JPA Implementation Patterns: Using UUIDs as Primary Keys

by Vincent Partington MVB · Sep. 01, 09 · Java Zone · Not set

The default way in JPA for primary keys is to use the @GeneratedValue annotation with the strategy attribute set to one of AUTO, IDENTITY, SEQUENCE, or TABLE. You pick the most appropriate strategy for your situation and that's it.
But you can also choose to generate the primary key yourself.

Using UUIDs for this is ideal and has some great benefits. In our current project we've used this strategy by creating an abstract base class our entities to inherit from which takes care of dealing with primary keys.

```
@MappedSuperclass
public abstract class AbstractBaseEntity implements Serializable {
        private static final long serialVersionUID = 1L;

        @Id
        private String id;

        public AbstractBaseEntity() {
                this.id = UUID.randomUUID().toString();
        }

        @Override
        public int hashCode() {
                return id.hashCode();
        }
        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (!(obj instanceof AbstractBaseEntity)) {
                        return false;
                }
                AbstractBaseEntity other = (AbstractBaseEntity) obj;
                return getId().equals(other.getId());
        }
}
```

Using UUIDs versus sequences in general has been widely discussed on the web, so I won't go into too much detail here. But here are some pro's and con's:

Pros

- Write this base class once and every entity gets an Id for free. If you implement equals and hashcode as above you also throw that one in as a bonus.
- UUID are Universal Unique(what's in a name). This means that you get great flexibility if you need to copy/merge records from location a to b without having to re-generate keys. (or use complex sequence strategies).
- UUIDs are not guessable. This means it can be safer to expose them to the outside world in let's say urls. If this would be good practice is another thing.

Cons

- Performance can be an issue. See http://johannburkard.de/blog/programming/java/Java-UUID-generators-compared.html, Some databases don't perform well with UUIDs (at least when they are stored as strings) for indexes.
- Ordering, Sorting. Speaks for itself.
- JPA specific: you cannot test if a record has already been persisted by checking if the Id field has been set. One might argue if you need such checks anyway.

Conclusion

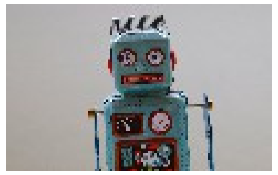
UUIDs are easy to use, but wouldn't it be nice if the JPA spec would open up to include UUID as a strategy? Some JPA implementations, like Hibernate, already have support for this:

```
@Id @GeneratedValue(generator="system-uuid")
@GenericGenerator(name="system-uuid",
  strategy = "uuid")
```

From http://blog.xebia.com/

Like This Article? Read More From DZone

Proposed Jakarta EE Design Principles



Advanced Research Initiatives: Focus on the Future

How to Configure NGINX High Availability Cluster Using Pacemaker on Ubuntu 16.04

Free DZone Refcard
Getting Started With Kotlin

Topics:

Opinions expressed by DZone contributors are their own.