Madura Perspectives Manager

User Guide

Document Revision: Revision: 1

Document Revision: Revision: 1

Table of Contents

1. Change Log

Author	Date	Comment
rogerparkinson	2016-11-18	fixed the demo URL
rogerparkinson	2016-11-16	updated the demo address
rogerparkinson	2016-11-16	Moved to Java 1.8.
rogerparkinson	2016-03-29	Added demo reference to docs
rogerparkinson	2016-03-25	Tidied links and docs
rogerparkinson	2016-01-21	Added detail on scope
rogerparkinson	2016-01-21	Tidied references and updated release notes
rogerparkinson	2016-01-21	Expanded docs to include BMI and blackboard
rogerparkinson	2015-05-10	Release notes for 2.4.0
rogerparkinson	2015-04-09	minor edits
rogerparkinson	2015-04-06	Clarified description of the bundles dir
rogerparkinson	2015-04-03	fixed up schema references
rogerparkinson	2015-04-03	initial commit of new structure

2. References

[1] Madura Objects
[2] Madura Rules
[3] MaduraBundles
[4] Vaadin
[5] Spring Framework
[6] madura-perspectives
[7] madura-perspectives-online

3. What is this?

I'll start with an example. Say you need an intranet application which presents your staff with various functions they need to do.

These functions might be:

- Recording and updating trouble tickets
- Entering their time sheets
- Looking at the current newsletter
- Viewing your product catalogue
- Maintaining customer details

We call these separate functions *perspectives*. They are delivered by sub-applications that are plugged into the main application, the manager. They can contribute various UI elements such as menu items and buttons. The manager decides which perspective's components should be visible at any time. This means the user can flick between various perspective-based applications while remaining in the same main (manager) application.

This has the following advantages over deploying the applications separately:

- Packaging the application as a Madura Bundle[3] is simpler than packaging it as a web application.
- The manager handles common functions such as login, bypassing the login if this has already been handled. The perspectives only need to assume they are already logged in.
- A blackboard system allows the different applications to communicate with each other if necessary, though they do most, or all, of their operations in isolation.
- New perspective applications can be added or removed dynamically to the manager without restarting the application environment. In flight updates of the perspective applications is supported. New logins see the newly added applications, otherwise the list of applications remains stable.
- Because there is only one CSS definition across all the application there is automatically a consistency in the look of all the perspectives.

The Manager uses Vaadin[4] and the perspective applications must use Vaadin to deliver their UI. There is no restriction on what parts of Vaadin the perspective applications can use.

4. Running the Demo

There is an on-line demo of the project here: [7]. If you use that you can skip building it and running it locally.

The first thing you need to do is check out the project from [6].

To build use: mvn install

To run with Jetty use: mvn jetty:run and open http://localhost:8080/. You can also run them with Eclipse WTP. We tested with Tomcat 7.

To login the user/password is always admin/admin, you can also use user/user

For the purposes of keeping the demo setup simple the bundles it uses are embedded in the war file, that means you don't have to mess about telling your application server what directory to sweep to find them. The configuration to support the more flexible use is in the `applicationContext.xml` file, commented out, near the bottom. We will look at those options in *3*

Figure (1) Madura Perspective Demo

What *Figure (1)* shows is the initial display when a user logs into the main application.

Worth noticing at this point is that there is not much there, but there is a list of sub applications on the left hand side. There is also a small menu and, over on the right, the 'Logged in as: admin' indicates who logged in.

So this is what the main application's UI looks like, but this is just a demo. The details of this UI can be customised to any requirements. For example if you want the list of sub-applications to appear somewhere else, perhaps over on the right, and as a drop down list you just change the code in the main layout. You probably do not want the Madura logo dominating the upper left hand side either, though you're welcome to put it there. Also you can adjust the CSS definitions that Vaadin uses to change colours and fonts etc. Vaadin themes are, of course, supported so you can select different themes for different users if you want. The selected theme is propagated to the sub-applications as well, ensuring the user has a consistent interface.

If the user selects the `User` option this loads the sub-application and changes the screen.

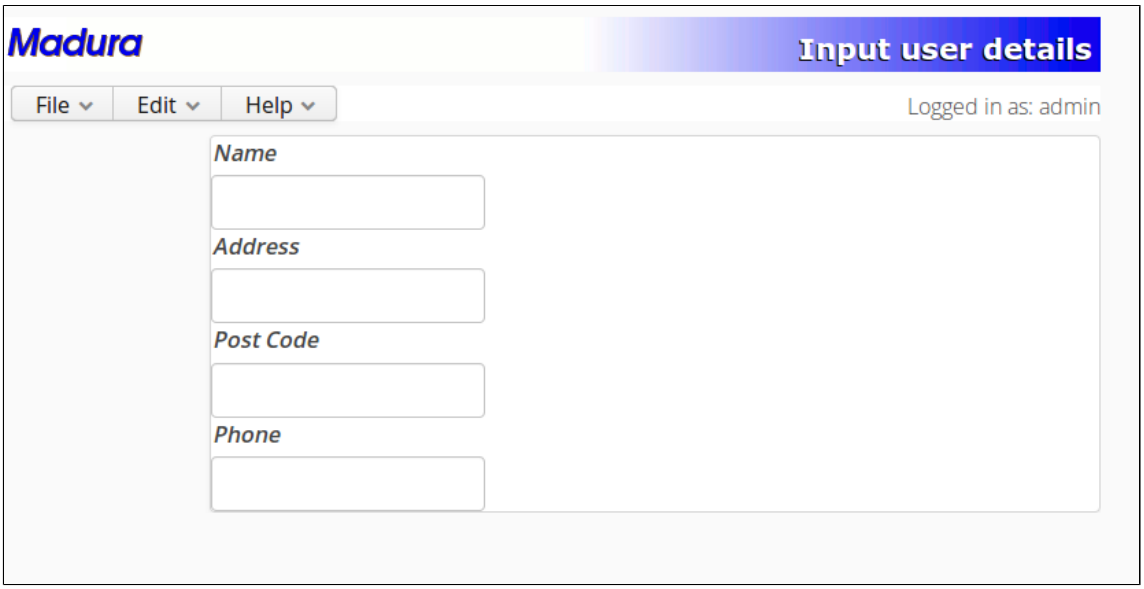

Figure (2) User Details

The User Details sub application is a very simple pure Vaadin form mapped to an object. It accepts information about a user but it does nothing much with it (this is only a demo). But it shows where the sub application UI appears on the screen. Notice that it retains the same theme as the main application. Notice the the title on the upper right has changed, it shows the name of the current sub application. If we now click the `pizza` option we see the `Pizza Order` UI.

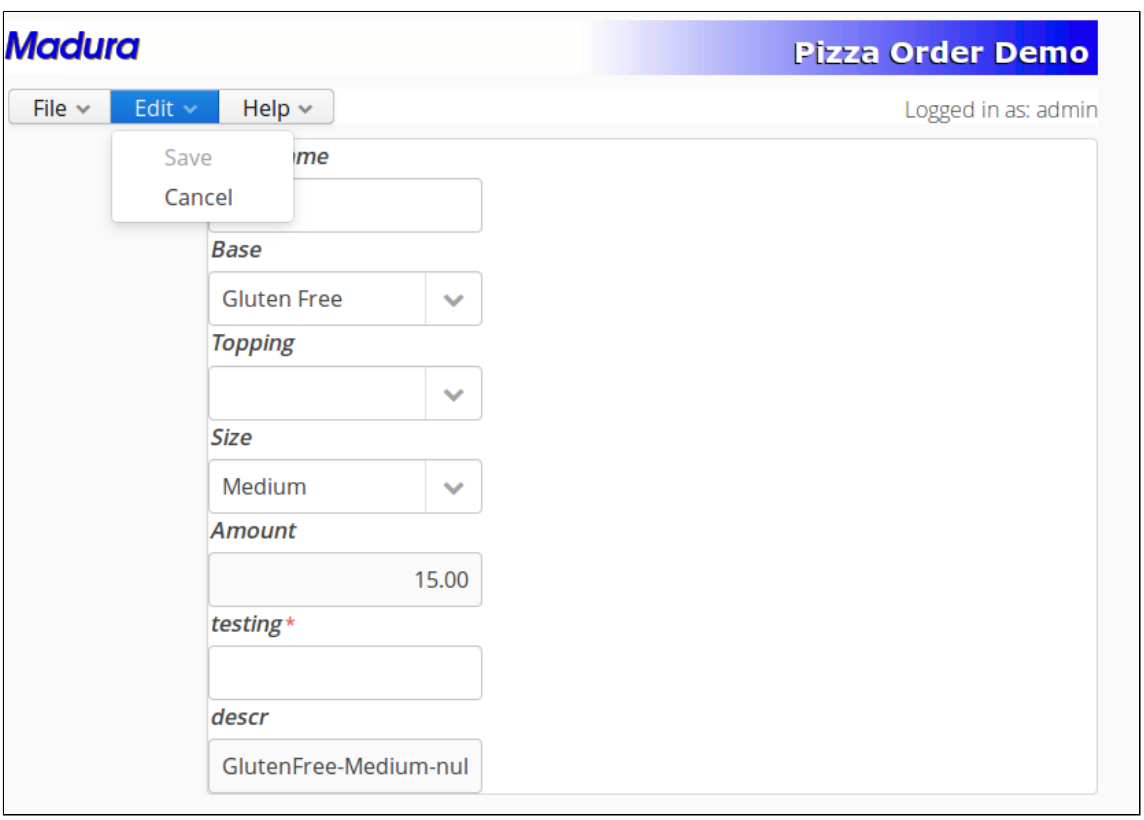

Figure (3) Pizza Order

This sub application uses Madura Objects[1] and Madura Rules[2] to drive the UI, so is a bit more interesting. It uses the same objects and rules as the pizza order demo. The point, though, is that this kind of functionality can be added to a sub application when we want. You might also notice that

the menu has an extra entry. This was contributed by the Pizza Order sub application. Menu items appear and disappear as different sub applications are selected.

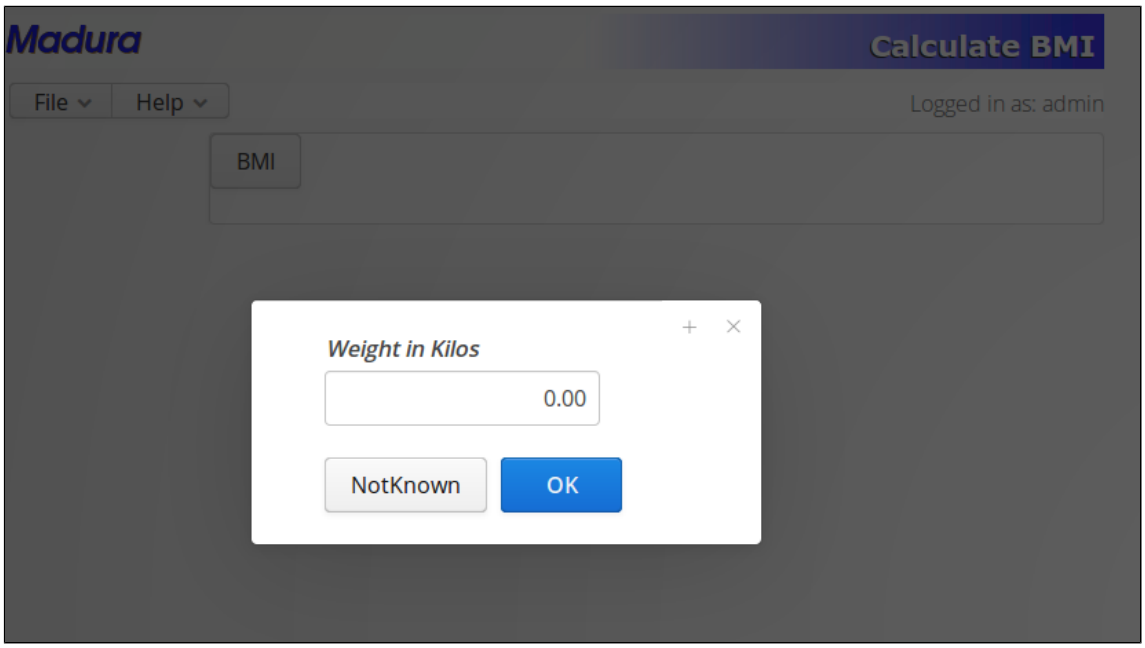

Figure (4) BMI

This is another sub application that uses Madura Objects and Madura Rules to drive the UI, in particular it uses the directed questioning feature of Madura Rules to drive single field prompts for the data points needed. The rules drive what prompts are used and, apart from the BMI button to kick it off it needs no actual UI design.

Now let's take a look at what these sub applications really are.

Each sub application is a Madura Bundle, which means it is a jar file with some extra bits, including a local Spring[5] beans configuration. Each sub application contributes:

- A name. This is used to create the link on the left.
- A description. This is what you see displayed on the upper right. It is also used to make a tooltip on the link.
- A UI, which is the bulk of the application.

The sub applications can be added to the main application dynamically. Once a user has established a session, ie logged in, then they will not see the list of applications change, but they will get the latest list of available applications. This means you can dynamically add sub applications to the main application while it is still running. You can add new versions of an existing sub application in the same way. All you need to do is copy the jar file to the directory Madura Bundles is monitoring.

The applications use the same theme information as the main application, so a change of theme will automatically propagate to the sub applications.

Security information, specifically permissions established at login time, is also available across all sub applications.

And the sub applications can influence each other through a blackboard.

The blackboard is a publish-and-subscribe system where one application publishes something to the blackboard and other applications can react to the change or ignore it.

5. Configuring for Production

As mentioned earlier this demo has the bundles embedded in the war file, which is fine for a demo but it makes the whole concept a bit weak.

A production configuration would dispense with the bundles embedded in the war file and use an external directory. To configure this you need to tell your application server where that directory is.

```
<jee:jndi-lookup id="bundlesDir" jndi-name="java:/comp/env/BundlesDir"
 expected-type="java.lang.String" />
<bean id="bundleManager"
 class="nz.co.senanque.madura.bundle.BundleManagerImpl">
 <property name="directory" ref="bundlesDir"/>
 <property name="time" value="10000"/>
</bean>
```

You will find the above configuration in `applicationContext.xml` just above the bundleManager that is used by default, which you should remove, of course, because you only want one of them. The only difference between the two is that this one has the directory and time properties set. Just above it a JNDI name is defined. Now, you could simply hard code your directory in there and dispense with the JNDI name but in a production system you probably want to use JNDI so that you can change it without having to rebuild the application.

The next step is to tell your application server what value to give that JNDI name. This depends on your application server. For Tomcat you can just edit it into your `context.xml` file like this:

```
<Environment name="BundlesDir" value="MY_DIRECTORY/bundles"
 type="java.lang.String" override="true"/>
```

Finally you want to actually add some bundles to that directory. There are four bundle projects you can use right away, these are the ones in the demo:

- madura-perspectives-name
- madura-perspectives-nameaddress
- madura-perspectives-pizzaorder
- madura-perspectives-bmi

These are all maven sub-projects and they are automatically added to the `WEB-INF/bundles` in the madura-perspectives-manager project. You can configure a sweep directory as described above and deploy your own bundle(s) to that. As long as you keep changing your bundle's version you can change, build and deploy it without restarting the main application, though you do have to log out and back in to see the new bundle you added.

6. What Does a Sub Application look like?

A sub application is a jar file, more specifically it is a Madura Bundle[3] which means it has some specific items in the jar file. The first thing to look at is the `MANIFEST.MF` file. Apart from the usual entries this has:

```
Bundle-Name: Name
Bundle-Version: 3.0.0
Bundle-Context: name-spring.xml
Bundle-Activator: nz.co.senanque.madura.bundle.BundleRootImpl
Bundle-Description: user.details
```

These are the usual entries needed for Madura Bundles. Two things to point out especially are that the `Bundle-Context` points to a Spring context file in the top directory, and the `Bundle-Description` refers to an entry in the resource bundle and the appropriate language resource will be used to turn it into a displayable description string and used in the UI.

The context file should look omething like this:

```
<bean id="PropertySourcesPlaceholderConfigurer"
 class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer"/
>

<bean id="bundleName"
 class="nz.co.senanque.madura.bundle.StringWrapperImpl">
 <constructor-arg value="${bundle.name}"/>
</bean>
<bean id="SubApplication"
 class="nz.co.senanque.perspectiveslibrary.SubApplicationimpl">
 <property name="version" value="${Bundle-Version}"/>
 <property name="caption" value="${Bundle-Name}"/>
 <property name="description" value="${Bundle-Description}"/>
 <property name="appFactory">
  <bean class="nz.co.senanque.bundle2.AppFactoryImpl">
  </bean>
 </property>
 <property name="messageSource">
  <bean
 class="org.springframework.context.support.ResourceBundleMessageSource">
   <property name="basenames">
    <list>
     <value>messages</value>
    </list>
   </property>
  </bean>
 </property>
</bean>
```

Almost all of this is creating the `SubApplication` bean and this is just about all boilerplate code. The one thing specific to this sub application is the `appFactory` field value, the class name of the appFactory for this application. We also defined a bean to hold the resource bundle. That appFactory class implements `nz.co.senanque.perspectiveslibrary.AppFactory`. The simplest looks like this:

```
public class AppFactoryImpl implements AppFactory {

 @Override
 public App createApp(Blackboard blackboard)
```

```
 {
  App ret = new App();
  final Layout layout = new Layout();
  layout.setBlackboard(blackboard);
  ret.setComponentContainer(layout);
  return ret;
 }
}
```

Its main job is to create the `layout` which is always a Vaadin component, usually a `VerticalLayout`, but if you need to manipulate the main menu or start a Madura session you do it here. The `layout` creates all the visible controls and adds the blackboard listeners to them (if any).

To add menu items you just create a `com.vaadin.ui.MenuBar` and add whatever menu items you want to that. Then use the `App.setMenuBar()` to set your menu bar into the returned app. You only need the *extra* entries in your menu bar, you don't need to re-add the standard ones.

7. Blackboard

The blackboard mechanism is a simple publish-subscribe mechanism to allow one sub application to publish something and the others to subscribe to that message, or to ignore it. You can see this working by running the demo and picking the File/User option on the menu. Enter a value into the Name field and then pick File/Name to go to the Name sub applicaiton. You should see the value you typed into the other sub application appear in this one. So how did that happen?

First notice that in both sub applications the `layout` implements a `setBlackboard()` method. In the User sub application there is a `TextChangeListener` on the name field. When the value changes it calls the blackboard to publish the new value with an associated name 'userName'. The UserName sub application adds a listener to the blackboard and when that listener is called it updates its own name field.

8. Bean Scopes

If you looked at the VaadinSupport demos you have seen the permission manager bean defined in the XML context file with scope 'vaadin-ui'. There is a note in the docs for that project pointing out that this could be done just as easily as a `@Bean` like this:

```
@Bean(name="permissionManager")
@UIScope
public PermissionManager getPermissionManager() {
 PermissionManagerImpl ret = new PermissionManagerImpl();
 return ret;
 }
```

But in this project we do something different. The permission manager bean is defined like this:

```
@Bean(name="permissionManager")
@Scope(value="vaadin-ui", proxyMode = ScopedProxyMode.TARGET_CLASS)
@BundleExport
public PermissionManager getPermissionManager() {
 PermissionManagerImpl ret = new PermissionManagerImpl();
 return ret;
}
```

Why the difference? In the earlier demos the permission manager bean needed to be instantiated per session and the `@UIScope` and 'vaadin-ui' settings achieve that. In Perspectives we need the same per session instantiation but we also need to export the permission manager bean to the sub applications so that they can query it for permissions and possibly the current user name. That explains the `@BundleExport`, but why has `@UIScope` changed to that more complex `@Scope` setting? The answer is that the `@Scope` setting is almost the same as `@UIScope`. Both effectively declare a scope of 'vaadin-ui' for this bean, which ensures it is a session level bean. But a simple `@UIScope` for not tell Spring enough about how we need this bean proxied, so we've added the proxyMode setting. It is vital the bean is proxied so that it can be exported correctly. But this only applies to session beans. Singletons do not need special proxying, nor to they need any scope setting.

The proxying is needed because it wraps a mechanism for finding the bean that applies to this session. The sub application calls the bean and the proxy manages locating the right bean and then calls it on behalf of the calling code.

The same applies to the hints bean which is also a session bean exported to the sub applications.

If you do want to create `@UIScope` beans in a sub application, as opposed to creating them in the manager and exporting them, you can. These are switched internally to 'bundle' scope. That means they are instantiated once per session/bundle, ensuring that two beans in two different bundles with the same name are not confused even if they belong to the same session.

A. License

The code specific to madura-perspectives is licensed under the Apache License 2.0 .

The dependent products have compatible licenses specified in their pom files. Madura Rules (optional) has a dual license to cover projects that do not qualify for the Apache License.

B. Release Notes

3.2.1

Documentation changes.

Updated dependency version for madura-vaadin.

3.2.0

Moved to Java 1.8.

3.1.0

Updated docs and deployed demo.

3.0.0

Updated underlying libraries and migrated to using FieldGroups rather than Forms in the samples.

2.4.0

Implemented bundle scope (from new version of Madura Bundles)

Reworked to use the new configuration mechanisms and Vaadin 7

2.3.1

Restructured the layout to include parent and sub-projects.

Added demo script link to menu.

Revised documentation.

2.3

Added internal bundles.

Added mechanism to specify the bundles dir in JNDI.

Fixed some I18n issues.

Switched to maven build.

2.2

Built for Java 1.7.

2.1

Updated for later Vaadin additions

2.0

Modifications to allow demo in Cloud Foundry

Initial release