_prepareWidgetProperty` is called which checks if the new value already is a widget and if not creates it. It also destroys the old widget unless the property should not be preserved (see `_preserveOnPropertyChangeProperties`). If the value is an array, it does so for each element in the array (only widgets which are not part of the new array will be destroyed).
3. If the widget is rendered, the old widget is removed unless the property should not be preserved. If there is a custom remove function (e.g. _removeXY where XY is the property name), it will be called instead of removing the widgets directly. Note that the widget may have already been removed by the destroy function at the prepare phase.
4. The model is updated (same as for regular properties).
5. The render method is called (same as for regular properties).

18.1.6. Events

Every widget supports event handling by using the class `scout.EventSupport`. This allows the widgets to attach listeners to other widgets and getting informed when an event happens.

The 3 most important methods are the following:

1. `on`: adds a listener
2. `off`: removes a listener
3. `trigger`: triggers an event

So if a widget is interested in an event of another widget, it calls the function `on` with a callback function as parameter. If it is not interested anymore, it uses the function `off` with the same callback function as parameter.

The following example shows how to handle a button click event.

Listing 101. Handling an event

```
var button = scout.create('scout.Button', {
  parent: scout.sessions[0].desktop,
  label: 'click me!'
});
button.render();
button.on('click', function(event) {
  // print 'Button "click me!" has been clicked'
  console.log('Button "' + event.source.label + '" has been clicked');
});
```

Every click on the button will execute the callback function. To stop listening, you could call `button.off('click')`, but this would remove every listener listening to the 'click' event. Better is to pass the same reference to the callback used with `on` as parameter for `off`.

Listing 102. Stop listening for an event

```
var button = scout.create('scout.Button', {
  parent: scout.sessions[0].desktop,
  label: 'click me!'
});
button.render();
var callback = function(event) {
  // print 'Button "click me!" has been clicked'
  console.log('Button "' + event.source.label + '" has been clicked');

  // stop listening, a second click won't print anything
  button.off('click', callback);
};
button.on('click', callback);
```

If the callback function is bound using `bind()`, the bound function has to be used when removing the listener using `off`. This is because `bind()` returns a new function wrapping the original callback.

In order to trigger an event rather than listening to one, you would use the function `trigger`. This is

what the button in the above example does. When it is being clicked, it calls `this.trigger('click')` (`this` points to the instance of the button). With the second parameter you may specify additional data which will be copied onto the event. By default the event contains the type (e.g. 'click') and the source which triggered it (e.g. the button).

Listing 103. Triggering an event with custom event data

```
trigger('click', {
  foo: 'bar'
});

// callback
function(event) {
  console.log(event.foo); // prints bar
}
```

Property Change Event

A special kind of event is the property change event. Whenever a property changes, such an event is triggered.

The event has the following properties:

1. `type`: the type of the event which is always `propertyChange`
2. `source`: the widget which triggered the event
3. `name`: the name of the property
4. `newValue`: the new value of the property
5. `oldValue`: the old value of the property

Listening to such an event works in the same way as for other events, just use the type `propertyChange`. The listening below shows how to handle the property change event if the `selected` property of a toggle button changes.

Listing 104. Example of a property change event

```
var button = scout.create('scout.Button', {
  parent: scout.sessions[0].desktop,
  label: 'click me!',
  displayStyle: scout.Button.DisplayStyle.TOGGLE
});
button.render();
button.on('propertyChange', function(event) {
  // prints 'Property selected changed from false to true'
  console.log('Property ' + event.propertyName + ' changed from ' + event.oldValue + '
to ' + event.newValue);
});
button.setSelected(true);
```

18.1.7. Icons

See chapter Chapter 5 for a general introduction to icons in Scout.

Widgets that have a property `iconId` (for instance `Menu`) can display an icon. This iconId references an icon which can be either a bitmap image (GIF, PNG, JPEG, etc.) or a character from an icon-font. An example for an icon-font is the *scoutIcons.ttf* which comes shipped with Scout.

Depending on the type (image, font-icon) the iconId property references:

- **Image**: iconId is an URL which points to an image resource accessible via HTTP.

  Example: `/icons/person.png`

- **Font-icon**: iconId has the format `font:[UTF-character]`.

  Example: `font:\uE043`, references a character in *scoutIcons.ttf*

  Example: `font:fooIcons \uE109`, references a character in custom font *fooIcons.ttf*

- **Icon Constants**: iconId has the format: `${iconId:[constant]}`, where constant is a constant in `scout.icons`. This format is especially useful when you configure a Scout widget with a JSON model. The value of the constant is again either an image or a font-icon as described above.

  Example: `${iconId:ANGLE_UP}` uses `scout.icons.ANGLE_UP`, icons predefined by Scout

  Example: `${iconId:foo.BAR}` uses `foo.icons.BAR`, use this for custom icon constant objects

18.2. Object Factory

As seen in the Section 18.1.2 a widget may be created using `scout.create`. When using this function, the call is delegated to the `ObjectFactory`.

The `ObjectFactory` is responsible to create and initialize a Scout object. A typical Scout object has an `objectType` and an `init` function. But actually any kind of object with a constructor function in the scout or a custom namespace may be created.

By default, objects are created using naming convention. This means when calling `scout.create('scout.Table', model)`, the `scout` namespace is searched for a class called `Table`. Since `scout` is the default namespace, it may be omitted. So calling `scout.create('Table', model)` has the same effect. If there is such a class found, it will be instantiated and the `init` function called, if there is one. The `model` is passed to that `init` function. So instead of using `scout.create` you could also use the following code:

Listing 105. Creating an object without the ObjectFactory

```
var table = new scout.Table();
table.init(model);
```

This will work fine, but you will loose the big benefit of the `ObjectFactory`: the ability to replace

existing classes. So if you want to customize the default `Table`, you would likely extend that table and override some functions. In that case you need to make sure every time a table is created, your class is used instead of the default. To do that you have to register your class in the `ObjectFactory` with the `objectType Table`. If `scout.create('Table')` is called the object factory will check if there is a class registered for the type `Table` and, if yes, that class is used. Only if there is no registration found, the default approach using the naming convention is performed.

In order to register you class, you need a file called `objectFactories` and add that to your JavaScript module (e.g. yourproject-module.js). The content of that file may look as following:

Listing 106. Adding a new object factory registration

```
scout.objectFactories = $.extend(scout.objectFactories, {
  'Table': function() {
    return new yourproject.CustomTable();
  }
});
```

This will simply add a new factory for the type `Table` the the list of existing factories. From now on `yourproject.CustomTable` will be instantiated every time a `Table` should be created.

18.3. Form

A form is typically used for two purposes:

1. Allowing the user to enter data in a structured way
2. Displaying the data in a structured way

This is achieved by using Section 18.4s. Every form has one root group box (also called main box) which has 1:n form fields. The form fields are layouted using the logical grid layout, unless no custom layout is used. This makes it easy to arrange the fields in a uniform way.

A form may be displayed in various ways, mainly controlled by the property `displayHint`. The following display hints are available by default:

- **view**: the form will be opened in a tab and will take full width and height of the bench
- **dialog**: the form will be opened as overlaying dialog and will be as width and height as necessary
- **popup-window**: the form will be opened in a separate browser window (please note that this feature does not work properly with Internet Explorer)

To display the form, just set one of the above display hints and call `form.open()`.

Beside opening the form as separate dialog or view, you can also embed it into any other widget because it is actually a widget by itself. Just call `form.render()` for that purpose.

18.3.1. Form Life Cycle

When working with forms, you likely want to load, validate and save data as well. The form uses a so called `FormLifecycle` to manage the state of that data. The life cycle is installed by default, so you don't have to care about it. So whenever the user enters some data and presses the save button, the data is validated and if invalid, a message is shown. If it is valid the data will be saved. The following functions of a form may be used to control that behavior.

- **open**: displays the form and calls `load`.
- **load**: calls `_load` and `importData` which you can implement to load the data and then marks the fields as saved to set their initial values. Finally, a `postLoad` event is fired.
- **save**: validates the data by checking the mandatory and validation state of the fields. If every mandatory field is filled and every field contains a valid value, the `exportData` and `_save` functions are called which you can implement to save the data. After that every field is marked as saved and the initial value set to the current value.
- **reset**: resets the value of every field to its initial value marking the fields as untouched.
- **ok**: saves and closes the form.
- **cancel**: closes the form if there are no changes made. Otherwise it shows a message box asking to save the changes.
- **close**: closes the form and discards any unsaved changes.
- **abort**: called when the user presses the "x" icon. It will call `close` if there is a close menu or button, otherwise `cancel`.

If you embed the form into another widget, you probably don't need the functions `open`, `ok`, `close`, `cancel` and `abort`. But `load`, `reset` and `save` may come in handy as well.

Because it is quite common to have a button activating one of these functions (like an 'ok' or 'cancel' button), the following buttons (resp. menus because they are used in the menu bar) are available by default: `OkMenu`, `CancelMenu`, `SaveMenu`, `ResetMenu`, `CloseMenu`.

18.4. Form Field

A form field is a special kind of a widget. It is mainly used on forms but may actually be added to any other widget.

Every form field contains of the following parts:

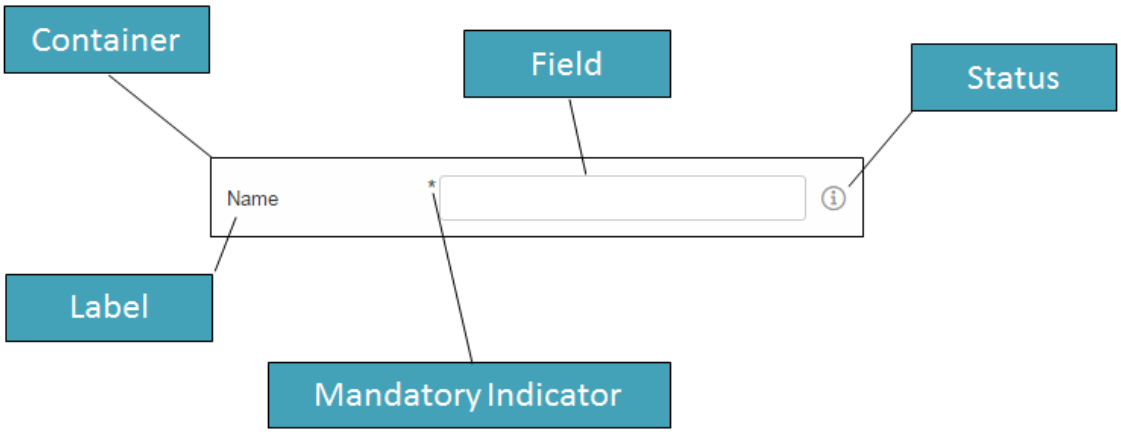


Figure 16. Parts of a form field

Typical form fields are `StringField`, `DateField` or `TableField`. All these fields have the API of `FormField` in common (like `setLabel()`, `setErrorStatus()`, etc.) but also provide additional API.

Some form fields are actually just a wrapper for another widget. This is for example the case for the `TableField`. The `Table` itself may be used stand-alone, just call `scout.create('Table', {})`. But if you want to use it in a `GroupBox`, which is a form field, you have to use a `TableField` wrapping the `Table`.

18.5. Value Field

A value field extends the form field by the ability to store a value. Typical form fields are `StringField`, `NumberField`, `DateField` or `SmartField`. All these fields provide a value which is accessible using `field.value` and may be set using `field.setValue(value)`.

18.5.1. Parser, Validator, Formatter

The value always has the target data type of the field. When using a `StringField` the type is `string`, when using a `NumberField` the type is `number`, when using a `DateField` the type is `date`. This means you don't have to care about how to parse the value from the user input, this will be done by the field for you. The field also validates the value, meaning if the user entered an invalid value, an error is shown. Furthermore, if you already have the value and want to show it in the field, you don't have to format the value by yourself.

This process of parsing, validating and formatting is provided by every value field. The responsible functions are `parseValue`, `validateValue` and `formatValue`. If a user enters text, it will be parsed to get the value with the correct type. The value will then be validated to ensure it is allowed to enter that specific value. Afterwards it will be formatted again to make sure the input looks as expected (e.g. if the user enters 2 it may be formatted to 2.0). If you set the vaue programmatically using `setValue` it is expected that the value already has the correct type, this means parse won't be executed. But the value will be validated, formatted and eventually displayed in the field.

Even though the fields already provide a default implementation of this functionality, you may want to extend or replace it. For that purpose you may set a custom parser and formatter or one or more validators.

Custom Parser and Formatter

Typically you don't have to add a custom parser or formatter for a `NumberField` or `DateField`. They work with a `DecimalFormat` or `DateFormat` which means you can specify a pattern how the number or date should be represented. By default, it uses the pattern of the current locale, so you don't even have to specify anything.

For a `StringField` on the other hand, adding a custom parser or formatter could make sense. Let's say you want to group the text into 4 digit blocks, so that if the user inputs 1111222233334444 it should be converted to 1111-2222-3333-4444. This could be done using the following formatter.

Listing 107. Example of a formatter

```
function formatter(value, defaultFormatter) {
  var displayText = defaultFormatter(value);
  if (!displayText) {
    return displayText;
  }
  return displayText.match(/.{4}/g).join('-');
};
```

Keep in mind that you should call the default formatter first unless you want to replace it completely.

To make your formatter active, just use the corresponding setter.

Listing 108. Setting the formatter

```
field.setFormatter(formatter);
```

Formatting the value is most of the time only half the job. You probably want to set a parser as well, so that if the user enters the text with the dashes it will be converted to a string without dashes.

Listing 109. Example of a parser

```
function parser(displayText, defaultParser) {
  if (displayText) {
    return displayText.replace(/-/g, '');
  }
  return defaultParser(displayText);
};
```

Use the corresponding setter to activate the parser.

Listing 110. Setting the parser

```
field.setParser(parser);
```

Custom Validator

The purpose of a validator is to only allow valid values. This mostly depends on your business rules, this is why the default validators don't do a whole lot of things.

See the following example of a validator used by a `DateField`.

Listing 111. Example of a validator

```
function(value) {
  if (scout.dates.isSameDay(value, new Date())) {
    throw 'You are not allowed to select the current date';
  }
  return value;
};
```

This validator ensures that the user may not enter the current date. If he does, en error status will be shown on the right side of the date field saying 'You are not allowed to select the current date'.

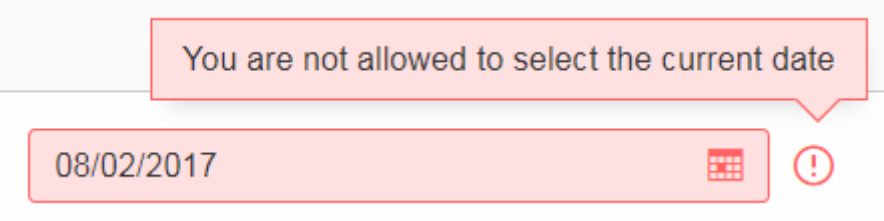


Figure 17. Validation error of a date field

As you can see in the example, in order to mark a value as invalid just throw the error message you want to show to the user. You could also throw an error or a `scout.Status` object. In that case a generic error message will be displayed.

In order to activate your validator, you can either call `setValidator` to replace the existing validator. In that case you should consider calling the default validator first, like you did it for the formatter or parser. Or you can use `addValidator` which adds the validator to the list of validators of the field.

Listing 112. Adding a validator

```
field.addValidator(validator);
```

Compared to parse and format you may have multiple validators. When the value is validated, every validator is called and has to aggree. If one validation fails, the value is not accepted. This should make it easier to reuse existing validators or separate your validation into tiny validators according to your business rules.

If you now ask yourself, why this is not possible for parsing and formatting, consider the following: `Validate` takes a value and returns a value, the data type is the same for input and output. `Parse` takes a text and creates a value, `format` takes a value and creates a text. The data type is likely not the same (besides for the `StringField`). If you had multiple parsers, the output of the previous parser would be the input of the next one, so depending on the index of your parser you would either get the text or the already parsed value as input. Confusing, isn't it? So in order to keep it

simple, there is only one parser and only one formatter for each field.

18.6. Extensibility

18.6.1. How to extend Scout objects

The extension feature in Scout JS works by wrapping functions on the prototype of a Scout object with a wrapper function which is provided by an extension. The extension feature doesn't rely on subclassing, instead we simply register one or more extensions for a single Scout class. When a function is called on an extended object, the functions are called on the registered extensions first. Since a Scout class can have multiple extensions, we speak of an extension chain, where the last element of the chain is the original (extended) object.

The base class for all extensions is `scout.Extension`. This class is used to extend an existing Scout object. In order to use the extension feature you must subclass scout.Extension and implement an `init` function, where you register the functions you want to extend. Example:

```
scout.MyExtension.prototype.init = function() {
  this.extend(scout.MyStringField.prototype, '_init');
  this.extend(scout.MyStringField.prototype, '_renderProperties');
};
```

Then you implement functions with the same name and signature on the extension class. Example:

```
scout.MyExtension.prototype._init = function(model) {
  this.next(model);
  this.extended.setProperty('bar', 'foo');
};
```

The extension feature sets two properties on the extension instance before the extended method is called. These two properties are described below. The function scope (this) is set to the extension instance when the extended function is called.

next

is a reference to the next extended function or the original function of the extended object, in case the current extension is the last extension in the extension chain.

extended

is the extended or original object.

All extensions must be registered in the `installExtensions` function of your `scout.App`. Example:

```
scout.MyApp.prototype.installExtensions = function() {
  scout.Extension.install([
    'scout.FooExtension',
    'scout.BarExtension',
    // ...
  ]);
};
```

18.6.2. Export Scout model from a Scout classic (online) application

With the steps described here you can export model-data (e.g. forms, tables, etc.) from an existing classic, online Scout application into JSON format which can be used in a Scout JS application. This is a fast and convenient method to re-use an existing form in a Scout JS application, because you don't have to build the model manually. Here's how to use the export feature:

- Activate the `TextKeyTextProviderService` in your Scout classic application. You can do this either by calling the static `register` Method at runtime (using the Debugger) or by setting the config property *scout.text.providers.show.keys*. Once the TextProvider is active, it returns text keys instead of localized texts. The format of the returned string is `${textKey:[text-key]}`. Strings in this format are interpreted browser side by Scout JS and are resolved and localized in *texts.js*.
- Call the Scout classic web application with the URL parameter `?adapterExportEnabled=true`. This parameter is checked by Scout JS and the effect of it is that Scout JS keeps adapter-data loaded from the server, so we can use it for a later export operation. Usually Scout JS deletes adapter-data right after it has been used to create an adapter instance.
- Start your browsers developer tools (F12) from your running Scout classic app, inspect the form or another adapter you'd like to export, and search for the ID of that adapter by looking at the `data-id` attribute in the DOM. Then call the following JavaScript function from the console: `JSON.stringify(scout.exportAdapter([id]))`. The console will now output the JSON code for that adapter. Copy/paste that string from the console and tidy it up with a tool of your choice (for instance jsoneditoronline.org).
- Now the formatted JSON code is ready to be used in your Scout JS project. Simply store the JSON to a .json File and load it in the `_jsonModel` function of your widget by calling `scout.models.getModel('[ID of json model]')`. Most likely you must edit the JSON file and apply some changes manually. For instance:
  - Replace numeric adapter IDs by human-readable, semantic string IDs. You will need these IDs to reference a model element from the JSON model in your JavaScript code, e.g. to register a listener on a Menu.
  - Maybe you'll have to adjust objectType and namespace of some widgets in the JSON. This needs to be done for widgets that do not belong to the default *scout* namespace.
  - Remove form fields and menus which are never used in your Scout JS app
  - Remove unused menu types from the menuTypes property of a scout.Menu

Note: The function exportAdapter can be used not only for forms, but for any other widget/adapter too.

Appendix A: Licence and Copyright

This appendix first provides a summary of the Creative Commons (CC-BY) licence used for this book. The licence is followed by the complete list of the contributing individuals, and the full licence text.

A.1. Licence Summary

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/3.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

A summary of the license is given below, followed by the full legal text.

You are free:

- **to Share** ---to copy, distribute and transmit the work
- **to Remix**---to adapt the work
- to make commercial use of the work

Under the following conditions:

Attribution ---You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:

Waiver ---Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain ---Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights ---In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice ---For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to https://creativecommons.org/licenses/by/3.0/.

A.2. Contributing Individuals

In the text below, all contributing individuals are listed in alphabetical order by name. For

contributions in the form of GitHub pull requests, the name should match the one provided in the corresponding public profile.

Bresson Jeremie, Fihlon Marcus, Nick Matthias, Schroeder Alex, Zimmermann Matthias

A.3. Full Licence Text

The full licence text is available online at http://creativecommons.org/licenses/by/3.0/legalcode

Do you want to improve this document? Have a look at the sources on GitHub.