Reference Manual

Version 1.0.11 (Work In Progress)

Jevgeni Kabanov, Rein Raudjärv, Toomas Römer, Taimo Peelo

Table of Contents

Chapter 1. Introduction

1.1. Overview

Aranea is a *Java Hierarchical Model-View-Controller Web Framework* that provides a common simple approach to building the web application components, reusing custom or general GUI logic and extending the framework. The framework is assembled from a number of independent modules with well-defined responsibilities and thus can be easily reconfigured to perform new and unexpected tasks. The controller is separated into a hierarchy of components that can react to user or system events. The framework is completely view agnostic, but provides a thorough library of JSP custom tags that target building GUIs without writing a line of HTML. All components and modules are simple *Plain Old Java* classes without any XML mappings and thus usual Object-Oriented design techniques can be applied. Aranea manages the component field persistence automatically and inherently supports nested state.

Aranea is logically separated in the following modules:

Aranea Core
: Contains the core interfaces and base implementations that define Aranea base abstractions and their contracts. Includes packages `org.araneaframework` and `org.araneaframework.core` and is packaged into `aranea-core.jar`.

Aranea Framework
: Framework module sits on top of the *Core* module and contains the implementation of the Aranea Web Framework that does not directly depend on any container. *Framework* module includes package `org.araneaframework.framework` and its subpackages and is packaged into `aranea-framework.jar`.

Aranea HTTP
: HTTP module extends the *Framework* module with services that use a Servlet container. Servlet module includes package `org.araneaframework.http` and its subpackages and is packaged into `aranea-servlet.jar`.

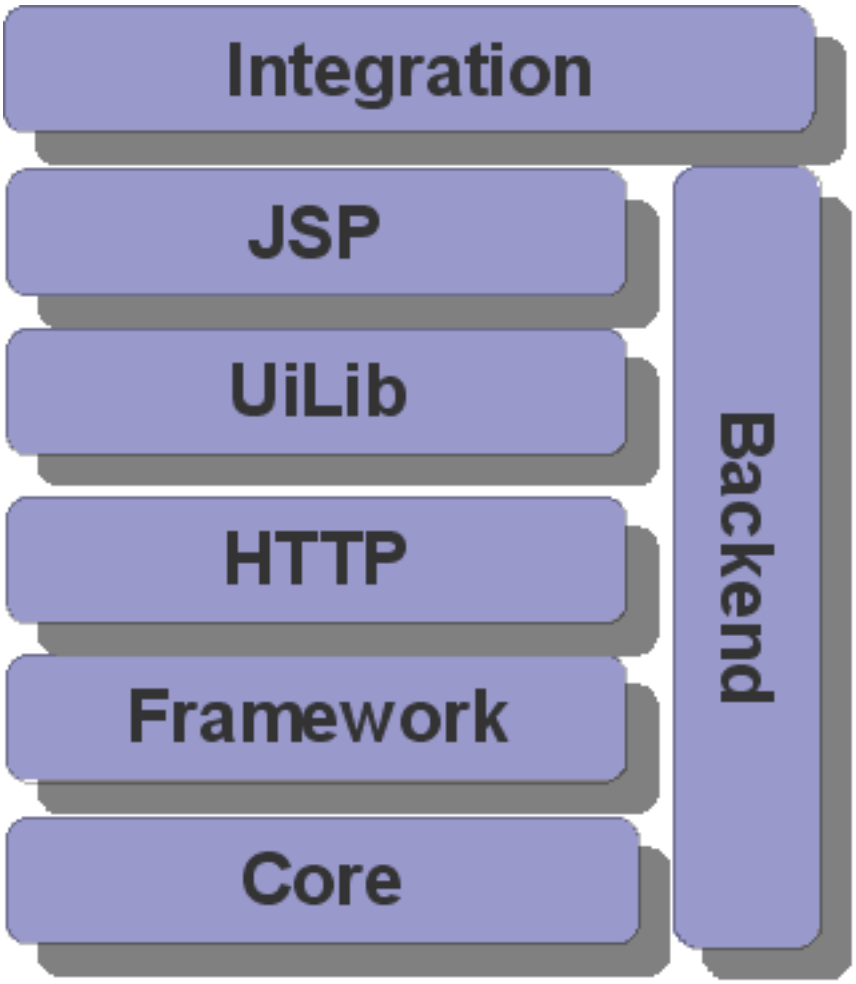
Aranea Integration
: Spring module integrates Aranea with the *Spring* IoC container. Spring module includes package `org.araneaframework.integration.spring` and its subpackages and is packaged into `aranea-spring.jar`.

Aranea UiLib
: UiLib module contains reusable GUI widgets and supporting API. UiLib module includes package `org.araneaframework.uilib` and its subpackages and is packaged into `aranea-uilib.jar`.

Aranea JSP
: JSP module contains a custom tag library, including tags that render *UiLib* widgets. JSP module includes package `org.araneaframework.jsp` and its subpackages and is packaged into `aranea-jsp.jar`.

Aranea Backend
: Backend module contains supporting classes that are to be used in the application service layer (e.g. backend list data provider helper classes). Backend module includes package `org.araneaframework.backend` and its subpackages and is packaged into `aranea-backend.jar`.

These modules depend on each other as follows:



1.2. Organization

The rest of this manual is organized as follows:

Components, Widgets and Services
: This chapter describes the core Aranea abstractions in detail generally not necessary to just develop application code so it can be skipped during the first reading. It is quite dry on the examples, but its understanding is crucial to develop Aranea extensions. To get a quick understanding of how to program with widgets read Section 2.7, “Application Widgets”.

Framework and Configuration
: This chapter describes how to assemble and configure both applications and the Aranea framework itself. It also describes in detail main components of the Aranea framework. The most interesting part for a beginner would be Section 3.2, “Application Configuration”.

JSP and Custom Tags
: This chapter describes how to render Aranea widgets and services with custom JSP tag library supplied in the Aranea distribution.

Forms and Data Binding
: This chapter describes how Aranea manages reading data from request, validating and converting it to the model objects.

Lists and Query Browsing
: This chapter describes how to make pageable, filterable and orderable tables in Aranea.

Third-party Integration
: This chapter describes Aranea integration hooks for third-party toolkits and frameworks. At the moment it includes Spring.

Javascript Libraries
: This chapter describes the Javascript libraries that Aranea uses and the Javascript API that Aranea provides.

Chapter 2. Components, Widgets and Services

2.1. Introduction

Aranea framework and component model are very simple and implemented purely in Plain Old Java. There are no XML mappings, code generation or bytecode enhancement. The whole component model consists mainly of five interfaces: `org.araneaframework.Component`, `org.araneaframework.Service`, `org.araneaframework.Widget`, `org.araneaframework.Environment`, `org.araneaframework.Message` and some conventions regarding their usage and implementation.

This chapter describes the core Aranea abstractions in detail generally not necessary to just develop application code so it can be skipped during the first reading. It is quite dry on the examples, but its understanding is crucial to develop Aranea extensions. To get a quick understanding of how to program applications with widgets read Section 2.7, “Application Widgets”.

2.2. Coding Conventions

2.2.1. Checked versus Unchecked Exceptions

It is our firm belief that checked exceptions are unnecessary in *Controller* and therefore Aranea will in most cases allow to just declare your overriding method as `throws Exception`. On the other hand no framework interfaces throw checked exceptions so the exception-handling boilerplate can be delegated to a single error-handling component.

2.2.2. Public versus Framework Interfaces

Since the application programmer implements the same components that are used for framework extension, it is important to discourage the access to public framework interfaces (which are necessarily visible in the overridden classes). Thus a simple convention is applied for core framework interfaces, which is best illustrated with the following example.

```
public interface Service extends Component, Serializable {
  public Interface _getService();

  public interface Interface extends Serializable {
    public void action(Path path, InputData input, OutputData output) throws Exception;
  }
}
```

As one can see, the real interface methods are relocated to an inner interface named `Interface`, that can be accessed using a method `_get<InterfaceName>()`, which starts with an underscore to discourage its use. As a rule of a thumb, in Aranea the methods starting with an underscore should only be used, when one really knows what one is doing.

2.2.3. Components and Their Orthogonal Properties

Aranea has three main types of components: `org.araneaframework.Component`, `org.araneaframework.Service` and `org.araneaframework.Widget`. These components also have a number of orthogonal properties (like `Viewable`, `Composite`), which are represented by interfaces that need to be

implemented. Since some particular API methods expect a particular type of component with a particular property (e.g. `ViewableWidget`) one would either have to abandon static type safety or define a lot of meaningless interfaces that would clutter the Javadoc index and confuse the source code readers. The approach chosen in Aranea is to make such interfaces internal to the property, like in the following example.

```
public interface Viewable extends Serializable {
  public Interface _getViewable();

  interface Interface extends Serializable {
    public Object getViewModel() throws Exception;
  }

  public interface ViewableComponent extends Viewable, Component, Serializable {}
  public interface ViewableService extends ViewableComponent, Service, Serializable {}
  public interface ViewableWidget extends ViewableService, Widget, Serializable {}
}
```

2.3. Components and Environment

`org.araneaframework.Component` represents the unit of encapsulation and reuse in Aranea. Components are used to both provide plug-ins and extensions to the framework and to implement the actual application-specific code. A component has (possibly persistent) state, life cycle, environment and a messaging mechanism.

```
public interface Component extends Serializable {
  public Component.Interface _getComponent();

  public interface Interface extends Serializable {
    public void init(Environment env) throws Exception;
    public void destroy() throws Exception;
    public void propagate(Message message) throws Exception;
    public void enable() throws Exception;
    public void disable() throws Exception;
  }
}
```

The component life cycle goes as follows:

1. `init()` —notifies the component that it should initialize itself passing it the `Environment` . A component can be initialized only once and the environment it is given stays with it until it is destroyed.
2. All other calls (like `propagate()`) should be done when a component is alive, initialized and enabled.
3. `disable()` —notifies the component that it will be disabled and will not receive any calls until it is enabled again. A component is enabled by default.
4. `enable()` —notifies the component that it has been enabled again. This call may only be done after a `disable()` call.
5. `destroy()` —notifies the component that it has been destroyed and should release any acquired resources and such. A component can be destroyed only once and should be initialized before that.

Further in the text we will refer to an initialized and not destroyed component instance that has a parent as *live* and one that has not been disabled or has been re-enabled as *enabled*.

Aranea provides a base implementation of the `Component` — `org.araneaframework.core.BaseComponent`. This implementation mainly enforces contracts (including life cycle and some basic synchronization). A base class for application development `org.araneaframework.core.BaseApplicationComponent` is also available.

2.3.1. Composite Pattern and Paths

Composite pattern refers to a design approach prominent (specifically) in the GUI modeling when objects implementing the same interface are arranged in a hierarchy by containment, where the nodes of the tree

propagate calls in some way to the leafs of the tree. It is shown on Figure 2.1, “*Composite* design pattern”.

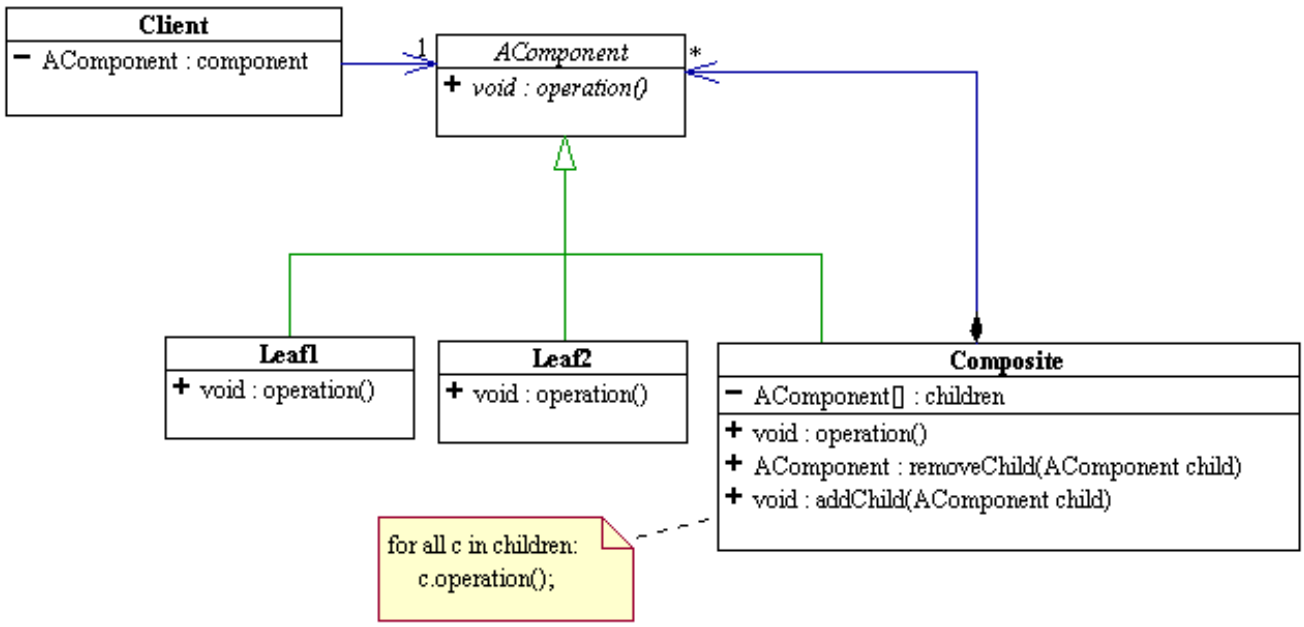


Figure 2.1. *Composite* design pattern

Composite is one of the main patterns used in Aranea. It is mainly used to create a *Hierarchical Controller* using `Component` containment. In terms of `Component` interface *Composite* is used to propagate life cycle events and route messages (see Section 2.3.3, “Messaging Components”).

The flavor of the *Composite* pattern as used in Aranea typically means that every contained component has some kind of an identifier or name that distinguishes it from other children of the same parent (note that the child is not typically aware of its identifier). This identifiers are used to route messages and events and can be combined to form a full identifier which describes a "path" from the root component to the child in question. This paths are represented by a *Iterator*-like interface `org.araneaframework.Path`.

```
public interface Path extends Cloneable, Serializable {
  public Object getNext();
  public Object next();
  public boolean hasNext();
}
```

Each `next()` call will return the identifier of the next child in the path to the descendant in question. Default implementation (`org.araneaframework.core.StandardPath`) uses simple string identifiers like "a" or "b" and combines them using dots forming full paths like "a.b.c".

A *Composite* component may want to make its children visible to the outside world by implementing the `org.araneaframework.Composite` interface:

```
public interface Composite extends Serializable {
  public Composite.Interface _getComposite();
  public interface Interface extends Serializable {
    public Map getChildren();
    public void attach(Object key, Component comp);
    public Component detach(Object key);
  }
```

```
}
```

This interface allows to both inspect and manipulate component children by attaching and detaching them from the parent component.

As most of the Aranea abstractions are built to be used with the *Composite* concept we will illustrate it in greater detail when examining other abstractions and their implementation. Further on we will assume that any `Component` has a parent that contains it and every child has some kind of name in relation to the parent unless noted otherwise (obviously there is at least one *Composite* that does not have a parent, but we don't really care about that at the moment).

2.3.2. Environment

`org.araneaframework.Environment` is another important concept that represents the way `Components` interact with the framework. `Environment` interface is rather simple:

```
public interface Environment extends Serializable {
  public Object getEntry(Object key);
}
```

It basically provides means of looking up entry objects by their key. A typical usage of the `Environment` can be illustrated with an example.

```
...
MessageContext msgCtx = (MessageContext) getEnvironment().getEntry(MessageContext.class);
msgContext.showInfoMessage("Hello world!");
...
```

As one can see from the example `Environment` will typically allow to look up implementations of the interfaces using their `Class` as the key (this is in fact an Aranea convention in using and extending the `Environment`). The interfaces serving as keys for `Environment` entries are referred to as *contexts*. It is thus not unlike JNDI or some other directory lookups that allow to hold objects, however unlike them `Environment` is very specific to the `Component` it is given to, and can be influenced by its parents. In fact, all contexts available in the `Environment` will be provided to the `Component` by its parents or ancestors (in the sense of containment rather than inheritance). Thus, two different `Components` may have completely different `Environments`.

A default implementation of `Environment` is `org.araneaframework.core.StandardEnvironment`. It provides for creating an `Environment` from a `java.util.Map`, or extending an existing environment with map entries.

A component can provide an environment entry to its descendant, by providing it to the initializer of its direct child. For instance the `MessageContext` could be provided by the following message component:

```
public class MessageFilterService implements MessageContext, Component, Service {
  protected Service childService;
  public void setChildService(Service childService) {
    this.childService = childService;
  }

  public void init(Environment env) {
    childService.init(
      new StandardEnvironment(env, MessageContext.class, this);
  }

  //MessageContext implementation...
  public String showInfoMessage(String message) {
    //Show message to user...
```

```
  }

  //...
}
```

After that the `childService`, its children and so on will be able to use the `MessageContext` provided by `MessageFilterService`. Of course this can be done simpler as shown in examples in chapter

2.3.3. Messaging Components

So far, we have looked at the component management and environment. However what makes the component hierarchy such a powerful concept is messaging. Basically, messaging allows us to send any events to any component in the hierarchy (including all components or a specific one). The messaging is incorporated using the `org.araneaframework.Message` interface

```
public interface Message extends Serializable {
  public void send(Object id, Component component) throws Exception;
}
```

and `Component.propagate(Message message)` method. The default behavior of the `propagate()` method should be to send the message to all component children, passing the `send()` method the identifier of the child and the child itself. It is up to the message what to do with the child further, but typically `Message` just calls the `propagate()` method of the child passing itself as the argument after possibly doing some custom processing (the *double-dispatch* OO idiom).

A standard `Message` implementation that uses double-dispatch to visit all the components in hierarchy is `org.araneaframework.core.BroadcastMessage`. It usage can be illustrated with the following example:

```
...
Message myEvent = new BroadcastMessage() {
  public void execute(Component component) throws Exception {
  if (component instanceof MyEventListener)
    ((MyEventListener) component).onMyEvent(data);
  }
}
myEvent.send(null, rootComponent);
...
```

This code will call all the components in the hierarchy that subscribed to the event and pass them a certain `data` parameter. As one can see, when calling `Message.send()` we will typically pass `null` as the first parameter, since it is needed only when propagating messages further down the hierarchy. Note that messages can be used to gather data from the components just as well as for passing data to them. For example one could construct message that gathers all `FormWidgets` from the widget hierarchy:

```
public static class FormWidgetFinderMessage extends BroadcastMessage {
  List formList = new ArrayList();

  protected void execute(Component component) throws Exception {
    if (component instanceof org.araneaframework.uilib.form.FormWidget) {
      formList.add(component);
    }
  }

  public List getAllForms() { return formList; }
}
```

Another standard `Message` implementation is `org.araneaframework.core.RoutedMessage`, which allows us to send a message to one specific component in the hierarchy as in the following example:

```
...
Message myEvent = new RoutedMessage("a.b.c") {
  public void execute(Component component) throws Exception {
    ((MyPersonalComponent) component).myMethod(...);
  }
}
myEvent.send(null, rootComponent);
...
```

This code will send the message to the specific component with path "a.b.c" and call `myMethod()` on it.

2.3.4. State and Synchronization

The handling of persistent state in Aranea is very simple. There are no scopes and every component state is saved until it is explicitly removed by its parent. This does not mean that all of the components are bound to the session, but rather that most components will live a period of time appropriate for them (e.g. framework components will live as long as the application lives, GUI components will live until user leaves them, and so on). This provides for a very flexible approach to persistence allowing not to clutter memory with unused components.

The end result is that typically one needs not worry about persistence at all, unless one is programming some framework plug-ins. All class fields (except in some cases `transient` fields) can be assumed to persist while the host object is *live*.

However such handling does not guarantee that the component state is anyhow synchronized. As a matter of fact most of the framework components outside the user session should be able to process concurrent calls and should take care of the synchronization themselves. However application components are typically synchronized by the framework. More information on the matter will follow in Section 2.7, “Application Widgets”.

2.4. InputData and OutputData

`InputData` is Aranea abstraction for a request, which hides away the Servlet API and allows us to run Aranea on different containers (e.g. in a portlet or behind a web service).

Both `InputData` and `OutputData` have a *scope*, which is information about the path to the current component. When they are propagated through the component hierarchy the following methods are used to construct the scope:

Method	Description
`pushScope(Object step)`	Adds child identifier to the end of the current scope, should *always* be used before the call to the child.
`Object popScope()`	Removes the last identifier from the scope, should *always* be used a call to a child.
`Path getScope()`	Returns the current scope as a `Path`.
`restoreScope(Path scope)`	Restore the scope back to a previously saved one.

Note

These methods should be used only when using *Composite* components, as framework components are scoped differently.

`InputData` also provides access to the data sent to the component. This data comes in two flavours:

- `getScopedData()` returns a `java.util.Map` with the data sent specially to this component, which is associated with the current scope.
- `getGlobalData()` returns a `java.util.Map` with the data sent to the application generally.

In case Aranea is running on top of a servlet both these maps will contain only `Strings` (or in case you submitted more than one value for a specific key it may contain a `String[]`). In case of the usual path and scope implementation (as dot-separated strings) *global data* will contain the submitted parameters with no dots in them and *scoped data* will contain the parameters prefixed with the current component scope string.

The main function of `OutputData` is to propagate attributes that are used when delegating rendering to a templating engine or by children to access some general information during rendering. The following methods are used to propagate these attributes:

Method	Description
`pushAttribute(Object key, Object value)`	Registers the attribute `value` under the key `key`. If some other component have already propagated an attribute under this key, then it will be temporary overridden.
`Object popAttribute(Object key)`	Deregisters the attribute from under the key `key`. If the attribute was temporary overridden it will restore the previous attribute value.
`Object getAttribute(Object key)`	Returns the last object registered under key `key`.
`Map getAttributes()`	Returns all currently registered attributes.

Finally, as `InputData` and `OutputData` are typically connected, they can be retrieved from the other `*Data` structure using correspondingly `getOutputData()` and `getInputData()` methods.

Warning

Since the sibling `*Data` structure is not propagated through the component hierarchy is not guaranteed to be scoped correctly!

2.4.1. Extensions

InputData an OutputData both implement a way to extend their functionality without wrapping or extending the objects themselves. This is achieved by providing the following two methods:

```
void extend(Class interfaceClass, Object extension)
Object narrow(Class interfaceClass);
```

The following example should give an idea of applying these methods:

```
input.extend(FileUploadExtension.class, new FileUploadExtension(input));

...
```

```
FileUploadExtension fileUploadExt =
  (FileUploadExtension) input.narrow(FileUploadExtension.class);
if (fileUploadExt.uploadSucceeded()) {
  //...
}
```

Note

Both `HttpServletRequest` and `HttpServletResponse` are available as `InputData` and `OutputData` extensions respectively.

2.4.2. HttpInputData and HttpOutputData

Although all of the core Aranea abstractions are independent of the Servlet API and web in general, we also provide a way to manipulate low-level HTTP constructs. To that goal we provide to interfaces, `HttpInputData` and `HttpOutputData`, which extend respectively `InputData` and `OutputData`.

Let's examine the `HttpInputData`. First of all it provides methods that are similar to the ones found in the `HttpServletRequest`:

Method	Description
`Iterator getParameterNames()`	Returns an iterator over names of the parameters submitted with the current request.
`String[] getParameterValues(String name)`	Returns the array of values of the particular parameter submitted with the current request.
`String getCharacterEncoding()`	Returns the character encoding that is used to decode the request parameters.
`setCharacterEncoding(String encoding)`	Sets the character encoding that is used to decode the request parameters. Note that this must be called before any parameters are read according to the Servlet specification.
`String getContentType()`	Returns the MIME content type of the request body or `null` if the body is lacking.
`Locale getLocale()`	Returns the preferred Locale that the client will accept content in, based on the Accept-Language header. If the client request doesn't provide an Accept-Language header, this method returns the default locale for the server.

Note

Unlike `InputData` methods the parameters are presented as is and include both global and scoped parameters (the scoped ones are prefixed by the full name of the enclosing widget).

However next methods are a bit different from the `HttpServletRequest` alternatives:

Method	Description
`String getRequestURL()`	Returns the target URL of the current request.
`String getContainerURL()`	Returns an URL pointing to the Aranea container (in most cases the dispatcher servlet).
`String getContextURL()`	Returns an URL pointing to the Aranea container context (in most cases the web application root).
`String getPath()`	Returns the path on the server starting from the dispatcher servlet that has been submitted as the part of the request target URL.
`pushPathPrefix(String pathPrefix)`	Consumes the path prefix allowing children to be mapped to a relative path.
`popPathPrefix()`	Restores the previously consumed path prefix.

The interesting part here are the methods that deal with the path. The problem is that unlike most common cases Aranea components form a hierarchy. Therefore if a parent is mapped to path prefix "myPath/*" and its child is mapped to a path prefix "myChildPath/*" if the path handling were absolute the child would never get the mapped calls. This is due to the child being really mapped to the path "myPath/myChildPath". Therefore the parent must consume the prefix "myPath/" using method `pushPathPrefix()` and then the child will be correctly matched to the relative path "myChildPath".

`HttpOutputData` contains methods that are comparable to the ones found in `HttpServletResponse`:

Method	Description
`String encodeURL(String url)`	Encodes the URL to include some additional information (e.g. HTTP session identifier). Note that Aranea may include some information not present in the servlet spec.
`sendRedirect(String location)`	Sends an HTTP redirect to a specified location URL.
`OutputStream getOutputStream()`	Returns an `OutputStream` that can be used to write to response. Note that unlike the Servlet specification, Aranea permits to use stream and writer interchangeably.
`PrintWriter getWriter()`	Returns a `PrintWriter` that can be used to write to response. Note that unlike the Servlet specification, Aranea permits to use stream and writer interchangeably.
`setContentType(String type)`	Sets the MIME content type of the output. May include the charset, e.g. "text/html; charset=UTF-8".
`Locale getLocale()`	Returns the locale associated with the response.
`String getCharacterEncoding()`	Returns the character encoding used to write out the response.
`void setCharacterEncoding(String encoding)`	Sets the character encoding used to write out the response.

2.5. Services

`org.araneaframework.Service` is a basic abstraction over an event-driven *Controller* pattern that inherits life cycle, environment and messaging from the `Component`. The difference from the `Component` is as follows:

```
public interface Service extends Component, Serializable {
  public Interface _getService();

  public interface Interface extends Serializable {
    public void action(Path path, InputData input, OutputData output) throws Exception;
  }
}
```

The method `action()` is similar to the `service()` method in the Servlet API, `InputData` being an abstraction over a request and `OutputData` being an abstraction over a response (see Section 2.4, “InputData and OutputData”). Thus a service will both process the request parameters and render itself during this method call. However unlike servlets services can be *Composite* and may be defined both statically (on application startup) or dynamically (adding/removing new services on the fly).

Services are the basic working horses of the Aranea framework. They can generally be both synchronized and unsynchronized depending on the context. Services may also have persistent state and their lifetime is explicitly managed by their parent (see Section 2.3.4, “State and Synchronization”). The service life cycle is very simple—as long as the service is live and enabled it can receive `action()` calls, possibly several at a time.

Aranea provides a base implementation of the `Service` — `org.araneaframework.core.BaseService` and a base class for application development `org.araneaframework.core.BaseApplicationService`.

2.5.1. Filter Services

One of the most common ways to use the services it to create a filter service, that wraps a child service and provides some additional functionality and/or environment entries. To that purpose Aranea provides a filter base class — `org.araneaframework.framework.core.BaseFilterService`. This class implements all of the Service methods, by default just delegating them to the corresponding child methods. A common thing to do is override the `action()` method to add functionality and `getChildEnvironment()` to add environment entries, as shown in the following example:

```
public class StandardSynchronizingFilterService
  extends BaseFilterService {

  protected Environment getChildEnvironment() {
    return new StandardEnvironment(
        getEnvironment(),
        SynchronizingContext.class,
        new SynchronizingContext() {});
  }

  protected synchronized void action(
      Path path,
      InputData input,
      OutputData output) throws Exception {
    super.action(path, input, output);
  }
}
```

More information on services and other components that make up the framework can be found in Chapter 3, *Framework and Configuration*.

2.6. Widgets

Widget is the main abstraction used to program applications in Aranea. Widget is specifically any class extending the `org.araneaframework.Widget` interface and adhering to a number of conventions. More generally, widgets are components that function both as controllers and GUI elements, and that have the following properties:

Synchronized
: The widget is always accessed by a single thread, therefore there is never any need to think about synchronization. One can assume that there is only one user using the application and program to service this user without any concern for concurrency.

Stateful
: When programming widgets there is no need to concern oneself with juggling the `HttpSession` attributes or similar low-level mechanics. Widget state (meaning the class fields) is guaranteed to be preserved as long as the widget is alive. One can just use these fields to save the necessary data without any external state management, thus adhering to the rules of object-oriented encapsulation.

The latter two properties make widgets ideal for programming custom application components.

Widgets extend services with a request-response cycle:

```
public interface Widget extends Service, Serializable {
  public Interface _getWidget();

  public interface Interface extends Serializable {
    public void update(InputData data) throws Exception;
    public void event(Path path, InputData input) throws Exception;
    public void process() throws Exception;
    public void render(OutputData output) throws Exception;
  }
}
```

Although widgets extend services, a widget will function during one request either as a widget or as a service—that is if a widget receives an `action()` call no other request-response cycle calls can occur.

The widget request-response cycle proceeds as follows:

1. `update()` —this method is called for all the widgets in the hierarchy. It allows widgets to read the data from request and possibly store some conversation state or at least temporary information to render the next view.
2. `event()` —this method is called on only one widget in the hierarchy. It allows to send widgets events from the user. The `path` is used to route the event to the correct widget and is empty when the event is delivered to its endpoint. This method is optional in the widget request-response cycle.
3. `process()` —this method is called on all the widgets in the hierarchy. It allows widgets to process the result of `event()` and `update()` calls before rendering.
4. `render()` —the way this method is called depends on how widgets are rendered (see Section 2.6.1, “ViewModel and Rendering”). It may be called only after `process()` and may be called more than once (or not at all) during one request-response cycle.

Aranea provides a base implementation of the `Widget`—`org.araneaframework.core.BaseWidget` and a base class for application development `org.araneaframework.core.BaseApplicationWidget`. More on the last one can be found in Section 2.7, “Application Widgets”.

2.6.1. ViewModel and Rendering

The default model of both widget and service rendering is that they render themselves. However, in most cases the widget might want to delegate the rendering to some templating language. In some other cases the widget might be rendered externally, without calling `render()` at all. Further on, we will describe these three cases in detail.

Self-rendering
: In the most basic situation the widget will just use `OutputData` for rendering by casting it into e.g. `HttpOutputData` . In such a case the widget will just write out markup and return from the `render()` method optionally rendering children as well. The data for rendering will be drawn from the widget fields as well as (possibly) `OutputData` attributes.

Using templates for rendering
: The most common case in application widgets is to delegate rendering to a templating language. A widget may basically choose to render itself in arbitrary templating language as Aranea does not impose any restrictions. In fact, one widget may be rendered with one templating language, while another one with a completely different language. The template can gain access to the widget using the knowledge of the widget's full name (which is gathered in the `OutputData` scope). It is then possible to acquire the widget View Model, which is a read-only representation of the widget state. For that the widget should implement `org.araneaframework.Viewable` :

```
public interface Viewable extends Serializable {
  public Interface _getViewable();

  interface Interface extends Serializable {
    public Object getViewModel() throws Exception;
  }
}
```

: View model is put together by the widget being rendered and should contain all the data necessary to render the widget.

External rendering
: Finally, a widget `render()` method may not be called altogether and a `Viewable` widget may be rendered externally using the available View Model. This is the case with some reusable widgets which are rendered using e.g. JSP tags.

2.7. Application Widgets

This section explains how to program applications using widgets as the main abstraction.

A typical application widget class will extend `org.araneaframework.uilib.core.BaseUIWidget`. This widget represents the usual custom application component that is rendered using Aranea custom JSP tags. `BaseUIWidget` inherits most of its functionality from `org.araneaframework.core.BaseApplicationWidget` the difference between the two being only that `BaseUIWidget` assumes to be connected with a JSP page (or another templating toolkit).

2.7.1. Children Management

`BaseApplicationWidget` provides a number of methods for managing child widgets:

```
public abstract class BaseApplicationWidget ... {
  ...
  public void addWidget(Object key, Widget child);
```

```
  public void removeWidget(Object key);
  public void enableWidget(Object key);
  public void disableWidget(Object key);
  ...
}
```

As one can see, every added child has an identifier which should be unique among its siblings. This identifier is used when rendering and sending events to the widget in question, to identify it among its peers.

Typically, children are added when created:

```
addWidget("myChildWidget", new MyChildWidget("String parameter", 1));
```

An added child will be initialized, will receive updates and events and may be rendered. A widget can be active only if added to a parent. It will live as long as the parent, unless the parent explicitly removes it:

```
removeWidget("myChildWidget");
```

Removing a child widget will destroy it and one should also dispose of any references that may be pointing to it, to allow the child to be garbage collected.

A usual idiom is to save a reference to the newly created and added child using a parent widget field:

```
public class MyWidget extends BaseUIWidget {
  private MyChildWidget myChildWidget;

  protected void init() {
    myChildWidget = new MyWidget("String parameter", 1);
    addWidget("myChildWidget", myChildWidget);
  }
}
```

This allows to call directly child widget methods and does not anyhow significantly increase memory usage, so this technique may be used everywhere when needed.

Disabling a child (`disableWidget("myChildWidget")`) will stop it from receiving any events or rendering, but will not destroy it. It can later be reenabled by calling `enableWidget("myChildWidget")`.

2.7.2. Event Listeners

Registering event listeners allows widgets to subscribe to some specific user events (widget will receive only events specially sent to it). The distinction comes by the "event identifier" that is assigned to an event when sending it. The events are handled by the classes extending `org.araneaframework.core.EventListener`:

```
public interface EventListener extends Serializable {
  public void processEvent(Object eventId, InputData input) throws Exception;
}
```

The event listeners are registered as following:

```
addEventListener("myEvent", new EventListener() {
  public void processEvent(Object eventId, InputData input) throws Exception {
```

```
    log.debug("Received event: " + eventId);
  }
}
```

Of course, the event listener does not have to be an anonymous class and can just as well be an inner or even a usual public class. A standard base implementation `org.araneaframework.core.StandardEventListener` is provided that receives an optional `String` event parameter:

```
addEventListener("myEvent", new StandardEventListener() {
  public void processEvent(Object eventId, String eventParam, InputData input) throws Exception;
    log.debug("Received event " + eventId + " with parameter " + parameter);
  }
}
```

Another useful way to process events is to register a proxy event listener (`org.araneaframework.core.ProxyEventListener`) that will proxy the event to a method call, e.g.:

```
protected void init() {
  addEventListener("myEvent", new ProxyEventListener(this));
}

public void handleEventMyEvent(String parameter) {
    log.debug("Received event myEvent with parameter " + parameter);
}
```

The convention is that the proxy event listener translates an event "<event>" into a method call `handleEvent<event>` making the first letter of <event> uppercase. The `"String parameter"` is optional and can be omitted.

A useful feature is the method `setGlobalEventListener(EventListener listener)` that allows to register a listener that will receive all events sent to the widget. In fact `BaseUIWidget` does that by default, and typically you will use the individual event listeners only when you want to override this default behaviour. This allows to just define correct method names (`handleEvent<event>`) and all events will be translated to the calls to these methods. Certainly this can also be cancelled by calling `clearGlobalEventListener()`, or overridden by adding your own global event listener.

2.7.3. Action Listeners

Registering action listeners allows widgets to subscribe to some specific user generated actions. Actions differ from events in that widget lifecycle execution for whole component tree is not triggered upon request—actions are just sent to the receiving widget's *ActionListener*, which is SOLELY responsible for generating the whole response. For rich UI components this allows a quick conversations with server, without requiring full form submits and generating whole view.

2.7.4. Environment

Every initialized widget has a reference to `org.araneaframework.Environment` available through the `getEnvironment()` method. Environment allows to look up framework services (called *contexts*):

```
MessageContext msgCtx = (MessageContext) getEnvironment().getEntry(MessageContext.class);
msgCtx.showInfoMessage("Hello world!");
```

As one can see from the examples, contexts are looked up using their interface `Class` object as key. All framework services in Aranea are accessible only using the environment.

To find out more about `InputData` and `OutputData` see Section 2.3.2, “Environment”

2.7.5. Overridable Methods

The main method that is typically overridden in a widget is `init()`. As widget does not get an environment before it is added and initialized it is impossible to access framework services in the constructor, therefore most of the initialization logic moves to the custom `init()` method. A dual overridable method is `destroy()`, though it is used much less.

In addition to event processing it is sometimes useful to do some kind of pre- and post-processing. The `BaseApplicationWidget` has the following methods that may be overridden to allow this processing:

```
protected void handleUpdate(InputData input) throws Exception {}
protected void handleProcess() throws Exception {}
```

`handleUpdate()` is called before event listeners are notified and allows to read and save request data preparing it for the event. More importantly, this method is called even when no event is sent to the current widget allowing one to submit some data to any widget. `handleProcess()` is called after the event listeners are notified and again is called event if current widgets receives no events at all. It allows to prepare the widget for rendering, post-processing the request results without concern whether or not events have been delivered.

2.7.6. InputData and OuputData

In Aranea one usually does not need to handle request manually in custom application widgets. Even more, the request is not accessible by default. The usual way to submit custom data to a widget and read it is using Aranea Forms (see Chapter 5, *Forms and Data Binding*). However, when one needs to access the submitted data, one can do that using the `org.araneaframework.InputData`. This class can be used as follows:

```
...
String myData1 =
  (String) getInputData().getScopedData().get("myData1");
String globalSubmittedParameter =
  (String) getInputData().getGlobalData().get("globalSubmittedParameter");
...
```

`getInputData()` is a `BaseApplicationWidget` method that returns the input data for the current request (one can also use the `input` parameter given to event listener directly).

`org.araneaframework.OutputData` is accessible through the `getOutputData()` method of `BaseWidget` or directly as the `output` parameter passed to `render()` method.

To find out more about `InputData` and `OutputData` see Section 2.4, “InputData and OutputData”

2.7.7. View Model and Rendering

`BaseApplicationWidget` also contains methods that facilitate transferring data to the presentation layer. This is achieved using a *View model*—an object containing a snapshot of the widget current state. The most typical way to use the view model it to add data to it:

```
...
putViewData("today", new Date());
putViewData("currentUser", userBean);
...
```

View data is typically accessible in the presentation layer as some kind of a variable (e.g. a JSP EL variable) for the current widget. If the data becomes outdated one can override it using `putViewData()` call or remove it using the `removeViewData()` call. In case one needs to put view data that would last one request only there is an alternative method:

```
...
putViewDataOnce("now", new Date());
...
```

Finally widget instance is also visible to the view, so one of the ways to make some data accessible is just to define a JavaBean style getter:

```
...
public Date getNow() {
  return new Date();
}
...
```

`BaseUIWidget` allows to render the current widget using a JSP page. To do that one needs to select a view as follows:

```
...
setViewSelector("myWidget/form");
...
```

This code makes the widget render itself using the JSP situated in `WEB-INF/jsp/myWidget/form.jsp` (of course the exact place is configurable). It is also possible to render the widget using other template technologies with the same view selector by overriding the `render()` method in the base project widget.

2.7.8. Putting It All Together

A typical application custom widget will look like that:

```
public class TestWidget extends BaseUIWidget {

  private static final Logger log = Logger.getLogger(TestWidget.class);

  private Data data;

  protected void init() throws Exception {
    //Sets the JSP for this widget to "/WEB-INF/jsp/home.jsp"
    setViewSelector("home");

    //Get data from the business layer
    data = ((TestService) lookupService("testService")).getData("test parameter");

    //Make the data accessible to the JSP for rendering
    putViewData("myData", data);
  }

  /*
```

```
    * Event listener method that will process "test" event.
    */
  public void handleEventTest() throws Exception {
    getMessageCtx().showInfoMessage("Test event received successfully");
  }
}
```

2.8. Standard Contexts

Contexts are the Aranea way to access framework services. They can be looked up from the environment as shown in Section 2.7.4, “Environment”. This section describes the most common Aranea contexts that should be available in any typical configuration. All these contexts are also available directly through `BaseUIWidget` methods as shown further on.

2.8.1. MessageContext

`org.araneaframework.framework.MessageContext` allows to show messages to the user. The messages can be of several types, including predefined error and informative types. Typically messages will be shown somewhere in the application (exact way is application-specific). `MessageContext` is available through a `BaseUIWidget` method `getMessageCtx()` and is typically used as follows:

```
getMessageCtx().showInfoMessage("Hello world!");
```

`MessageContext` divides messages by type (with predefined "info", "warning" and "error" types available) and life span (usual or permanent). Usual messages are shown to user once and then cleared, while permanent messages will be shown to user until explicitly cleared by the programmer:

Method	Description
`showMessage(String type, String message)`	Shows a message `message` of type `type` to the user. Message is cleared after the user sees it once.
`showInfoMessage(String message)`	Shows an error message to the user.
`showWarningMessage(String message)`	Shows a warning message to the user.
`showErrorMessage(String message)`	Shows an informative message to the user.
`clearMessages()`	Clears all non-permanent messages.
`showPermanentMessage(String type, String message)`	Shows a permanent message `message` of type `type` to the user. The message will be shown until hidden explicitly.
`hidePermanentMessage(String message)`	Clears the specific permanent message, under all message types where it might be present.
`clearPermanentMessages()`	Clears all of the permanent messages.
`clearAllMessages()`	Clears all messages (both permanent and usual).

Note

Messages should already be localized when passed to the `MessageContext`, it does not do any further processing. Use `LocalizationContext` described in Section 2.8.2, “LocalizationContext” to do the actual localization of the added message.

For information on implementation of the `MessageContext` see Section 3.5.8, “User Messages Filter”.

2.8.2. LocalizationContext

`org.araneaframework.framework.LocalizationContext` allows to get and set current session locale, localize strings and messages, and lookup resource bundles. The context is available through the `BaseUIWidget` method `getL10nCtx()`. Typically it is used as follows:

```
...
String message = getL10nCtx().localize("my.message.key");
getMessageCtx().showInfoMessage(message);
...
```

`LocalizationContext` provides the following methods:

Method	Description
`Locale getLocale()`	Returns the current session locale.
`setLocale(Locale locale)`	Sets the current session locale.
`String localize(String key)`	Localizes a string returning one that corresponds to the current locale.
`ResourceBundle getResourceBundle()`	Returns a resource bundle corresponding to the current locale.
`ResourceBundle getResourceBundle(Locale locale)`	Returns a resource bundle corresponding to arbitrary locale.
`String getMessage(String code, Object[] args)`	Localizes the code and uses it to format the message with the passed arguments. The format of the localized message should be acceptable by `java.text.MessageFormat`.
`String getMessage(String code, Object[] args, String defaultMessage)`	Localizes the code and uses it to format the message with the passed arguments. The format of the localized message should be acceptable by `java.text.MessageFormat`. If the localized message cannot be resolved uses `defaultMessage` instead.

For information on implementation of the `LocalizationContext` see Section 8.1.2, “Spring Localization Filter”.

2.8.3. FlowContext

A common need in a web programming is to support navigation style known as *flows*—interactive stateful processes that can navigate to each other passing arguments when needed. A more complex case is when we

also have flow nesting—a flow can call a subflow, and wait for it to finish, then reactivate again. In this case we can have at any given moment a stack of flows, where the top one is active, and the next one will reactivate when the top one finishes. It is also useful if nested flows can return resulting values when they finish.

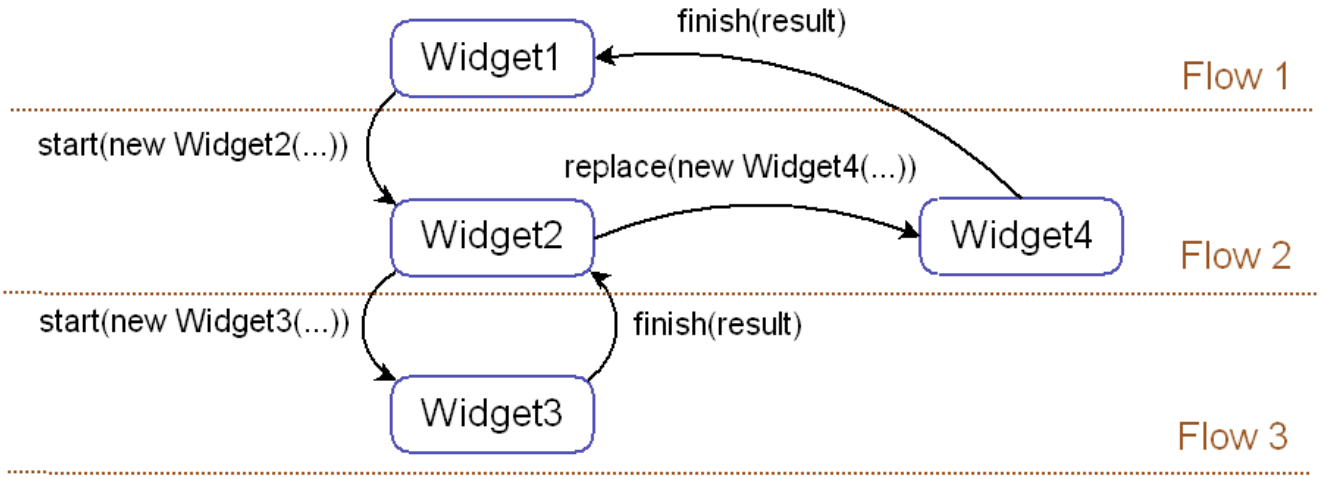


Figure 2.2. Flow diagram

`org.araneaframework.framework.FlowContext` is the Aranea context that provides support for nested flow navigation. Aranea flow is a widget that is running in the flow container (using the `FlowContext.start()` method. Aranea abstraction for the nested state is that of a function—the nested *flow* takes in some parameters and when finished may return some value or signal that no value can be returned. The context is available as `getFlowCtx()` method of `BaseUIWidget` and allows to start flows, finish flows and return the resulting value.

To start a new flow one needs to create a widget as usual. The widget may take some parameters in the constructor—they are considered to be the incoming parameters of the flow:

```
...
getFlowCtx().start(new TestFlow(new Long(5)), null, null);
...
```

This call will start a new nested flow for the widget `TestFlow` making the current flow inactive. `TestFlow` will render and receive event until it explicitly returns control to the starting flow. Note that this code will start the flow and then return the control, so it is important not to do anything in the same method after starting a new flow.

To end the flow successfully one needs to do as follows:

```
...
getFlowCtx().finish(new Long(8));
...
```

This call will finish the current flow (in our case `TestFlow`) and return the control to the starting flow and its widget.

Often one needs to handle the return from the flow, processing the returned result. This corresponds to our abstraction of a method, however since Java does not support continuations we chose to allow the caller to register a handler when starting the flow by passing a `FlowContext.Handler`:

```
...
getFlowCtx().start(new TestFlow(new Long(5)), null,
  new FlowContext.Handler() {
    public void onFinish(Object result) {
      getMessageCtx().showInfoMessage("TestFlow returned value " + result);
    }
    public void onCancel() {
      //Ignore cancelled flow
    }
  });
...
```

A less common but nevertheless useful feature is to configure the starting flow after it has been initialized. For that the caller needs to pass a `FlowContext.Configurator`:

```
...
getFlowCtx().start(new TestFlow(new Long(5)),
  new FlowContext.Configurator() {
     public void configure(Component comp) {
      ((TestFlow) comp).setStrategy(TestFlow.ATTACK);
     }
  }, null);
...
```

Finally `FlowContext` also allows to replace the current flow instead of deactivating it by using the `replace()` method and to cancel the current flow by using the `cancel()` method.

For standard implementation, please see Section 3.5.19, “Root Flow Container”

Chapter 3. Framework and Configuration

3.1. Overview

Aranea framework consists of a number of independent components each performing a single well-defined function. Aranea uses Spring to wire these components into a working framework. Though other IoC containers and configuration frameworks would also work we support Spring by default since it provides a very comfortable and versatile syntax for configuring beans. The dispatcher servlet that uses Spring is called `org.araneaframework.integration.spring.AraneaSpringDispatcherServlet`. Note that Aranea itself does not depend on Spring except the classes in the `org.araneaframework.integration.spring` package.

3.2. Application Configuration

3.2.1. web.xml

The simplest way to configure Aranea for a web application is to set the `araneaApplicationStart` init parameter of the dispatcher servlet to the starting widget or flow of the application:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <listener>
    <listener-class>
      org.araneaframework.http.core.StandardSessionListener
    </listener-class>
  </listener>

   <servlet>
      <servlet-name>araneaServlet</servlet-name>
      <servlet-class>
        org.araneaframework.integration.spring.AraneaSpringDispatcherServlet
      </servlet-class>
      <init-param>
        <param-name>araneaApplicationStart</param-name>
        <param-value>example.StartWidget</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>araneaServlet</servlet-name>
      <url-pattern>/main/*</url-pattern>
   </servlet-mapping>
</web-app>
```

This configuration will load Aranea using `example.StartWidget` as the application starting point.

Note

The servlet must be mapped to a all subpathes starting from some prefix (in our case `/main/*`) so that Aranea could do some path-dependent operations like extension file importing.

Note

`org.araneaframework.http.core.StandardSessionListener` is required to allow Aranea to process events like session invalidation.

3.2.2. aranea-conf.xml

Aranea can also be configured using a Spring configuration file located in `/WEB-INF/aranea-conf.xml`. Particularly it may be used to set the starting widget instead of the init-parameter:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="araneaApplicationStart"
      class="example.StartWidget"
      singleton="false"/>
</beans>
```

This seems to be more verbose, but it also allows to configure the framework components as described in Section 3.4, “Framework Configuration”.

3.2.3. aranea-conf.properties

Aranea also takes into account a property file located in `/WEB-INF/aranea-conf.properties`. The following properties are recognized:

Property	Description
`l10n.resourceBundle`	The base name of the resource bundle used for localization. This value isn't used if default `araneaLocalizationFilter` is overidden (e.g. by the `SpringLocalizationFilterService`) **Default value:** `org.araneaframework.http.support.DefaultResourceBundle`
`l10n.defaultLocale`	The default locale to be used in the application. **Default value:** `en`
`l10n.encoding`	The default character encoding to be used throughout the application (e.g. for request and response). **Default value:** `UTF-8`
`jsp.path`	The path from the webapp root to the directory that will act as JSP root. The JSPs put there can be selected using widget view selectors (see Section 2.7.7, “View Model and Rendering”). **Default value:** `/WEB-INF/jsp`

3.2.4. AraneaSpringDispatcherServlet

`AraneaSpringDispatcherServlet` provides a number of init-params that allow to further customize Aranea configuration:

init-param	Description
`araneaCustomConfXML`	The custom location of a Spring XML file used to configure Aranea. **Default value:** `/WEB-INF/aranea-conf.xml`
`araneaCustomConfProperties`	The custom location of a property file used to configure Aranea. **Default value:** `/WEB-INF/aranea-conf.properties`
`araneaApplicationStart`	The class name of an Aranea widget that will serve as the starting point of an Aranea application. If omitted the Spring bean `araneaApplicationStart` will be used.
`araneaApplicationRoot`	The class name of an Spring bean describing an Aranea component that will serve as the framework root. If omitted the Spring bean `araneaApplicationRoot` will be used. Can be used to override the default configuration altogether.

3.2.5. Extending Dispatcher

Currently, the most common way to put Aranea to work is to host it in a Servlet 2.3 or compatible container. The most generic way to do that is to extend the `org.araneaframework.http.core.BaseAraneaDispatcherServlet` and build the root component of type `org.araneaframework.http.ServletServiceAdapterComponent` in the overrided method `buildRootComponent()`:

```
package com.foobar.myapp;

class MyServlet extends BaseAraneaDispatcherServlet {
   protected ServletServiceAdapterComponent buildRootComponent() {
     StandardServletServiceAdapterComponent root = new StandardServletServiceAdapterComponent();

     //Configure the child components, service widgets using setter injection
     //...

     return root;
   }
}
```

And one can then use such a servlet to configure Aranea in a web application as usually replacing the standard dispatcher servlet with our custom one.

3.3. Framework Assembly

Aranea framework is made up of the same Components, Services and Widgets that are also used to develop Aranea applications. Each component performs a single well-defined function and may depend on other components only via Environment. The framework components mostly fall in one of the three following categories:

Filter
: Filter components are the simplest. The component contains a single unnamed child and implements the *Filter* pattern by either passing each call to the child or not. However in addition it may enrich the child's environment with contexts and provide more functionality like exception handling or synchronization.

Typical examples of filters are localization filter (provides a localization context), synchronization filter (synchronizes on `action()` method) and transactional filter that does not let through double submits.

Router
: Router typically contains many named children, and chooses only one to propagate the calls to according to some `InputData` parameter. Router may have the children either statically preconfigured or created dynamically when the request comes (the latter is the case with session service router). It may also allow us to add/remove children while the application is running. A typical application of a router is to distinguish among major application parts by some attribute (like component corresponding to a user session, or one of the popup window of current user).

Container
: Container can have one or many children, but it typically will do more with them than just passing the calls to one of them. A typical example is the *widget container* service which translates `action()` calls into widget `update()/event()/process()/render()` cycle.

The frameworks itself is assembled using a hiearchy of components (this hierarchy is mostly flat, except for the routers and application components). The hierarchy is arrange simply by containment, with each component containing its chidren as fields as illustrated on Figure 3.1, “Framework assembly example”.

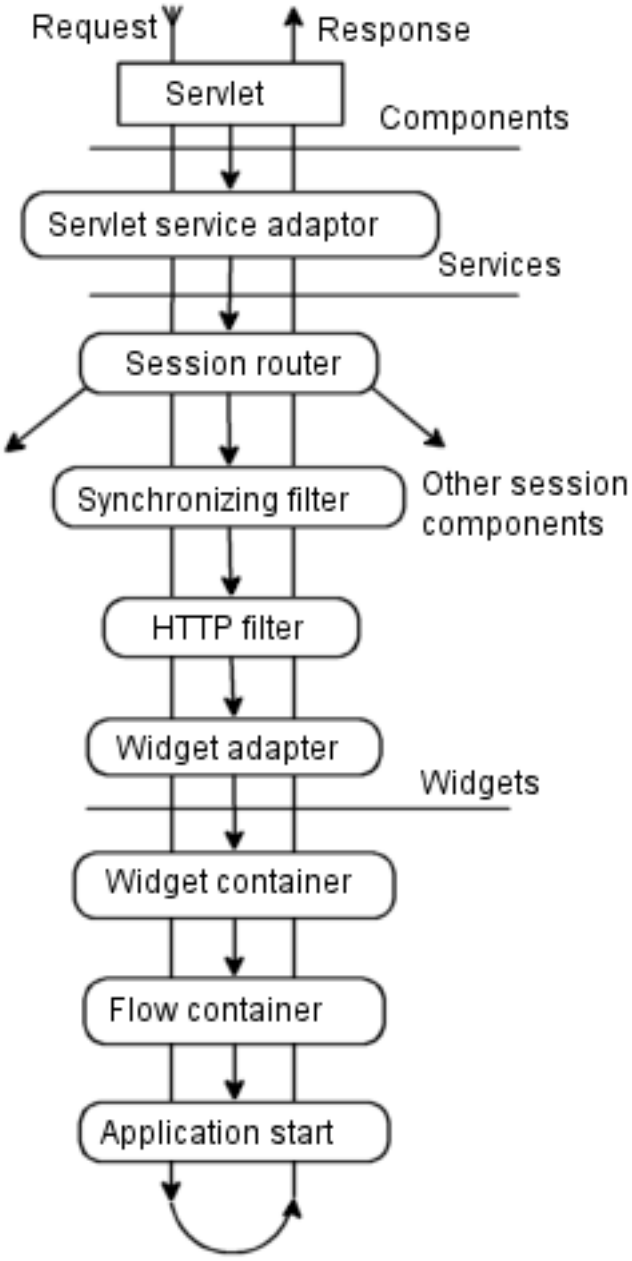


Figure 3.1. Framework assembly example

Of course this illustration is simplified, omitting most of the components described in Section 3.5, “Framework Components”. If you want to find out more about the way framework is built and assembled, see the Aranea Technical Paper [http://www.araneaframework.org/docs/aranea-technical-paper.pdf].

3.4. Framework Configuration

Aranea framework is assembled into a mostly-flat hierarchy using Spring beans. The default Aranea configuration is loaded by the `AraneaSpringDispatcherServlet`, but it can be overriden with the custom configuration in `aranea-conf.xml`. The dispatcher servlet loads the configuration in such a way that same named beans in `aranea-conf.xml` override the ones specified in the default configuration. However, not all beans can be safely or comfortably overriden, since many of them will also refer to their child beans.

It is always safe to override filters, as they should never refer directly to their children. To override a filter just make a bean definition with the same name as in default configuration (filters and their default configuration names among other components are described in Section 3.5, “Framework Components”). For instance to override the default localization context with a custom-made one, one would need to add the following lines:

```
<bean class="example.LocalizationFilterService"
  id="araneaLocalizationFilter" singleton="false">
  <property name="languageName" value="ee"/>
</bean>
```

There is no good way in Spring to undefine a bean, so instead we use a "No OPeration" filter to nullify a filter from the default configuration:

```
<bean class="org.araneaframework.framework.core.NopFilterWidget"
  id="araneaTransactionFilter" singleton="false"/>
```

Warning

Since filters can be both services and widgets, you have to be careful to use the appropriate one for the current context. In current case you have override service filters with `NopFilterService` and widget filters with `NopFilterWidget`.

There is no generic way to insert filters into an arbitrary place in the framework component hierarchy. However there are several predefined places left for optional bean insertion at various levels of the hierarchy, which should cover most of customization needs. To allow inserting more than one filter at a time a filter chain bean is provided that allows putting together an arbitrary long chain of filters:

```
<bean id="araneaCustomSessionFilters" singleton="false"
  class="org.araneaframework.framework.filter.StandardFilterChainService">
  <property name="filterChain">
    <list>
      <ref bean="araneaSerializingAudit"/>
      <ref bean="myCustomFilter1"/>
      <ref bean="myCustomFilter2"/>
    </list>
  </property>
</bean>
```

Note

Use `StandardFilterChainService` for hosting service filters and `StandardFilterChainWidget` for hosting widget filters.

Follows a description of the insertion point beans and their scope:

Bean name	Scope and Description
`araneaCustomApplicationFilters`	These filters are created only once and live as long as the application does. They are not synchronized and should be use to add features generic to the whole application, not specific users. The exceptions thrown by this filters are intepreted as critical and are handled by the critical exception handler. **Examples:** araneaFileUploadFilter, araneaStatisticFilter.

Bean name	Scope and Description
`araneaCustomSessionFilters`	These filters are created for every HTTP user session and live as long as the session does. They are generally synchronized and should be used to add features specific to the current user session. **Examples:** araneaLocalizationFilter.
`araneaCustomThreadFilters`	These filters are created for every user browser window and live as long as the window does. They are synchronized and should be used to add features specific to the individual browser window (e.g. most rendering filters will fall into this category). **Examples:** araneaThreadCloningFilter .
`araneaCustomWidgetFilters`	These filters are created for every user browser window and live as long as the window does. They are synchronized and should be used to add features specific to the individual browser window. Unlike the rest of the filters this can be widgets and thus can take advantage of the widget update/event/process/render cycle. **Examples:** araneaTransactionFilter, araneaMessagingFilter.

3.5. Framework Components

Aranea configuration is determined by request-processing components that can be assembled in many different ways. Following sections are a brief reference for pre-existing standard components, most of which are also used in Aranea framework default configuration.

3.5.1. Localization Filter

Java class:	`StandardLocalizationFilterService`
Default configuration name:	`araneaLocalizationFilter`
Provides:	`LocalizationContext`
Depends on:	-

Provides localization services to children. See Section 2.8.2, “LocalizationContext”.

Injectable properties	Description
languageName `java.lang.String`	A valid ISO Language Code. Sets `Locale` according to given language.
resourceBundleName	Name of the used resource bundle used to localize the application.

Injectable properties	Description
`java.lang.String`	
locale `java.util.Locale`	`Locale` to use. Either that or *languageName* should be specified, but not both.

3.5.2. AJAX Update Regions Filter

Java class:	`StandardUpdateRegionFilterService`
Default configuration name:	`araneaUpdateRegionFilter`
Provides:	-
Depends on:	-

When serving AJAX requests made with AjaxAnywhere (see Section 9.1.2, “AjaxAnywhere (http://ajaxanywhere.sourceforge.net/)”), extracts from the initial response these page regions that need to be updated (request includes the names of these regions), then modifies the response to include just these regions (not the whole page).

Injectable properties	Description
characterEncoding `java.lang.String`	The character encoding for responses served by this filter, default being "UTF-8".
existingRegions `java.util.List<String>`	When the filter is activated, it tries to extract specified regions from response, even if they were not explicitly asked to be updated in request. This is useful for always updating some common page regions (feedback message regions etc). Some default region names that should be updated when AJAX request is made are `messages`, `popupRegistrationRegion`.

3.5.3. Environment Configuration Filter

Java class:	`StandardContextMapFilterWidget`
Default configuration name:	`araneaEnvContextFilter`
Provides:	-
Depends on:	-

Filter widget that enriches children environment with specified context entries.

Injectable properties	Description
contexts `java.util.Map`	A map of contexts that will be added to environment. The keys can contains strings of kind "package.ClassName.class", which will use a Class object of the specified classname as the context key. The context value should be an object instance of the context interface. By convention a context should be registered under a key that is an interface it implements.

3.5.4. Critical Exception Handler

Java class:	`StandardCriticalExceptionHandlingFilterService`
Default configuration name:	`araneaCriticalErrorHandler`
Provides:	-
Depends on:	-

Catches the exceptions (if any) occuring while executing children methods; passes the exceptions on to `Service` that deals with exception handling (obtained from `ExceptionHandlerFactory`).

Injectable properties	Description
exceptionHandlerFactory `ExceptionHandlerFactory`	A factory for creating exception handlers. An exception handler is a service, which handles the user notification and recovery.

3.5.5. File Uploading Filter

Java class:	`StandardCriticalExceptionHandlingFilterService`
Default configuration name:	`araneaFileUploadFilter`
Provides:	`FileUploadContext`, `FileUploadInputExtension`
Depends on:	-

Enriches child environment with `FileUploadContext` (which is just a marker interface). When incoming request is multi-part request, children's `InputData` is extended with `FileUploadInputExtension` that allows children easy access to uploaded files.

Injectable properties	Description
multipartEncoding `java.lang.String`	Character encoding that will be used to decode the `multipart/form-data` encoded strings. The default encoding is determined by *Apache Commons* `FileUpload` class.
useRequestEncoding	When set to "true" request character encoding will be used to parse the

Injectable properties	Description
`boolean`	`multipart/form-data` encoded strings.
maximumCachedSize `java.lang.Integer`	Maximum size of file that may be cached in memory.
maximumSize `java.lang.Long`	Maximum size of file that may be uploaded to server.
maximumRequestSize `java.lang.Long`	Maximum size of the request that server will parse to the end.
tempDirectory `java.lang.String`	Temporary directory to use when uploading files.

3.5.6. HTTP Response Headers Filter

Java class:	`StandardHttpResponseFilterService`
Default configuration name:	`araneaResponseHeaderFilter`
Provides:	-
Depends on:	-

Filter that sets necessay headers of the response.

Injectable properties	Description
cacheable `boolean`	Whether the response is cacheable or not. By default it is not cacheable.
contentType `java.lang.String`	Sets the content type of the response. Default is "text/html; charset=UTF-8".
cookies `java.util.Map`	Constructs cookies from the <cookieName, cookieValue> pairs in the map and sets them in response.
headers `java.util.Map`	Sets the headers of the response from the map of <headerName, headerValue>.
cacheHoldingTime `long`	Sets the cache-control's max-age parameter, value is in milliseconds. Response must be cacheable for this to have any effect.

3.5.7. JSP Configuration Filter

Java class:	`StandardJspFilterService`
Default configuration name:	`araneaJspConfigFilter`
Provides:	`JspContext`
Depends on:	`LocalizationContext`

Provides JSP specific information to children.

Injectable properties	Description
submitCharset `java.lang.String`	Sets the "accept-charset" attribute value that will be used for rendering Aranea JSP specific systemForm.
jspPath `java.lang.String`	Path where widgets rendering themselves with jsp templates should search for them. Default is "/WEB-INF/jsp".
jspExtension `java.lang.String`	File name extension jsp templates are assumed to have. Default is ".jsp".

3.5.8. User Messages Filter

Java class:	`StandardMessagingFilterWidget`
Default configuration name:	`araneaMessagingFilter`
Provides:	`MessageContext`
Depends on:	-

See Section 2.8.1, “MessageContext”.

3.5.9. Popup Windows Filter

Java class:	`StandardPopupFilterWidget`
Default configuration name:	`araneaPopupFilter`
Provides:	`PopupWindowContext`
Depends on:	`ThreadContext`, `TopServiceContext`, `TransactionContext`

Provides methods for opening new session-threads and renders these in different browser windows at

client-side.

Injectable properties	Description
threadServiceFactory `ServiceFactory`	Factory that should build the component chain according to effective Aranea configuration, beginning with sessionthread-level filters.

3.5.10. Serialization Control Filter

Java class:	`StandardSerializingAuditFilterService`
Default configuration name:	`araneaSerializingAudit` (not included in default filter chain)
Provides:	-
Depends on:	-

Serializes the the session during the request routing. This filter helps to be aware of serializing issues during development as when the session does not serialize, exception is always thrown. In production configuration, this filter should never be enabled, thus it is disabled by default.

Injectable properties	Description
testXmlSessionPath `java.lang.String`	The path where the serialized sessions should be logged in XML format. If not specified, serialization tests are performed in-memory.

3.5.11. Statistics Logging Filter

Java class:	`StandardStatisticFilterService`
Default configuration name:	`araneaStatisticFilter`
Provides:	-
Depends on:	-

Filter that logs the time it takes for the child service to serve the request (complete its action method).

Injectable properties	Description
message `java.lang.String`	The prefix of the statistics log statement.

3.5.12. Synchronization Filter

Java class:	`StandardSynchronizingFilterService`
Default configuration name:	`araneaSynchronizingFilter`
Provides:	`SynchronizingContext`
Depends on:	-

Synchronizes the calls to its child widget. Enriches environment with `SynchronizingContext` (which is just marker interface).

3.5.13. Browser Window Cloning Filter

Java class:	`StandardThreadCloningFilterService`
Default configuration name:	`araneaThreadCloningFilter`
Provides:	`ThreadCloningContext`
Depends on:	`ThreadContext`, `TopServiceContext`

Implementation of a service that clones currently running session thread upon request and sends a response that redirects to cloned session thread. It can be used to support "open link in new window" feature in browsers. Cloning is generic and resource demanding, as whole tree of session thread components is recreated. Custom applications may find that they can implement some application specific cloning strategy that demands less memory and processing power.

Injectable properties	Description
timeToLive `java.lang.Long`	Inactivity time for cloned thread after which thread router may kill the thread service. This is specified in milliseconds. If unset, threads created by cloning service usually live until HTTP session in which they were spawned expires.

3.5.14. Multi-submit Protection Filter

Java class:	`StandardTransactionFilterWidget`
Default configuration name:	`araneaTransactionFilter`
Provides:	`TransactionContext`
Depends on:	-

TransactionContext implementation that filters routing of duplicate requests. The detection of duplicate requests is achieved through defining new transaction ID in each response and checking that next request submits the consistent transaction ID. Missing (*null*) transaction ID is always considered inconsistent. For purposes of asynchronous requests, `override` transaction ID is always considered consistent.

Request parameter name	Description
`transactionId`	Transaction id must be equal to the last one generated for the transaction to be consistent.

3.5.15. Class Reloading Filter

Java class:	`StandardClassReloadingFilterWidget`
Default configuration name:	-
Provides:	-
Depends on:	-

This filter allows to reload the underlying object classes dynamically. This means that you can just change the widget source file, compile it (e.g. with IDE built-in compiler) and it will be reloaded seamlessly in Aranea. This will apply only to Aranea widget classes under this filter and the classes they contain (but not e.g. Spring beans). This filter must be registered instead of the `araneaApplicationStart` to function.

Warning

None of the classes under this filter may be configured by Spring or anything else using its own classloader!

Injectable properties	Description
childClass `java.lang.String`	The full names of the child widget class.

3.5.16. Client State Serialization Filter

Java class:	`StandardClientStateFilterWidget`
Default configuration name:	`araneaClientStateFilter` (not included in default filter chain)
Provides:	-
Depends on:	-

This filter will serialize the state of underlying widgets onto client-side. This significantly decreases the server-side session size and thus memory use. It is especially useful in intranet applications with lots of spare bandwidth. The filter should be positioned as the first custom widget filter for most gain.

Note

The filter will protect against tampering with the serialized state and will throw an exception if modified state is submitted from the client-side. As a bonus this filter will also allow a user to make up to 10 steps back and forward in browser history, restoring the correct state.

Injectable properties	Description
compress `boolean`	If true the serialized state will also be GZIP'ed, trading processor time for bandwidth. False by default.

3.5.17. Extension File Import Filter

Java class:	`StandardFileImportFilterService`
Default configuration name:	`araneaFileImportFilter`
Provides:	-
Depends on:	-

This filter is responsible for providing a virtual file system so that extensions could make use of the resources included in .JAR files. See Section 3.6.1, “Extension Resources”

3.5.18. Bookmarking/URL Mounting Filter

Java class:	`StandardMountingFilterService`
Default configuration name:	`araneaMountingFilter`
Provides:	`MountContext`
Depends on:	-

Implementation of a service that allows to "mount" flow components to a publicly accessible URL. It is used when it is needed that some (read-only) parts of application are accessible to users who are not able to enter the session-based conversation with application.

Injectable properties	Description
mounts `java.util.Map<String, MountContext.MessageFactory>`	Keys in the map are URL prefixes under which the flow component is mapped. Values are `org.araneaframework.Message` factories of type `MountContext.MessageFactory`—producing messages that generate component hierarchy for serving wanted content.

3.5.19. Root Flow Container

Java class:	`RootFlowContainerWidget`

Default configuration name:	`araneaRootFlowContainer`
Provides:	`RootFlowContext`, `FlowContext`
Depends on:	-

See Section 2.8.3, “FlowContext” for purpose and philosophy behind `FlowContext`. `RootFlowContext` is same as `FlowContext`, but allows to acces the root flow container at any time.

Tip

Flow containers are not generally a part of the framework and can be used in your application as needed. In a typical Aranea application the menu will inherit from `ExceptionHandlingFlowContainerWidget` that besides providing the flow container functionality also allows to handle flow exceptions inside the container, preserving the menus and current state. See business application tutorial for more information.

Injectable properties	Description
top `org.araneaframework.Widget`	First widget to be started in this container.

3.6. Other

3.6.1. Extension Resources

External resources, such as javascript, style and image files of Aranea components are managed through different configuration files. The resources are listed in XML files and can be accessed through `StandardFileImportFilterService`. This approach makes it possible to package all the resources into the aranea `jar` archives and no manual copying of necessary files to fixed locations is needed.

Aranea comes bundled with a `aranea-resources.xml` file which defines all the external resources.

```
<?xml version="1.0" encoding="UTF-8"?>
<resources>
  <files content-type="text/css" group="defaultStyles">
    <file path="styles/_styles_global.css"/>
    ...
    <file path="styles/_styles_screen.css"/>
  </files>

  <files content-type="image/gif">
    <file path="gfx/i01.gif"/>
    ...
    <file path="gfx/i02.gif"/>
  </files>
  ...
</resources>
```

All the files listed in the configuration files are allowed to be loaded through the `FileImportFilter`. Some are grouped by name to provide an easy access for reading files in bulk.

To *override* specific files in the configuration file, the new file should be placed in a subdirectory `override`. When loading a file, Aranea first trys to open the file in the `override` directory and on failure trys to read the file without the prefix directory.

To *add* files to the defined list, construct a configuration file and name it `aranea-resources.xml`. All such configuration files from the classpath are parsed for the resources. If two file groups are defined with the same name, the group is formed by taking a union from the files in the group.

Groupnames `defaultStyles` and `defaultScripts` are predefined groups for managing the necessary core files that must be included for Aranea to work correctly.

For custom loading a resource, the URL to use is `/fileimporter/filepath`. The `fileimporter` is `StandardFileImportFilterService.FILE_IMPORTER_NAME` and `filepath` is the path that is defined for the file in the resource configuration file.

Extensions of the framework provide their own configuration files for configuring their resources. New extensions cannot be defined right now on the fly.

Chapter 4. JSP and Custom Tags

4.1. Aranea Standard Tag Library

Aranea supports JSP rendering by providing a JSP 1.2 custom tag library that tries to abstract away from HTML and allow programming in terms of widgets, layouts and logical GUI elements. The tag library URI is "http://araneaframework.org/tag-library/standard" and it is contained in `aranea-presentation.jar`, so putting this JAR in the classpath (e.g. `WEB-INF/lib`) is enough to put it to work. Library tags support JSP Expression Language that is used in JSTL 1.0.

Aranea examples use JSP XML form and in such form importing the library should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:ui="http://araneaframework.org/tag-library/standard" version="1.2">
  ...
</jsp:root>
```

In a usual JSP file it should look like this:

```
<%@ taglib uri="http://araneaframework.org/tag-library/standard" prefix="ui" %>
...
```

The suggested prefix for the tag library is "ui".

There is otherwise identical taglib that has `<rtexprvalue>` set to `true` for each tag attribute. URI for that taglib is `http://araneaframework.org/tag-library/standard_rt`. When using JSP version 2.0 or higher, this taglib should be used, otherwise EL in attributes is rejected by containers.

4.2. System Tags

Aranea JSP rendering should start from some root JSP (called *JSP template*) that will include the root widgets. To support widgets and other custom tags one needs to make sure that the template looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:ui="http://araneaframework.org/tag-library/standard" version="1.2">
  <ui:root>
    <ui:viewPort>
      <html>
        <head>
          <title>Aranea Template Application</title>

          <ui:importScripts/>
          <ui:importStyles/>

        </head>

          <ui:body>
            <ui:systemForm method="POST">
              <h1>Aranea Template Application</h1>

              <ui:messages/>

              <ui:widgetInclude id="root"/>
            </ui:systemForm>
```

```
        </ui:body>
      </html>
    </ui:viewPort>
  </ui:root>
</jsp:root>
```

Next are described all these tags except <ui:widgetInclude>, which is described in the following section.

4.2.1. <ui:root>

This tag should always be the root of any Aranea template JSP. Its main function is to allow JSP access the controller.

Variables

Variable	Description	Type
outputData	The `org.araneaframework.OutputData` that represents the current response.	`org.araneaframework.OutputData`

4.2.2. <ui:viewPort>

This tag should be immediately under the <ui:root> tag for any application that uses widgets. It allows widgets to be rendered and included.

Variables

Variable	Description	Type
outputData	The `org.araneaframework.OutputData` that represents the current response.	`org.araneaframework.OutputData`

4.2.3. <ui:importScripts>

Aranea comes bundled with different external resources: javascript libraries, stylesheets and images. To automate the process of loading the javascript files without the manual copying of them to specific webapp locations, a special filter is used. The filter is able to read files from aranea `jar` files.

<ui:importScripts> depends on the filter *StandardServletFileImportFilterService* being set. The filter provides the functionality of reading files from the jars on the server.

If no attributes specified, the default group of javascript files are loaded.

Attributes

Attribute	Required	Description
file	*no*	Writes HTML `<script>` tag to load the specific file.
group	*no*	Writes HTML `<script>` tag to load a group of javascript files.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:importScripts/> <!-- imports files from 'defaultScripts' group -->
<ui:importScripts group="debugScripts"/> <!-- imports additional debug scripts (js logger) -->
```

4.2.4. <ui:importStyles>

Aranea comes bundled with CSS files to provide custom look for different predefined components (the template app, calendar, htmleditor, etc.). Just as with javascript, to use them one would have to extract them from the jars and use them just like any other css file would be used. To automate this process with aranea css files one can use the <ui:importStyles> tag to include the css files *automatically*.

<ui:importStyles> depends on the filter *StandardServletFileImportFilterService* being set. The filter provides the functionality of reading files from the jars on the server.

If no attributes specified, the default group of css files are loaded.

Attributes

Attribute	Required	Description
file	*no*	Writes out the HTML's CSS handling `link` to load the specific file.
group	*no*	Writes out the HTML's CSS handling `link` to load the group of files.
media	*no*	Media type to which imported styles are applied.

4.2.5. <ui:body>

This tag will render an HTML <body> tag with Aranea JSP specific *onload* and *onunload* events attached. It usually writes out some other page initialization scripts too, depending on the circumstances.

Attributes

Attribute	Required	Description
onload	*no*	Overwrite the standard Aranea JSP HTML body onload event. Use with caution.
onunload	*no*	Overwrite the standard Aranea JSP HTML body onload event. Use with caution.
id	*no*	HTML BODY id.
dir	*no*	HTML BODY dir attribute.
lang	*no*	HTML BODY lang attribute.
title	*no*	HTML BODY title attribute.

4.2.6. <ui:systemForm>

This tag will render an HTML <form> tag along with some Aranea-specific hidden fields. When making custom web applications it is strongly suggested to have only one system form in the template and have it submit using *POST*. This will ensure that no matter what user does no data is ever lost. However Aranea does not impose this idiom and one may just as well submit using *GET*, define system forms in widgets and use usual HTML links instead of JavaScript. See Section 4.2, “System Tags” for usage example.

Attributes

Attribute	Required	Description
id	*no*	The HTML "id" of the `<form>` tag that may be used in JavaScript. It will be autogenerated if omitted.
method	*yes*	HTTP submit method, either `GET` or `POST`.
enctype	*no*	Same as HTML `<form>` attribute `enctype`, defines how form data is encoded.

Variables

Variable	Description	Type
systemFormId	SystemForm FORM id.	`String`

4.2.7. <ui:messages>

This tag will render messages of given type if they are present in current `MessageContext`. When type is not specified, all types of messages are rendered. As `MessageContext` is typically used for error messages, it is common to render these messages somewhere near top of the page, where they can easily be spotted.

Attributes

Attribute	Required	Description
type	*no*	Message type.
styleClass	*no*	CSS class applied to rendered messages, default being `aranea-messages`.
style	*no*	CSS inline style applied to rendered messages. Use `styleClass` instead.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
<ui:messages type="info"/>
<ui:messages type="error" styleClass="custom-error-message-class"/>
<ui:messages/>
...
```

4.3. Basic Tags

4.3.1. <ui:attribute>

Defines an attribute of the containing element, where possible. See also Section 4.3.3, “<ui:element>”. Most form element tags accept attributes set by this tag too, see Section 4.3.1.1, “Examples”.

Attribute	Required	Description
name	*yes*	Attribute name.
value	*yes*	Attribute value.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
  <!-- set the onkeypress attribute for HTML input produced by ui:textInput-->
  <ui:textInput>
    <ui:attribute name="onkeypress" value="upperCase(this);"/>
  </ui:textInput>
...
```

4.3.2. <ui:elementContent>

Defines an HTML element content, meaning the body of the HTML element where text and other tags go.

4.3.3. <ui:element>

Defines HTML node, can be used together with `<ui:attribute>` and `<ui:elementContent>` to define a full HTML node.

Attribute	Required	Description
name	*no*	HTML element name.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:element name="span">
  <ui:attribute name="class" value="fancy"/>
  <ui:elementContent>Contents of fancy span.</ui:elementContent>
</ui:element>
```

4.3.4. <ui:keyboardHandler>

Registers a simple javascript keyboard handler.

Attribute	Required	Description
scope	*no*	When a keyboard event happens, it is usually associated with a certain form element / form / widget / etc. The object with which an event is associated is identified by a hierarchical id (e.g. there may be widget 'somelist', containing form 'somelist.form', containing textbox 'somelist.form.textbox'. The scope is a prefix of that id that must match in order for the handler to be triggered. For example, the handler with scope='somelist.form.textbox' will be triggered only when the event in the textbox occurs, but the handler with scope="somelist" will be triggered when any event in any of the elements inside any of the forms of "somelist" occurs. I.e. for any element with ID beginning with 'somelist'. When scope is not specified, a global handler is registered, that reacts to an event in any form/widget.
handler	*yes*	A javascript handler function that takes two parameters - the event object and the element id for which the event was fired. Example: `function(event, elementId) { alert(elementId); }`
keyCode	*no*	Keycode to which the event must be triggered. 13 means enter. Either keyCode or key must be specified, but not both.
key	*no*	Key, to which the event must be triggered. Key is specified as a certain 'alias'. The alias may be an ASCII character or a digit (this will denote the corresponding key on a US keyboard), a space (' '), or one of the following: 'return', 'escape', 'backspace', 'tab', 'shift', 'control', 'space', 'f1', 'f2', ..., 'f12'.

Examples

```
<!-- Globally-scoped F2 listener -->
<ui:keyboardHandler
  scope=""
  key="f2"
  handler="function() { alert('You pressed F2. Do it again if you dare!');}"/>
```

4.3.5. <ui:eventKeyboardHandler>

Registers a 'server-side' keyboard handler that sends an event to the specified widget.

Attribute	Required	Description
scope	*no*	Section 4.3.4, “<ui:keyboardHandler>”
widgetId	*no*	Id of Widget that is target of event produced by keyboard handler.

Attribute	Required	Description
eventId	*no*	Id of event that should be sent to target widget.
eventParam	*no*	Event parameters
updateRegions	*no*	Enumerates the regions of markup to be updated in this widget scope. Please see <ui:updateRegion> for details.
globalUpdateRegions	*no*	Enumerates the regions of markup to be updated globally. Please see <ui:updateRegion> for details.
keyCode	*no*	Keycode to which the event must be triggered. 13 means enter. Either keyCode or key must be specified, but not both.
key	*no*	Key, to which the event must be triggered. Key is specified as a certain 'alias'. The alias may be an ASCII character or a digit (this will denote the corresponding key on a US keyboard), a space (' '), or one of the following: 'return', 'escape', 'backspace', 'tab', 'shift', 'control', 'space', 'f1', 'f2', ..., 'f12'.

Examples

```
<!-- F2 listener that sends event 'add' to context widget upon activation -->
<ui:eventKeyboardHandler eventId="add" key="f2" widgetId="${widgetId}"/>
```

4.4. Widget Tags

4.4.1. <ui:widgetContext>

This tag should generally be the root of every widget JSP. It makes the widget view model accessible as an EL variable. It can also be used to render a descendant widget in the same JSP with the current widget. In the latter case you should set the *id* attribute to the identifier path of the descendant widget in question. Note that all widget-related tags inside of this tag will assume that the widget in question is their parent or ancestor (that is all the identifier paths will start from it).

Attributes

Attribute	Required	Description
id	*no*	A dot-separated widget identifier path leading from the current context widget to the new one.

Variables

Variable	Description
widget	The context widget instance. Can be used to access JavaBean property data from the widget (e.g. `${widget.foo}` will translate to a `getFoo()` widget

Variable	Description
	call}.
widgetId	The full dot-separated identifier of the context widget.
viewData	The view data of the context widget (see `BaseApplicationWidget.putViewData()`).
viewModel	The view model of the context widget.
scopedWidgetId	The scoped id of the context widget.

Examples

The most common usage of <ui:widgetContext> is as root tag for widget JSPs:

```
<?xml version="1.0" encoding="UTF-8"?>
...
  <ui:widgetContext>
    ...
    <c:out value="${viewData.myMessage}"/>
    ...
  </ui:widgetContext>
...
```

The other use case is to render a descendant widget:

```
<?xml version="1.0" encoding="UTF-8"?>
...
  <ui:widgetContext>
    ...
    <ui:widgetContext id="child.ofMyChild">
      <c:out value="${viewData.messageFromChildOfMyChild}"
    </ui:widgetContext>
    ...
  </ui:widgetContext>
...
```

4.4.2. <ui:widget>

This tag is used when one needs to render a child or descendant widget while still retaining in both current widget context and JSP. It publishes the widget view model and full identifier as EL variables, but does little else and does not setup a widget context (e.g. `<ui:widgetInclude>` tag will not take it into account).

Attributes

Attribute	Required	Description
id	*yes*	A dot-separated widget identifier path leading from the current context widget to the target widget.

Variables

Variable	Description
widget	The widget instance. Can be used to access JavaBean property data from the widget (e.g. `${widget.foo}` will translate to a `getFoo()` widget call}.

Variable	Description
widgetId	The full dot-separated identifier of the widget.
viewData	The view data of the widget (see `BaseApplicationWidget.putViewData()`).
viewModel	The view model of the widget.
scopedWidgetId	The scoped id of the context widget.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
  <ui:widgetContext>
    ...
    <ui:widget id="child.ofMyChild">
      <c:out value="${viewData.messageFromChildOfMyChild}"
          <ui:widgetInclude id="child"/>
    </ui:widget>
    ...
  </ui:widgetContext>
...
```

4.4.3. <ui:widgetInclude>

This tag is used to render some child or descendant widget. It will call the widget's `render()` method, which will allow the target widget to choose how to render itself.

Attributes

Attribute	Required	Description
id	*yes*	A dot-separated widget identifier path leading from the current context widget to the target widget.
path	*no*	Path to JSP, relative to `jspPath` of `StandardJspFilterService`.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
  <ui:widgetContext>
    ...
    <ui:widgetInclude id="child.ofMyChild"/>
    ...
  </ui:widgetContext>
...
```

4.4.4. <ui:globalWidgetInclude>

Much like `<ui:widgetInclude>` , but this tag allows to include not only descendants of current context widget but any widget that is accessible from global scope.

Attributes

Attribute	Required	Description
id	*yes*	A dot-separated full widget identifier.

4.5. Event-producing Tags

4.5.1. <ui:eventButton> and <ui:eventLinkButton>

These tags will render a button (or a link) that when clicked will send a specified event to the target widget with an optional `String` parameter.

Attributes

Attribute	Required	Description
id	*no*	HTML "id" of the element that can be used to access it via DOM.
labelId	*no*	The key of the localizable label that will be displayed on the button.
eventId	*no*	The identifier of the event that will be sent to the target widget.
eventParam	*no*	`String` event parameter that will accompany the event.
eventTarget	*no*	ID of receiving widget. Almost never set directly. Defaults to current context widget.
disabled	*no*	If set to a not null value will show the button disabled.
renderMode	*no*	Allowed values are (button | input) - the corresponding HTML tag will be used for rendering. Default is button. This attribute only applies to `<ui:eventButton>`, `<ui:eventLinkButton>` is always rendered with HTML link.
styleClass	*no*	The CSS class that will override the default one.
updateRegions	*no*	Comma separated list of update regions that should be updated upon button receiving event. This attribute is only needed when using AJAX features—ordinary HTTP requests always update whole page.
globalUpdateRegions	*no*	Comma separated list of global update regions that should be updated upon button receiving event. This attribute is only needed when using AJAX features—ordinary HTTP requests always update whole page.
onClickPrecondition	*no*	Precondition for deciding whether onclick event should go server side or not. If left unspecified, this is considered to be `true`.
tabindex	*no*	This attribute specifies the position of the current element in the tabbing order for the current document. This value must

Attribute	Required	Description
		be a number between 0 and 32767.

HTML, Styles and JavaScript

The `eventButton` tag writes out an HTML `<button>` closed tag with a default CSS class of "aranea-button".

The `eventLinkButton` tag writes out an HTML `<a>` open tag with a default CSS class of "aranea-link-button".

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
  <ui:widgetContext>
    ...
    <ui:eventButton eventId="test" eventParam="${bean.id}"/>
    <ui:eventLinkButton eventId="edit" eventParam="${bean.id}">
      <img src="editButton.png"/>
    </ui:eventLinkButton>
    ...
  </ui:widgetContext>
...
```

4.5.2. <ui:onLoadEvent>

This tag will register events that are executed when HTML page body has completely loaded. This tag can be used multiple times, all specified events will be added to event queue and executed in order of addition.

Attributes

Attribute	Required	Description
event	*yes*	Event to register.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
    <ui:onLoadEvent event="activateFlashLights();"/>
    <ui:onLoadEvent event="changeMenuBackGroundColor();"/>
...
```

4.5.3. <ui:registerPopups>

This tag checks presence of server-side session-threads that represent popups and adds system loadevent for opening them in new browser window at client-side. For tag to have an effect, HTML page `BODY` tag must have attribute onload event set to *AraneaPage* (See Aranea Clientside Javascript) onload event. Also, this tag only works inside <ui:systemForm> tag.

Attributes

This tag has no attributes.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
<ui:body>
  <ui:systemForm method="POST">
    <ui:registerPopups/>
  </ui:systemForm>
</ui:body>
...
```

4.6. HTML entity Tags

HTML entities can be inserted by using the predefined entity tags or using the <ui:entity> for entities that have not been defined by Aranea JSP library.

The `entity` tag accepts a attribute `code` which is used as &`code`; to get the HTML entity.

Attribute	Required	Description
code	*no*	HTML entity code, e.g. *nbsp* or *#012*.
count	*no*	Number of times to repeat the entity.

4.6.1. Predefined entity tags

The following predefined entities also accept the `count` attribute. It defines the number of times to repeat the entity.

Tag	Description
<ui:acute>	HTML `&acute;` entity.
<ui:copyright>	HTML `&copyright;` entity.
<ui:gt>	HTML `>` entity.
<ui:laquo>	HTML `&laquo;` entity.
<ui:lt>	HTML `<` entity.
<ui:nbsp>	HTML ` ` entity.
<ui:raquo>	HTML `&raquo;` entity.
<ui:acute>	HTML `&acute;` entity.

4.7. Putting Widgets to Work with JSP

Now we have defined enough JSP tags to render our example widget (see Section 2.7.8, “Putting It All Together”):

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:ui="http://araneaframework.org/tag-library/standard" version="1.2">
  <ui:widgetContext>
    <h3>Test widget</h3>

    Data field: <c:out value="${viewData.myData.field}"/>
    <ui:eventButton labelId="#Test" eventId="test"/>
  </ui:widgetContext>
</jsp:root>
```

We can use just usual JSTL Core library tags to access the widget view data, as long as the `<ui:widgetContext>` is present via the `viewData` EL variable.

4.8. Layout Tags

4.8.1. <ui:layout>

Represents a layout. Layouts allow to describe the way content will be placed on the page.

Attribute	Required	Applicable to:
width	*no*	Layout width.
rowClasses	*no*	Default style of rows in this layout.
cellClasses	*no*	Default style of cells in this layout.
styleClass	*no*	CSS class for tag.

Variables

Variable	Description	Type
rowClassProvider	Provides row class, usually should not be used from JSP.	`RowClassProvider`
cellClassProvider	Provides cell class, usually should not be used from JSP.	`CellClassProvider`

4.8.2. <ui:row>

Represents a row in layout.

Attribute	Required	Applicable to:
height	*no*	Row height.
cellClasses	*no*	Default style of cells in this row..
styleClass	*no*	Cell css class, defines the way the cell will be rendered.
overrideLayout	*no*	Boolean that determines whether row's

Attribute	Required	Applicable to:
		own styleClass completely overrides styleClass provided by surrounding layout (default behaviour), or is appended to layout's styleClass.

Variables

Variable	Description	Type
cellClassProvider	Provides cell class, usually should not be used from JSP.	`CellClassProvider`

4.8.3. <ui:cell>

Represents a cell in layout.

Attribute	Required	Applicable to:
height	*no*	Row height.
width	*no*	Row width.
colSpan	*no*	Cell colspan, same as in HTML.
rowSpan	*no*	Cell rowspan, same as in HTML.
styleClass	*no*	Cell css class, defines the way the cell will be rendered.
overrideLayout	*no*	Boolean that determines whether cells's own styleClass completely overrides styleClass provided by surrounding layout or row (default behaviour), or is appended to layout's or row's styleClass.

Examples

Layouts, rows and cells are used together like this:

```
<?xml version="1.0" encoding="UTF-8"?>
...
  <ui:layout rowClasses="even,odd" cellClasses="one,two,three,four">
    <ui:row>
      <ui:cell>
         <!-- cell content -->
      </ui:cell>
    </ui:row>
  </ui:layout>
...
```

4.8.4. <ui:updateRegion> and <ui:updateRegionRows>

These two tags define the update regions in the output that can be updated via AJAX requests. The update regions chosen to be updated when some event occurs is decided by tags that take the `updateRegion` attribute (See Section 5.2.1, “Common attributes for all form element rendering tags.”).

The `<ui:updateRegion>` should be used when defining updateregion when the region is not contained in HTML table (layout). The `<ui:updateRegionRows>` is for defining a region which is contained in HTML table and contains table rows itself. Updating just cells is not possible.

Attribute	Required	Description
id	*no*	The id of the region. Will be used to reference the region when POST'ing a form.
globalId	*no*	When not using the `globalId`, the full id will be formed by concatenating the context widget id with the specified `id`. If for a reason you would want to avoid that, then you specify the id with the `globalId` attribute.

Either `id` or `globalId` attribute is required.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- First update region, placed outside HTML table -->
<ui:updateRegion id="outsideTable">
</ui:updateRegion>

<ui:layout>
  <!-- Second update region, placed inside HTML table -->
  <ui:updateRegionRows id="insideTable"/>
    <ui:row>
      ...
    </ui:row>
  </ui:updateRegionRows>
</ui:layout>

<!-- Button that makes a background submit of specified event.
     When response arrives specified updateregions are updated -->
<ui:eventButton id="test" updateRegions="outsideTable,insideTable"/>
```

4.9. Presentation Tags

Aranea JSP library contains synonyms for some (deprecated) HTML presentation tags.

4.9.1. <ui:bold>

Acts as the HTML <*b*> tag.

4.9.2. <ui:italic>

Acts as <*i*> HTML tag.

4.9.3. <ui:font>

Acts as <*font*> HTML tag.

Attribute	Required	Applicable to:
face	*no*	The font face of the font.
color	*no*	The color of the font.

4.9.4. <ui:style>

Sets a CSS class for the tag content, acts as a <*span*> HTML tag with the *class* atribute set.

Attribute	Required	Applicable to:
styleClass	*no*	CSS class for tag.

4.9.5. <ui:newline>

Puts a visual new line (`<br/>`).

4.9.6. <ui:basicButton>

Represents an HTML form button.

Attribute	Required	Applicable to:
renderMode	*no*	Allowed values are (button | input) - the corresponding HTML tag will be used for rendering. Default is button.
id	*no*	Button id, allows to access button from JavaScript.
labelId	*no*	Id of button label.
onclick	*no*	`onClick` Javascript action.
styleClass	*no*	CSS class for button.
style	*no*	Inline CSS style for button.

4.9.7. <ui:basicLinkButton>

Represents a link with an onClick JavaScript action.

Attribute	Required	Applicable to:
id	*no*	Button id, allows to access button from

Attribute	Required	Applicable to:
		JavaScript.
styleClass	*no*	CSS class for tag.
style	*no*	Inline CSS style for tag.
onclick	*no*	`onClick` Javascript action.
labelId	*no*	Id of button label.

4.9.8. <ui:link>

Usual HTML link, acts as a <*a*> HTML tag.

Attribute	Required	Applicable to:
disabledStyleClass	*no*	CSS class for disabled link.
id	*no*	Link id, allows to access link from JavaScript.
href	*no*	Link target URL.
target	*no*	Link target, same as <*a*> HTML tag *target* attribute.
disabled	*no*	Controls whether the link is disabled, disabled link doesn't link anywhere.
styleClass	*no*	CSS class for tag.
style	*no*	Inline CSS style for tag.

4.10. Programming JSPs without HTML

Aranea standard tag library should mostly be enough to shelter end-users from the need to write HTML inside JSPs. Snippets of HTML are alright but using it too often tends to lead to inflexible UI; instead of embedding HTML in JSPs custom tags should be written if the need arises.

When writing JSPs without embedded HTML, programmers best friends are *styleClass* attributes of presentation tags, allowing tuning of tag appearances and *layout* tags.

Layout tags are tags extending `BaseLayoutTag`. Layout tags allow placing of rows inside them (and rows allow using of cells inside). Standard layout tag (<ui:layout>) outputs HTML *table*, and standard row and cell tags output HTML *tr* and *td* tags, respectively. This is by no means a requirement for layout tags—there are probably ways to achieve the same behaviour with correctly styled HTML *div* tags; but the tables should do just fine for majority of needs.

4.11. Customizing Tag Styles

Presentation tags (tags extending `PresentationTag` or implementing `StyledTagInterface`) have attribute `styleClass` that specifies the CSS style class used for rendering the tag. When `styleClass` attribute for tag is not specified, some default style bundled with Aranea is used; or in some cases no HTML `class` attribute is output at all—allowing cascading styles from some parent (HTML) tag to take over the presentation.

Presentation tags also have `style` attribute for specifying inline style for tag. Using it is discouraged—tweaking style classes to fit ones specific needs is highly recommended.

Some tags may have more than one attributes for defining tag style. For example `<ui:layout>` tag and other layout tags that extend `LayoutHtmlTag` or `BaseLayoutTag` have attributes `rowClasses` and `cellClasses` that specify the default styles for `<ui:row>` and `<ui:cell>` tags used within the layout. These can be overriden with row and cell own `styleClass` attribute.

To actually use new style(s) for some tag one often can just write a new CSS style (i.e. `"somestyle { background: #ffc; color: #900; text-decoration: none; }"`)—apply that and be done with it. For more complicated tags, one may need to take a quick peek at tag source code to see what HTML tags are output and design their styles accordingly. Most of the time that should not be necessary.

Changing default tag styles can be done in two ways—modifying CSS files or extending the tag one wants to customize with dynamic initializer like this:

```
{
    styleClass = "some-wanted-style";
}
```

needless to say, first method is very much preferred because creating custom tags just for changing tag styles is quite pointless.

There is also a `renderMode` attribute; in current tag library there are very few tags supporting this attribute. One of those is `ButtonHtmlTag` (`<ui:basicButton>`)—its renderMode should have value `"input"` or `"button"` (default) and it specifies whether the button should be rendered in HTML with `<input type=button ... >` or `<button ... >` tag. In the future, number of JSP tags having renderMode attribute will probably increase (this can be used to get rid of multiple JSP tags for rendering different types of (multi)selects, inputs and displays).

Attributes defining tag styles

Attribute	Required	Applicable to:
style	Inline CSS style applied to tag. Avoid.	Presentation tags.
styleClass	CSS class applied to tag.	Presentation tags.
rowClass	CSS class applied to rows inside the tag.	Layout tags.
cellClass	CSS class applied to cells inside the tag.	Layout tags, row tags.
renderMode	Defines the renderMode used for rendering the tag.	`<ui:basicButton>`, `<ui:eventButton>`, `<ui:button>`.

4.12. Making New JSP Tags

JSP tags are very application specific, need for additional or modified JSP tags arises quite often. Due to presentational nature of HTML and Javascript, extending the tags that really output HTML instead of providing some information to subtags is messy. We look here at some more general tags and contracts that should be

followed when writing Aranea JSP tags.

4.12.1. Utilities and base classes

Custom tags should extend at least *org.araneaframework.jsp.tag.BaseTag* that provides methods for registering subtags, manipulation of pagecontext and attribute evaluation.

```
import java.io.Writer;
import org.araneaframework.jsp.tag.entity.NbspEntityHtmlTag;
import org.araneaframework.jsp.util.JspUtil;

public class DummyTag extends BaseTag {
  public static String KEY = "org.araneaframework.jsp.tag.DummyTag";

  BaseTag subTag;

  @Override
  protected int doStartTag(Writer out) throws Exception {
    int result = super.doStartTag(out);

    // make this tag implementation accessible to subtags which
    // is quite pointless since this tag does not implement any useful interface.
    // it demonstrates Aranea JSP convention for providing info to subtags
    addContextEntry(KEY, this);

    // write some real output that ends up at the served web page
    JspUtil.writeOpenStartTag(out, "div");
    JspUtil.writeAttribute(out, "id", "dummyDivId");
    JspUtil.writeCloseStartTag(out);

    // it is possible to register in JAVA code too, this one just writes out nbsp entity.
    subTag = new NbspEntityHtmlTag();
    registerSubtag(subTag);
    executeStartSubtag(subTag);

    return result;
  }

  @Override
  protected int doEndTag(Writer out) throws Exception {
    executeEndTagAndUnregister(subTag);

    JspUtil.writeEndTag(out, "div");

    return super.doEndTag(out);
    // Now everything about this tag ceases to exist,
    // context entries are removed, souls are purged.
  }
}
```

`org.araneaframework.jsp.util.JspUtil` that was used here is an utility class containing some functions for writing out (XML) tags with somewhat less room for errors than just `out.write()`. Other notable methods provided by `BaseTag` are `getOutputData()` that returns response data, `getConfiguration()` and `getLocalizationContext()`. For tags with attributes, attribute evaluation functions that support *Expression Language (EL)* expressions are provided in `BaseTag`. Typical usage of these functions is following:

```
public void setWidth(String width) throws JspException {
  this.width = (String)evaluate("width", width, String.class);
}
```

Another common base tag for tags that output real HTML is `org.araneaframework.jsp.PresentationTag`. The `DummyTag` should really extend it too, since it outputs some HTML. `PresentationTag` defines *style* and *styleClass* attributes that can be applied to most HTML tags.

Important tag cleanup method is `doFinally()` that is called after rendering. It should be used to clear references to objects that should no longer be referenced after rendering. As in containers tag instances can live very long time, they can leak quite a lot of memory unless resources are deallocated.

4.12.2. Inheriting tag attributes from base tags.

Custom tags extending Aranea tags are able to accept all supertag attributes, but these must be also defined in TLD, otherwise the JSP containers will complain. As some base tags may be abstract, information about their attributes cannot be deduced from Aranea JSP standard TLD. To address this problem, Aranea distribution does the following: aranea.jar and aranea-jsp.jar include the file META-INF/aranea-standard.tcd (*TCD* stands for *Tag Class Descriptor*) which includes the attribute information for all Aranea Standard JSP classes. To make use of this information, one first generates TLD for custom tag classes and then merges the TCD information into it. It is done with `org.araneaframework.buildutil.TcdAndTldMerger` utility included in aranea.jar (since 1.0.10, previously it had to be compiled separately after downloading distribution). All custom compiled tag classes as well as Aranea JSP tag classes must be available on classpath when using this utility.

Example of using the `TcdAndTldMerger` utility:

```
<target name="tld">
  <!-- generate TLD without parent attribute information -->
  <webdoclet destdir="somedir" force="false" >
    <fileset dir="${src.dir}" includes="**/*Tag.java"/>

    <jsptaglib validatexml="true"
      shortName="shortName"
      filename="filename.tld"
      uri="customuri"
      description="description"
    />
  </webdoclet>

  <!-- invoke the TcdAndTldMerger utility -->
  <java classname="org.araneaframework.buildutil.TcdAndTldMerger" fork="true">
    <arg value="META-INF/aranea-standard.tcd"/>    <!-- Tag class descriptor to merge with -->
    <arg value="somedir/filename.tld"/>            <!-- Source TLD -->
    <arg value="somedir/filename.tld"/>            <!-- Destination TLD -->
    <classpath>
    <path refid="araneaclasspath"/>
      <path refid="compiledcustomtagclasses"/>
      <path refid="varia">
    </classpath>
  </java>

</target>
```

When running given target, one should see messages similar to following:

```
8 attributes for 'custom.RandomTag' found from 'org.araneaframework.jsp.tag.presentation.Presentation
```

4.12.3. Widgets and events

Sending events to widgets is accomplished with javascript submit functions, helpful utility being `org.araneaframework.jsp.util.JspUtil` and `org.araneaframework.jsp.util.JspWidgetCallUtil`. First one would construct `org.araneaframework.jsp.UiEvent` and (in case of HTML element which receives only one event) calls `JspUtil.writeEventAttributes(Writer out, UiEvent event)` and afterwards `writeSubmitScriptForEvent(Writer out, String attributeName)`.

```
//public UiEvent(String eventId, String eventTargetWidget, String eventParameter)
UiEvent event = new UiEvent("hello", "contextWidgetId", "name");
// long way to ouput custom attributes version
JspUtil.writeEventAttributes(out, event);
JspWidgetCallUtil.writeSubmitScriptForEvent(out, attributeName);
// short version
JspWidgetCallUtil.writeSubmitScriptForEvent(out, "onclick", event);

// both will output something like this:
    // arn-evntId="hello"
    // arn-trgtwdgt="contextWidgetId"
    // arn-evntPar="name"
    // onclick="return _ap.event(this);"
```

4.12.4. Layouts

New layouts are mostly concerned with styles or render layouts with some additional tags instead plain `table`, `tr`, `td`. As simple example, we define a layout that applies a class "error" to cells which contain invalid `FormElement`. Note that approach we use only works when cell tag is aware of the surrounding `FormElement` at the moment of rendering, meaning that `FormElement` is rendered in JSP something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<ui:formElement id="someId">
    <ui:cell>
        <ui:label/>
    </ui:cell>

    <ui:cell>
        <ui:textInput/>
    </ui:cell>
</ui:formElement>
...
```

What is needed foremost is a decorator for cells that are used inside invalid `FormElement`.

```
public class ErrorMarkingCellClassProviderDecorator implements CellClassProvider {
  protected CellClassProvider superProvider;
  protected PageContext pageContext;

  // constructs a decorator for superProvider, makes pageContext accessible
  public ErrorMarkingCellClassProviderDecorator(CellClassProvider superProvider, PageContext pageConte
    this.superProvider = superProvider;
    this.pageContext = pageContext;
  }

  public String getCellClass() throws JspException {
    FormElement.ViewModel formElementViewModel = (FormElement.ViewModel)
      pageContext.getAttribute(FormElementTag.VIEW_MODEL_KEY, PageContext.REQUEST_SCOPE);
    // superProvider.getCellClass() may only be called once, otherwise moves on to next cell's style
    String superClass = superProvider.getCellClass();

    if (formElementViewModel != null && !formElementViewModel.isValid()) {
      if (superClass != null)
        return superClass + " error";
      else
        return "error";
    }

    return superClass;
  }
}
```

Actual layout tag that decorates its cells according to described logic:

```
public class CustomLayoutTag extends LayoutHtmlTag {
  protected int doStartTag(Writer out) throws Exception {
    int result = super.doStartTag(out);
    addContextEntry(CellClassProvider.KEY, new ErrorMarkingCellClassProviderDecorator(this, pageContex

    return result;
  }
}
```

Chapter 5. Forms and Data Binding

One of the most common tasks in web applications is gathering user input, converting it to model objects and then validating it. This is typically referred to as *data binding* and every major web framework has support for this activity. In this chapter we will introduce the widgets and supporting API that implement this tasks.

5.1. Forms

Unlike many other frameworks, in Aranea request processing, validating and data binding is not a separate part of the framework, but just another component. Specially it is widget `org.araneaframework.uilib.form.FormWidget` and some support widgets. A typical form is shown on Figure 5.1, “Form example”.

Figure 5.1. Form example

5.1.1. FormWidget

Let's say we have a `Person` model JavaBean that looks like this:

```
public class Person {
  private Long id;
  private String name;
  private String surname;
  private String phone;

  public Long getId() {return id;}
  public void setId(Long id) {this.id = id;}

  public String getName() {return name;}
  public void setName(String name) {this.name = name;}

  public String getSurname() {return surname;}
  public void setSurname(String surname) {this.surname = surname;}

  public String getPhone() {return phone;}
  public void setPhone(String phone) {this.phone = phone;}
}
```

A typical form will be created and used like this:

```
...
private BeanFormWidget personForm;
private Person person;
...
protected void init() {
  ...
  personForm = new BeanFormWidget(Person.class);
```

```
  addWidget("personForm", personForm);

  personForm.addBeanElement("name", "#Name", new TextControl(new Long(3), null), true);
  personForm.addBeanElement("surname", "#Last name", new TextControl(), true);
  personForm.addBeanElement("phone", "#Phone no", new TextControl(), true);
  ...
  person = lookupPersonService().getSomePerson();
  personForm.readFromBean(person);
  ...
}
...
```

Note that here we basically do three things:

Create and register the form
: The line `new BeanFormWidget(Person.class)` creates a new form widget that is associated with the *JavaBean* model class `Person` . The line `addWidget("personForm", personForm)` initializes and registers the form allowing it to function.

Add form fields
: The line `personForm.addBeanElement("name", "#Name", new TextControl(new Long(3), null), true)` adds an element associated with the JavaBean property "name" (this is also the identifier of the field), with a label "Name" (labels in Aranea are localizable by default and "#" escapes a non-localizable string), a text box control with a minimal length of 3 and that is mandatory.

Write JavaBean
: The line `personForm.readFromBean(person)` reads the data from JavaBean properties to the corresponding form fields.

Now that we have created the form we show how to process events, validate and read the request data. The following example code should be in the same widget as the previous:

```
...
private void handleEventSave() {
  if (personForm.convertAndValidate()) {
   personForm.writeToBean(person);
   ...
   lookupPersonService()().savePerson(person);
  }
}
...
```

This code will execute if an event "save" comes and will do the following:

- Convert the request data to the JavaBean types and validate it according to the rules specified in controls (e.g. minimal length). Wrapping event body in `if (personForm.convertAndValidate()) {...}` is a generic idiom in Aranea as we believe that explicitly leads to flexibility. By default the values will be just read from request without any parsing, conversion or validation and the latter will be done only after the `convertAndValidate()` call. This allows for example to validate only a subform or even one element, by calling only their `convertAndValidate()` method.
- Read the `person` object from the form, filling it in with the user data. Note that the same object that was originally read from the business layer is used here and forms take care of merging the new data and preserving the old.

Note the use of the `getValueByFullName()` method. Form API contains several such methods (named `*ByFullName()`), which allow to access fields, controls and values using full dot-separated element names.

If you have a composite JavaBean (containing other JavaBeans) you may want to create a form with a similar

structure. Let's say that our `Person` bean contains an `Address` under "address" JavaBean property:

```
...
personForm = new BeanFormWidget(Person.class);
addWidget("personForm", personForm);
...
BeanFormWidget addrForm = personForm.addBeanSubForm("address");
addrForm.addBeanElement("postalCode", "#Postal code", new TextControl(), true);
addrForm.addBeanElement("street", "#Street", new TextControl(), true);
...
```

Note that the fields will be available from the main form using a dot-separated name, e.g. `String street = (String) personForm.getValueByFullName("address.street")`.

5.1.2. Controls

At the core of the data binding API lies the notion of *controls* (`org.araneaframework.uilib.form.Control`). Controls are the widgets that do the actual parsing of the request parameters and correspond to the controls found in HTML forms, like textbox, textarea, selectbox, button, ... Additionally controls also do a bit of validating the submitted data. For example textbox control validates the minimum and maximum string length, since the HTML tag can do the same. Programmer usually does not read values from `Control` directly, but from `FormElement` that takes care of converting value of `Control` to `FormElement Data`.

The following example shows how to create a control:

```
...
TextControl textBox = new TextControl(new Long(10), null);
...
```

This code will create a textbox with a minimal length of 10. Note that this code does not yet put the control to work, as controls are never used without forms, which are reviewed in the next section.

Follows a table of standard controls all found in `org.araneaframework.uilib.form.control` package:

Control	Description
`ButtonControl`	A control that represents a HTML form button.
`CheckboxControl`	A control that represents a binary choice and is usually rendered as a checkbox.
`DateControl`	A date selection control that allows to choose a date. Supports custom formats of date input and output.
`DateTimeControl`	A date and time selection control that allows to choose a date with a corresponding time. Supports custom formats of date and time input and output.
`DisplayControl`	A control that can be used to render a read-only value that will not be submitted with an HTML form.
`FileUploadControl`	A control that can be used to upload files to the server.
`FloatControl`	A textbox control that constrains the text to be floating-point numbers. Can also check the allowed minimum and maximum limits.
`HiddenControl`	A control that can be used to render an invisible value that will be submitted with an HTML form.

Control	Description
`NumberControl`	A textbox control that constrains the text to be integer numbers. Can also check the allowed minimum and maximum limits.
`TimeControl`	A time selection control that allows to choose a time of day. Supports custom formats of time input and output.
`TextareaControl`	A multirow textbox control that can constrain the inserted text minimal and maximal length.
`TextControl`	A simple textbox control with one row of text that can constrain the inserted text minimal and maximal length.
`AutoCompleteTextControl`	`TextControl` with autocompletion capability.
`TimestampControl`	Similar to `DateControl` but works with `java.sql.TimeStamp`.
`SelectControl`	A control that allows to select one of many choices (may be rendered as a dropdown list or option buttons). Ensures that the submitted value was one of the choices.
`MultiSelectControl`	A control that allows to select several from many choices (may be rendered as a multiselect list or checkbox list). Ensures that the submitted values are a subset of the choices.

`SelectControl` and `MultiSelectControl` deserve a special mention, as they need a bit more handling than the rest. The difference comes from the fact that we also need to handle the selectable options, which we refer to as `DisplayItem`. Each `DisplayItem` has a label, a string value and can be disabled. Disabled display items cannot be selected in neither select box nor multiselect box.

Both `SelectControl` and `MultiSelectControl` implement the `DisplayItemContainer` interface that allows to manipulate the `DisplayItem`:

```
interface DisplayItemContainer {
  void addItem(DisplayItem item);
  void addItems(Collection items);
  void clearItems();
  List getDisplayItems();
  int getValueIndex(String value);
}
```

In addition to this interface we also provide a `DisplayItemUtil` that provides some support methods on display items. These include the method `addItemsFromBeanCollection` that allows to add the items to a (multi)select control from a business method returning a collection of model JavaBean objects (which is one of the most common use cases). So a typical select control will be filled as follows:

```
SelectControl control = new SelectControl();
control.addItem(new DisplayItem(null, "- choice -"));
DisplayItemUtil.addItemsFromBeanCollection(
  control,
  lookupMyService().getMyItemCollection(),
  "value",
  "label");
```

Controls can also listen to user events. For example `ButtonControl` can react to an `onClick` event, while most others can react to an `onChange` event. The only thing needed to receive the control events is to register an appropriate event listener:

```
...
SelectControl selControl = new SelectControl();
FormElement selEl = form.addBeanElement("clientId", "#Client id", selControl, true);
selControl.addOnChangeEventListener(new OnChangeEventListener() {
  public void onChange() {
    //We convert and validate one element only as the rest of the form
    //might be invalid
    if (selEl.convertAndValidate()) {
      Long clientId = (Long) selEl.getValue();
      //Now we can use the client id to do whatever we want
      //E.g. update another select control
    }
  }
});
...
```

`onChange` events are also produced by text boxes and similar, so the user input can processed right after the user has finished it.

5.1.3. Constraints

Though controls provide some amount of validation they are limited only to the rules that can be controlled on the client-side. To support more diverse rules Aranea has `org.araneaframework.uilib.form.Constraint`, that allows to put any logical and/or business validation rules. Typically constraints are used as follows:

```
...
myForm.addBeanElement("registration", "#Registration", new DateControl(), true);
myForm.getElement("registration").setConstraint(new AfterTodayConstraint(false));
...
```

The `org.araneaframework.uilib.form.constraint.AfterTodayConstraint` makes sure that the date is today or later, with the boolean parameter indicating whether today is allowed. The constraint will validate if the form or the element in question is validated (e.g. `convertAndValidate()` is called) and will show an error message to the user, if the constraint was not satisfied. The error message is customizable using localization and involves the label of the field being validated.

The following is a more complex example that shows how to use constraints that apply to more than one field, and how to combine constraints using logical expressions:

```
...
searchForm = new FormWidget();

//Adding form controls
searchForm.addElement("clientFirstName", "#Client first name",
  new TextControl(), new StringData(), false);
searchForm.addElement("clientLastName", "#Client last name",
  new TextControl(), new StringData(), false);

searchForm.addElement("clientAddressTown", "#Town",
  new TextControl(), new StringData(), false);
searchForm.addElement("clientAddressStreet", "#Street",
  new TextControl(), new StringData(), false);

//First searching scenario
AndConstraint clientNameConstraint = new AndConstraint();
clientNameConstraint.addConstraint(
  new NotEmptyConstraint(searchForm.getElementByFullName("clientFirstName")));
clientNameConstraint.addConstraint(
  new NotEmptyConstraint(searchForm.getElementByFullName("clientLastName")));

//Second searching scenario
AndConstraint clientAddressConstraint = new AndConstraint();
```

```
clientAddressConstraint.addConstraint(
  new NotEmptyConstraint(searchForm.getElementByFullName("clientAddressTown")));
clientAddressConstraint.addConstraint(
  new NotEmptyConstraint(searchForm.getElementByFullName("clientAddressStreet")));

//Combining scenarios
OrConstraint searchConstraint = new OrConstraint();
searchConstraint.addConstraint(clientNameConstraint);
searchConstraint.addConstraint(clientAddressConstraint);

//Setting custom error message
searchConstraint.setCustomErrorMessage("Not enough data for search!");

//Setting constraint
searchForm.setConstraint(searchConstraint);

//Putting the widget
addWidget("searchForm", searchForm);
...
```

The example use case is a two scenario search—either both client first name and client last name fields are filled in or both town and street address fields are filled in, otherwise an error message "Not enough data for search!" is shown. The constraints will be validated when `convertAndValidate()` method is called on `searchForm`. Note that the constraint is added to the form itself, rather than to its elements—this is a typical idiom, when the constraint involves several elements.

Table of standard `Constraints`.

Constraint	Purpose
`AfterTodayConstraint`	Field constraint, checks that field contains `Date` later than current date.
`NotEmptyConstraint`	Field constraint, checks that field contains non-empty value.
`NumberInRangeConstraint`	Field constraint, checks that number in a field belongs on given range (integer only).
`StringLengthInRangeConstraint`	Field constraint, checks that length of a string in a field falls within given boundaries.
`RangeConstraint`	Multiple field constraint, checks that value of one field is lower than value of other field. Field values must `Comparable`.
`AndConstraint`	Composite constraint, checks that all subconstraints are satisfied.
`OrConstraint`	Composite constraint, checks that at least one subconstraint is satisfied.

There are two constraints that deserve a special mention. One of them is `OptionalConstraint` that will only let its subconstraint to validate the field, if the field has been submitted by user (it is very useful for instance when non-mandatory fields must nevertheless follow some pattern, whereas empty input should still be allowed).

The other constraint is called `GroupedConstraint`. It is useful in cases when different constraints should be activated depending on the particular state of the component (a typical use case is that some groups of fields are made mandatory in different states of document approval). The constraint is created using the `ConstraintGroupHelper` as follows:

```
...
ConstraintGroupHelper groupHelper = new ConstraintGroupHelper();
AndConstraint andCon = new AndConstraint();
andCon.addConstraint(
```

```
  groupHelper.createGroupedConstraint(new NotEmptyConstraint(field1), "group1"));
andCon.addConstraint(
  groupHelper.createGroupedConstraint(new NotEmptyConstraint(field2), "group1"));
andCon.addConstraint(
  groupHelper.createGroupedConstraint(new NotEmptyConstraint(field3), "group2"));
andCon.addConstraint(
  groupHelper.createGroupedConstraint(new NotEmptyConstraint(field4), "group2"));
form.setConstraint(andCon);

//Now only field1 and field2 will be required from user!
groupHelper.setActiveGroup("group1");
...
```

Custom Constraints

It is a very common need to validate some additional logic for a particular field (e.g. a field must follow some particular pattern). In this case it is comfortable to create a custom constraint. Most often the constraint is associated with one field only, so we will extend the `BaseFieldConstraint`, which supports this particular idiom:

```
...
public class PersonIdentifierConstraint extends BaseFieldConstraint {
  public void validateConstraint() {
    if (!PersonUtil.validateIdentifier(getValue()) {
      addError("Field '" + getLabel() + "' is not a valid personal identifier");
    }
  }
}
...
```

Note that we can use `getValue()` that contains the converted value of the field. We can also use the fields label via `getLabel()`. We might also want to localize the message and in such a case you will find `MessageUtil` to contain some helpful methods.

If we need to validate more than one field we should extend the `BaseConstraint` and take those fields into the constructor. In this case the developer will have to provide this fields to the constraint and the constraint should be added to the enclosing form.

5.1.4. Data

The typical use of forms includes associating the form fields with JavaBean properties. However this is not always possible, since it is not feasible to make a JavaBean property for each and every form field. In such cases one may still want to use type conversion and data validation. To do that forms allow the `org.araneaframework.uilib.form.Data` and its subclasses (subclasses correspond to specific types) to be associated with the field:

```
...
personForm = new BeanFormWidget(Person.class);
addWidget("personForm", personForm);
...
personForm.addElement("numberOfChildren", "#No. of chidren",
  new NumberControl(), new LongData(), true);
...
```

In such a case one can retrieve the data directly from the field:

```
...
private void handleEventSave() {
  if (myForm.convertAndValidate()) {
```

```
    ...
    Long numberOfChildren = (Long) personForm.getValueByFullName("numberOfChildren");
    //Alternative:
    //FormElement nocEl = (FormElement) personForm.getElement("numberOfChildren");
    //Long numberOfChildren = (Long) nocEl.getValue();
    ...
  }
}
...
```

If there is no JavaBean to associate the form with `org.araneaframework.uilib.form.FormWidget` may be used instead of `BeanFormWidget`.

Note that the reason for existence of `Data` objects is that Java types correspond poorly to some restricted types—for instance enumerations, type encodings and collections container types (this problem is somewhat solved in Java 5, but Aranea is compatible with Java 1.3).

Table of `Data` types.

Data	Value Type
`BigDecimalData`	`java.math.BigDecimal`
`BigDecimalListData`	`List <java.math.BigDecimal>`
`BooleanData`	`java.lang.Boolean`
`BooleanListData`	`List <java.lang.Boolean>`
`DateData`	`java.util.Date`
`DisplayItemListData`	`List <org.araneaframework.uilib.support.DisplayItemDisplayItem>`
`FileInfoData`	`org.araneaframework.uilib.support.FileInfo`
`IntegerData`	`java.lang.Integer`
`IntegerListData`	`List <java.lang.Integer>`
`LongData`	`java.lang.Long`
`LongListData`	`List <java.lang.Long>`
`StringData`	`java.lang.String`
`StringListData`	`List <java.lang.String>`
`TimestampData`	`java.sql.Timestamp`
`YNData`	`java.lang.String`

Finally `Data` constructor also accepts both a `Class` instance and a simple string. So if you have a custom datatype with an appropriate converter (see next section) you can just assign the data with the same type (in fact if you have your own converter the type doesn't matter that much, it will just allow some checks to be done on the programmer).

5.1.5. Converters

Converter sole purpose is conversion of values with one type to values of another type. Conventionally

converter which `convert()` method accepts object of type *A* and returns object of type *B* is named `AToBConverter`. Converter from type *B* to type *A* is obtained with `new ReverseConverter(new AToBConverter())`.

```
public interface Converter extends Serializable, FormElementAware {
    public void setFormElementCtx(FormElementContext feCtx);
    public Object convert(Object data);
    public Object reverseConvert(Object data);
    public Converter newConverter();
}
```

Converters are used internally to convert `Control` values to values of `FormElement Data` and vice-versa. Converters are usually looked up from `ConverterFactory`, but each `FormElement` can be set explicit `Converter` by calling `FormElement.setConverter()`. Direction of `Converter` set this way should be from `FormElement Control` value type to `FormElement Data` type.

5.2. Forms JSP Tags

Form JSP tags can be divided into two categories—tags providing *contexts* (<ui:form>, <ui:formElement>) and tags for *rendering* form elements containing different controls. We will first describe the attributes that are common to all form element rendering tags; then proceed to explain *context* tags and different form element rendering tags with their unique attributes.

5.2.1. Common attributes for all form element rendering tags.

Attribute	Required	Description
id	*no/yes*	Id of form element to be rendered. If not specified, it is usually taken from current form element context (Section 5.2.3, “<ui:formElement>”). For few tags, it is required.
events	*no*	Whether element will send events that are registered by server-side, `true` by default.
validateOnEvent	*no*	Whether the form should be validated on the client-side (or by making AJAX request to server) when *element generates an event* (this is `false` by default and is not supported by any default Aranea JSP tags).
tabindex	*no*	This attribute specifies the position of the current element in the tabbing order for the current document. This value must be a number between 0 and 32767.
updateRegions	*no*	Comma separated list of update regions that should be updated upon button receiving event. This attribute is only needed when using AJAX features— ordinary HTTP requests always update whole page.
globalUpdateRegions	*no*	Comma separated list of global update regions that should be updated upon button receiving event. This attribute is only needed when using AJAX features— ordinary HTTP requests always update whole page.
styleClass	*no*	CSS class applied HTML tag(s) that are used for rendering

Attribute	Required	Description
		element.
style	*no*	Inline CSS style applied to HTML tag(s) that are used for rendering element.

5.2.2. <ui:form>

Specifies form context for inner tags. Form view model and id are made accessible to inner tags as EL variables.

Attributes

Attribute	Required	Description
id	*no*	Id of context form. When not specified, current form context is preserved (if it exists).

Variables

Variable	Description	Type
form	View model of form.	`FormWidget.ViewModel`
formId	Id of form.	`String`
formFullId	Full id of form.	`String`
formScopedFullId	Full scoped id of form.	`String`

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="loginForm">
    ...  <!-- formElements, formElementLabels, ... --> ...
</ui:form>
```

5.2.3. <ui:formElement>

Specifies form element context for inner tags. Must be surrounded by <ui:form> tag. Form element view model, id and value are made accessible to inner tags as EL variables.

Attributes

Attribute	Required	Description
id	*yes*	Id of context form element.

Variables

Variable	Description	Type
formElement	View model of form element.	`FormElement.ViewModel`
formElementId	Id of form element.	`String`
formElementValue	Value currently kept inside form element.	`Object`

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="loginForm">
    <ui:formElement id="username">
        ...
    </ui:formElement>
</ui:form>
```

5.2.4. <ui:label>

Renders localizable label bound to form element. Rendered with HTML and <label> tags.

Attributes

Attribute	Required	Description
id	*no*	Id of form element which label should be rendered. If left unspecified, form element id from form element context is used.
showMandatory	*no*	Indicates whether mandatory input fields label is marked with asterisk. Value should be `true` or `false`, default is `true`
showColon	*no*	Indicates whether colon is shown after the label. Default is `true`.

Also has standard `style` and `styleClass` attributes.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="loginForm">
    <ui:row>
        <ui:formElement id="username">
            <ui:cell>
                <ui:label/>
            </ui:cell>
        </ui:formElement>
    </ui:row>
</ui:form>
```

5.2.5. <ui:simpleLabel>

Renders localizable label (with HTML and <label> tags).

Attributes

Attribute	Required	Description
labelId	*yes*	ID of label to render.
showMandatory	*no*	Indicates whether label is marked with asterisk. Value should be `true` or `false`, default is `false`
showColon	*no*	Indicates whether colon is shown after the label. Default is `true`.
for	*no*	ID of the form element for which the label is created.

Also has standard `style` and `styleClass` attributes.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="loginForm">
    <ui:row>
        <ui:cell>
            <ui:simpleLabel labelId="username.input.label" showMandatory="true" for="username"/>
        </ui:cell>
    </ui:row>
</ui:form>
```

5.2.6. <ui:button>

Renders form buttons that represent `ButtonControls`. Either HTML <button> or <input type="button" ... > will be used for rendering.

Attributes

Attribute	Required	Description
showLabel	*no*	Indicates whether button label is shown.Value should be `true` or `false`, default is `true`
onClickPrecondition	*no*	Precondition for deciding whether onclick event should go server side or not. If left unspecified, this is considered to be `true`.
renderMode	*no*	Allowed values are `button` and `input`—the corresponding HTML tag will be used to render the button. Default is button.

Also see Section 5.2.1, “Common attributes for all form element rendering tags.”.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="loginForm">
    <ui:button id="loginButton"/>
</ui:form>
```

5.2.7. <ui:linkButton>

Renders HTML link that represents `ButtonControl`. HTML <a href="javascript:" ... > tag will be used for rendering. Default `styleClass="aranea-link"`.

Attributes

Attribute	Required	Description
showLabel	*no*	Indicates whether button label is shown.Value should be `true` or `false`, default is `true`
onClickPrecondition	*no*	Precondition for deciding whether registered onclick event should go server side or not. If left unspecified, this is considered to be `true`.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.8. <ui:formKeyboardHandler>

Registers a simple keyboard handler. Invokes a `uiRegisterKeyboardHandler` javascript. This is basically the same stuff as <ui:keyboardHandler> with a few modifications.

There is no `scope` attribute. Instead, the tag assumes that it is located inside a form, and takes the full id of that form as its scope.

As an alternative to specifying the `handler` attribute, you may specify a form element and a javascript event to invoke on that element. You specify the element by its id relative to the surrounding form. The event is given as a name of the javascript function to be invoked on the element. For example, if you specify the element as "someButton", and event as "click", then when the required keyboard event occurs, the following javascript will be executed:

```
var el = document.getElementById("<form-id>.someButton");
el.click();
```

Attributes

Attribute	Required	Description
handler	*no*	A javascript handler function that takes two parameters - the event object and the element id for which the event was fired. Example: `function(event, elementId) { alert(elementId); }` Either handler or elementId/event pair should be specified, not both.
subscope	*no*	Specifies form element which is the scope of this handler. By default the "scope" (as in `<ui:keyboardHandlerTag>`) of this keyboard handler is the form inside which the handler is defined. By specifying this, scope of certain element may be narrowed. For example if the handler is defined inside form "myForm", and subscope is specified as "myelement", the scope of the handler will be "myForm.myelement", not the default "myForm". The handler will therefore be active only

Attribute	Required	Description
		for the element 'someElement'".
elementId	*no*	Sets the (relative) `id` of the element whose javascript event should be invoked. The `id` is relative with respect to the surrounding form. Instead of this attribute, element's full id may be set using the `fullElementId` attribute, but only one of those attributes should be set at once.
fullElementId	*no*	Sets the full `id` of the element whose javascript event should be invoked.
event	*no*	Set the javascript event that should be invoked when keypress is detected—"click" and "focus" are safe for most controls. If target element (the one given by `elementId`) is a selectbox "select" may be used. For more, javascript reference should be consulted. This attribute is not foolproof and invalid javascript may be produced when it is not used cautiously.
keyCode	*no*	Keycode to which the event must be triggered. Either keyCode or key must be specified, but not both.
key	*no*	Key to which the event must be triggered, accepts key "aliases" instead of codes. Accepted aliases include `F1..F12`, `RETURN`, `ENTER`, `BACKSPACE`, `ESCAPE`, `TAB`, `SHIFT`, `CONTROL`, `SPACE`.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="loginForm">
    <ui:eventButton id="btnLogin" eventId="login" labelId="button.login.enter"/>
    <ui:formKeyboardHandler fullElementId="btnLogin" key="enter"/>
</ui:form>
```

5.2.9. <ui:formEnterKeyboardHandler>

Same as `<ui:formKeyboardHandlerTag>` except `key` is already set to `enter`.

5.2.10. <ui:formEscapeKeyboardHandler>

Same as `<ui:formKeyboardHandlerTag>` except `key` is already set to `escape`.

5.2.11. <ui:textInput>

Form text input field, represents `TextControl`. It is rendered in HTML with `<input type="text" ...>` tag. Default `styleClass="aranea-text"`.

Attributes

Attribute	Required	Description
size	*no*	Maximum length of accepted text (in characters).
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`. For this tag, *onchange* event is simulated with *onblur*.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="someForm">
  <ui:formElement id="firstField">
     <!-- Renders input field binded to form's firstField element -->
     <ui:textInput/>
  </ui:formElement>
</ui:form>
```

5.2.12. <ui:autoCompleteTextInput>

Form text input field, represents `AutoCompleteTextControl`. It is rendered in HTML with `<input type="text" ...>` tag. Default `styleClass="aranea-text"`. It is able to make background AJAX request to the server, fetching suggested completions to user input and displaying these to the user.

Attributes

Attribute	Required	Description
size	*no*	Maximum length of accepted text (in characters).
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`. For this tag, *onchange* event is simulated with *onblur*.
divClass	*no*	CSS class attribute assigned to <DIV> inside which suggestions are presented.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.13. <ui:textInputDisplay>

Form text display field, represents `TextControl`. It is rendered in HTML with `<span ...>` tag. Default `styleClass="aranea-text-display"`.

Attributes

Has standard `id` and `styleClass` attributes.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="someForm">
  <ui:formElement id="firstField">
     <!-- Renders display field for form's firstField element -->
     <ui:textInputDisplay/>
  </ui:formElement>
</ui:form>
```

5.2.14. <ui:numberInput>

Form number input field, represents `NumberControl`. It is rendered in HTML with `<input type="text" ...>` tag. Default `styleClass="aranea-number"`.

Attributes

Attribute	Required	Description
size	*no*	Maximum length of accepted text (in characters).
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`. For this tag, *onchange* event is simulated with *onblur*.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.15. <ui:numberInputDisplay>

Form number display field, represents `NumberControl`. It is rendered in HTML with `<span ...>` tag. Default `styleClass="aranea-number-display"`.

Attributes

Has standard `id` and `styleClass` attributes.

5.2.16. <ui:floatInput>

Form floating-point number input field, represents `FloatControl`. It is rendered in HTML with `<input type="text" ...>` tag. Default `styleClass="aranea-float"`.

Attributes

Attribute	Required	Description
size	*no*	Maximum length of accepted floating-point number (in characters).
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`. For this tag, *onchange* event is simulated with *onblur*.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.17. <ui:floatInputDisplay>

Form floating-point number display field, represents `FloatControl`. It is rendered in HTML with `<span ...>` tag. Default `styleClass="aranea-float-display"`.

Attributes

Has standard `id` and `styleClass` attributes.

5.2.18. <ui:passwordInput>

Form number input field, represents `TextControl`. It is rendered in HTML with `<input type="password" ...>` tag. Default `styleClass="aranea-text"`.

Attributes

Attribute	Required	Description
size	*no*	Maximum length of password (in characters).
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`. For this tag, *onchange* event is simulated with *onblur*.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.19. <ui:textDisplay>

Form text display field, represents `DisplayControl`, displays element value as `String`. It is rendered in HTML with `<span ...>` tag.

Attributes

Has standard `id` and `styleClass` attributes.

5.2.20. <ui:valueDisplay>

Puts form element value in page scope variable, represents `DisplayControl`. It does not output any HTML.

Attributes

Attribute	Required	Description
var	*true*	Name of the page-scoped EL variable that will be assigned element value.

Also has standard `id` attribute.

5.2.21. <ui:textarea>

Form text input area, represents `TextareaControl`. It is rendered in HTML with `<textarea ...>` tag. Default

`styleClass="aranea-textarea"`.

Attributes

Attribute	Required	Description
cols	*true*	Number of visible columns in textarea.
rows	*true*	Number of visible rows in textarea.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="someForm">
  <ui:formElement id="longLongText">
    <ui:cell>
      <ui:textarea rows="15" cols="150"/>
    </ui:cell>
  </ui:formElement>
</ui:form>
```

5.2.22. <ui:richtextarea>

Form text input area, represents `TextareaControl`. It is rendered in HTML with `<textarea ...>` tag with `styleClass="richTextEditor"`. The area is displayed as a rich text editor. The configuration of the editor is done via `<ui:richTextAreaInit>`. The tag shares all the attributes of the `<ui:textarea>` except the `styleClass` which cannot be set for this tag.

5.2.23. <ui:richTextAreaInit>

A tag for configuring the rich textareas. The tinyMCE [http://tinymce.moxiecode.com/] WYSIWYG editor is attached to the textareas defined via `<ui:richTextarea>` . The configuration lets you choose the looks, buttons, functionality of the editor. See tinyMCE configuration reference [http://tinymce.moxiecode.com/tinymce/docs/reference_configuration.html] for different configurable options.

The configuration is done via nesting key value pairs inside the `<ui:richTextAreaInit>`. For the key value pairs the `<ui:attribute>` tag is used. See the example for an overview.

The `editor_selector` and `mode` options are set by default and should not be changed. The default `theme` is "simple".

Important: the configuration should be done in the `<head>` section of the HTML document.

Example

```
<ui:richTextAreaInit>
  <ui:attribute name="theme" value="advanced"/>
  <ui:attribute name="theme_advanced_buttons1" value="bold,italic,underline,separator,code"/>
  <ui:attribute name="theme_advanced_toolbar_location" value="top"/>
  <ui:attribute name="theme_advanced_toolbar_align" value="left"/>
  <ui:attribute name="theme_advanced_path_location" value="bottom"/>
</ui:richTextAreaInit>
```

5.2.24. <ui:textareaDisplay>

Form text display area, represents `TextareaControl`. It is rendered in HTML with `<span ...>` tag. Default `styleClass="aranea-textarea-display"`.

Attributes

Attribute	Required	Description
escapeSingleSpaces	*false*	Boolean, specifying whether even single spaces (blanks) should be replace with entities in output. It affects browser performed text-wrapping. Default value is `false`. Attribute is available since tag-library version 1.0.6.

Also has standard `id` and `styleClass` attributes.

5.2.25. <ui:hiddenInput>

Represents a "hidden" form input element—`HiddenControl`. It is rendered in HTML with `<input type="hidden" ...>` tag.

Attributes

See Section 5.2.1, "Common attributes for all form element rendering tags.". However, rendered tag is not visible to end-user, thus using any attributes is mostly pointless.

5.2.26. <ui:checkbox>

Form checkbox *input* field, represents `CheckboxControl`. By default `styleClass="aranea-checkbox"`. Rendered in HTML with `<input type="checkbox" ...>` tag.

Attributes

Attribute	Required	Description
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.27. <ui:checkboxDisplay>

Form checkbox *display* field, represents `CheckboxControl`. By default `styleClass="aranea-checkbox-display"`. Rendered in HTML inside `<span>` tag.

Attributes

Has standard `id` and `styleClass` attributes.

5.2.28. <ui:fileUpload>

Form file upload field, represents `FileUploadControl`.

Attributes

See Section 5.2.1, "Common attributes for all form element rendering tags.".

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
<ui:form id="uploadForm">
   <ui:row>
       <ui:cell styleClass="name">
           <ui:fileUpload id="file"/>
       </ui:cell>
   </ui:row>
</ui:form>
...
```

5.2.29. <ui:dateInput>

Form date input field, represents `DateControl`. Default `styleClass="aranea-date"`.

Attributes

Attribute	Required	Description
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.30. <ui:dateInputDisplay>

Form date display field, represents `DateControl`. Default `styleClass="aranea-date-display"`.

Attributes

Has standard `id` and `styleClass` attributes.

5.2.31. <ui:timeInput>

Form time input field, represents `TimeControl`. Default `styleClass="aranea-time"`. HTML <select>s for easy hour/minute selection are rendered too, unless `showTimeSelect` attribute forbids it.

Attributes

Attribute	Required	Description
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`.
showTimeSelect	*no*	Boolean, specifying whether HTML <select>'s should be

Attribute	Required	Description
		rendered for easy hour/minute selection. Default is to render them (`true`).

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.32. <ui:timeInputDisplay>

Form time display field, represents `TimeControl`. Default `styleClass="aranea-time-display"`.

Attributes

Has standard `id` and `styleClass` attributes.

5.2.33. <ui:dateTimeInput>

Form input field for both date and time, represents `DateTimeControl`. It is rendered as input fields for date and time + date picker and time picker (time picker can be switched off by setting `showTimeSelect="false"` if so desired).

Attributes

Attribute	Required	Description
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`.
showTimeSelect	*no*	Boolean, specifying whether HTML <select>'s should be rendered for easy hour/minute selection. Default is to render them (`true`).
dateStyleClass	*no*	*styleClass* for date. Default is "aranea-date".
timeStyleClass	*no*	*styleClass* for time. Default is "aranea-time".

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.34. <ui:dateTimeInputDisplay>

Form display field for both date and time, represents `TimeControl`. Default `styleClass="aranea-datetime-display"`.

Attributes

Has standard `id` and `styleClass` attributes.

5.2.35. <ui:select>

Form dropdown list input field, represents `SelectControl`. Default `styleClass="aranea-select"`, rendered with HTML `<select ...>` tag.

Attributes

Attribute	Required	Description
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`.
size	*no*	Number of select elements visible at once.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.36. <ui:selectDisplay>

Form select display field, represents `SelectControl`. Default `styleClass="aranea-select-display"`, rendered with HTML `<span ...>` tag.

Attributes

Has standard `id` and `styleClass` attributes.

5.2.37. <ui:multiSelect>

Form list input field, represents `MultiSelectControl`. Default `styleClass="aranea-multi-select"`, rendered with HTML `<select multiple="true" ...>` tag.

Attributes

Attribute	Required	Description
size	*no*	Vertical size, number of options displayed at once.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.38. <ui:multiSelectDisplay>

Form multiselect display field, represents `MultiSelectControl`. Default `styleClass="aranea-multi-select-display"`, rendered with HTML `<span ...>` tag.

Attributes

Attribute	Required	Description
separator	*no*	The separator between list items, can be any string or '\n' for newline. Default is ', ').

Has standard `id` and `styleClass` attributes.

5.2.39. <ui:radioSelect>

Form radioselect buttons field, represents `SelectControl`. Default `styleClass="aranea-radioselect"`. It

takes care of rendering all its elements; internally using <ui:radioSelectItemLabel> and <ui:radioSelectItem> tags.

Attributes

Attribute	Required	Description
type	*no*	The way the radio buttons will be rendered - can be either *vertical* or *horizontal*. By default "horizontal".
labelBefore	*no*	Boolean that controls whether label is before or after each radio button, `false` by default.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.40. <ui:radioSelectItem>

Form radio button, represents one item from `SelectControl`. Default `styleClass="aranea-radio"`. It will be rendered with HTML `<input type="radio" ...>` tag.

Attributes

Attribute	Required	Description
value	*no*	The value of this radio button that will be submitted with form if this radio button is selected.
onChangePrecondition	*no*	Precondition for deciding whether registered onchange event should go server side or not. If left unspecified, this is considered to be `true`.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.41. <ui:radioSelectItemLabel>

Form radio button label, represents label of one item from `SelectControl`. It will be rendered with HTML `<span ...>` tag.

Attributes

Attribute	Required	Description
value	*no*	Select item value.
showMandatory	*no*	Indicates whether label for mandatory input is marked with asterisk. Value should be `true` or `false`, default is `true`.
showColon	*no*	Indicates whether colon is shown between the label and value. Default is `true`

Also has standard `id` and `styleClass` attributes.

5.2.42. <ui:checkboxMultiSelect>

Form multiselect checkbox field, represents `MultiSelectControl`. It takes care of rendering all its elements; internally using <ui:checkboxMultiSelectItemLabel> and <ui:checkboxMultiSelectItem> tags.

Attributes

Attribute	Required	Description
type	*no*	The way the checkboxes will be rendered - can be either *vertical* or *horizontal*. Default is *horizontal*.
labelBefore	*no*	Boolean that controls whether label is before or after each cehckbox, `false` by default.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.43. <ui:checkboxMultiSelectItem>

Form radio button, represents one item from `MultiSelectControl`. Default `styleClass="aranea-multi-checkbox"`. It will be rendered with HTML `<input type="checkbox" ...>` tag.

Attributes

Attribute	Required	Description
value	*no*	The value of this checkbox that will be submitted with form if this checkbox is selected.

Also see Section 5.2.1, "Common attributes for all form element rendering tags.".

5.2.44. <ui:checkboxMultiSelectItemLabel>

Form checkbox label, represents label of one item from `MultiSelectControl`. It will be rendered with HTML `<span ...>` tag.

Attributes

Attribute	Required	Description
value	*no*	Select item value.
showMandatory	*no*	Indicates whether label for mandatory input is marked with asterisk. Value should be `true` or `false`, default is `true`.
showColon	*no*	Indicates whether colon is shown between the label and value. Default is `true`

Also has standard `id` and `styleClass` attributes.

5.2.45. <ui:conditionalDisplay>

Depending whether form element boolean value is *true* or *false* display one or other content, represents `DisplayControl`. `<ui:conditionFalse>` and `<ui:conditionFalse>` tags must be used inside this tag to define

alternative contents. This tag itself is not rendered.

Attributes

Has standard `id` attribute.

5.2.46. <ui:conditionFalse>

The content of this tag will be displayed when form element of surrounding `<ui:conditionalDisplay>` was `false`. Tag has no attributes.

5.2.47. <ui:conditionTrue>

The content of this tag will be displayed when form element of surrounding `<ui:conditionalDisplay>` was `true`. Tag has no attributes.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:form id="someForm">
  <ui:conditionalDisplay id="isActive">
    <ui:conditionTrue>
      <img src="red_flag.png"/>
    </ui:conditionTrue>
    <ui:conditionFalse>
      <img src="green_flag.png"/>
    </ui:conditionFalse>
  </ui:conditionalDisplay>
</ui:form>
```

5.2.48. <ui:listDisplay>

Display form element value as list of strings, represents `DisplayControl` and requires that element value would be of type `Collection`.

Attributes

Attribute	Required	Description
separator	*no*	The separator between list items, can be any string and "\n", meaning a newline (default is "\n").

Also has standard `id` and `styleClass` attributes.

5.2.49. <ui:automaticFormElement>

Sometimes the type of `FormElement` is not known for sure when writing JSP (it could be `textInput`, `floatInput`, `select`, ...). For that purpose, `FormElement` that has some known identifier can be dynamically associated with some JSP tag in Java code and then rendered with `<ui:automaticFormElement>` tag which uses associated tag to render `FormElement`.

Attributes

See Section 5.2.1, “Common attributes for all form element rendering tags.”.

Examples

In Java code, setting tag that should rendering element is done by setting `FormElement` property.

```
element.setProperty(FormElementViewSelector.FORM_ELEMENT_VIEW_SELECTOR_PROPERTY, new FormElementViewSe
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:formElement id="someForm">
  <ui:cell>
    <ui:automaticFormElement/>
  </ui:cell>
</ui:formElement>
```

5.3. Form Lists

A common need in handling data is allowing a user to list of data, where the number of rows is not known beforehand (a typical example being user inputting one to many addresses). Aranea supports such a use case by providing a special type of `FormElement` that deals an arbitrary amount of subforms. This element is called `FormListWidget` and it can be used both on its own or as a subelement just like a `FormWidget`. An example of a form list is shown on Figure 5.2, “Insert your name display”.

Figure 5.2. Insert your name display

5.3.1. FormListWidget

Unlike usual forms, form lists are "lazy", from the point that they are tied to a model and update themselves according to it. To create a form list widget we pass it a model and a handler:

```
...
  public void init() throws Exception {
    private FormListWidget personFormList;
    ...
    Map persons = lookupMyService().getPersons();

    personFormList = new BeanFormListWidget(
        new PersonFormRowHandler(),
        new MapFormListModel(persons),
        Person.class);

    addWidget("personFormList", personFormList);
  }
...
```

Note here that we have tied the form list to the model that uses a `Map` as the underlying storage. When we update that map, the form list will also be updated. Note also that the form list widget is associated with the `Person` bean class, which can be used to manipulate the beans under the model.

However this code doesn't yet tell us much. The bulk of the custom logic of the form lists is hidden in the `PersonFormRowHandler` class. Let's inspect it step by step.

Every form row handler must implement the `FormRowHandler` interface. In our case we choose to extend `ValidOnlyIndividualFormRowHandler`, which processes only valid form rows and allows to process them one by one, not all at once:

```
class PersonFormRowHandler
  extends ValidOnlyIndividualFormRowHandler {
  ...
}
```

The first method we have to implement is `getRowKey`. It is used by the form list widget to identify the row among the others. Since typically the row is just a bean we can identify it using its identifier (either a natural one or artificial, as long as its unique in this context):

```
...
public Object getRowKey(Object rowData) {
  return ((Person) rowData).getId();
}
...
```

The next method is called `initAddForm` and it will create a form used to add new rows to the form list:

```
...
public void initAddForm(FormWidget addForm) throws Exception {
  addForm.addBeanElement("name", "#First name", new TextControl(), true);
  addForm.addBeanElement("surname", "#Last name", new TextControl(),  true);
  addForm.addBeanElement("phone", "#Phone no", new TextControl(), false);

  FormListUtil.addAddButtonToAddForm("#", formList, addForm);
}
...
```

The bulk of the logic is just adding the fields to the add form. But we also use the `FormListUtil` to add a button "Add" to the form, that will take care of the actual adding a new row (or at least calling the form row handler to do that). `FormListUtil` contains a lot of helpful methods for manipulating form lists and more on it can be found in Section 5.3.2, “FormListUtil”. The next step would be to handle the user clicking the add button and add a new row to the model. Since we process only valid rows the method will be named `addValidRow`:

```
...
public void addValidRow(FormWidget addForm) throws Exception {
  Person person = (Person) (((BeanFormWidget)addForm).writeToBean(new Person()));
  //We want to save changes immediately
  person = lookupPersonService.addPerson(person);
  data.add(person.getId(), person);
}
...
```

Note that although we save the changes here immediately, form lists also support deferring this until some later point as described in Section 5.3.5, “In Memory Form List”. Now that we have added a row to the model we will also have to initialize a form for that using `initFormRow` method:

```
...
public void initFormRow(FormRow formRow, Object rowData) throws Exception {
```

```
  // Set initial status of list rows to closed - they cannot be edited before opened.
  formRow.close();

  BeanFormWidget form = (BeanFormWidget)formRow.getForm();

  form.addBeanElement("name", "#First name", new TextControl(), true);
  form.addBeanElement("surname", "#Last name", new TextControl(),  true);
  form.addBeanElement("phone", "#Phone no", new TextControl(), false);

  FormListUtil.addEditSaveButtonToRowForm("#", formList, form, getRowKey(rowData));
  FormListUtil.addDeleteButtonToRowForm("#", formList, form, getRowKey(rowData));

  form.readFromBean(rowData);
}
...
```

Note that most of the fields are same for add form and edit forms, so in a real setup we could easily have added a method `addCommonFields(FormWidget)` that would add those fields to any given form (it is actually a very common idiom to do that). Finally we have to handle the saving of row form:

```
...
public void saveValidRow(FormRow formRow) throws Exception {
  BeanFormWidget form = (BeanFormWidget) formRow.getForm();
  Person person = (Person) form.writeToBean(data.get(formRow.getKey()));

  lookupPersonService().save(rowData);
  data.put(person.getId(), person);
}
...
```

And the last one left is deletion:

```
...
public void deleteRow(Object key) throws Exception {
  Long id = (Long) key;
  lookupPersonService().remove(id);
  data.remove(id);
}
...
```

5.3.2. FormListUtil

`FormListUtil` provides a couple of methods that help to handle form maps passed to some of the handler methods. However of main interest are the methods that add various buttons with ready logic to the add forms and row forms.

Method	Description
`addSaveButtonToRowForm()`	Button that will save the current row.
`addDeleteButtonToRowForm()`	Button that will delete the current row.
`addOpenCloseButtonToRowForm()`	Button that will open or close the current row for editing (it inverts the current state).
`addEditSaveButtonToRowForm()`	Button that will open/close the row for editing, however will also save it after editing is finished and the row is closed.
`addAddButtonToAddForm()`	Button that will add a new row, should be added to the addition form.

5.3.3. Form Row Handlers

Since row form handler interface supports bulk saving/adding/deleting of row forms it is comfortable to use one of the base classes that will do some of the work for you.

Class	Description
`DefaultFormRowHandler`	Implements all of the menthods and default handling of opening/closing rows.
`ValidOnlyFormRowHandler`	Checks that all of the added/saved rows are valid.
`IndividualFormRowHandler`	Supports one by one processing of row saving and deleting.
`ValidOnlyIndividualFormRowHandler`	Supports one by one processing of row saving and deleting. Checks that all of the added/saved rows are valid.

Note that row handlers also have an `openOrCloseRow` method that may be overridden if one wants more than just inverting the row state on user action.

5.3.4. Models

5.3.5. In Memory Form List

Often it is the case that we do not want to save the changes in the form list to the database until the user presses the "Save" button. For such a use case we provide `InMemoryFormListHelper`. To use the helper we first need to initialize the form list to use the helper model:

```
...
private BeanFormListWidget personFormList;
private InMemoryFormListHelper inMemoryHelper;

public void init() throws Exception {
  private FormListWidget personFormList;
  ...
  Map persons = lookupMyService().getPersons();

  personFormList = new BeanFormListWidget(new PersonFormRowHandler(), Person.class);
  inMemoryHelper = new InMemoryFormListHelper(
    personFormList,
    lookupPersonService().getSomePersonList());

  addWidget("personFormList", personFormList);
}
...
```

Now we just have to add/save/delete the row to/from the helper:

```
...
public void saveValidRow(FormRow editableRow) throws Exception {
  ...
  inMemoryHelper.update(editableRow.getKey(), rowData);
}

public void deleteRow(Object key) throws Exception {
  ...
  inMemoryHelper.delete(key);
}
```

```
public void addValidRow(FormWidget addForm) throws Exception {
  ...
  inMemoryHelper.add(rowData);
}
...
```

And when the user presses "Save" we can just process the changes:

```
...
protected void handleEventSave() {
  lookupPersonService.addAll(inMemoryHelper.getAdded().values());
  lookupPersonService.saveAll(inMemoryHelper.getUpdated().values());
  lookupPersonService.deleteAll(inMemoryHelper.getDeleted());
}
...
```

5.4. Form Lists JSP Tags

5.4.1. <ui:formList>

Formlist is a list of forms, an editable list. This tag specifies editable list context for its inner tags.

Attributes

Attribute	Required	Description
id	*no*	Id of editable list. When not specified, attempt is made to construct it from existing list context—it this does not succeed, tag fails.

Variables

Variable	Description	Type
formList	Editable list view model.	`FormListWidget.ViewModel`
formListId	Editable list id.	`String`

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:list id="list">
    <ui:formList>
        ...
    </ui:formList>
</ui:list>
```

5.4.2. <ui:formListRows>

Iterating tag that gives access to each row and row form on the editable list current page. The editable row is accessible as "editableRow" variable.

Attributes

Attribute	Required	Description
var	*no*	Name of variable that represents individual row (by default "row").

Variables

Variable	Description	Type
formRow	Current editable list row view model.	`FormRow.ViewModel`
row (unless changed with *var* attribute).	Object held in current row.	`Object`

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:list id="list">
    <ui:formList>
        <ui:formListRows>
            ...
        </ui:formListRows>
    </ui:formList>
</ui:list>
```

5.4.3. <ui:formListAddForm>

Allows for adding new forms (rows) to editable list.

Attributes

Attribute	Required	Description
id	*no*	Editable list id. Searched from context, if not specified.

Variables

Variable	Description	Type
form	View model of form.	`FormWidget.ViewModel`
formId	Id of form.	`String`
formFullId	Full id of form.	`String`
formScopedFullId	Full scoped id of form.	`String`

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:formListAddForm>
    <ui:row>
        <ui:cell>
            <ui:textInput id="name"/>
        </ui:cell>

        <ui:cell>
            <ui:textInput id="surname"/>
        </ui:cell>

        <ui:cell>
            <ui:textInput id="phone"/>
        </ui:cell>

        <ui:cell>
            <ui:dateInput id="birthdate"/>
        </ui:cell>
    </ui:row>
</ui:formListAddForm>
```

Chapter 6. Lists and Query Browsing

6.1. Introduction

A common task in web applications is displaying tables. This is a simple task if the tables are small and do not contain a lot of rows. The whole table can be visible at once and there is no need to split the data into pages as well as provide an option to order the table by new column or display a search form that can be used to filter the data. In such cases where these features must be available, Aranea provides widget `org.araneaframework.uilib.list.ListWidget` and some support classes. In this chapter we will introduce these widgets and supporting API and show how to use and extend them.

`ListWidget` is used to define the list columns, the way the list can be filtered and ordered, how many items are shown per pages etc. `ListWidget` uses `org.araneaframework.uilib.list.dataprovider.ListDataProvider` to get the current list items range that also match with the current filter and order conditions. `ListDataProvider` can cache whole table and provide the `ListWidget` with the appropriate item range or fetch only the specific item range from database. Aranea provides two implementations:

`org.araneaframework.uilib.list.dataprovider.MemoryBasedListDataProvider`
: This is the memory-based solution that must be provided with the whole data at once. It does *filtering* , *ordering* and *paging* memory-based. The data source is not restricted here. This is a very fast and easy-to-use solution for tables with few (typically less than 100) rows.

`org.araneaframework.uilib.list.dataprovider.BackendListDataProvider`
: This is the database solution that will cache only the current item range and executes a new database query each time a *filtering* , *ordering* or *paging* conditions change. This is a powerful solution for tables with more than 500 rows.

6.2. Lists API

6.2.1. A Typical List

A typical list will be created used like this:

```
...
private BeanListWidget myList;
...
protected void init() {
  ...
  myList = new BeanListWidget(MyModel.class);

  myList.setOrderableByDefault(true);
  myList.addField("name", "#Name").like();
  myList.addField("surname", "#Surname").like();
  myList.addField("phone", "#Phone no").like();
  myList.addField("birthdate", "#Birthdate").range();
  ...
  myList.setInitialOrder("name", true);
  myList.setListDataProvider(new MyListDataProvider());
  addWidget("myList", myList);
  ...
}
...
```

Note that here we basically do following things:

Create the list
: The line `new BeanListWidget(MyModel.class)` creates a new list widget that is associated with the *JavaBean* model class `MyModel` .

Make fields orderable
: The line `myList.setOrderableByDefault(true)` configures the following fields as orderable.

Add list fields
: The line `myList.addField("name", "#Name").like()` adds a field associated with the JavaBean property "name" (this is also the identifier of the field), with a label "Name", makes the field filterable by `Like` filter.

Set the initial list order
: The line `myList.setInitialOrder("name", true)` sets the list to be ordered by field "name" by default.

Set the list data provider
: The line `myList.setListDataProvider(new MyListDataProvider())` sets the data provider for the list.

Register the list
: The line `addWidget("myList", myList)` initializes and registers the list allowing it to function.

Now that we have created the list we show how to build a simple data provider. The following example code should be in the same widget as the previous:

```
...
private class MyMemoryBasedListDataProvider extends MemoryBasedListDataProvider {
  protected MyMemoryBasedListDataProvider() {
    super(MyModel.class);
  }
  public List loadData() throws Exception {
    return lookupMyService().findAllMyModel();
  }
}
...
```

The line `super(MyModel.class)` associated this `MemoryBasedListDataProvider` with the *JavaBean* model class `MyModel`. The method `List loadData()` implements loading all table rows and returning it as a `List` object.

Later, we will also discover using `org.araneaframework.uilib.list.dataprovider.BackendListDataProvider`.

6.2.2. Fields

As the list may be displayed as a table, it is basically an ordered collection of items. In the previous example, we defined a list of `MyModel.class` typed items that have fields 'name', 'surname', 'phone' and 'birthdate'. By listing of `MyModel.class`, we also told `ListWidget` the corresponding field types, e.g `String.class`, `String.class`, `Long.class` and `Date.class`. In fact this feature of reflection is the only distinction between the `ListWidget` and `BeanListWidget`.

Each list field have its own *Id*, *label* and *type*. The labels are used to automatically create a corresponding title row above the displayed table. The types are used to describe how to order or filter the whole data using this field. E.g `String.class` is treated differently than other types, because usually one would prefer to order by this field ignoring the case. Both the labels and types are also used to build a corresponding search form - an automatically built `FormWidget` - for the list.

If we would like to add some list fields that are not `MyModel.class` fields, we can pass it's type to the

`ListWidget` like following:

```
myList.addField("income", "#Income", BigDecimal.class);
```

Here the `myList` could be just a `ListWidget` rather than a `BeanListWidget`.

When adding a list field, we can also provide this field-related ordering and filtering information.

6.2.3. Ordering

Each list field can be orderable or not. We already discovered `ListWidget`'s method `setOrderableByDefault(boolean)` that switch whether to configure fields that are added afterwards orderable or not. This method can be used several times in the list configuration.

Another way is to set each field individually orderable or not when they are added to the list. In such case add additional boolean argument to the `addField()` method such as:

```
myList.addField("phone", "#Phone no", false);
```

Notice the `false` as third parameter. `true` means that the list can be ordered by this field and `false` means the opposite. By not providing this parameter, simply the last value is used which has been set by `setOrderableByDefault(boolean)` method.

In addition, we already used method `setInitialOrder(String, boolean)`. It sets a specified field (the first argument) to be ordered by default. `true` as the second argument tells the ordering should be ascending, `false` would mean descending. By not providing this information, the list is displayed in the original order.

6.2.4. Filtering

Filtering means that we only display a certain list items. The list can be filtered using its fields and data provided by the search from of this list.

For this, we must provide the `ListWidget` with the corresponding `org.araneaframework.uilib.list.structure.ListFilters` and `FormElements`. As the form elements are dummy "boxes" that hold search data, each `ListFilter` is related to a certain filter test, e.g. equality, greater than comparison etc. Each `ListFilter` also knows what information it must consider. In general, one list field is compared against a value provided by the search form. It's also assumed that a blank search field means that this particular `ListWidget` is currently disabled.

Fortunately, in most cases it's unnecessary to add these search fields manually. Instead, if one is adding a list field, he or she can assign both the `ListFilter` and `FormElement` for this field very simply:

```
myList.addField("address", "#Address").like();
```

Here we simply add an 'Address' field providing it with label and telling there's should be a *Like filter* for this field. By this, we automatically add a `TextControl` into the search form. By filling it with value 'Paris', we will see only rows which 'Address' field contain 'Paris', 'paris', 'PARIS' etc.

To describe, how this works, we show a longer version of the previous code:

```
myList.addField("address", "#Address");
myList.getFilterHelper().like("address");
```

So there's a special class `org.araneaframework.uilib.list.structure.filter.FilterHelper` that is used to add list filters. All `ListWidget.addField()` methods just return a little different version of this helper class, called a `FieldFilterHelper`. It's methods do not need a field Id and thus make one not to repeat the same field Id for each filter. In general, the shorter usage is recommended of course. However some filters are more

complicated and may be related to more than one list field. For those, one must use the `FilterHelper` instead.

By default all filters that deal with the Strings are case insensitive. To configure some filters to be different, use the following:

```
myList.addField("country", "#Country").setIgnoreCase(false).like();
myList.addField("city", "#City").like();
myList.addField("address", "#Address").setIgnoreCase(true).like();
```

This can be explained following: Before adding a Like filter for the 'country' field, we switched to the case sensitive mode. And before adding a filter for the 'address' field, we switched to the case insensitive mode. Thus the city's filter is case sensitive as the country's but the address' filter does ignore the case.

This state is held by the `FilterHelper` and can be modified either by calling a method of the `FilterHelper` or the `FieldFilterHelper`. In such way, the following parameters can be set:

Case sensitivity
: By using `setIgnoreCase(boolean)` one assigns new filters to ignore case (default) or not. This applies to filters that use String comparison.

Strict/non-strict
: By using `setStrict(boolean)` one assigns new filters to disallow equality or not. By default equality is not allowed (strict). This applies to filters such as GreaterThan, LowerThan, Range etc.

Now, let's show which filters we have got:

`FilterHelper` method	**`ListFilter` class**	**Description**
`eq()`	`EqualFilter`	Tests if the value of a certain list field is equal to the value of a certain search form field. The filter is disabled if the search field is blank.
`eqConst()`	`EqualFilter`	Tests if the value of a certain list field is equal to a certain constant. This filter is always enabled.
`gt(), lt()`	`GreaterThanFilter`, `LowerThanFilter`	Tests if the value of a certain list field is greater than (lower than) the value of a certain search form field. This filter is disabled if the search field is blank.
`gtConst()`, `ltConst()`	`GreaterThanFilter`, `LowerThanFilter`	Tests if the value of a certain list field is greater than (lower than) a certain constant. This filter is always enabled.
`like()`	`LikeFilter`	Tests if the pattern in a certain search form field matches with the value of a certain list field. This corresponds to the `LIKE` expression in SQL with some modifications. By default, it takes '%' and '*' symbols as any-string wildcards and '_', '.' and '?' as any-symbol wildcards. In addition, the pattern does not have to match with the whole string ('%' is automatically added before and after the pattern string). The wildcards and their automatic adding is configured by the `org.araneaframework.uilib.list.util.like.LikeConfigurati` which is found from the Aranea `org.araneaframework.uilib.ConfigurationContext`.

`FilterHelper` method	`ListFilter` class	Description
		This filter is identical in memory-based and database backend usage. This filter is disabled if the search field is blank.
`likeConst()`	`LikeFilter`	Tests if a certain constant pattern matches with the value of a certain list field. This filter is always enabled.
`isNull()`, `notNull()`	`NullFilter`	Tests if the value of a certain list field is null (is not null) if the value of a certain search form field equals to a specified value.
`isNullConst()`, `notNullConst()`	`NullFilter`	Tests if the value of a certain list field is null (is not null). This filter is always enabled.
`range()`	`RangeFilter`	Tests if the value of a certain list field is between two values of certain search form fields. The filter is identical to the greater than or lower than filter in case of one of the search fields is blank. This filter is disabled if both search fields are blank.
`fieldRangeInValueRange()`, `valueRangeInFieldRange()`, `overlapRange()`	`RangeFilter`	Tests if two values of certain list fields are between two values of certain search form fields, vice-versa or do they have a non-empty intersection. This filter is disabled if both search fields are blank.
`sqlFunction()`	`SqlFunctionFilter`	Tests if the value returned from a certain SQL function is equal (or is greater than or is lower than) to the value of a certain list field, search form field or a constant. The arguments of the SQL function can also be chosen among the values of list fields, search form fields and constants. This filter cannot be used memory-based. This filter is always enabled.

By default the `FormElements` added into the search form have the same identifiers as the list fields. Therefore there can be only one search field per list field. If one would like to override the used Id for `FormElement`, any filter could be added like following:

```
        myList.addField("country", "#Country").like();
        myList.getFilterHelper().like("country", "anotherCountry");
```

The first line adds a list field 'country' and a *Like filter* associated with it as well as a new `FormElement` with Id of 'country'. The second line adds another *Like filter* associated to the list field 'country' and a new `FormElement` with Id of 'anotherCountry'.

By adding a filter, the corresponding `FormElement` is automatically created and added to the search form of the list. Now we cover the properties of the few `FormElement` describing their default values and showing how to customize them:

Property	Default value	Customizing
Id	Same as the list field Id.	Call

<table>
<tr><th>Property</th><th>Default value</th><th>Customizing</th></tr>
<tr><td></td><td></td><td><code>addField(...).<filter>("myCustomId");</code></td></tr>
<tr><td>Label</td><td>Same as the label of the associated list field.</td><td>Call <code>addField(...).useCustomLabel("myCustomLa</code></td></tr>
<tr><td>Control</td><td>Is selected considering the type of the associated list field:
<table><tr><th>Type</th><th>Control</th></tr><tr><td><code>java.lang.String</code></td><td><code>TextControl</code></td></tr><tr><td><code>java.math.BigInteger</code>, <code>java.lang.Long</code>, <code>java.lang.Integer</code>, <code>java.lang.Short</code>, <code>java.lang.Byte</code></td><td><code>NumberControl</code></td></tr><tr><td><code>java.math.BigDecimal</code>, <code>java.lang.Double</code>, <code>java.lang.Float</code></td><td><code>FloatControl</code></td></tr><tr><td>Other subclasses of <code>java.lang.Number</code></td><td><code>FloatControl</code></td></tr><tr><td><code>java.util.Date</code>, <code>java.sql.Date</code></td><td><code>DateControl</code></td></tr><tr><td><code>java.sql.Time</code></td><td><code>TimeControl</code></td></tr><tr><td><code>java.sql.Timestamp</code></td><td><code>DateTimeControl</code></td></tr><tr><td><code>java.lang.Boolean</code></td><td><code>CheckboxControl</code></td></tr><tr><td>All others</td><td><code>TextControl</code></td></tr></table></td><td>Call <code>addField(...).xxx(new MyCustomControl());</code></td></tr>
<tr><td>Data</td><td>Corresponds to the type of the associated list field.</td><td>Call <code>addField(...).useFieldType(MyType.class)</code></td></tr>
<tr><td>Initial value</td><td>Always <code>null</code> to disable the filter by default.</td><td>After adding the field and the filter call <code>myList.getForm().setValueByFullName("fie customInitialValue);</code> or add a custom <code>Formelement</code>.</td></tr>
<tr><td>Mandatory</td><td>Always <code>false</code> as all search conditions are optional.</td><td>After adding the field and the filter call <code>myList.getForm().getElementByFullName("f</code> or add a custom <code>Formelement</code>.</td></tr>
<tr><td>FormElement</td><td>See all properties above.</td><td>To use a custom <code>Formelement</code>, call <code>addField(...).xxx(new MyCustomFormElement(...));</code>. To disable adding it at all, call <code>addField(...)._xxx();</code> (notice</td></tr>
</table>

Property	Default value	Customizing
		the underscore).

The `xxx` marks any filter adding method. As one can count, there are 6 overridden methods for each list filter: 2 versions for providing a custom Id or not and 3 versions for providing a custom `FormElement`, `Control` or neither of them. In addition there are methods that start with an _ for disabling adding a form element. Using the `FilterHelper` instead of `FieldFilterHelper` is analogous except all filter adding methods take the list field Id as the first argument in addition.

It's import to notice that `xxxxConst` methods do not create a form element because they are independent of the search form at all - they are constant. However they can actually take a value Id for the defined constants as well. These Ids can be used later to convert specific values when creating a database query. Of course non-constant filters have the same Ids but just use them mainly to get values from the search form. `xxxxConst` filters have 2 overridden add methods depending on whether the custom value Id is provided or not. By default it's the same as the field Id.

6.2.5. Backend Data Provider

Now that we have demonstrated defining lists and also creating `MemoryBasedListDataProvider`, we will discover using `BackendListDataProvider`. The following example code should be in the same widget as constructing of the related `ListWidget`.

```
...
private class MyBackendListDataProvider extends BackendListDataProvider {
  public MyBackendListDataProvider() {
    super(true);
  }
  protected ListItemsData getItemRange(ListQuery query) throws Exception {
    return lookupMyService().findMyModel(query);
  }
}
...
```

The line `super(true)` constructs `BackendListDataProvider` with cache enabled (only used when there are no change in *query*). Notice that there is no association with any *JavaBean* class here. The method `ListItemsData getItemRange(ListQuery query)` implements loading current item range according to the range indexes and filtering and ordering conditions. `org.araneaframework.backend.list.mode.ListQuery` and `org.araneaframework.backend.list.mode.ListItemsData` may be thought as being input and output of each list data query.

`ListQuery` is a simple JavaBean that holds the following properties:

List item range indexes
: This is 0-based start index and items count (`Long` objects) that define the range. By default, lists are shown by pages. Although all items can be shown at once also. Then the start index is zero and items count is omitted.

Filter expression
: This could be thought as an abstraction of previously defined bunch of `ListFilter` objects that are provided with the bound values from *filter form* . This is a basis of constructing a SQL `WHERE` clause later.

Order expression
: This could be thought as an abstraction of previously defined ordered columns that are provided with the current ordering condition. This is a basis of constructing a SQL `ORDER BY` clause later.

Generally, this whole object is just passed to `org.araneaframework.backend.list.helper.ListSqlHelper` class that is used to generate SQL statements and fetching the results from database. Latter is hold in `ListItemsData` object which is a simple JavaBean that holds the following properties:

List items range
: Model objects that are the result of the *query* .

Total count
: Total count (`Long` object) of the list. This is important information for navigating through the whole list. Notice that this depends only on filtering conditions.

Notice that `BackendListDataProvider` actually do not depend on using databases. It just provides a simple *query* object and expects a simple result to be returned. Thus, you have the power to use it as you like. At the same, Aranea provides a very useful class `org.araneaframework.backend.list.helper.ListSqlHelper` that generate SQL statements and fetches the results from database. We strongly recommend it together with its subclasses that support different database systems. Currently `Oracle` (`OracleListSqlHelper`) and `HSQL` (`HsqlListSqlHelper`) databases are supported.

The following example discovers the simplest usage of `ListSqlHelper`. The following code should be in a service class instead of previously discovered Widget:

```
public class MyService {
  ...
  private DataSource dataSource;
  ...
  public ListItemsData findMyModel(ListQuery query) {
    ListSqlHelper helper = new OracleListSqlHelper(this.dataSource, query);

    helper.addMapping("name", "NAME");
    helper.addMapping("surname", "SURNAME");
    helper.addMapping("phone", "PHONE_NO");

    helper.setSimpleSqlQuery("PERSON");
    return helper.execute(MyModel.class);
  }
  ...
}
```

Method `ListItemsData findMyModel(ListQuery query)` does the following:

Constructs and initializes the helper
: The line `ListSqlHelper helper = new OracleListSqlHelper(this.dataSource, query)` constructs `OracleListSqlHelper` - an Oracle specific `ListSqlHelper` - and passes it the `DataSource` and `ListQuery` data.

Adds column mappings
: The line `helper.addMapping("name", "NAME")` defines that identifier of column "name" will be converted into `"NAME"` when used in an SQL statement. There may be lot of differnece between JavaBean properties names and database fields names. The same database identifier (`"NAME"`) is used when fetching data from `ResultSet` by default. This could also have another identifier set by providing it as the third argument.

Provides the helper with a simple SQL query
: The line `helper.setSimpleSqlQuery("PERSON")` sets the whole SQL query with parameters using only the given database table name. Filtering and ordering is added automatically according to the `ListQuery` data.

Executes the query and retrieve the data
: The line `return helper.execute(MyModel.class)` executes and retrieves data of both total count and

items range queries. The `ResultSet` is read using the default `BeanResultReader` .

The following example discovers the custom usage of `ListSqlHelper`.

```
public class MyService {
  ...
  private DataSource dataSource;
  ...
  public ListItemsData findMyModel(ListQuery query) {
    ListSqlHelper helper = new OracleListSqlHelper(this.dataSource, query);

    helper.addMapping("name", "NAME");
    helper.addMapping("surname", "SURNAME");
    helper.addMapping("phone", "PHONE_NO");

    StringBuffer s = new StringBuffer();
    s.append("SELECT ");
    s.append(helper.getDatabaseFields());
    s.append(" FROM PERSONS");
    s.append(helper.getDatabaseFilterWith(" WHERE ", ""));
    s.append(helper.getDatabaseOrderWith(" ORDER BY ", ""));

    helper.setSqlQuery(s.toString());
    helper.addStatementParams(helper.getDatabaseFilterParams());
    helper.addStatementParams(helper.getDatabaseOrderParams());

    return helper.execute(MyModel.class);
  }
  ...
}
```

Method `ListItemsData findMyModel(ListQuery query)` does the following:

Constructs and initializes the helper

The line `ListSqlHelper helper = new OracleListSqlHelper(this.dataSource, query)` constructs `OracleListSqlHelper` - an Oracle specific `ListSqlHelper` - and passes it the `DataSource` and `ListQuery` data.

Adds column mappings

The line `helper.addMapping("name", "NAME")` defines that identifier of column "name" will be converted into `"NAME"` when used in an SQL statement. There may be lot of differnece between JavaBean properties names and database fields names. The same database identifier (`"NAME"`) is used when fetching data from `ResultSet` by default. This could also have another identifier set by providing it as the third argument.

Gets SQL substrings from the helper

The line `helper.getDatabaseFields()` returns just the comma-separated list of database column identifiers that were just defined in the mapping. This does not depend on the original set of list columns at all. The line `helper.getDatabaseFilterWith(" WHERE ", "")` returns the `WHERE` clause body with the provided prefix and suffix. It returns an empty string if there is no filter condition currently set (it does not mean there are no filters defined). Notice that we only deal with SQL strings here. As `ListSqlHelper` uses `PreparedStatement` objects to execute queries, there must be provided statement parameters in addition to the SQL string. This generally provides better performance of executing similar queries.

Constructs SQL query string

`StringBuffer` is used to construct the whole SQL query string. Notice that the helper does not construct it totally by itself. This lends user more power for complex queries. It is very important that the constructed query is for getting all rows that match with the current filter and order conditions, but not the range conditions. `ListSqlHelper` always executes two queries: one for getting the items count and another for getting the items range. Generally, both of these can be easily constructed from this one provided query. This implementation depends on the database system and therefore the concrete `ListSqlHelper` subclass.

Gets SQL parameters from the helper

The line `helper.getDatabaseFilterParams()` returns SQL parameters of `WHERE` clause or empty list if there are none.

Provides the helper with the SQL query

The line `helper.setSqlQuery(...)` sets the SQL string and the line `helper.addStatementParams(...)` adds the query parameters (`ListSqlHelper` uses `PreparedStatement` s). Of course, the order of parameters must match with the SQL string.

Executes the query and retrieve the data

The line `return helper.execute(MyModel.class)` executes and retrieves data of both total count and items range queries. The `ResultSet` is read using the default `BeanResultReader`.

ListSqlHelper mappings and converters

All Aranea List filters that are propagated with values from the filter form construct an expression chain. This chain is built each time any condition is changed. E.g if one is searching for persons whose birthday is between July 6th, 1950 and Sept 2nd, 1990 then there's one value 'Birthday' and two values 'July 6th, 1950' and 'Sept 2nd, 1990' which have 'Birthday_start' and 'Birthday_end' as names. Ordering the list is done the same. When retrieving data from database all these information must be considered to build an appropriate query. Therefore all these variables must be mapped to database fields. When reading the query results Bean fields must be mapped to `ResultSet` columns. In general, these Bean fields match exactly with the variables. But considering more specific cases, they are not assumed to be the same.

The following list covers the terms that are used when configuring `ListSqlHelper`:

Expression variable

Variables can be thought as list columns that are used in filtering and ordering the list. Variable's name can be for example 'Birthday'.

Expression value

Values are the temporary information in the list filtering. '1980-08-21' is a value. 'Birthday_start' is a name of a value. In simple cases each variable matches with one value. In case of the range filter two different values (start and end of the range) are used. Also one value can be used together with two or more variables. A value identifier is used for optional converting before using it in a query. This is done by adding a `Converter` object to `ListSqlHelper` . E.g. booleans have to be converted into numeric (0 or 1) values.

Database field

Database field can be for example 'age' or 'company.name' as well as 'MAX(price)' or '(SELECT(COUNT(*) FROM document WHERE userId = user.id)' (an inner SELECT) - any expression that is part of a SQL string.

Database field alias

Database field alias is for example 'name', 'total_price' etc. It's just an identifier not a whole expression. In `ListSqlHelper` one can assign an alias for each database field or have it automatically generated. The result of a query is a table - a `ResultSet` - which columns have the same names as the aliases in the query. An alias can also be used in a custom filter condition (WHERE clause) to identify the same database field or expression that was added in the SELECT clause.

ResultSet column

`ResultSet` is a table that is retrieved from database as a result of executing the query. Generally all the columns that where added in a SELECT can be retrieved from the ResultSet by their aliases. But the

ResultSet can also contain additional database-specific data such as `rownum` etc. ResultSet column can also be assigned a `Converter` . E.g. to convert numeric value into a Boolean. Here is the Converter used reversely.

Bean field
: Bean field (or property) is for example 'phone' or 'address.city' that is an instance field of a Java object. During the ResultSet reading new instance of the given type is created and its fields are propagated with values. E.g. 'phone' is assigned by calling setPhone(String phone), 'address.city' is assigned by calling setAddress(new Address()) (if getAddress() == null) and getAddress().setCity(String city).

`ListSqlHelper` methods for configuring mappings:

Method	Purpose
`addMapping(String variable, String dbField, String dbAlias, String beanField)`	Adds mapping between expression variable, database field and Bean field. Database field alias and Bean field are assigned manually.
`addMapping(String variable, String dbField, String dbAlias)`	Adds mapping between expression variable, database field and Bean field. Database field alias is assigned manually. Bean field is assigned the same as the variable.
`addMapping(String variable, String dbField)`	Adds mapping between expression variable, database field and Bean field. Database field alias is generated automatically. Bean field is assigned the same as the variable. (mostly used)
`addDatabaseFieldMapping(String variable, String dbField, String dbAlias)`	Adds mapping between expression variable and database field. Database field alias is assigned manually.
`addDatabaseFieldMapping(String variable, String dbField)`	Adds mapping between expression variable and database field. Database field alias is generated automatically.
`addResultSetMapping(String rsColumn, String beanField)`	Adds mapping between `ResultSet` column (matches with database field alias) and Bean field. (rarely used)

`ListSqlHelper` methods for configuring converters:

Method	Purpose
`addDatabaseFieldConverter(String value, Converter converter)`	Adds converter for expression value.
`addResultSetDeconverterForBeanField(String beanField, Converter converter)`	Adds deconverter for `ResultSet` column by Bean field that is mapped with that Column.
`addResultSetDeconverterForColumn(String rsColumn, Converter converter)`	Adds deconverter for `ResultSet` column.

6.3. List JSP Tags

6.3.1. <ui:list>

Starts a list context. List view model, list sequence view model and list id are made accessible to inner tags as EL variables.

Attributes

Attribute	Required	Description
id	*no*	List widget id.
varSequence	*no*	Name of variable that represents list sequence info (by default *listSequence*).

Variables

Variable	Description	Type
list	View model of list.	`ListWidget.ViewModel`
listSequence (unless changed with *varSequence* attribute).	View model of list sequence info.	`SequenceHelper.ViewModel`
listId	Id of list.	`String`
listFullId	Full id of list.	`String`

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:list id="list">
    ...
</ui:list>
```

6.3.2. <ui:listFilter>

Represents list filter. Introduces an implicit form (<ui:form>), so one can place form elements under it.

This tag has no attributes.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:list id="list">
  <ui:listFilter>
    <ui:row>
        <ui:cell>
            <ui:textInput id="field1Filter"/>
        </ui:cell>

        <ui:cell>
            <ui:textInput id="field2Filter"/>
        </ui:cell>

        ...
```

```
    </ui:row>
  </ui:listFilter>
</ui:list>
```

6.3.3. <ui:listFilterButton> and <ui:listFilterClearButton>

`<ui:listFilterButton>` renders list's filter form filtering activation button and registers a keyboard handler, so that pressing `ENTER` key in any filter form field activates list filtering. `<ui:listFilterClearButton>` renders list's filter form clearing button, pressing it sends server-side event that clears all active list filters.

Both of these tags must be used inside `<ui:listFilter>` tag.

Attributes

Attribute	Required	Description
renderMode	*no*	Possible values are `button`, `input`—filter button is rendered with corresponding HTML tags, or `empty` in which case JSP author must provide suitable content for this tag by themself (with an image, for example). Default rendermode is `button`.
onClickPrecondition	*no*	Precondition for deciding whether registered onclick event should go server side or not. If left unspecified, this is considered to be `true`.
showLabel	*no*	Indicates whether button label is shown.Value should be `true` or `false`, default is `false`—using `true` is pointless with these particular tags, it only has some effect when specified `renderMode` is `empty` and tags body is left empty too.

Also have all common form element rendering attributes plus standard `style` and `styleClass` attributes.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
...
  <ui:listFilter>
    <ui:row>
      <!-- Bunch of filter fields in cells -->
      <ui:cell>
        <ui:listFilterButton/>
        <ui:listFilterClearButton/>
      </ui:cell>
    <ui:row>
  </ui:listFilter>
```

6.3.4. <ui:listRows>

Iterating tag that gives access to each row on the current list page. The row is by default accessible as EL variable *row*.

Attributes

Attribute	Required	Description
var	*no*	Name of variable that represents individual row (by default "row").

Variables

Variable	Description	Type
row (unless changed with *var* attribute).	Object held in current row.	`Object`

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:list id="list">
  <ui:listFilter>
    ...
  </ui:listFilter>

  <ui:listRows>
    <ui:row>
        <!-- In each row, object in this list row is accessible -->
        <ui:cell>
            <c:out value="${row.field1}"/>
        </ui:cell>

        <ui:cell>
            <c:out value="${row.field2}"/>
        </ui:cell>

        ...
    </ui:row>
  </ui:listRows>
</ui:list>
```

6.3.5. <ui:listRowButton>

Represents an HTML form button (not tied to any `Control` or `FormElement`). Default `styleClass="aranea-button"`, rendered with HTML `<button ...>` tag.

Attributes

Attribute	Required	Description
eventId	*no*	Event triggered when button is clicked.
id	*no*	Button id, allows to access button from JavaScript.
labelId	*no*	Id of button label.
onClickPrecondition	*no*	Precondition for deciding whether onclick event should go server side or not. If left unspecified this is set to `return true;`.
tabindex	*no*	This attribute specifies the position of the current element in the tabbing order for the current document. This value must

Attribute	Required	Description
		be a number between 0 and 32767.

Also has standard `styleClass`, `updateRegions` and `globalUpdateRegions` attributes.

6.3.6. <ui:listRowLinkButton>

Represents a HTML link with an `onClick` JavaScript event. Default `styleClass="aranea-link-button"`, rendered with HTML `<a href="javascript:" ...>` tag.

Attributes

Attribute	Required	Description
eventId	*no*	Event triggered when link is clicked.
id	*no*	Link id, allows to access link from JavaScript.
labelId	*no*	Id of link label.
onClickPrecondition	*no*	Precondition for deciding whether onclick event should go server side or not. If left unspecified this is set to `return true;`.
tabindex	*no*	This attribute specifies the position of the current element in the tabbing order for the current document. This value must be a number between 0 and 32767.

Also has standard `styleClass`, `updateRegions` and `globalUpdateRegions` attributes.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:list id="list">
  ...
  <ui:listRows>
    <ui:row>
      ...
      <ui:cell>
        <ui:listRowLinkButton eventId="edit">
          <img src="editButton.png"/>
        </ui:listRowLinkButton>
      </ui:cell>
      ...
    </ui:row>
  </ui:listRows>
</ui:list>
```

6.4. Editable Lists

`EditableListWidget` and `EditableBeanListWidget` are `ListWidgets` wrapped around `FormListWidget` (See Section 5.3, “Form Lists” about it) which gathers data with the help from `ListWidget`.

Both editable list widgets have just one constructor and one additional getter (compared to `ListWidget`):

```
public EditableListWidget(FormRowHandler rowHandler);
public EditableBeanListWidget(FormRowHandler rowHandler, Class beanClass);

// gets the wrapped form list
public BeanFormListWidget getFormList();
```

Most important component of editable lists is `FormListWidget`'s `RowHandler`, refer to Section 5.3, “Form Lists” about implementing that interface. Other than required implementation of `RowHandler`, editable lists do not differ from `ListWidgets`.

```
public class SampleEditableListWidget {
  private EditableBeanListWidget list;

  protected void init() throws Exception {
    setViewSelector("sampleEditableListView");

    list = new EditableBeanListWidget(buildFormRowHandler(), SomeBean.class);
    list.setDataProvider(buildListDataProvider());
    list.setOrderableByDefault(true);
    // list has only two columns of which only one is editable
    list.addField("immutable", "#ImmutableColumnLabel", false);
    list.addField("mutable", "#MutableColumnLabel").like();

    addWidget("sampleEditableList", list);
  }

  private FormRowHandler buildFormRowHandler() throws Exception {
    // return formRowHandler, see the form list example
  };

  private private ListDataProvider buildListDataProvider() throws Exception {
    // return data provider
  }
}
```

JSP view for this sample widget is presented below:

```
...
<ui:formList id="sampleEditableList">
  <!-- List filter definition, usual -->
  <!-- Editable lists body -->
  <ui:formListRows>
    <ui:row>
      <ui:cell>
        <!-- Row object is accessible as 'row' just as in lists -->
        <c:out value="${row.immutable}"/>
      </ui:cell>
      <ui:cell>
        <!-- But the implicit form tag for current row form is also present, so... -->
        <ui:formElement id="mutable">
          <ui:textInput/>
        </ui:formElement>
      </ui:cell>
    </ui:row>
  </ui:formListRows>
</ui:formList>
...
```

Full editable list example is bundled with Aranea examples.

Chapter 7. Trees

7.1. TreeWidget & TreeNodeWidget

`org.araneaframework.uilib.tree.TreeWidget` allows representation of hierarchical data in a manner that has become traditional in GUIs, as an expandable tree. `TreeWidget` represents trees' root node, which is special in that it is not really rendered on-screen but serves as point where child nodes are attached. Child nodes of `TreeWidget` are either attached by the programmer or acquired from associated `TreeDataProvider`. The `TreeWidget` supports expanding and collapsing of all those nodes.

`TreeDataProvider` is a simple interface with ability to return the data under given tree node.

```
public interface TreeDataProvider extends Serializable {
  /**
   * Returns a list of child nodes for specified parent node.
   */
  List<TreeNodeWidget> getChildren(TreeNodeContext parent);

  /**
   * Returns if the specified tree node has any children.
   */
  boolean hasChildren(TreeNodeContext parent);
}
```

As is apparent from the definition of `TreeDataProvider`, descendants of the `TreeWidget` that are to be presented in a tree, must be of type `TreeNodeWidget`. `TreeNodeWidget` is the superclass of `TreeWidget` that also has child nodes and will be rendered too. Node rendering is done with display widget that is passed to `TreeNodeWidget` in its constructor.

```
public TreeNodeWidget(Widget display);
public TreeNodeWidget(Widget display, List nodes);
public TreeNodeWidget(Widget display, List nodes, boolean collapsed);
```

Display widget can be any widget that can render itself, it is rendered in the place of tree node instead of `TreeNodeWidget`, which is just a data holder. Very often, display widget is `BaseUIWidget` which renders itself according to some JSP template. `TreeNodeWidget` does not accept independent `TreeDataProvider`, its children are acquired from `TreeWidget`'s `TreeDataProvider`.

`TreeWidget` enriches the `Environment` with `TreeContext`. `TreeNodeWidget` enriches the `Environment` of its display widget with `TreeNodeContext`. Through these contexts display widgets have access to owning tree node and root of the tree.

7.2. Tree JSP tags

7.2.1. <ui:tree>

Renders tree with given id.

Attributes

Attribute	Required	Description
id	*yes*	ID of the tree widget.

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:tree id="simpleTree"/>
<!-- nothing more required, nodes' display widgets will take care of rendering the tree nodes.>
```

Chapter 8. Third-party Integration

8.1. Spring Application Framework

8.1.1. BeanFactory, ApplicationContext, WebApplicationContext

`AraneaSpringDispatcherServlet` will always add a BeanFactory to the environment. It can be retrieved as follows:

```
BeanFactory beanFactory =
  (BeanFactory) getEnvironment().getEntry(BeanFactory.class)
```

Or using the method `getBeanFactory()` in `BaseUIWidget`. By default it will contain only beans configured by Aranea, however if one also uses usual Spring configuration:

```
...
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/services.xml
  </param-value>
</context-param>
...
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
...
```

Then the `AraneaSpringDispatcherServlet` will integrate with Spring and make `BeanFactory` provide all of the configure beans, as well as add `ApplicationContext` and `WebApplicationContext` to the environment.

Warning

`AraneaSpringDispatcherServlet` must be initialized after Spring context loader listener or servlet to integrate with Spring successfully.

8.1.2. Spring Localization Filter

Java class:	`SpringLocalizationFilterService`
Default configuration name:	-
Provides:	`LocalizationContext`, `SpringLocalizationContext`
Depends on:	`WebApplicationContext`

Provides localization services, see Section 2.8.2, “LocalizationContext”. The difference from the usual localization filter is that this one delegates the actual localization to a Spring `MessageSource`.

8.1.3. Widget Dependency Injection

Aranea does not by default support configuring widgets with Spring, as they are assumed to be created by the programmer and their life-cycle is managed by Aranea. The main problem however is that widgets are assumed to be serializable and Spring beans are often not (especially since they often are proxies with references to bean factory and so on). As a solution we provide a utility class SpringInjectionUtil that allows to inject Spring beans after a following convention:

```
...
injectSomeSpringBean(ISomeBean someBean) {
  this.someBean = someBean;
}
...
```

This method is similar to a setter method, but starts with "inject". The remainder of the method name is interpreted as the name of Spring bean to be injected, with the first letter lowercase (in the case of our example bean named "someSpringBean" would be injected). To actually inject the beans to all similarly called methods in the current widget call `injectBeans()` in widget `init()` method as follows:

```
...
protected init() {
  ...
  SpringInjectionUtil.injectBeans(getEnvironment(), this);
}
...
```

You may even put this call into the base widget of your application to ensure that all application widgets would get their dependencies injected.

Note

The injected bean must be an interface, as Aranea will construct an indirection proxy. This will ensure that the referenced object will be serializable (and small for that matter), but will also introduce a small performance penalty (we believe to be negligible next to the proxies of Spring itself).

Chapter 9. Javascript Libraries

9.1. Third-party Javascript Libraries

Aranea distribution includes some third party Javascript libraries. Most of these are not needed for using Aranea functionality, but extend the functionality of both framework and *UiLib*.

9.1.1. Behaviour (http://bennolan.com/behaviour/)

Neat little library allowing usage of CSS selectors for applying behaviour to HTML page elements. It is required for full function of Aranea pages rendered with Aranea JSP tags. With standard Behaviour rules, following functionality is provided:

- keyboard handlers are registered for form elements.
- addition of urls (<href="...") to page elements that allow cloning of the session thread (opening link in new window—and in different session thread).

9.1.2. AjaxAnywhere (http://ajaxanywhere.sourceforge.net/)

AjaxAnywhere allows separating web page into multiple zones, and refreshing only those zones that needs to be updated. Inclusion of this library is required for building AJAX capable web-page with Aranea JSP tags.

9.1.3. The DHTML Calendar (http://www.dynarch.com/projects/calendar/)

Nice DHTML calendar, required if one wants to use Aranea JSP <*ui:dateInput*> or <*ui:dateTimeInput*> tags.

9.1.4. Prototype (http://www.prototypejs.org/)

Prototype is a JavaScript framework that aims to ease development of dynamic web applications. Aranea JSP aims to make more use of it in the future. It is a prerequisite for using Uilib's *AutoCompleteTextControl* and action-enabled *TreeWidget* components.

9.1.5. script.aculo.us (http://script.aculo.us/)

script.aculo.us provides easy-to-use, cross-browser user interface JavaScript libraries. Only subset of script.aculo.us libraries are included—only JSP tag that depends on it is <*ui:autoCompleteTextInput*>.

9.1.6. TinyMCE (http://tinymce.moxiecode.com/)

TinyMCE is a platform independent web based Javascript HTML WYSIWYG editor control. Required for using Aranea JSP <*ui:richTextarea*> tag.

9.1.7. log4javascript (http://www.timdown.co.uk/log4javascript/)

log4javascript is a JavaScript logging framework similar to Java logging framework log4j. Include log4javascript scripts and call *AraneaPage.setDefaultLogger()* to receive a popup window where Aranea JS debug output is logged.

9.2. Aranea Clientside Javascript

Aranea uses javascript to do form submits. This provides AJAX enabled webapps and more control over form submitting logic. Each page served with Aranea has associated *AraneaPage* object:

```
/* AraneaPage object is present on each page served by Aranea and contains common
 * functionality for setting/getting page related variables, events and functions. */
function AraneaPage() {
  /* Getters and setters for URL of aranea dispatcher servlet serving current page. */
  function getServletURL();
  function setServletURL(url);

  /** Indicates whether the page is completely loaded or not. Page is considered to
   * be loaded when all system onload events have completed execution. */
  function isLoaded();
  function setLoaded(loaded);

  /* Returns the server-side reported locale (AraneaLocale). */
  function getLocale();

  /** Indicates whether some form on page is (being) submitted already
    * by traditional HTTP request. */
  function isSubmitted();
  /* Custom submit functions using plain HTTP request should call this before submit. */
  function setSubmitted();

  /** Returns Aranea JSP specific DOM tree traverser (AraneaTraverser). */
  function getTraverser();

  /** Add events that should be executed upon page load/unload to execution queue. */
  function addSystemLoadEvent(event);
  function addClientLoadEvent(event);
  function addSystemUnLoadEvent(event);

  /** Called on page load, executes registered system -- after which page is considered
    * loaded and client load events are executed too. */
  function onload();
  /** Called on page load, executes registered unload events. */
  function onunload();

  /** Adds general callback executed before any form submit. */
  function addSubmitCallback(callback);

  /** Adds general callback executed before form with given id is submitted. */
  function addSubmitCallback(systemFormId, callback);

  /** Executes all callbacks that should run before submitting the form with given id. */
  function executeCallbacks(systemFormId);

  /**
   * Chooses appropriate submitting method and submittable form given the HTML element
   * that initiated the submit request. */
  function event(element);

  /**
   * Called by event() to determine the appropriate form submitter.
   */
  function findSubmitter(element, systemForm);

  /** another submit function, takes all params that are possible to
    * use with Aranea JSP currently. */
  function event_6(systemForm, eventId, eventTarget,
                   eventParam, eventPrecondition, eventUpdateRegions);

  /**
   * Provides preferred way of overriding AraneaPage object functions.
   * @param functionName name of AraneaPage function that should be overridden.
```

```
   * @param f replacement function
   */
  function override(functionName, f);

  /**
   * Adds keepalive function f that is executed periodically after time
   * milliseconds has passed
   */
  function addKeepAlive(f, time);

  /** Clears/removes all registered keepalive functions. */
  function clearKeepAlives();

  /** Logs message on DEBUG level, if logger is present. */
  function debug(message);
}

/** Random request id generator. Sent only with AA ajax requests.
  * Currently only purpose of it is easier debugging (identifying requests). */
function AraneaPage.getRandomRequestId();

/* Returns a default keepalive function -- to make periodical requests to expiring thread
 * or top level services. */
AraneaPage.getDefaultKeepAlive = function(topServiceId, threadServiceId, keepAliveKey);

/** Page initialization function, should be called upon page load. */
AraneaPage.init();

/* Aranea page object is accessible in two ways -- _ap and araneaPage() */
_ap = new AraneaPage();
function araneaPage() { return _ap; }
```