Optimizing your monolithic application by extracting modules into separate JARs and publishing them to JFrog Artifactory is a great strategy for better modularity, reusability, and easier dependency management.
Here's a step-by-step guide using Maven and Gradle, the most common build tools for Java applications.
Prerequisites:

1. **JFrog Artifactory Instance:** You need a running JFrog Artifactory instance with a local Maven or Gradle repository configured (e.g., libs-release-local for releases, libs-snapshot-local for snapshots).
2. **User Credentials:** An Artifactory user with deploy/write permissions to the target repository.
3. **Build Tool:** Maven or Gradle installed and configured on your local machine.

Part 1: Publishing Modules as JARs to JFrog Artifactory

A. For Maven Projects

Let's assume you have a module you want to extract, e.g., my-shared-module.
Step 1: Isolate the Module into its own Maven Project
If my-shared-module is currently a subdirectory or sub-module within your monolithic project, it's best to move it to a separate directory, making it an independent Maven project.
Step 2: Configure pom.xml for my-shared-module
Navigate to the root of my-shared-module. Open its pom.xml file.

- **Define GroupId, ArtifactId, Version:**
```
<groupId>com.yourcompany.shared</groupId>
<artifactId>my-shared-module</artifactId>
<version>1.0.0-SNAPSHOT</version> <packaging>jar</packaging>
```

- **Add Distribution Management:** This section tells Maven where to deploy the artifact.
```
<distributionManagement>
    <repository>
        <id>jfrog-releases</id>
        <name>JFrog Releases Repository</name>

<url>http://your-artifactory-url:8081/artifactory/libs-release-loc
al</url>
    </repository>
    <snapshotRepository>
        <id>jfrog-snapshots</id>
        <name>JFrog Snapshots Repository</name>

<url>http://your-artifactory-url:8081/artifactory/libs-snapshot-lo
cal</url>
    </snapshotRepository>
</distributionManagement>
```

- **Replace:** http://your-artifactory-url:8081 with your actual Artifactory URL.
- **libs-release-local / libs-snapshot-local:** These are common default local repository keys in Artifactory. Use the appropriate one for your artifact type (release or snapshot).

Step 3: Configure Maven settings.xml for Artifactory Authentication

You need to tell Maven how to authenticate with your Artifactory instance. Open your Maven settings.xml file (usually located at ~/.m2/settings.xml).

- **Add Server Entry:**

```
<servers>
    <server>
        <id>jfrog-releases</id>
        <username>your-artifactory-username</username>
        <password>your-artifactory-password</password>
    </server>
    <server>
        <id>jfrog-snapshots</id>
        <username>your-artifactory-username</username>
        <password>your-artifactory-password</password>
    </server>
</servers>
```

- **Important:** The <id> in settings.xml **must match** the <id> in the distributionManagement section of your pom.xml.
- **Security:** For better security, consider using encrypted passwords in settings.xml or environment variables for credentials in CI/CD pipelines.

Step 4: Build and Deploy the Module

Navigate to the my-shared-module project directory in your terminal and run:

```
mvn clean deploy
```

This command will:

1. Clean the project.
2. Compile, test, and package your module into a JAR file.
3. Deploy the JAR (and its pom.xml) to the configured Artifactory repository.

You should see output indicating a successful deployment to your Artifactory instance. You can then log into your Artifactory UI and verify that the my-shared-module JAR is present.

B. For Gradle Projects

Let's assume you have a module you want to extract, e.g., my-shared-module.

Step 1: Isolate the Module into its own Gradle Project

Similar to Maven, move my-shared-module to a separate directory, making it an independent Gradle project.

Step 2: Configure build.gradle for my-shared-module

Navigate to the root of my-shared-module. Open its build.gradle file.

- **Apply Plugins:**

```
plugins {
    id 'java' // For Java projects
```

```
    id 'maven-publish' // For publishing to Maven repositories
    id 'com.jfrog.artifactory' version '5.+' // Or the latest
version of the Artifactory plugin
}
```

- **Define Group, Version:**

```
group 'com.yourcompany.shared'
version '1.0.0-SNAPSHOT' // Use -SNAPSHOT for development, remove
for release
```

- **Configure Publishing:**

```
publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
}
```

- **Configure Artifactory Deployment:**

```
artifactory {
    contextUrl = "${artifactory_contextUrl}" // Define this in
gradle.properties or environment variables
    publish {
        repository {
            repoKey = 'libs-release-local' // For releases
            // Or 'libs-snapshot-local' for snapshots if version
ends with -SNAPSHOT
            // repoKey = 'libs-snapshot-local'
            username = "${artifactory_user}" // Define this in
gradle.properties or environment variables
            password = "${artifactory_password}" // Define this in
gradle.properties or environment variables
            maven = true
        }
        defaults {
            publications('mavenJava')
            // Optional: Capture build info for better
traceability
            // publishBuildInfo = true
        }
    }
}
```

Step 3: Define Artifactory Credentials in gradle.properties (or Environment Variables)
Create a gradle.properties file in your Gradle user home directory (e.g., ~/.gradle/gradle.properties) or within your project for development (but **never commit sensitive info**).

```
artifactory_contextUrl=http://your-artifactory-url:8081/artifactory
artifactory_user=your-artifactory-username
artifactory_password=your-artifactory-password
```

Step 4: Build and Deploy the Module
Navigate to the my-shared-module project directory in your terminal and run:
```
./gradlew clean artifactoryPublish
```

This command will:
1. Clean the project.
2. Compile, test, and package your module into a JAR file.
3. Deploy the JAR (and its pom.xml / module descriptor) to the configured Artifactory repository.

You should see output indicating a successful deployment to your Artifactory instance. Verify its presence in Artifactory UI.

Part 2: Importing Published JARs in Your Main Module (Monolith)

Now that your shared module is published, you can import it as a dependency in your main monolithic application.
Important: For your main module to resolve dependencies from Artifactory, you need to configure its build tool to look for artifacts in your Artifactory repositories. It's often recommended to use a **Virtual Repository** in Artifactory that aggregates your local and remote repositories. This provides a single endpoint for all your dependencies.

A. For Maven Projects

Open the pom.xml of your main monolithic application.
Step 1: Add Artifactory Repository to pom.xml (or settings.xml)
While you can add the repository directly to your pom.xml, it's generally a better practice to configure it in your settings.xml to avoid hardcoding repository URLs in every project.
Option 1: Add to pom.xml (less preferred for enterprise)
```
<repositories>
    <repository>
        <id>your-artifactory-virtual-repo</id>
        <name>Your Artifactory Virtual Repository</name>

<url>http://your-artifactory-url:8081/artifactory/your-virtual-reposit
ory</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
```

```
</repositories>
```

- **Replace:** http://your-artifactory-url:8081/artifactory/your-virtual-repository with the URL of your Artifactory Virtual Repository (e.g., http://your-artifactory-url:8081/artifactory/maven-public).

Option 2: Add to settings.xml (recommended)

In your settings.xml file, add a <profile> section:

```
<profiles>
    <profile>
        <id>artifactory</id>
        <repositories>
            <repository>
                <id>your-artifactory-virtual-repo</id>
                <name>Your Artifactory Virtual Repository</name>

<url>http://your-artifactory-url:8081/artifactory/your-virtual-reposit
ory</url>
                <releases>
                    <enabled>true</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
            </repository>
        </repositories>
        <pluginRepositories>
            <pluginRepository>
                <id>your-artifactory-virtual-repo</id>
                <name>Your Artifactory Virtual Repository</name>

<url>http://your-artifactory-url:8081/artifactory/your-virtual-reposit
ory</url>
                <releases>
                    <enabled>true</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
            </pluginRepository>
        </pluginRepositories>
    </profile>
</profiles>

<activeProfiles>
    <activeProfile>artifactory</activeProfile>
</activeProfiles>
```

This activates the "artifactory" profile by default, so Maven will always look in your Artifactory

virtual repository.

Step 2: Add Dependency in pom.xml of Main Module

In the pom.xml of your main monolithic application, add the dependency for my-shared-module:

```
<dependencies>
    <dependency>
        <groupId>com.yourcompany.shared</groupId>
        <artifactId>my-shared-module</artifactId>
        <version>1.0.0-SNAPSHOT</version> </dependency>
    </dependencies>
```

Step 3: Rebuild Your Main Module

Navigate to your main module's directory and run:

```
mvn clean install
```

Maven will now resolve my-shared-module from your JFrog Artifactory instance.

B. For Gradle Projects

Open the build.gradle of your main monolithic application.

Step 1: Configure Artifactory Repository

In your build.gradle file, configure the repositories section to include your Artifactory instance. It's recommended to use a virtual repository.

```
repositories {
    mavenLocal() // Optional: if you also use local Maven repo
    maven {
        url
"http://your-artifactory-url:8081/artifactory/your-virtual-repository"
// Replace with your Artifactory Virtual Repo URL
        credentials {
            username = "${artifactory_user}"
            password = "${artifactory_password}"
        }
    }
    // Add other repositories like mavenCentral() if needed
    mavenCentral()
}
```

- **Replace:** http://your-artifactory-url:8081/artifactory/your-virtual-repository with the URL of your Artifactory Virtual Repository (e.g., http://your-artifactory-url:8081/artifactory/gradle-public).
- **Credentials:** Ensure artifactory_user and artifactory_password are defined in gradle.properties or as environment variables as shown in Part 1B, Step 3.

Step 2: Add Dependency in build.gradle of Main Module

In the dependencies block of your main monolithic application's build.gradle:

```
dependencies {
    implementation
'com.yourcompany.shared:my-shared-module:1.0.0-SNAPSHOT' // Or 1.0.0
for release
```

```
    // Other dependencies
}
```

- Use implementation for compile-time and runtime dependencies, or api if your module exposes the dependency as part of its public API.

Step 3: Rebuild Your Main Module

Navigate to your main module's directory and run:

```
./gradlew clean build
```

Gradle will now resolve my-shared-module from your JFrog Artifactory instance.

Key Considerations and Best Practices:

- **Version Management:**
  - **Snapshots (-SNAPSHOT):** Use snapshots during development for modules that are still undergoing active changes. This allows you to rapidly iterate without needing to increment the version number for every small change. Artifactory automatically handles snapshot redeployments.
  - **Releases:** Once a module is stable, remove -SNAPSHOT from its version and deploy it as a release. Release artifacts are immutable in Artifactory.
  - **Semantic Versioning:** Follow semantic versioning (Major.Minor.Patch) for your modules (e.g., 1.0.0).
- **Dependency Collisions:** Be mindful of potential dependency collisions if your extracted modules bring in new versions of libraries already used in your monolith. Use dependency management features (e.g., Maven's <dependencyManagement> or Gradle's platform() for BOMs) to control versions.
- **CI/CD Integration:** Automate the publishing of your extracted modules as part of your CI/CD pipeline. Tools like Jenkins, GitLab CI, or GitHub Actions can be configured to build and deploy to Artifactory upon successful commits to your module's repository.
- **Artifactory Virtual Repositories:** Always use Virtual Repositories for consuming dependencies. They provide a single URL to resolve artifacts from multiple underlying local and remote repositories, simplifying configuration and offering better performance (caching) and control.
- **Security:**
  - Avoid hardcoding passwords in pom.xml or build.gradle. Use settings.xml (Maven) or gradle.properties (Gradle) for local development.
  - For CI/CD, use environment variables or secure credential management systems.
  - Consider using Artifactory's API keys or access tokens for more granular and secure access.
- **Build Info (JFrog Artifactory Plugin for Gradle/Maven):** The JFrog Artifactory plugins for Maven and Gradle automatically capture comprehensive build information (dependencies, environment variables, etc.) and publish it to Artifactory. This is invaluable for traceability, debugging, and security scanning with JFrog Xray.
- **Breaking Changes:** When extracting modules, carefully consider the interfaces and APIs you expose. Any breaking changes in a published JAR will affect all consumers of that JAR.

By following these steps, you can effectively modularize your monolithic application, making it more manageable and scalable.