DUMPSBOSS.

# Portal Developer

## Liferay LRP-614

### Version Demo

**Total Demo Questions: 17**

**Total Premium Questions: 172**

**Buy Premium PDF**

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

QUESTION NO: 1

The settings in portal-developer.properties:

**A.** Enable faster deployment of plugins

**B.** Disable minification of CSS and JavaScript

**C.** Assist deployment from a development to a production environment

**D.** Precompile Velocity and FreeMarker templates

**ANSWER: B**

**Explanation:**

**Disable minification of CSS and JavaScript** is correct. The `portal-developer.properties` file is a development-oriented property set that configures Liferay for easier debugging and iteration rather than optimized production delivery. In particular, its development defaults turn off resource minification, allowing developers to inspect the original CSS and JavaScript source more easily in browser developer tools. Keeping these resources unminified also makes source-level troubleshooting and changes to front-end assets more transparent while developing themes, fragments, and client-side functionality. The file is intended to be included or used as an override for a development environment; production installations should use appropriately optimized settings instead. Liferay's source distribution contains the developer property defaults, including the resource-processing settings, in portal-developer.properties. For the general mechanism Liferay uses to define and override portal properties across environments, see the Liferay configuration documentation.

**QUESTION NO: 2**

Portlets can use interportletcommunication to: (Please select all correct answers.)

**A.** Pass any serializable object as an event payload

**B.** Trigger multiple events

**C.** Invoke events directly from the render phase

**D.** Enforce the processing order of events

**ANSWER: A B**

**Explanation:**

Inter-portlet communication through the Portlet event mechanism lets one portlet publish information that other portlets have declared an interest in receiving. The event payload may be an object that can be serialized, allowing structured application data—not merely a text value—to be transferred with the event. In practice, developers define the event and its payload type in `portlet.xml`, publish the event from an action or event-processing request, and consume it in a portlet's event-processing method. This makes event payloads suitable for communicating state or a domain object between cooperating portlets.

A portlet may also trigger multiple events in the same request by calling `setEvent` more than once on the action response. Each published event is then available for delivery to portlets that have registered the corresponding event QName. This supports workflows where one user action needs to notify separate consumers or communicate several distinct pieces of event data. Liferay's documentation describes portlet events as an inter-portlet communication mechanism, while the Portlet API documents the action-response methods used to publish events. See Liferay Learn: Using Inter-Portlet Communication and Portlet API ActionResponse reference.

**QUESTION NO: 3**

The business logic for UserLocalServiceUtil is found in:

**A.** Use rLocalServiceUtil

**B.** UserLocalServiceImpl

**C.** Use rLocalService

**D.** UserServiceImpl

**E.** UserServiceUtil

**ANSWER: B**

**Explanation:**

**UserLocalServiceImpl** is correct because it is the service implementation class that contains the local User service's business logic. In Liferay's service architecture, `UserLocalServiceUtil` is a static utility facade generated for convenient access to the local service. Its methods delegate to the underlying `UserLocalService` implementation rather than serving as the location for custom business rules. The implementation class, `UserLocalServiceImpl`, is where the service methods are implemented and where logic such as validation, orchestration of related persistence operations, permission-aware service behavior where applicable, and model lifecycle handling belongs. This separation lets callers use a stable utility API while keeping implementation details in the service layer. Liferay's API reference identifies `UserLocalServiceUtil` as the utility entry point for the local service, and the corresponding implementation class is part of the portal service implementation package. See the UserLocalServiceUtil API reference and Liferay's Service Builder documentation for the generated service facade and implementation pattern.

## QUESTION NO: 4

Creating a Struts action hook requires the following elements in liferay-hook.xml: (Please select all correct answers.)

**A.**

**B.**

**C.**

**D.**

**E.**

**ANSWER: B C E**

**Explanation:**

A Struts action hook is declared with a `<struts-action>` element in `liferay-hook.xml`. This enclosing element defines one action override or extension and contains the configuration needed for Liferay to register the custom action. Within it, `<struts-action-path>` identifies the path of the existing portal Struts action that the hook targets, such as a login-related action path. The path is the key used to associate the hook configuration with the portal action being customized. The `<struts-action-impl>` element supplies the fully qualified class name of the custom Java action implementation that Liferay invokes for that path. Together, these elements form the required declaration: a container for the action mapping, the portal action path to intercept or replace, and the implementing class. The Liferay hook deployment descriptor DTD defines `<struts-action>` with `<struts-action-path>` and `<struts-action-impl>` as its child elements. See the official liferay-hook.xml DTD and Liferay's source definition for the descriptor structure.

## QUESTION NO: 5

Liferay's core local services: (Please select all correct answers.).

**A.** Contain the business logic of the service

**B.** Enforce permission checking

**C.** Are required if using remote services

**D.** Communicate to the database through the persistence layer

**ANSWER: A C D**

**Explanation:**

Liferay's core local services contain the application's business logic and provide the normal server-side API used to create, read, update, and delete entities. Service Builder implementations commonly coordinate validation, workflow-related operations, indexing, asset handling, and calls to other services in this local-service layer. Local services communicate with the database through the generated persistence layer rather than embedding direct database-access logic in callers. This separation keeps persistence operations encapsulated and provides a consistent transactional service boundary.

Local services are also required when using remote services because a remote service is the permission-aware entry point layered over the corresponding local-service functionality. Remote service methods perform the security-related work appropriate to externally accessible calls and then delegate the underlying business operation to local services. Consequently, the local service remains the foundation for the business behavior and persistence interaction even when an operation is invoked through a remote API. Liferay's Service Builder documentation describes the local service as the layer for business logic and persistence access, while remote services expose that functionality with permission checking. See Liferay Service Builder and Generating Service Builder Code.

**QUESTION NO: 6**

The Plugins SDK uses:

**A.** portal-service.jar

**B.** JAR files from the user's home directory

**C.** portal-impl.jar

**D.** portal-plugin.jar

**ANSWER: A**

**Explanation:**

**portal-service.jar** is correct. In the classic Liferay Plugins SDK, this JAR supplied the public portal service API that plugin projects compile against. It contains the service-layer interfaces, model APIs, and related public contracts needed by traditional plugin types, including portlets, hooks, layouts, themes, and Ext plugins. Using this API artifact lets a plugin call supported portal services while maintaining the intended separation between a plugin's code and Liferay's internal implementation details.

This distinction was important to the SDK's design: plugins were built with the public service API available on their compile classpath, and the portal runtime supplied the corresponding implementation when the plugin was deployed. Developers could therefore use Liferay's service interfaces and generated model/service classes without bundling portal implementation code inside the plugin. Liferay's legacy documentation describes the Plugins SDK as the development environment for these traditional plugin projects and identifies its dependency and build setup. For background on the legacy SDK and plugin development model, see Liferay Help Center: Plugins SDK and Liferay Learn: Development.

**QUESTION NO: 7**

MVCPortlet directs page flow using:

**A.** The controller() method

**B.** A render parameter called "mvcPath"

**C.** A direct link to the JSP with a relative URL

**D.** A public render parameter called "path"

**ANSWER: B**

**Explanation:**

A render parameter called `mvcPath` is the standard mechanism that `MVCPortlet` uses to select the JSP rendered for a request. During an action or render request, application code can set this parameter—for example, with `actionResponse.setRenderParameter("mvcPath", "/view_details.jsp")`. When the subsequent render phase occurs, `MVCPortlet` reads the parameter and dispatches to the specified view path. This provides a predictable way to move between views while following the portlet lifecycle: actions process user input first, then the render phase displays the resulting view. If no `mvcPath` is supplied, the portlet normally falls back to its configured default view template, commonly `/view.jsp`. Liferay's MVC framework also supports render-command-based navigation through `MVCCommandNames.MVC_RENDER_COMMAND_NAME`, but the direct JSP view-selection convention implemented by `MVCPortlet` is specifically the `mvcPath` render parameter. See Liferay's documentation on MVC Render Commands and the MVCPortlet API reference.

## QUESTION NO: 8

An administrator would like to add a new travel preferences section in the Miscellaneous section of the user form. The travel preferences are stored as custom fields. The recommended way to implement the solution using a hook plugin is to: (Please select all correct answers.)

**A.** Modify html/portlet/users_admin/edit_user.jsp to add the travel preferences section to theMiscellaneous section

**B.** Set the portal property "users.form.my.account.miscellaneous" to include "travel-preferences" and add the corresponding JSP

**C.** Create a Struts action hook to persist the Expando value to the database

**D.** Add the custom fields to the relevant JSP using

**ANSWER: B D**

**Explanation:**

Using the `users.form.my.account.miscellaneous` portal property to register `travel-preferences` is the supported hook-based extension pattern for legacy Liferay user-form navigation. The property contributes the new section identifier to the Miscellaneous area of the My Account user form, and the hook supplies the JSP associated with that section. This keeps the customization modular: it extends the configured form navigator rather than replacing the core user-edit JSP. The property can be delivered through the hook's portal-properties configuration, along with the JSP placed in the hook using Liferay's expected users-admin JSP location and naming conventions.

Adding the custom fields with the `<liferay-ui:custom-attribute />` tag is also appropriate because travel preferences are Expando custom attributes. This tag renders the configured custom-field UI for the relevant model instance, allowing Liferay's standard custom-attribute infrastructure to load and save the values. As a result, the implementation uses the platform's existing user-form extension point and custom-field persistence behavior rather than duplicating that behavior in application code. Liferay's portal-property reference documents user-form configuration properties, and its JSP customization guidance describes using hooks to contribute customized JSP resources: Portal Properties Reference and Customizing JSPs with Hooks.

## QUESTION NO: 9

Public render parameters are of the type:

**A.** List

**B.** RenderParameter

**C.** String

**D.** Object

**E.** RenderRequest

**ANSWER: C**

**Explanation:**

**String** is correct because a public render parameter is a named request parameter whose individual value is represented as text. Public render parameters let cooperating portlets on the same page share rendering state, such as a selected category, keyword, or identifier, through the portlet request and render URL. A portlet reads an individual parameter with `PortletRequest.getParameter(String)`, which returns a `String`. When a parameter has multiple submitted values, the API exposes those values through `getParameterValues(String)` as a `String[]`; that does not change the type of each public render-parameter value.

In Liferay, developers declare support for a public render parameter by its name and then access the shared value from the render request using the standard Portlet API. The parameter is therefore passed and interpreted as a string value; application code can subsequently convert that text to a numeric value, date, or domain-specific object when needed. This behavior follows the Portlet request parameter contract documented by the Jakarta Portlet API, where `getParameter` returns a string and parameter maps contain string-value arrays. See the PortletRequest API documentation and Liferay's MVC render command documentation for the render-request context in which these parameters are consumed.

## QUESTION NO: 10

The standard JSR-286 portlet modes are: (Please select all correct answers.)

**A.** Help

**B.** Configuration

**C.** Print

**D.** Edit

**E.** View

**F.** About

**ANSWER: A D E**

**Explanation:**

The standard JSR-286 portlet modes are **Help**, **Edit**, and **View**. These modes define the principal interaction states that a JSR-286-compliant portlet must support. **View** is the normal mode for presenting the portlet's primary content and functionality to an end user. **Edit** provides a standard place for a portlet to expose user-specific preferences or settings that affect its behavior or presentation. **Help** provides contextual documentation or guidance for using the portlet. The Portlet API represents these standard modes through the predefined `PortletMode` constants `VIEW`, `EDIT`, and `HELP`. A portal can expose additional vendor-specific or optional modes, but those are not part of the core standard set named by the JSR-286 Portlet API. See the PortletMode API documentation and the Portlet 2.0 (JSR-286) specification for the standard mode definitions.

## QUESTION NO: 11

Portlet events are distributed to portlets on different pages by setting the property:

**A.** portlet.event.distribution=all-pages

**B.** event.distribution =all-pages

**C.** portlet.event.distribution=layout-set

**D.** No specific property setting is required

**ANSWER: C**

**Explanation:**

`portlet.event.distribution=layout-set` is the correct setting for extending portlet event delivery beyond the page where the event was fired. In Liferay's portlet-event implementation, a layout is a page and a layout set is the collection of pages belonging to the same site context, such as that site's public pages or private pages. Configuring the event distribution scope as `layout-set` instructs the portal to distribute a published portlet event to compatible portlets located on pages within that layout set. This enables inter-portlet communication across different pages while retaining the site and page-set boundary that Liferay uses for layouts. The participating portlets must still declare matching event definitions and processing support in their portlet configuration; the property controls the portal-level distribution scope rather than replacing standard event declarations. Configure this property through the appropriate portal property override for the target Liferay installation, rather than modifying the shipped defaults directly. Liferay's portal properties reference documents the available portal-level configuration settings, and its portlet communication documentation explains the event-based communication model used by portlets. See the Liferay Portal Properties Reference and Liferay Portlet Communication documentation.

**QUESTION NO: 12**

The "companyId" is a(n):

**A.** Portal instance

**B.** Organization

**C.** Site

**D.** Team

**ANSWER: A**

**Explanation:**

Portal instance is correct. In Liferay's data model and APIs, a `companyId` identifies the portal instance (also commonly called a virtual instance or tenant) to which an entity belongs. Liferay historically uses the term "company" internally for this top-level tenant boundary, so many persisted models, services, permission checks, and context objects include a `companyId`. This identifier establishes the instance-level scope for data and configuration, allowing one Liferay installation to host multiple isolated portal instances with separate users, sites, roles, content, and settings.

The current request context normally supplies this value through the theme display or service context, rather than requiring developers to derive it manually. For example, code executing for a request can obtain the active portal instance's identifier from `ThemeDisplay.getCompanyId()`. Liferay's virtual-instance documentation describes virtual instances as separate portal instances hosted by the same Liferay installation, while the `Company` API represents that top-level instance entity and exposes its identifier. See Liferay's Virtual Instances documentation and the Company API reference.

**QUESTION NO: 13**

The recommended way to turn off portlet borders in a custom theme is to:

**A.** Modifyportlet.vm to hide theportletborders

**B.** Set the theme setting "portlet-setup-show-borders" to "false"

**C.** Create a JSP hook that overrides port!et_wrapper.jsp to hide theportletborders

**D.** Set the portal property "portlet.setup.show.borders" tofalse"

**ANSWER: B**

**Explanation:**

Set the theme setting `"portlet-setup-show-borders"` to `"false"` is the appropriate theme-level mechanism because portlet border presentation is part of a page's look and feel. A custom theme can declare this setting in its `liferay-look-and-feel.xml` configuration, allowing the theme to control the visual chrome associated with portlet setup without modifying portal rendering files. This keeps the customization scoped to the theme that needs it, so other themes and sites can retain their own presentation settings. It also makes the behavior portable: when the theme is deployed or moved between environments, the setting travels with the theme's configuration rather than depending on a server-wide change. Liferay themes are intended to contain both styling assets and theme-specific display configuration, making this setting the maintainable choice for a borderless portlet presentation. Liferay's theme-development guidance explains how themes package and apply site appearance behavior, while the Portal source repository provides the versioned implementation context for legacy theme settings. See Liferay Theme documentation and the Liferay Portal 6.2 source repository.

## QUESTION NO: 14

The tag defines the variables: (Please select all correct answers.)

**A.** "portletPreferences"

**B.** "renderRequest"

**C.** "remoteUser"

**D.** "portletSession"

## ANSWER: A B D

**Explanation:**

The `<portlet:defineObjects />` JSP tag makes standard Portlet API objects available as named scripting variables in a portlet JSP. `"renderRequest"` is the current render-phase request and provides access to request attributes, parameters, locale information, user information exposed by the Portlet API, and related request context. `"portletPreferences"` supplies the portlet's preference set, allowing the JSP to read configured preference values through the standard `PortletPreferences` API. `"portletSession"` provides the current portlet session, which can hold attributes for the portlet scope or application scope as appropriate. The tag also exposes related standard objects, including the portlet configuration and render response, so JSPs can interact with the portlet runtime without repeatedly obtaining these objects manually. These variables are defined by the standard Portlet JSP tag library and are intended for rendering views in a traditional JSP-based portlet. Liferay supports this standard Portlet API model in portlet development. See Liferay's Using JSPs documentation and the Portlet API documentation for PortletRequest.

## QUESTION NO: 15

The recommended way to implement a service wrapper hook that customizes the authenticateByScreenName() method of the User service using a class called com.sample.MyUserLocalServiceImpl is to: (Please select all correct answers.)

**A.** Add the following to liferay-hook.xml:com.liferay.portal.service.UserLocalServicecom.sample.MyUserLocalServiceImpl

**B.** Copy all of the methods from UserLocalServiceImpl to MyUserLocalServiceImpl

**C.** Override the authenticateByScreenName() method in MyUserLocalServiceImpl

**D.** Add the following to liferay-plugin-package.properties: required-deployment-contexts=UserLocalService

**E.** Implement custom logic and call super.authenticateByScreenName() if applicable

## ANSWER: A C E

**Explanation:**

A service wrapper hook registers a replacement implementation for a portal service while retaining access to the original service through the wrapper superclass. The `<service>` declaration in `liferay-hook.xml`, with `com.liferay.portal.service.UserLocalService` as the service type and the custom implementation as the service implementation, is what instructs the portal to apply the wrapper. The custom class should extend the appropriate `UserLocalServiceWrapper` class and override `authenticateByScreenName()`, which is the extension point where the authentication-specific behavior is customized. Within that override, custom logic can run before or after delegation to `super.authenticateByScreenName()`. Delegating when appropriate retains Liferay's established authentication processing and allows the customization to supplement rather than unnecessarily replace standard behavior. This wrapper pattern provides a focused way to alter service behavior without copying the entire service implementation. Liferay documents service wrappers as the supported mechanism for overriding portal service behavior; see the Liferay service-behavior override documentation and the UserLocalServiceWrapper API reference.

## QUESTION NO: 16

As a best practice, a portlet plugin imports classes from: (Please select all correct answers.)

**A.** portal-impl.jar

**B.** portal-service.jar

**C.** portlet.jar

**D.** ext-impl.jar

## ANSWER: B C

**Explanation:**

For the legacy Liferay Plugin SDK architecture reflected by this question, **portal-service.jar** and **portlet.jar** are the appropriate dependencies for a portlet plugin. **portal-service.jar** exposes Liferay's supported service-layer API, including generated local and remote service interfaces, model interfaces, and related utility access classes. A portlet that needs to read or update portal data should depend on these public service contracts, so its code uses the intended application boundary rather than internal implementation details. This supports maintainability and makes the plugin less coupled to the portal's internals.

**portlet.jar** supplies the standard Java Portlet API used to implement portlets. It provides the portlet programming contracts, such as portlet request and response types, lifecycle methods, preferences, and related API classes. Importing this API allows the plugin to compile against the portable portlet specification and interact correctly with the portlet container. The same design principle continues in modern Liferay development: custom applications should use documented public APIs and service interfaces. See Liferay's development documentation and the public portal API reference for the supported API-oriented development model.

## QUESTION NO: 17

In portal.properties:

login,events.post=\

com.liferay.portal.events.ChannelLoginPostActionA,\

com.liferay.portal.events.DefaultI_andingPageAction,\

com.liferay.portal.events.LoginPostAction

A hook plugin can insert a new class between DefaultLandingPageAction and LoginPostAction.

**A.** True

**B.** False

**ANSWER: A**

**Explanation:**

True is correct. In the legacy Hooks framework, a hook plugin can supply a `portal.properties` file to override or extend supported portal configuration properties. The `login.events.post` property is an ordered, comma-delimited list of post-login action classes. Liferay executes these actions in the order in which their class names appear in the effective property value. Therefore, the hook can provide the event-action sequence with its custom action placed immediately after `DefaultLandingPageAction` and before `LoginPostAction`. This preserves the intended behavior of the existing login processing while establishing the required execution point for the custom code. In practice, the hook's property value must contain the relevant existing action class names as well as the custom class, in the desired order; it is not enough merely to declare a class without considering the complete effective list. Login event actions are a standard extension mechanism in the older plugin architecture and are loaded by class name, so the custom action must also be packaged with the hook and implement the appropriate lifecycle action contract. Liferay's portal-property reference documents event-action properties, while the legacy Hooks documentation describes using hooks to customize portal behavior: Portal Properties Reference and Using Hooks.