Toolkit Productivity Tools v 1.0

Developer Handbook

Table of Contents

Introduction

Toolkit Productivity Tools (here and later - TPT) is a companion library to an excellent rich UI web based java framework Vaadin. The main's TPT goal is to provide most common functions, elements, patterns and classes, which are typically used with any Vaadin-based application but is not present in Vaadin core.

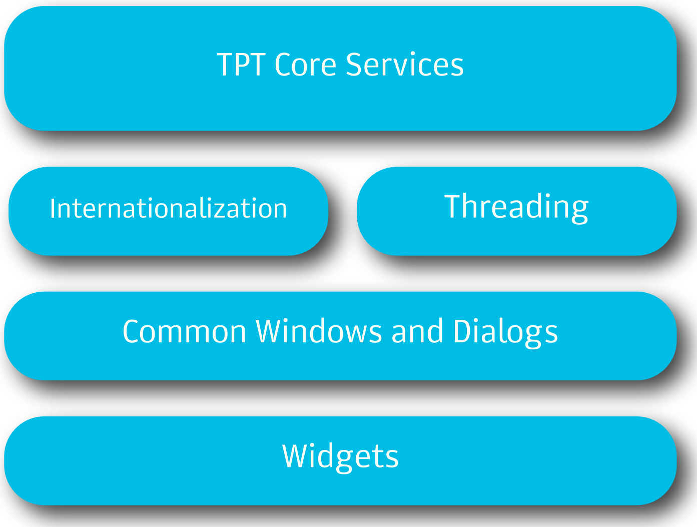
TPT has born in Alee Software, after a number of real Vaadin-based web applications was developed and authors discovered that on each startup, a number of common things are made. So, they was moved into the separate library to be commonly used in all projects. Later on it was decided to publish this library as a stand-alone open source project to the net.

TPT provides the following commonly used functions for the Vaadin-based application:

- Internationalization support for widgets and html layouts
- Application instance services and multi-threading
- Common dialogs collection
- Common windows collection
- Widgets collection

Architecture

TPT consists in several layers - the core, the internationalization, the common dialogs and windows collection, the widgets collection. In order to use any of TPT functions, a core must be attached to the project.



Core services replaces the standard Vaadin's Application class with a TPTApplication, which implements a number of commonly used patterns and provides a number of services for the actual ancestor.

Internationalization module provides services for automatic management of dictionaries and translation of the user strings. In addition, it provides internationalization support for Vaadin's CustomLayout class to allow internationalization of html layouts.

Threading module provides function to run a server-side threads with keeping the automatic management of Vaadin application context, making it (and all functions related to) transparently accessible from a thread as it was a main application thread.

Common windows is a set of enhanced Vaadin Window classes, adding the extra functionality, such as automatic handling of keystrokes, setting the html background, etc.

Common dialogs is a set of message and input dialogs, very similar to Swing's JOptionPane functions.

Configuring an application to use TPT

TPT consists from the single self-container JAR file which needs to be added to the application classpath (both runtime and development). As TPT does not contain any client-side code, GWT widget recompilation and widgetset change is not required. You may use TPT with any custom widgetset without any extra modifications.

In order to use any of TPT functions, it's core must be attached to the application project. To do this, simply replace Vaadin's original Application class with the TPTApplication in the definition of your application class as well as overridden init() method to applicationInit() :

```
public MyApplication extends Application
{
      @Override
      public void init()
      {
              ...
      }

      ...
}
```

```
public MyApplication extends TPTApplication
{
      @Override
      public void applicationInit()
      {
              ...
      }

      ...
}
```

In TPTApplication, method init() is overridden internally to perform initialization of TPT core services, so you must to move your own init code into the applicationInit() method which will be automatically called by TPT on application startup. But if you want to use the exact init(), you still can do this with the only one requirement: you must delegate super.init() call in the first line on your init code to let TPT core services completely initialize.

That's all steps you have to perform in order to start using the TPT in your new or existing application.

TPT components and functions reference

Application core services

Just by attaching the TPT to your project, you already get the set of useful functions and patterns which may be used anywhere in your application and save your coding time.

Getting an application instance

Calling static method TPTApplication.getCurrentApplication() from any part of your application will always return you the instance of your application. This is called as "ThreadLocal" pattern on the Vaadin's wiki and implemented by attaching a transaction listener to your application as well as managing application instances on transaction start and end. TPT does this all for your and you only need to call the getCurrentApplication() method to get an instance to your application.

Application-specific one-time initialization

applicationInit() or just init() method in Vaadin is called on each application instance startup. In some cases you may want to perform some common for all instances initialization, e.g. perform this only one time, when first application is started. By using TPT, you just override the firstApplicationInit() method and place your extra initialization code there. This method will be called when only the first instance of your application is started and after the original init() / applicationInit() method. For all subsequent application startups, this method will be skipped. Note, that on multi-server (clustered) environments, the firstApplicationInit() method may be, however, called several times - one time per each server in a cluster. So you have to check such situations in your code yourself.

Application instance aware threads

If you just start a new thread from within the application, the automatic reference discovery via TPTApplication.getCurrentApplication() will be broken as application instance manager does not know anything about this thread and thus cannot provide an application instance for it when asked. To workaround such situations, you need to use TPTApplication's invokeLater(Runnable r) method:

```
        TPTApplication.getCurrentApplication().invokeLater( new Runnable() {
                    public void run()
                    {
                          // thread actual code
                    }
             });
```

In this thread you can now freely use such constructions as TPTApplication.getCurrentApplication() - reference between application instance and this thread will be then automatically maintained.

Transaction start and end callbacks

Some applications require to catch the transaction life cycle events to get an HttpServletRequest object or perform another operations. When your application object extends TPTApplication, it is already has the transaction listeners attached, so you just need to override the transactionStart(...) and transactionEnd(...) methods:

```
@Override
public void transactionStart ( Application application, Object o )
{
      super.transactionStart ( application, o );
      // your code goes here
}

@Override
public void transactionEnd ( Application application, Object o )
{
      super.transactionEnd ( application, o );
      // your code goes here
}
```

Note, that you must delegate call to super implementation of the corresponding methods as a first line in order not to break TPT functionality, right as described in the code example above.

Closing the application

Close your application as usual, by calling the method close() of your application object. In some cases, you need to put your own code into the close() method as it is also called automatically by the Vaadin when application closes because of the session inactivity. TPT also manages it's own shutdown code on application close, so if you'll be overriding the close() method, do not forget to delegate the method call to the super implementation after your custom shutdown code:

```
@Override
public void close()
{
      // your shutdown code goes here
      super.close();
}
```

Internationalization

TPT provides a pre-built support for managing internationalization resources for your application. Internationalization breaks up to two sections:

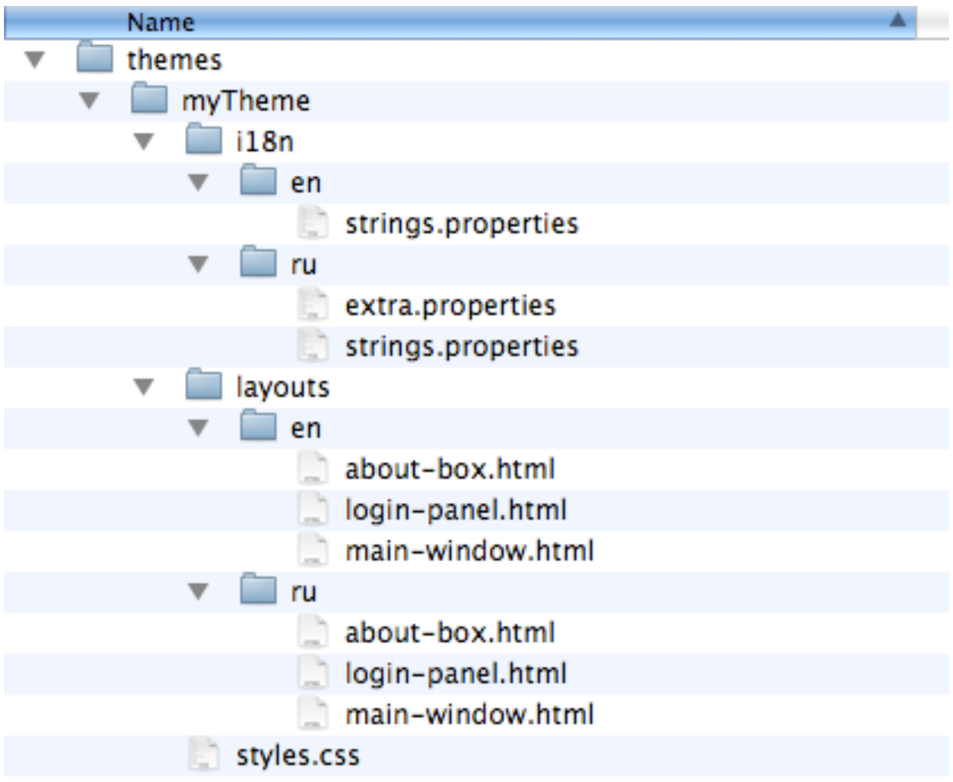
- International versions of application messages and other strings
- International versions of application layout (CustomLayout) files

When properly used, the above two parts are automatically managed by the TPT in your application and you just need to provide the set of resource files for different languages.

Creating and managing the resource files

Resource files are bound to the application theme, so you'll need to create a new theme for your application, if it is not created yet. If you do not need custom styling, you may just reference the original Runo or Reindeer theme in your custom theme CSS file - it is not important for internationalization.

Resource files for the localized messages and strings are stored in the special i18n folder, inside the root of your theme folder. Inside the i18n folder, you must create a subfolders for every language you want to support. Language folder name must be the same as language code. TPT does not use full notation such as EN_us or RU_ru, so for supporting english and russian locales, just create two subfolders "en" and "ru".



```
Name
themes
  myTheme
    i18n
      en
        strings.properties
      ru
        extra.properties
        strings.properties
    layouts
      en
        about-box.html
        login-panel.html
        main-window.html
      ru
        about-box.html
        login-panel.html
        main-window.html
    styles.css
```

Inside the language subfolder you may place any number of standard java property files with the key=value pairs of your localized strings. Place any number of property files with any names. TPT will scan those folders on application startup and load all property files into the memory for further querying.

For the CustomLayout files, you just create subfolders with the language code names inside the standard “layouts” folder of your custom theme. There a standard layout html files must be present. For custom layout, you need to have the equal files in all language subfolders - when you’ll use the CustomLayout (described below), TPT will automatically look html layout file inside the appropriate language subfolder instead of the root “layouts” folder. And only if the layout file will not be found in the current and default language subfolders, TPT will look for the same file in the root “layouts” folder of your theme.

Initializing the internationalization module

Internationalization module is initialized fully automatically when your TPT application starts, you don’t need to do anything extra. However, you may want to set the default language code, to instruct TPT what language dictionary to query for translation if specified key is not found in the current language dictionary.

The default language is set by calling the setDefaultLanguage() method of TPT’s translation manager dictionary. The following code sets the “Russian” as default language:

```
TM.getDictionary ().setDefaultLanguage ( "ru" );
```

Querying for translated string

In any place of your application, when you want to get a translated version of some text, call the TPT’s translation manager get(…) method:

```
String appTitle = TM.get ( “app.title” );
String appInfo = TM.get ( “app.info” );
```

In the example above, the key “app.title” will be first searched in the current locale dictionary. In case the key will not be found, it will be then searched in the default language dictionary. In case key will not be found even there, the exact key will be returned back as a translated string. This allows the application developer easily detect what translation is missed.

Translating custom layout templates

Custom layout template files can also be translated. As described in previous chapter, translated versions of the layout file are placed inside the language subfolders of the “layouts” folder of your theme.

To load the appropriate template, just use the TPT specific version of CustomLayout class:

```
TranslatableCustomLayout customLayout = new TranslatableCustomLayout ( “main-
window.html );
```

The template file “main-window.html” will be searched in your theme `layouts/<current language>` folder, then in `layouts/<default language>` folder and finally in the `layouts` folder.

Common Dialogs

Common dialogs package provides the possibility for quick creation and management of a standard message and input modal dialogs - to show a informational or warning message, confirmation dialog, data input dialog. Also the package provides the base framework for creating of custom dialogs.

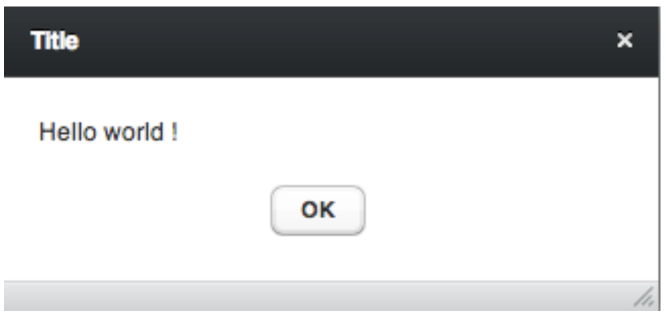
Common dialogs package is located at `eu.livotov.tpt.gui.dialogs`

Option Dialogs

Option dialogs displays a modal message box with the title and text, also providing user a set of options to do: for example, to click an “OK” button for an informational message, answer “Yes” or “No” for question and so on.

Option dialogs are powered by the class OptionDialog. Once constructed, you can use the set of dialog methods to display different kinds of dialogs.

Simple Message Dialog

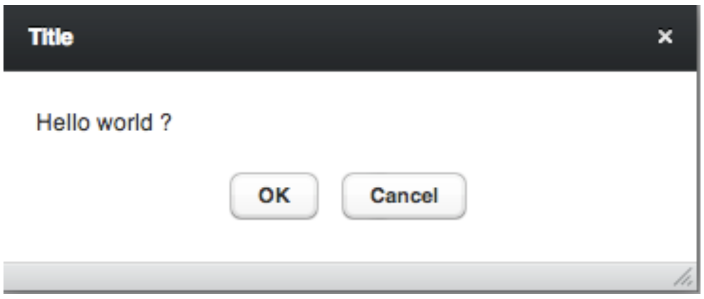


Simple message dialog shows a title, message and “OK” button to accept the message and close the dialog. The dialog also closes by the “Enter” or “ESC” keystroke.

```
OptionDialog dlg = new OptionDialog ( TPTApplication.getCurrentApplication () );

dlg.showMessageDialog ( "Title", "Hello world !", new
OptionDialog.OptionDialogResultListener () {
            public void dialogClosed ( OptionKind closeEvent )
            {
                  // dialog closed here;
            }
     });
```

Confirmation Dialog



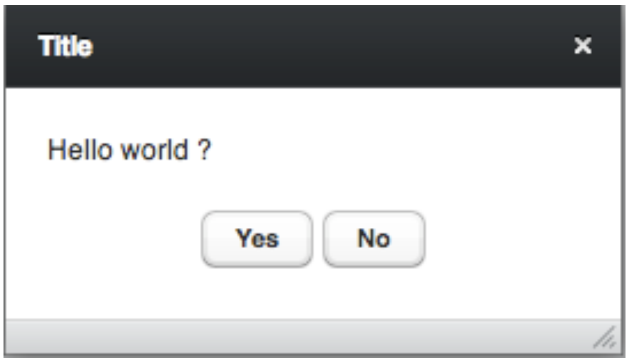
Confirmation dialog displays a title, text and “OK” and “Cancel” buttons to confirm or to abort the confirmation. “OK” button has also an equivalent of “Enter” keystroke and “Cancel” button - a “ESC” keystroke.

The dialog invocation code is as follows:

```
        dlg.showMessageDialog ( "Title", "Hello world ?", new
        OptionDialog.OptionDialogResultListener () {
                    public void dialogClosed ( OptionKind closeEvent )
                    {
                          // dialog closed here;
                    }
             });
```

closeEvent in dialogClosed(...) method will contain either OptionKind.OK or OptionKind.CANCEL values.

Question Dialog



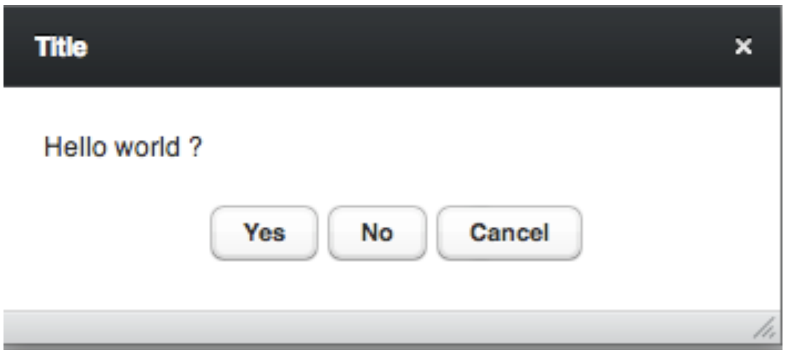
Question dialog displays title, question text and “Yes” and “No” buttons. Button “Yes” has also a “Enter” keystroke and button “No” - a “ESC” keystroke.

The dialog invocation code is as follows:

```
dlg.showMessageDialog ( "Title", "Hello world ?", new
OptionDialog.OptionDialogResultListener () {
            public void dialogClosed ( OptionKind closeEvent )
            {
                  // dialog closed here;
            }
      });
```

closeEvent in dialogClosed(...) method will contain either OptionKind.YES or OptionKind.NO values.

Cancellable Question Dialog



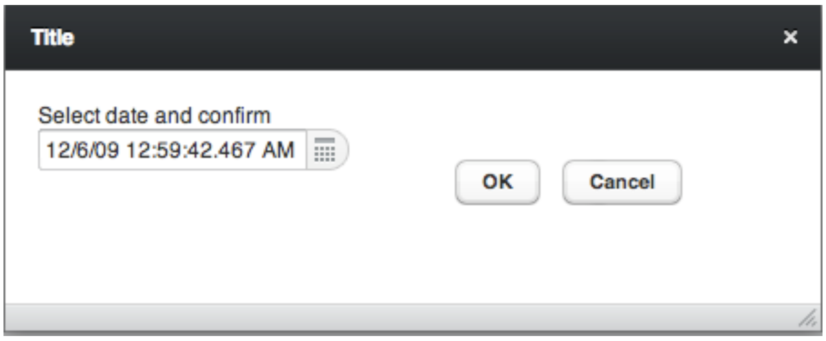
Cancellable question dialog acts like a normal question dialog but in addition to “Yes” and “No” buttons adds a “Cancel” button. In this dialog, an “Enter” keystroke is bound to the “Yes” button but “ESC” keystroke is bound to the “Cancel” button.

The dialog invocation code is as follows:

```
dlg.showYesNoCancelDialog ( "Title", "Hello world ?", new
OptionDialog.OptionDialogResultListener () {
            public void dialogClosed ( OptionKind closeEvent )
            {
                  // dialog closed here;
            }
      });
```

closeEvent in dialogClosed(...) method will contain one of the following values: OptionKind.YES, OptionKind.NO or OptionKind.CANCEL

Custom Message Dialog



Custom message dialog displays the title, custom Vaadin component as a message and a developer-defined set of buttons. The keystrokes in this dialog a bound as:

- “Enter” keystroke - “OK” or “YES” button in the listed priority
- “ESC” keystroke - “Cancel” or “NO” button in the listed priority

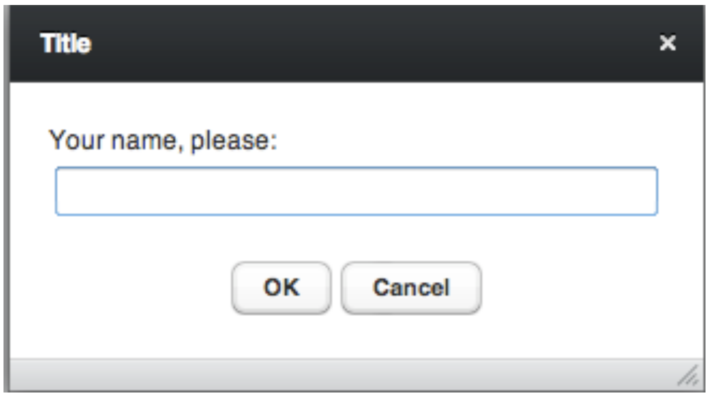
The dialog invocation code is as follows:

```
DateField df = new DateField ();
df.setCaption ( "Select date and confirm");

dlg.showCustomDialog ( "Title", df, new OptionDialog.OptionDialogResultListener
() {
            public void dialogClosed ( OptionKind closeEvent )
            {
                  // dialog closed here;
            }
      }, OptionKind.OK, OptionKind.CANCEL);
```

The closeEvent will contain an appropriate value of a pressed dialog button or a keystroke.

Input Dialogs



Input dialogs usually provide a method to input some value and option to either confirm or cancel it.

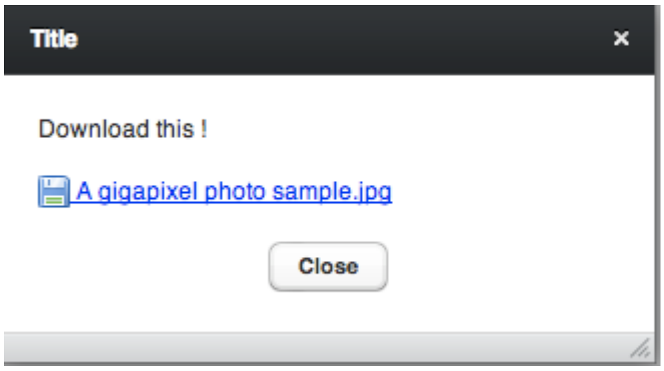
The dialog invocation code is as follows:

```
        InputDialog dlg = new InputDialog ( TPTApplication.getCurrentApplication () );

        dlg.showInputDialog ( "Title", "Your name, please:", "", new
        InputDialog.InputDialogResultListener () {
                    public void dialogClosed ( OptionKind closeEvent, String value )
                    {
                          // dialog result handling code here
                    }
             } );
```

The dialogClosed(...) method will contain a close event code as well as current valie, which was entered into an input field at the moment of dialog close.

Download Dialog



Download dialog provides a simple dialog with a title, message text, link to some internal or external resource to download and a "OK" button to close the dialog. This dialog is useful for providing to an end user a possibility to download something.

The dialog can be invoked as follows:

```
DownloadDialog dlg =
     new DownloadDialog ( TPTApplication.getCurrentApplication () );


dlg.showDownloadDialog (
     "Title",
     "Download this !",
     new ExternalResource ( "http://www.sharehost.com/xxx.jpg" ),
     "A gigapixel photo sample.jpg",
     new DownloadDialog.DownloadDialogResultListener ()
     {
            public void dialogClosed ()
            {
                   // dialog closed.
            }
     } );
```

Another variant of the showDownloadDialog(...) method allows to use a server File object as a download target. It automatically wraps the File object with a Vaadin's resource:

```
dlg.showDownloadDialog (
     "Title",
     "Download this !",
     new File ( “/var/tmp/sample.tiff” ),
     "A gigapixel photo sample.jpg",
     new DownloadDialog.DownloadDialogResultListener ()
     {
            public void dialogClosed ()
            {
                   // dialog closed.
```

```
                }
        } );
```

Common WIndows

Common windows in TPT extends the standard Vaadin's Window class and adds a couple of interesting features. Common windows lives in a package eu.livotov.tpt.gui.windows

TPTWindow

TPTWindow acts like a normal Vaadin's Window but also automatically catches the "Enter" and "ESC" keystrokes. In order to use them, just override the following methods:

- void enterKeyPressed () - called when "Enter" key is pressed.
- void escapeKeyPressed () - called when "ESC" key is pressed.

TPTWindow also provides the utility methods like showXXXMessage (...) - they delegate calls to a Vaadin's showNotification (...) method but with the difference that "\n" control characters in the title and message text are (optionally) replaced to html "<br/>" line breaks to make line terminated strings to look correctly on the screen.

TPTHtmlWindow

TPTHtmlWindow extends TPTWindow with all of its features but also adds the possibility to use a html resources as a background for the window. Basically, this is done simply by using a custom layout which is set as a main content for the window.

All rules, which applies to standard CustomLayout are correct for the TPTHtmlWindow. Layout file name (without an extension) or input stream can be provided to a TPTHtmlWindow constructor.

TPTHtmlWindow is i18n-aware, so it will search the correct layout file according to the current application language.

TPTHtmlWindow also allows to add any Vaadin widgets to itself by proxying addComponent(...) calls to a custom layout with which the window was initialized. Thus, TPTHtmlWindow is the same as if one would create a Window, create a CustomLayout and set it as a content to this Window and then add components to it.

Widgets

TPT contains the number of stand-alone widgets, which may be used anywhere in application. Those widgets are 100% server-side, e.g. uses the default client widgetset and does not require any GWT recompilation.

TPTMultiView

TPTMultiView is an iPhone OS like view controller, which manage the set of user interfaces, called “views”, displays one of them (current) at a time and manages the current view switching. Using this widget one can create a complex switchable interfaces.

Views can be switched either programmatically from within the application or from external events - an application address URL. This provides the transparent integration with the browser, enabling the bookmarking features and direct access of application components via the direct links.

Views can not only be switched, but also receive or send parameters to another views. Parameters may also be sent programmatically or embed into the direct URLs.

Each view is a standard Vaadin’s Component object, for instance, VerticalLayout, Panel or other layout or widget. View does not need to implement any special interface, however it can do this in order to receive notifications on its life cycle events.

Creating a view controller

Creating a view controller is simple as follows:

```
boolean isManageUrls = true;
TPTMultiView controller = new TPTMultiView ( isManageUrls );

myWindow.addComponent ( controller );
```

Field isManageUrls determines is view controller should install an URIFragmentUtility to manage the state of an application URL as well as intercept and trigger views from direct links to the application.

Technically, the TPTMultiView is a VerticalLayout with all its features. Once view controller is created, it can be added as any other Vaadin widget to the Window or other layout.

Adding views into the controller

View is a standard Vaadin’s Component object. So simply add them to a view controller instance, accomplishing the name of the view. Name must be unique for all views and it will be used as a reference to a view:

```
        controller.addView ( “main”, new MainDataPanel() );
        controller.addView ( “settings”, new SettingsPanel() );
```

You can also use view component class instead of the instance - this will delay the view instantiation until first invocation of this view:

```
        controller.addView ( “settings”, SttingsPane.class );
```

When you add a view to a controller, it will not become visible and therefore does not change the current view. The exception is a first view - when you add a view to an empty controller, this view will automatically become visible.

You may also remove or replace views in any time by calling the appropriate methods:

```
        controller.removeView ( “settings” );
        controller.replaceView ( “main” , new MainDataPanel() );
```

Switching between views

You may switch the current view programmatically by calling the switchView(...) method of a view controller. The method argument is a view name:

```
        controller.switchView ( “settings” );
```

If your view controller was created with enabled option for URL management, you can switch the view by accessing your application with the view name in the application URL:

```
        localhost:8080/myapp#settings
```

The application URL string in a browser will also reflect the current view when it is switched programmatically.

If invalid view name is passed via switchView(...) method or via an URL, view controller throws an IllegalArgumentException. To suppress this, you may define a failsafe view name, which will be displayed in such cases:

```
        controller.setFailsafeViewName ( “main” )’
```

Passing parameters to a views

When switching to a particular view, you can pass a parameter to it. For instance, if you have a view named “display”, which displays the documents from a database, you may be interested to have a direct links to your application, to open a particular document. So, you will need to pass a document ID to a view.

Parameters are added to the view name, separated by the "/" character from it. If such slash delimited view name is passed to a switchView (…) method or called in browser, text before slash is threaten as view name, and text after slash - as view parameter. This parameter can be the read by a view if it will implement an interface for listening of life cycle events (see below).

In out example about "display" view, the URL might look like as follows:

```
localhost:8080/myapp#display/1223
```

or you may write in your code:

```
controller.switchView ( "document/1223" );
```

Listening to view lifecycle events

A view controller emits the life cycle events when view is added, removed, opened and closed. In order to catch those events, a view must implement a TPTView interface. Once implemented, it's methods will be automatically called by a controller, so no registration necessary.

```
public class SettingsView extends VerticalLayout implements TPTMultiView.TPTView
{
     public void viewActivated ( String previousViewName, String parameters )
     {
     }

     public void viewDeactivated ( String newViewName )
     {
     }

     public void viewAttached ()
     {
     }

     public void viewRemoved ()
     {
     }
}
```

- viewActivated method is called when this view is displayed (e.g. controller has switched to this view). Controller also sends a name of a previous view and view parameters (if any).
- viewDeactivated is called when this view is hidden, e.g. controlled has switched to another view. A new active view name is passed as a parameter.
- viewAttached is called when a view component is added to the controller.
- viewRemoved is called when a view component is removed from the controller.

TPTLazyLoadingLayout - a layout to organize lenghty operation tasks

TPTLazyLoadingLayout can be used to display a UI components which initialization takes a long time. During the initialization process, a layout will display a temporary UI with a progress bar and optional message and when initialization process ends - an actual UI is shown.

As TPTLazyLoadingLayout runs the initialization task in a separate thread, the main UI and web browser window is not locked, stays responsive and even allows to navigate between other application parts while UI initialization process is in progress.

To use a lazy loading layout, simply write your actual UI component , implement a `LazyLoader` interface in it and move all lengthy initialization stuff into the `LazyLoader.lazyLoad(...)` method. Then create a `TPTLazyLoadingLayout`, passing your actual component into the constructor.

`TPTLazyLoadingLayout` will display a progress indicator and also invoke your implemented `lazyLoad` method in a separate thread and once it will finish - put the method result as an actual UI component instead of progress indicator.