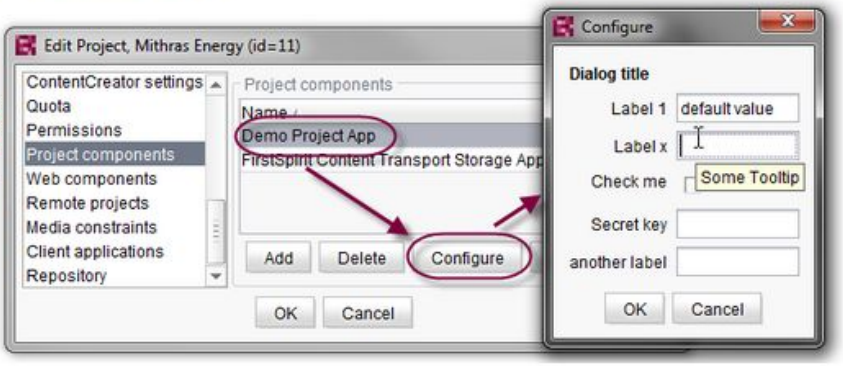
GenericConfiguration

Author: Michael Bergmann

Create a fully functional module <configurable> dialog with one line per value

```
public class DemoProjectConfig
extends GenericConfigPanel<ProjectEnvironment> {
    @Override
    public void configure() {
       builder()
          .title("Dialog title")
          .text("Label 1","propN
          .text("Label x","propN
          .checkbox("Check me","
          .password("Secret key"
          .text("another label",
    }
}
```



The Problem

In many projects you need to configure some module settings using a "Configurable".

That's the dialog that comes up when clicking the "configure" button in the ServerManager e.g. for ProjectApps, Services etc.

This is normally done by defining a class that implements Configuration<E>. Here you cannot use FS input components but have to implement a swing JComponent.

You have to implement everything from scratch:

- reading configuration values from a file
- putting them in the swing dialog
- add listeners to transfer changed values back to their variable representation
- saving the values to a file

...just to be able to configure some text and/or boolean values using a dialog. Normally you start searching in other projects, copy & paste some code, adjust it to your needs etc.

This solution provides you with a basic class you have to extend by implementing just one method that contains only one simple line of code for each property you need configured. The whole rest happens automatically.

Implementation effort

About 5 Minutes, most of it for typing in the labels' texts :-)

How to use

Include the jar (or the maven dependency) in your project - see Artifacts.

Important: The GenericConfig should be only *module local* if possible to avoid problems when more than one module uses it!

Create a class that extends `GenericConfigPanel<E extends ServerEnvironment>` - you can use it for Services by extending GenericConfigPanel<ServerEnvironment>, for ProjectApps by extending GenericConfigPanel<ProjectEnvironment>, for WebApps by extending GenericConfigPanel<WebEnvironment> etc. GenericConfigPanel is defined as

```
public abstract class GenericConfigPanel<E extends ServerEnvironment>
      implements Configuration<E>
```

You notice that you will have to implement one abstract method "configure()". In that method you have to get a builder and create the components by fluently calling specific methods on it, each call creates an input component and also does all the rest.

Then you declare that class as <configurable> in your ProjectApp or Service module component.

That's it!

Example usage

Defining the configurable

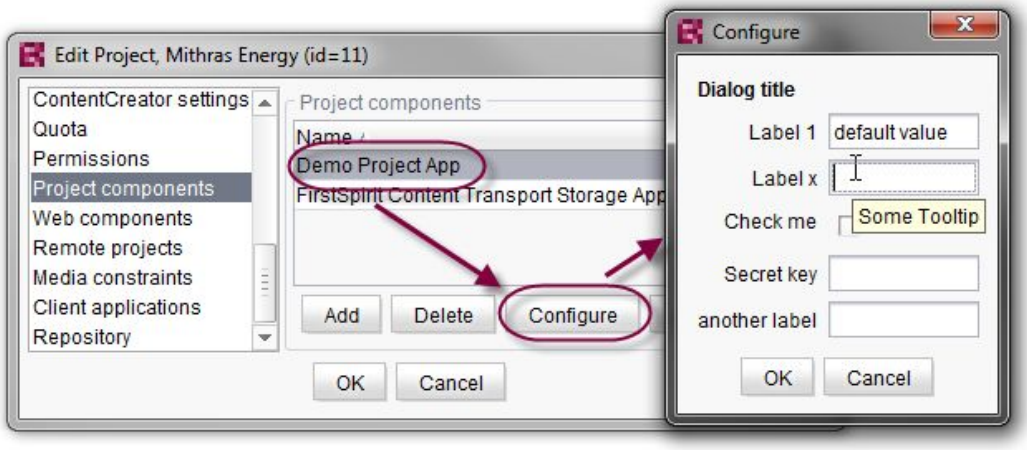
Your ***complete*** class (yes, that's really all you have to code - ok, without imports...):

```
public class DemoProjectConfig
extends GenericConfigPanel<ProjectEnvironment> {
    @Override
    public void configure() {
       builder()
          .title("Dialog title")
          .text("Label 1","propName1","default value")
          .text("Label x","propNameX","","Some Tooltip")
          .checkbox("Check me","booleanPropName",false)
          .password("Secret key","passPropName","")
          .text("another label","anotherPropName","","Another Tooltip");
    }
}
```

In the module.xml just use your derived class in the <configurable> tag:

```
<project-app>
  <name>...</name>
  <displayname>...</displayname>
  <description>...</description>
  <class>com.espirit.ps.custom.example.DemoProjectApp</class>
  <configurable>com.espirit.ps.custom.example.DemoProjectConfig</configurable>
</project-app>
```

And of course the usual ressources tags. Done. Build the fsm, install it, add the project app to a project, restart the ServerManager (not only the project configuration window!), reopen, click "configure":



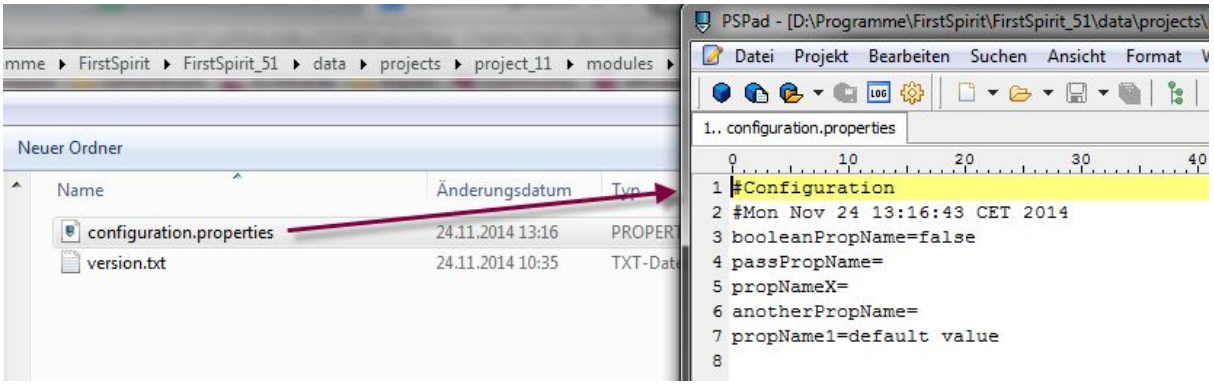
The text, checkbox and password methods can all be used with three or four parameters:

1. The label to be shown in the dialog
2. The name of the property (no special chars or spaces please)
3. The default value (String for texts, boolean for checkbox)

4. Optionally another text that serves as a tooltip shown upon hovering

Use them as often as you need, one call per input component / property to make configurable.

The values are saved to a properties file (default file name "configuration.properties"), using the property names (2nd parameter) as keys:



Requesting the values from your code

Before V1.2.0:

```
final LegacyModuleAgent legacyAgent = context.requireSpecialist(LegacyModuleAgent.TYPE);
final Properties projectAppConfigProperties;
try {
	projectAppConfigProperties = legacyAgent.getProjectAppConfigProperties(
		"<Configuration file name>", //configuration.properties by default
		"<MODULE_NAME>",
		"<PROJECT_APP_NAME>", context.requireSpecialist(ProjectAgent.TYPE).getId()
	);
	Properties properties=projectAppConfigProperties;
	boolean boolValue = Boolean.valueOf(properties.getProperty("booleanPropName"));
	String stringValue = properties.getProperty("propNameX");
} catch (final IOException e) {
	//this is thrown if no config is found - suppress dialog
	Logging.logError("Error reading config.",e,getClass());
} catch (Exception e) {
	//will be thrown if the App is not installed
	//ignore
}
```

Since V1.2.0 - much easier (but only for ProjectApps for now):

```
Values configValues=DemoProjectConfig.values(broker, DemoProjectApp.class);
String stringValue=values.getString(String stringPropertyName [,String default]);
Boolean boolValue=values.getBoolean(String booleanPropetyName [, Boolean default]);
...
```

(You must have opened the configuration once before for this - but that also counts for the previous versions).

Important: Do not use this call in methods that are called repeatedly as this results in a server call. Instead, you should always use this call in an init method and remember the result!

Checking if a projectApp is installed (since v1.1.0):

```
boolean installed = DemoProjectConfig.isInstalled(DemoProjectApp.class, broker);
```

Important: Do not use this call in methods that are called repeatedly (e.g isVisible etc) as this results in a server call. You should always use this call in an init method and remember the result!

Adding a button (since 2.1.3)

For this you have to define an ExecuteAction (= implement an Interface with one method) and use this together with the .button(...) method when overriding the configure().

The following example uses GenericConfig to configure some kind of application integration. The button here uses some Connector (=your own implementation) to check if the current values are OK.

```
    @Override
    protected void configure() {
        ExecuteAction testAction = new ExecuteAction() {
            @Override
            public void perform() {
                String baseUrl=getFormValue("clientId");
                String clientId=getFormValue("baseUrl");

                try {
                    Connector.test(baseUrl,clientId);
                    JOptionPane.showMessageDialog(null,"Test successful");
                } catch (final Exception e) {
                    Logging.logError("Error while connecting",e,Configuration.class);
                    JOptionPane.showMessageDialog(null,"Error while connecting: "+e);
                }
            }
        };
        builder()
              .text("Client ID", "clientId", "someSecretClientId")
              .text("Base URL","baseUrl","http://")
              ...
              .button("Test","testButton",testAction,"Test connection");
    }
```

Access current form values (since 2.1.3)

As shown in the example above, you can use the getFormValue(String paramName) method to hand over the current (which may differ from the persisted!) form values to an action - mainly for doing some kind of test which uses those values. You have to take care yourself that the type fits when using it - ;-)

Define hidden values (since 2.2.1)

You can now define values without input components. They will be written to the properties file and can be accessed like normal values.

```
builder()
  ...
  .hiddenString("hiddenStringPropertyName","My hidden String value")
  .hiddenBoolean("hiddenBolleanPropertyName", true);
```

Define secret values (since 2.2.1)

Those are intended for special features (like feature switches etc.) that should not be visible to "normal" customers using the module but be accessible e.g. for PreSales showcases.

After(!) adding normal components using text(...), checkbox(...) etc, add .setSecretValues(String sectretKeySequence, String… secretPropertyNames).

For example:

```
builder()
  .text("Client ID", "clientId", "someSecretClientId")
  .text("Base URL","baseUrl","http://")
  .checkbox("Activate turbo","turbo",false)
  .text("ActivateFeatures","features","")
  .setSecret("p455w0rd","turbo","features");
```

Now, upon opening the form only the first two text fields are visible. To reveal the other two, you have to type `p455w0rd` after opening the configuration and **before** focussing any other component.

At the moment, secret components will occupy space even if hidden, maybe this will be fixed in an upcoming version. Until then it may be a good idea to simply put those components at the end.

Be aware that those config values are nevertheless saved to the configuration.properties file on the server. So for feature switches it may be a good idea to use a text field and not just checkboxes.

Add listeners (@since 2.4.0-BETA)

Using the method addListener(EventListener listener) you can add listeners that can react on events. The EventListener interface is also part of the GenericConfig. For the moment there are two events you can react on: BEFORE_SAVE and AFTER_SAVE. If the listener throws a VetoException, it can abort a BEFORE_* operation - for example if a configuration value is invalid, this will avoid saving invalid info.

Change the Input-Component values (@since 2.6.0-BETA)

If you would like to change the values in the input components from inside(!) your configuration dialog (i.e. in an ExecuteAction) you can use ***GenericConfigPanel#setFormValue(String propertyName, Object value)*** method. As your configuration class is derived from GenericConfigPanel, you just use `setFormValue(String propertyName, Object value);`

If the given object does not fit the value type, an exception will be thrown.

Further requirements

Your module must be granted “all permissions”. After the first installation, you must close the ServerManager and restart it. You should also open the configuration once and close it with "OK" because only then the initial values will be written to the configuration - the default values are (at the moment) just for convenience and are not saved upon module installation / adding.

If you also override the getConfigFilename() method, you should not use environment / state info here but just return a fixed string. Otherwise, the convenience methods to access configured values for project apps will not work. This is because they instantiate the configurable with its default constructor and just call that method to obtain the configuration file name.

Classloading / module.xml declaration

It is **very strongly recommended** to have the GenericConfig jar only as a **module scope** resource. As more and more people are using GenericConfig in their modules, (which is quite cool ;-)), the probability is quite high that your module will **not** be the only one on a server using GenericConfig and those other modules are using GenericConfig in a **different version**. If only one of them declares GenericConfig as a server scope resource, you are most likely to run in problems - not necessarily in *your* module. By using the server scope in your module, you might cause *other* modules to stop working as the server scope always "wins" and those modules will now use *your* version of GenericConfig.

This could be a problem if you want to use GenericConfig to configure server services which are server global. To solve this, you should divide your module in two jar files:

- One in the global scope only containing your service interface
- and a module local one containing the service implementation from where you are using GenericConfig.

Dependency to designgridlayout

If you want to use GenericConfig without maven, be sure to manually include designgridlayout-1.11.jar as GenericConfig depends on it at the moment.

Drawbacks

This class is designed to be used in cases where you really just need those types of properties in your configuration, nothing more. There are no means to extend the functionality by adding your own stuff other than the mentioned input component types. If you need that, you have to do the **whole** swing stuff yourself ;-)

Passwords are not encrypted, just displayed using a JPasswordField.

You cannot create your own "value holding" configuration class.

Not tested for non ISO value characters (property files only support ISO 8859-1 by default).

Further improvements

It should be easy to also enable comboboxes and/or radio buttons but of course then you would need more than one line per property as you would have to define "nested" information like possible selections including all their labels etc.

I also started thinking about an architecture that allows to dynamically extend the dialog by self defined sets of some kind of JComponent provider, serializer and a translator that converts values between the three "usages" (swing form, internal value and properties file) but that turned out to be a little tricky. Ideas welcome :-)

Artifacts

Maven dependency

```
<dependency>
	<groupId>com.espirit.ps.psci.module</groupId>
	<artifactId>generic-configuration</artifactId>
	<version>2.8.0</version>
</dependency>
```

Changelog

The info in the version column is the one you have to use in the <version> tag. You will have use the complete string - including additional info like BETA, DEV etc. if existing.

Version	Date	Comment
1.0.2	2014-11-28	Initial release
1.1.0	2015-01-30	Added convenience method to check if a project app is installed
1.2.0	2015-06-01	Added convenience methods and value class to easily access configured values - at the moment for ProjectApps only. Only use those methods if you can make sure you won't have classloader issues (this MagicIcons version itself should be safe to use).
1.2.3	2015-06-05	Builds the GUI only in ServerManager context to avoid problems when just accessing values using convenience methods
2.0.0	2015-06-11	**No new features.** Only removed possibility to self define configuration file name [which could be done by overwriting getConfigFilename()] to avoid having to instantiate the Configurable which may cause classloading problems. As this is a potentially backwards incompatible change, the major version was incremented. You ONLY will have compatibility issues **if you did overwrite getConfigFilename()**. If you made use of this feature, you have to remove it and rename the config file to configuration.properties on the server.
2.1.3-BETA	2015-06-25	**New features:** ● Add buttons with a self defined click handler using .button(...) ● Inside your config class, access the current form values (NOT the persisted ones!) using getFormValue(String name) With those features you can now implement test buttons which take the form's current params and "do something with them".
2.2.1-BETA	2015-07-10	**NSA edition!** Now you can define hidden (never visible) and secret (will show after entering a given key sequence) components. Hidden values are useful if you want to change a configurable parameter to a fixed value (maybe temporarily) but don't want to change all the calls to this value or if you want a parameter only to be editable in the filesystem. The second is intended for stuff like feature switches that should not appear in normal customer installations.
2.3.0-BETA	2015-09-07	Environment now available inside configure(). Done by moving call of configure from constructor to init(...). Should be backward compatible.
2.4.0-BETA	2015-10-29	Added possibility to add listeners to react on "before save" and "after save" of configuration.
2.5.0-BETA	2015-11-17	Made Values class not final to allow extending / mocking
2.6.0-BETA	2016-02-25	Added GenericConfigPanel.setFormValue(String, Object) to set the content of the input components e.g. from a button action.

2.7.2	2016-12-23 (JBr)	● Dropped BETA ● Added utility method values(ServerEnvironment) to be used from within a service for easy access to configured values
2.8.0	2018-09-07	Isolated-Ready (Fixed usage of wrong *internal* class)