## **Summary of Attempts to Integrate a Custom CapabilityStatement Provider**

**Goal:** To integrate a custom Java class, extending ca.uhn.fhir.rest.server.provider.ServerCapabilityStatementProvider, into a HAPI FHIR JPA server (WAR deployment on Tomcat 10.1) to dynamically modify the server's /metadata (CapabilityStatement) output. The custom provider needs to read a whitelist from a local SQLite database and filter the listed resources in the CapabilityStatement. A mechanism to enable/disable this custom functionality via configuration was also desired.

**Environment:**

* HAPI FHIR Version: 8.0.0 (JPA Server Starter deployed as ROOT.war)
* Application Server: Apache Tomcat 10.1
* Java Version: 17
* Database for custom whitelist: SQLite (accessed via JDBC from the custom Java class)
* Main HAPI FHIR Database: H2 (configured in application.yaml)
* Custom Code: Packaged as a JAR (fhirflare-custom-1.0.0.jar) containing the custom provider and SQLite JDBC driver, placed in Tomcat's /usr/local/tomcat/lib/ directory.

Observed Problem:

The HAPI FHIR server fails to start when the custom Java provider is activated, consistently throwing an org.springframework.beans.factory.UnsatisfiedDependencyException because a bean of type javax.sql.DataSource cannot be found when required by ca.uhn.fhir.jpa.starter.common.StarterJpaConfig for the entityManagerFactory method.

Baseline Working State:

The HAPI FHIR server starts correctly only when the custom Java provider is not activated by any means (neither Spring component scanning nor HAPI's custom-provider-classes property). In this working state, the application.yaml relies on Spring Boot's default DataSourceAutoConfiguration (i.e., spring.autoconfigure.exclude for DataSourceAutoConfiguration is not present) to successfully create the DataSource bean for HAPI's JPA persistence.

**Integration Attempts and Outcomes:**

1. **Attempt 1: Spring @Bean Registration with HAPI Managing DataSource**
   * **Configuration:**
     + Custom Java class (ResourceWhitelistCapabilityStatementCustomizer) annotated with @Component.
     + A Spring @Configuration class (FhirConfig.java) defined to create the customizer as a @Bean, injecting RestfulServer.
     + application.yaml:
       - spring.context.component-scan.base-packages set to scan the package containing FhirConfig.java.
       - spring.autoconfigure.exclude: org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration was active (intending for HAPI's starter to configure the DataSource).
       - hapi.fhir.custom-provider-classes was *not* used.
   * **Outcome:** Server startup failed with "DataSource not found". This indicated that HAPI's internal DataSource configuration was not providing the bean when the custom Spring configuration was active.
2. **Attempt 2: Spring @Bean Registration with Explicit DataSource Bean**
   * **Configuration:**
     + Same Java setup as Attempt 1 (@Component on customizer, FhirConfig.java with @Bean for customizer).
     + FhirConfig.java was modified to also explicitly define a @Bean @Primary public DataSource dataSource() using DataSourceBuilder and @ConfigurationProperties(prefix = "spring.datasource").
     + pom.xml: Added spring-boot-starter-jdbc and com.h2database:h2 (runtime) dependencies.
     + application.yaml:
       - spring.context.component-scan.base-packages remained active.
       - spring.autoconfigure.exclude: ...DataSourceAutoConfiguration was **removed/commented out** (to allow our explicit @Primary DataSource bean to be the primary candidate).
       - spring.datasource.\* properties correctly defined.
   * **Outcome:** Server startup still failed with "DataSource not found". This suggested that even with an explicit @Primary DataSource bean, its creation or registration was not happening in the correct order relative to HAPI's JPA components needing it, particularly when the customizer bean was also being managed by Spring.
3. **Attempt 3: HAPI's custom-provider-classes Property (HAPI Manages DataSource)**
   * **Configuration:**
     + ResourceWhitelistCapabilityStatementCustomizer.java: All Spring annotations (@Component, @Value) removed. A constructor public ResourceWhitelistCapabilityStatementCustomizer(RestfulServer server) was used. An environment variable (FHIRFLARE\_CUSTOM\_WHITELIST\_ENABLED) was used to toggle functionality.
     + FhirConfig.java: Removed entirely from the custom JAR.
     + application.yaml:
       - hapi.fhir.custom-provider-classes: com.fhirflare.custom.ResourceWhitelistCapabilityStatementCustomizer was added/uncommented.
       - spring.context.component-scan was *not* used for the custom package.
       - spring.autoconfigure.exclude: ...DataSourceAutoConfiguration was active (relying on HAPI for DataSource).
   * **Outcome:** Server startup still failed with "DataSource not found".
4. **Attempt 4: HAPI's custom-provider-classes with Default Spring Boot DataSource**
   * **Configuration:**
     + Java setup same as Attempt 3 (customizer reads env var, no FhirConfig.java).
     + application.yaml:
       - hapi.fhir.custom-provider-classes remained active.
       - spring.autoconfigure.exclude: ...DataSourceAutoConfiguration was **removed/commented out** (to allow Spring Boot default DataSource creation).
   * **Outcome:** Server startup still failed with "DataSource not found".

**Summary of the Core Issue for a Bug Ticket:**

When attempting to integrate a custom Java class extending ca.uhn.fhir.rest.server.provider.ServerCapabilityStatementProvider into a HAPI FHIR JPA server (v8.0.0, WAR deployment on Tomcat 10.1), the server fails to start due to an UnsatisfiedDependencyException for javax.sql.DataSource. This occurs when the custom provider is activated, either through Spring @Bean registration (via @Configuration and component scanning) or through HAPI FHIR's hapi.fhir.custom-provider-classes property.

The server *only* starts successfully if:

1. Spring Boot's default DataSourceAutoConfiguration is allowed to run (i.e., it's *not* excluded in application.yaml).
2. The custom ServerCapabilityStatementProvider is *not* activated by any mechanism.

This suggests that the introduction of the custom HAPI provider, regardless of its registration method, triggers an early initialization of the JPA persistence stack. This occurs before the DataSource bean (whether it's supposed to be provided by HAPI's internal configuration or by Spring Boot's default auto-configuration) is fully created and available in the application context. Consequently, the ca.uhn.fhir.jpa.starter.common.StarterJpaConfig class cannot obtain the required DataSource to create the entityManagerFactory, leading to the startup failure.

The issue appears to be a conflict in the bean/component initialization lifecycle or classpath visibility when custom HAPI providers are added to a WAR-deployed HAPI FHIR JPA server on Tomcat. Explicitly defining a @Primary DataSource bean via Spring @Configuration also did not resolve the issue when the custom provider bean was simultaneously active.