Widget Templates in Liferay 7.4

Application Display Templates, FreeMarker, Widget Presentation, Configuration, Reuse, Security, and Best Practices

Liferay DXP Technical Notes

September 28, 2026

Abstract

This document explains Widget Templates in Liferay DXP 7.4, historically associated with Application Display Templates (ADTs). Widget Templates allow administrators and developers to customize how supported Liferay widgets render their data without modifying the widget's underlying Java implementation.

The document covers template architecture, FreeMarker syntax, template creation and selection, context variables, collections, conditional rendering, localization, URLs, custom portlet support, security, performance, comparison with fragments and themes, troubleshooting, interview questions, and multiple-choice questions.

Contents

1 Introduction 4

2 Application Display Templates 4

3 Why Use Widget Templates? 5

4 Simple Architecture 5

5 Presentation Separation 6

6 Example 6

7 Where Widget Templates are Managed 7

8 Creating a Widget Template 7

9 Template Language 7

10 Basic FreeMarker Example 8

11 Template Context 8

12 Using Variables 9

13 Default Values 9

14 Conditional Rendering 9

15 Looping Through Entries 10

16 Card-Based Example 10

17 Using Clay/Bootstrap Classes 10

18 Example: Grid Layout 11

19 Widget Configuration 11

20 Reusable Presentation 12

21 Widget Templates and Asset Publisher 12

22 Widget Templates and Asset Lists 13

23 Localization 13

24 Escaping Output 13

25 FreeMarker Restrictions 13

26 Bad Template Design 14

27 Avoid Heavy Processing 14

28 N+1 Problem 15

29 Template Error Handling 15

30 Inspecting Available Variables 16

31 themeDisplay 16

32 URLs in Templates 16

33 Avoid Hard-Coded Site URLs 17

34 Custom Widget Support 17

35 Custom Portlet Architecture 17

36 Example Quantum Assessment Template 18

37 Widget Templates vs Fragments 19

38 Widget Template vs Fragment Example 19

39 Widget Templates vs Themes 19

40 Widget Template vs CSS Client Extension 20

41 Widget Template vs Custom Element 20

42 Decision Guide 20

43 Permissions 21

44 Security Considerations 21

45 Do Not Use Templates to Bypass Permissions 22

46 Performance Best Practices 22

47 Separation of Responsibilities 22

48 Template Reusability 23

49 Development Lifecycle 23

50 Testing Checklist 24

51 Responsive Design 24

52 Accessibility 25

53 Troubleshooting: Template Not Visible 25

54 Troubleshooting: Invalid Reference 25

55 Troubleshooting: Template Works for Some Entries 26

56 Troubleshooting: Slow Rendering 26

57 Troubleshooting: Layout Looks Different 26

58 Practical Example Architecture 27

59 Interview Questions and Answers 28

60 Multiple Choice Questions 28

61 Quick Revision Table 32

62 Complete Architecture 32

63 Key Takeaway 33

64 Version Considerations 34

65 Official Reference 34

1 Introduction

A Liferay widget generally contains two broad concerns:

1. Business/data logic
2. Presentation logic

For example, an Asset Publisher may retrieve a collection of assets and render them using its default presentation.

Conceptually:

```
Widget
  |
  +--> Retrieve Data
  |
  +--> Default Rendering
```

Sometimes the data retrieval is correct, but the presentation must change.

For example, instead of:

```
Title
Description

Title
Description

Title
Description
```

we may want:

```
+------------+  +------------+  +------------+
|   Image    |  |   Image    |  |   Image    |
|   Title    |  |   Title    |  |   Title    |
|   Button   |  |   Button   |  |   Button   |
+------------+  +------------+  +------------+
```

A Widget Template provides this presentation customization.

Core Idea

A Widget Template changes how a supported widget displays its data without requiring the widget's underlying business logic to be rewritten.

2 Application Display Templates

Widget Templates are closely related to what Liferay traditionally called:

Application Display Templates (ADTs)

The fundamental idea is:

$$\text{Widget Data} + \text{Template} = \text{Custom Presentation}$$

Therefore, the same widget data can have multiple visual representations.

3 Why Use Widget Templates?

Widget Templates are useful when:

- Default widget rendering is not suitable
- A list should become cards
- Additional HTML structure is needed
- Business data should remain unchanged
- The same display should be reusable
- A developer wants to avoid modifying Liferay core JSPs

The design principle is:

Separate Data Retrieval from Presentation

4 Simple Architecture

Without a custom template:

```
Database
   |
   v
Liferay Service
   |
   v
Widget
   |
   v
Default Display
```

With a Widget Template:

```
Database
   |
   v
Liferay Service
   |
   v
Widget
   |
   v
Widget Template
   |
   v
Custom HTML
```

The Widget Template operates mainly at the presentation layer.

5 Presentation Separation

Conceptually:

$$D = \text{Data}$$

$$T = \text{Template}$$

$$R = \text{Rendered Output}$$

Then:

$$\boxed{R = T(D)}$$

The same data:

$$D$$

can therefore be rendered by:

$$T_1(D) = \text{List}$$

$$T_2(D) = \text{Cards}$$

$$T_3(D) = \text{Grid}$$

without changing the underlying data.

6 Example

Suppose an Asset Publisher returns three news articles.

The default presentation may be:

```
Quantum Computing Workshop

New Quantum Research Published

Photonics Laboratory Opens
```

A Widget Template could instead produce:

```
+-----------------------+
| Image                 |
| Quantum Computing     |
| Workshop              |
| [Read More]           |
+-----------------------+

+-----------------------+
| Image                 |
| New Quantum Research  |
| Published             |
| [Read More]           |
+-----------------------+
```

The content has not changed.
Only its presentation has changed.

7 Where Widget Templates are Managed

Depending on the Liferay 7.4 release and site configuration, Widget Templates can be managed through the site administration interfaces associated with design/templates.

The typical workflow is:

```
Site Administration
       |
       v
Design / Templates
       |
       v
Widget Templates
       |
       v
Create Template
```

Version Awareness

Liferay 7.4 has evolved through many GA and quarterly releases. The exact menu labels and navigation path can differ slightly between releases.

8 Creating a Widget Template

The general process is:

1. Open the appropriate Site Administration area.
2. Navigate to Widget Templates.
3. Click Add.
4. Select the widget/application for which the template is being created.
5. Enter a template name.
6. Write the template logic.
7. Save the template.
8. Add/configure the target widget on a page.
9. Select the new Widget Template from the widget's display configuration where supported.

9 Template Language

FreeMarker is the primary templating language used for many modern Liferay templates.

FreeMarker uses syntax such as:

```
${variable}
```

for values.
Conditions use:

```
<#if condition>
    ...
</#if>
```

Loops use:

```
<#list entries as entry>
    ...
</#list>
```

10 Basic FreeMarker Example

Consider:

```
<h2>My Widget Template</h2>

<#if entries?has_content>

    <ul>

        <#list entries as entry>

            <li>
                ${entry}
            </li>

        </#list>

    </ul>

</#if>
```

The exact objects available in the template depend on the widget.

11 Template Context

A Widget Template executes with a template context.

Conceptually:

$$Context = \{variables, objects, services, helpers\}$$

The exact variables depend on the application.

For example, a collection-oriented widget might expose something conceptually similar to:

```
entries
```

The template then iterates through those entries.

Do Not Assume Context Variables

Different widgets expose different template context objects. Do not assume that a variable available in one Widget Template exists in another. Use Liferay's template editor/context information and the documentation for the target widget.

12 Using Variables

Basic FreeMarker output:

```
${title}
```

Another example:

```
<h2>${title}</h2>
```

A variable should be checked where it may be absent.
For example:

```
<#if title??>
    <h2>${title}</h2>
</#if>
```

The operator:

```
??
```

checks whether a value exists.

13 Default Values

FreeMarker can provide a fallback value.
Conceptually:

```
${title!"Untitled"}
```

If:

$$title = \varnothing$$

then:

```
Untitled
```

is displayed.
This improves template robustness.

14 Conditional Rendering

Example:

```
<#if entries?has_content>

    <p>Entries are available.</p>

<#else>

    <p>No entries found.</p>

</#if>
```

This prevents blank or confusing output.

15 Looping Through Entries

A common Widget Template operation is:

```
<#list entries as entry>

    <div class="item">

        ${entry}

    </div>

</#list>
```

Conceptually:

$$Entries = \{E_1, E_2, \ldots, E_n\}$$

and the template performs:

$$\forall E_i \in Entries, \quad Render(E_i)$$

16 Card-Based Example

Suppose a widget exposes a list of items.

A card-style template might conceptually be:

```
<div class="row">

<#list entries as entry>

    <div class="col-md-4">

        <div class="card">

            <div class="card-body">

                <h3>
                    ${entry}
                </h3>

            </div>

        </div>

    </div>

</#list>

</div>
```

This demonstrates the primary purpose:

$$\text{Same Entries} \rightarrow \text{Different HTML Structure}$$

17 Using Clay/Bootstrap Classes

Liferay's frontend uses Clay and Bootstrap-oriented styling conventions.

A template can use available classes such as:

```
<div class="container">

    <div class="row">

        <div class="col-md-6">

            Content

        </div>

    </div>

</div>
```

This helps custom output visually integrate with the surrounding Liferay UI.

18 Example: Grid Layout

```
<div class="row">

<#if entries?has_content>

    <#list entries as entry>

        <div class="col-lg-4 col-md-6 mb-4">

            <div class="card h-100">

                <div class="card-body">

                    <h4 class="card-title">
                        ${entry}
                    </h4>

                </div>

            </div>

        </div>

    </#list>

</#if>

</div>
```

19 Widget Configuration

After creating a template, the relevant widget normally needs to be configured to use it.

Conceptually:

```
Page
 |
 v
```

```
Widget
 |
 v
Configuration
 |
 v
Display / Template
 |
 v
Custom Widget Template
```

This allows the same widget to use different templates in different contexts.

20 Reusable Presentation

Suppose a template called:

```
Corporate Card Layout
```

is created.
It can potentially be reused by multiple instances of the supported widget.
Thus:

One Template → Multiple Widget Instances

This improves consistency.

21 Widget Templates and Asset Publisher

Asset Publisher is a classic use case for display templates.
Conceptually:

```
Asset Publisher
      |
      +--> Web Content
      +--> Documents
      +--> Blogs
      +--> Other Assets
      |
      v
Widget Template
      |
      v
Custom Display
```

For example, assets could be rendered as:

- News cards
- Image gallery
- Featured content
- Compact list
- Timeline

22 Widget Templates and Asset Lists

A collection-oriented widget can provide data while the template controls how that data appears.

This separation can be expressed as:

$$\boxed{\text{Selection Logic} \neq \text{Presentation Logic}}$$

The widget selects entries.

The template renders entries.

23 Localization

A production template should consider multilingual portals.

Instead of hard-coding:

```
Read More
```

a localized value should be used where the template context and supported APIs provide localization helpers.

Conceptually:

$$\text{Language Key} \rightarrow \text{Current Locale} \rightarrow \text{Localized Text}$$

For example:

```
English -> Read More
French  -> Lire la suite
Hindi   -> Aur padhen
```

24 Escaping Output

Output security is important whenever dynamic values are rendered.

Conceptually:

$$UserInput \rightarrow Escape \rightarrow HTML$$

rather than:

$$UserInput \rightarrow RawHTML$$

because unsafe output can contribute to cross-site scripting (XSS) risks.

Security

Never assume that arbitrary dynamic content is safe to inject as raw HTML. Understand the escaping behavior of the template engine and the origin of the data being rendered.

25 FreeMarker Restrictions

A template environment should not be viewed as unrestricted Java execution.

For security reasons, Liferay controls what can be accessed from templates.

A template should primarily perform:

- Presentation
- Formatting

- Simple conditions
- Iteration

It should not become a replacement for the application's service layer.

26 Bad Template Design

A poor design is:

```
Widget Template
     |
     +--> Complex Database Logic
     |
     +--> Remote API Calls
     |
     +--> Business Transactions
     |
     +--> Complex Permission Logic
```

A better design is:

```
Service / Widget
      |
      v
Prepared Data
      |
      v
Widget Template
      |
      v
Presentation
```

Thus:

$$\boxed{\text{Template} \approx \text{Presentation Logic}}$$

27 Avoid Heavy Processing

Suppose a template renders:

$$N = 100$$

entries.
If each iteration performs an expensive operation of cost:

$$C$$

the total cost becomes approximately:

$$T = N \times C$$

If:

$$C = 50ms$$

then:

$$T = 100 \times 50ms = 5000ms$$

or approximately:

$$5 \text{ seconds}$$

before considering other processing.
Therefore, expensive service calls inside loops should be avoided.

28 N+1 Problem

Consider:

```
<#list entries as entry>

    getAdditionalData(entry)

</#list>
```

If each item causes a database or service call:

$$1 \text{ initial request} + N \text{ additional requests}$$

then:

$$1 + N$$

operations are performed.
For:

$$N = 100$$

this becomes:

$$101$$

operations.
Prepare data efficiently before rendering wherever possible.

29 Template Error Handling

A template can fail if it assumes a value always exists.
For example:

```
${entry.title}
```

may fail or behave unexpectedly if the expected value is unavailable.
Safer logic can include:

```
<#if entry.title??>
    ${entry.title}
</#if>
```

or an appropriate fallback.

30 Inspecting Available Variables

When developing a template, determine which variables the selected widget exposes.

Conceptually:

```
Widget
 |
 +--> Template Context
       |
       +--> entries
       +--> request
       +--> themeDisplay
       +--> helper objects
       +--> application-specific values
```

The exact set varies by widget and Liferay version.

Use the editor's available variable/reference information and official documentation instead of guessing object methods.

31 themeDisplay

In supported template contexts, `themeDisplay` provides information related to the current portal request.

Conceptually it can provide access to contextual information such as:

- Current user
- Company
- Site/group
- Locale
- Portal URL context

Example concept:

```
<#if themeDisplay??>

    Current Site Context Available

</#if>
```

The actual accessible methods depend on the template environment.

32 URLs in Templates

A template often needs links.

For example:

```
<a href="${viewURL}">
    Read More
</a>
```

The URL should normally come from the widget/template context rather than being hard-coded.

Bad:

```
<a href="/web/guest/news?id=123">
```

Better:

```
<a href="${viewURL}">
```

This improves portability.

33 Avoid Hard-Coded Site URLs

Hard-coded URLs can break when:

- Site friendly URL changes
- Virtual host changes
- Environment changes
- Portal context changes
- Application moves between development and production

Therefore:

Context-Aware URL > Hard-Coded URL

34 Custom Widget Support

Developers can also design custom Liferay applications to support Application Display Templates / Widget Templates where the relevant Liferay extension APIs are used.

Conceptually:

```
Custom Portlet
     |
     +--> Provides Data
     |
     +--> Registers Template Support
     |
     v
Widget Template
     |
     v
Custom Presentation
```

This provides administrators with presentation flexibility without requiring a new application deployment for every visual variation.

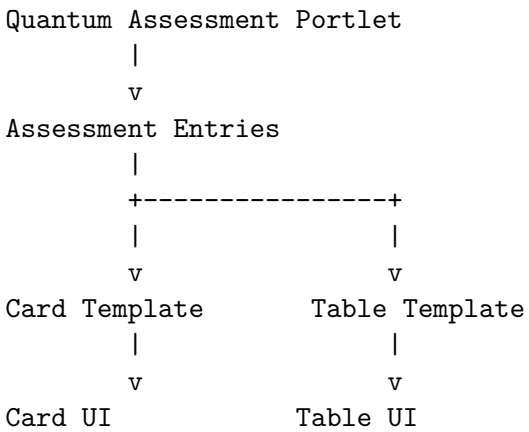
35 Custom Portlet Architecture

Suppose a custom application displays quantum assessments.

Without templates:

```
Quantum Assessment Portlet
        |
        v
Fixed JSP
```

With template support:



The same data can therefore support multiple presentations.

36 Example Quantum Assessment Template

Conceptually:

```
<div class="row">

<#if entries?has_content>

    <#list entries as assessment>

        <div class="col-md-4 mb-3">

            <div class="card h-100">

                <div class="card-body">

                    <h3>
                        ${assessment}
                    </h3>

                    <p>
                        Quantum assessment
                    </p>

                    <button
                        class="btn btn-primary">

                        Start Assessment

                    </button>

                </div>

            </div>

        </div>

    </#list>

</#if>
```

```
</div>
```

The exact assessment properties depend on the custom widget's template context.

37 Widget Templates vs Fragments

Widget Templates and Fragments solve different problems.

Characteristic	Widget Template	Fragment
Primary purpose	Render widget data	Build page components
Data source	Widget/application	Fragment context/configuration
Typical language	FreeMarker	HTML/CSS/JavaScript/FreeMarker features
Relationship to widget	Strong	Usually independent
Use case	Alternative widget display	Page composition
Example	Asset cards	Hero banner

38 Widget Template vs Fragment Example

Requirement:

> "Display Asset Publisher news entries as cards."

Use:

Widget Template

Requirement:

> "Create a reusable hero section containing heading, image and button."

Use:

Fragment

39 Widget Templates vs Themes

A Theme affects broad site presentation.

A Widget Template affects a specific supported widget's rendering.

Conceptually:

$$\text{Theme} \rightarrow \text{Site-Level Presentation}$$

while:

$$\text{Widget Template} \rightarrow \text{Widget-Level Presentation}$$

40 Widget Template vs CSS Client Extension

CSS Client Extension:

Changes Styling

Widget Template:

Changes Markup/Rendering

For example, if the HTML is already correct but the color must change:

CSS

If:

```
<ul>
  <li>...</li>
</ul>
```

must become:

```
<div class="card">
...
</div>
```

then:

Widget Template

is more appropriate.

41 Widget Template vs Custom Element

A Widget Template renders data supplied by a Liferay widget.

A Custom Element Client Extension is an independently developed frontend application.

	Widget Template	**Custom Element**
Data	Widget-provided	Application-controlled
Technology	FreeMarker	React/Vue/Angular/JS
Complexity	Lower	Higher
Use case	Custom rendering	Full frontend application
Runtime	Liferay template rendering	Browser/external frontend

42 Decision Guide

A useful decision model is:

```
Need UI customization?
        |
        +--> Existing Widget Data?
        |       |
        |       +--> Change Rendering?
        |               |
        |               +--> Widget Template
```

```
|
+--> Reusable Page Component?
|       |
|       +--> Fragment
|
+--> Full React/Vue Application?
|       |
|       +--> Custom Element CX
|
+--> Styling Only?
        |
        +--> CSS / Theme CSS CX
```

43 Permissions

Widget Templates can have permissions governing who can:

- View
- Update
- Delete
- Manage

The exact actions depend on the Liferay version and resource.
Templates containing important presentation logic should not be editable by unauthorized users.

44 Security Considerations

Template authors should consider:

- Cross-site scripting
- Dynamic output escaping
- Unsafe HTML
- Excessive service access
- Sensitive data exposure
- Permissions
- User-generated content

A useful rule is:

Display Only What the User Is Authorized to See

45 Do Not Use Templates to Bypass Permissions

A Widget Template should not be treated as a mechanism for bypassing Liferay permissions.

Correct architecture:

```
User
 |
 v
Permission Check
 |
 v
Widget Data
 |
 v
Template
 |
 v
Display
```

not:

```
Template
 |
 v
Retrieve Unauthorized Data
```

46 Performance Best Practices

A good Widget Template should:

1. Keep logic simple.
2. Avoid expensive operations inside loops.
3. Avoid unnecessary service calls.
4. Check for missing values.
5. Avoid excessive nested loops.
6. Render only necessary data.
7. Use appropriate pagination at the widget/data layer.
8. Avoid large amounts of inline JavaScript.

47 Separation of Responsibilities

A well-designed architecture is:

```
Service Layer
     |
     | Retrieves Data
     v
Widget
```

```
     |
     | Prepares Context
     v
Widget Template
     |
     | Renders
     v
HTML
```

Therefore:

$$\boxed{Business\ Logic \rightarrow Service}$$

$$\boxed{Presentation\ Logic \rightarrow Template}$$

48 Template Reusability

Suppose three pages use the same visual design:

```
Page A -> Asset Publisher
Page B -> Asset Publisher
Page C -> Asset Publisher
```

Instead of implementing the card layout three times:

$$T_A = T_B = T_C$$

use one shared template:

$$\boxed{T_{CorporateCard}}$$

This improves maintainability.

49 Development Lifecycle

A practical development process is:

```
Requirement
    |
    v
Choose Widget
    |
    v
Inspect Template Context
    |
    v
Create Widget Template
    |
    v
Write FreeMarker
    |
    v
Test with Real Data
```

```
    |
    v
Check Permissions
    |
    v
Test Empty/Missing Values
    |
    v
Performance Test
    |
    v
Publish / Use
```

50 Testing Checklist

Test:

- Zero entries
- One entry
- Many entries
- Missing optional fields
- Long titles
- Special characters
- Multiple languages
- Guest user
- Authenticated user
- Mobile layout
- Tablet layout
- Desktop layout

51 Responsive Design

Templates should consider responsive layouts.

For example:

```
<div class="
    col-12
    col-md-6
    col-lg-4
">
    ...
</div>
```

Conceptually:

$$Mobile = 1\ Card/Row$$

$$Tablet = 2\ Cards/Row$$

$$Desktop = 3\ Cards/Row$$

52 Accessibility

Custom templates should preserve accessibility.

Consider:

- Semantic HTML
- Heading hierarchy
- Alternative image text
- Keyboard navigation
- Accessible links
- Button labels
- Color contrast

For example:

```
<img
    src="${imageURL}"
    alt="${imageAlt}"
/>
```

is preferable to an image with no meaningful alternative text.

53 Troubleshooting: Template Not Visible

Check:

- Was the correct widget/application selected?
- Was the template saved successfully?
- Does the widget support Widget Templates?
- Is the template available in the current site context?
- Does the user have permission?

54 Troubleshooting: Invalid Reference

An error such as a missing FreeMarker variable may indicate that the template expects a variable not supplied by the widget.

For example:

```
${articleTitle}
```

when:

$$articleTitle \notin Context$$

Use the actual variables available to the template.

55 Troubleshooting: Template Works for Some Entries

If some entries render while others fail, check for optional fields.

Bad:

```
${entry.image.url}
```

when some entries have no image.

Safer conceptually:

```
<#if entry.image??>
    ...
</#if>
```

56 Troubleshooting: Slow Rendering

Check for:

- Service calls inside loops
- Deep nested loops
- Too many entries
- Large images
- Excessive HTML
- Expensive template operations

If:

$$RenderTime \propto N \times C$$

reduce either:

$$N$$

or:

$$C$$

57 Troubleshooting: Layout Looks Different

Check:

- Theme CSS
- Clay/Bootstrap classes
- Custom CSS Client Extensions
- Browser developer tools
- CSS specificity
- Responsive breakpoints

Remember:

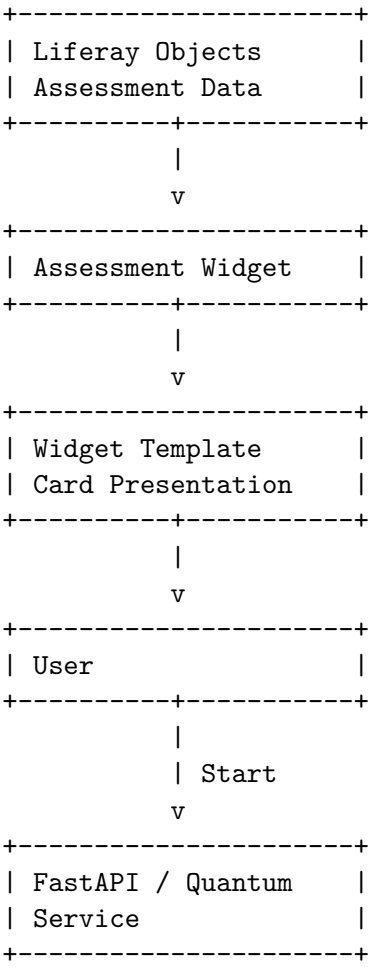
$$\text{Template HTML} + \text{Site CSS} = \text{Final Appearance}$$

58 Practical Example Architecture

Suppose a portal has:

- Liferay Objects storing Assessments
- A widget retrieving Assessments
- A Widget Template rendering cards
- FastAPI performing quantum calculations

Architecture:



The responsibilities remain clean:

$$Objects = \text{Data}$$

$$Widget = \text{Application Logic}$$

$$Template = \text{Presentation}$$

$$FastAPI = \text{External Computation}$$

59 Interview Questions and Answers

What is a Widget Template in Liferay?

A Widget Template provides an alternative presentation for a supported Liferay widget without requiring changes to the widget's underlying business logic.

What was the traditional name for Widget Templates?

They are historically associated with Application Display Templates, commonly abbreviated as ADTs.

Which template language is commonly used?

FreeMarker is commonly used for Liferay Widget Templates.

What does a Widget Template change?

It primarily changes the presentation/rendering of the data supplied by a widget.

Can one template be reused?

Yes. A Widget Template can be reused by supported widget instances where the template is available and applicable.

Should complex business logic be placed in FreeMarker?

Generally no. Business logic should remain in the application's service/widget layer while the template focuses on presentation.

What is the difference between a Widget Template and a Fragment?

A Widget Template changes how an existing widget renders its data, whereas a Fragment is primarily a reusable page-building component.

What is the difference between a Widget Template and a Custom Element?

A Widget Template renders data supplied by an existing widget. A Custom Element Client Extension is a more independent frontend application that can be built with technologies such as React.

60 Multiple Choice Questions

Question 1

What is the main purpose of a Widget Template?

A. Change database schema

B. Customize widget presentation

C. Configure Elasticsearch

D. Create users

Answer: B

Question 2

Widget Templates were traditionally associated with which term?

A. ADT

B. OSGi

C. JDBC

D. JNDI

Answer: A

Question 3

Which template language is commonly used for Widget Templates?

A. SQL

B. FreeMarker

C. Java bytecode

D. YAML only

Answer: B

Question 4

Which FreeMarker construct performs iteration?

A. `#list`

B. `SELECT`

C. `forEach()` only

D. `GROUP BY`

Answer: A

Question 5

What should normally remain outside a Widget Template?

A. Presentation

B. HTML markup

C. Complex business logic

D. Conditional display

Answer: C

Question 6

Which is a classic application of Widget Templates?

A. Asset Publisher custom display

B. Database clustering

C. JVM heap tuning

D. DNS configuration

Answer: A

Question 7

Which is more appropriate for a reusable page hero component?

A. Fragment

B. Widget Template

C. Database trigger

D. Search index

Answer: A

Question 8

Which is more appropriate for rendering Asset Publisher entries as cards?

A. Widget Template

B. Database View

C. Cluster Link

D. OAuth Provider

Answer: A

Question 9

Why should service calls inside template loops be minimized?

A. They can create performance problems

B. FreeMarker cannot display HTML

C. They disable CSS

D. They delete entries

Answer: A

Question 10

Which expression conceptually checks whether a FreeMarker variable exists?

A. `??`

B. `DROP`

C. `JOIN`

D. `DELETE`

Answer: A

Question 11

Which Client Extension is generally more suitable for a full React application?

A. Custom Element

B. Widget Template

C. Theme Favicon

D. Batch

Answer: A

Question 12

What does the Widget Template primarily receive?

A. Widget/application context and data

B. Database passwords

C. JVM bytecode

D. Search cluster credentials

Answer: A

Question 13

What should be done with optional values?

A. Assume they always exist

B. Check them or provide fallbacks

C. Delete the widget

D. Disable permissions

Answer: B

Question 14

What is the main security concern when rendering untrusted dynamic content?

A. XSS

B. Database replication

C. JVM garbage collection

D. DNS caching

Answer: A

Question 15

Which statement best describes good template design?

A. Put all application logic in FreeMarker

B. Keep presentation in the template and business logic in the application/service layer

C. Access the database directly from every loop

D. Hard-code all URLs

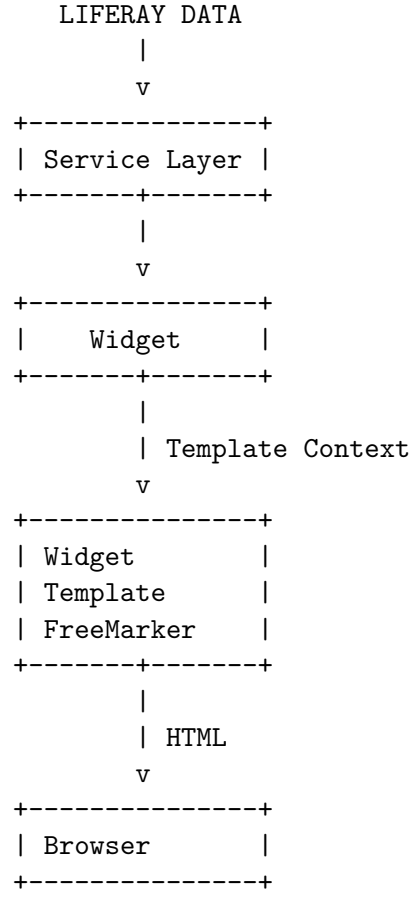
Answer: B

61 Quick Revision Table

Concept	Meaning
Widget Template	Alternative presentation for a supported widget
ADT	Application Display Template
FreeMarker	Common Liferay template language
Template Context	Variables and objects available to a template
`#list`	FreeMarker iteration
`#if`	Conditional rendering
`??`	Checks whether a value exists
Default Value	Fallback when a value is absent
Asset Publisher	Classic Widget Template use case
Fragment	Reusable page-building component
Custom Element	Independent frontend application integration
Theme	Site-wide presentation mechanism
CSS Client Extension	External/custom styling mechanism
Localization	Render text according to current locale
Escaping	Protect output from unsafe dynamic content
N+1 Problem	Repeated expensive calls while rendering a collection

62 Complete Architecture

A practical Liferay presentation architecture can be visualized as:



The division of responsibility is:

$$\boxed{Data \rightarrow Service}$$

$$\boxed{Application\ Behavior \rightarrow Widget}$$

$$\boxed{Presentation \rightarrow Widget\ Template}$$

$$\boxed{Final\ Rendering \rightarrow Browser}$$

63 Key Takeaway

The simplest way to remember Widget Templates is:

$$\boxed{\text{Same Widget} + \text{Same Data} + \text{Different Template} = \text{Different Presentation}}$$

For example:

$$AssetPublisher + T_1 = List$$

$$AssetPublisher + T_2 = Cards$$

$$AssetPublisher + T_3 = Grid$$

without changing the underlying Asset Publisher business logic.

A good Liferay architecture therefore follows:

$$\boxed{\text{Business Logic} \neq \text{Presentation Logic}}$$

with:

$$\boxed{\text{Business Logic} \rightarrow \text{Portlet / Service Layer}}$$

and:

$$\boxed{\text{Presentation Logic} \rightarrow \text{Widget Template}}$$

Use a Widget Template when an existing widget already supplies the correct data and behavior but its **presentation needs to change**.

64 Version Considerations

Liferay 7.4 spans many GA and quarterly releases.

Before implementing a production Widget Template, verify:

- Exact Liferay DXP/Portal release
- Whether the target widget supports Widget Templates
- Available template context variables
- Supported FreeMarker capabilities
- Security restrictions
- Available frontend/Clay classes
- Site permissions

The template context supplied by the target widget should be treated as the authoritative source for the variables and objects that the template can use.

65 Official Reference

For exact configuration and release-specific details, consult Liferay's official documentation for creating Widget Templates.