Could not load class : java.lang.NoClassDefFoundError

Local SDK · https://namazustudios.com/docs/troubleshooting-guide/local-sdk/could-not-load-class-java-lang-noclassdeffounderror/

When using the Local SDK, you will often times run into errors related to the ClassLaoder. A log will typically look like this. This article outlines a few causes and their solutions to this process.

```
java.lang.IllegalArgumentException: Could not load class
dev.getelements.robloxkit.service.StandardRobloxAuthService : java.lang.NoClassDefFoundError:
dev/getelements/robloxkit/RobloxAuthService
	at io.github.classgraph.ScanResult.loadClass(ScanResult.java:1459)
	at io.github.classgraph.ScanResultObject.loadClass(ScanResultObject.java:228)
	at io.github.classgraph.ScanResultObject.loadClass(ScanResultObject.java:252)
	at io.github.classgraph.FieldInfo.loadClassAndGetField(FieldInfo.java:260)
	at java.base/
```

Code is missing the @ElementPublic annotation.

Any class, interface, or package that is exposed as a service (via the `@ElementServiceExport` annotation) must also be made public. There's a few ways to do this.

Ensure All Related Types Are Public

The requirement to be public requires to the interface type itself as well as any types referenced within the interface.

Ensure that the package is annotated with ElementPublic

```
@ElementPublic
package com.example.mystudio.mygame;
```

```
import dev.getelements.elements.sdk.annotation.ElementPublic;
```

Ensure that the type is annotated with ElementPublic

```
package com.example.mystudio.mygame;

import dev.getelements.elements.sdk.annotation.ElementPublic;

@ElementPublic
public interface MyInterface {

 void foo();

}
```

A Required Dependency is Missing

Each Element must supply its own dependencies. This means that the Element must have included that dependency on the Classpath. Our example projects automate this process using the Maven Copy Dependencies Plugin and the local SDK expects those dependencies to be provided in `target/element-libs` relative to your project directory. The local SDK Maven runner will attempt to run Maven just before your code runs, but this may not always work correctly.

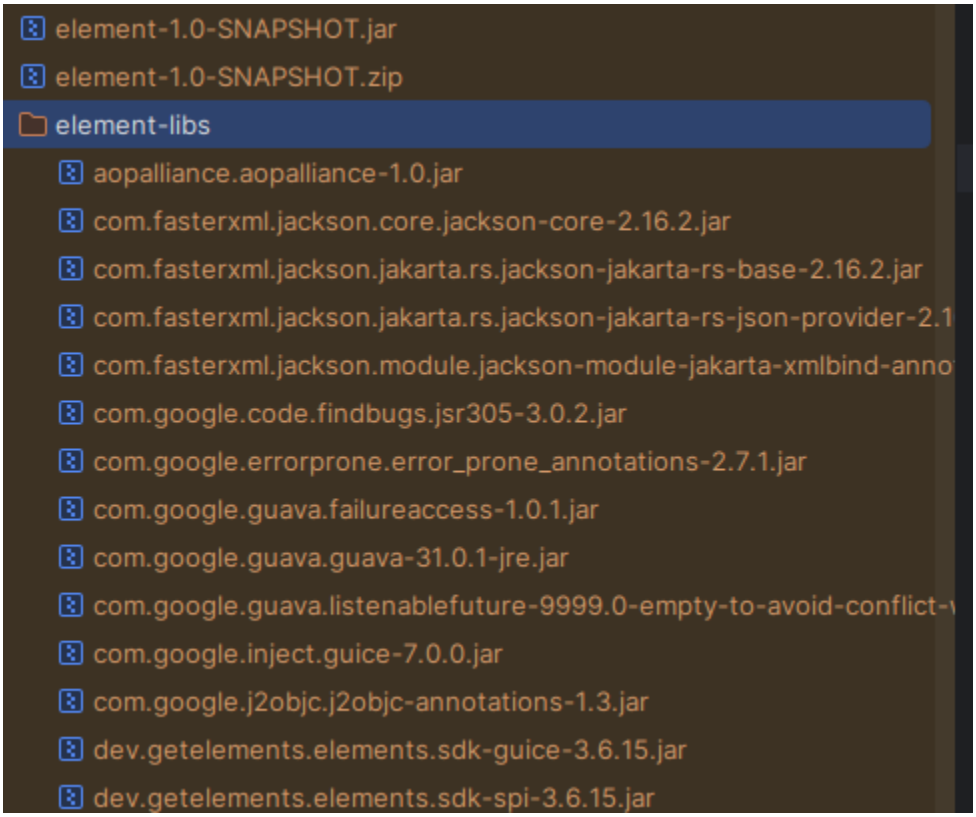
Check your `pom.xml`

The pom should have a step to copy those dependencies in. The snippet in our example code ensures that only third-party code is included in that directory.

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <version>3.6.0</version>
                <executions>
                    <execution>
                        <id>copy-element-deps</id>
                        <phase>generate-resources</phase>
                        <goals>
```

```
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <prependGroupId>true</prependGroupId>
                            <excludeScope>provided</excludeScope>
                            <excludeGroupIds>ch.qos.logback</excludeGroupIds>
                            <excludeArtifactIds>sdk-local,sdk-local-maven,sdk-
logback</excludeArtifactIds>
                            <outputDirectory>${project.build.directory}/element-
libs</outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
```

Typically, everything you need should appear in your IDE:



Jars need Installed

Because we use Maven to organize the local SDK, it will always copy from the jars installed in your Local Maven Repository. In short a Maven Repository is a directory containing compiled Java code in

an organized directory structure complete with version tagging. Just before your code runs, the LocalSDK copies pre-built code from that directory and copies it to your project. If you are running a project for the first time, you may have to manually build and install the code.

To do this quickly, execute the following command in the root of your project:

```
mvn -DskipTests clean install
```

This tells Maven to clean, build, and install all jars in your project to your local Maven Repository. The `-DskipTests` ensures that integration and unit tests do not run, saving time and allowing you to debug code that may not be currently passing tests.

Special Cases with OAS3 and MongoDB Drivers

The Server does everything it can to avoid forcing its opinions on the Elements. In doing so there are some oddities and nuance required with certain situations. Specifically when dealing with the following libraries:

- MongoDB Client Code
- OpenAPI Annotations

To make some features work, we allow your code to inherit those opinions when necessary. We cover these cases here. In general, when using a type from the server, this must be done using the `@ElementTypeRequest` (3.7 and up) annotation. Additionally, you must ensure that the classpath entry is set to `provided`. This way the class does not appear in multiple places.

Future Proofing

This is not an exhaustive list. We do our best to keep this up to date. If you encounter these issues, please put them in the feedback section of this page, email us, or reach out in Discord so we can update.

Opinion: Using the Server's MongoClient, MongoDatabase, or OpenAPI

```
<dependency>
    <groupId>org.mongodb</groupId>
```

```
    <artifactId>mongodb-driver-core</artifactId>
    <scope>provided</provided>
</dependency>
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <scope>provided</provided>
</dependency>
```

When you want to use the actual share connection with the database, you are inheriting the `MongoClient` from the Elements server core. To do so you must ensure that your pom does not include the jar on the classpath of the Element. You must ensure that it is included as a provided dependency.

```
@ElementPublic
@ElementTypeRequest("com.mongodb.client.MongoClient")
package com.mystudio.mygame;

import dev.getelements.elements.sdk.annotation.ElementTypeRequest;

import dev.getelements.elements.sdk.annotation.ElementPublic;
```

These combined indicate htat because the scope is `provided` the jars do not get packed in the Element and the `ElementTypeRequest` indicates that they will be made available to your Element. **In most cases, this is what you want**.

Opinion: Providing your Own Types

If you wish to use your own types, you may do so by including the drivers in the compiled scope. This ensures you are able to include the code in the Element. If you do not set the scope, then these will not appears in the resulting bundle.

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-core</artifactId>
    <scope>compile</provided>
</dependency>
<dependency>
    <groupId>org.mongodb</groupId>
```

```
    <artifactId>mongodb-driver-sync</artifactId>
    <scope>compile</provided>
</dependency>
```

Sharing Types

Do not add these types to your Element if you wish to return or hand the type with the server. You will likely get an instance of `ClassCastExcpption` for the same types. eg `Cannoot cast com.mongodb.client.MongoDatabase to com.mongodb.client.MongoDatabase`.

In most cases you do not want this, unless you have a particularly advanced use case.