Liferay Client Extensions Deck

Must-Know Concepts, Explained

Coverage	Core Concepts, Extension Types Overview, Custom Elements, Theming & Styling, Configuration as Code, Packaging & Deployment, Integration Patterns, Best Practices & Migration
Format	Question → Concise answer → YAML / example where useful
Level	Liferay Frontend/Integration Developer to Solution Architect rounds

---

Reflects Liferay DXP 7.4+ / current Liferay DXP Cloud (quarterly release) Client Extensions. Liferay adds new extension types and capabilities every quarter – always confirm feature availability against the release notes for your version.

Generated September 19, 2026

1 Core Concepts 2

2 Extension Types Overview 4

3 Custom Element Extensions 6

4 Theming & Styling 8

5 Configuration as Code 10

6 Packaging & Deployment 12

7 Integration Patterns 14

8 Best Practices & Migration 16

Core Concepts

Q1.1

What are Liferay client extensions, and what fundamental problem do they solve compared to the traditional OSGi module/plugin approach to customizing Liferay?

Client extensions are a mechanism for extending Liferay's look, feel, and behavior – adding custom widgets, styling, scripts, or integrations – without deploying an OSGi module into Liferay's own runtime at all. Traditional plugin development (themes, portlets, and other OSGi modules) runs code *inside* Liferay's own Java/OSGi container, meaning a plugin's code shares the same runtime as Liferay itself, requires a redeploy (and in some cases a restart) for even a minor change, and carries real risk of breaking during a Liferay upgrade if it depends on internal APIs that change between versions. Client extensions instead run **outside** Liferay entirely – as external, independently-hosted or independently-packaged assets that Liferay references and loads at runtime – fundamentally decoupling extension code from Liferay's own internal implementation details, and this separation is precisely what protects both the core platform and the extension itself from upgrade-related breaking changes.

Q1.2

Why does deploying client extensions "outside of Liferay" matter specifically for compatibility across Liferay's different deployment models (Self-Hosted, PaaS, and SaaS)?

Liferay SaaS (DXP Cloud) customers don't have direct filesystem or server access to modify the running Liferay instance the way Self-Hosted or PaaS customers might – historically, this made SaaS deployments far more restrictive about what kind of customization was even possible at all, since traditional OSGi module deployment generally assumes some form of direct server/filesystem access. Because client extensions are packaged and deployed as external archives referenced by Liferay, rather than requiring direct modification of Liferay's own running server environment, the exact same client extension package works identically across **all** deployment types – Self-Hosted, PaaS, and SaaS alike – with only the specific upload/deployment mechanism differing (e.g. dropping a file into a folder for Self-Hosted/PaaS, versus using a command-line tool to upload to SaaS) rather than the extension's own code or packaging needing to differ by deployment type.

Q1.3

What is the standard naming convention for a client extension project directory, and what do the "owner" and "project" parts of that name represent?

A client extension's directory name follows the pattern `owner-project-type[-n]`, with each part separated by a hyphen. The **owner** identifies who created the extension (commonly an organization or team name), the **project** identifies which broader project or initiative the extension belongs to, and the final part identifies the extension's specific **type** (`custom-element`, `global-css`, `theme-css`, and so on, covered in Section 2) – with an optional trailing number distinguishing multiple extensions of the same type within the same project. Because hyphens are used specifically to separate these parts, the owner and project names themselves must not contain hyphens – a small but important naming constraint to keep in mind when scaffolding a new client extension project.

Q1.4

What is a Liferay Workspace, and how does it organize multiple client extension projects together for local development and deployment?

A Liferay Workspace is the standard project structure Liferay's developer tooling (Gradle-based, typically scaffolded via Liferay's workspace generator) provides for organizing all of a project's Liferay-related code – modules, themes, and client extensions – in one coherent, buildable repository. Within a workspace, client ex-

tensions typically live under a dedicated `client-extensions` directory, each in its own subdirectory following the naming convention above, and the workspace's Gradle tooling provides commands to build and deploy either *all* client extensions at once (`./gradlew deploy`) or a *specific* individual extension (`./gradlew :client-extensions:my-extension-name:deploy`) – giving developers a consistent, unified local development and deployment workflow regardless of how many separate client extensions a given project ultimately includes.

Q1.5

What is the difference between a client extension and a remote app, and how does the ”remote application” deployment pattern change what's actually included in the deployed archive?

In the standard, most common client extension pattern, the deployable archive contains both the extension's metadata (the `client-extension.yaml` file, Section 6) *and* the actual application code itself (JavaScript bundles, CSS files) – Liferay serves these bundled assets directly. In the **remote application** pattern (most commonly used with custom element extensions, Section 3), the deployed archive contains *only* the metadata file – the actual application code is instead hosted externally, at a URL the metadata simply references, and Liferay loads the application directly from that external location at runtime rather than serving it itself. This pattern is particularly valuable for applications that are independently developed, built, and deployed on their own release cycle (a separately-maintained microfrontend) and that need to be updated without requiring a fresh Liferay-side deployment every time the remote application itself changes.

Extension Types Overview

Q2.1

What are the main categories the various client extension types fall into, at a high level, before getting into each specific type's details?

The many specific client extension types broadly cluster into a few functional categories: **widget/UI** types that add new interactive components to pages (`custom-element`, `iframe`); **styling** types that modify a site's appearance (`theme-css`, `global-css`, `theme-favicon`, `theme-spritemap`); **scripting** types that inject JavaScript logic (`global-js`, `theme-js`); **data/content** types (`object-action`, `fds-cell-renderer`, notification and workflow-related types); **integration/auth** types for connecting external systems (`oauth-application-headless-server`, `oauth-application-user-agent`); and **configuration/bulk-data** types (`instance settings`, `site-initializer`, `batch`) for deploying configuration and seed data as code. Understanding which category a given customization need falls into is usually the fastest way to narrow down which specific extension type is the right tool for the job.

Q2.2

What is a `custom-element` client extension, and what does it let you add to a Liferay page that the other extension types don't directly provide?

A custom-element client extension packages a JavaScript application (built with any modern frontend framework – React, Vue, Angular, or plain JavaScript) as a genuine, registered **widget** that site administrators can add to any page through the ordinary page builder, exactly like any of Liferay's own built-in widgets. This is the primary mechanism for embedding a genuinely custom, interactive application's UI directly into a Liferay page – a custom dashboard, a specialized data-entry form beyond what a standard Objects form container provides, or any bespoke frontend experience – registered in the browser as a real Web Component (custom HTML element), letting it integrate naturally into the page's DOM alongside Liferay's own components.

Q2.3

What is an `iframe` client extension, and when would you choose it over building a proper `custom-element` extension for embedding external content or functionality on a page?

An iframe client extension embeds an entirely separate web application or page, hosted anywhere, inside a sandboxed iframe widget on a Liferay page – the simplest possible way to surface external content or functionality without needing that external application to be rebuilt or adapted for Liferay integration at all. This is the right choice specifically when you need to embed something you don't control or don't want to deeply integrate (an existing third-party dashboard, tool, or application that already exists as a standalone web app) and don't need tight, bidirectional communication between the embedded content and the surrounding Liferay page – a custom-element extension is the better choice instead whenever you need the embedded application to feel like a genuinely native part of the page (sharing styling, participating directly in the page's own DOM, or communicating more directly with Liferay's own frontend state) rather than living in an isolated iframe sandbox.

Q2.4

What is the difference between a `global-js`/`global-css` client extension and a `theme-js`/`theme-css` client extension, at the level of what each one is fundamentally scoped to affect?

Global JS/CSS extensions add script or style resources that apply broadly across pages, independent of which specific theme a given site happens to use – suited to site-wide behavior or styling that should apply consistently regard-

less of theme. **Theme** JS/CSS extensions instead specifically modify or replace a *theme's own* scripts and styles, tightly coupled to that theme's own structure and design system (Section 4 covers this distinction, and the broader theme-vs-global-CSS choice, in much more depth) – the key structural difference is that theme-scoped extensions are conceptually part of defining or overriding a specific theme's own identity, while global extensions layer additional behavior or styling on top of whatever theme is already active, without needing to know or care about that theme's own internal implementation.

Q2.5

What are `object-action` and `fds-cell-renderer`/`fds-filter` client extensions, and how do they let you extend Liferay Objects and frontend data sets with genuinely custom logic?

An `object-action` client extension lets you implement custom business logic (beyond what the built-in object action triggers and conditions, covered in the companion Liferay Objects deck, can express) as external code that Liferay invokes at the appropriate point in an object entry's lifecycle. `fds-cell-renderer` and `fds-filter` (Frontend Data Set) client extensions let you customize how a specific column is *displayed* in a data set table (a custom cell renderer – e.g. rendering a status field as a colored badge rather than plain text) or add a genuinely custom *filter* control beyond the data set widget's built-in filtering options, respectively – together, these extension types are the standard way to extend Liferay Objects' and data sets' out-of-the-box capabilities with bespoke, project-specific logic without needing a full custom-element replacement for what is otherwise a mostly-standard Objects-backed UI.

Custom Element Extensions

Q3.1

What two files/components make up the deployable archive for a standard (non-remote) custom element client extension, and what role does each play?

A standard custom element extension's deployable ZIP archive contains, at minimum, two key components: the `client-extension.yaml` file, containing the extension's metadata – its type, name, and configuration – that tells Liferay how to register and expose it as a widget; and the compiled **JavaScript file(s)**, containing the application's entire logic, bundled and ready to run directly in the browser. When a site administrator adds the resulting widget to a page, Liferay serves this bundled JavaScript directly to the browser, which then registers the corresponding custom HTML element and renders the application wherever that widget is placed on the page.

```
assembleType: client-extension
name: my-react-custom-element
type: customElement
elementPath: /my-react-app.js
htmlElementName: my-react-app
friendlyURLMapping: my-react-app
```

Q3.2

Why does building a modern frontend framework application (React, Vue, Angular) as a custom element require specifically bundling it as a self-contained Web Component, rather than deploying the framework's ordinary build output directly?

Liferay pages can host many independently-developed widgets simultaneously, each potentially built with entirely different frontend frameworks or even different versions of the *same* framework – without careful isolation, two different widgets each expecting to fully control the page's global JavaScript environment could easily conflict with each other (colliding global variables, incompatible framework versions loaded simultaneously). Building a custom element as a genuine, encapsulated **Web Component** (using the browser's native Custom Elements API) ensures the application's internal implementation details – its specific framework, its component tree, its internal state – remain cleanly encapsulated behind the standard, framework-agnostic custom-element boundary, letting it coexist safely on the same page alongside Liferay's own components and any number of other independently-built custom elements, each isolated from interfering with one another.

Q3.3

What are "externals" in a custom element's build configuration, and why does explicitly excluding React (or another shared library) from your own bundle – relying on Liferay's own already-loaded copy instead – reduce bundle size and avoid duplicate-library conflicts?

When building a custom element with a framework like React, the build tool (Vite, Webpack, or similar) can be configured to treat certain dependencies as **external** – meaning the build assumes those libraries will already be available in the page's runtime environment, rather than bundling its own separate copy of them into the extension's own JavaScript output. Since Liferay itself already loads React, ReactDOM, and its own Clay component library for its native UI, marking these same libraries as externals in your own custom element's build configuration lets your extension reuse Liferay's already-loaded copies – meaningfully shrinking your own extension's bundle size (no need to duplicate an entire copy of React within your bundle) and avoiding subtle bugs or conflicts that can arise from having two separate, potentially different-versioned copies of the same library simultaneously loaded and active on the same page.

Q3.4

What is the practical value of deploying a custom element as a remote application (Section 1) rather than bundling its code directly, specifically in a scenario where the same underlying application needs to be updated frequently and independently of Liferay itself?

Consider a custom element built and maintained by a separate frontend team with its own independent release cadence, deployed to many different Liferay environments (client sites, staging, production) that shouldn't all need to be individually redeployed every time that frontend team ships an update. Deploying it as a remote application means the Liferay-side deployment package contains only the lightweight metadata referencing the remote application's URL – once that initial metadata deployment is in place, the frontend team can update and redeploy the actual application code at its externally-hosted URL as often as needed, with every Liferay environment referencing that same URL automatically picking up the update the next time the widget loads, entirely without needing a fresh client-extension redeployment on the Liferay side for every single application update.

Theming & Styling

Q4.1

What is the theme CSS client extension, and how does it let you fully replace a site's styling using the same modern client-extension deployment model, without the traditional drawbacks of building a theme as an OSGi plugin?

The theme CSS client extension provides much of the same capability as a traditional theme plugin – fully replacing a site's styling with custom settings – but deployed as a client extension rather than an OSGi module, gaining all the upgrade-safety and deployment-flexibility benefits discussed in Section 1. Because it's replacing the theme's styling wholesale, you must include any existing styling you want to keep, rather than layering on top of it – fully overriding a theme this way remains, by its very nature, a complex and resource-intensive undertaking, which is precisely why Liferay recommends limiting how often an organization replaces or overhauls a site's entire theme, and generally favors the more targeted global CSS extension (discussed next) or style-book-based token customization for anything short of a genuinely complete visual overhaul.

Q4.2

What is the global CSS client extension, and why has it become the recommended default tool for most day-to-day styling customization, rather than reaching for the theme CSS extension?

The global CSS client extension adds CSS resources directly to a page or specific element, entirely independent of whichever theme happens to be active – rather than needing to override an entire theme's styling to make one targeted visual change. This makes it considerably more versatile for the vast majority of real-world styling needs: implementing a one-off styling change, modifying just one specific part of the platform, or adding entirely custom, non-Clay-based styling components – all without needing to touch, understand, or risk breaking anything about the site's underlying theme. Because of this flexibility and the ability to load only the specific CSS resources actually needed on a given page (rather than an entire theme's full styling payload), the global CSS client extension is Liferay's recommended primary tool for deploying custom stylesheets whenever you don't specifically need to define style-book tokens.

Q4.3

What are frontend tokens, and how does defining them within a theme CSS client extension let a site's visual design be configured through style books rather than requiring hand-written CSS overrides for every styling change?

Frontend tokens are named styling rules – each corresponding to a specific CSS variable – collectively defined as a theme's **frontend token definition**, exposed to site administrators as configurable options within Liferay's style book UI. Rather than repeatedly writing and redeploying hand-crafted CSS overrides for common styling adjustments (a brand color, a heading font, a spacing scale), defining a well-structured set of frontend tokens lets those same adjustments be made by a non-developer administrator directly through the style book's UI, simply picking a color or adjusting a value – Liferay specifically recommends this frontend-token-definition approach as one of the best uses of the theme CSS client extension, rather than using it purely to hand-write a large volume of literal, one-off CSS rules.

Q4.4

When using both a theme CSS client extension and a global CSS client extension together on the same site, why does the specific load order between them matter, and which one is guaranteed to load second?

When both extension types are used together – theme CSS establishing a site's foundational styling and structure, global CSS then layering more targeted, specific adjustments on top – the **load order** directly determines which set of rules wins when both define conflicting styles for the same element, following ordinary CSS cascade rules where later-loaded rules generally take precedence over earlier ones for equally-specific selectors. Liferay's documented behavior is that **global CSS loads after theme CSS** – meaning global CSS extensions can reliably be used to override or fine-tune specific aspects of whatever the theme CSS extension has already established, making "theme CSS for the foundation, global CSS for the fine-tuning" a reliable, predictable, and commonly recommended layering pattern for combining the two extension types on the same site.

Configuration as Code

Q5.1

What is ”Configuration as Code” in the context of client extensions, and what class of Liferay settings – previously configured manually, click by click, through the admin UI – can now be deployed automatically as part of a client extension instead?

Configuration as Code refers to defining Liferay instance-level and application-level settings – previously requiring an administrator to manually navigate through the admin UI and configure each setting by hand, separately, in every single environment – as declarative configuration files bundled within a client extension, deployed automatically and consistently the same way any other client extension is deployed. Instance settings client extensions specifically let a wide variety of Liferay configurations be defined this way, each referenced by its underlying configuration **PID** (persistent identifier), and other specialized configuration-as-code extension types (like `oauth-application-headless-server`) handle specific categories of configuration (OAuth2 application registration, for instance) with their own dedicated, purpose-built YAML structure.

Q5.2

What is a site-initializer client extension, and how does it let you define an entire site’s structure – pages, content, and configuration – as deployable, version-controlled code rather than requiring manual setup through the UI?

A site-initializer client extension packages a complete definition of a site’s initial structure – its pages and page layout, site-level configuration, and seed content – as a deployable archive that, when applied, automatically provisions a brand-new site matching that exact definition, rather than requiring an administrator to manually build out that same site structure by hand through the Liferay UI every time it’s needed. This is particularly valuable for organizations that need to reliably, repeatedly stand up many similarly-structured sites (a consistent site template rolled out across many different departments, regions, or client instances), or that want a new site’s baseline structure to be genuinely version-controlled and reproducible, rather than depending on an administrator’s manual, potentially inconsistent recreation of the same setup steps each time.

Q5.3

What is a `batch` client extension, and how does it let you deploy seed or reference data (such as initial object entries) as part of an automated deployment, rather than requiring that data to be manually entered after the fact?

A batch client extension bundles a defined set of data operations – creating, updating, or otherwise seeding records – that Liferay executes automatically as part of the extension’s deployment, letting an environment’s baseline reference data (initial Objects entries, lookup/picklist values, or other seed content) be provisioned consistently and automatically alongside the rest of an automated deployment pipeline, exactly the kind of workflow discussed in the companion Liferay Objects deck’s coverage of promoting object definitions and their associated data across environments in a reliable, repeatable way – avoiding the need for someone to manually re-enter the same reference data by hand in every new or rebuilt environment.

Q5.4

Why does treating instance configuration, site structure, and seed data as client-extension-deployable code represent a broader shift in how Liferay projects are built and maintained, echoing the ”infrastructure as code” philosophy familiar from general DevOps practice?

Just as the broader software industry moved from manually, individually configuring servers toward defining infras-

tructure declaratively in version-controlled code (directly discussed in the CI/CD deck's coverage of Infrastructure as Code), Liferay's Configuration as Code client extensions represent the same underlying philosophy applied specifically to Liferay instance/site configuration: every environment's configuration becomes reviewable through the same pull-request process as application code, reproducible identically across development, staging, and production, and automatically re-deployable if an environment ever needs to be rebuilt from scratch – directly eliminating the configuration drift, inconsistency, and "it worked in staging but not production" class of problems that manual, UI-driven, click-by-click configuration across multiple separate environments has historically been prone to.

Packaging & Deployment

Q6.1

What is a Liferay Universal File Format Archive (an ”LFF” archive), and what does the packaging process actually assemble into this final deployable artifact?

Client extension projects are built as deployable archives called Liferay Universal File Format Archives – a standardized ZIP-based packaging format containing the extension's `client-extension.yaml` metadata file plus (for non-remote-application extensions) its actual bundled assets (JavaScript, CSS, images), assembled by the build tooling from wherever the project's build process outputs its compiled artifacts (commonly a `build/static` or similar output directory, referenced in the YAML's `assemble` configuration) into this one final, self-contained archive ready to be deployed to a Liferay instance.

```
assemble:
  - from: build/static
    into: static
```

Q6.2

What is the difference in how a client extension archive is actually deployed across Liferay Self-Hosted, PaaS, and SaaS environments, given that the same underlying archive works across all three deployment types?

For **Self-Hosted** and **PaaS** environments, an administrator with direct or managed filesystem access deploys a client extension simply by placing its built archive into the appropriate `osgi/client-extensions` directory of the running Liferay instance, where it's automatically picked up and activated. For **SaaS** (Liferay DXP Cloud) environments, where customers don't have this kind of direct filesystem access, deployment instead goes through a dedicated command-line tool that uploads the archive to Liferay's cloud platform for automated deployment – the underlying archive itself doesn't need to differ at all between these scenarios, only the specific mechanism used to get that same archive actually deployed and running on the target environment.

Q6.3

Why do remote-application custom elements deployed to different environments (development, staging, production) typically need environment-specific configuration for their remote URL, and how is this commonly handled using environment variables in the underlying OSGi configuration?

A remote-application custom element's deployed package references an external URL where its actual application code is hosted – but that URL will often genuinely differ between environments (a development team's own local or staging server URL during development, versus the production-hosted application's real URL once live) rather than being one single, universally-correct value baked permanently into the deployment package. This is commonly handled by defining the remote application's base URL in its underlying OSGi configuration using an environment-variable reference (e.g. `baseURL="$[env:REMOTE_DOMAIN]"`) rather than a hardcoded literal value – letting each environment's own `REMOTE_DOMAIN` environment variable supply the correct, environment-specific URL at runtime, so the exact same underlying client extension package can be deployed unchanged across every environment, with only that one environment variable differing between them.

Q6.4

What is the difference between configuring a client extension through the Liferay UI (Applications → Client Extensions) versus defining it entirely through project files in a Liferay Workspace, and can a UI-configured client extension later be exported for source control?

Client extensions can be added and configured directly through the Liferay admin UI (under Applications → Client Extensions, choosing a type and configuring its properties interactively) as a faster, more immediate way to get a client extension running without setting up a full local Workspace development project – particularly convenient for simpler styling or configuration-only extensions. For client extensions specifically configured this way through the UI, Liferay provides an export/import capability that preserves the extension's external reference code and any page-level references already established, letting a UI-configured extension later be exported into a proper, version-controlled project file structure – though this export/import path is specifically available only for extensions that were originally configured through the UI in the first place, not a general mechanism for converting between every possible client extension configuration approach.

Integration Patterns

Q7.1

What is a JavaScript import map entry client extension, and how does it let multiple separate frontend client extensions share a common utility library without each one bundling and re-requesting its own duplicate copy?

An import map entry client extension registers a JavaScript module (a shared utility library, a common component set) into the browser's import map, making it available to be imported by name from any other client extension on the page, rather than each individual extension needing to bundle its own private copy of that same shared code. This directly addresses the same kind of duplication concern discussed in Section 3 regarding "externals" (avoiding bundling a duplicate copy of React when Liferay already provides one) – but generalized to any custom, project-specific shared library a team wants multiple independent client extensions to reuse in common, letting the shared code be loaded and cached once and referenced by many different extensions, rather than being redundantly duplicated (and separately re-downloaded) inside each individual extension's own bundle.

Q7.2

How does the micro frontend architectural pattern relate conceptually to Liferay's custom element client extensions, and why does Liferay's own documentation explicitly frame client extensions within this broader industry pattern?

Micro frontends extend the general idea behind microservices – breaking a large, monolithic application into smaller, independently developed and deployed pieces – specifically to the frontend/UI layer, letting a large web application be assembled from several independently-built, independently-deployed frontend pieces rather than one single, tightly-coupled frontend codebase. Liferay's custom element client extensions map directly onto this same architectural pattern: each custom element is an independently developed, independently deployed frontend "micro-application," assembled together onto a shared Liferay page at runtime – Liferay explicitly frames client extensions this way specifically to help developers already familiar with micro frontend architecture (from working with other platforms) quickly map their existing mental model onto how Liferay's own client extension system works, rather than needing to learn an entirely unfamiliar, Liferay-specific paradigm from scratch.

Q7.3

How would you combine a custom-element client extension with a Liferay Objects headless API to build a genuinely dynamic, data-driven widget – for example, a React-based chart rendering live Object data on a dashboard?

This is one of the most common and practically valuable client-extension integration patterns: build the chart's actual rendering and interactivity as a React application, packaged as a custom-element client extension (Section 3) so it can be placed on any Liferay page like any other widget; at runtime, that React application makes calls to the target Liferay Object's generated REST or GraphQL API (covered in depth in the companion Liferay Objects deck) to fetch the live, current data it needs to render; and the resulting widget then displays that data using whatever charting library the React application bundles. This pattern – a custom-element frontend consuming an Objects-generated headless API – is precisely the combination that lets the two decks in this pair (Objects for the data model, Client Extensions for the custom frontend) work together to deliver a genuinely custom, dynamic, data-driven application experience, built entirely on top of Liferay's own low-code data layer without needing any traditional OSGi backend development at all.

Q7.4

What are notification-type client extensions, and how do they let you add a genuinely custom notification channel (beyond Liferay's built-in email/in-app notifications) that other parts of the platform, like object actions, can trigger?

A notification-type client extension registers a new, custom category of notification that Liferay's broader notification framework (and features that trigger notifications, like the object actions covered in the companion Liferay Objects deck) can then send through – letting an organization integrate a notification channel Liferay doesn't natively support out of the box, such as posting a message to a Slack channel, sending an SMS, or triggering a call to some other external notification service, using exactly the same triggering mechanisms (an object entry being created, a workflow stage being reached) that would otherwise only be able to send a standard built-in email or in-platform notification.

Best Practices & Migration

Q8.1

Why does Liferay's documentation explicitly note that "modules and themes are supported for backwards compatibility," and what does this framing signal about the platform's own long-term recommended direction?

Traditional OSGi modules and theme plugins remain fully supported – existing customizations built this way will continue to work, and there's no forced, immediate migration requirement – but explicitly framing this ongoing support as being specifically "for backwards compatibility" (rather than as an equally-recommended, first-class ongoing option) signals that Liferay's own official, forward-looking guidance considers client extensions the **preferred**, modern approach for new development going forward. This is a common and recognizable pattern in platform evolution more broadly: an older approach remains supported to avoid breaking existing investments, while new documentation, tooling investment, and official guidance increasingly steers toward the newer approach – developers and architects planning new Liferay customization work should treat this framing as a clear, explicit signal to default to client extensions for new work, reserving traditional OSGi module development specifically for the comparatively narrow set of cases client extensions genuinely cannot yet address.

Q8.2

What kinds of customizations still generally require a traditional OSGi module, rather than being achievable through any of the current client extension types?

While client extensions now cover an increasingly broad range of customization needs – widgets, styling, scripting, configuration, and integration – some categories of deep platform customization still generally require traditional OSGi module development: implementing genuinely new backend service logic that must run *within* Liferay's own server-side process (rather than being called as an external integration), overriding or extending certain core platform behaviors at a level client extensions aren't designed to reach into, or building certain categories of deeply platform-integrated portlets with requirements beyond what a custom-element widget can address. Recognizing which specific category a given customization requirement falls into – and confirming whether a suitable client extension type genuinely exists for it before defaulting to traditional module development – is an important, recurring architectural decision point on any real Liferay project going forward.

Q8.3

What is a reasonable decision process for choosing which specific client extension type to use for a given customization requirement, given the range of types covered throughout this deck?

A reasonable, structured decision process: first ask whether the requirement is primarily about **visual appearance** (Section 4 – choose global CSS for targeted styling, theme CSS for wholesale theme replacement or frontend-token definition); then whether it's about adding **new interactive functionality** to a page (Section 3 – choose custom-element for a tightly-integrated, framework-built widget, or iframe for simply embedding an existing external application); then whether it's about **extending Objects or data sets** with custom logic (Section 2 – object-action, fds-cell-renderer, fds-filter); then whether it's about **deploying configuration or seed data** as part of an automated pipeline (Section 5 – instance settings, site-initializer, batch); and finally whether it's a specialized **integration** need (Section 7 – OAuth application types, notification types, import map entries) – working through this rough decision sequence, rather than reaching immediately for the most flexible, general-purpose option (typically custom-element) for every single requirement, generally leads to a simpler, more maintainable, and more purpose-appropriate final implementation.

Q8.4

Why does migrating an existing, traditional OSGi-based theme or portlet to the equivalent client extension typically deliver a genuine, measurable upgrade-safety benefit – beyond simply being the currently recommended approach – for organizations planning their next Liferay major version upgrade?

Traditional OSGi modules are directly coupled to Liferay's own internal Java APIs and runtime structure, meaning a major Liferay version upgrade can require real, sometimes substantial rework to an existing module if the specific internal APIs it depended on have changed or been removed between versions – exactly the kind of upgrade friction the companion Liferay Objects deck also discusses in the context of object-related customizations, and exactly what the Publications improvements discussed there were specifically designed to reduce. Because client extensions run outside Liferay's own runtime and interact with the platform only through its stable, externally-facing configuration and API surface (rather than internal implementation details), a client extension is considerably less likely to break during a major version upgrade in the first place – making proactive migration of existing OSGi-based customizations to their client-extension equivalents a genuinely valuable, upgrade-risk-reducing investment for organizations planning ahead for their next major Liferay upgrade cycle, not merely a matter of following current best-practice guidance for its own sake.

End of deck – see the companion Liferay Objects deck for the data-modeling side of the platform.