Quarkus - Practice Sheet

10 Q&A on native image, CDI, Panache, reactive, and testing

Q1. What is Quarkus and why is it designed for cloud-native environments?

A: Quarkus is a Kubernetes-native Java framework that moves as much work as possible to build time. This produces extremely fast startup (< 50 ms native) and low memory footprint, which directly reduces cloud compute costs and enables efficient scale-to-zero.

Q2. How does Quarkus native compilation with GraalVM work?

A: Quarkus performs AOT analysis at build time using GraalVM native-image. It traces reachable classes/methods, eliminates dead code, and compiles to a platform-specific binary. Reflection and dynamic proxies must be declared via metadata for native builds.

Q3. How does Quarkus's build-time CDI differ from standard runtime CDI?

A: Quarkus resolves CDI injection points, validates beans, and generates synthetic bytecode at build time using the Jandex index. This eliminates the runtime reflection that standard CDI requires, reducing startup time and memory significantly.

Q4. How do you build a REST endpoint in Quarkus with RESTEasy Reactive?

A: Annotate a class with @Path('/items') and methods with @GET, @POST using standard JAX-RS annotations. RESTEasy Reactive processes requests on the event loop without blocking; return Uni<T> from SmallRye Mutiny for true reactive endpoints.

Q5. What are Quarkus Dev Services?

A: Dev Services automatically start required infrastructure (PostgreSQL, Kafka, Redis) as Testcontainers when running in dev or test mode and no explicit connection config is found. This eliminates the need for local service setup during development.

Q6. What are Uni and Multi types in SmallRye Mutiny?

A: Uni<T> represents a single asynchronous result (0 or 1 item), similar to a promise. Multi<T> is a reactive stream of multiple items. Both are lazy they start processing only when subscribed. Quarkus RESTEasy Reactive handles subscription automatically.

Q7. What is Quarkus Panache and how does it simplify JPA?

A: Panache provides two patterns: Active Record (entities extend PanacheEntity and have static query methods like User.find('email', email)) and Repository (inject a PanacheRepository). Both reduce boilerplate compared to raw JPA repositories.

Q8. How do you consume Kafka messages in Quarkus?

A: Add quarkus-smallrye-reactive-messaging-kafka. Annotate a method with @Incoming('topic-name') and define kafka channel configuration in application.properties. The framework handles consumer lifecycle, deserialization, and back-pressure automatically.

Q9. What happens during Quarkus's augmentation (build) phase?

A: Augmentation runs annotation processors, resolves CDI, validates configuration, generates synthetic classes, and produces a pre-built bytecode store. At runtime, the app loads pre-computed data instead of re-analyzing classpath, giving near-instant startup.

Q10. How do you test a Quarkus application?

A: @QuarkusTest starts the application on a random port for the test. Inject a REST-assured spec to call endpoints. @QuarkusIntegrationTest tests the packaged artifact (JVM JAR or native binary). Use @InjectMock to replace CDI beans with Mockito mocks.