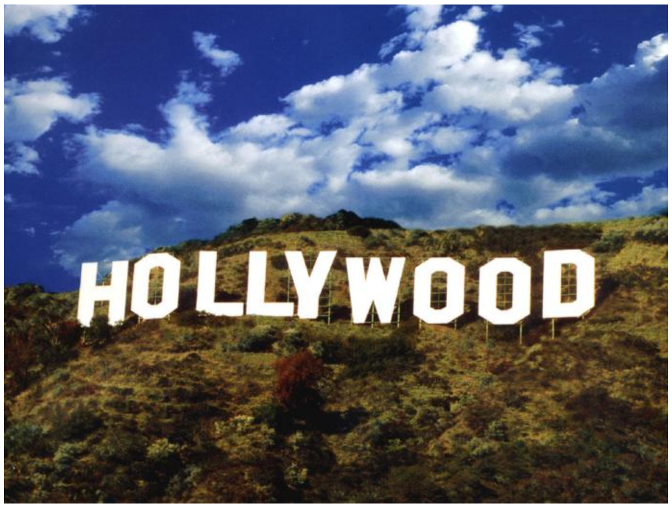
The Hollywood Principle - on dependency injection

It is interesting to find that a city not typically known for its moral rectitude lends its name to an important principle, even if it is one that applies to IT. Yet this Hollywood principle plays an important part in Oracle's Fusion Middleware stack - and beyond - and is crucial for that holy grail of modern day software development: decoupling.



On a Thursday afternoon in the year 1926, somewhere in The Valley, it is movie director Sam Ornstein who is the first to utter the words that were later to become so notorious: "Don't call us [..], We'll call you!". Whether Samuel was trying to keep hopeful but talentless wannabe actors out of his hair or he was sincere in his effort to spare them the trouble of contacting him - especially with him being hard to get hold of with all his running around - is not recorded in the annals of that time. However, the concept of not making a party depend on - and privy to - your contact details by contacting them yourself (and only when you have a need to do so) caught on. First in the movie business, and more recently in software design methodology.

Is it Hard Coding dependencies

Most of the software we create does not stand alone. Our programs live in an environment with other components that they may have relationships with. For example because these components call out to our programs - or because our programs depend on these surrounding components. In the latter case, a program has one or more dependencies.

A different type of dependency is not so much one on other programs, but one on information. Many of our programs may perform calculations, do validations or make decisions. They typically do so using specific settings, values that are used in the algorithms in these programs. Examples of such values are the price of a product, the discount percentage for bulk orders, the expiration period of an offer or simply the shipping address of our warehouse. These values that are used in our programs can be part of the program code: inline, right where they are needed, or as global variables or even constants.

The objective for today's software architecture - if nothing else - is to embrace change: facilitate changing business requirements by providing flexibility and configurability in our applications and make sure the impact of any change is minimal. One way of doing so is to strive for maximum decoupling - as we have discussed in a previous installment of this column. Even though dependencies are unavoidable and synergy resulting from close collaboration between components is a good thing, components should not become too close as tight integration and intimate relationships are the enemy of flexibility and agility.

Closely associated with agility is the pursuit of reuse. When components lend themselves to reuse, we can more rapidly compose implementations for business requirements by combining such reusable components. The additional savings from reduced development and test effort as a result from reuse are an additional incentive. In order to consume components in multiple environments and contexts, that we probably do not know about at design and development time, we have to prevent these components from having hard coded dependencies on objects and settings that may not be available or applicable in the environment in which they are to be reused.

Here is where the Design Pattern called "Dependency Injection" comes to the rescue. This pattern is the computer system's equivalent of the Hollywood Principle: "Do not call out to us, let us first tell you where to reach us" (and do not assume anything - we will tell you how it is).

The design pattern prescribes how components should not have any hard coded dependencies or value constants but instead must provide hooks or entry point where values and references (to satisfy internal dependencies) can be injected from outside the component. This injection must be done at least once, prior to calling the component, usually upon initialization of the component. Additional injection takes place whenever a change takes place in the environment - when new settings are to be applied or other services are to be called from the component than before.

Injection is done on behalf of the re-user - when the context in which the component is (re)used changes. Typically the component itself does not (need to) know about such a change.

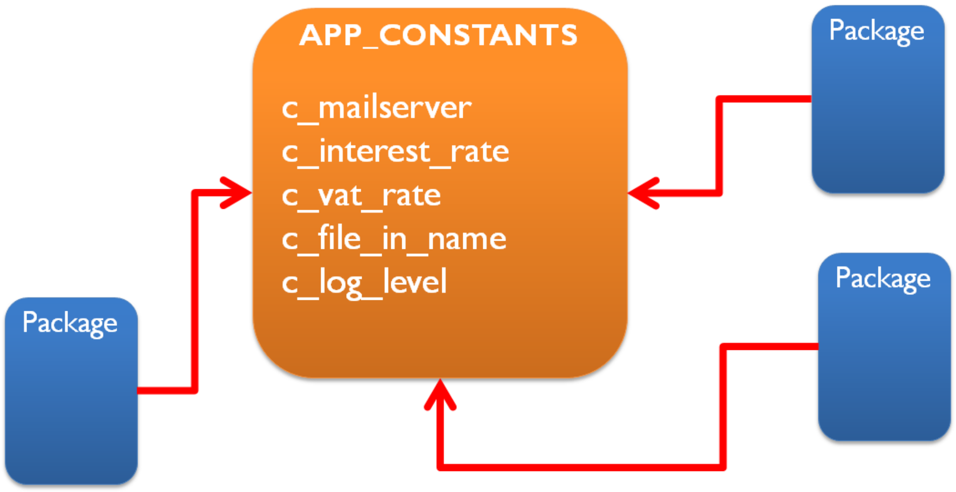
One of the added benefits of dependency injection is that it enables true unit testing of software components. Since we can make injection of dependencies part of the preparation of the unit test of a component, we can inject mock implementations for the dependencies of the component - that intercept the calls and return fake responses. That way, the test is really only about the component itself.

Another term used in conjunction with Dependency Injection is Inversion of Control. What meant is that instead of a component deciding whom to call, it is the assembler of the run time software composition who determines which calls a component will make and which settings to work with. The control - and responsibility - has shifted away from the component to the consumer of the component (who after all knows best).

Dependency Injection in PL/SQL

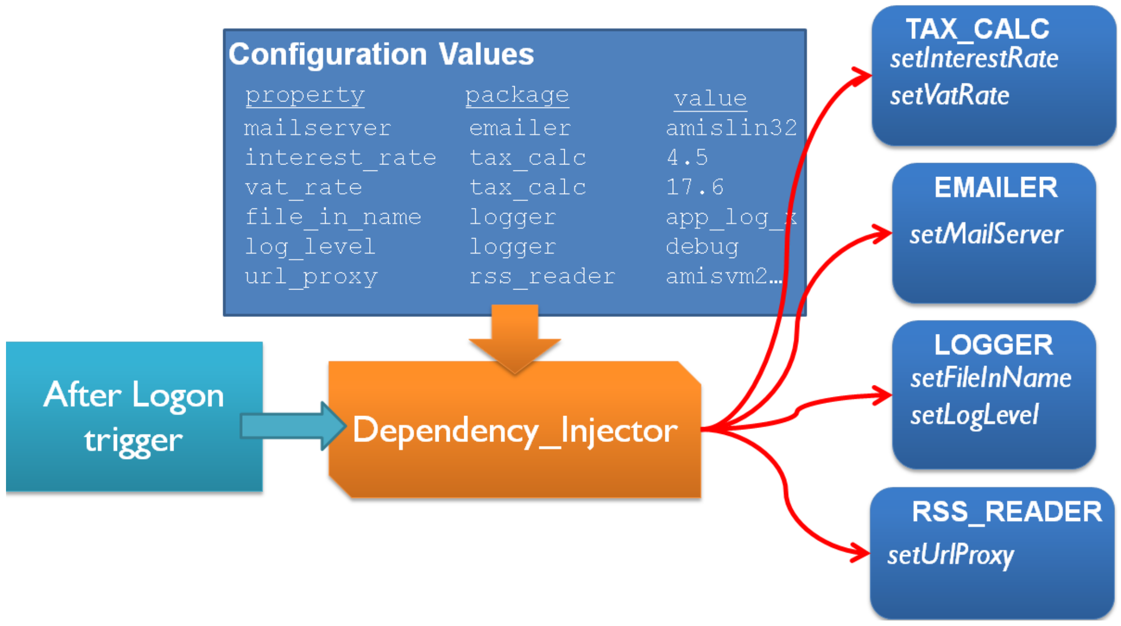
Even though this column focuses on Fusion, it is good to take a moment and apply the above to the world of PL/SQL where it has as much relevance as it does in Fusion Middleware. The components that we may select for reuse in PL/SQL - and that are or should typically be the subject of unit tests - are packages. It is into packages that we may need to inject dependencies, in order to make them less coupled, more agile and more reusable.

It should come as no surprise to any PL/SQL programmer that using hard coded values between begin and end statements is not a good idea. Much better is it to use constants at package level or even to use a single package that has no other function than to hold constants for all programming units in the application. This gives us a central location for maintaining values, should they require changing.



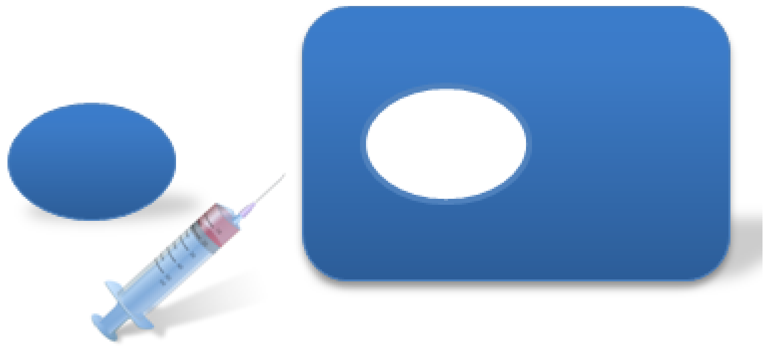
While this is a fairly elegant way of working with hard coded values, it is not the ultimate in decoupling. We still require a recompilation of the APP_CONSTANTS package with every change in one of the values it controls. However, much worse: every package shown in the diagram has a dependency on APP_CONSTANTS. These packages cannot be compiled successfully unless APP_CONSTANTS is around. This severely restricts the reusability of the packages. And it does not allow us to easily integrate a reusable package from a third party - as that package probably does not know about APP_CONSTANTS and the values it currently defines. An alternative could be the use of a Global Application Context - a cross-session collection of values, possibly including these application constants.

A more sophisticated way of dealing with such package settings is true dependency injection. This requires every package to expose setter-procedures through which the values of the settings can be injected. Because package live and run in the context of user sessions, they should be initialized afresh for every new user session - meaning that all relevant setter procedures should be invoked when a new session is started. The database provides us with an AFTER-LOGON trigger that we can implement to inject the current values of all package settings, during the instantiation of a new session. When the packages are invoked to perform their work, they will already have had the values injected and can do their thing in a context specific manner.


Configuration Values

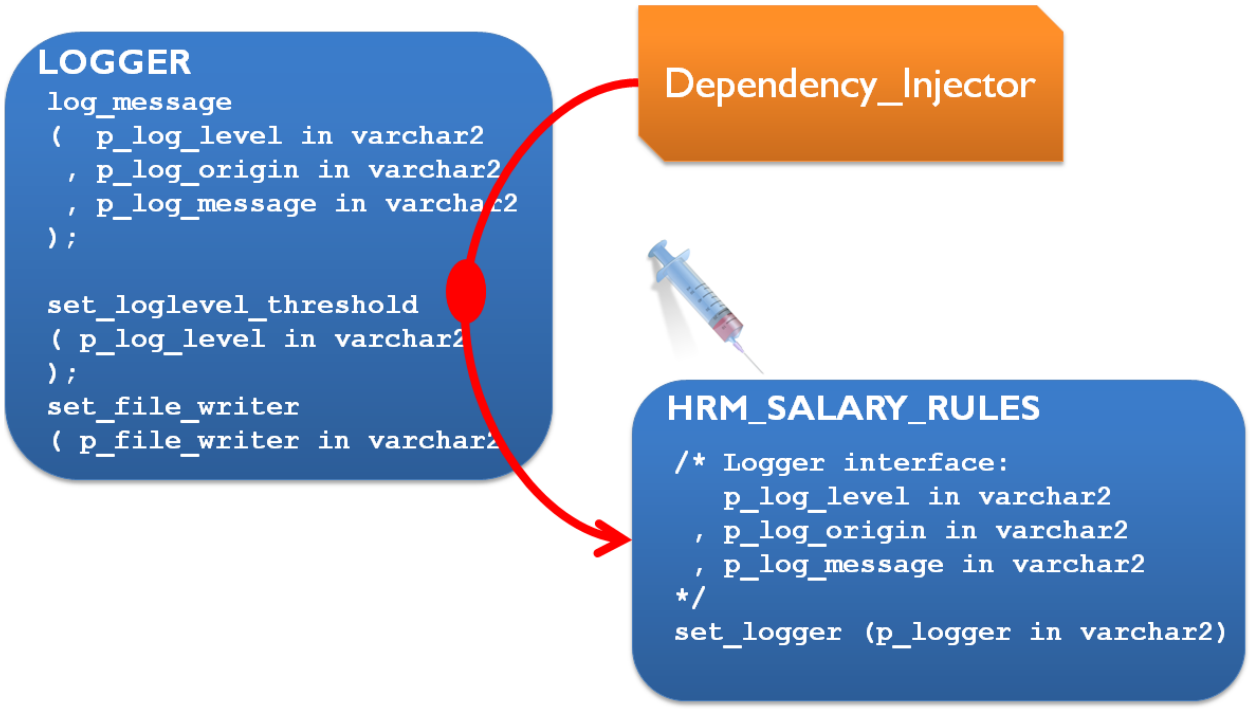
property	package	value
mailserver	emailer	amislin32
interest_rate	tax_calc	4.5
vat_rate	tax_calc	17.6
file_in_name	logger	app_log_x
log_level	logger	debug
url_proxy	rss_reader	amisvm2…

In addition to their use of settings and configuration values, package rely on other program units to perform tasks on their behalf. Most packages will invoke other packages to provide information or execute some operation. When these calls are hard coded in the package, the package is intimately tied to the invoked units. Sometimes that is perfectly alright, when packages are really together in a functional unit. However, sometimes such calls should be seen in a much more decoupled way. For example take a look at a package that is capable of producing logging information about its internal operations. It could contain all logic required to write logging, for example to a database pipe, the server output or a table. However, by doing that, the package would be tightly coupled to whichever output channel it supports and would not easily be reused in an environment that has a specific approach to logging in place.



Instead, the package could expose a procedure set_logger through which a logger companion can be injected. If no logger is injected, the package will refrain from writing logging. However, when a specific logger is set, the package will invoke it whenever it has logging information to share. The package can be reused in any environment, with any logging solution, and has no tight logging relations itself.

In this simplistic example, the logger reference that is injected is a simple string "LOGGER.LOG_MESSAGE" that identifies a PL/SQL procedure that has the signature prescribed by the HRM_SALARY_RULES package. That is: this package can invoke the logger with a dynamic PL/SQL call that passes in three varchar2 parameters that represent the logging level, the origin of the log message and the actual message. It is then up to the logger how to process that information from HRM_SALARY_RULES.



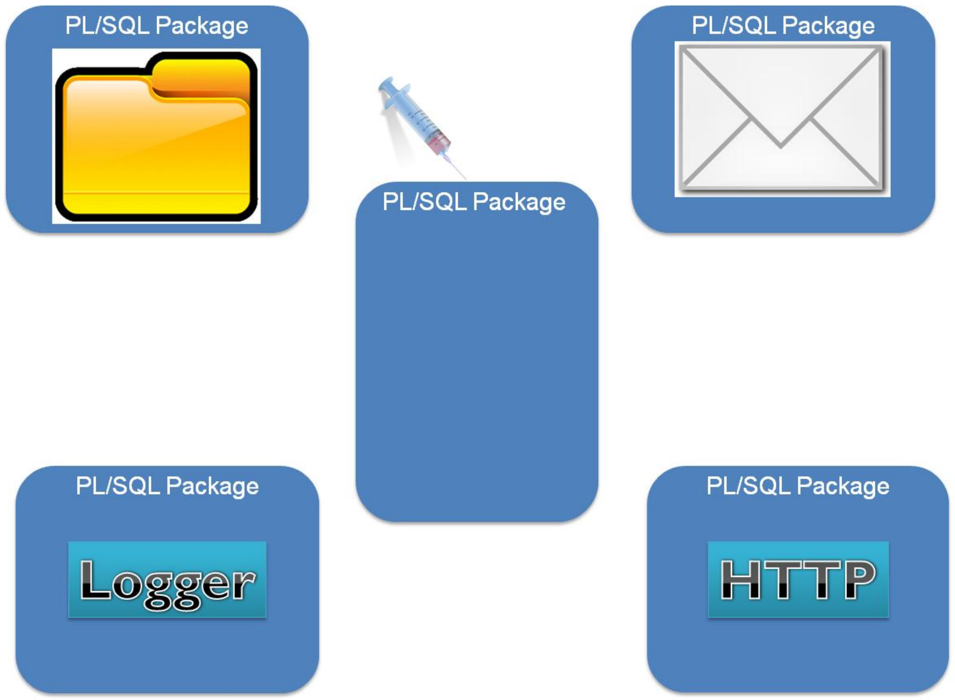
```
log_message
(  p_log_level in varchar2
 , p_log_origin in varchar2
 , p_log_message in varchar2
);

set_loglevel_threshold
( p_log_level in varchar2
);
set_file_writer
( p_file_writer in varchar2
```

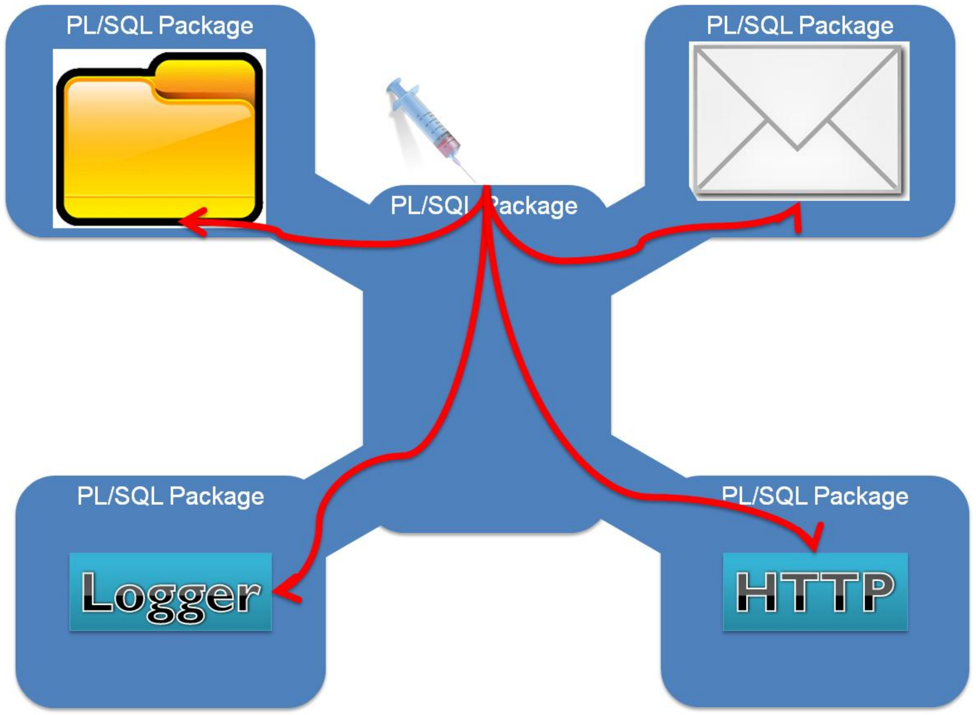
```
/* Logger interface:
   p_log_level in varchar2
 , p_log_origin in varchar2
 , p_log_message in varchar2
*/
set_logger (p_logger in varchar2)
```

Other examples of dependencies that PL/SQL packages could have include emailing, file writing and printing as well as validation, calculation and decision making. Note that some of these dependencies can be optional: the package will work fine if a dependency is not satisfied; however the package offers the consumer a chance to interfere at specific points in the process (this resembles another design pattern, often called the decorator pattern).

In general, in much the same way as with the dependency injector for package settings, we can have a central dependency injector that is invoked from the after logon trigger visit all packages with dependency requirements and satisfy their requirements by injecting references.



Once the dependencies have been injected, each package can perform its task, calling upon the references that were injected to get specific help when needed.



Note: the blog article http://technology.amis.nl/blog/1086/dependency-injection-in-plsql-remove-hard-coded-dependencies-from-your-code gives a more detailed example including working code and some refreshing comments.

Fusion Style Dependency Injection

Back to the overall focus of this column series: Oracle Fusion technology. It should be clear by now how dependency injection decouples components: a component is told what to use and who to call, it does not have these dependencies pre-configured or hard-wired. Promoting reuse of components and lowering the impact of change are the main objectives and the stand-alone unit testing of components through the injection of mock dependencies is a nice bonus.

So how and where does this work out in Fusion Middleware? As was stated before, Dependency Injection and Inversion of Control are among the driving principles behind Oracle FMW (as well as most modern technology stacks). I will discuss some examples for various parts of the stack, to give you an idea of the pervasiveness of this approach. This should at least allow you to recognize it when you come across it. And perhaps you see ways of applying that principle in the software you are involved with yourself.

JDBC Data Sources

One example of dependency injection is found in JEE Application Servers, such as WebLogic Server, with JDBC Data Sources. Java Web Applications frequently have a dependency on a relational database, to access tables, views and packages. However, if an application contains the exact connection details for the target database, the application is very inflexible in terms of deployment in various environments. Besides, application developers would have to have information they typically do not possess. The solution in JEE is quite simple and elegant: applications indicate they have a need to access a database by specifying one or more JDBC Data Sources. That is: the application configures the name of the data source and may contain code - or use frameworks like ADF BC or JPA (through EclipseLink) - that access the data source through that data source.

The administrator configures JDBC Data Sources on the application server, independently of the application, specifying all database connection details. When the application executes, the application server will inject the JDBC Data Source. Thus, the application can access the database, without having the faintest idea about its whereabouts. There is no dependency from the application on the database and the application can be reused against various databases.

More in general, components in Java Web Applications such as Servlets, Listeners and JSF managed beans can use the @Resource annotation to specify a dependency on resources such as simple String values, JDBC Data Sources, Web Services, EJBs and JMS constructs. The JEE container will inject the requested resources at run time into the components.

EL Expression in JavaServer Faces

User interfaces in Fusion applications are typically created using JavaServer Faces (JSF) - primarily ADF Faces Rich Client components. The pages are created in JSF using components. Examples of components include input text field, button, link, table and tree. Each component is configured on how exactly to render itself, through attributes such as value, disabled, label, inlineStyle, hint, valueChangeListener and actionListener.

```
<af:inputText label="Username:"
              columns="25"
              required="no"
              value="John Doe" />
```

These attributes can be set with hard coded values - as shown in the code snippet above. However, much more interestingly is the use of EL (Expression Language) expressions to specify the values of attributes. The following snippet illustrates the use of these expressions:

```
<af:inputText label="#{labels.username}"
              columns="25"
              required="no"
              value="#{user.username}" />
```

EL expressions set on an attribute inject the component with references to an object that they can call when they need to know the currently applicable value. In the example above, the value for the label attribute - which is used for the prompt of the inputText component - is retrieved by the component from an object called labels (probably a Java Map object), using the key 'username'. In a similar way, the component is injected with the expression #{user.username} that instructs the component to read the current value to display in the input field from the property username on an object called user. When the user enters a new value for the username, this value is also to be sent by the inputText component to the user object, to set it on the username property.

Note that once more an object - the inputText component in this case - is wired with references to an object that it knew nothing about at design time yet it will call at run time. The inputText component is eminently reusable - a single page can already contain many instances of this component.

Managed Beans

JavaServer Faces has a built-in capacity to instantiate Java objects at run time, based on a bean configuration file. For example the user object that was used in the EL Expressions in the previous section, is instantiated by the JSF framework at run time. The bean configuration file specifies how to create the user object - for example the Java Class name on which it is to be based and the memory scope in which to create it (request, session or application):

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>nl.amis.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>password</property-name>
      <value>welcome123</value>
   </managed-property>
</managed-bean>
```

These objects - called managed beans - are created by the framework when it first encounters an EL expression that refers to them - and subsequently injected into the application. JSF can set (or should I say inject) the (initial) value of properties when the bean is instantiated.

An interesting feature of the managed bean definitions and JSF bean creation is the option to inject other beans into managed properties. Beans can thus call other beans, even when the other beans are unknown at design time. See this example, where the user bean is injected into the specialBean through dependency injection.

```
<managed-bean>
   <managed-bean-name>specialBean</managed-bean-name>
   <managed-bean-class>nl.amis.SpecialBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>manager</property-name>
      <value>#{user}</value>
   </managed-property>
</managed-bean>
```

Note: this style of bean management, where a container creates and configures objects and injects dependencies based on configuration files or - as is more common these days - annotations in Java Classes is what got the well known Spring Framework started.

Resource Bundles

Another example of dependency injection in Web Applications is the use of resource bundles to provide the translation of boilerplate text elements such as prompts, messages, hint texts and button labels. Resource bundles can be simple text files - one per locale (language and region) that contain translations for each text element.

For example three resource bundles:

ApplicationLabels_en.properties:
usernameLabel=User Name

ApplicationLabels_nl.properties:
usernameLabel=Gebruikersnaam

ApplicationLabels_ca_fa.properties:
usernameLabel=nom d'utilisateur

Boilerplate text elements should not be hard coded in the pages; instead, references should be used to properties in the resource bundles, such as:

```
<af:inputText label="#{msg.usernameLabel}:"
              value="#{user.username}" />
```

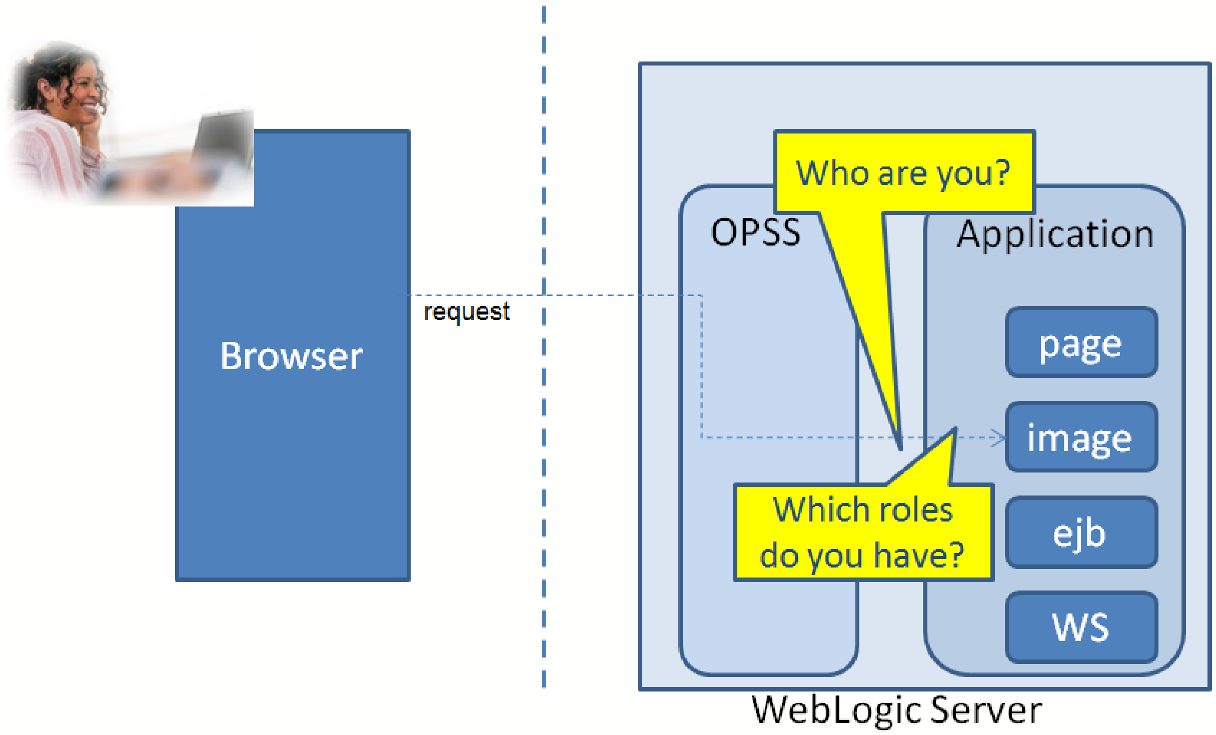
The use of msg in the label attribute above is interpreted by the JSF framework as a reference to a resource bundle entry that needs to be localized.

At run time, the framework will look at the desired language locale for the current session - based on the user preference or the browser language setting - and retrieve the value for the username input field from the appropriate resource bundle property file, using the key usernameLabel. The relevant translation is then injected into the inputText component. Without any knowledge of translation and internationalization or even the mere existence of the resource bundles, the application is injected with the correct localized value for each boilerplate text element. You would not tell from the snippet above, but at runtime the component can render with different injected translated prompt values *username, gebruikersnaam* or *nom d'utilisateur*.

Injection of User identity and Role information

The Oracle Platform Security Service (OPSS) are another example of an injection into Fusion based applications. ADF applications as well as other applications running on WebLogic Server (WLS) typically have a need to know the identity of the current user as well as the roles that were granted to the user. Through OPSS, this information is determined by the platform and made available in a generic way to applications. When the user submits a request to the server, WebLogic Server will intercept the request, check it for user credentials, challenge the user to provide these credentials if they are not yet available and next continue with authentication and authorization.

The user credentials are processed by one or more security providers that the WLS administrators may have configured for the domain. These providers can use an LDAP or Active Directory, a SAML identity token, a custom table with user identities or even custom code that processes the credentials in any way it sees fit. The result of the work of these providers is two-fold: the identity of the user is established - along with additional user details retrieved from the identity store - and all roles granted to the user are associated with the user object. OPSS is the facility on WLS that injects this information into the applications running on the JEE server.

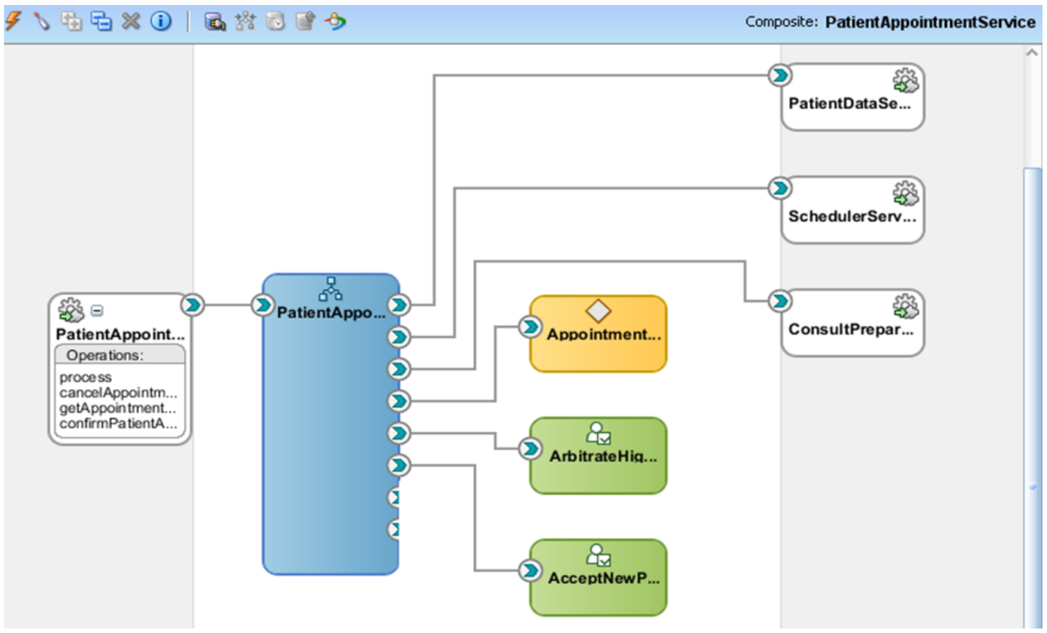


Note: one could argue that the Oracle RDBMS, through USER and sys_context values such as client_identifier also performs a kind of dependency injection. In general, any information that is always available from the 'generic infrastructure' can be considered a form of injection - as the information is made available to software components, without the component explicitly having to look for it.

SCA References in SOA Composite Applications

When (web) services are published by the SOA Suite 11g, these are the publicly exposed 'tip of the iceberg' of SOA Composite applications - very much like the specification of PL/SQL packages. SOA Composite applications provide these services by executing one or more service components. These components are for example BPEL processes, Business Rules, BPM processes, Human Tasks or Mediators. Each component exposes a service interface through which it can be invoked by other components in the composite application.

Take a look at the Composite application PatientAppointmentService that is shown in the next illustration. The publicly exposes service interface for this composite is also called PatientAppointmentService with four operations to request, cancel or confirm an appointment at the hospital or to retrieve the appointment details. The application contains one Business Rule component and two Human Task components - the orange and green boxes. These three components publish an interface - in terms of a WSDL or service definition - through which other components can invoke them. The blue box represents the BPEL process PatientAppointmentProcess that is wired to the business rule and the two human tasks. The consequence at run time is that the BPEL process calls these three components.



All of the above is based on the SCA specification - Service Component Architecture. And dependency injection is also at the core of SCA:

1. service components do not explicitly invoke other components.
2. service components indicate their need for injection of another component by specifying a reference element (an XML specification) that describes the interface that this other component should implement
3. the composite developer (or assembler) ensures that all references for each component are wired to service components that implement the indicated interface; in the example: the BPEL process has indicated a dependency on an interface intended for deciding whether an appointment has high, medium or low priority. The composite developer injects the AppointmentPriorityRule (the orange component) into the BPEL component by creating the wire between the two. Note that a different implementation (as a Human Task or Java component for example) of the interface stipulated by the blue BPEL component can be wired or injected at any moment
4. some of the references from the composite application cannot be satisfied within the application. For example: the BPEL component requires data about the Patient that has requested the appointment. This data is not available inside the application, it needs to be retrieved from somewhere outside the composite. In this case, the reference from the BPEL component is published as a reference from the entire application: PatientDataService. The composite indicates that when it is deployed, it should be into an environment that provides access to a service that implements the PatientDataService interface. Upon deployment, that implementation should be injected into the composite application. Once more, a software

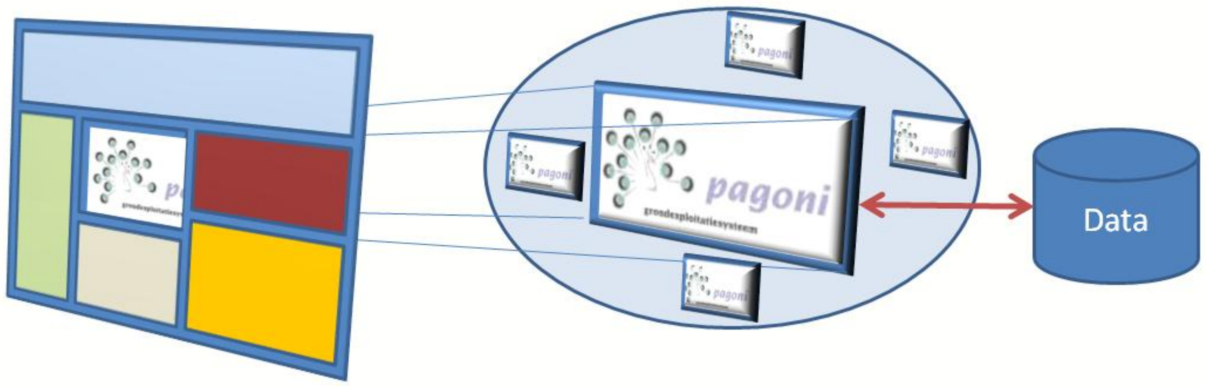
component relies on another component whose existence and location it knew nothing about beforehand.

Note that the public reference from a composite application is similar to an electric appliance that comes with a power plug: the power plug indicates the dependency of the appliance on an external service that provides electrical power. It is up to the deployer of the appliance to inject this dependency by wiring the appliance to the service provider: the power outlet. The service contract in this case specifies the required voltage and the shape of the power socket that the appliance expects.

Run-time Portlet Injection through Oracle Composer

One final example of dependency injection, slightly different from the previous examples, as this time it is the end user who does the injection. Pages in ADF and WebCenter applications can leverage the capabilities of Composer to edit page definitions at run time. This means that privileged users can go into edit mode for a page and make changes to the page - by moving components around, changing the properties set on components and also through the addition of new components in the page.

One such component that can be added is a Portlet (reference). Portlets are typically remote user interfaces that can be embedded in a portlet-enabled web page. When the page that has been edited to include a portlet is rendered, the application will call the remote portlet on the url specified by the end user who edited the page. The remote portlet - the Pagoni portlet in the figure - returns the HTML output that the application should include in the page.



The application's output will be different from what the developers originally created, in a way probably not foreseen at design time, through the injection by the end user of the dependency on the remote portlet. For this to be possible, the developers need to Composer-enable the page and include customizable areas in the page into which such Portlet references can be injected.

Conclusion

This article discussed Dependency Injection - a powerful design pattern that helps to prevent tight coupling between components that may collaborate at run time. Application of the dependency injection design pattern helps to foster reuse, increase agility and as a bonus makes it easier to perform unit tests on components that cannot really be isolated because of their dependencies.

Dependency Injection is intrinsically part of the Fusion Middleware stack. For example, there are many places in Fusion applications where the runtime infrastructure injects values into the applications running on it. More importantly: using dependency injection is a native element in some of the core technology pieces of the FMW stack. For example: in JavaServer Faces we use EL expressions that provide UI components with the link to beans that will provide them at run time with the values and references they need. In SOA Composite applications, components stipulate their dependencies and are wired to other components to satisfy the dependencies.

Knowing about dependency injection is useful in order to recognize and understand the pattern when you encounter it in Fusion Middleware applications. Additionally, you may be inspired to apply the idea of dependency injection in other areas as well. For example: prevent hard coding - make values settable! Also think about call outs from your programs - should they be hard wired or can they be dynamically configurable? It is always a good idea to think about dependencies between software components and strive for reducing the dependencies where this may enhance the reuse potential and the run time agility of components.

Lucas Jellema

AMIS is an Oracle Certified Advantage Partner with SOA and Database specializations, founded in 1991 and located in The Netherlands. AMIS are friends of the Oracle community, for example through frequent presentations at the ODTUG Kaleidoscope, OPP and Oracle Open World conferences and its popular technology weblog: http://technology.amis.nl/blog. AMIS staff were awarded four Oracle ACE Director and two Oracle ACE nominations in recent years. This edition of the column AMIS on Fusion was written by Lucas Jellema, Oracle ACE Director and CTO of AMIS and author of Oracle SOA Suite 11g Handbook (Oracle Press, September 2010). He can be contacted at lucas.jellema@amis.nl