Microservice Client Extensions in Liferay DXP

Architecture, Configuration, OAuth 2, Deployment, Security, and Integration

Liferay DXP Technical Notes

September 28, 2026

Abstract

This document explains the architecture and deployment of Microservice Client Extensions in Liferay DXP. It discusses how external services can integrate with Liferay without being deployed inside the Liferay JVM, how client extensions describe the integration contract, how OAuth 2 can secure communication, how Docker and container-based deployment models fit into the architecture, and how microservices can interact with Liferay Headless APIs.

Contents

1 Introduction **4**

2 What is a Client Extension? **4**

3 What is a Microservice Client Extension? **5**

4 Traditional OSGi vs Microservice Architecture **5**

5 Architectural Comparison **6**

6 Why Use Microservice Client Extensions? **6**

7 High-Level Architecture **7**

8 Role of client-extension.yaml **8**

9 Project Structure **8**

10 Example FastAPI Microservice **9**

11 Containerizing the Microservice **9**

12 Client Extension Build **10**

13 Deployment Concept **11**

14 Deployment Flow **11**

15 Communication Direction **11**

15.1 Liferay to Microservice 12

15.2 Microservice to Liferay 12

16 OAuth 2 Authentication **12**

17 Client Credentials Flow **13**

18 Token Request **13**

19 Calling Liferay Headless APIs **13**

20 Python Example **14**

21 Environment Variables **15**

22 Development vs Production **15**

23 Deployment Using Containers **15**

24 Kubernetes Architecture **16**

25 Independent Scaling **17**

26 Failure Isolation **17**

27 Timeouts **17**

28 Retries **17**

29 Circuit Breaker **18**

30 Health Checks **18**

31 Logging and Observability **18**

32 Correlation IDs **19**

33 Security Architecture **19**

34 Least Privilege **19**

35 Example AI Architecture **20**

36 Example Quantum Architecture **20**

37 Microservice vs Remote App **21**

38 Deployment Pipeline **21**

39 Versioning **22**

40 API Contract **22**

41 Common Deployment Problems **23**
41.1 Microservice is Unreachable 23
41.2 401 Unauthorized 23
41.3 403 Forbidden 23
41.4 Connection Refused 23
41.5 Timeout 23

42 Recommended Production Architecture **24**

43 Best Practices **24**

44 Interview Questions **25**

45 Multiple Choice Questions **26**

46 Quick Revision **29**

47 Key Takeaway **29**

48 Official Reference **29**

1 Introduction

Traditional Liferay customizations are frequently deployed as OSGi modules inside the Liferay runtime.

For example:

```
Liferay JVM
    |
    +-- MVC Portlet
    +-- Service Builder
    +-- REST Builder
    +-- OSGi Component
    +-- Event Listener
```

This architecture is powerful, but it creates runtime coupling between the customization and Liferay.

A **Client Extension** provides another integration model.

Instead of deploying all custom application logic directly into the Liferay JVM, functionality can run externally and integrate with Liferay through supported interfaces.

Core Idea

A Microservice Client Extension allows application logic to run outside the Liferay JVM while maintaining a defined integration relationship with Liferay DXP.

The architecture becomes:

```
+-----------------------+
|      Liferay DXP      |
+-----------+-----------+
            |
            | HTTP / OAuth 2 / APIs
            |
            v
+-----------------------+
| External Microservice |
| Java / Python / Node  |
| Go / .NET / etc.      |
+-----------------------+
```

2 What is a Client Extension?

A Liferay Client Extension is a mechanism for extending or integrating with Liferay while reducing direct coupling to the portal runtime.

Client extensions can support different extension scenarios.

Depending on the Liferay version and platform capabilities, examples can include:

- Custom element integrations
- Global CSS
- Global JavaScript
- Theme CSS

- Object actions
- Object validation
- Workflow-related integrations
- OAuth applications
- External microservices

The exact supported types depend on the Liferay release.

3 What is a Microservice Client Extension?

A Microservice Client Extension is intended for external application logic that needs to participate in a Liferay solution.

The service can be developed independently using technologies such as:

- Java / Spring Boot
- Python / FastAPI
- Node.js
- Go
- .NET
- Other HTTP-capable frameworks

For example:

```
Liferay
   |
   +------> FastAPI AI Service
   |
   +------> Spring Boot Payment Service
   |
   +------> Node.js Notification Service
   |
   +------> Python Quantum Service
```

This allows teams to choose technology according to the workload rather than requiring every feature to execute inside Liferay.

4 Traditional OSGi vs Microservice Architecture

Consider a custom business operation.

In the traditional model:

```
Browser
   |
   v
Liferay
   |
   v
```

```
OSGi Module
   |
   v
Business Logic
```

The custom logic runs inside the Liferay JVM.
With an external microservice:

```
Browser
   |
   v
Liferay
   |
   +----------------+
                    |
                    v
              Microservice
                    |
                    v
              Business Logic
```

The application logic executes outside Liferay.

5 Architectural Comparison

Characteristic	**OSGi Module**	**Microservice Client Extension**
Runtime	Inside Liferay JVM	Outside Liferay JVM
Technology	Primarily Java/OSGi	Language/framework independent
Deployment	Liferay deployment lifecycle	External/container deployment lifecycle
Failure isolation	Lower	Potentially stronger
Scaling	Usually tied to Liferay	Can scale independently
Communication	Java/OSGi services	HTTP/API-based
Security	Portal/OSGi security model	OAuth/API/network security
Upgrade coupling	Potentially stronger	Potentially reduced

6 Why Use Microservice Client Extensions?

They are useful when the application requires:

- Independent deployment
- Independent scaling
- Non-Java technologies
- AI/ML workloads
- Quantum computing integration
- Data processing

- External APIs
- Long-running computation
- Specialized libraries
- Better failure isolation

Example

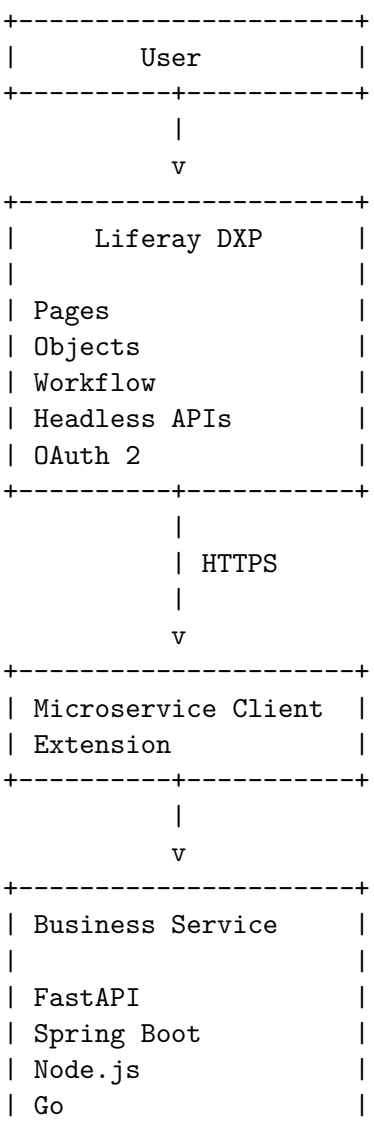
Suppose Liferay provides the user-facing portal while a Python FastAPI service performs machine-learning inference.
Instead of embedding Python inside Liferay:

$$\text{Liferay} \rightarrow \text{FastAPI} \rightarrow \text{ML Model}$$

The Python service can run independently while Liferay handles portal users, permissions, pages, Objects, workflows, and presentation.

7 High-Level Architecture

A typical architecture is:



```
+----------+-----------+
           |
           v
+----------------------+
| DB / AI / Quantum /  |
| External Systems     |
+----------------------+
```

8 Role of client-extension.yaml

A central artifact in a client-extension project is typically:

`client-extension.yaml`

It describes the client extension and its integration configuration.
Conceptually:

```
my-microservice:
    name: My Microservice
    type: microservice
```

The exact schema and properties depend on the Liferay version and client extension type.

Important

Do not treat conceptual YAML examples as universally deployable. The supported properties and structure must be validated against the target Liferay DXP release.

9 Project Structure

A conceptual project structure can look like:

```
my-microservice/
|
+-- client-extension.yaml
|
+-- Dockerfile
|
+-- src/
|   |
|   +-- application code
|
+-- requirements.txt
|
+-- README.md
```

For Java:

```
my-service/
|
+-- client-extension.yaml
+-- Dockerfile
+-- build.gradle
+-- src/
    +-- main/
        +-- java/
```

For Python:

```
quantum-service/
|
+-- client-extension.yaml
+-- Dockerfile
+-- requirements.txt
+-- app/
    +-- main.py
    +-- services/
    +-- models/
```

10 Example FastAPI Microservice

Consider a simple Python service:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/health")
def health():
    return {
        "status": "UP"
    }

@app.post("/api/process")
def process(payload: dict):

    return {
        "success": True,
        "input": payload
    }
```

The service runs independently from Liferay.
For example:

```
http://microservice:8000
```

or in production:

```
https://service.example.com
```

11 Containerizing the Microservice

A common deployment strategy is Docker.
For the FastAPI example:

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir \
    -r requirements.txt
```

```
COPY . .

CMD [
    "uvicorn",
    "app.main:app",
    "--host",
    "0.0.0.0",
    "--port",
    "8000"
]
```

Build the image:

```
docker build -t my-liferay-service .
```

Run locally:

```
docker run \
    -p 8000:8000 \
    my-liferay-service
```

Then test:

```
curl http://localhost:8000/health
```

Expected result:

```
{
    "status": "UP"
}
```

12 Client Extension Build

In a Liferay workspace, client extensions are commonly located under:

```
client-extensions/
```

For example:

```
liferay-workspace/
|
+-- client-extensions/
|   |
|   +-- my-microservice/
|       |
|       +-- client-extension.yaml
|       +-- ...
|
+-- modules/
+-- themes/
+-- gradle/
```

A typical Gradle workflow may involve:

```
./gradlew build
```

or building the specific client extension project.
On Windows:

```
gradlew.bat build
```

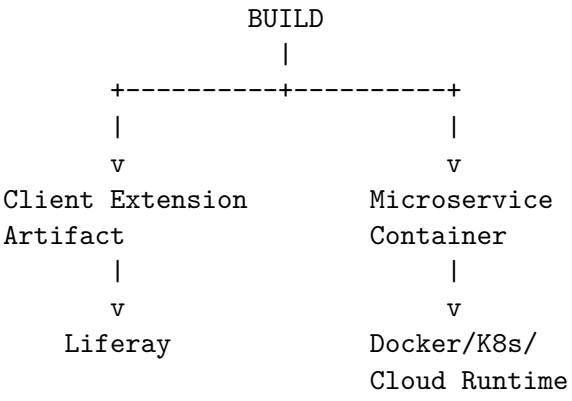
The build packages the client-extension configuration into an artifact that Liferay can deploy.

13 Deployment Concept

A key distinction is that there are effectively two deployment concerns:

Liferay Configuration + External Service Deployment

The client extension registers/configures the integration with Liferay.
The actual microservice runs separately.
Conceptually:



14 Deployment Flow

A practical flow can be understood as:

1. Develop the microservice.
2. Test the service independently.
3. Define the client extension.
4. Configure authentication.
5. Build the client extension.
6. Build the service/container.
7. Deploy the client extension to Liferay.
8. Deploy the microservice to its runtime.
9. Configure environment variables and secrets.
10. Verify connectivity.
11. Test authentication.
12. Execute end-to-end integration testing.

15 Communication Direction

There are two important directions of communication.

15.1 Liferay to Microservice

Liferay → Microservice

Examples:

- Object Action
- Workflow action
- Webhook
- External validation
- AI inference request

15.2 Microservice to Liferay

Microservice → Liferay Headless API

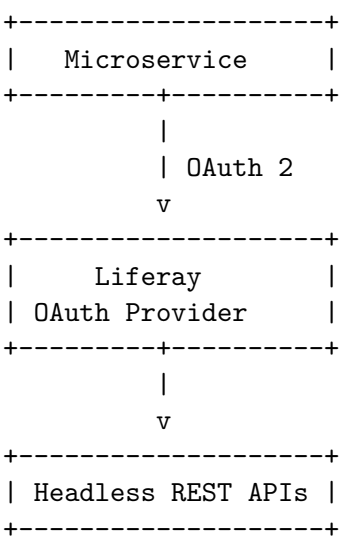
Examples:

- Read Liferay Objects
- Update Object entries
- Retrieve documents
- Create content
- Query users
- Invoke Headless APIs

Many real systems use both directions.

16 OAuth 2 Authentication

A microservice should generally not call protected Liferay APIs without authentication.
OAuth 2 can provide service-to-service authorization.
The architecture becomes:



17 Client Credentials Flow

For machine-to-machine communication, a common OAuth pattern is:

Client Credentials Grant

There is no interactive user login.
Instead:

```
Microservice
    |
    | client_id
    | client_secret
    v
OAuth Token Endpoint
    |
    | access_token
    v
Microservice
    |
    | Bearer Token
    v
Liferay API
```

Conceptually:

Client Credentials → Access Token → API Call

18 Token Request

A conceptual token request is:

```
POST /o/oauth2/token

grant_type=client_credentials
client_id=YOUR_CLIENT_ID
client_secret=YOUR_CLIENT_SECRET
```

The response can contain:

```
{
   "access_token": "...",
   "token_type": "Bearer",
   "expires_in": 600
}
```

The service then sends:

```
Authorization: Bearer ACCESS_TOKEN
```

when calling Liferay.

19 Calling Liferay Headless APIs

Suppose the microservice needs to retrieve Liferay Object entries.
Conceptually:

```
GET /o/c/orders/
Authorization: Bearer ACCESS_TOKEN
```

The response may contain:

```
{
    "items": [
        {
            "id": 101,
            "orderNumber": "ORD-1001"
        }
    ]
}
```

Thus:

$$\text{Microservice} \rightarrow \text{OAuth} \rightarrow \text{Liferay Headless API}$$

20 Python Example

A simplified Python example is:

```
import requests

LIFERAY_URL = "https://portal.example.com"

CLIENT_ID = "my-client"
CLIENT_SECRET = "my-secret"

token_response = requests.post(
    f"{LIFERAY_URL}/o/oauth2/token",
    data={
        "grant_type": "client_credentials",
        "client_id": CLIENT_ID,
        "client_secret": CLIENT_SECRET
    }
)

token_response.raise_for_status()

token = token_response.json()["access_token"]

response = requests.get(
    f"{LIFERAY_URL}/o/c/orders/",
    headers={
        "Authorization": f"Bearer {token}"
    }
)

response.raise_for_status()

print(response.json())
```

Security

Do not hard-code production OAuth secrets directly in application source code.
Use environment variables, container secrets, or an appropriate secret-management system.

21 Environment Variables

A microservice can use environment variables such as:

```
LIFERAY_URL
LIFERAY_CLIENT_ID
LIFERAY_CLIENT_SECRET
DATABASE_URL
LOG_LEVEL
```

Python example:

```
import os

liferay_url = os.environ["LIFERAY_URL"]

client_id = os.environ["LIFERAY_CLIENT_ID"]

client_secret = os.environ[
    "LIFERAY_CLIENT_SECRET"
]
```

This separates configuration from application code.

22 Development vs Production

Development might use:

```
Liferay:
http://localhost:8080

Microservice:
http://localhost:8000
```

Production might use:

```
Liferay:
https://portal.company.com

Microservice:
https://api.company.com
```

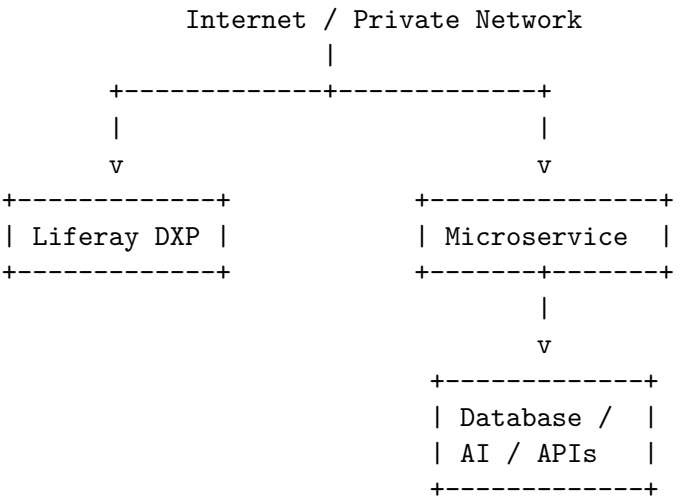
The application should therefore avoid hard-coded URLs.

23 Deployment Using Containers

The external service can be deployed using:

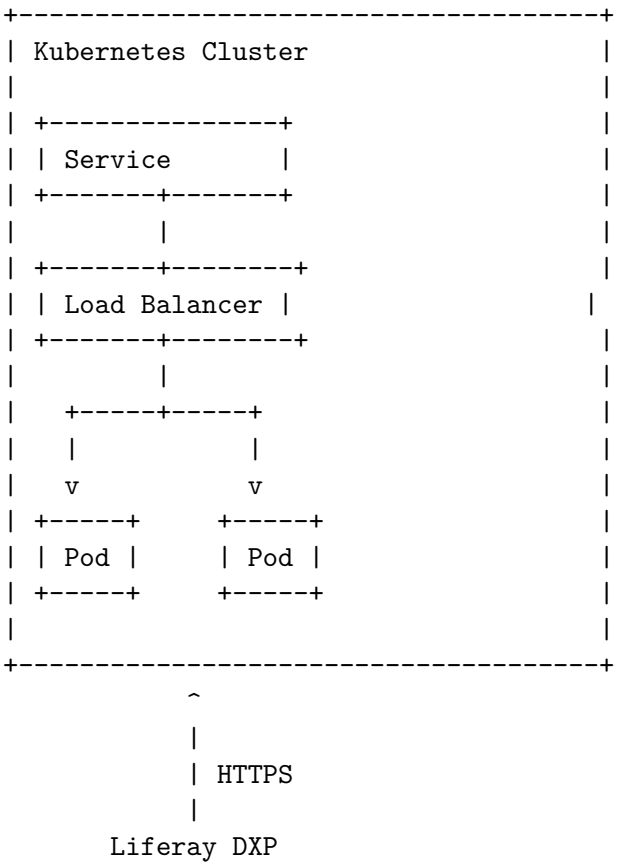
- Docker
- Kubernetes
- Cloud container platforms
- Virtual machines
- Managed application platforms

Conceptually:



24 Kubernetes Architecture

For larger deployments:



The microservice can scale independently:

$$1 \text{ pod} \rightarrow 5 \text{ pods} \rightarrow 20 \text{ pods}$$

without necessarily scaling the Liferay runtime itself.

25 Independent Scaling

Suppose Liferay handles:

$$1,000 \text{ portal users}$$

while an AI service receives:

$$50,000 \text{ inference requests}$$

The AI workload may require more compute resources than the portal.
With a microservice architecture:

$$\text{Scale(Liferay)} \neq \text{Scale(AI Service)}$$

Each component can be scaled according to its workload.

26 Failure Isolation

Suppose an AI service crashes.
With tightly embedded application logic:

$$\text{Application Failure} \rightarrow \text{Possible Portal Impact}$$

With an external microservice:

$$\text{Microservice Failure} \not\Rightarrow \text{Liferay JVM Failure}$$

Liferay can remain available and display an appropriate service error or retry later.

27 Timeouts

External calls must have explicit timeouts.
Bad approach:

```
requests.get(url)
```

Better:

```
requests.get(
    url,
    timeout=10
)
```

Without timeouts, slow external services can consume application resources indefinitely.

28 Retries

Transient failures may justify retry logic.
For example:

$$\text{Attempt } 1 \rightarrow 1s \rightarrow \text{Attempt } 2 \rightarrow 2s \rightarrow \text{Attempt } 3$$

This is an example of exponential backoff.
However, retries must be used carefully.
A POST operation that creates data should not automatically be repeated unless the operation is designed to be idempotent or otherwise protected against duplication.

29 Circuit Breaker

If an external service is repeatedly failing, continuing to call it can worsen system instability.

A circuit breaker conceptually behaves as:

$$\text{Closed} \rightarrow \text{Failures} \rightarrow \text{Open} \rightarrow \text{Recovery Test} \rightarrow \text{Closed}$$

This can protect Liferay and the external service during outages.

30 Health Checks

A microservice should expose a health endpoint.

For example:

```
GET /health
```

Response:

```
{
    "status": "UP"
}
```

A more advanced health endpoint might check:

- Database connectivity
- Liferay connectivity
- Model availability
- Message broker availability
- Required external APIs

31 Logging and Observability

A distributed architecture requires strong observability.

Useful information includes:

- Timestamp
- Request ID
- Correlation ID
- Service name
- HTTP status
- Response time
- Error details

For example:

```
requestId=abc-123
service=quantum-service
endpoint=/simulate
status=200
duration=840ms
```

32 Correlation IDs

A request may cross multiple systems:

$$\text{Browser} \rightarrow \text{Liferay} \rightarrow \text{Microservice} \rightarrow \text{Database}$$

A correlation ID can follow the request:

```
X-Correlation-ID: abc-123
```

Then logs across systems can be associated with the same transaction.

33 Security Architecture

A secure architecture should consider:

- HTTPS/TLS
- OAuth 2
- Least privilege
- Secret management
- Network restrictions
- Input validation
- Authentication
- Authorization
- Logging
- Token expiration

The goal is:

$$\boxed{\text{Authenticate} + \text{Authorize} + \text{Encrypt} + \text{Validate} + \text{Audit}}$$

34 Least Privilege

A microservice should receive only the permissions required for its function.

For example, if a service only reads orders:

$$\text{Permission} = \text{Read Orders}$$

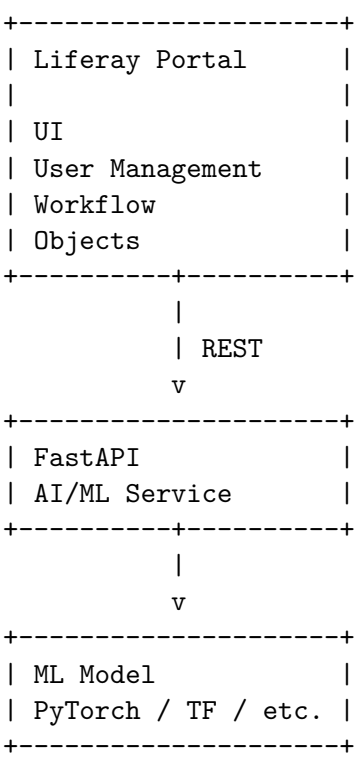
not:

$$\text{Permission} = \text{Portal Administrator}$$

This limits the impact of credential compromise.

35 Example AI Architecture

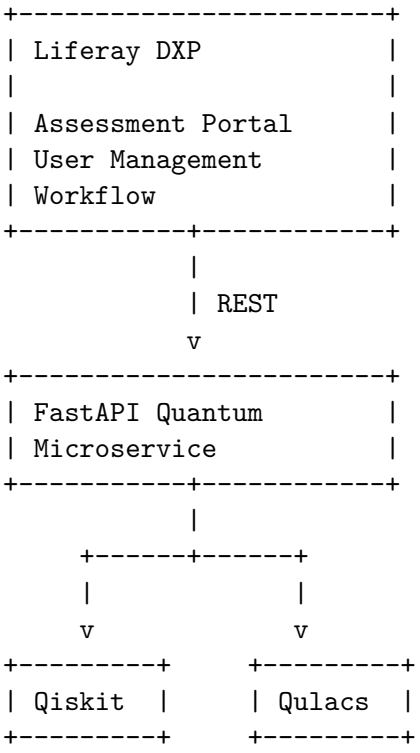
A useful Liferay architecture is:

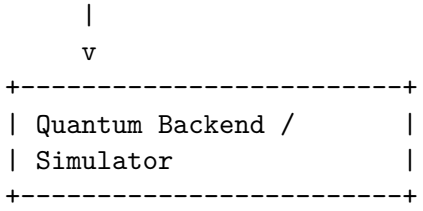


This avoids forcing Python machine-learning dependencies into the Liferay JVM.

36 Example Quantum Architecture

The same architecture works particularly well for quantum services.





This provides a clean separation:

$$\text{Liferay} = \text{Portal} + \text{Governance}$$

while:

$$\text{Microservice} = \text{Quantum Computation}$$

37 Microservice vs Remote App

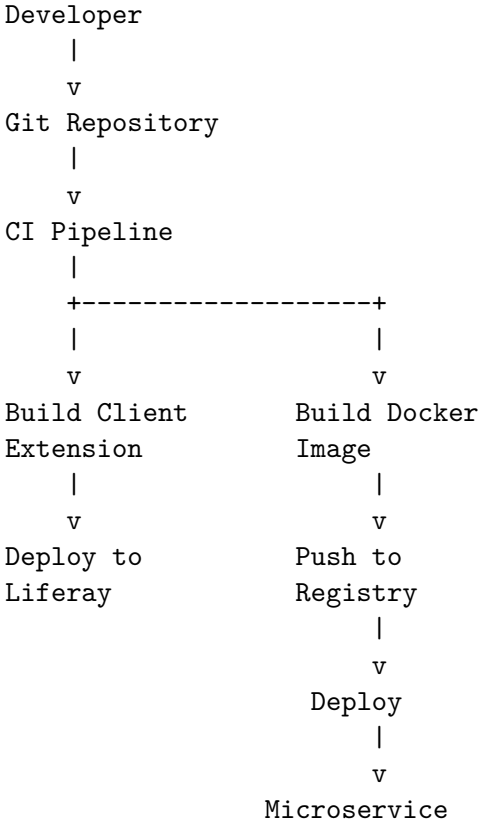
A useful conceptual distinction is:

	Microservice	**Remote UI/App**
Primary role	Backend/business logic	External user interface
Interaction	API/service calls	Browser-facing application
Typical technologies	FastAPI, Spring Boot, Node	React, Angular, Vue
UI required	No	Usually yes

A solution may contain both.

38 Deployment Pipeline

A CI/CD pipeline can be:



This allows the portal integration and service runtime to have controlled but independent deployment lifecycles.

39 Versioning

External services should expose versioned APIs.

For example:

```
/api/v1/diagnose
/api/v2/diagnose
```

instead of relying permanently on:

```
/diagnose
```

Versioning makes backward-compatible upgrades easier.

40 API Contract

The integration between Liferay and a microservice should have a clearly defined contract.

For example:

```
POST /api/v1/diagnose

Request:
{
    "patientId": "P10025",
    "imageId": 7812
}

Response:
{
    "reportId": "R-981",
    "prediction": "Normal",
    "confidence": 0.94
}
```

The contract should define:

- Endpoint
- HTTP method
- Authentication
- Request schema
- Response schema
- Error codes
- Timeout
- Version

41 Common Deployment Problems

41.1 Microservice is Unreachable

Check:

- DNS
- Service URL
- Firewall
- Container port
- Reverse proxy
- TLS certificate

41.2 401 Unauthorized

Check:

- Client ID
- Client secret
- Access token
- Token expiration
- OAuth configuration

41.3 403 Forbidden

Authentication may have succeeded, but the service identity may not have sufficient permissions.

41.4 Connection Refused

Verify:

```
host
port
container status
service status
network
```

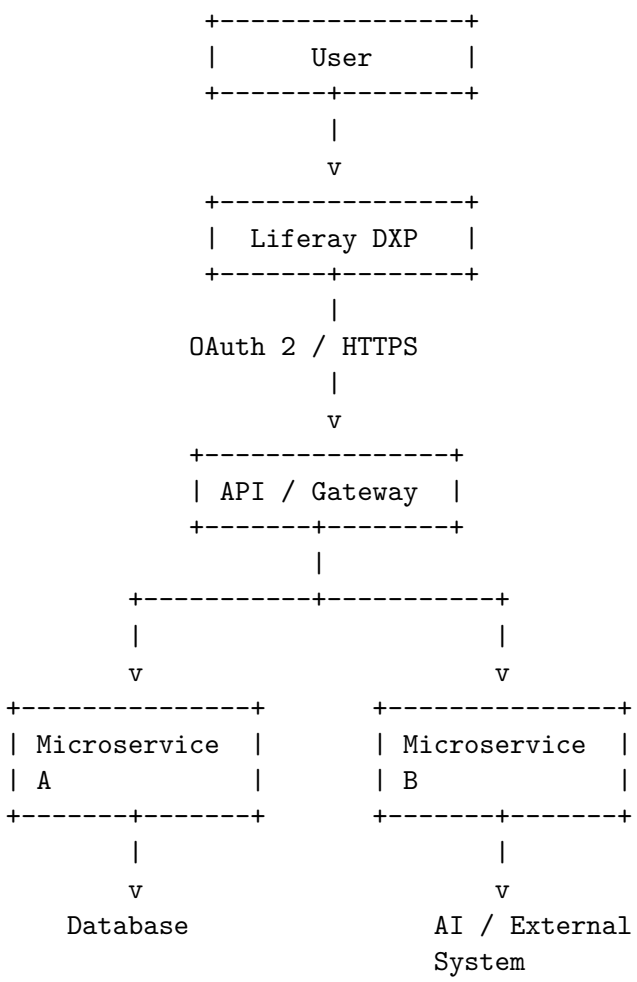
41.5 Timeout

Check:

- Service processing time
- Network latency
- Downstream systems
- Timeout configuration

42 Recommended Production Architecture

A robust enterprise architecture can be represented as:



Optional enterprise components can include:

- API gateway
- Message broker
- Centralized logging
- Metrics
- Distributed tracing
- Secret manager
- Kubernetes

43 Best Practices

For production deployments:

1. Keep the microservice stateless where practical.
2. Use HTTPS.

3. Use OAuth 2 for protected Liferay APIs.
4. Keep secrets outside source code.
5. Apply least privilege.
6. Define explicit HTTP timeouts.
7. Use retries only where safe.
8. Use health checks.
9. Add correlation IDs.
10. Centralize logs.
11. Version APIs.
12. Validate input.
13. Design idempotent operations where practical.
14. Monitor latency and failures.
15. Separate development, test, and production configuration.

44 Interview Questions

What is a Microservice Client Extension?

It is an integration approach where external application or business logic runs outside the Liferay JVM and integrates with Liferay through supported APIs and extension mechanisms.

Why use it instead of an OSGi module?

It enables technology independence, independent deployment, independent scaling, and stronger runtime isolation for suitable use cases.

What is client-extension.yaml?

It is a configuration descriptor used to define client extension metadata and integration configuration.

How can a microservice call Liferay securely?

A common approach is OAuth 2 with an appropriate grant, such as client credentials for machine-to-machine communication, combined with HTTPS and appropriately scoped permissions.

Does the microservice run inside Liferay?

No. The defining architectural characteristic is that the microservice runs externally from the Liferay JVM.

45 Multiple Choice Questions

Question 1

Where does a Microservice Client Extension's application logic primarily execute?

A. Inside the browser only

B. Inside the Liferay JVM

C. Outside the Liferay JVM

D. Inside Elasticsearch

Answer: C

Question 2

Which file commonly describes a Liferay client extension?

A. portal-ext.properties

B. client-extension.yaml

C. service.xml

D. web.xml

Answer: B

Question 3

Which technology can be used to implement an external microservice?

A. Spring Boot

B. FastAPI

C. Node.js

D. All of the above

Answer: D

Question 4

Which OAuth flow is commonly suitable for machine-to-machine communication?

A. Client Credentials

B. Interactive password entry on every API call

C. JSP session flow

D. None

Answer: A

Question 5

Which header commonly carries an OAuth access token?

A. Content-Length

B. Host

C. Authorization

D. Accept-Language

Answer: C

Question 6

What is a major benefit of external microservices?

A. They require every service to use Java

B. They can scale independently

C. They eliminate network communication

D. They remove all security requirements

Answer: B

Question 7

Where should production client secrets normally be stored?

A. Public GitHub repository

B. JavaScript source

C. Environment/secret management system

D. HTML page

Answer: C

Question 8

What does HTTP 401 generally indicate?

A. Authentication problem

B. Successful request

C. Database backup

D. Client extension build success

Answer: A

Question 9

What does HTTP 403 generally indicate?

A. Authorization/permission problem

B. DNS success

C. Docker build success

D. Liferay startup completion

Answer: A

Question 10

Why are correlation IDs useful?

A. They increase CPU speed

B. They trace a request across distributed services

C. They replace OAuth

D. They replace TLS

Answer: B

Question 11

Which is generally preferable for a computationally intensive Python AI model?

A. Embedding Python dependencies directly in the Liferay JVM

B. External Python microservice

C. JSP scriptlet

D. Theme CSS

Answer: B

Question 12

Which component provides REST APIs for external services to interact with Liferay resources?

A. Liferay Headless APIs

B. CSS Loader

C. JSP Compiler

D. Theme Builder

Answer: A

46 Quick Revision

Concept	Meaning
Client Extension	External extension/integration mechanism
Microservice	Independently running application service
client-extension.yaml	Client extension configuration descriptor
OAuth 2	Authorization framework for secured API access
Client Credentials	Machine-to-machine OAuth grant
Headless API	REST API exposed by Liferay
Docker	Container packaging/runtime technology
Kubernetes	Container orchestration platform
Health Check	Endpoint indicating service health
Correlation ID	Identifier for tracing distributed requests
Circuit Breaker	Resilience pattern for failing services

47 Key Takeaway

The fundamental architecture is:

$$\boxed{\text{Liferay DXP} \longleftrightarrow \text{Microservice Client Extension} \longleftrightarrow \text{External Business Logic}}$$

Instead of placing every customization inside the Liferay JVM:

$$\text{Liferay JVM} + \text{All Custom Logic}$$

the system can be decomposed into:

$$\boxed{\text{Liferay} = \text{Portal} + \text{Identity} + \text{Workflow} + \text{Governance}}$$

and:

$$\boxed{\text{Microservices} = \text{Specialized Business/Compute Services}}$$

This architecture is particularly useful for applications involving AI/ML, quantum computing, data processing, external enterprise systems, or independently scalable backend services.

The production design should emphasize:

$$\boxed{\text{API Contract} + \text{OAuth 2} + \text{TLS} + \text{Secret Management} + \text{Observability} + \text{Resilience}}$$

rather than treating a microservice as simply another HTTP endpoint.

48 Official Reference

For current, version-specific configuration and deployment details, refer to the official Liferay documentation:

https://learn.liferay.com/w/dxp/development/integrating-microservices/using-a-microservice-client-extension

The exact `client-extension.yaml` schema, supported properties, OAuth configuration, and deployment process should always be verified against the Liferay DXP release used by the target environment.