spring-mvp 1.0 Manual

Christoph Guse <info@flexguse.de>

Table of Contents

1. Introduction

spring-mvp is an Vaadin addon which mainly realizes two general ideas: event driven applications and the pattern of Model, View and Presenter (mvp). The Spring Framework [http://www.springsource.org/spring-framework] is used to configure and inject Views and Presenters.

Vaadin Addon Repository https://vaadin.com/directory [https://vaadin.com/directory]

Git Repository https://bitbucket.org/flexguse/springmvp-addon [https://bitbucket.org/flexguse/springmvp-addon]

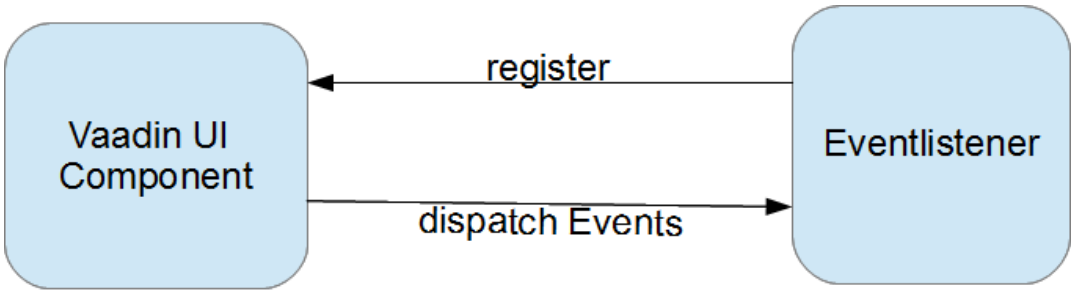
Issues Tracker https://bitbucket.org/flexguse/springmvp-addon/issues [https://bitbucket.org/flexguse/springmvp-addon/issues]

Demo application http://springmvp-demo.flexguse.cloudfoundry.com/ [http://springmvp-demo.flexguse.cloudfoundry.com/]

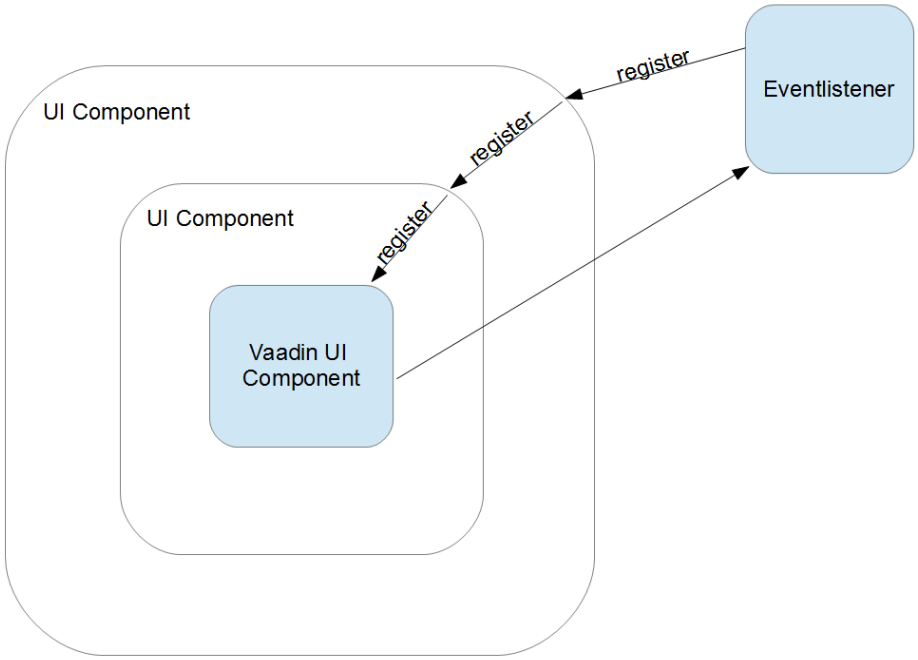
1.1. Event Driven Applications

The idea of a event driven application is simple. Somewhere in the application an event is dispatched and somewhere in the application there is a listener which listens for the event and does something with the idea.

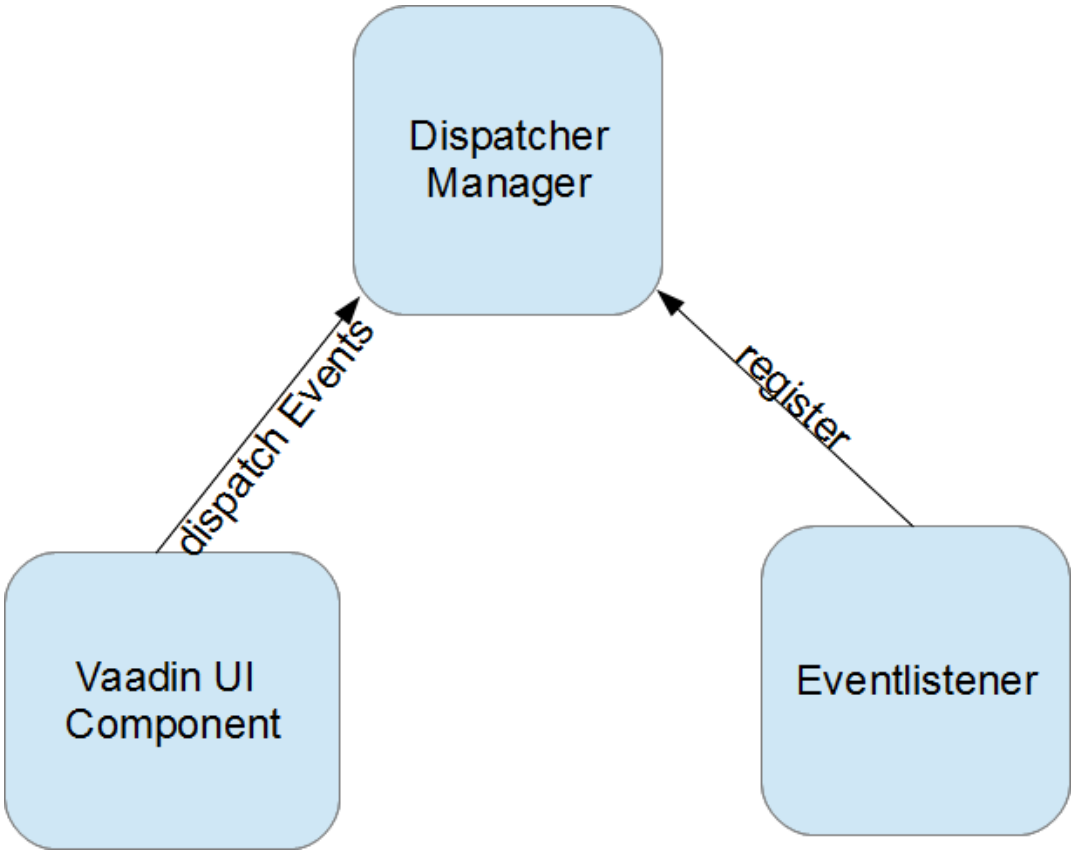
In Vaadin there already exists the possibility for UI components to dispatch events.



This is a good approach to listen for events but it gets difficult if the Eventlisteners need to be centralized and reused or the layout of the application gets complex. Then the EventListener needs detail knowledge of the UI and is tightly coupled. No chance to change the UI without changing the EventListener.



In spring-mvp there is the idea of a `DispatcherManager` which is available all over the application. A UI component registers itself als EventListener in the `DispatcherManager` and there is absolutely no need for th EventListener how it is used in the UI. Events are dispatched using the `DispatcherManager`.



1.2. Model View Presenter (MVP)

Developing Applications in Vaadin is easy. Only some UI components needs to be instantiated and combined, some listeners need to be registered and the small application is complete. This procedure leads to quick results but the resulting code tends to be long and difficult to maintain. An example taken from the Vaadin sampler.

```
 1
 2 package com.vaadin.demo.sampler.features.trees;
 3
 4 import java.util.Collection;
 5
 6 import com.vaadin.data.Item;
 7 import com.vaadin.data.Property;
 8 import com.vaadin.data.Property.ValueChangeEvent;
 9 import com.vaadin.demo.sampler.ExampleUtil;
10 import com.vaadin.event.Action;
11 import com.vaadin.ui.AbstractSelect;
12 import com.vaadin.ui.Alignment;
13 import com.vaadin.ui.Button;
14 import com.vaadin.ui.Button.ClickEvent;
15 import com.vaadin.ui.HorizontalLayout;
16 import com.vaadin.ui.TextField;
17 import com.vaadin.ui.Tree;
18
19 @SuppressWarnings("serial")
20 public class TreeSingleSelectExample extends HorizontalLayout implements
21         Property.ValueChangeListener, Button.ClickListener, Action.Handler {
```

```
22
23  // Actions for the context menu
24  private static final Action ACTION_ADD = new Action("Add child item");
25  private static final Action ACTION_DELETE = new Action("Delete");
26  private static final Action[] ACTIONS = new Action[] { ACTION_ADD,
27          ACTION_DELETE };
28
29  private Tree tree;
30
31  HorizontalLayout editBar;
32  private TextField editor;
33  private Button change;
34
35  public TreeSingleSelectExample() {
36      setSpacing(true);
37
38      // Create the Tree,a dd to layout
39      tree = new Tree("Hardware Inventory");
40      addComponent(tree);
41
42      // Contents from a (prefilled example) hierarchical container:
43      tree.setContainerDataSource(ExampleUtil.getHardwareContainer());
44
45      // Add Valuechangelistener and Actionhandler
46      tree.addListener(this);
47
48      // Add actions (context menu)
49      tree.addActionHandler(this);
50
51      // Cause valueChange immediately when the user selects
52      tree.setImmediate(true);
53
54      // Set tree to show the 'name' property as caption for items
55      tree.setItemCaptionPropertyId(ExampleUtil.hw_PROPERTY_NAME);
56      tree.setItemCaptionMode(AbstractSelect.ITEM_CAPTION_MODE_PROPERTY);
57
58      // Expand whole tree
59      for (Object id : tree.rootItemIds()) {
60          tree.expandItemsRecursively(id);
61      }
62
63      // Create the 'editor bar' (textfield and button in a horizontallayout)
64      editBar = new HorizontalLayout();
65      editBar.setMargin(false, false, false, true);
66      editBar.setEnabled(false);
67      addComponent(editBar);
68      // textfield
69      editor = new TextField("Item name");
70      editor.setImmediate(true);
71      editBar.addComponent(editor);
72      // apply-button
73      change = new Button("Apply", this, "buttonClick");
74      editBar.addComponent(change);
75      editBar.setComponentAlignment(change, Alignment.BOTTOM_LEFT);
76  }
77
78  public void valueChange(ValueChangeEvent event) {
79      if (event.getProperty().getValue() != null) {
80          // If something is selected from the tree, get its 'name' and
81          // insert it into the textfield
```

```
82            editor.setValue(tree.getItem(event.getProperty().getValue())
83                  .getItemProperty(ExampleUtil.hw_PROPERTY_NAME));
84            editor.requestRepaint();
85            editBar.setEnabled(true);
86        } else {
87            editor.setValue("");
88            editBar.setEnabled(false);
89        }
90    }
91
92    public void buttonClick(ClickEvent event) {
93        // If the edited value contains something, set it to be the item's new
94        // 'name' property
95        if (!editor.getValue().equals("")) {
96            Item item = tree.getItem(tree.getValue());
97            Property name = item.getItemProperty(ExampleUtil.hw_PROPERTY_NAME);
98            name.setValue(editor.getValue());
99        }
100   }
101
102   /*
103    * Returns the set of available actions
104    */
105   public Action[] getActions(Object target, Object sender) {
106       return ACTIONS;
107   }
108
109   /*
110    * Handle actions
111    */
112   public void handleAction(Action action, Object sender, Object target) {
113       if (action == ACTION_ADD) {
114           // Allow children for the target item, and expand it
115           tree.setChildrenAllowed(target, true);
116           tree.expandItem(target);
117
118           // Create new item, set parent, disallow children (= leaf node)
119           Object itemId = tree.addItem();
120           tree.setParent(itemId, target);
121           tree.setChildrenAllowed(itemId, false);
122
123           // Set the name for this item (we use it as item caption)
124           Item item = tree.getItem(itemId);
125           Property name = item.getItemProperty(ExampleUtil.hw_PROPERTY_NAME);
126           name.setValue("New Item");
127
128       } else if (action == ACTION_DELETE) {
129           Object parent = tree.getParent(target);
130           tree.removeItem(target);
131           // If the deleted object's parent has no more children, set its
132           // childrenallowed property to false (= leaf node)
133           if (parent != null) {
134               Collection<?> children = tree.getChildren(parent);
135               if (children != null && children.isEmpty()) {
136                   tree.setChildrenAllowed(parent, false);
137               }
138           }
139       }
140   }
141 }
```

```
142
```

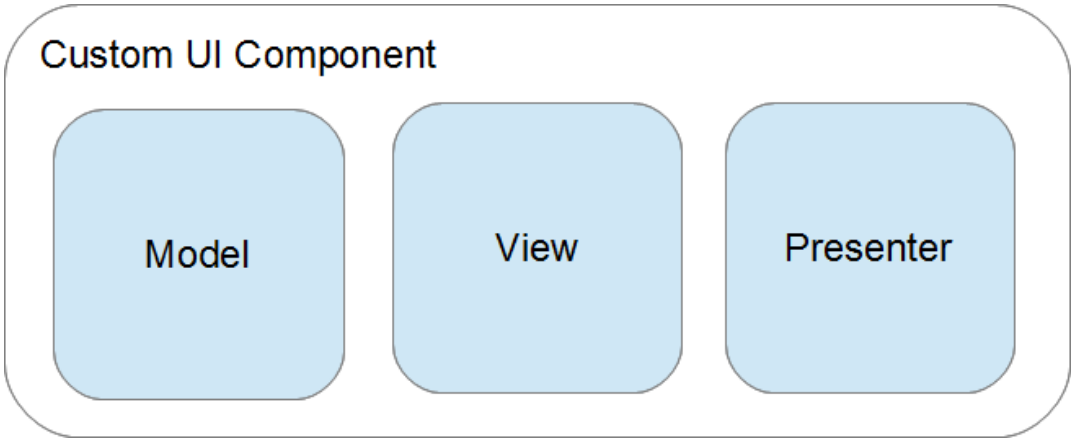
Maybe this example is a little bit long, but it it demonstrates that pure Vaadin code tends to be Spaghetti-Code which is hard to maintain, especially if the maintainer has not created the code.

The spring-mvp idea tries to decouple code and to create a structure so it is clear where to find the layout and the logic of a application:

- The model (known as Bean, ValueObject(VO) etc.) contains the data to show in the application.
- The view contains only the layout.
- The presenter contains the logic for the layout.

With this approach the code is more structured, each part is cleary arranged.

Example 1. spring-mvp



2. Prerequisites

spring-mvp is based on Vaadin 6.x, Java 1.6, Spring 3.1.2 and dellroad-stuff-vaadin 1.0.594. This addon works with Java 1.7, but there is currently no support for Vaadin 7 available.

3. Using spring-mvp

3.1. Spring configuration

First of all the basis, Spring, needs to be configured and enabled for the Vaadin web application. This configuration is not spring-mvp specific but a common Spring web application configuration.

There are several configurations necessary to get spring-mvp up and running. Spring can also be configured using Annotations, which is documented in the Spring documentation. If Maven is used (which is highly recommended), all needed dependencies are provided by spring-mvp.

All configuration examples are taken from the demo application. In the Demo-Application Spring is mainly configured using XML configuration files.

Spring can also be configured using Annotations, which is documented in the Spring documentation.

3.1.1. WEB-INF/web.xml

Set the location of the Spring configuration XML file.

```
1 <context-param>
2  <param-name>contextConfigLocation</param-name>
3  <param-value>/WEB-INF/spring/web-application-context.xml</param-value>
4 </context-param>
```

The path to the web application context configuration file can of course be customized.

Add the necessary listeners.

```
1 <listener>
2  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
3 </listener>
4 <!-- needed to set the translators locale from the browser request -->
5 <listener>
6  <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
7 </listener>
```

3.1.2. Spring application context

Define all Objects (Services, Resource-Bundles etc.) in this context if they should be available in all Vaadin application instances.

Example 2. WEB-INF/spring/web-application-context.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/
beans/spring-beans.xsd">
5
6 </beans>
7
```

3.2. spring-mvp Application

To use spring-mvp in Vaadin applications the Vaadin application needs to extend `SpringMvpVaadinApplication`.

Example 3. src/main/java/de/flexguse/vaadin/addon/springMvp/applicationAddonDemoApplication.java

```
1 public class AddonDemoApplication extends SpringMvpVaadinApplication...{}
```

Then the application needs to be known by Spring.

Example 4. WEB-INF/spring/vaadin-application-context.xml

```
1 <bean id="demoApplication"
2  class="de.flexguse.vaadin.addon.springMvp.demo.AddonDemoApplication"
3  factory-method="get">
4 </bean>
```

Warning

Do not forget to set the factory-method, otherwise the Autowiring won't work properly.

And the last necessary configuration is to configure Spring to look for annotations.

Example 5. WEB-INF/spring/vaadin-application-context.xml

```
1 <!-- enable configuration by annotation like @Autowired or @HandleSpringMvpEvent -->
2 <context:annotation-config />
```

3.3. Vaadin Server push

While dispatching spring-mvp events Vaadin applications are changed serverside. Without a Server push mechanism the changes in the Vaadin applications are not reported to the Client browsers.

It is strongly recommended to use a Server push mechanism to ensure the correct behavior of spring-mvp. In the springMvp demo application dontpush-addon-ozonelayer [https://vaadin.com/directory#addon/dontpush-ozonelayer:vaadin] was used wich performed quite well.

Currently there is no Server push available in Vaadin 7, that's the reason why spring-mvp does not support Vaadin 7 but Vaadin 6.x.

3.3.1. No Server push example

A Vaadin application was created in which a User with the role Administrator is able to send a notification message to all Vaadin application instances. Using spring-mvp this is not a hard task using the `DispatcherManger`. The DispatcherManager calls the event listener methods of all Vaadin application instances and the application the Administrator uses show the notification message.

All other applications do not show the notification immediately. The users have to click somewhere so the browser part of the Vaadin application gets awave of the notification message.

4. Event Dispatching

Eventdispatching is one of the key features of spring-mvp. There are a couple of objects involved in dispatching evenents, the mostly used are `SpringMvpDispatcher` and the `DispatcherManager`.

Dispatching an Event is simple. First of all create an Event class that extends .

Example 6. de.flexguse.vaadin.addon.springMvp.demo.ui.events.ShowArticlesViewEvent

```
1 package de.flexguse.vaadin.addon.springMvp.demo.ui.events;
2
3 import de.flexguse.vaadin.addon.springMvp.events.SpringMvpEvent;
4
5 /**
6  * Dispatch this event to show the Article View.
7  *
8  * @author Christoph Guse, info@flexguse.de
9  *
10  */
11 public class ShowArticlesViewEvent extends SpringMvpEvent {
12
13  private static final long serialVersionUID = -7682300622168995827L;
14
15  public ShowArticlesViewEvent(Object eventSource) {
16   super(eventSource, null);
17  }
18
19 }
```

After that create a EventListener which listens for the event.

Example 7. de.flexguse.vaadin.addon.springMvp.demo.AddonDemoApplicationPresenter

```
1 package de.flexguse.vaadin.addon.springMvp.demo;
2
3 ...
4 /**
5  * This Presenter contains logic for the {@link AddonDemoApplication}.
6  *
7  * @author Christoph Guse, info@flexguse.de
8  *
9  */
10 public class AddonDemoApplicationPresenter extends
11   AbstractPresenter<AddonDemoApplication> {
12
13  @Autowired
14  private ApplicationContext springContext;
15
16 ...
17  @HandleSpringMvpEvent
18  public void handleShowArticlesView(ShowArticlesViewEvent event) {
19   getView().setView(springContext.getBean(ArticlesManagement.class));
20  }
21 }
22
```

The EventListener can be any object. Just implement a method which has your `SpringMvpEvent` as attribute and annotate this method with `@HandleSpringMvpEvent`.

And then just dispatch the `ShowArticlesViewEvent` somewhere in your application.

Example 8. de.flexguse.vaadin.addon.springMvp.demo.AddonDemoApplication

```
1 package de.flexguse.vaadin.addon.springMvp.demo;
2
3 /**
4  * The Application's "main" class.
5  *
6  * @author Christoph Guse, info@flexguse.de
7  */
8 @SuppressWarnings("serial")
9 public class AddonDemoApplication extends SpringMvpVaadinApplication implements
10   View<AddonDemoApplicationPresenter> {
11
12  @Override
13  protected void initSpringApplication(ConfigurableWebApplicationContext arg0) {
14
15     ...
16   presenter.dispatchEvent(new ShowShoppingCartViewEvent(this));
17
18  }
19  ...
20 }
```

spring-mvp is designed to have a SpringMvpDispatcher in several scopes:

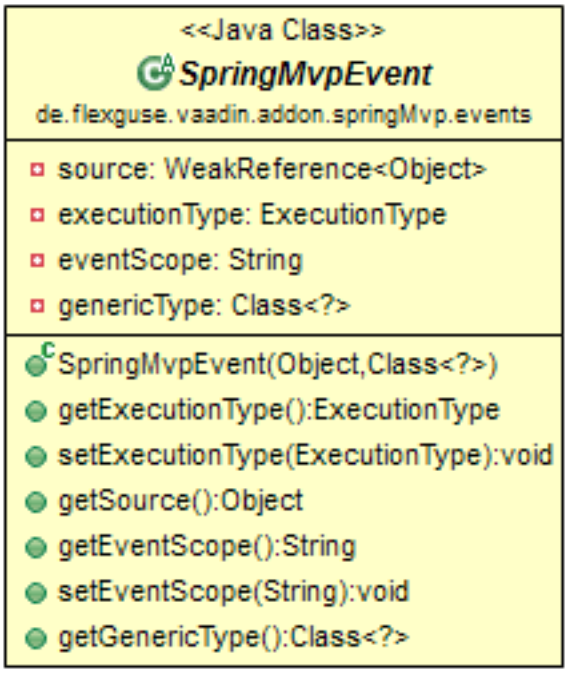
- Web Application scope: only one instance for all Vaadin application instances.
- Vaadin application scope: events dispatched in this scope are only sent to the current Vaadin application. Eventdispatching to other than the current Vaadin application instance is not possible.
- Custom scope: a SpringMvpDispatcher can be instantiated in a custom component and only lives as long the custom component lives

The different Dispatcher scopes can be achieved by configuring SpringMvpDispatcher in different Spring context configuration files.

A SpringMvpDispatcher implementation is a kind of Map in which the Key is the SpringMvpEvent class and the value is a list of event listeners.

4.1. `SpringMvpEvent`

All custom events handled by spring-mvp must extend `SpringMvpEvent`.



The abstract class `SpringMvpEvent` contains some information which is essential for spring-mvp event dispatching.

4.1.1. ExecutionType

spring-mvp provides two execution types of event handling. Setting `ExecutionType.SYNC` means the event listening methods are executed immediately and the application part which dispatched the event waits until all listening methods were called.

In case of long running listening methods this behavior is not wanted. Set `ExecutionType.ASYNC` means the listening methods are executed in a background task and the application is immediately ready to proceed.

4.1.2. EventScope

Setting the EventScope decides in the `DispatcherManager` which `SpringMvpDispatcher` is used for event listener method execution.

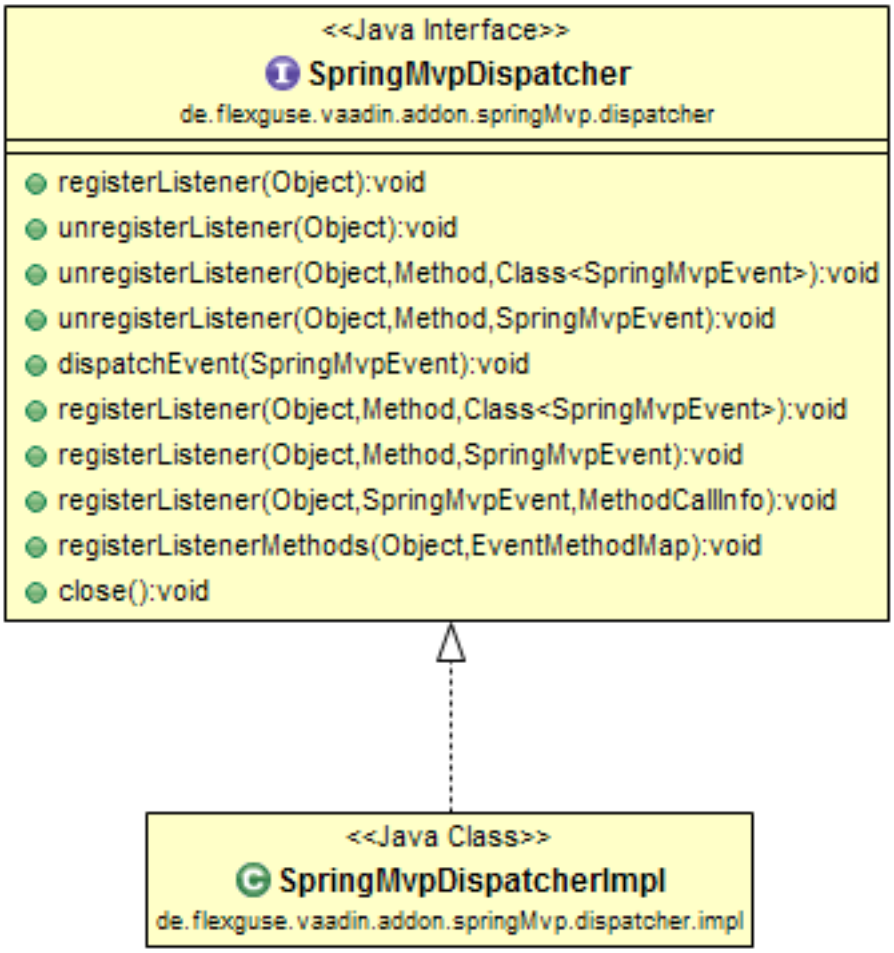
By default in spring-mvp there are two EventScopes defined: `EventScope.SpringMvpApplication` and `EventScope.AllSpringMvpApplications`. The predefined EventScopes can be easily extended by setting own EventScopes.

4.1.3. GenericType

spring-mvp allows the creation and dispatching of generic `SpringMvpEvents` (see `ModelWasAddedEvent<T>`). At runtime in Java it is not possible to get the generic information which is essential for correct event listener method registration. Therefore the generic type information must be set as attribute in the `SpringMvpEvent` class.

4.2. `SpringMvpDispatcher`

The `SpringMvpDispatcher` provides several methods to register/unregister EventListeners and to dispatch events.



Please have a look at the Javadoc to get an idea what the methods do in detail.

It is up to the user to use the `SpringMvpDispatcher` directly or to use the `DispatcherManager`.

For convenience it is advocated to use the `DispatcherManager`.

4.2.1. Configuration

The `SpringMvpDispatcher` is used for dispatching event. To dispatch events to all Vaadin application instances (from one browser to another) the `SpringMvpDispatcher` needs to be configured in the Spring context for the complete web application. In the demo application this is `WEB-INF/spring/web-application-context.xml` in Spring singleton scope.

Example 9. WEB-INF/spring/web-application-context.xml

```
1  <!-- The dispatchers and dispatcherManager used for this vaadin application -->
2  <bean id="overallSpringMvpDispatcher"
3   class="de.flexguse.vaadin.addon.springMvp.dispatcher.impl.SpringMvpDispatcherImpl"
4   destroy-method="close" scope="singleton" parent="abstractDispatcher">
5   <constructor-arg name="syncTaskExecutor" ref="commonSyncTaskExecutor" />
6   <constructor-arg name="asyncTaskExecutor" ref="commonAsyncTaskExecutor" />
7  </bean>
```

4.2.2. `AbstractEventDispatcher`

The AbstractEventDispatcher is the base class for all SpringMvpDispatcher implementations. Beside some useful methods it contains the `SpringmvpHandlerUtil` which is used to do the EventListener class introspection. Define the AbstractEventDispatcher and SpringMvpHandlerUtil only once in your Spring context. In the demo application this is done in

Example 10. springMvp-addon/src/main/resources/springMvp-context.xml

```
1 <!-- The Utility which examines objects for annotations -->
2 <bean id="springMvpUtil" class="de.flexguse.vaadin.addon.springMvp.util.SpringMvpHandlerUtil"
3  scope="singleton"/>
```

which is imported into `springMvp-demo/src/main/webapp/WEB-INF/spring/web-application-context.xml`

4.2.3. `EventHandlersCaller`

An `EventHandlersCaller` is a spring-mvp technical class manages the method calling for one `SpringMvpEvent`. There is no need to use `EventHandlerCaller` directly, but it must be defined in the Spring configuration with Spring scope "prototype".

Example 11. springMvp-addon/src/main/resources/springMvp-context.xml

```
1 <bean class="de.flexguse.vaadin.addon.springMvp.dispatcher.impl.EventHandlersCallerImpl"
2  destroy-method="cleanUp" scope="prototype" parent="abstractDispatcher"/>
```

4.2.4. TaskExecutors

In spring-mvp Events can be dispatched synchronously and asynchronously. If you know the Event causes a long running task, dispatch it asynchronously. The execution of the Events is done using the Spring `TaskExecutor` abstraction (see Spring documentation). [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/scheduling.html#scheduling-task-executor]

A `SpringMvpDispatcher` always has two `TaskExecutors`: syncTaskExecutor and asyncTaskExecutor. Configure the `TaskExecutor`for your needs. For the demo application this was done in

Example 12. springMpv-demo/src/main/webapp/WEB-INF/spring/vaadin-application-context.xml

```
1 <!-- The Spring Task executors -->
2 <bean id="syncTaskExecutor" class="org.springframework.core.task.SyncTaskExecutor" />
3
4 <bean id="asyncTaskExecutor"
5  class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
6  <property name="corePoolSize" value="5" />
7  <property name="maxPoolSize" value="10" />
8  <property name="queueCapacity" value="25" />
9 </bean>
```

4.2.5. `SpringMvpHandlerUtil`

The `SpringMvpHandlerUtil` does the Eventhandler introspection to get the methods which need to be registered in the EventHandlers. In future versions there are plans to implement caching in the `SpringMvpHandlerUtil` so it is a good idea to only have one instance per JVM.

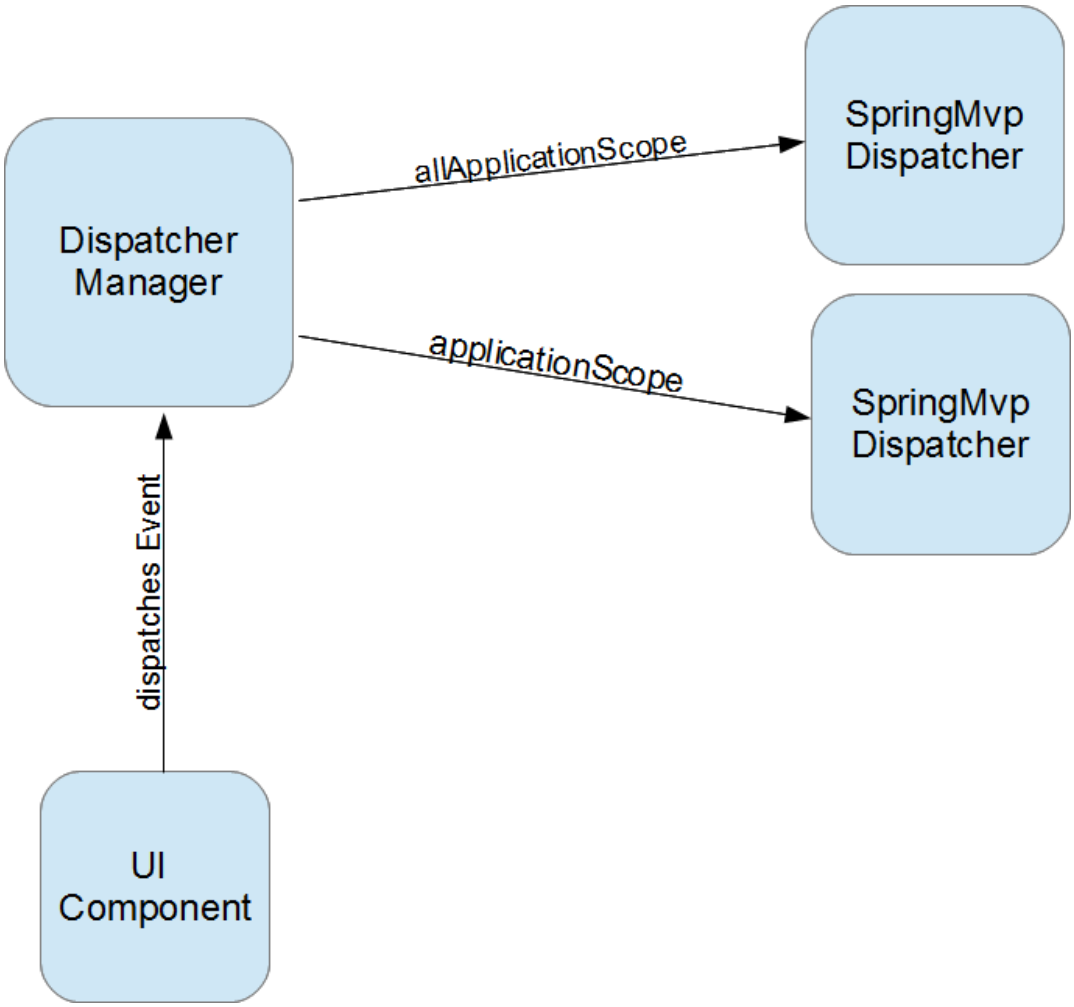
In the demo application the configuration of `SpringMvpHandler` is done in

Example 13. springMvp-addon/src/main/resources/springMvp-context.xml

```
1 <!-- The Utility which examines objects for annotations -->
2 <bean id="springMvpUtil" class="de.flexguse.vaadin.addon.springMvp.util.SpringMvpHandlerUtil"
3  scope="singleton"/>
```

4.3. `DispatcherManager`

The `DispatcherManager` is a wrapper for multiple `SpringMvpDispatcher` which allocates the correct dispatcher by the EventScope. In the current Implementation there are convenience method to have `SpringMvpDispatcher` for two EventScopes, but you are free to register more.

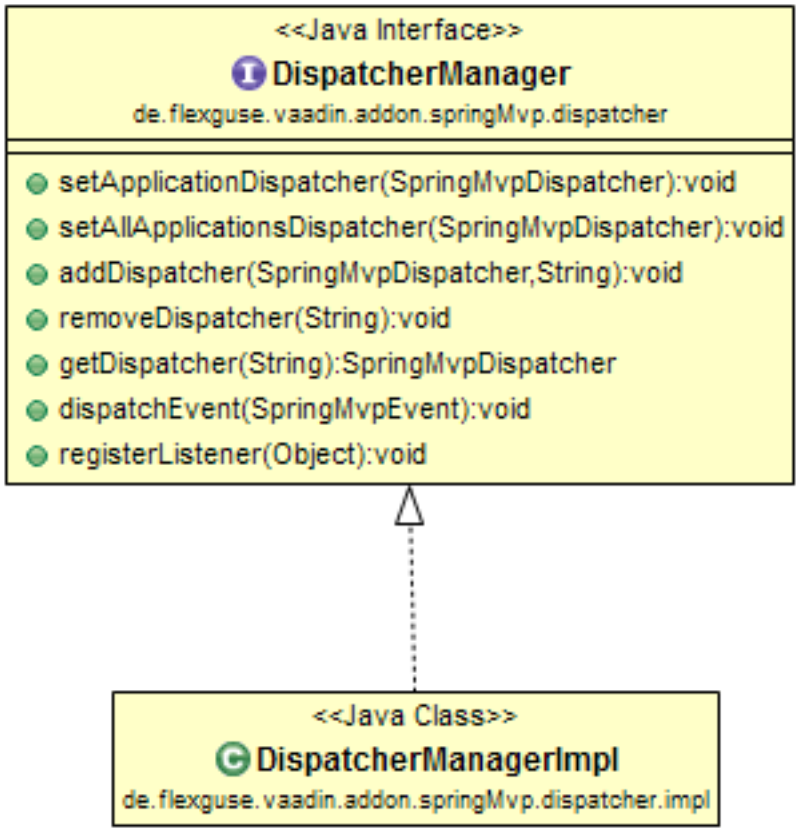


Using the `DispatcherManager` is simple. Inject it into your classes and dispatch an Event.

```
1 @Autowired
2 private DispatcherManager dispatcherManager;
3 ...
4 public void dispatchEvent(){
5     dispatcherManager.dispatcheEvent(new SpringMvpEvent());
6 }
```

Which `SpringMvpDispatcher` is used is set by the EventScope set in the `SpringMvpEvent`. By default the EventScope `EventScope.SpringMvpApplication` is set, the event is dispatched in the current Vaadin application.

If the event shall be dispatched to all Vaadin applications, set the EventScope to `EventScope.AllSpringMvpApplications`.



4.3.1. `DispatcherManager` configuration

Ensure only one instance of `DispatcherManager` exists once per Vaadin application. In the demo application it is defined in

Example 14. springMvp-demo/src/main/webapp/WEB-INF/spring/vaadin-application-context.xml

```
<bean id="dispatcherManager"
 class="de.flexguse.vaadin.addon.springMvp.dispatcher.impl.DispatcherManagerImpl">
 <property name="applicationDispatcher" ref="springMvpDispatcher" />
 <property name="allApplicationsDispatcher" ref="overallSpringMvpDispatcher" />
 <property name="springMvpHandlerUtil" ref="springMvpUtil" />
</bean>
```

4.4. Event Registration

Event registration can be done manually or automatically. For manual registration inject the `SpringMvpDispatcher` or the `DispatcherManaegr` and register a method for an event.

In some cases this can be useful, especially if the Vaadin methods

```
attach()
```

and

```
detach()
```

are overwritten in which methods are registered/unregistered at a injected `SpringMvpDispatcher`.

But for convenience it is much better to use annotations.

4.4.1. `SpringMvpBeanPostProcessor`

To get the spring-mvp annotations `@HandleSpringMvpEvent` and `@HandleSpringMvpEvents` running the `SpringMvpBeanPostProcessor` must be registered in the Spring context so each Spring controlled bean is examined for the spring-mvp annotations and eventually registerd in the `DispatcherManager`.

In the demo application the `SpringMvpBeanPostProcessor` is defined in

Example 15. springMvp-demo/src/main/webapp/WEB-INF/spring/vaadin-application-context.xml

```
1 <!-- The Spring postprocessor which handles the annotated beans -->
2 <bean class="de.flexguse.vaadin.addon.springMvp.util.SpringMvpBeanPostProcessor">
3  <constructor-arg name="dispatcherManager" ref="dispatcherManager" />
4 </bean>
```

Each time Spring instantiates a new bean the bean is searched for annotated methods and if that methods are found, the bean is registered as EventListener in the `DispatcherManager`.

Using the `SpringMvpBeanPostProcessor` means ony Spring instantiated beans are searched for annotated methods. Creating beans containing annotated methods using

```
new EventListener()
```

in your code means **NOT** the bean is automatically registered at the `SpringMvpDispatcher`.

As described before unregistering of EventListeners can be done manually. In case of using `SpringMvpBeanPostProcessor` and annotation driven event registration no unregistration is done. This means no memory lock because internally the `SpringMvpDispatcher`s use WeakReferences so the EventListener can be garbage collected even if it still registered in a dispatcher.

4.4.2. `@HandleSpringMvpEvent`

This annotation is used to register a method for a single event. With no explicitly given `eventScopes`, the method is registered in `EventScope.SpringMvpApplication` which means the method is only executed if the event was dispatched from the current Vaadin application.

```
1 @HandleSpringMvpEvent
2 public void handleOpenShoppingCartEditorEvent(final OpenModelEditorEvent<ShoppingCart> event){...}
```

If the method needs to listen for the event in several scopes, set the `eventScopes` explicitely.

```
1 @HandleSpringMvpEvent(eventScopes = { EventScope.AllSpringMvpApplications,
EventScope.SpringMvpApplication })
2 public void handleArticleWasAdded(ModelWasAddedEvent<Article> event) {
3  eventList.addEvent(event);
4 }
```

4.4.3. `@HandleSpringMvpEvents`

In some cases a EventListener method shall be called for several events. In this case the method argument must be of type `SpringMvpEvent` and the method must be annotated with `@HandleSpringMvpEvents`. If no `EventScope` is given, the default listening scope is `EventScope.SpringMvpApplication`.

```
1 @HandleSpringMvpEvents(value = {
2   ShowErrorMessageEvent.class, ShowHumanizedMessageEvent.class,
3   ShowTrayNotificationEvent.class, ShowWarningMessageEvent.class,
4   OpenMessageEditorEvent.class, ShowArticlesViewEvent.class,
5   ShowShoppingCartViewEvent.class })
6 public void handleSpringMvpEvent(SpringMvpEvent event) {
7
8  eventList.addEvent(event);
9 }
```

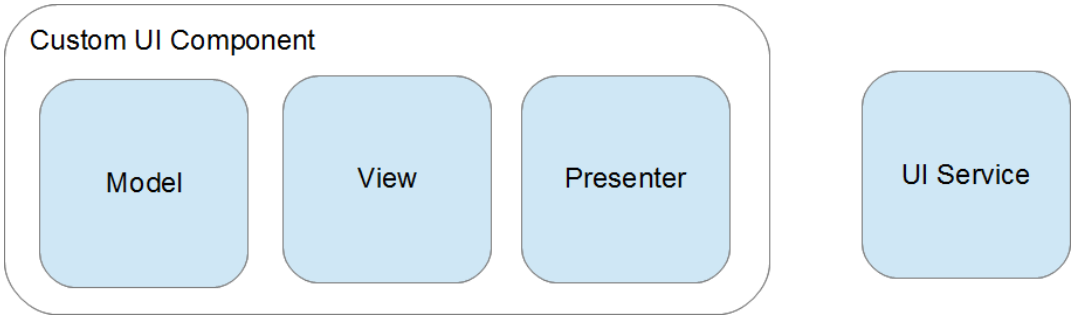
The method in this example is called if any of the given events is dispatched.

Additionally the listening `EventScope` can be given.

```
1  @HandleSpringMvpEvents(eventScopes = { EventScope.AllSpringMvpApplications,
2    EventScope.SpringMvpApplication }, value = {
3    ShowErrorMessageEvent.class, ShowHumanizedMessageEvent.class,
4    ShowTrayNotificationEvent.class, ShowWarningMessageEvent.class,
5    OpenMessageEditorEvent.class, ShowArticlesViewEvent.class,
6    ShowShoppingCartViewEvent.class })
7  public void handleSpringMvpEvent(SpringMvpEvent event) {
8
9    eventList.addEvent(event);
10 }
```

5. Model-View-Presenter (MVP)

The MVP part in spring-mvp is nothing which must be implemented but it is a general idea how to structure a complex application codebase so it is easy to find layout or functionality for maintenance. The previously shown MVP figure needs some extention.



5.1. Model

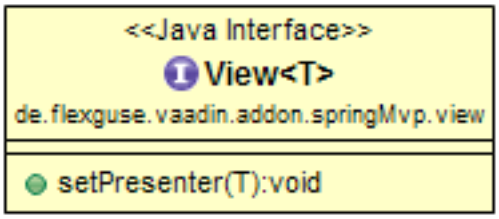
The Model is a Value Object (VO) or just Bean. VOs and Beans are objects which hold business specific data and mostly no logic. In the demo application these VOs are `Article`, `Model` and `ShoppingCart`.

In spring-mvp Models do not need to extend or implement anything.

5.2. View

In spring-mvp the View is a Vaadin UI component. A View can be the complete layout of an application or maybe a PopUp window or a custom Table component.

In a View there should be only the layout and normally no logic. The logic is implemented in the Presenter which is part of each View. The setter for the Presenter is the only mandatory method in the View.



An example for a View is `AddonDemoApplication`:

```
1 public class AddonDemoApplication extends SpringMvpVaadinApplication implements
2   View<AddonDemoApplicationPresenter> {
3 ...
4
5  /**
6   * If you want to autowire the Presenter, add @Autowired to the setter so
7   * the registration of the View is done in this method.
8   * <p>
9   * Normally it would be a good idea to configure the presenter in the Spring
10   * xml configuration, this does not work for the Application.
11   * </p>
12   */
13  @Autowired
14  public void setPresenter(AddonDemoApplicationPresenter presenter) {
15   this.presenter = presenter;
16   presenter.setView(this);
17  }
18 ...
19 }
```

While setting the Presenter in the View the View needs to be set in the Presenter.

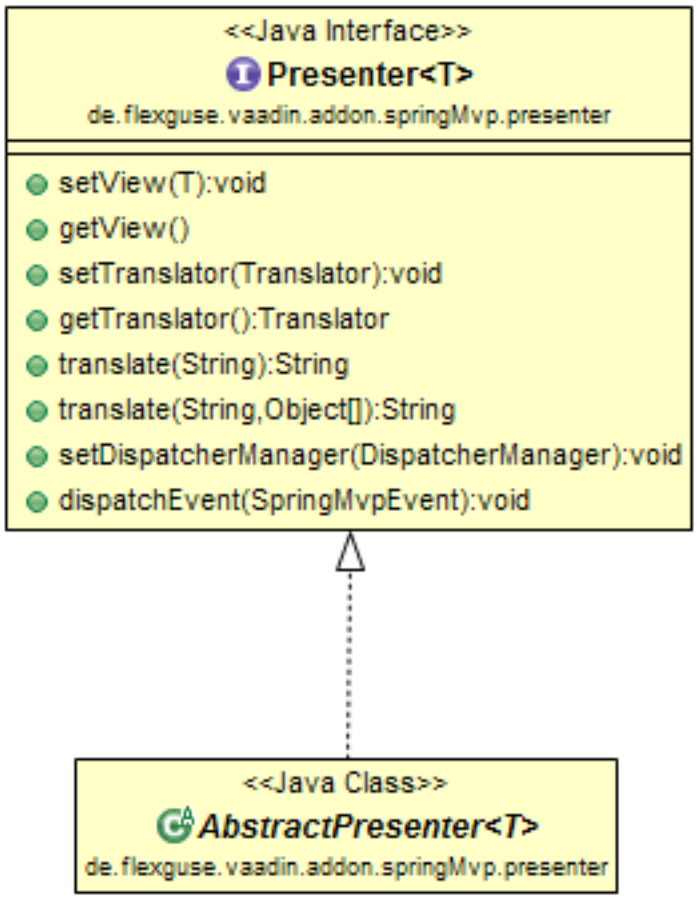
To use Spring dependency injection all Views need to be configured as Spring beans. It is essential to set the Spring scope "prototype" so every time a new instance of the View is created. If another scope is set, like "singleton" for reusing View components, the View components are not properly shown in the Vaadin application.

```
1  <bean id="articlesManagement"
2   class="de.flexguse.vaadin.addon.springMvp.demo.ui.component.articles.ArticlesManagement"
3   scope="prototype">
4   <property name="presenter" ref="articlesManagementPresenter" />
5   <property name="translationPrefix" value="articles" />
6  </bean>
```

5.3. Presenter

The Presenter contains all the logic for Views. Registering Vaadin event listeners on Vaadin components, implementing spring-mvp Event listener methods and other logic can be bulky and should not be mixed with the layout and should be clearly separated into the Presenter.

In spring-mvp there is an `AbstractPresenter` which contains some logic needed by all Presenters. All Presenters in a spring-mvp application must extend `AbstractPresenter`.



The Presenter is a good place to implement View specific Event listener methods like in the `AddonDemoApplicationPresenter`.

```
1 /**
2  * This Presenter contains logic for the {@link AddonDemoApplication}.
3  *
4  * @author Christoph Guse, info@flexguse.de
5  *
6  */
7 public class AddonDemoApplicationPresenter extends
8   AbstractPresenter<AddonDemoApplication> {
9
10  @Autowired
11  private ApplicationContext springContext;
12
13  @HandleSpringMvpEvent
14  public void handleShowShoppingCatView(ShowShoppingCartViewEvent event) {
15   getView().setView(springContext.getBean(ShoppingCartManagement.class));
16  }
17
18  @HandleSpringMvpEvent
19  public void handleShowArticlesView(ShowArticlesViewEvent event) {
20   getView().setView(springContext.getBean(ArticlesManagement.class));
21  }
22 }
```

This example is quite short but shows how the Presenter communicates with the View: using the `getView()` method.

A more complex Presenter is `ModelManagementPresenter` which is too long to show here but it demonstrates how the View logic can grow enormously.

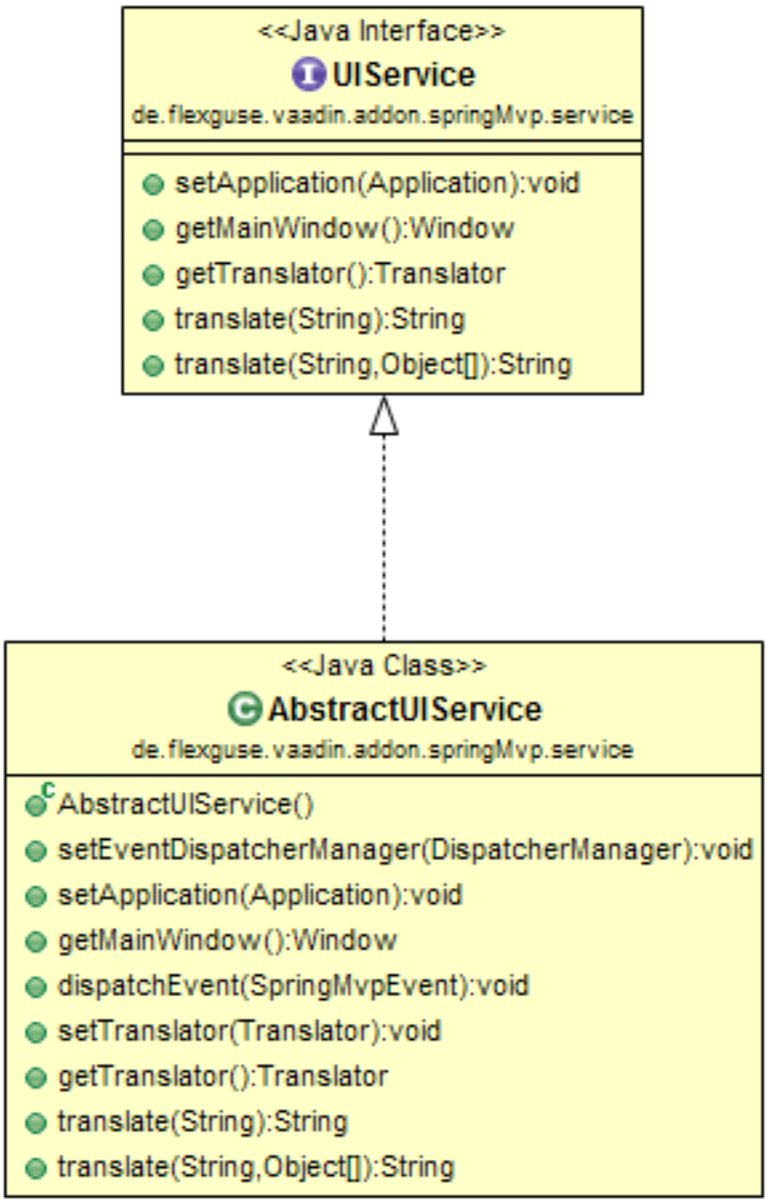
Like the Views Presenters needs to be configured in Spring with scope "prototype". This ensures each View instance has it's own Presenter instance.

```
1 <bean id="articlesManagementPresenter"
2  class="de.flexguse.vaadin.addon.springMvp.demo.ui.component.articles.ArticlesManagementPresenter"
3  scope="prototype" parent="abstractPresenter">
4  <property name="modelService" ref="uiArticlesService" />
5 </bean>
```

5.4. UI Service

The UI Service is the place where the backend services (for handling the Models) meet the UI. Some could say this layer is not necessary but a UI Service gives us the possibility to do custom exception handling and the UI Service is a very good place to implement event listener methods.

All UI Services in a spring-mvp application should extend `AbstractUIService`.



In the spring-mvp demo application Models are always edited in PopUps. UI Services are the perfect central place to implement the event listening methods which open the Editor PopUps. UI Services normally have a Spring singleton scope, they exist all the time the Vaadin application exist and are always ready to open the editor whereas Views and Presenters may have another Spring scope and do not exist the whole time.

An example for a UI Service is `UIArticlesService`.

```
1 /**
2  *
3  */
4 package de.flexguse.vaadin.addon.springMvp.demo.ui.service;
5
6 import java.util.List;
7
8 import org.springframework.beans.factory.annotation.Autowired;
```

```
 9
10 import com.vaadin.data.Validator.InvalidValueException;
11
12 import de.flexguse.vaadin.addon.springMvp.annotations.HandleSpringMvpEvent;
13 import de.flexguse.vaadin.addon.springMvp.demo.backend.model.Article;
14 import de.flexguse.vaadin.addon.springMvp.demo.backend.service.ArticleService;
15 import de.flexguse.vaadin.addon.springMvp.demo.ui.component.articles.ArticleForm;
16 import de.flexguse.vaadin.addon.springMvp.demo.ui.component.model.ModelFormInterceptor;
17 import de.flexguse.vaadin.addon.springMvp.demo.ui.events.DeleteModelEvent;
18 import de.flexguse.vaadin.addon.springMvp.demo.ui.events.OpenModelEditorEvent;
19 import de.flexguse.vaadin.addon.springMvp.events.SpringMvpEvent.ExecutionType;
20 import de.flexguse.vaadin.addon.springMvp.events.messages.ShowTrayNotificationEvent;
21 import de.flexguse.vaadin.addon.springMvp.events.model.ModelWasAddedEvent;
22 import de.flexguse.vaadin.addon.springMvp.events.model.ModelWasDeletedEvent;
23 import de.flexguse.vaadin.addon.springMvp.events.model.ModelWasUpdatedEvent;
24 import de.flexguse.vaadin.addon.springMvp.service.AbstractUIService;
25 import de.flexguse.vaadin.addon.springMvp.service.UIService;
26 import de.steinwedel.vaadin.MessageBox;
27 import de.steinwedel.vaadin.MessageBox.ButtonType;
28
29 /**
30  * This ArticleService is designed to be used directly in the UI components.
31  *
32  * @author Christoph Guse, info@flexguse.de
33  *
34  */
35 public class UIArticlesService extends AbstractUIService implements UIService,
36   UiModelService<Article> {
37
38  @Autowired
39  private ArticleService articleService;
40
41  @Autowired
42  private ArticleForm articleForm;
43
44  /**
45   * This method gets all Articles from the backend persistence.
46   *
47   * @return
48   */
49  public List<Article> getAllModels() {
50
51   return articleService.getAllActiveArticles();
52
53  }
54
55  /**
56   * This method listens for an {@link OpenModelEditorEvent} and opens an
57   * article editor containing the {@link Article} given by the event.
58   *
59   * @param event
60   */
61  @SuppressWarnings("static-access")
62  @HandleSpringMvpEvent
63  public void handleOpenArticleEditorEvent(
64    final OpenModelEditorEvent<Article> event) {
65
66   articleForm.setArticle(event.getModel());
67
68   String editorTitle = translate("articles.editor.title.new");
```

```
69  if (!event.isNewModel()) {
70   editorTitle = translate("articles.editor.title.edit");
71  }
72
73  final MessageBox messageBox = new MessageBox(getMainWindow(), "45%",
74    "450px", editorTitle, MessageBox.Icon.NONE, articleForm,
75    MessageBox.BUTTON_DEFAULT_ALIGNMENT,
76    new MessageBox.ButtonConfig(ButtonType.CANCEL,
77      translate("cancel")), new MessageBox.ButtonConfig(
78      ButtonType.SAVE, translate("save")));
79  final ModelFormInterceptor interceptor = new ModelFormInterceptor();
80  messageBox.VISIBILITY_INTERCEPTOR = interceptor;
81  messageBox.show(true, new MessageBox.EventListener() {
82
83   private static final long serialVersionUID = 2948376877660010667L;
84
85   @Override
86   public void buttonClicked(ButtonType buttonType) {
87    if (buttonType.equals(ButtonType.SAVE)) {
88     try {
89      articleForm.commit();
90      articleService.saveArticle(articleForm.getArticle());
91
92      if (event.isNewModel()) {
93       ModelWasAddedEvent<Article> event = new ModelWasAddedEvent<Article>(
94         this, Article.class);
95       event.setAddedModel(articleForm.getArticle());
96       dispatchEvent(event);
97
98       // show added tray notification
99       ShowTrayNotificationEvent trayEvent = new ShowTrayNotificationEvent(
100        this, translate("articles.added.title"),
101        translate("articles.added.info",
102          new Object[] { articleForm
103            .getArticle().getName() }));
104      trayEvent.setExecutionType(ExecutionType.ASYNC);
105      dispatchEvent(trayEvent);
106
107     } else {
108      ModelWasUpdatedEvent<Article> event = new ModelWasUpdatedEvent<Article>(
109        this, Article.class);
110      event.setUpdatedModel(articleForm.getArticle());
111      dispatchEvent(event);
112
113      // show updated tray notification
114      ShowTrayNotificationEvent trayEvent = new ShowTrayNotificationEvent(
115        this, translate("articles.updated.title"),
116        translate("articles.updated.info",
117          new Object[] { articleForm
118            .getArticle().getName() }));
119      trayEvent.setExecutionType(ExecutionType.ASYNC);
120      dispatchEvent(trayEvent);
121     }
122
123     interceptor.setCloseAble(true);
124     messageBox.close();
125
126    } catch (InvalidValueException e) {
127     interceptor.setCloseAble(false);
128    }
```

```
129
130     } else {
131       interceptor.setCloseAble(true);
132       messageBox.close();
133
134     }
135
136   }
137  });
138
139 }
140
141  @HandleSpringMvpEvent
142  public void handleDeleteArticleEvent(final DeleteModelEvent<Article> event) {
143
144   MessageBox deleteConfirmation = new MessageBox(getMainWindow(),
145     translate("articles.delete.confirmation.title"),
146     MessageBox.Icon.QUESTION, translate(
147       "articles.delete.confirmation.question",
148       new Object[] { event.getToDelete().getName() }),
149     new MessageBox.ButtonConfig(ButtonType.NO, translate("no")),
150     new MessageBox.ButtonConfig(ButtonType.YES, translate("yes")));
151   deleteConfirmation.show(true, new MessageBox.EventListener() {
152
153    private static final long serialVersionUID = -7925625315552580719L;
154
155    @Override
156    public void buttonClicked(ButtonType buttonType) {
157     if (buttonType.equals(ButtonType.YES)) {
158      articleService.deleteArticle(event.getToDelete());
159
160      // dispatch deleted event
161      ModelWasDeletedEvent<Article> deletedEvent = new ModelWasDeletedEvent<Article>(
162        this, Article.class);
163      deletedEvent.setDeletedModel(event.getToDelete());
164      dispatchEvent(deletedEvent);
165
166      // show notification
167      ShowTrayNotificationEvent notificationEvent = new ShowTrayNotificationEvent(
168        this, translate("articles.deleted.title"),
169        translate("articles.deleted.info",
170          new Object[] { event.getToDelete()
171            .getName() }));
172      notificationEvent.setExecutionType(ExecutionType.ASYNC);
173      dispatchEvent(notificationEvent);
174     }
175
176    }
177   });
178
179  }
180
181 }
182
```

As you can see the `UIArticleService` contains the backend `ArticleService` and two event handler methods for opening editors in PopUps or confirmation dialogs.

UI Services normally have a singleton scope for a Vaadin application. This means each Vaadin application instance contains only one instance of the UI Service.

In the spring-mvp application all UI Services are configured in one Spring context configuration file.

Example 16. springMvp-demo/src/main/webapp/WEB-INF/spring/services/ui-services-context.xml

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <beans xmlns="http://www.springframework.org/schema/beans"
 3  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/
beans/spring-beans.xsd">
 5
 6  <!-- Set the scope of the UI service to singleton so there is only one instance
 7   of the service per Vaadin application -->
 8  <bean id="abstractUiService"
 9   class="de.flexguse.vaadin.addon.springMvp.service.AbstractUIService"
10   scope="singleton" abstract="true">
11    <property name="translator" ref="translator" />
12    <property name="eventDispatcherManager" ref="dispatcherManager" />
13  </bean>
14
15     <bean id="uiShoppingCartService"
16   class="de.flexguse.vaadin.addon.springMvp.demo.ui.service.UIShoppingCartService"
17   scope="singleton" parent="abstractUiService" />
18
19  <bean id="uiArticlesService"
20   class="de.flexguse.vaadin.addon.springMvp.demo.ui.service.UIArticlesService"
21   scope="singleton" parent="abstractUiService" />
22
23  <bean id="uiMessageService"
24   class="de.flexguse.vaadin.addon.springMvp.demo.ui.service.UIMessageService"
25   scope="singleton" parent="abstractUiService" />
26
27 </beans>
```

6. spring-mvp additional features

spring-mvp provides additional features which makes Vaadin application development easier.

6.1. Application notifications

Vaadin provides the possibility to show different notification types. spring-mvp provides events for all different notification types. All notification events take the title and the message as constructor arguments.

The event handler methods are implemented in `SpringMvpVaadinApplication` which listen for `EventScope.SpringMvpApplication` and `EventScope.AllSpringMvpApplications`.

6.1.1. `ShowErrorMessageEvent`

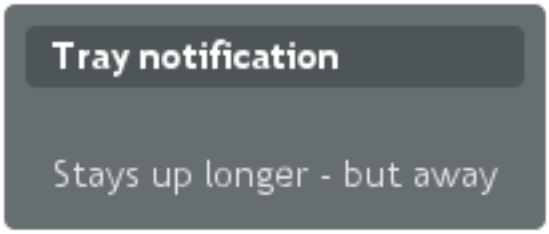
Dispatching this event shows an Error message.

6.1.2. `ShowHumanizedMessageEvent`

Dispatching this event shows a notification.

6.1.3. `ShowTrayNotificationEvent`

Dispatching this event shows a Tray notification.



6.1.4. `ShowWarningMessageEvent`

Dispatching this event shows a warning message.

6.2. Model events

Let's say there is a need to build an application which is aware of Model changes (like the the springMvp-demo application is).

In this scenario every time a model was added, updated or deleted an event is dispatched. For this three szenarios generic events were created.

All following events have the event scope `EventScope.AllSpringMvpApplications` and the execution type `ExecutionType.ASYNC`. Normally they are dispatched in the `UIServices`.

6.2.1. `ModelWasAddedEvent`

Dispatch this event if the Model was newly created and added.

```
1 ModelWasAddedEvent<Article> event = new ModelWasAddedEvent<Article>(this, Article.class);
2       event.setAddedModel(articleForm.getArticle());
3       dispatchEvent(event);
```

6.2.2. `ModelWasDeletedEvent`

Dispatch this event if the Model was deleted.

```
1 ModelWasDeletedEvent<Article> deletedEvent = new ModelWasDeletedEvent<Article>(this, Article.class);
2     deletedEvent.setDeletedModel(event.getToDelete());
3     dispatchEvent(deletedEvent);
```

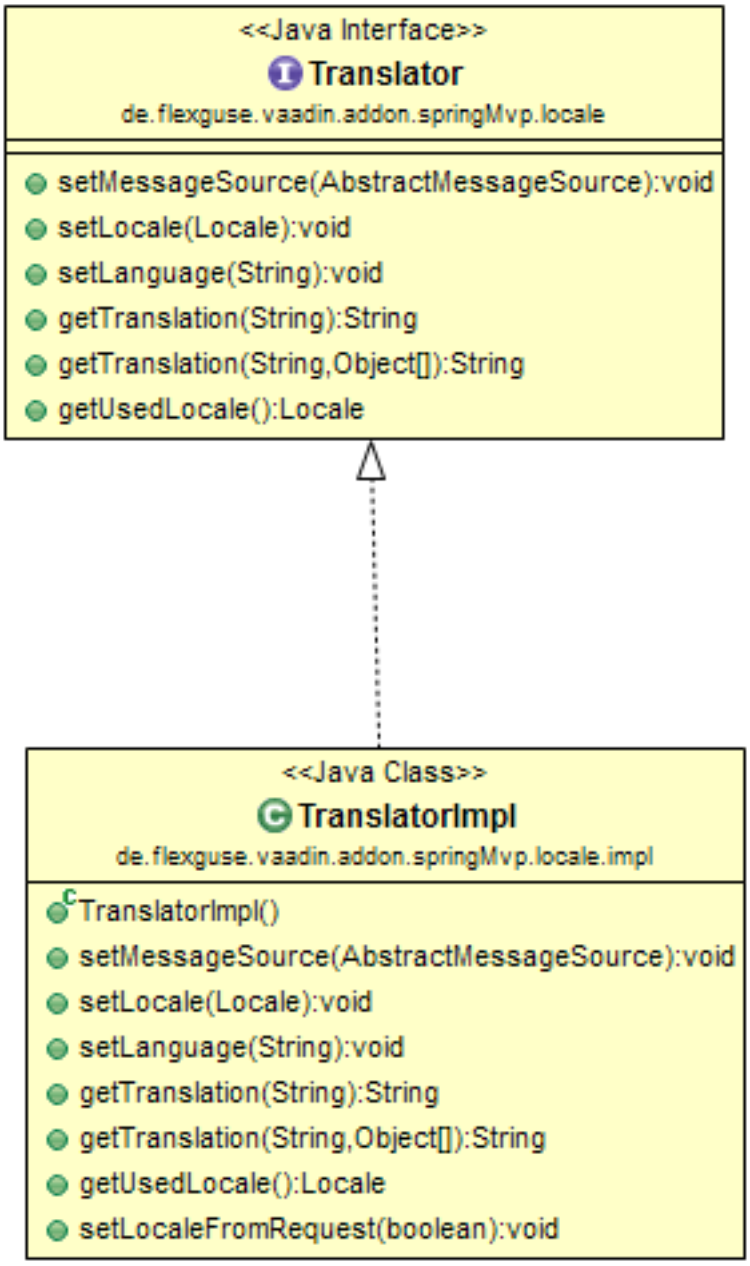
6.2.3. `ModelWasUpdatedEvent`

Dispatch this event if the Model was updated.

```
1 ModelWasUpdatedEvent<Article> event = new ModelWasUpdatedEvent<Article>(this, Article.class);
2       event.setUpdatedModel(articleForm.getArticle());
3       dispatchEvent(event);
```

6.3. Internationalization

Most production ready applications need to be available in several languages. spring-mvp provides a `Translator` which helps to get translation from resource bundles utilizing the Spring message source.



To use the spring-mvp `Translator` define a Spring MessageSource in the web application scope.

Example 17. springMvp-demo/src/main/webapp/spring/web-application-context.xml

```
1  <!-- the Spring ResourceBundleSource to access the ResourceBundles containing
2   the translations for the application -->
3  <bean id="resourceBundleMessageSource"
4   class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
5   <property name="basenames" value="WEB-INF/locale/demo" />
6   <property name="cacheSeconds" value="60" />
7  </bean>
```

The `Translator` itself must be defined in Vaadin application scope because it is possible each Vaadin application uses another locale.

Example 18. springMvp-demo/src/main/webapp/spring/vaadin-application-context.xml

```
1 <!-- The Translator for this application -->
2 <bean id="translator"
3  class="de.flexguse.vaadin.addon.springMvp.locale.impl.TranslatorImpl"
4  scope="singleton">
5  <property name="messageSource" ref="resourceBundleMessageSource" />
6  <property name="localeFromRequest" value="true" />
7 </bean>
```

If the Vaadin application uses authentication and user profiles the locale in the `Translator` can be set accordingly to the usersettings. In case of the springMvp-demo application there is no authentication and the locale is taken from the HTTP request.

Make the HTTP request available for `TranslatorImpl` by adding a listener to web.xml.

Example 19. springMvp-demo/src/main/webapp/web.xml

```
1 <!-- needed to set the translators locale from the browser request -->
2 <listener>
3  <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
4 </listener>
```

After having the `Translator` configured, it can be autowired to each place it is needed (or taken i.e. from `Presenter` or `UIService`).

A missing resource in the resource bundle does not cause an exception but an information which resource is missing for which locale.