23. Database Portability Considerations23.1. Portability Basics One of the selling points of Hibernate (and really Object/Relational Mapping as a whole) is the notion of database portability. This could mean an internal IT user migrating from one database vendor to another, or it could mean a framework or deployable application consuming Hibernate to simultaneously target multiple database products by their users. Regardless of the exact scenario, the basic idea is that you want Hibernate to help you run against any number of databases without changes to your code, and ideally without any changes to the mapping metadata. 23.2. Dialect The first line of portability for Hibernate is the dialect, which is a specialization of the org.hibernate.dialect.Dialect contract. A dialect encapsulates all the differences in how Hibernate must communicate with a particular database to accomplish some task like getting a sequence value or structuring a SELECT query. Hibernate bundles a wide range of dialects for many of the most popular databases. If you find that your particular database is not among them, it is not terribly difficult to write your own. 23.3. Dialect resolution Originally, Hibernate would always require that users specify which dialect to use. In the case of users looking to simultaneously target multiple databases with their build that was problematic. Generally, this required their users to configure the Hibernate dialect or defining their own method of setting that value. Starting with version 3.2, Hibernate introduced the notion of automatically detecting the dialect to use based on the java.sql.DatabaseMetaData obtained from a java.sql.Connection to that database. This was much better, except that this resolution was limited to databases Hibernate know about ahead of time and was in no way configurable or overrideable. Starting with version 3.3, Hibernate has a far more powerful way to automatically determine which dialect to be used by relying on a series of delegates which implement the org.hibernate.dialect.resolver.DialectResolver which defines only a single method: public Dialect resolveDialect(DatabaseMetaData metaData) throws JDBCConnectionException The basic contract here is that if the resolver 'understands' the given database metadata then it returns the corresponding Dialect; if not it returns null and the process continues to the next resolver. The signature also identifies org.hibernate.exception.JDBCConnectionException as possibly being thrown. A JDBCConnectionException here is interpreted to imply a non-transient (aka non-recoverable) connection problem and is used to indicate an immediate stop to resolution attempts. All other exceptions result in a warning and continuing on to the next resolver. The cool part about these resolvers is that users can also register their own custom resolvers which will be processed ahead of the built-in Hibernate ones. This might be useful in a number of different situations: it allows easy integration for auto-detection of dialects beyond those shipped with Hibernate itself. it allows you to specify to use a custom dialect when a particular database is recognized. To register one or more resolvers, simply specify them (separated by commas, tabs or spaces) using the 'hibernate.dialect\_resolvers' configuration setting (see the DIALECT\_RESOLVERS constant on org.hibernate.cfg.Environment). 23.4. Identifier generation When considering portability between databases, another important decision is selecting the identifier generation strategy you want to use. Originally, Hibernate provided the native generator for this purpose, which was intended to select between a sequence, identity, or table strategy depending on the capability of the underlying database. However, an insidious implication of this approach comes about when targeting some databases which support identity generation and some which do not. identity generation relies on the SQL definition of an IDENTITY (or auto-increment) column to manage the identifier value. It is what is known as a post-insert generation strategy because the insert must actually happen before we can know the identifier value. Because Hibernate relies on this identifier value to uniquely reference entities within a persistence context, it must then issue the insert immediately when the user requests that the entity be associated with the session (e.g. like via save() or persist()), regardless of current transactional semantics. Hibernate was changed slightly, once the implications of this were better understood, so now the insert could be delayed in cases where this is feasible. The underlying issue is that the actual semantics of the application itself changes in these cases. Starting with version 3.2.3, Hibernate comes with a set of enhanced identifier generators targeting portability in a much different way. There are specifically 2 bundled enhanced generators: org.hibernate.id.enhanced.SequenceStyleGenerator org.hibernate.id.enhanced.TableGenerator The idea behind these generators is to port the actual semantics of the identifier value generation to the different databases. For example, the org.hibernate.id.enhanced.SequenceStyleGenerator mimics the behavior of a sequence on databases which do not support sequences by using a table. 23.5. Database functions HQL now provides a large set of functions which are portable between databases. You can find them listed in the chapter describing the query language. There’s even a way for a program to contribute its own function definitions. Of course, SQL functions occurring in handwritten SQL fragments or queries usually aren’t very portable. 23.6. Type mappings TODO 23.6.1. BLOB/CLOB mappings TODO 23.6.2. Boolean mappings 23.6.3. Jakarta Persistence portability HQL/JPQL differences naming strategies basic types simple id types generated id types composite ids and many-to-one "embedded composite identifiers"