Madura Vaadin Support

User Guide

Document Revision: Revision: 1

Table of Contents

1. Change Log

Author	Date	Comment
rogerparkinson	2017-02-02	release notes
rogerparkinson	2017-01-12	documented the applicationVersion mechanism
rogerparkinson	2016-11-18	fixed reference to apk file location
rogerparkinson	2016-11-17	setting up for new version
rogerparkinson	2016-11-15	updated version to 3.2.0
rogerparkinson	2016-03-29	several deployment issues
rogerparkinson	2016-03-29	Added references to online demos
rogerparkinson	2016-03-25	removed reference to vaadin addon because the current format doesn't work The plan is to implement this in a later version.
rogerparkinson	2016-03-24	more tidying of readme files
rogerparkinson	2016-03-23	Revised documentation with better examples
rogerparkinson	2016-01-21	Added more detail on permission manager.
rogerparkinson	2015-12-26	minor syntax fixes
rogerparkinson	2015-12-14	Added logout to MaduraSessionManager
rogerparkinson	2015-12-13	Added images to docs
rogerparkinson	2015-12-11	typo fix
rogerparkinson	2015-12-10	Added documentation on the help url
rogerparkinson	2015-12-09	checked and fixed the references
rogerparkinson	2015-12-08	Heading changes and typo fixing
rogerparkinson	2015-12-08	Documented Phonegap build
rogerparkinson	2015-12-07	documented madura-touchkit-theme
rogerparkinson	2015-12-06	reworked mobile docs
rogerparkinson	2015-11-30	Documented simpler way of configuring madura-login
rogerparkinson	2015-11-24	Added auto-login option
rogerparkinson	2015-11-24	Added more detail on the address book demo
rogerparkinson	2015-11-22	Major rework of docs and examples
rogerparkinson	2015-09-27	Initial commit

2. References

[1] Spring Framework

[2] Vaadin

[3] JPAContainer

[4] madura-login

[5] MaduraObjects

[6] MaduraRules

[7] Madura Rules Demo

[8] vaadin-touchkit

[9] vaadin-addon

[10] PhoneGap-tutorial

[11] PhoneGap

[12] PhoneGap-Build

[13] Apache Licence 2.0

[14] Madura Rules Online

[15] Madura Mobile Online

[16] Madura Mobile Demo.apk

3. Purpose

`madura-vaadin` ties together all the back-end Madura projects (Madura Objects[5] and Madura Rules[6] etc), and delivers them with a Vaadin UI[2]. So it is worth taking a brief moment to review what those back-ends do:

- Madura Objects builds your domain objects as annotated POJOs from an XSD file (using JAXB). The resulting objects behave just like POJOs except, when configured with the Madura Objects validation engine, they self validate as well as maintaining field metadata such as choice lists, permissions, labels etc.
- Madura Rules plugs into the Madura Objects validation engine to support a rules/constraints based environment that does cross-field validation as well as deriving new values (eg total of the invoice lines on this invoice). It does 'truth maintenance' which means when the data changes rules might be 'unfired', keeping the derived data always 'true'. The rules can also operate on metadata which means they can change the list of valid choices on a choice field, make a field visible or read-only etc.

This project wraps those tools to make them easy to use in a Vaadin application. The result delivers highly dynamic applications with very little application code. For example:

- You can pass a POJO to a generic form which builds display fields for all the POJO fields. Each field is automatically validated according to specifications in the XSD (eg field length, numeric range checks) and error messages delivered where necessary. Required fields (again, as specfied in the XSD) are noted and the submit button is disabled until they are filled in, it is also disabled if there is an error. The generic form can accept a list of fields so you can specify which ones you want, you don't have to have them all.
- All field captions are fetched from the relevant XSD label and translated to the current locale.
- All validations on all fields are handled automatically. That means range checks, regex checks etc, but it also means rules based validation if you have configured it. The application code is unaware any validation is active.
- Error handling and delivery of any messages is synched with Vaadin's error message delivery and messages are locale translated.
- Permissions are enforced. If this user only has read-only permission on a field it will be rendered but disabled. If they do not have read permission it will not be rendered. If no permissions are specified on the field it will be rendered normally, so you only add permissions to fields you care about.
- Fields marked Secret in the XSD are rendered as Password style fields.
- Default values, if specified in the XSD, will be loaded in the obvious way.
- Required fields are indicated as such using Vaadin defaults.
- The submit buttons disable themselves until all the required fields are all filled in. They also disable if there are any errors in the fields they are watching. Menu items are treated much the same as buttons, so a submit menu item will be disabled until the form is complete.
- Any appropriate Vaadin controls can be used, not just the standard ones. All the usual theme features available in ordinary Vaadin applications are still in place. The main change is the use of a specialised FieldFactory which can be used on any form or field group, though this is optional anyway.

In addition, if you have configured a rules plugin, such as Madura Rules, into your validation engine:

- The validation can include cross field validation.
- Choice lists dynamically change as the available choices change.
- Fields may change to/from read-only or invisible or required as rules fire.
- One or more buttons may be tied to boolean fields that are, in turn, controlled by rules. These buttons become enabled or disabled depending on the current value of the boolean.
- Labels and read-only fields may contain data derived from rules. This automatically updates as the rules change the data.

Still with us? Good. In addition to all that we added some extra things that come in handy:

- Support for Mobile applications using Touchkit[8] and PhoneGap[11].
- An extension of the Vaadin JPAContainer[3] which supports `@Transactional` better. It also supports nice popup edit forms for each row. The edit forms use Madura Objects.
- A login filter that pops a login dialog if this user is not yet logged in. We use this for demos rather than production, but it could be customised for production.

The details of how to use all this are best explained by examples.

- madura-vaadin-demo: This is a basic demo of Madura working with Vaadin. Specifically Madura Objects without Madura Rules.
- madura-rules-demo: This is the full demo that shows Madura Objects and Madura Rules working with Vaadin. See [14] for the on-line demo.
- madura-address-book: Demonstrates the extended JPA container and a pop-up row editor which has Madura Objects backing the fields.
- madura-mobile-demo: Demonstrates an application that presents both a desktop and mobile UI, both backed by Madura Objects and Madura Rules. See [15] for the on-line demo.

These demos form the basis of this documentation and we will refer to them extensively.

4. Common Techniques

All the demos use Vaadin 7, Spring[1], Madura and Maven. We assume you are reasonably familiar with all of these.

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

The login process is managed by the `madura-login`*8* project. This provides a simple login facility suitable for demos simply by making it a maven dependency and a little configuration.

As is usual in Madura Objects we have some files in the resources directory. These are mostly things like `logback.xml` as well as files covered in Madura Objects or Madura Rules documentation. But because they are so important we will mention the XSD files which define the objects. There is an entry in the maven `pom.xml` file that invokes `madura-objects-maven-plugin` which is just a jacket for JAXB that saves you having to configure JAXB for Madura. That generates annotated Java files from the XSD and puts them into the `generated-sources/xjc` directory. You never edit those Java file, you change the XSD and regenerate.

Vaadin also has a maven plugin: `vaadin-maven-plugin` which generates the widgetsets.

Each demo makes use of Spring Configuration in preference to Spring XML and we also make use of the Vaadin-Spring Addon. However it is simpler to allow one XML component because of the way the defaults work. So each project has an `applicationContext.xml` file in the `WEB-INF` directory. It looks like this (excluding the headers etc for brevity):

```
...
<context:component-scan base-package="nz.co.senanque.vaadindemo,
 nz.co.senanque.login" />
<bean id="permissionManager"
 class="nz.co.senanque.vaadin.permissionmanager.PermissionManagerImpl"
 scope="vaadin-ui"/>
<bean id="messageSource"
 class="nz.co.senanque.resourceloader.ResourceBundleMessageSourceExt">
 <property name="basenames">
  <list>
   <value>ApplicationMessages</value>
  </list>
 </property>
</bean>
</bean>
```

This does three things. It tells Spring to scan the application package for component beans, and the login package as well because it has components we want too. It defines the `permissionManager` bean which is needed to hold this user's permissions. There is more about this in *8*. The context file also creates the message source. `ApplicationMessages` are the messages for the current demo. `ResourceBundleMessageSourceExt` is just like Spring's `ResourceBundleMessageSource` except it can go find the properties files in various jar files by itself.

But most of the Spring configuration is in the code. Each demo has a class that extends `com.vaadin.ui.UI` called either `MyUI` or `AddressBookUI` or similar. This contains the configuration.

```
@Theme("mytheme")
@Title("Madura Vaadin Demo")
@Widgetset("com.vaadin.DefaultWidgetSet")
@SpringUI
public class MyUI extends UI {

    @Autowired private MaduraSessionManager m_maduraSessionManager;
    @Autowired private DefaultView m_defaultView;

    @WebServlet(name = "MyUIServlet", urlPatterns = "/*", asyncSupported =
 true)
    public static class MyUIServlet extends SpringVaadinServlet {

  private static final long serialVersionUID = 1L;
    }

    @WebListener
    public static class MyContextLoaderListener extends
 ContextLoaderListener {
     // This causes the applicationContext.xml context file to be loaded
     // per session.
    }
...
```

The first three annotations are normal Spring-Vaadin things that ensure this class is loaded with the right theme etc. You can also see a static class that declares a servlet as well as a web listener. The listener is what loads the `applicationContext.xml` you saw above.

There are also two auto wirings, one for the current MaduraSessionManager and one for a local class which defines most of the UI. There is often more than one of these classes.

After that there is a static class that defines the configuration. It is worth noting that the above section is created and wired *per session* whereas this following section is *per application* ie singletons, except for those annotated `UIScope` which are session beans.

```
...
@Configuration
@EnableVaadin
@ComponentScan(basePackages = {
  "nz.co.senanque.vaadin",
  "nz.co.senanque.validationengine"})
@PropertySource("classpath:config.properties")
public static class MyConfiguration {

 public MyConfiguration() {
 }
    // needed for @PropertySource
    @Bean
    public static PropertySourcesPlaceholderConfigurer
 propertyConfigInDev() {
     return new PropertySourcesPlaceholderConfigurer();
    }
    @Bean(name="hints")
    @UIScope
    public Hints getHints() {
     return new HintsImpl();
    }
}
...
```

The things to note here are that there are more packages to scan, namely the validation engine and the components of the madura-vaadin project. We also specify a properties file which injects values into these scanned components. You will see the `config.properties` file in the resources directory and it holds those values. The `hints` bean is defined here rather than scanned so that you can replace it with your own implementation. The hints bean helps the field factory decide what kind of field to create in various circumstances. The one used here works just fine but you might want something different.

The `config.properties` only needs to know what package the generated Java classes are in, like this:

```
nz.co.senanque.validationengine.metadata.AnnotationsMetadataFactory.packages=nz.co.sen
```

Next up is the `init` method which is called when the user session is created.

```
...
    protected void init(VaadinRequest vaadinRequest) {

     MessageSourceAccessor messageSourceAccessor= new
 MessageSourceAccessor(m_maduraSessionManager.getMessageSource());
     final String logout = messageSourceAccessor.getMessage("logout");
...
```

This is fairly ordinary Spring code that uses a `MessageSourceAccessor` to translate the logout string. It is interesting because it shows that for various headings or captions of components that are not controlled by Madura we still need to call a `MessageSourceAccessor` for I18n, but most of the time we don't.

The `DefaultView` class was mentioned above. It is auto wired into the `MyUI` class and all of the demos have one or more classes like this, so let's take a brief look at one of them:

```
...
@UIScope
@SpringComponent
public class DefaultView extends VerticalLayout implements
 MessageSourceAware {

 @Autowired private MaduraSessionManager m_maduraSessionManager;
...
```

The two class annotations ensure that Spring loads this as a session component, and the `MaduraSessionManager` is auto wired. The `MaduraSessionManager` is the general purpose API we use to get to Madura Objects so we need it everywhere.

There is a handy mechanism for reporting the version details in here. The component bean `applicationVersion` is created by the Spring scan. It looks in the war file manifest for entries for Implementation-Title, Implementation-Version and Implementation-build and makes up a string of these. You can inject the string anywhere you like eg:

```
...
@Autowired(required=false) @Qualifier("applicationVersion") private String
 m_applicationVersion;
...
```

If there is no manifest it looks for a file called `/ApplicationVersion.properties`. Make sure your war build actually sets the implementation-build entry in the manifest, it is not there by default, and note the capitals.

We use the `applicationVersion` bean in the default login prompt as a tooltip that shows when you mouse over the logo. We also use it in the About boxes. The About boxes here use the `aboutInfo` bean which is also created by the scan. That gets the `applicationVersion` injected. But it does more than that. It scans all the beans for any that implement the `nz.co.senanque.version.Version` interface. For those beans it calls each one to get the version and builds all the bean names and versions into a string suitable for displaying on the about box. This means you can report not only your main version but selected dependencies as well. Typically all you do to provide a version bean is provide a class like this:

```
@Component("MyUniqueComponentName")
```

```
public class LocalVersion extends Version {

 public LocalVersion() {
 }
 public String getVersion() {
  return this.getClass().getPackage().getImplementationVersion();
 }
}
```

Give the bean a unique name, ie replace 'MyUniqueComponentName' and put this in a package that your Spring configuration will scan.

Now we can look inside the first demo.

5. madura-vaadin-demo

The interesting things in this project are in the `nz.co.senanque.vaadindemo.DefaultView`. You saw it briefly in the previous section so you know it is a session dependent bean and it has a `MaduraSessionManager` injected.

It has an `init` method which runs once Spring has instantiated the bean and finished injecting everything. All it does is set up a layout called `panel`.

The `load()` method is passed a `Person` object. It does the following:

- clears the `panel` layout we created in the `init()`.
- makes sure the `Person` is bound to our validation session.
- clears the `panel` layout we created in the `init()`.
- Wrap the `Person` in a `BeanItem`.
- Create a new layout and add it to the `panel`.
- Use the Madura Session Manager to create a Madura Field Group.
- Create the buttons, more detail below.
- Create the fields using `buildAndBind`. Then add them to the form using a loop.

Creating the buttons is done *before* the rest of the fields are created. The code is in the `createActions` method which creates two buttons. The two buttons don't do much (this is a demo, remember). But how they are created is important.

```
...
Button cancel = fieldGroup.createButton("button.cancel", new
 ClickListener(){
...
 }});
...
```

There is not much to this really. It creates a button, translates the name and adds the listener. But we might have used a variation on this:

```
...
Button cancel = fieldGroup.createButton("button.cancel", "ADMIN", new
 ClickListener(){
...
 }});
...
```

In this case we added a permission. If the user does not have this permission the button will be disabled.

The second button is similar:

```
...
Button submit = fieldGroup.createSubmitButton("button.submit", new
 ClickListener(){
...
 }});
...
```

This button is a *submit* button. For us that means all fields in the field group that are flagged as required must have values before the button will be enabled. They also have to be valid. The validation rules and the required flags are specified in the XSD file, not here, so the UI designer does not need to worry about those things. As with the previous button you could add a permission argument. Permissions always trump everything else so even if a user has filled in the fields correctly the submit button will not enable unless they have the permission. If you want to make one of the buttons the default button, ie what happens when the user presses enter, you do this:

```
submit.setClickShortcut( KeyCode.ENTER );
submit.addStyleName( ValoTheme.BUTTON_PRIMARY );
```

The style is optional and we've chosen the one one appropriate to Vaadin's Valo theme. If we were using a different theme we would use a different style here.

After the buttons are created the next thing that happens is:

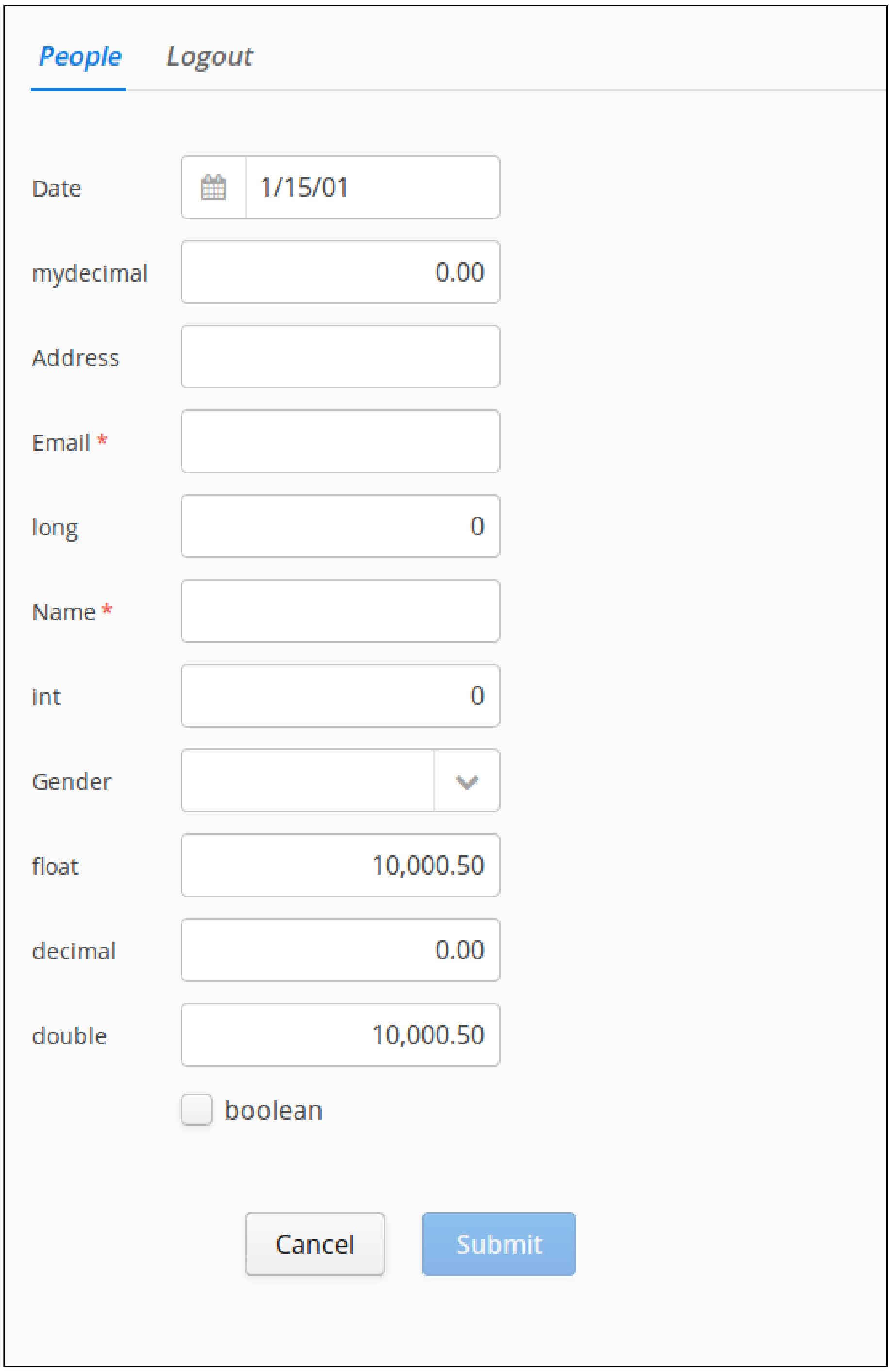
```
Map<String,Field<?>> fields = fieldGroup.buildAndBind(m_fields, beanItem);
```

This calls the field factory to create a field for each of the property ids in the `m_fields` array, binding each one to the `beanitem` object. A short loop just after that adds these fields to the layout created earlier and they end up on the form.

The result is a display with several fields, complete with I18n captions, backed by the Madura Object `person`. The buttons etc are also bound so that Submit button will be disabled if the `person` does not have all the required fields completed. The fields are all wired with validation and the field types vary by data type, so there are three text fields, one drop down which is populated with choices, a date and several numeric text fields.

This is not the only way to achieve this, and you will see alternative approaches in subsequent demos. They are summarised in *6*

The result is in *Figure (1)*



Date	1/15/01
mydecimal	0.00
Address	
Email *	
long	0
Name *	
int	0
Gender	
float	10,000.50
decimal	0.00
double	10,000.50
	☐ boolean

Figure (1) Madura Vaadin Demo

Notice that two of the numeric fields are formatted with commas and periods. This is done according to the formating rules in the XSD. These adapt according to locale. The different field types, including the boolean, are derived from the `hints` bean which you can customise for your own needs.

6. madura-rules-demo

There is an on-line demo of this at [14].

This demo is more complicated, and it uses Madura Rules so there are some configuration differences too. We will look at those first.

The `applicationContext.xml` has an extra package to scan, this is the package we put the generated rules in.

Madura Rules uses a maven plugin to generate Java rules in much the same way Madura Objects generated Java objects from the XSD. When you use Madura Rules you always use Madura Objects as well, so we have the usual XSD file and an entry for it the `pom.xml`. We also have a RUL file in the resources directory. That defines the rules and there is plugin configured in the pom file called `madura-rules-maven-plugin`. That puts the rules Java into `generated-sources/xjc` package `nz.co.senanque.pizzaorder.rules` which is the package specified in `applicationContext.xml`.

The `choices.xml` file is more interesting in this demo because it includes more choice list entries (some aren't actually used), a decision table and a constant. There is more Java code as well, but this is not because we added rules, it is because we added more demo.

As usual there is a `com.vaadin.ui.UI` extension (called `MyUI`). It looks much the same, but we are scanning one more package: `nz.co.senanque.rules` to pick up the rules components. These are the out-of-the-box rules component classes as opposed to the generated rules classes scanned in the `applicationContext.xml` file. Other than that all this class does is create a `TabSheet` on which it places the injected UI components so we will look a those one by one.

An important point to notice here is that apart from adding those directories to be scanned the code is comletely unaware of the rules.

6.1. CustomerView

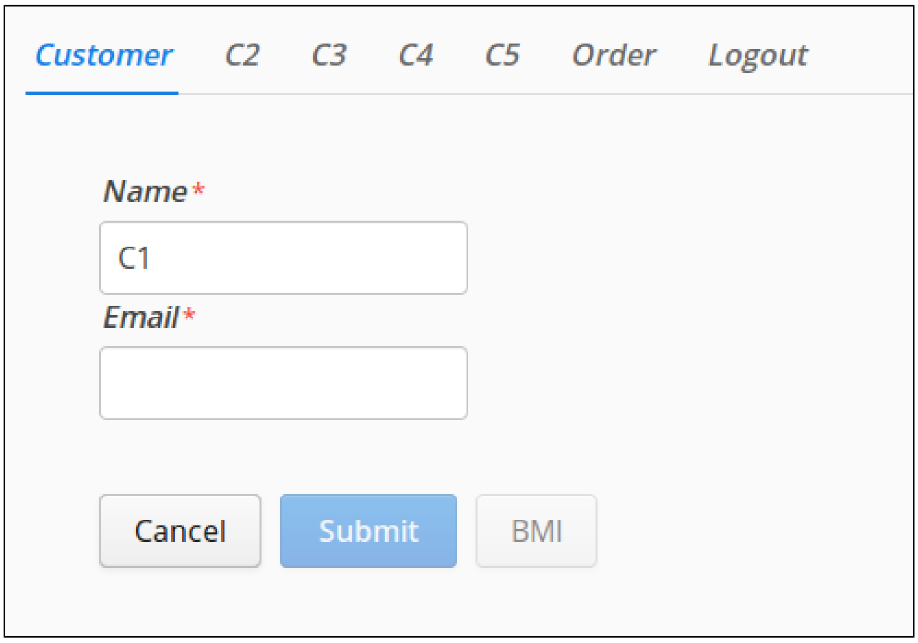


Figure (2) Customer Screen

This is labelled 'Customer' on the UI (unless you are using the French version). It looks a bit like the first demo but with fewer fields and more buttons.

There are 5 variants of this customer UI and you can see the others on tabs labelled C2...C5. They are functionally equivalent but each uses a different coding technique to achieve the same result. The details of the code are covered in . For now you need to know the fields you see are bound to a `Customer` object and will display fields from that object. There are two required fields (marked with red asterisks) on the form. That means they have to be given values, and they have to be correct values. This becomes important in the `email` field which needs to be of the form `xx@xxx`. You should try the demo with different values and see how errors are displayed etc. Notice that the Submit button does not enable until both required fields have valid values. This is the same as the madura-vaadin-demo *3*.

The demo also has a disabled BMI button. This is a different sort of button, created using:

```
Button bmi = customerForm.createFieldButton("button.bmi",
 "dynamic","ADMIN", new ClickListener(){
```

There is a boolean field called `dynamic` in the `Customer` object and this code creates a button that is disabled or enabled depending on whether `dynamic` is false or true. This would not be all that useful if nothing changed the value of `dynamic`, but there is a rule in the `PizzaOrder.rul` file:

```
rule: Customer "dynamic"
{
 if (name == "fred")
 {
  dynamic = true;
 }
}
```

This may be fairly obvious, but if it is not then you can find more details in the Madura Rules documentation. If we enter 'fred' into the `name` field the rules will fire and set `dynamic` to true, and that will cause the button to enable, unless this user doesn't have the ADMIN permission. This example should give you a feel for how the operation of the rules is detached from the UI code. The UI knows about a field called `dynamic`, it does not know about the rule. Similarly the rule knows about `dynamic` but it knows nothing about the UI. Yet a change from the rules propagates to the UI. We will see this a lot.

Just after we created the `MaduraForm` we set the field list into it. This list is turned into fields with captions on a `VerticalLayout`. All it needs after that is an actual object and that is set in the `enter` method.

The `BMI` button has a a listener attached that actually does something:

```
m_oneFieldWindowFactory.createWindow(m_customer,
 "bmi",ValoTheme.BUTTON_PRIMARY);
```

This invokes the directed questioning mode of the rules. What happens is quite complex, but this is all you need to use it. We pass the object, it has to be a Madura Object of course, ie generated from the XSD, the name of a property on the object and, optionally, a style for a submit button. The rules engine will look for rules that output that property and try and work out a value using them. Each time it finds it needs more information it will prompt for it. In this case there is a rule:

```
formula: Customer "BMI"
{
 bmi = weight / (height * height);
}
```

So it asks for weight and height on two different pop-up windows. It can be more complicated than that. Some people like to give these measurements in imperial units, some in metric, and they don't know, say, their height in metres. But that is okay. You can say you don't know and it will ask for feet then inches (two prompts). This feature can be used to simplify complicated input sequences that may need only a few relevant inputs, depending on the context. If the property is already filled in, perhaps the height was obtained earlier, then it knows not to ask for it. This example shows a fairly simple level of nesting, but it can go infinitely deep.

The other tabs (C2...C5) do the same thing, with a slight variation in C3 which uses a menu item rather than a Submit button.

6.2. OrderView

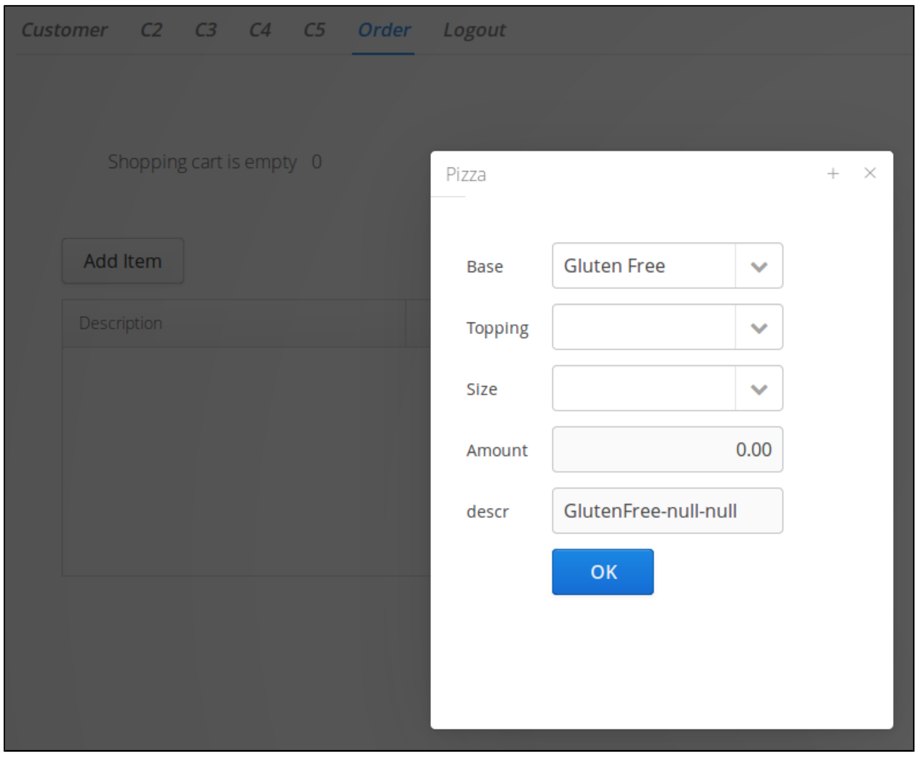
The order view is labelled 'Order' on the TabSet. This one does a little order entry function, like a shopping cart. We are ordering pizza but we need to say what kind of topping, size and base we want. And not all combinations are allowed. As we order pizzas they appear in the table on the order view.

In this view we use a `MaduraFieldGroup` to display the status of the order in two labels. The interesting thing about this is that instead of using input fields, which you have already seen, these components are labels and they are dynamic as we shall see. If we get the `MaduraFieldGroup` to create the fields it will always create input fields, possibly disabled ones, because it doesn't know any better. So this is an example of adding customisation.

The table is a `FormattingTable` which extends Vaadin's table to handle I18n headings and formating numerics. There are two columns on this table and one of them is numeric. To set it up all we need to do is name the properties we want to display ("description","-amount") and add a minus sign to the amount to right justify it. And we also name the headings ("Description","Amount"). Those headings will be used as is if there is no resource translation, but if there is the translation will be used.

The 'Add Item' button creates a popup window which allows you to create an order item.

6.2.1. PizzaWindow



Base: Gluten Free
Topping:
Size:
Amount: 0.00
descr: GlutenFree-null-null

Figure (3) Pizza Screen

The 'Add Item' button invokes the `PizzaWindow`. This uses a `MaduraFieldGroup` to present the fields on the `Pizza` object. The code is similar to Customer view. It is worth running the demo at this point to see what it does. You can see the three fields: Base, Topping, Size. Because only some combinations are valid you will find that as you pick options in, say, the Topping field, and list of options on the Size field will change, and vice versa. It doesn't matter what order you pick them. This is driven by the decision table in `choices.xml`. It does not have to be hard coded XML either, the values can be pulled from external sources such as a database. How to do that is in the Madura Rules documentation.

The other thing you will see when picking options is that the amount field, which is read only, will get a value and a new field 'testing' will appear. This is because of rules like this:

```
rule: Pizza "p2"
{
 if (size == "Small")
 {
  readonly(testing);
  activate(testing);
  amount = 10;
 }
}
```

The 'testing' field varies from inactive, active (ie visible), read-only, read-write and required. A description field displays the concatenation of the base, topping and size fields.

As usual there is a submit button on the display and the submit button is aware of whether the 'testing' field is required and not filled in, or not required, so it can enable and disable accordingly.

Once you click the submit button the pizza is added to the order, which means it displays in the shopping cart. The labels at the top of the Order view will change. Previously they said 'Shopping cart is empty' and a value of zero. Once you add a pizza it will say '1 items in cart' and the sum of the pizza values. How does that happen? Where is the code?

You probably guessed it is in the rules.

```
rule: Order "shoppingcartsize"
{
 if (count(pizzas) > 0)
 {
  orderStatus = format("shopping.cart.status",count(pizzas));
 }
}
rule: Order "shoppingcartsize"
{
 if (count(pizzas) == 0)
 {
  orderStatus = format("shopping.cart.status.empty",0);
 }
}
```

It relies on two resource strings:

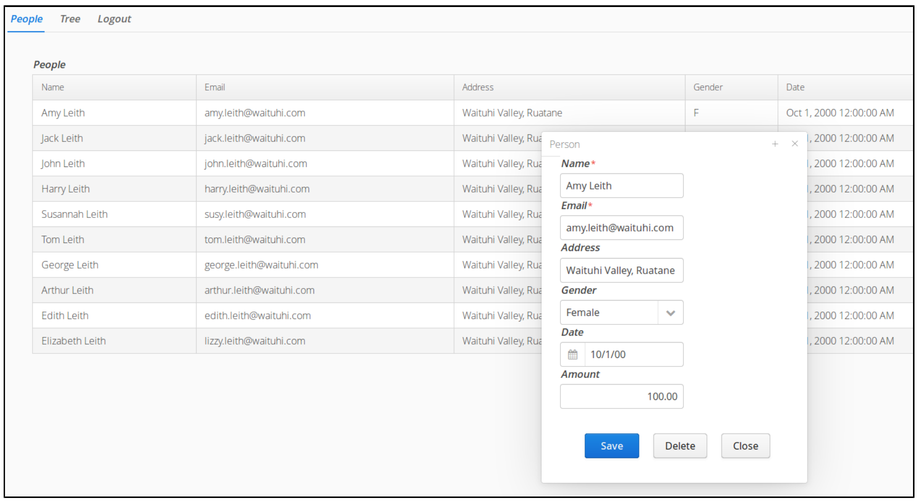
```
shopping.cart.status.empty=Shopping cart is empty
shopping.cart.status={0} items in cart
```

'pizzas' is a field on the Order which holds a list, and there are a number of built in functions that handle lists. What about the total? Yes, another rule:

```
formula: Order "sum"
{
 amount = sum(pizzas.amount);
}
```

Again, the rules do not know about the UI and the UI does not know about the rules, but the values automatically update. This demo does not provide a way to delete pizzas but if it did the total and the status would update appropriately because the rules know to re-fire when the data they worked with changes.

7. madura-address-book



People

Name	Email	Address	Gender	Date
Amy Leith	amy.leith@waituhi.com	Waituhi Valley, Ruatane	F	Oct 1, 2000 12:00:00 AM
Jack Leith	jack.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM
John Leith	john.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM
Harry Leith	harry.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM
Susannah Leith	susy.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM
Tom Leith	tom.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM
George Leith	george.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM
Arthur Leith	arthur.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM
Edith Leith	edith.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM
Elizabeth Leith	lizzy.leith@waituhi.com	Waituhi Valley, Rua		, 2000 12:00:00 AM

Person

Name* Amy Leith
Email* amy.leith@waituhi.com
Address Waituhi Valley, Ruatane
Gender Female
Date 10/1/00
Amount 100.00

Figure (4) Address Book

This demo is all about displaying and editing rows from a database in a table. It uses Madura Objects but not Madura Rules. It does use the madura-tableeditor so that needs to be specified as a maven dependency and the `nz.co.senanque.vaadin.tableeditor` package has to be scanned so that is added to the `ComponentScan` in the `AddressBookUI` class.

We also define beans in the `MyConfiguration` class: a table layout for persons and for trees.

The `nz.co.senanque.addressbook.jpa` package defines a data source and demo code to load hard coded values into an in-memory H2 database.

There is actually very little code in this demo because it is driven from the data structures. Once the table is loaded and visible you can right click to see a context menu giving options to edit, add or delete records. When adding or editing you have a Madura-backed Vaadin form with the usual validation and dynamic submit button you have seen in the other demos. It does not use Madura Rules but if it did there would be no change to the code except for some extra packages to scan.

7.1. Table Editor

The table editor is used in the address book demo. It is a bit more complicated to configure than the other components so this section details that.

First there needs to be a database defined. The address book demo only uses one database, which is more common than multiple database connections. The connection is defined in the `ConfigJPA` class. If defines three beans: a data source, and entity manager factory and a transaction manager. Because this is a demo we've used an in-memory H2 database and it gets loaded with data by the `LoadJPA` bean. Naturally this would be reworked for a production system.

The next class to look at is `ConfigContainer`. This class defines the two containers used in the demo: one for the people and one for the trees. Apart from their data types and bean names these beans are the same, and because they are so similar we need to give them names to make sure the right bean is injected into the right place. Do not be tempted to put container beans into your `ConfigJPA` file. These beans need the `EntityManager` established and that is actually done in the `ConfigJPA`, which means you cannot use it in that class hence the need for the `ConfigContainer` class.

All of the above beans are singletons. The rest of this is tied to a session.

In `AddressBook.MyConfiguration` the two containers are injected using their names. They each help define the `personTableLayout` and `treeTableLayout` beans. Those are the actual table layouts, they are Vaadin components and they are added to the `PersonView` and `TreeView` tabs

The container effectively defines a database query on a table. In the demo case the query is for all records but it can be refined in various ways. See the Vaadin documentation for that.[3]

8. MaduraFieldGroup and MaduraForm

Most of the demos make use of `MaduraFieldGroup` in various ways, and the ones that do not use `MaduraForm`. How do you know what technique to use?

Bear in mind there are two distinct phases that are always present. The first phase is the initialise phase. This is called once at application startup or, more usually, at session startup. The second phase is when we have an object such as a person, a customer or maybe a pizza, to bind to. This might be called multiple times as we operate on different customer or different pizzas.

This section refers to a number of examples and they are all found in Madura Rules Demo[7] under `src/main/java`.

8.1. MaduraForm

`MaduraForm` is based on Vaadin's `Form` which is deprecated, a possible reason not to use it. `Form`, and therefore `MaduraForm` is a `Component` which can be added to the display. You normally let the `Form` create its own fields and position them where it likes, usually in a vertical layout.

The steps you perform to use `MaduraForm` are:

- Initialise it with the constructor passing `MaduraSessionManager`. You can pass an optional layout in the constructor as well and this is what the created fields will be added to. The default is a vertical layout but you do have more options than that.
- Set the list of field names using `setFieldList`. The fields will be created when there is an object bound. If you set an invalid name it will be ignored without error.
- create any buttons you want using the form methods and position them where you like.

That was the initialisation phase. For the bind phase use `setItemDataSource`, to set a data source object. This triggers the creation of the UI fields.

The example for this is `nz.co.senanque.madurarulesdemo.CustomerView` and you will find this class running in the CustomerView tab.

8.2. MaduraFieldGroup

The `MaduraFieldGroup` is the (non-deprecated) replacement to `MaduraForm`, which parallels the developments at Vaadin. The essential difference in the new class is that it is *not* a component. You get to create the fields yourself and bind them. At least that seems to be Vaadin's intention, but we rather like passing a list of fields and having them created for us. There are three different ways to use `MaduraFieldGroup` and these are covered in four examples: `nz.co.senanque.madurarulesdemo.CustomerView2`, `nz.co.senanque.madurarulesdemo.CustomerView3`, `nz.co.senanque.madurarulesdemo.CustomerView4`, `nz.co.senanque.madurarulesdemo.CustomerView5`. When running the demo you can find these classes running under tabs C2, C3, C4 and C5.

But before you feel overcome by too many choices here we can say that our preference is for C5, and that C3 is a minor variation on C2, the difference being that it uses a menu bar rather than a button, so it does not really count as a different approach. So what are the essential differences between these approaches and how do they work?

First let's look at the common elements: each of these classes is a Vaadin component, they extend `com.vaadin.ui.VerticalLayout`. They are all `@UIScope` components which means they are all created per session by Spring. They each have a `@PostConstruct` method (named `init`) called by Spring, a `load()` method called by the aplication. They are all injected with a `MaduraSessionManager` object because they all use Madura Objects.

When `init()` is called we do not have an object to display and update, that is passed in the `load()` method.

8.2.1. C2 (and C3)

The `init()` method creates the field components and adds them to the layout, including the buttons and/or menu items. It does this using the injected `MaduraSessionManager` object to create a

`MaduraFieldGroup` and this is called to create menu items and buttons, as well as to bind the fields to the field group.

The `load()` just calls the `setItemDataSource()` on the field group already created.

You get complete control over what kinds of fields are created and where they are put with this approach. This can be an advantage and a disadvantage. You get precisely what you want, of course, but you have to maintain consistency yourself. If, say, the status field is normally rendered as a `ComboBox` it is up to you to ensure it is a `ComboBox` here.

8.2.2. C4

In this example the `init()` method does very little, it just sets up a layout for us to add fields to later. The main work is done in the `load()` method which calls the field group `buildAndBind()` method. The `buildAndBind()` method accepts a list of property ids and for each of those it creates a field driven by the customisable hints (`nz.co.senanque.vaadin.Hints`) bean. It returns a map of property ids and components which are then added to the layout in whatever way we want.

The advantages of this approach is that you will get consistencey of field types generated for properties automatically, you do not have to remember that the status field ought to be a `ComboBox` but you get complete control over where the fields are placed on the layout. You can even coerce them into different enabled/disabled states if you want, though we prefer to use Madura Objects to do that for us because, like the field type, it gives us automatic consistency.

8.2.3. C5

This is very like the C4 example except that it uses another variant of the `buildAndBind()` method. It passes a layout in the first argument and dispenses with the loop C4 used to add the fields to the layout. This `buildAndBind()` variant does the loop internally. It gives you less control over the placement of the fields but it is less to code.

9. Mobile Applications

There is an on-line demo of the mobile application described here at [15].

9.1. Touchkit

Touchkit is an addon product produced by Vaadin to enable 'touch' devices like phones and tablets, so this is what you need for a mobile application. There is a lot to Touchkit that we will not cover here (just as there is a lot to Vaadin that we don't cover). But we will do the basics and show it working with Madura.

Touchkit is a dual licence product. If your project qualifies for AGPL then you use the free agpl version. If not you need to buy a licence from Vaadin. Our demos are AGPL so they use the agpl version.

At the time of writing the version of Touchkit we need was not in the maven central repository so there are some repository references in the pom file for the demos.

But the demo project does not refer to Touchkit directly. There is a `madura-vaadin-touchkit`library that adds some Touchkit specifics to help Madura, and that has the dependency

```
<dependency>
 <groupId>com.vaadin.addon</groupId>
 <artifactId>vaadin-touchkit-agpl</artifactId>
 <version>4.1.0</version>
 <type>jar</type>
</dependency>
```

But what you will find in the `madura-mobile-demo` is this:

```
<dependency>
 <groupId>nz.co.senanque</groupId>
 <artifactId>madura-vaadin-touchkit</artifactId>
</dependency>
```

9.2. The Demo

The `madura-mobile-demo` project is the one we are looking at. This combines *two* UIs: a desktop UI and a mobile UI. The two UIs do much the same thing, they certainly have the same underlying objects and rules (using Madura Objects and Madura Rules, of course). Part of the point is to show that while you might have to deliver a different UI for a different technology you do not have to re-engineer the object and rules to doit.

The demos displays a message complaining about the widgeset being the wrong version. This is assumed to be a bug in Touchkit and seems to cause no problems.

The `MaduraMobileDemoDesktopUI` is essentially no different from the UI class in the other demos. It is an extention of the `com.vaadin.ui.UI` class and has similar annotations. The one thing you might find puzzling there is that the various static classes that define the web servlet, web listener and Spring configuration are missing. They are in the mobile UI class. There have to be only one instance of these in the application and in this case the one instance is in the other class. But it does not matter where they are as long as they are present somewhere.

The more interesting class at this point is the `MaduraMobileDemoTouchKitUI` class.

The first thing to notice abut this class is that the static classes are present here, but the servlet class extends `SpringAwareTouchkitServlet`. This implements some things needed for Touchkit to work correctly with Spring. The class is adapted from Matti Tahvonen's version. Matti Tahvonen works for Vaadin.

The next thing to notice is the annotations on the UI are a little different.

```
@Widgetset("com.vaadin.addon.touchkit.gwt.TouchKitWidgetSet")
```

```
@Theme("madura-touchkit-theme")
@SpringUI(path="mobile")
public class MaduraMobileDemoTouchKitUI extends UI {
...
```

The widgetset is the default one for Touchkit.

The theme is one derived from the Touchkit theme, which is different from the desktop theme. Touchkit requires its own theme and we've added a small addition to it to make numeric fields justify right. The theme comes from the `madura-vaadin-touchkit` dependency. If you don't care about the right justify you can use 'touchkit' there instead.

To get to the mobile UI there needs to be a `/mobile` added to the URL. This is actually done for us by the login sequence is you are using `madura-login`, but if your application uses something else instead you will need to have it redirect to the right UI.

The rest of the UI, including the classes it invokes are much the same as the desktop UI except that there are some Touchkit specific controls used such as `NavigationManager`. You can use the same techniques for creating and binding objects to your UI. When it creates UI fields for you a different field factory is invoked to supply Touchkit fields rather than desktop ones.

That brings us to the last thing to notice about the UI class namely the definition of the `hints` bean. In the other demos this just returned `HintsImpl` but here we need to make a decision about what hints we want because they are different. As with the other demos this gives you the opportunity to customise the hints by supplying your own hints classes.

The desktop screen looks much like the rules demo you saw earlier. The mobile screens look like this:

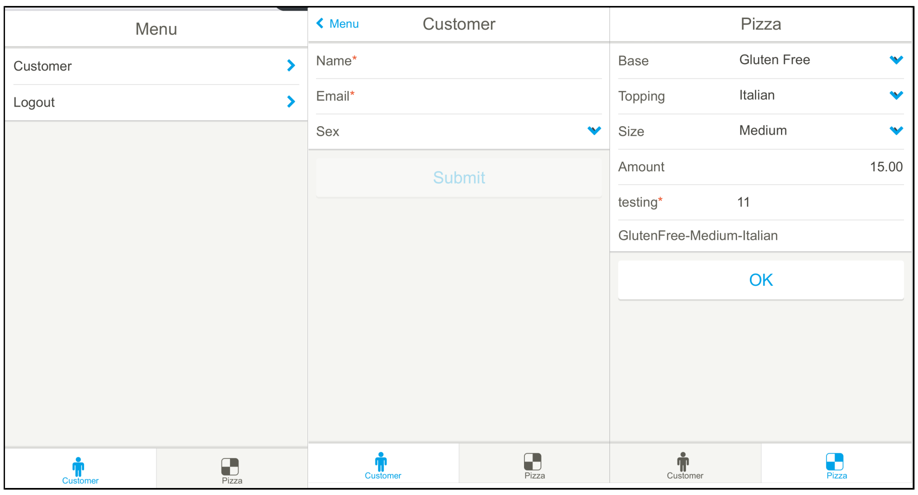


Figure (5) Mobile Screens

9.3. Building the Mobile App

If you build `madura-mobile-demo` with maven in the usual way you will get a war file you can deploy and browse to, including from your mobile devices. But this still looks like a web application rather than a mobile application. For example you still see the URL at the top of the browser display. Also we cannot put this into an app store and have users install it from there. Yes we can!

The key to this is a product called PhoneGap. The details of how to use it to package a Vaadin application for an app store are in [11].

But there is very little to it. In the madura-mobile-demo sources you will find a directory called `cordova` which contains the files to build the PhoneGap application. Most of this is specific to Phonegap and it is documented by that product, but the thing you need to know for now is that this application is essentially a wrapper for the browser. Phonegap can do a lot more than that but this is a demo so we will keep it simple. The URL for our deloyed war file in the `index.html` file. There is also a `config.xml` file which holds the name and description of the application, the name generates the final file name of the apk file.

The Phonegap part of the build is disabled by default and to use it you need to run the `phonegap` maven profile. But first you need to sign up to Phonegap and get a user name, password and create an application id. These need to be defined as maven properties `phonegap-build.username`, `phonegap-build.password` and `phonegap-build.appId`.

This apk file is the simplest kind of application you can build, and because it is not signed it cannot be put into the app store. Only the Android variation lets you generate unsigned applications, which is why we are using it here. To generate signed applications you need to apply to the relevant app stores for an id and tell Phonegap about it. Phonegap supports builds for Apple iOS devices and several others.

If you want to try out a pre-built copy of the application you can find a copy at [16]. This can be side loaded onto an Android device. You will need to modify the security settings on the device to allow it to install files from unknown sources, and you do this at your own risk. We have tested this on several Samsung devices.

10. madura-login

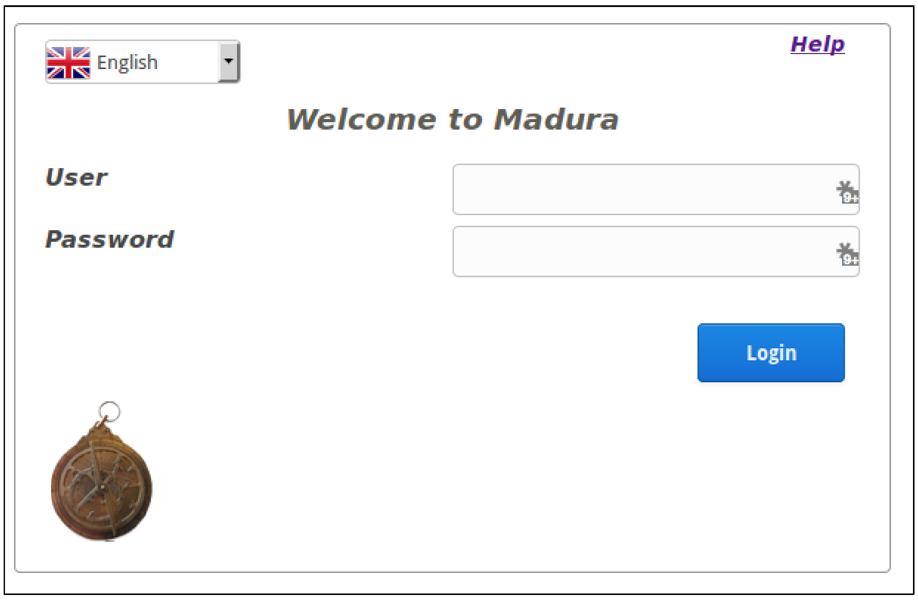


Figure (6) Login Screen

This sub project provides a demo-ware login facility. It uses a web fragment to provide authentication and authorization services.

To configure it into an application we need to tell Spring which directories to scan. All the demos here do this in the `WEB-INF/applicationContext.xml` file. That file also defines the `permissionManager` bean, although we could have added it with a `@Bean` in the `@Configuration` like this:

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

The `@UIScope` does the same job as the `scope="vaadin-ui"` on the bean definition in the context file.

That is the minimum. But we can do more with security. First, we can remove the existing login mechanism on any of the demos and replace it with something else. Second, we can keep the existing login mechanism and reconfigure it or extend it.

10.1. Replacing madura-login

The connection between madura-login and the application is deliberately loose. The login process puts some attributes into the session and the application code pulls them out and uses them to configure the `PermissionManager`. Madura uses the `PermissionManager` for all security references. So if you don't want to use madura-login you need to provide another way to configure the `PermissionManager`.

This is easily done by supplying a bean that implements the `PermissionResolver` interface:

```
@Component("permissionResolver")
@UIScope
public class PermissionResolverImpl implements PermissionResolver {
 @Autowired PermissionManager permissionManager;

 @PostConstruct
 public void unpackPermissions() {
  WrappedSession session =
 VaadinService.getCurrentRequest().getWrappedSession();
     String currentUser =
 (String)session.getAttribute(AuthenticationDelegate.USERNAME);
     @SuppressWarnings("unchecked")
  Set<String> currentPermissions =
 (Set<String>)session.getAttribute(AuthenticationDelegate.PERMISSIONS);
     permissionManager.setPermissionsList(currentPermissions);
     permissionManager.setCurrentUser(currentUser);
 }
}
```

This is the out-of-the-box behaviour.

The class is instantiated once per session and the `PostConstruct` method will be run once the `PermissionManager` has been injected and is ready for further configuration with the permissions list and the user name fetched, in this case, from the session attributes. Using session attributes is a typical way to pass authentication information but the actual details and structure of what is passed varies from authentication product to authentication product. Implementing your own version with different attribute names should be fairly simple.

10.2. Reconfiguring madura-login

You can configure the following:

- Add a customised login.html file to your project's resources directory
- Add a customised login.css file to your project's resources/css directory
- Add a customised logo.gif file to your project's resources/images directory
- Add a customised users.csv file to your project's WEB-INF directory this holds the list of valid users, passwords and their permissions
- Deploy a different `AuthenticationDelegate` implementation as a bean

Sometimes when testing your application you can get weary of logging in over and over. Add this to your `config.properties` file:

```
nz.co.senanque.login.RequestValidatorImpl.defaultLogin:admin/admin
```

That will automatically log you in as user/password admin/admin. Naturally you remove this for production.

10.3. Login and Mobile

When developing mobile applications you often need to include two UIs in the same application. The login page itself uses `@media` queries in the css file to adjust itself for different devices, specifically a Galaxy S4 and an iPad mini, but the adjustments are generic enough to cover more devices than that. It will also redirect to your application's mobile UI as well. This is controlled by another config setting:

```
nz.co.senanque.login.RequestValidatorImpl.mobilePathPrefix:mobile
```

This is the default setting so all you need to do is have your application's mobile UI look like this:

```
@SuppressWarnings("serial")
@Widgetset("com.vaadin.addon.touchkit.gwt.TouchKitWidgetSet")
@Theme("madura-touchkit-theme")
@SpringUI(path="mobile")
```

```
public class MaduraMobileDemoTouchKitUI extends UI {
...
```

The vital bit there is the `@SpringUI` which specifies a path. The path there must match the value in the config file. The widgetset being used is the default touchkit widgetset.

10.4. Extending madura-login

Each demo includes its own version of `login.html` with a help link, mostly to show how it can be done and also to help people using the online demos. The link uses different urls for different demos.

Things a production version would have to have:

- A more secure user storage, you would implement this with a replacement `AuthenticationDelegate`. The replacement should not store passwords in clear text etc.
- A 'remember me' checkbox.
- Security timeout
- forgot password/change password. This would need to tie in with your `AuthenticationDelegate` implementation.

The out-of-the-box implementation has just two user/passwords: admin/admin and user/user. Admin has the ADMIN permission which is used in the rules demo.

A. License

The code specific to Madura-VaadinSupport is licensed under the Apache License 2.0 [13].

Most of the dependent products have compatable licenses detailed in their pom files.

TouchKit, which is an optional dependency, has a dual AGPL 3.0 and Commercial Vaadin Addon License (CVAL) 2.0. That basically means that if you are building a project compatible with AGPL you can use this for free, if you are doing something else you have to buy a license from Vaadin. Madura Rules, also an optional dependency, has a similar license arrangement.

B. Release Notes

You need Java 1.7 to compile this project.

3.2.2

Added Docker file to vaadin7-demo.

Upgraded Spring and Madura dependencies.

3.2.1

Updated phonegap dependency.

Removed title bar in mobile.

Fixed reference to api location.

3.2.0

Fixed a problem with the slash at the end of the URL.

Added dual widgeset to mobile demo.

Compiled under Java 1.8, Vaadin 7.7.3, Touchkit 4.1.0.

3.1.0

Fixed several problems in login which prevented clean deloyment on Openshift.

3.0.0

Major rework/rerwite of the Madura Vaadin code.

Supports Vaadin 7, Touchkit 4, Madura 3.0.0 and Spring 4.

Uses more compact configuration.

Better organisation of sub projects.

Uses Vaadin's Spring addon rather than SpringApplicationLoader.

Removed dependencies on Madura Bundles.

2.6.0

Just realigning the versions, including using a later version of Madura Objects and Madura Bundles.

2.5.4

Upgraded the BundleListener to properly handle bundle deletion.

2.5.3

Changed dependency on Madura Objects and Madura Bundles to 2.2.4 and 4.0.3 respectively.

2.5.2

Changes for git.

2.5.1

Subapplication caption now tries to translate 'project.name' first, then falls back to bundle name.

Fixed confusion between buttons and checkboxes.

Set MaxLength on text fields.

Eliminate BeanUtils references and use MaduraObjects metadata instead. BeanUtils was giving odd results.

More flexible handling of message source accessor, specifically when the message source may exist in a bundle rather than the main application, it now gets passed around as an argument.

Better clean up of registered fields and labels on close of session.

Changed field generations so that if no write permission we set the field to read-only. Previously it was disabled instead.

Changed SubmitButtonPainter so that if you set the MaduraForm to readOnly then the submit button will be disabled.

Updated Spring dependencies.

2.5.0

Moved to maven build.

2.4.1

Fixed a bad pom file. The version was incorrect.

2.4

Added pom file for maven projects.

Touchkit is now a compile only dependency so it is not automatically pulled from the repository. This is to allow more explicit handling of the Touchkit licence.

2.3

Built with Java 1.7.

2.2

Changed default layout to FormLayout from VerticalComponentLayout in TouchkitMaduraForm because the latter fails to handle dynamic required fields.

Added a login mechanism for Touchkit.

2.1

Fixed problem with help screen not popping back.

Added TouchkitMaduraForm.

2.0

This is a major rework which eliminates the several factories that relied on statics and singletons to work. This version, with a loss of backward compatibility, improves the useability of the library as well as making it compatible with Vaadin's Touchkit.

Added a proper sample application showing the use of the Hibernate Container.

1.9

Fixed problem with logout.

1.8

Fixed problem with source never displaying from ivy

Added method getMaduraPropertyWrapper() to MaduraSessionManager

Fixed problem in FormatterDouble which always failed to parse a valid number

1.7

Clearing a field doesn't fire the rules, issue #3: fixed.

Added support for pluggable applications (perspective manager).

Removed use of MessageSourceAccessorFactory because it does not play well with Madura Bundles.

Improved numeric parsing. Previously something like 1x000 would return 1 rather than invalid. It now returns invalid.

Added menu item control (visibility, enabled/disabled) from rules using the same mechanism as buttons.

Made some more classes Serializable.

1.6

Added serialisation to some classes Tomcat complains about on shutdown. There are some Spring classes it still complains about, though.

Added default login.html in case the one in the theme is missing, also show warning. The access of the login.html file has been tightened up so it works on Tomcat, ie more portable.

Added the generic login listener: SpringLoginListener. This uses Spring Security and handles unpacking the authorities and using them as permissions.

1.5

Fixed several issues with the formattingTable.

Adjusted dependencies.

1.4

Handling case where we add a row and then cancel, row is removed.

Fixed problem with using evict which stopped us adding new rows.

1.3

Boolean fields now rendered as checkbox.

Submit button not disabling when required field is blanked. Fixed.

Fixed NPE when login fails.

Setting a required field to blank failed to disable the submit button. Fixed.

1.2

Simplified the binding needed in the table editor.

1.1

Support for @Secret fields

Error handling/reporting added.

1.0

Initial version