

# ARANEA INTEGRATION LAYER

## Whitepaper

Jevgeni Kabanov

### 1 Requirements

One of the main design decisions in Aranea was to enforce Object-Oriented principles like encapsulation. This means that Aranea components are first-class objects that can be used abstractly, without any care for their implementation. This immediately gives rise to a question—is this abstraction powerful enough that we could implement Aranea components using third-party frameworks?

Application integration has two main aspects<sup>1</sup>:

**Encapsulation** Ensures that application behavior does not depend on any other applications.

**Communication** Allows different applications to interact with each other.

It is obvious that these aspects are conflicting with each other, since the need to communicate breaks encapsulation. To solve this problem we introduce a level of indirection—encapsulation will be provided by specific Aranea components that are aware of the application implementation. All communication between different applications will only be done using Aranea API. This way we provide full encapsulation from the Aranea point of view as well as enable arbitrary communication when needed.

Let's examine how our integrated application will be structured and what requirements encapsulation must satisfy.

First of all it is a logical step to embed encapsulation into component base classes. Although both services and widgets can provide integration, we will examine widgets more, since they are used more often and have more issues. For example Struts encapsulation may have a `StrutsWidget` base class and JSF encapsulation a `JsfWidget`. These widgets should encapsulate a particular application, or rather an application module, by taking a starting URI into their constructor. For example `new StrutsWidget("/Logon.do")`.

To encapsulate a particular module we may create a widget that is tied to a particular starting URI and takes any necessary extra parameters as constructor arguments. This way we can use this widget as we do usually, both including it as a reusable component and starting it in a flow. We can also include widgets from the encapsulated module, including widgets implemented in yet another third-party framework. In fact we should

---

<sup>1</sup>We could also identify a third one—*coherence*—that ensures all applications are using the same basic set of services, but this is less about application integration and more about framework integration.

provide access to Aranea API also to the native integrated application modules, so that they could make use of Aranea components and flows.

From this we can deduce several requirements:

- Several widgets can appear on one HTML page. Even more important those widgets could be instances of one and the same class, thus implemented by one and the same integrated module. In such a case encapsulation becomes all the more important, since the integrated modules will definitely have overlapping namespace, state and so on.
- All of the requests made from the integrated module must go through the Aranea component hierarchy to the encapsulating component that originated the request and only then the module should be included and rendered. Otherwise, if we let the request go to the originating application, the response will display only the integrated application, breaking the encapsulation.
- Since all inter-module communication takes place via Aranea components, we must implement application basic features using Aranea. This includes such features as authentication (usually implemented by a login screen) and module starting points (menus and similar).
- To integrate the applications (or modules) they need to undergo some amount of conversion. At the very least they will depend on Aranea API for communication and use Aranea authentication services. This means that they will not be able to run standalone anymore.

## 2 Implementation

Our approach allows to integrate several applications written using different web frameworks inside a single web container<sup>2</sup> application. The applications are inside the same JVM and they share access to Java Servlet API. Thus we can simulate parts of this API to enable independence of integrated applications and their modules.

### 2.1 Request and State

First of all let's examine how servlets can interact with the user and among themselves:

**Request URL.** The basic URL that request was submitted to. Different servlets are mapped to different URLs and the specific path after the servlet URL is used for internal application communication. Since we want to fully encapsulate the applications in Aranea components we need to ensure that all requests will go through the Aranea component hierarchy to the correct component before being processed.

---

<sup>2</sup>Since we use the term *web applications* generically we will use the term *web container applications* to refer to the Java web application deployment unit that contains a single `web.xml` mapping.

**Request parameters.** Carry the information submitted by a user to the server. Since different applications share the same namespace of parameter names we must make sure that applications get access only to parameters submitted by themselves.

**Request attributes.** Allow communication between different components of a web application inside the same request. Since these are also shared by all applications involved in the request we must make sure that they do not influence each other.

**Session attributes.** Allow storing user session-specific data between requests. Also shared by all applications, so we need to ensure that they can only access their own attributes.

**Servlet attributes.** Allow communication between servlets in the same web container application. Not as important as the rest, since web applications must anyway assume this is shared and provide their own namespace.

It seems that most of the important interactions is request-based<sup>3</sup>. HTTP Request is represented in Java by the interface `HttpServletRequest`, which includes methods related to:

- Request URL, like `getRequestURL()`, `getServletPath()`.
- Request parameters, like `getParameter()`, `getParameterMap()`.
- Request attributes, like `setAttribute()`, `getAttribute()`.
- Getting associated session: `getSession()`.

Since `HttpServletRequest` is an interface, it can be wrapped using the *Decorator* pattern [?]. Servlet API even provides a default wrapper implementation—`HttpServletRequestWrapper`. Using this pattern we override the default behavior of the methods by making them *local* as follows:

- We make the wrapper take the request URI as the constructor parameter. It should return it and its derivatives from all associated methods. We can safely assume that the URL part besides the URI (that is host, port, etc) is the same for all applications.
- We assume that all of the request parameters have been prefixed with some namespace. We take this namespace as a constructor parameter and make the application see only the request parameters inside this namespace.
- We create a local request attribute map. All attribute modification methods act only on this map, while all attribute query methods act first on the local map then globally. This ensures that although application can see the global attributes (e.g. set by Aranea) it cannot influence other applications.

---

<sup>3</sup>Sessions in Servlet API are accessible only via the request

Additionally, since `HttpSession` is an interface and can only be accessed from the `HttpServletRequest`, we can also wrap it and provide a local attribute map acting same way as the request attributes. A logical place to put the attribute map is the `Aranea` component that encapsulates the module, since its scope should exactly correspond to the user session with the integrated application.

The resulting wrapper will look something like this:

```
class RequestWrapper extends HttpServletRequestWrapper {
    private Map localRequestAttributes = new HashMap();
    private String prefix;
    private String requestURI;
    private SessionWrapper session;

    public RequestWrapper(HttpServletRequest request,
        String requestURI, String prefix, Map sessionAttributeMap) {

        super(request);

        this.prefix = prefix;
        this.requestURI = requestURI;
        this.session = new SessionWrapper(sessionAttributeMap);
    }

    public HttpSession getSession() {
        return session;
    }

    public String getParameter(String name) {
        if (name.startsWith(prefix))
            return super.getParameter(name);

        return null;
    }

    public void setAttribute(String name, Object o) {
        localRequestAttributes.put(name, o);
    }

    public Object getAttribute(String name) {
        if (localRequestAttributes.containsKey(name))
            return localRequestAttributes.get(name);
        return super.getAttribute(name);
    }

    public String getRequestURI() {
        return requestURI;
    }
}
```

```
//...
}
```

This wrapper encapsulates all of the basic servlet-to-servlet communication, and we can proceed with implementing the rest of application integration.

## 2.2 Navigation

In most of the web frameworks request URL is used for navigation. In all of the web frameworks that we are aware of request URL is relied upon by the framework for some of its functionality.

Thus we must preserve the original URL when passing the request to the integrated module. At the same time we must make sure that Aranea handles the request first and includes the integrated module in its proper component. There are two problems with these requirements:

1. Most of the web frameworks will have their servlet URLs hard-coded into their HTML output, thus producing requests to their own servlets.
2. Even if they would produce requests to Aranea we would still need to know both the original servlet path and some Aranea-specific data like the identifier of the component hosting the integrated module.

To solve the first problem we make use of servlet *filters*. We implement the `IntegrationFilter` filter that saves the current URL as a request parameter and after that redirects to the Aranea servlet, then map this filter to all of the integrated application servlets.

To solve the second one we need to wrap the `HttpServletResponse` in the encapsulating component. `HttpServletResponse` is the Java Servlet API abstraction for the HTTP response to the submitted request, and besides the output stream and headers it has a method `encodeURL()`. This method is used by the Servlet API to allow for cookie-less session tracking. To support this it must be called on all the URLs embedded in the generated HTML output.

By overriding its default implementation we can add encoded Aranea-specific information that allows `IntegrationFilter` to route the request to the intended Aranea component. The resulting `ResponseWrapper` class may look something like this:

```
class ResponseWrapper extends HttpServletResponseWrapper {
    //...

    public String encodeURL(String url) {
        StringBuffer urlB = new StringBuffer(url);

        urlB.append(url.indexOf('?') != -1 ? '&' : '?');
        urlB.append("araInfo=")
            .append(':').append(topServiceId)
            .append(':').append(threadServiceId)
    }
}
```

```

        .append(':').append(widgetEventPath);

    return super.encodeURL(urlB.toString());
}

//...
}

```

The filter can then decode this information and dispatch the request to the Aranea servlet that will cause the hosting component to set up the encapsulation and dispatch the request further to the integrated module. This will appear as if the navigation took place inside the component, since other components will not change.

## 2.3 Rendering

Now let's look into rendering the hosting widgets<sup>4</sup> and the integrated modules. There are two main issues with rendering:

1. It would be logical to dispatch the request to the integrated module on `render()` call. However we have to keep in mind that the integrated module has access to the Aranea API and thus can modify the component hierarchy e.g. by starting a new flow. In this case rendering would be incorrect, as the flow container will render the old flow (possibly failing).
2. Not every framework is capable of just re-rendering the current "page". Since in Aranea only one component at a time receives an event, all the rest must re-render their previous state. But for a framework like Struts this might mean calling an action and possibly producing undesired side-effects.

Both problems admit one and the same solution. To allow component hierarchy modification we do the actual rendering during an event `event()` call. We then cache the resulting output as a character array and write it out during the actual render phase. We save the cached character array between requests and render it again if no event comes. Since at each request only one component gets an event call (typically the one that originated the request), the rest of the components will just display the previously cached response that is identical to the previous one.

To cache the output we must wrap the output stream provided by the `HttpServletResponse`. We can do this by adding a `getOutputStream()` method to the `ResponseWrapper` that returns the `OutputStreamWrapper`<sup>5</sup>:

```

class OutputStreamWrapper extends ServletOutputStream {
    private ByteArrayOutputStream out;

    OutputStreamWrapper() {
        reset();
    }
}

```

---

<sup>4</sup>Services need no additional support and can just dispatch the encapsulating request to the appropriate URI from the `action()` call.

<sup>5</sup>We will also need to wrap it in a `PrintWriter` for the `getWriter()` method.

```

}

public void write(int b) throws IOException {
    out.write(b);
}

//...

public void flush() throws IOException {
    out.flush();
}

public void reset() {
    out = new ByteArrayOutputStream(20480);
}

public byte[] getData() {
    return out.toByteArray();
}
}

```

Although we fixed one problem, we introduced another. Since we want the integrated modules to be able to include other widgets, they will also re-render by returning the cached response. However widgets are perfectly capable of re-rendering themselves and it is not a good idea to cache their output. The solution is to cache only those parts of response that are produced by the integrated module, not by included widgets. To do that we need to introduce a new tag `<ui:hostedWidgetInclude>` that would call the hosting widget `renderWidget()` method. Then we can create an array of closures that will render either the cached output or the included widget in the correct order as shown on Figure 1.

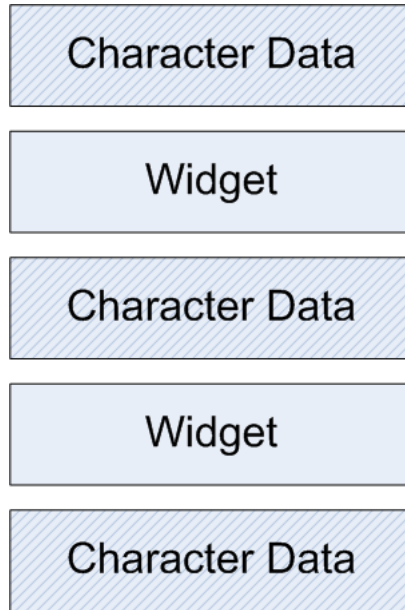
## 2.4 View Sanitizing

Although the response generated by the integrated modules is partially preprocessed as described in subsection 2.2, it may still contain some HTML not acceptable in an Aranea component. This is mostly due to HTML limitations:

1. Every HTML page can contain only one `html` tag, `head` tag and `body` tag.
2. `form` tag cannot be nested. Since Aranea defines a single root form, integrated modules are not allowed to define their own forms.
3. Form element names must be prefixed with the component identifier to allow their later separation as described in subsection 2.1.

These problems can be solved by one of the two approaches:

**Conversion** The simplest solution is to change the integrated module view code and introduce the necessary changes (remove root tags like `html`, remove `form` tag and



Joonis 1: Renderer chain

prefix the element names). Most probably some associated changes will also have to be done (Javascript references to form elements, form submission logics, etc). The problem with this solution is that it implies a significant effort unless the logic was already well encapsulated, which is a relatively rare case.

**Postprocessing** A different approach is to render the response and the postprocess it by parsing the resulting HTML and applying the changes automatically. This approach has the benefit of allowing in simpler cases to just drop the application in and get integration to work in a number of minutes. The main drawback is the inevitable decrease in performance.

In reality full conversion should only be used if the performance penalty is unacceptable. The overhead is not as great as could be expected, since it does not involve Input/Output, which is the main bottleneck in web applications.

Typically postprocessing is combined with some amount of conversion, since some things are simpler to postprocess while other are simpler to just change in place. The ratio of one to another depends on the particular application.

Postprocessing is applied on `render()` call and uses the cached response gather during the `event()` call.

#### 2.4.1 AJAX

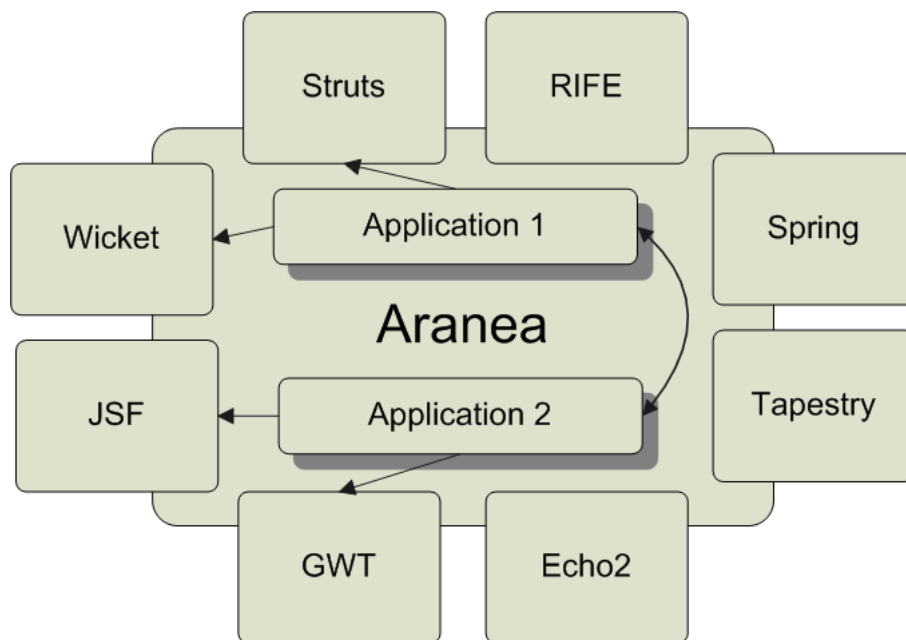
The final issue to consider is handling AJAX calls from the integrated module. By itself it does not differ much from usual integration, except that Aranea widgets are by default synchronized, which may cause trouble if the request is expected to be processed asynchronously.

The request will be submitted to some third-party servlet and should be handled by mapping a subclass of the `IntegrationFilter` to the servlet. This subclass should be

able to distinguish AJAX requests from common ones (this distinction being, of course, framework-specific) and send AJAX requests to the hosting widget as an unsynchronized action call. The hosting widget should also define this action, which should just dispatch the request to the integrated module after setting up request encapsulation.

### 3 Legacy Migration

Integration allows to integrate several applications together as illustrated by Figure 2. However in addition to that it also allows to migrate legacy applications from outdated



Joonis 2: Web integration illustration.

technologies to newer ones.

To do that we need to prepare the application as follows:

1. *Implement adapters for in-house web frameworks.* A typical legacy web application is likely to use some in-house web framework. We need to introduce a specific Aranea adapter component that will handle the framework-specific quirks.
2. *Analyze the application design.* Since most application do not use Object-Oriented technologies we often need to reverse-engineer the design behind them. Mainly we are interested in splitting the application into logical use cases and identifying their interdependencies.
3. *Convert the legacy application to use Aranea integration.* This involves rewriting some basic features and wrapping use cases in Aranea widgets. We also need to lift all communication among use cases to use Aranea API. The result is a fully Object-Oriented application with components implemented using the legacy technology.

After the application is prepared we may start implementing new use cases using any other technology integrated with Aranea. We do it by defining a subclass of the wrapping widget corresponding to the implementation technology. Since all communication among use cases is done in terms of Aranea API the legacy components cannot be distinguished from the non-legacy ones.

To actually migrate away from the outdated technology we apply this simple rule—rewrite legacy use cases only when business requirements change. Since in such a case a significant portion of the use case would have to be rewritten anyway and the development costs using an outdated technology are higher, the migration overhead cost is very low.

We call such an approach *step-by-step legacy migration*, since instead of doing an all-at-once rewriting of the system functionality we run the old and new technologies side-by-side and migrate only when costs and time allow.

In addition to the direct economical benefit the following extra benefits are attained:

- Since development of features requiring a new technology can start immediately the time-to-market decreases.
- Newer frameworks might provide features that increase usability without any extra effort from the developers.
- Object-Oriented design means that development is easier to scale, since particular use cases can be written by any third-party in technology of their choice.
- When a newer and better technology appears the enterprise is somewhat insured against the migration costs, since the application design is already technology-agnostic and migration to the newer beneficial technology can begin immediately.