AtInject Next proposed features by implementers

1 Warning

This document is a proposal in draft state. Its purpose is to be a starting point for discussion on new features or enhancement.

2 Contribution to this document

Contributions to this document are open the the community, but they can't be done anonymously. If you want to comment or suggest modification / new content, please make sure you are connected with your google Account and ask access by clicking on the share button at the upper right corner of the screen.

By asking the right to contribute you agree that:

- For all code provided, you are licensing the code under the Apache License, Version 2.
- For all other ideas provided, the provider waives all patent and other intellectual property rights inherent in such information.

3 CDI proposals

3.1 Support more than one qualifier

JSR 330 states that there shouldn't be more than one qualifier on an injected instance. We propose to remove this restriction to have better alignment with CDI.

3.2 Clarify type resolution done by the injector

JSR 330 is ambiguous regarding type resolution done for injection. The "Qualifiers" paragraph states:

> *A qualifier may annotate an injectable field or parameter and, combined with the type, identify the implementation to inject.*

and the "Injectable Values" section reads:

> *For a given type T and optional qualifier, an injector must be able to inject a user-specified class that:*
> - *is assignment compatible with T and*
> - *has an injectable constructor.*

It’s not clear how the qualifier is used to resolve injection. Regarding the type resolution, nothing is said about primitive types, arrays or parameterized types.

We propose a total or partial alignment with CDI type resolution. Rephrased with the JSR 330 concept it could be something like:

For a given type T and an optional set of qualifiers, the injector must be able to inject a user user-specified class :

- *Whose valid candidate to be instantiated by the Injector*
  - *Concrete Class*
  - *With an injectable constructor*
- *Whose type is assignable to the field or parameter to inject in. For this purpose, primitive types are considered to match their corresponding wrapper types in java.lang and array types are considered to match only if their element types are identical.*
- *Which has all or a subset of qualifiers present on the field or parameter to inject in. A class has a required qualifier if it has a qualifier with (a) the same type and (b) the same annotation member value for each member which is not annotated `@Nonbinding`.*

Moreover, the AtInject spec could also adopt the same rules than CDI regarding parameterized types as described in section 5.2.4 of the current specification, removing type erasure limitation for Dependency Injection.

3.3 Provider enhancement

In JSR 330 `Provider<T>` provides a way to lazily inject one or more instances of T, but it could be a way to have runtime resolution for a given instance (e.g. Choosing a qualifier at runtime). We propose to enhance `Provider` by adding methods present in CDI `Instance<T>` interface and optionally make `Provider` implement `Iterable` to be able to loop across the set of instances answering the filters.

We propose to create a subinterface of provider containing the following methods :

```
public interface IterableProvider<T> extends Provider<T>,Iterable<T> {

  /**
   * Provides a fully-constructed and injected instance of {@code T}.
   */
  T get();

  /**
   * Obtains a child <tt>Instance</tt> for the given additional required qualifiers.
   */
  Provider<T> select(Annotation... qualifiers);

  /**
   * Obtains a child <tt>Instance</tt> for the given required type and additional
   * required qualifiers.
   */
  <U extends T> Provider<U> select(Class<U> subtype, Annotation... qualifiers);

  /**
   * Determines if there is no bean that matches the required type and qualifiers
   * and is eligible for injection into the class
   * into which the parent <tt>Instance</tt> was injected.
   */
  boolean isUnsatisfied();

  /**
   * Determines if there is more than one bean that matches the required type and
   * qualifiers and is eligible for injection
   * into the class into which the parent <tt>Instance</tt> was injected.
   */
  boolean isAmbiguous();
}
```

3.4 Moving javax.enterprise.util package to AtInject

The `javax.enterprise.util` package contains 2 classes and one annotation. These 3 elements will complete the above proposals with these 2 features

3.4.1 Add AnnotationLiteral and Typeliteral

In order to have the proposed `Provider<T>` working, the spec need a mechanism to create instances of annotations ar parameterized types. The classes `AnnotationLiteral` and `TypeLiteral` for are in `javax.enterprise` package are designed for that.

3.4.2 Allow qualifiers to have non binding members

In AtInject there's no way to add a member to a qualifier annotation without making it part of the qualification.

By adding the `@NonBinding` meta annotation (included in `javax.enterprise` package) the spec will allow that.

4 Dagger proposals

4.1 Provider<T> semantic clarification

Provider<T> presently is described with a variety of uses, but not a lot of semantic clarity. This has led Spring to interpret injections of Provider<T> similarly to how Dagger views Lazy<T>. It also offers options that are not possible in some containers, but does not clarify what is optional behavior in a container, required treatment by the container, or simply clarifications of meaning *if* the interface is used for the listed purpose. The interpretation of Provider<T> at injection sites needs to be clarified.

4.2 Lazy<T>

Dagger introduced a concept of **Lazy<T>**. The API surface is identical to Provider<T> with a specific semantic difference - while **Provider<T>** indicates that using an injected provider may be useful for: "*retrieving multiple instances, lazy or optional retrieval of an instance, breaking circular dependencies, abstracting scope so you can look up an instance in a smaller scope from an instance in a containing scope",* **Lazy<T>** specifically returns one and only one instance of **T** per injection site, while Provider<T> may return a new instance for each call to get();. The semantic promise of Lazy is only that the object returned may be lazily constructed, but each call site will get its own. For example

```
// not a singleton
class Bar {
   @Inject Bar() {}
}

class Foo {
  @Inject Lazy<Bar> lazyBar1;
  @Inject Lazy<Bar> lazyBar2;
  @Inject Provider<Bar> providerOfBar1;
  @Inject Provider<Bar> providerOfBar2;
}

@Test public void testLazy {
  Foo foo = … // inject with a JSR-330 compatible thingy.
  assertThat(foo.lazyBar1).isSameAs(lazyBar2);
  assertThat(foo.providerOfBar1).isNotSameAs(providerOfBar2);
}
```

These two assertions would operate the same (at least in Dagger/Guice) if Bar was a @Singleton (or some other memoizing scope).

Advantages of Lazy over just using Provider…

- you can get lazy initialization of values of which you want to have only one instance at the call-site, but which cannot be singleton or otherwise memoized in the graph.
- For the usages of Provider<T> that were solely to provide lazy initialization, the word better captures the developer's intent.

Guice has an interest in implementing Lazy, making it potentially a candidate for the JSR spec, so as to allow common APIs to be shared for common abstractions.

5 Guice proposals

6 Spring proposals