**Write fine-grained classes and map them using <component>:**

Use an Address class to encapsulate street, suburb, state, postcode. This encourages code reuse and simplifies refactoring.

**Declare identifier properties on persistent classes:**

Hibernate makes identifier properties optional. There are a range of reasons why you should use them. We recommend that identifiers be 'synthetic', that is, generated with no business meaning.

**Identify natural keys:**

Identify natural keys for all entities, and map them using <natural-id>. Implement equals() and hashCode() to compare the properties that make up the natural key.

**Place each class mapping in its own file:**

Do not use a single monolithic mapping document. Map com.eg.Foo in the file com/eg/Foo.hbm.xml. This makes sense, particularly in a team environment.

**Load mappings as resources:**

Deploy the mappings along with the classes they map.

**Consider externalizing query strings:**

This is recommended if your queries call non-ANSI-standard SQL functions. Externalizing the query strings to mapping files will make the application more portable.

**Use bind variables.**

As in JDBC, always replace non-constant values by "?". Do not use string manipulation to bind a non-constant value in a query. You should also consider using named parameters in queries.

**Do not manage your own JDBC connections:**

Hibernate allows the application to manage JDBC connections, but his approach should be considered a last-resort. If you cannot use the built-in connection providers, consider providing your own implementation of org.hibernate.connection.ConnectionProvider.

**Consider using a custom type:**

Suppose you have a Java type from a library that needs to be persisted but does not provide the accessors needed to map it as a component. You should consider implementing org.hibernate.UserType. This approach frees the application code from implementing transformations to/from a Hibernate type.

**Use hand-coded JDBC in bottlenecks:**

In performance-critical areas of the system, some kinds of operations might benefit from direct JDBC. Do not assume, however, that JDBC is necessarily faster. Please wait until you *know* something is a bottleneck. If you need to use direct JDBC, you can open a Hibernate Session and usingfile:///usr/share/doc/HTML/en-US/index.html that JDBC connection. This way you can still use the same transaction strategy and underlying connection provider.

**Understand Session flushing:**

Sometimes the Session synchronizes its persistent state with the database. Performance will be affected if this process occurs too often. You can sometimes minimize unnecessary flushing by disabling automatic flushing, or even by changing the order of queries and other operations within a particular transaction.

**In a three tiered architecture, consider using detached objects:**

When using a servlet/session bean architecture, you can pass persistent objects loaded in the session bean to and from the servlet/JSP layer. Use a new session to service each request. Use Session.merge() or Session.saveOrUpdate() to synchronize objects with the database.

**In a two tiered architecture, consider using long persistence contexts:**

Database Transactions have to be as short as possible for best scalability. However, it is often necessary to implement long running *application transactions*, a single unit-of-work from the point of view of a user. An application transaction might span several client request/response cycles. It is common to use detached objects to implement application transactions. An appropriate alternative in a two tiered architecture, is to maintain a single open persistence contact session for the whole life cycle of the application transaction. Then simply disconnect from the JDBC connection at the end of each request and reconnect at the beginning of the subsequent request. Never share a single session across more than one application transaction or you will be working with stale data.

**Do not treat exceptions as recoverable:**

This is more of a necessary practice than a "best" practice. When an exception occurs, roll back the Transaction and close the Session. If you do not do this, Hibernate cannot guarantee that in-memory state accurately represents the persistent state. For example, do not use Session.load() to determine if an instance with the given identifier exists on the database; use Session.get() or a query instead.

**Prefer lazy fetching for associations:**

Use eager fetching sparingly. Use proxies and lazy collections for most associations to classes that are not likely to be completely held in the second-level cache. For associations to cached classes, where there is an a extremely high probability of a cache hit, explicitly disable eager fetching using lazy="false". When join fetching is appropriate to a particular use case, use a query with a left join fetch.

**Use the *open session in view* pattern, or a disciplined *assembly phase* to avoid problems with unfetched data:**

Hibernate frees the developer from writing tedious *Data Transfer Objects* (DTO). In a traditional EJB architecture, DTOs serve dual purposes: first, they work around the problem that entity beans are not serializable; second, they implicitly define an assembly phase where all data to be used by the view is fetched and marshalled into the DTOs before returning control to the presentation tier. Hibernate eliminates the first purpose. Unless you are prepared to hold the persistence context (the session) open across the view rendering process, you will still need an assembly phase. Think of your business methods as having a strict contract with the presentation tier about what data is available in the detached objects. This is not a limitation of Hibernate. It is a fundamental requirement of safe transactional data access.

**Consider abstracting your business logic from Hibernate:**

Hide Hibernate data-access code behind an interface. Combine the *DAO* and *Thread Local Session* patterns. You can even have some classes persisted by handcoded JDBC associated to Hibernate via a UserType. This advice is, however, intended for "sufficiently large" applications. It is not appropriate for an application with five tables.

**Do not use exotic association mappings:**

Practical test cases for real many-to-many associations are rare. Most of the time you need additional information stored in the "link table". In this case, it is much better to use two one-to-many associations to an intermediate link class. In fact, most associations are one-to-many and many-to-one. For this reason, you should proceed cautiously when using any other association style.

**Prefer bidirectional associations:**

Unidirectional associations are more difficult to query. In a large application, almost all associations must be navigable in both directions in queries.