Quarkus with GraalVM native images operates under a "closed-world assumption," which significantly impacts how reflection is handled. This assumption means that all code and resources required by the application must be known at build time.

Here's why this leads to challenges with reflection:

* **Static Analysis Limitations:**

GraalVM's native-image tool performs static analysis to determine which parts of the Java application's bytecode are actually used and should be included in the native executable. Reflection, by its nature, allows for dynamic access to classes, methods, and fields at runtime. This dynamic behavior makes it difficult for static analysis to predict which elements will be accessed via reflection.

* **Dead Code Elimination:**

If the native-image tool cannot statically determine that a class, method, or field will be used, it may be considered "dead code" and removed from the native executable to reduce its size. This can lead to ClassNotFoundException or NoSuchMethodException at runtime if reflection attempts to access an element that was removed.

* **Need for Explicit Registration:**

To address this, any elements intended for reflective access in a GraalVM native image must be explicitly registered for reflection. This registration provides the native-image tool with the necessary information to include these elements in the final executable, even if they are only accessed dynamically.

Quarkus leverages this understanding by providing mechanisms to handle reflection registration, often through build-time processing and annotations, to ensure that necessary components are included in the native image while still benefiting from the performance and resource advantages of GraalVM. However, it requires a different approach than traditional Java applications that rely heavily on unconstrained reflection.