Where to go with CDI

By Tomas Langer and Graeme Rocher

This document summarizes problems we see with current CDI specification and outlines options for the future version of the specification.

Definition of the problem

CDI has served us well in Java & Jakarta EE, yet the innovations in Java space have shown us a few areas where CDI is lacking.

The following sections describe limitations of CDI and explanation of impact. Most of the problems are caused by the specification itself, as these are required. Some of the problems are specific to implementations (though sometimes this may be the only way to implement the spec as is).

Used abbreviations/terms:
AOT: Ahead of time compilation into native executable (such as when using GraalVM native image)
Build time injection: Analyzing the code at build time (using source code or byte code) to generate metainformation and tools for runtime processing
Spec: short for “Specification”, see
https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.pdf

Classpath Scanning

CDI uses runtime classpath scanning to discover bean archives (and beans).
Why:

1. The spec defines the concept of a bean archive, that is marked by a presence of META-INF/beans.xml file and its bean-discovery-mode (Spec: 12.1. Bean archives), the spec also explicitly mentions war, EJB, jar, or rar bean archives
2. The spec defines that types must be discovered in type discovery step from bean archives (Spec 12.4.1. Type discovery); type discovery is mandatory part of application initialization lifecycle

This has the following impact:

1. Performance: scanning jar files on file system (or even within an .ear file) can take a lot of time and CPU resources
2. Classpath assumptions: all classpath is assumed to be URI based (and implementations usually restrict this to be file base) - this approach may be insufficient for unusual classloaders (as we see problems with separated classpath for classes and resources), and for AOT (where classpath is structured differently and not based on external resources at all)
3. We need to have a runtime dependency on a tool capable of scanning bytecode of classes (unless we want to just load every single class on classpath) to determine annotations

Usage of Reflection

CDI uses runtime reflection to analyze classes, methods and fields; and to inject values into them.
Why:

1. The spec requires an AnnotatedType is created for each discovered type. As annotated type contains information that can only be obtained at runtime through reflection, it implies usage of reflection (Spec 12.4.1. Type discovery).
2. Portable extensions (Spec 11. Portable extensions) imply usage of reflection, as they allow the developer to:

    a. Add beans, interceptors and decorators
    b. Inject dependencies
    c. Provide custom scope
    d. Update metadata of annotated types

This has the following impact:

1. Performance: handling reflection requires more CPU and memory than direct access
2. AOT: reflection points need to be configured explicitly and can be easily missed
3. Larger stack traces that are more difficult for the JIT to inline than direct calls
4. Security: It is less safe to allow arbitrary reflective access on user supplied classes than it is to only a limited subset of code declared in the source code to be accessed reflectively. There have been many security vulnerabilities in reflective libraries that could be mitigated by abandoning the use of reflection and ensuring only classes instrumented at build time can be accessed.

Usage of Proxies

CDI uses runtime proxies to support injection into different scopes and to support AOT (such as in interceptors).

Why:
1. Portable extensions can add interceptors and beans at runtime, implying runtime generation of proxies
2. CDI prescribes usage of proxies (Spec 5.4. Client proxies), to allow injection of beans with different scope (such as RequestScoped beans into ApplicationScoped bean); there is no definition when such a proxy is generated

This has the following impact:

1. Performance: handling of proxies is usually fully reflection based - see above “Usage of Reflection)
2. AOT: proxies must configured explicitly to be supported
3. Proxies for interfaces can use the JDK proxy features, but for classes often require the use of runtime bytecode generation using libraries like CGLib or Bytebuddy

Extensibility

CDI extensions allow full runtime modification to beans - adding annotations, or even creating brand new synthetic beans at CDI container initialization time.

This has the following impact:

1. Performance: all of the information needs to be processed at runtime and all bean definition infrastructure must be present at runtime including the ability to generate proxies (potentially via runtime bytecode generation)
2. Difficult / Impossible to support AOT as bean registration and definition has to be done at runtime

See also https://github.com/eclipse-ee4j/cdi/pull/451 from Red Hat which tries to address this design concern in CDI extensions as they are today (although with a clear Quarkus focus).

Runtime bytecode Generation

CDI implementations usually generate code at runtime. This is usually related to the requirement to proxy (see Usage of Proxies) and to generate synthetic beans at runtime. This is a requirement for beans introduced by portable extensions.

This has the following impact:

1. Performance: classes need to be generated at runtime, slowing down startup, also increasing memory usage in most cases
2. Security: generating new classes must follow rules for module system, making it complicated and error prone
3. AOT: this is unsupported as the code is not compiled into native code at all

Usage of Deprecated/Unsupported JDK features

CDI implementations often fall back to the use of **Unsafe**, to be able to implement some of the features in CDI (such as injection into final fields). This is not prescribed by the spec, but it is a usual approach taken by implementations.

Why:
1. The specification allows injection into private fields
2. The specification allows usage of private methods as producers (Spec 3.2. Producer methods)
3. The specification allows usage of private fields as producers (Spec 3.3. Producer fields)
4. The specification allows usage of private constructors to instantiate a bean (Spec 3.5. Bean constructors)

This has the following impact:

1. Compatibility: implementations need to be updated to latest JDK versions, sometimes generating warnings because of unsupported features used
2. AOT: this may be supported, though requires detailed understanding of the use cases

Base specification for other specification

There seems to be an agreement in the Jakarta EE community, that CDI should be a base specification for other specifications, such as for JAX-RS (similar features would be needed by other features, if we decided to have a GraphQL specification, gRPC specification etc.).
There are several areas that are not covered by the CDI specification, yet needed for downstream, such as:

- Support for the concept of "executable methods" - methods that must be executed by the framework on beans, that require parameter injection
- Support for the concept of injected parameters - CDI currently supports `@Inject`, yet that is only useful for injection into constructors or instance creating methods

To avoid confusion where each downstream specification defines handling of such features on its own, this should be covered in CDI and re-used.

The Proposal

The following describes the direction we think CDI should take. The code in examples is mostly conceptual, and needs to be verified through a POC if the community agrees on the direction.

The Goal

To mitigate or eliminate the problems described above, we would need to:

- Modify the specification making the usage of reflection optional and extending the API to allow reflection-free access to Bean metadata.
- Create a new build time extension model that allows mutating and synthesizing new beans at build time.
- Eliminate all aspects of the specification that require runtime classpath scanning including the requirement for beans.xml.
- Modify the specification to make the runtime portable extension model optional, allowing bean infrastructure to be mutable at build time but immutable at runtime.
- Modify the specification by reducing the scope of CDI and eliminating features that are considered out of scope (ConversionScope, Passivation etc.) for such a build time approach.
- Update the specification to serve as a stable base for other specifications

Backward compatibility

Given the existing design issues with CDI make CDI inappropriate for low-memory footprint and performance sensitive scenarios (AOT, Serverless, Low-memory footprint Microservices, CLI applications that require fast startup, Mobile applications, IoT Devices and so on) this proposal suggests creating a new profile for CDI (tentatively named "CDI lite") that adapts the CDI programming model to these scenarios by reducing the scope of the CDI specification and providing the needed extensibility (as described above)

Backward compatibility goal:

- The build-time extension model should provide API that would allow this model to be processed by runtime, allowing a CDI portable extension that could be a bridge between the build time extensions and portable extensions
- If such an option would not be feasible, Build-time extensions should not be required by CDI EE

Envisioned profiles in CDI:

1. “CDI Lite” - limited scope (no passivation support, runtime reflection support optional etc.), build-time extensions
2. SE - as is + support for build-time extensions
3. EE - as is + support for build-time extensions if feasible

Why not portable extensions?

The first question we asked ourselves was if this could be achieved with the current portable extension model.
Unfortunately it seems that this is not feasible.
The following sections describe the reasons.

Portability

One of the main reasons is the requirement for portability. This is one of the main overall features of Jakarta EE specification - the same application binary should be runnable on different application servers (or different SE runtimes).
As the build-time approach generates classes and resources, the runtime must be able to process such. As a result, the specification needs to define these resources and metadata, so an application compiled with one implementation can be executed by another.

Bootstrapping

The bootstrap process of CDI is defined by the events that are happening, such as “BeforeBeanDiscovery”. The specification implies that all of this is happening at runtime.
There is no place in the specification that would allow us to assume which part is “safe” to be done at build time, and which parts need to be executed at runtime - the bootstrap cannot be “split” into a build time part and runtime part, as portable extensions can keep state (such as threads, open sockets, file descriptors) that cannot be kept between build and run time.
Even if one vendor defined an approach to this (as Helidon has done for portable extensions, even though silently for AoT, and Quarkus has done with their extensions), there is no way to provide portability, **unless** CDI specifies the requirements.

Bean landscape

CDI expect the overall bean structure to be constructed at runtime. If I were to create a portable extension that is fully build time compatible, application may have an additional extension that is runtime only, that adds features to beans we did not anticipate (such as a new interceptor). If this would happen, the whole build-time processing would become useless, and we would need to rescan all the beans and create proxies at runtime, defeating the purpose of build time processing

Bean Metadata

Currently the metadata model is not complete and heavily depends on `java.lang.reflect` classes such as direct usage of `java.lang.reflect.Method` and `java.lang.reflect.Field`. The AnnotatedMember interface directly exposes reflective access via the `getJavaMember()` method.

To be able to fully support build time processing, we would need to fully abstract the model of

- Classes
- Methods
- Fields
- Annotations

We need to analyze whether we could update the existing metamodel, or we would be required to create a new package with a new model.
The meta model needs to provide information about each element (Class, Method, Field, Parameter, return type) in a reflection free way.
The model would need to provide information about annotations on an element that is decoupled from the use of reflection and JDK proxies and include computed annotation metadata defined as per the stereotype rules of the current CDI specification.

Such an abstraction should be compatible with build time generation through either bytecode processing or annotation processors.

Implementations of CDI lite should make this meta model available at runtime through the existing interfaces defined in `jakarta.enterprise.inject.spi`.

The runtime model should by default be immutable, unless an explicit dependency is defined by the user to add support for runtime portable extensions in which case additional features are added to the container environment to support the creation and definition of beans at runtime.

Extensibility

The current extension mechanism of CDI fully relies on runtime information and the capability to modify beans at bootstrap time. This requires the runtime implementation of CDI to include significant complexity in order to deal with the creation of runtime proxies, generate byte code and process reflective data and so on.

The CDI lite profile makes the existing runtime extension model optional for implementers, but adds the possibility to add extensions that are processed at build time.

An implementation can, optionally, provide an additional module that can be added to the classpath in order to enable support for runtime portable extensions, but this is not a requirement of the CDI lite specification.

Extensions in CDI lite are expected to be discovered by `ServiceLoader` and implement a new interface (such as `jakarta.enterprise.inject.extension.Extension`).

The following example shows a possible shape of this interface:

```
public interface Extension {
   default void discover(DiscoveryContext context) {};

   default void enhance(EnhancementContext context) {};

   default void synthesize(SynthesisContext context) {};

   default void validate(ValidationContext context){};
}
```

Each method represents a phase the order of which is defined by the method order above. A phase receives a context object which allows for future extensibility and is defined as a default interface method allowing users to optionally implement each phase.

The DiscoveryContext could be defined as follows:

```
interface DiscoveryContext {
   AppArchiveBuilder classes();
   Contexts contexts();
}
```

And provides the ability to register additional `Context` objects and classes that can be found by class queries.

The EnhancementContext could be defined as follows:

```
interface EnhancementContext {
   AppArchiveConfig archiveConfig();
   Messages messages();
}
```

The enhancement phase will allow developers to register annotation mutations such as adding or removing annotations on types.

The SynthesisContext could be defined as follows:

```
interface SynthesisContext {
   AppArchive archive();
   AppDeployment deployment();
   SyntheticComponents components();
   Messages messages();
}
```

Adds allow registration of synthetic beans.

Optional Scanning

Since CDI lite extensions are processed at build time the requirement to define `beans.xml` becomes optional for CDI lite implementations. The build time implementation can instead implement build time generated logic that eliminates completely the need for classpath scanning or XML-based bean discovery configuration.

Reduction of Scope

In addition to the above mentioned modification to the spec for the CDI lite profile some aspects of the current definition of CDI are considered out of scope and not supported by CDI lite because they either no longer make sense in the context of the profile or require a revision of another specification:

- Passivation - https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.html#passivating_scope
- Conversations - https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.html#conversation_context

- ConversionScope - https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.html#conversation
- Expression Language (EL) - https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.html#name_resolution_ee
- Runtime Portable Extensions - https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.html#spi

Other updates

There are some areas where we either need to use implementation specific features, or where each depending specification may come up with a custom approach. The following sections describe some of these that could be unified.

Executable methods

Currently CDI does not have a concept of a bean method that can be executed from another bean with parameter binding.
Such a concept is used in other specs, such as JAX-RS, where resource methods are executed at runtime with the need to bind their parameters, and provide the support for interceptors and other CDI features.
The CDI spec should add API for such a feature, so other specs do not need to define a custom solution.

The following should be possible:

1. Obtain a reference to executable method with all metadata, including annotations on method and on fields
2. Execute an executable method on a bean instance
3. Obtain executable method metadata in interceptors (to analyze annotations using reflection-less approach)
4. Create a parameter binder, that would support a specific annotation on a parameter (basically a parameter qualifier)

Pseudocode:

Let's consider the following bean:

```
@RequestScoped
@Path("/greet")
public class Resource {
   @GET
   String helloWorld(@HeaderParam("HEADER") String header) {
   }
}
```

The annotation definition:

```
@Qualifier
@Target(ElementType.PARAMETER)
public @interface HeaderParam {
   String value();
}
```

Now we could create a parameter binder:

```
@Singleton
@Binder(HeaderParam.class)
public class HeaderParamBinder implements ParamBinder {
   // HttpHeaders is a request scoped bean
   private final HttpHeaders headers;
   private final Converter converter;

   @Inject
   HeaderParamBinder(HttpHeaders headers, Converter converter) {
       this.headers = headers;
       this.converter = converter;
   }

   public <T> T bind(HeaderParam annotation, GenericType<T> type) {
       return
converter.convert(headers.getRequestHeader(annotation.value()),
type);
   }
}
```

And the execution of this method would be something like:

```
@MethodHandler(Path.class)
public class JaxRsExecutor implements ExecMethodHandler<Object, Object> {
   private final ParameterBinder binder;
   private final Container container;

   @Inject
```

```java
    JaxRsExecutor(ArgumentBinder binder) {
        this.binder = binder;
    }

    @Override
    public void processMethod(ExecutableMethod<Object, Object> method) {
        // process registration with web server
    }

    private void executeMethod(ExecutableMethod<Object, Object> method) {
        // start request scope
        Object beanInstance = container.select(method.getDeclaringType()).get();
        method.invoke(beanInstance, binder.bind(method));
    }
}
```

Bean defining annotations

Currently CDI specifies the annotations that must create a bean (such as @ApplicationScoped). This is limiting the extensibility of CDI, as we need to use implementation specific ways to add bean defining annotations for beans not annotated with “@Stereotype”.

It would be beneficial to add a specification way to add beans as bean defining annotations for such cases.
There is currently a method on `BeforeBeanDiscovery` that allows to add a stereotype, though it is not honored at runtime (as we may use some compile time index, such as Jandex).
Similar concept should exist for build-time processing, that would allow us to define a new stereotyped annotation to be honored by runtime processing.

Conceptual code:

```java
public class PathBda implements TypedAnnotationMapper<Path> {
   public List<AnnotationValue<?>> map(AnnotationValue<Path> annotation..)
{
    return List.of(AnnotationValue.builder(Path.class)
         .addStereotype(Stereotype.class)
         .addAnnotation(ApplicationScoped.class)
         .build());
}
}
```

Why do this in CDI?

When we discussed these topics, there were voices to move this outside of the CDI spec.

Here are a few reasons why we should make this part of CDI:

1. A new spec would mean a new set of downstream specifications - such as JAX-RS. If JAX-RS is CDI based, we would need to come up with JAX-RS Lite based on CDI Lite
2. A new spec would create fragmentation of the Java enterprise space. Currently there are two main forces in this space - Jakarta EE and Spring. Spring also uses/supports a set of Jakarta EE specifications. I think it is in the best interest of the community and Vendors to keep the number of competing specifications/approaches low
3. A certain amount of modernization will make CDI interesting even for modern microservices frameworks that focus on speed and footprint
4. The whole Eclipse MicroProfile would benefit from this, as if we split implementation, MP would need to decide which path to take. If it would take path of incompatible CDI Lite, all vendors that base their implementation on existing Jakarta EE application servers would not be able to implement this at all; if it would take path of CDI, vendors would not be able to benefit from the features of CDI Lite