Join the

# MARKETPLACE

Liferay's community awaits your next, big app.

## PARTICIPATE IN A GROWING PLATFORM

**350**
Marketplace Apps

**850**
Marketplace Developers

**370%**
Growth in # Apps
Since Launch 8/2012

**70%**
Growth in #Developers
Over the past year

## INCREASE YOUR VISIBILITY

**97%**
Inc. Avg.
Weekly Downloads

**84%**
Inc. Avg.
Weekly Views

## DISCOVER AND ADD ESSENTIAL TOOLS

Communication/Social

Reporting

Themes

Admin Tools

Auth/Integration

**EXPLORE AND PARTICIPATE IN THE LIFERAY MARKETPLACE TODAY!**

**Explore:** www.liferay.com/marketplace | **Participate:** www.liferay.com/developer/marketplace

LIFERAY

CONTENTS



# Mastering Portal UI Development With Vaadin and Liferay

By Sami Ekblad and James Falkner

The open source Liferay Portal has become a popular way of implementing enterprise websites. Providing an integrated platform for application development and deployment, Liferay has also become an environment for running business applications. For application development, Liferay Portal includes Vaadin as a pre-packaged framework for developing attractive, easy-to-use applications.

### ABOUT THIS REFCARD

This Refcard gives a quick overview of the user interface development with Vaadin on Liferay. It covers topics like portlet setup, configuration, inter-portlet communication (IPC), UI composition, and theming. To get a more general understanding of Liferay Portal and Vaadin framework, see Liferay Essentials and Vaadin: A Familiar Way to Build Web Apps With Java.

## STARTING THE DEVELOPMENT

### STRATEGIES FOR PORTAL USER INTERFACE

Portlets are small web applications written in Java. They run in a piece of a web user interface within a portal. Portal manages the lifecycle and aggregation of portlets to a single visible web page. Here are some things to consider when choosing the tools for portlet development:

**UI granularity** - How modular are your portlets? Are you building big application-like portlets, or small portlets that communicate with each other?

**Security** - Is the business logic running on the server side or on the browser

**User experience** - What kind of user experience is wanted in terms of graphics and transitions?

**Programming language** - What kind of development tools are available, and is a typed language preferred over scripting languages?

### AVAILABLE UI FRAMEWORKS

Liferay supports a number of server-side and client-side web frameworks for development of portlet user interfaces. Which you should use depends on your background, as well as the strategy you choose for your application.

| FRAMEWORK | DESCRIPTION |
|---|---|
| Alloy UI | Rich client-side JavaScript/ CSS/SASS/HTML Framework |
| AngularJS | Rich Client-side MVC Framework based on JavaScript. |
| JavaServer Faces (JSF) | Server-side UI component framework based on JSP and taglibs. |
| jQuery | Client-side JavaScript library easing DOM manipulation, AJAX and event handling. |
| ReactJS | Client-side JavaScript library for building rich, reactive UIs |
| Spring MVC | Action-oriented MVC framework using Java, XML, and JSP |
| Vaadin | A rich Java-only component framework based on AJAX/ GWT |

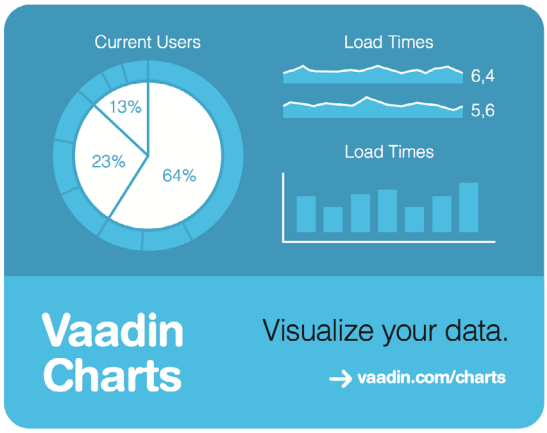
Different portlets can use different frameworks to implement the user interface.

## PORTLET DEVELOPMENT WITH VAADIN

Vaadin is a server- and component-oriented user interface framework for Java web applications. Vaadin applications can be hosted as standalone web applications as well as portlets in portals like Liferay. Vaadin is a good choice for building applications that use Liferay as a platform.

Portlets created with Vaadin are essentially Ajax web applications that can be considered single-page applications.

- **No page reloads** - Once running, the portlet page does not change when navigating inside a Vaadin application.
- **Rich user experience** - User interface relies on modern HTML5 and CSS3 web features like animations, transitions, and effects.



- **Desktop-like development and UX** - Applications are programmed and behave much like desktop applications, rather than web applications.
- **Event-based XHR communication** - User events are sent to server-side and UI changes are passed in response.
- **Java-only development** - Vaadin applications are meant to be built with Java.

## DEPLOYMENT STRATEGIES

Depending on your needs and requirements, there are two different ways to deploy a Vaadin application. The deployment setup also affects how your portlet is developed, so it is important to know the type of setup you are targeting.[h]

| FRAMEWORK | DESCRIPTION |
|---|---|
| Standalone [Preferred] | Vaadin applications and resources are bundled together, creating a completely self-contained application with its own configuration separate from global configuration. Recommended for most deployments. |
| Shared | Vaadin applications are deployed separately and share a common Vaadin core. This makes it easy to start developing and saves server resources, but prevents per-application configuration and requires all applications to use the same version of Vaadin. Recommended for projects with many small Vaadin applications. |

**HOT TIP**

When using the Shared strategy, all Vaadin applications deployed to Liferay will use the single embedded Vaadin version in Liferay. Before deploying Vaadin applications, you will need to Upgrade Vaadin to Vaadin 7. You can then use the Vaadin Control Panel to manage the global installation, including compiled widgetsets and global configuration.

You can access the Control Panel in Liferay after logging in as an administrator at *Manage > Control Panel > Portal > Vaadin*.

## TOOLS FOR VAADIN DEVELOPMENT

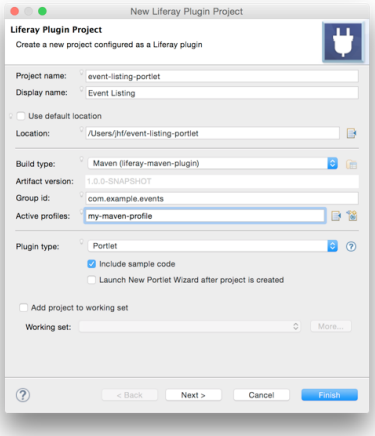
Since Liferay 6.x, there have been several tools to help you in developing portlets with Vaadin. These tools are meant to simplify the creation of portlets and help portal administrators maintain the system.

### LIFERAY IDE

Liferay IDE is an extension for the Eclipse IDE that adds support for the development of plug-in projects for the Liferay Portal platform. Since version 2.1, Liferay IDE has supported Vaadin 7 + Maven by offering wizards for creating portlet plugin projects.

### CREATING A PROJECT WITH LIFERAY IDE

Plugins for Liferay Portal must be created inside of a Liferay project. A Liferay project is essentially a root directory with a standardized structure containing the project's (and each of its plugins') necessary files. To create a new Liferay project within Liferay IDE, select *File > New > Liferay Plugin Project* and use the wizard to fill in project details, as shown below:




**HOT TIP**

The Vaadin Plugin for Eclipse can also be used with the Liferay IDE to give developers the ability to easily create Vaadin + Liferay projects and visually compose Vaadin components and portlets for use within Liferay.

## DEVELOPING WITH VAADIN AND LIFERAY

Vaadin applications deployed to Liferay can access the complete portal environment, including Liferay services, user context, and content that lives within Liferay. This section provides an overview of how Vaadin applications integrate with Liferay, along with best practices for deployment, initialization, and interaction with the Liferay environment.

### LIFERAY/VAADIN PROJECT FILES

Vaadin applications built for Liferay are created as Portlet projects, which have several configuration files, such as portlet.xml. The project structure generated by Liferay IDE follows a Maven-style layout. For detailed information on all of these files, see the Liferay Developer Guide. Specific files with Vaadin-related content include:

| PATH IN PROJECT | DESCRIPTION |
|---|---|
| `src/main/java/` | Java source code for Vaadin portlets |
| `src/main/webapp/ (or docroot/)` | The "root" of your Vaadin portlet application ("/" at runtime) |
| `src/main/webapp/VAADIN` | Vaadin-specific content, such as custom themes and compiled widgetsets |

| PATH IN PROJECT | DESCRIPTION |
|---|---|
| src/main/resources/ | Web resources for the application, including the custom.gwt.xml (which should inherit from com.vaadin.DefaultWidgetSet) |
| src/main/webapp/liferay-plugin-package.properties | Liferay deployment configuration, most notably portal-dependency-jars, which declare Liferay libraries required by the application. These are copied into your application's classpath when the application is deployed to Liferay. |
| src/main/webapp/portlet.xml | JSR-286 Portlet descriptor, including declaration of the Vaadin UI class as an <init-param>. This wires the Vaadin application to Liferay. |
| src/main/webapp/liferay-portlet.xml | src/main/webapp/liferay-portlet.xml |
| src/main/webapp/liferay-display.xml | Describes the category in which the application appears in Liferay's *Add > Applications* menu. |
| src/main/webapp/web.xml | Servlet configuration needed to serve Vaadin resources from portlet context in "Standalone" deployment setup. |
| pom.xml | Maven build script controlling building, and deploying, and specifying project dependencies like Vaadin add-ons. |

**HOT TIP**

If you are using the "Shared Deployment" strategy, where all of your Vaadin applications share a single Vaadin version deployed with Liferay, you must put the Vaadin libraries in your liferay-plugin-package.properties. For example:

```
portal-dependency-jars=\
vaadin.jar,\
confirmdialog-2.1.1.jar
```

Here confirmdialog-2.1.1.jar is an example of a shared Vaadin add-on used by the different portlets. Refer to the application server setup to see where these JAR dependencies should be installed to work at compile time and runtime.

### PORTLET DESCRIPTOR

To wire Vaadin to your portlet, configure portlet mapping in the portlet.xml. Here is an example of content of the file for Standalone Deployment setup:

```
<portlet>
        <portlet-name>My simple Vaadin portlet</portlet-name>
        <display-name>My Vaadin portlet</display-name>

        <portlet-class>com.vaadin.server.VaadinPortlet</
portlet-class>
        <init-param>
                <name>UI</name>
                <value>com.example.plugins.MyPortletUI</value>
        </init-param>
<init-param>
                <name>vaadin.resources.path</name>
                <value>PORTLET_CONTEXT</value>
        </init-param>
</portlet>
```

When using Liferay IDE, this file is automatically generated by the Vaadin-Liferay Maven archetype. The parameters are explained below:

| PARAMETER NAME | DESCRIPTION |
|---|---|
| UI | The class name of the application UI. This must extend **com.vaadin.ui.UI** |
| vaadin.resources.path | Specifies where Vaadin public resources are served from. Use PORTLET_CONTEXT for standalone deployments or use /html for "shared" deployments. |

### LIFERAY PORTLET DESCRIPTOR

Liferay also requires a liferay-portlet.xml descriptor file that defines Liferay-specific parameters. In particular, Vaadin portlets must be defined as "instanceable," but not as "ajaxable":

```
<liferay-portlet-app>
  <portlet>
    <portlet-name>Portlet Example portlet</portlet-name>
    <instanceable>true</instanceable>
    <ajaxable>false</ajaxable>
  </portlet>
</liferay-portlet-app>
```

This is because Vaadin portlets handle the Ajax requests internally without Liferay's Ajax mechanisms.

### LIFERAY PORTLET DISPLAY DESCRIPTOR

The **liferay-display.xml** file defines the portlet category under which portlets are located in the Add Application window in Liferay. Without this definition, portlets will be organized under the "Undefined" category.

The following puts the application in a new category called "Vaadin":

```
<display>
  <category name="Vaadin">
    <portlet id="MyVaadinExamplePortlet" />
  </category>
</display>
```

For more information on these and other optional descriptors, see Chapter 11.8 of the Book of Vaadin and the Liferay Developer Guide.

## COMPOSING THE USER INTERFACE WITH VAADIN

With Vaadin, the user interface is built from user interface components. They are server-side Java classes that implement a single UI control such as a button, select, or layout. With layout components, you can compose larger components that hierarchically build up the application UI.

### VAADIN APPLICATION

A Vaadin application is defined in a class that extends **com.vaadin.ui.UI**. This is the class that you should define as the UI initialization parameter in **portlet.xml** as described in earlier sections.

A new instance of this class is created when a new user comes to portal view where the portlet resides.

Example minimal working Vaadin application using the "Valo" theme:

```
@Title("My Vaadin Portlet UI")
@Theme("valo")
public class HelloPortlet extends com.vaadin.UI {

  @Override
  protected void init(VaadinRequest request) {

    // Create the content root layout for the UI
    VerticalLayout content = new VerticalLayout();
    setContent(content);


    // Display the greeting
    content.addComponent(new Label("Hello Portlet!"));
  }
}
```

## VAADIN UI COMPONENTS

Vaadin Framework includes nearly 100 stock components, and you can find hundreds of open-source add-on components in the Vaadin Add-on Directory.

You can find all the core components in the com.vaadin.ui.* Java package. Add-ons may use their own package naming, but it is typical that they start with org.vaadin.*.

**HOT TIP** You can test and try different Vaadin components online at http://vaadin.com/sampler. All the demos include source code and documentation.

## USER INTERFACE LAYOUT

Start by creating a main Window for your application and putting the initial content in there. The user interface structure is a hierarchy of nested layouts and components. Here is an example of a simple user interface hierarchy:

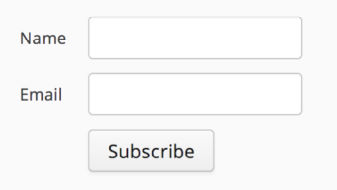
VerticalLayout (form)

- TextField (name)
- TextField (email)
- Button (subscribeBtn)

The above UI could be created in Java as follows:

```
setTitle("Subscribe Newsletter");
VerticalLayout form = new VerticalLayout();
form.setSpacing(true);
TextField name = new TextField("Name");
TextField email = new TextField("Email");
Button subscribeBtn = new Button("Subscribe");
form.addComponent(name);
form.addComponent(email);
form.addComponent(subscribeBtn);
```

The result will look like:




**HOT TIP** You should avoid creating deeply nested layout structures. In particular, older browsers can become slow. Instead, use the CustomLayout, GridLayout, or some lightweight layouts like the CSSLayout.

## USER INTERFACE EVENTS

Vaadin is an event-based framework. You can receive user-triggered events in your application by registering a listener for it. Here is an example for Button.ClickEvent:

```
subscribeBtn.addListener(new Button.ClickListener() {
  @Override
  public void buttonClick(ClickEvent event) {
  // …
  }
});
```

Event listeners are executed in the server side synchronously. You can fetch data and update the user interface by adding and removing components.

**HOT TIP** A good practice for event listeners is to only call your Java control code and let them do the UI updates. This is better object-oriented design, and it enhances readability of your Java code.

## THEMING VAADIN APPLICATIONS

Vaadin is designed to support the parallel work of application developers and graphic designers by strongly separating the graphical elements from the functionality.

All Vaadin applications have an associated theme. Themes are essentially a collection of SCSS files, images, fonts, and other assets that define the look and feel of Vaadin's user interface components. The following Vaadin themes are included in Liferay by default:

| THEME NAME | DESCRIPTION |
|---|---|
| valo | Comprehensive and highly customizable Sass-theme. This is the recommended starting point for building your own themes for Liferay 6.2 and above. |
| base | The base theme for creating your own customized theme. Handles most of the cross-browser issues. |
| liferay | A Liferay 6.0 look-a-like theme. Use this to create applications that match the Liferay 6 default styles. |
| reindeer | The default look and feel of Vaadin. It provides a minimalistic look for business applications. |
| runo | A more colorful and rounded theme for web applications. |

## STRUCTURE OF A VAADIN THEME

Vaadin themes are located in the themes folder of the portal. They are a collection of CSS and images that give the Vaadin components their look and feel. The theme folder must

contain a styles.css stylesheet, and custom layouts must be placed in the layouts/ sub-folder. Other contents may be named freely.

An example Vaadin theme structure extending "Valo":

| FILE | PURPOSE |
|---|---|
| mytheme.scss | Sass/CSS rules for your own theme. |
| addons.scss | SCSS/CSS for Vaadin add-ons. This file is automatically generated by the Vaadin widgetset compiler. |
| styles.css | Generated CSS file. This file is the result of Sass compilation of styles.scss, and you should not edit this file manually. |
| styles.scss | Main level theme file importing mytheme.scss, addon.scss, and other inherited themes. |
| layouts/ | Directory for CustomLayout definition files. |

To inherit a Vaadin Valo-theme, insert it in the main level styles.scss file:

```
@import "../valo/valo";
```

You should put your SCSS customization rules into the separate mytheme.scss file and import those rules into styles.scss. To activate the theme in your portlet, add the following to the init() method of your application:

```
public void init(VaadinRequest request) {
  setTheme("mytheme"); // ...
}
```

## SCSS CLASS NAMES IN VAADIN

To maximize the use of theme inheritance and to help customize components, the CSS class selectors in Vaadin are defined using the scheme .**v-***<component>*-*<item>*. All style names are lowercase.

As an example, the following SCSS rules extend the Valo theme and change the color of all captions and add borders to all TextFields in the mytheme.scss file:

```
.mytheme {

  // include valo theme rules
  @include valo;


  .v-texfield {
          border: 1px solid red;
  }

  .v-caption {
    color: red;
  }
}
```

The relevant CSS class names are:

| CSS CLASS NAME | DESCRIPTION |
|---|---|
| .v-app | The top-level DIV container for the whole application UI. |
| .v-<component> | A container for a specific component type. |
| .v-<component>-<element> | A CSS class for elements inside components. |

**HOT TIP** Always scope your selectors appropriately. This helps to avoid side effects and style leakage outside the Vaadin application.

```
.mytheme {
  /* … other SCSS rules … */
  .v-textfield {
    .v-caption {color: green; }
  }
}
```

## EXTENDING VALO-THEME

The Valo theme was introduced in Vaadin 7.3. Unlike other themes, Valo is meant to be configured using variables instead of using CSS rules. Variables define a feature of a theme, and they can affect many CSS rules at the same time. For example, changing background color luminance also configures the highlight color. Variable names start with **v-** and can be used like this:

```
$v-background-color: rgb(123,123,123);
```

Full reference documentation is available at https://vaadin.com/api/valo.

## ACCESSING LIFERAY APIs FROM VAADIN: BEST PRACTICES

Vaadin applications typically begin by constructing the UI using Vaadin APIs, and then allowing the application to respond to user events such as button clicks. In many cases, UI construction or event handling requires some data from the Liferay environment. Here are two example methods demonstrating how to access Liferay services from Vaadin UI code:

```
private String getPortletContextName(VaadinRequest request) {

  WrappedPortletSession wps = (WrappedPortletSession)
      request.getWrappedSession();

  PortletSession ps = wps.getPortletSession();
  PortletContext ctx = ps.getPortletContext();
  return ctx.getPortletContextName();
}

private Integer getPortalCountOfRegisteredUsers() {
  Integer result = null;

  try {
    result = UserLocalServiceUtil.getUsersCount();
  } catch (SystemException e) {
    log.error(e);
  }

  return result;
}
```

The above examples use several APIs explained below:

| | |
|---|---|
| `VaadinRequest` | This object is passed to the application's init() method, and contains several methods to get information about the incoming request and the portal context. |
| `VaadinRequest.getWrappedSession()` | This returns a generic wrapper around a more specific session object. When Vaadin applications are deployed to Liferay, this wraps the JSR Standard `javax.portlet.PortletSession` object and can be used to get information about Liferay, including the user making the request, and any shared session data (e.g. for IPC). |
| `WrappedPortletSession.getPortletSession()` | This unwraps the underlying JSR Standard javax.portlet.PortletSession object, giving access to session data using standard Portlet interfaces. |
| `PortletSession.getPortletContext()` | This object provides a handle to the portlet, as deployed to the server (see the JSR Portlet Specification for more information). |
| `PortletContext.` | The name of the context in which the portlet has been deployed on the application server (see the JSR Portlet specification for more detail). |

These can be used throughout a Vaadin application, including init(), and within listeners and other Vaadin-specific code.

## ACCESSING LIFERAY SERVICES

In the above code, there is a method call to `UserLocalServiceUtil.getUsersCount()` - this is a Liferay service that returns the number of users registered in Liferay. The Liferay Developer Guide contains descriptions of many other Liferay services, but here are a few to get you started (and which can be used throughout Vaadin applications).

| SERVICE NAME | DESCRIPTION |
|---|---|
| *User* | User account information, e.g. `UserLocalService.getUsersCount()` |
| *Team, Role, Permission, Company, Group, UserGroup* | Multi-tenant companies, sites and organization data, e.g. `GroupLocalServiceUtil.getGroupByName(companyId, "my-site");` |
| *BlogsEntry, Bookmarks, JournalArticle, MBMessage, WikiPage, DL** | Web content and user-generated content like blogs and wikis, e.g. |
| *Portal, Portlet, PortletPreferences* | Portal-wide and per-portlet configuration and preferences, e.g. `PortalUtil.getUser(request);` |
| *SocialActivity* | Social Activities Framework, e.g. |

**HOT TIP** There are two ways to call a service. For example, UserLocalServiceUtil.getUsersCount() vs. UserServiceUtil.getUsersCount(). The local service skips permission checking, and assumes the caller has the required permissions. You can either do the permission checking yourself, or call the non-local variation, and let Liferay check permissions for you.

## INTER-PORTLET COMMUNICATION (IPC)

Liferay offers different IPC mechanisms to allow portlets to communicate with each other. The following table summarizes the different IPC methods in Liferay:

| TYPE | DESCRIPTION |
|---|---|
| JSR-286 Portlet Events | Standard portlet communication mechanism. Requires page reload. |
| JavaScript/AJAX | Traditional client/server communication, using client-side JavaScript, calling other portlets running in the same page using Liferay's client-side JavaScript API. |
| Vaadin Add-on for Liferay IPC | Mechanism for sending and receiving events between Vaadin and non-Vaadin portlets. |
| Ajax Push (Reverse Ajax) or WebSockets | Typically used for server > client notifications (for example, in-browser chat). |

## IPC IN VAADIN PORTLETS

Vaadin applications deployed to Liferay can easily communicate with each other through the above IPC mechanisms. The easiest approach is to use the Vaadin IPC Add-on for Liferay. To use this add-on in your project, simply declare the dependency within your pom.xml:

```
<dependency>
  <groupId>org.vaadin.addons</groupId>
  <artifactId>vaadin-ipc-for-liferay</artifactId>
  <version>2.0.0</version>
</dependency>
```

Or, if you are using Ivy:

```
<dependency org="org.vaadin.addons"
            name="vaadin-ipc-for-liferay"
            rev="2.0.0" />
```

Once declared, your Vaadin applications can communicate using the LiferayIPC APIs, as shown here:

## SENDING A MESSAGE TO ANOTHER VAADIN APPLICATION

```
/* class member declaration */
private LiferayIPC ipc;
@Override
protected void init(VaadinRequest request) {
  /* UI Initialization */
  ipc =  new LiferayIPC();
  ipc.extend(this);
}
some_button.addClickListener(new ClickListener() {
  @Override
  public void buttonClick(ClickEvent event) {
    /* Send message on button click */
    ipc.sendEvent("MyIPCMessageName", "The Payload");
  }
});
```

Receiving the event in another Vaadin application:

```
ipc.addLiferayIPCEventListener("MyIPCMessageName",
  new LiferayIPCEventListener() {

  @Override
  public void eventReceived(LiferayIPCEvent event) {
    Notification.show("Got the payload: " + event.getData());
  }
});
```

## RECEIVING AN EVENT FROM A VAADIN APPLICATION/PORTLET USING JAVASCRIPT

```
<script type="text/javascript">
  Liferay.on("uniqueEventId",
    function(event) {
        alert(event);
    }
  );
</script>
```

## ANOTHER PORTLET USING JAVASCRIPT

```
<script>
  Liferay.fire("uniqueEventId", "someData");
</script>
```

This method is not suitable for sending a lot of data, but rather notifying the portlet that something has updated. The actual data should be shared using the portal session, the database, files, or some external storage. For example, to use the portal session to store and retrieve large objects:

```
WrappedPortletSession wrappedPortletSession =
(WrappedPortletSession) request.getWrappedSession();

PortletSession portletSession = wrappedPortletSession.
getPortletSession();

// save data to session
portletSession.setAttribute("data-key", data, PortletSession.
APPLICATION_SCOPE);

// retrieve data from session
```

SomeType data = (SomeType)portletSession.
getAttribute("data-key", PortletSession.APPLICATION_
SCOPE);

## ABOUT THE AUTHORS

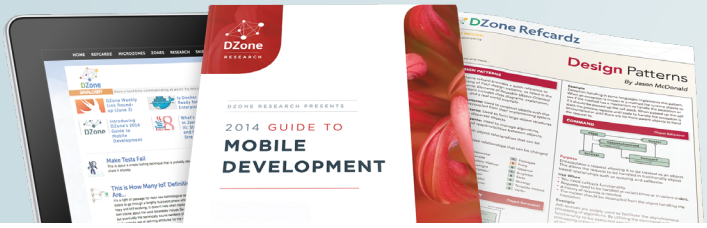
Sami Ekblad is one of the original authors of the Vaadin framework. Working in web application development since 1998, he now works as Partner Manager at Vaadin Ltd to help professional web developers to get most out of the Vaadin framework and tools. He holds B.Sc. degree in Computer Science from the University of Turku.

James Falkner is an open-source evangelist, community manager, and software developer working at Liferay, producers of the world's leading open source enterprise portal. In addition to Liferay, James has been active ina number of other open source products and projects, includingthe GlassFish Enterprise portfolio, Community/Social Equity, OpenSolaris, OASIS standards, and more. contributor and speaker at industry events such as JavaOne, JAX, and others.

## RECOMMENDED BOOK


Liferay in Action is a comprehensive and authoritative guide to building portals on the Liferay 6 platform. Fully supported and authorized by Liferay, this book guides you smoothly from your first exposure to Liferay through the crucial day-to-day tasks of building and maintaining an enterprise portal that works well within your existing IT infrastructure.








DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.



DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

**Refcardz Feedback Welcome**
refcardz@dzone.com

**Sponsorship Opportunities**
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

Version 1.0 $7.95