scores might have a Team and a Game object. Whatever it is, there's a good chance that you've modeled the concepts in your system in an object model. It is a common practice in Maven projects to separate this project into a separate project which is widely referenced. In this system we are capturing each query to the Yahoo Weather feed with a Weather object which references four other objects. Wind direction, chill, and speed are stored in a Wind object. Location data including the zip code, city, region, and country are stored in a Location class. Atmospheric conditions such as the humidity, maximum visibility, barometric pressure, and whether the pressure is rising or falling are stored in an Atmosphere class. A textual description of conditions, the temperature, and the date of the observation is stored in a Condition class.

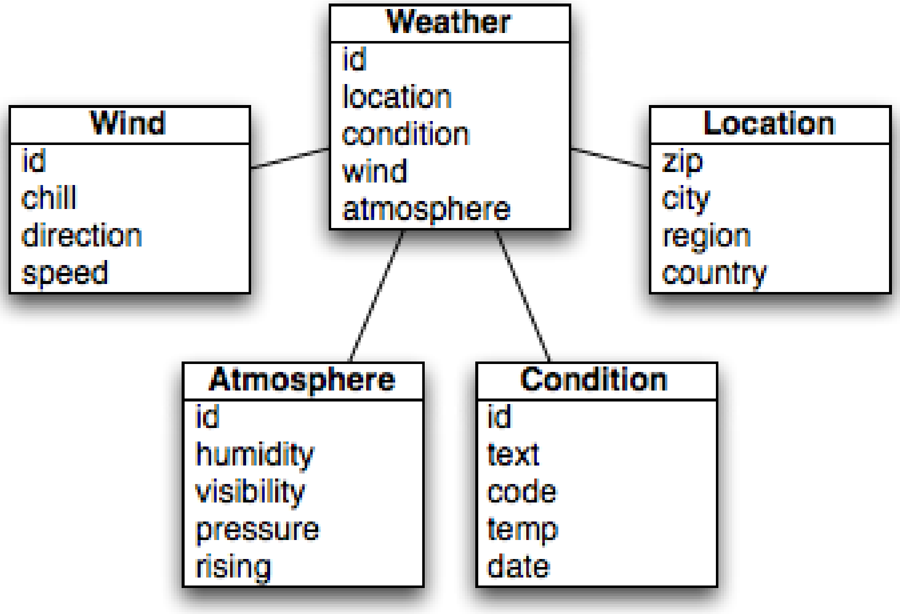

Figure 7.2: Simple Object Model for Weather Data

The pom.xml file for this simple object model contains one dependency that bears some explanation. Our object model is annotated with Hibernate Annotations. We use these annotations to map the model objects in this model to tables in a relational database. The dependency is org.hibernate:hibernate-annotations:3.3.0.ga. Take a look at the pom.xml shown in simple-model pom.xml, and then look at the next few examples for some illustrations of these annotations.

simple-model pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.sonatype.mavenbook.multispring</groupId>
        <artifactId>simple-parent</artifactId>
        <version>1.0</version>
    </parent>
    <artifactId>simple-model</artifactId>
    <packaging>jar</packaging>

    <name>Simple Object Model</name>

    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-annotations</artifactId>
            <version>3.3.0.ga</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-commons-annotations</artifactId>
            <version>3.3.0.ga</version>
        </dependency>
    </dependencies>
</project>
```

In `src/main/java/org/sonatype/mavenbook/weather/model`, we have `Weather.java`, which contains the annotated `Weather` model object. The `Weather` object is a simple Java bean. This means that we have private member variables like `id`, `location`, `condition`, `wind`, `atmosphere`, and `date` exposed with public getter and setter methods that adhere to the following pattern: if a property is named `name`, there will be a public no-arg getter method named `getName()`, and there will be a one-argument setter named `setName(String name)`. Although we show the getter and setter methods for the `id` property, we've omitted most of the getters and setters for most of the other properties to save a few trees. See Annotated Weather Model Object.

Annotated Weather Model Object

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

import java.util.Date;

@Entity
```

```
@NamedQueries({
  @NamedQuery(name="Weather.byLocation",
              query="from Weather w where w.location = :location")
})
public class Weather {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id;

  @ManyToOne(cascade=CascadeType.ALL)
  private Location location;

  @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
  private Condition condition;

  @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
  private Wind wind;

  @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
  private Atmosphere atmosphere;

  private Date date;

  public Weather() {}

  public Integer getId() { return id; }
  public void setId(Integer id) { this.id = id; }

  // All getter and setter methods omitted...
}
```

In the `Weather` class, we are using Hibernate annotations to provide guidance to the `simple-persist` project. These annotations are used by Hibernate to map an object to a table in a relational database. Although a full explanation of Hibernate annotations is beyond the scope of this chapter, here is a brief explanation for the curious. The `@Entity` annotation marks this class as a persistent entity. We've omitted the `@Table` annotation on this class, so Hibernate is going to use the name of the class as the name of the table to map `Weather` to. The `@NamedQueries` annotation defines a query that is used by the `WeatherDAO` in `simple-persist`. The query language in the `@NamedQuery` annotation is written in something called Hibernate Query Language (HQL). Each member variable is annotated with annotations that define the type of column and any relationships implied by that column:

Id

The `id` property is annotated with `@Id`. This marks the `id` property as the property that contains the primary key in a database table. The `@GeneratedValue` controls how new primary key values are generated. In the case of id, we're using the IDENTITY GenerationType, which will use the underlying database's identity generation facilities.

Location
Each Weather object instance corresponds to a Location object. A Location object represents a zip code, and the @ManyToOne makes sure that Weather objects that point to the same Location object reference the same instance. The cascade attribute of the @ManyToOne makes sure that we persist a Location object every time we persist a Weather object.

Condition, Wind, Atmosphere
Each of these objects is mapped as a @OneToOne with the CascadeType of ALL. This means that every time we save a Weather object, we'll be inserting a row into the Weather table, the Condition table, the Wind table, and the Atmosphere table.

Date
Date is not annotated. This means that Hibernate is going to use all of the column defaults to define this mapping. The column name is going to be date, and the column type is going to be the appropriate time to match the Date object.

Note
If you have a property you wish to omit from a table mapping, you would annotate that property with @ Transient.

Next, take a look at one of the secondary model objects, Condition, shown in simple-model's Condition Model Object.. This class also resides in src/main/java/org/sonatype/mavenbook/weather/model.

simple-model's Condition Model Object.

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

@Entity
public class Condition {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id;

  private String text;
  private String code;
  private String temp;
  private String date;
  @OneToOne(cascade=CascadeType.ALL)
  @JoinColumn(name="weather_id", nullable=false)
  private Weather weather;

  public Condition() {}

  public Integer getId() { return id; }
  public void setId(Integer id) { this.id = id; }

  // All getter and setter methods omitted...
}
```

The `Condition` class resembles the `Weather` class. It is annotated as an `@Entity`, and it has similar annotations on the `id` property. The `text`, `code`, `temp`, and `date` properties are all left with the default column settings, and the `weather` property is annotated with a `@OneToOne` annotation and another annotation that references the associated `Weather` object with a foreign key column named `weather_id`.

7.4 The Simple Weather Module

The next module we're going to examine could be considered something of a "service." The Simple Weather module is the module that contains all of the logic necessary to retrieve and parse the data from the Yahoo Weather RSS feed. Although Simple Weather contains three Java classes and one JUnit test, it is going to present a single component, `WeatherService`, to both the Simple Web Application and the Simple Command-Line Utility. Very often an enterprise project will contain several API modules that contain critical business logic or logic that interacts with external systems. A banking system might have a module that retrieves and parses data from a third-party data provider, and a system to display sports scores might interact with an XML feed that presents real-time scores for basketball or soccer. In simple-weather Module POM, this module encapsulates all of the network activity and XML parsing that is involved in the interaction with Yahoo Weather. Other modules can depend on this module and simply call out to the `retrieveForecast()` method on `WeatherService`, which takes a zip code as an argument and which returns a `Weather` object.

simple-weather Module POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
```

```
        <groupId>org.sonatype.mavenbook.multispring</groupId>
        <artifactId>simple-parent</artifactId>
        <version>1.0</version>
    </parent>
    <artifactId>simple-weather</artifactId>
    <packaging>jar</packaging>

    <name>Simple Weather API</name>

    <dependencies>
        <dependency>
            <groupId>org.sonatype.mavenbook.multispring</groupId>
            <artifactId>simple-model</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.14</version>
        </dependency>
        <dependency>
            <groupId>dom4j</groupId>
            <artifactId>dom4j</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>jaxen</groupId>
            <artifactId>jaxen</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-io</artifactId>
            <version>1.3.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

The `simple-weather` POM extends the `simple-parent` POM, sets the packaging to `jar`, and then adds the following dependencies:

`org.sonatype.mavenbook.multispring:simple-model:1.0`

`simple-weather` parses the Yahoo Weather RSS feed into a `Weather` object. It has a direct dependency on `simple-model`.

log4j:log4j:1.2.14

simple-weather uses the Log4J library to print log messages.

dom4j:dom4j:1.6.1 and jaxen:jaxen:1.1.1

Both of these dependencies are used to parse the XML returned from Yahoo Weather.

org.apache.commons:commons-io:1.3.2 (scope=test)

This test-scoped dependency is used by the YahooParserTest.

Next is the WeatherService class, shown in WeatherService Class. This class is going to look very similar to the WeatherService class from The WeatherService Class. Although the WeatherService is the same, there are some subtle differences in this chapter's example. This version's retrieveForecast() method returns a Weather object, and the formatting is going to be left to the applications that call WeatherService. The other major change is that the YahooRetriever and YahooParser are both bean properties of the WeatherService bean.

WeatherService Class

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherService {

    private YahooRetriever yahooRetriever;
    private YahooParser yahooParser;

    public WeatherService() {
    }

    public Weather retrieveForecast(String zip) throws Exception {
        // Retrieve Data
        InputStream dataIn = yahooRetriever.retrieve(zip);

        // Parse DataS
        Weather weather = yahooParser.parse(zip, dataIn);

        return weather;
    }

    public YahooRetriever getYahooRetriever() {
        return yahooRetriever;
    }

    public void setYahooRetriever(YahooRetriever yahooRetriever) {
        this.yahooRetriever = yahooRetriever;
    }

    public YahooParser getYahooParser() {
        return yahooParser;
    }

    public void setYahooParser(YahooParser yahooParser) {
        this.yahooParser = yahooParser;
    }

}
```

Finally, in this project we have an XML file that is used by the Spring Framework to create something called an `ApplicationContext`. First, some explanation: both of our applications, the web application and the command-line utility, need to interact with the `WeatherService` class, and they both do so by retrieving an instance of this class from a Spring `ApplicationContext` using the name `weatherService`. Our web application uses a Spring MVC controller that is associated with an instance of `WeatherService`, and our command-line utility loads the `WeatherService` from an `ApplicationContext` in a static `main()` function. To encourage reuse, we've included an `applicationContext-weather.xml` file in `src/main/resources`, which is available on the classpath. Modules that depend on the `simple-weather` module can load this application context using the `ClasspathXmlApplicationContext` in the Spring Framework. They can then reference a named instance of the `WeatherService` named `weatherService`.

Spring Application Context for the simple-weather Module

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
       default-lazy-init="true">

    <bean id="weatherService"
          class="org.sonatype.mavenbook.weather.WeatherService">
        <property name="yahooRetriever" ref="yahooRetriever"/>
        <property name="yahooParser" ref="yahooParser"/>
    </bean>

    <bean id="yahooRetriever"
          class="org.sonatype.mavenbook.weather.YahooRetriever"/>

    <bean id="yahooParser"
          class="org.sonatype.mavenbook.weather.YahooParser"/>
```

```
</beans>
```

This document defines three beans: `yahooParser`, `yahooRetriever`, and `weatherService`. The `weatherService` bean is an instance of `WeatherService`, and this XML document populates the `yahooParser` and `yahooRetriever` properties with references to the named instances of the corresponding classes. Think of this `applicationContext-weather.xml` file as defining the architecture of a subsystem in this multi-module project. Projects like `simple-webapp` and `simple-command` can reference this context and retrieve an instance of `WeatherService` which already has relationships to instances of `YahooRetriever` and `YahooParser`.

7.5 The Simple Persist Module

This module defines two very simple Data Access Objects (DAOs). A DAO is an object that provides an interface for persistence operations. In an application that makes use of an Object-Relational Mapping (ORM) framework such as Hibernate, DAOs are usually defined around objects. In this project, we are defining two DAO objects: `WeatherDAO` and `LocationDAO`. The `WeatherDAO` class allows us to save a `Weather` object to a database and retrieve a `Weather` object by `id`, and to retrieve `Weather` objects that match a specific `Location`. The `LocationDAO` has a method that allows us to retrieve a `Location` object by zip code. First, let's take a look at the `simple-persist` POM in simple-persist POM.

simple-persist POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.sonatype.mavenbook.multispring</groupId>
        <artifactId>simple-parent</artifactId>
        <version>1.0</version>
    </parent>
    <artifactId>simple-persist</artifactId>
    <packaging>jar</packaging>

    <name>Simple Persistence API</name>

    <dependencies>
        <dependency>
            <groupId>org.sonatype.mavenbook.multispring</groupId>
            <artifactId>simple-model</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate</artifactId>
            <version>3.2.5.ga</version>
            <exclusions>
                <exclusion>
                    <groupId>javax.transaction</groupId>
                    <artifactId>jta</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-annotations</artifactId>
            <version>3.3.0.ga</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-commons-annotations</artifactId>
            <version>3.3.0.ga</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.4</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring</artifactId>
            <version>2.0.7</version>
        </dependency>
    </dependencies>
</project>
```

This POM file references `simple-parent` as a parent POM, and it defines a few dependencies. The dependencies listed in `simple-persist`'s POM are:

`org.sonatype.mavenbook.multispring:simple-model:1.0`

Just like the `simple-weather` module, this persistence module references the core model objects defined in `simple-model`.

`org.hibernate:hibernate:3.2.5.ga`

We define a dependency on Hibernate version 3.2.5.ga, but notice that we're excluding a dependency of Hibernate. We're doing this because the `javax.transaction:jta` dependency is not available in the public Maven repository. This dependency happens to be one of those Sun dependencies that has not yet made it into the free central Maven repository. To avoid an annoying message telling us to go download these nonfree dependencies, we simply exclude this dependency from Hibernate.

`javax.servlet:servlet-api:2.4`
Since this project contains a Servlet, we need to include the Servlet API version 2.4.

`org.springframework:spring:2.0.7`
This includes the entire Spring Framework as a dependency. It is generally a good practice to depend on only the components of Spring you happen to be using. The Spring Framework project has been nice enough to create focused artifacts such as `spring-hibernate3`.

Why depend on Spring? When it comes to Hibernate integration, Spring allows us to leverage helper classes such as `HibernateDaoSupport`. For an example of what is possible with the help of `HibernateDaoSupport`, take a look at the code for the `WeatherDAO` in simple-persist's WeatherDAO Class.

simple-persist's WeatherDAO Class

```
package org.sonatype.mavenbook.weather.persist;

import java.util.ArrayList;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherDAO extends HibernateDaoSupport { ❶

  public WeatherDAO() {}

  public void save(Weather weather) { ❷
    getHibernateTemplate().save( weather );
  }

  public Weather load(Integer id) { ❸
    return (Weather) getHibernateTemplate().load( Weather.class, id);
  }

  @SuppressWarnings("unchecked")
  public List<Weather> recentForLocation( final Location location ) {
    return (List<Weather>) getHibernateTemplate().execute(
        new HibernateCallback() { ❹
          public Object doInHibernate(Session session) {
              Query query =
                getSession().getNamedQuery("Weather.byLocation");
              query.setParameter("location", location);
              return new ArrayList<Weather>( query.list() );
        }
    });
  }
}
```

That’s it. No really, you are done writing a class that can insert new rows, select by primary key, and find all rows in Weather that join to an id in the Location table. Clearly, we can’t stop this book and insert the five hundred pages it would take to get you up to speed on the intricacies of Hibernate, but we can do some very quick explanation:

❶ This class extends `HibernateDaoSupport`. What this means is that the class is going to be associated with a Hibernate `SessionFactory` which it is going to use to create Hibernate `Session` objects. In Hibernate, every operation goes through a `Session` object, a `Session` mediates access to the underlying database and takes care of managing the connection to the JDBC `DataSource`. Extending `HibernateDaoSupport` also means that we can access the `HibernateTemplate` using `getHibernateTemplate()`. For an example of what can be done with the `HibernateTemplate`...

❷ The `save()` method takes an instance of `Weather` and calls the `save()` method on a `HibernateTemplate`. The `HibernateTemplate` simplifies calls to common Hibernate operations and converts any database specific exceptions to runtime exceptions. Here we call out to `save()` which inserts a new record into the `Weather` table. Alternatives to `save()` are `update()` which updates an existing row, or `saveOrUpdate()` which would either save or update depending on the presence of a non-null `id` property in Weather.

❸ The `load()` method, once again, is a one-liner that just calls a method on an instance of `HibernateTemplate`. `load()` on `HibernateTemplate` takes a `Class` object and a `Serializable` object. In this case, the `Serializable` corresponds to the `id` value of the `Weather` object to load.

❹ This last method `recentForLocation()` calls out to a `NamedQuery` defined in the `Weather` model object. If you can think back that far, the `Weather` model object defined a named query `"Weather.byLocation"` with a query of `"from Weather w where w.location =:location"`. We’re loading this `NamedQuery` using a reference to a Hibernate `Session` object inside a `HibernateCallback` which is executed by the `execute()` method on `HibernateTemplate`. You can see in this method that we’re populating the named parameter `location` with the parameter passed in to the `recentForLocation()` method.

Now is a good time for some clarification. HibernateDaoSupport and HibernateTemplate are classes from the Spring Framework. They were created by the Spring Framework to make writing Hibernate DAO objects painless. To support this DAO, we'll need to do some configuration in the simple-persist Spring ApplicationContext definition. The XML document shown in Spring Application Context for simple-persist is stored in src/main/resources in a file named applicationContext-persist.xml.

Spring Application Context for simple-persist

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
       default-lazy-init="true">

  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation. ↩
       AnnotationSessionFactoryBean">
    <property name="annotatedClasses">
      <list>
        <value>org.sonatype.mavenbook.weather.model.Atmosphere</value>
        <value>org.sonatype.mavenbook.weather.model.Condition</value>
        <value>org.sonatype.mavenbook.weather.model.Location</value>
        <value>org.sonatype.mavenbook.weather.model.Weather</value>
        <value>org.sonatype.mavenbook.weather.model.Wind</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.show_sql">false</prop>
        <prop key="hibernate.format_sql">true</prop>
        <prop key="hibernate.transaction.factory_class">
          org.hibernate.transaction.JDBCTransactionFactory
        </prop>
        <prop key="hibernate.dialect">
          org.hibernate.dialect.HSQLDialect
        </prop>
        <prop key="hibernate.connection.pool_size">0</prop>
        <prop key="hibernate.connection.driver_class">
          org.hsqldb.jdbcDriver
        </prop>
        <prop key="hibernate.connection.url">
          jdbc:hsqldb:data/weather;shutdown=true
        </prop>
        <prop key="hibernate.connection.username">sa</prop>
        <prop key="hibernate.connection.password"></prop>
        <prop key="hibernate.connection.autocommit">true</prop>
        <prop key="hibernate.jdbc.batch_size">0</prop>
      </props>
    </property>
  </bean>

  <bean id="locationDAO"
        class="org.sonatype.mavenbook.weather.persist.LocationDAO">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

  <bean id="weatherDAO"
        class="org.sonatype.mavenbook.weather.persist.WeatherDAO">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>
</beans>
```

In this application context, we're accomplishing a few things. The `sessionFactory` bean is the bean from which the DAOs retrieve Hibernate `Session` objects. This bean is an instance of `AnnotationSessionFactoryBean` and is supplied with a list of `annotatedClasses`. Note that the list of annotated classes is the list of classes defined in our `simple-model` module. Next, the `sessionFactory` is configured with a set of Hibernate configuration properties (`hibernateProperties`). In this example, our Hibernate properties define a number of settings:

`hibernate.dialect`
This setting controls how SQL is to be generated for our database. Since we are using the HSQLDB database, our database dialect is set to `org.hibernate.dialect.HSQLDialect`. Hibernate has dialects for all major databases such as Oracle, MySQL, Postgres, and SQL Server.

`hibernate.connection.*`
In this example, we're configuring the JDBC connection properties from the Spring configuration. Our applications are configured to run against a HSQLDB in the `./data/weather` directory. In a real enterprise application, it is more likely you would use something like JNDI to externalize database configuration from your application's code.

Lastly, in this bean definition file, both of the `simple-persist` DAO objects are created and given a reference to the `sessionFactory` bean just defined. Just like the Spring application context in `simple-weather`, this `applicationContext-persist.xml` file defines the architecture of a submodule in a larger enterprise design. If you were working with a larger collection of persistence classes, you might find it useful to capture them in an application context which is separate from your application.

There's one last piece of the puzzle in `simple-persist`. Later in this chapter, we're going to use `hibernate.cfg.xml` in `src/main/resources`. The purpose of this file (which duplicates some of the configuration in `applicationContext-persist.xml`) is to allow us to leverage the Maven

Hibernate3 plugin to generate Data Definition Language (DDL) from nothing more than our annotations. See simple-persist hibernate.cfg.xml.

simple-persist hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <!-- SQL dialect -->
    <property name="dialect">
      org.hibernate.dialect.HSQLDialect
    </property>

    <!-- Database connection settings -->
    <property name="connection.driver_class">
      org.hsqldb.jdbcDriver
    </property>
    <property name="connection.url">jdbc:hsqldb:data/weather</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.shutdown">true</property>

    <!-- JDBC connection pool (use the built-in one) -->
    <property name="connection.pool_size">1</property>

    <!-- Enable Hibernate’s automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache  -->
    <property name="cache.provider_class">
          org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>

    <!-- disable batching so HSQLDB will propagate errors correctly. -->
    <property name="jdbc.batch_size">0</property>

    <!-- List all the mapping documents we’re using -->
    <mapping class="org.sonatype.mavenbook.weather.model.Atmosphere"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Condition"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Location"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Weather"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Wind"/>
  </session-factory>
</hibernate-configuration>
```

The contents of Spring Application Context for simple-persist and simple-parent Project POM are redundant. While the Spring Application Context XML is going to be used by the web application and the command-line application, the hibernate.cfg.xml exists only to support the Maven Hibernate3 plugin. Later in this chapter, we'll see how to use this hibernate.cfg.xml and the Maven Hibernate3 plugin to generate a database schema based on the annotated object model defined in simple-model. This hibernate.cfg.xml file is the file that will configure the JDBC connection properties and enumerate the list of annotated model classes for the Maven Hibernate3 plugin.

7.6 The Simple Web Application Module

The web application is defined in a simple-webapp project. This simple web application project is going to define two Spring MVC Controllers: WeatherController and simple-weather and the applicationContext-persist.xml file in simple-persist. The component architecture of this simple web application is shown in Figure 7.3.

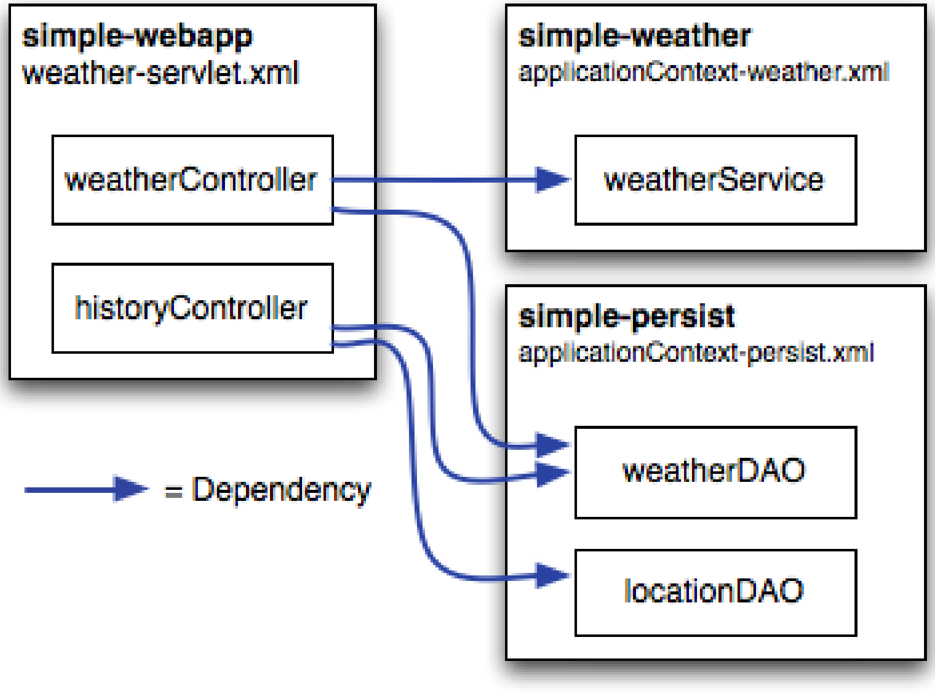

Figure 7.3: Spring MVC Controllers Referencing Components in simple-weather and simple-persist.

The POM for `simple-webapp` is shown in POM for simple-webapp.

POM for simple-webapp

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>Simple Web Application</name>
  <dependencies>
    <dependency> ❶
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-persist</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin> ❷
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <dependencies> ❸
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId> ❹
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.0</version>
        <configuration>
          <components>
```

```
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation> ❺
          </component>
        </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

As this book progresses and the examples become more and more substantial, you'll notice that the `pom.xml` begins to take on some weight. In this POM, we're configuring four dependencies and two plugins. Let's go through this POM in detail and dwell on some of the important configuration points:

❶ This `simple-webapp` project defines four dependencies: the Servlet 2.4 specification, the simple-weather service library, the simple-persist persistence library, and the entire Spring Framework 2.0.7.

❷ The Maven Jetty plugin couldn't be easier to add to this project; we simply add a `plugin` element that references the appropriate `groupId` and `artifactId`. The fact that this plugin is so trivial to configure means that the plugin developers did a good job of providing adequate defaults that don't need to be overridden in most cases. If you did need to override the configuration of the Jetty plugin, you would do so by providing a `configuration` element.

❸ In our build configuration, we're going to be configuring the Maven Hibernate3 Plugin to hit an embedded HSQLDB instance. For the Maven Hibernate 3 plugin to successfully connect to this database using JDBC, the plugin will need to reference the HSQLDB JDBC driver on the classpath. To make a dependency available for a plugin, we add a dependency declaration right inside the plugin declaration. In this case, we're referencing hsqldb:hsqldb:1.8.0.7. The Hibernate plugin also needs the JDBC driver to create the database, so we have also added this dependency to its configuration.

❹ The Maven Hibernate plugin is when this POM starts to get interesting. In the next section, we're going to run the `hbm2ddl` goal to generate a HSQLDB database. In this `pom.xml`, we're including a reference to version 2.0 of the `hibernate3-maven-plugin` hosted by the Codehaus Mojo plugin.

5 The Maven Hibernate3 plugin has different ways to obtain Hibernate mapping information that are appropriate for different usage scenarios of the Hibernate3 plugin. If you were using Hibernate Mapping XML (.hbm.xml) files, and you wanted to generate model classes using the hbm2java goal, you would set your implementation to configuration. If you were using the Hibernate3 plugin to reverse engineer a database to produce .hbm.xml files and model classes from an existing database, you would use an implementation of jdbcconfiguration. In this case, we're simply using an existing annotated object model to generate a database. In other words, we have our Hibernate mapping, but we don't yet have a database. In this usage scenario, the appropriate implementation value is annotationconfiguration. The Maven Hibernate3 plugin is discussed in more detail in the later section Section 7.7.

Next, we turn our attention to the two Spring MVC controllers that will handle all of the requests. Both of these controllers reference the beans defined in simple-weather and simple-persist.

simple-webapp WeatherController

```
package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;
import org.sonatype.mavenbook.weather.WeatherService;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class WeatherController implements Controller {

  private WeatherService weatherService;
  private WeatherDAO weatherDAO;

  public ModelAndView handleRequest(HttpServletRequest request,
                                    HttpServletResponse response)
                                  throws Exception {

    String zip = request.getParameter("zip");
    Weather weather = weatherService.retrieveForecast(zip);
    weatherDAO.save(weather);
    return new ModelAndView("weather", "weather", weather);
  }

  public WeatherService getWeatherService() {
    return weatherService;
  }

  public void setWeatherService(WeatherService weatherService) {
    this.weatherService = weatherService;
  }
  public WeatherDAO getWeatherDAO() {
    return weatherDAO;
  }

  public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
  }
}
```

WeatherController implements the Spring MVC Controller interface that mandates the presence of a handleRequest() method with the signature shown in the example. If you look at the meat of this method, you'll see that it invokes the retrieveForecast() method on the weatherService instance variable. Unlike the previous chapter, which had a Servlet that instantiated the WeatherService class, the WeatherController is a bean with a weatherService property. The Spring IoC container is responsible for wiring the controller to the weatherService component. Also notice that we're not using the WeatherFormatter in this Spring controller implementation; instead, we're passing the Weather object returned by retrieveForecast() to the constructor of ModelAndView. This ModelAndView class is going to be used to render a Velocity template, and this template will have references to a ${weather} variable. The weather.vm template is stored in src/main/webapp/WEB-INF/vm and is shown in weather.vm Template Rendered by WeatherController.

In the WeatherController, before we render the output of the forecast, we pass the Weather object returned by the WeatherService to the save() method on WeatherDAO. Here we are saving this Weather object—using Hibernate—to an HSQLDB database. Later, in HistoryController, we will see how we can retrieve a history of weather forecasts that were saved by the WeatherController.

weather.vm Template Rendered by WeatherController

```
<b>Current Weather Conditions for:
${weather.location.city}, ${weather.location.region},
${weather.location.country}</b><br/>

<ul>
<li>Temperature: ${weather.condition.temp}</li>
<li>Condition: ${weather.condition.text}</li>
<li>Humidity: ${weather.atmosphere.humidity}</li>
<li>Wind Chill: ${weather.wind.chill}</li>
<li>Date: ${weather.date}</li>
</ul>
```

The syntax for this Velocity template is straightforward: variables are referenced using ${} notation. The expression between the curly braces references a property, or a property of a property on the weather variable, which was passed to this template by the WeatherController.

The HistoryController is used to retrieve recent forecasts that have been requested by the WeatherController. Whenever we retrieve a forecast from the WeatherController, that controller saves the Weather object to the database via the WeatherDAO. WeatherDAO then uses Hibernate to dissect the Weather object into a series of rows in a set of related database tables. The HistoryController is shown in simple-web HistoryController.

simple-web HistoryController

```
package org.sonatype.mavenbook.web;

import java.util.*;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import org.sonatype.mavenbook.weather.model.*;
import org.sonatype.mavenbook.weather.persist.*;

public class HistoryController implements Controller {

  private LocationDAO locationDAO;
  private WeatherDAO weatherDAO;

  public ModelAndView handleRequest(HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    String zip = request.getParameter("zip");
    Location location = locationDAO.findByZip(zip);
    List<Weather> weathers = weatherDAO.recentForLocation( location );

    Map<String,Object> model = new HashMap<String,Object>();
    model.put( "location", location );
    model.put( "weathers", weathers );

    return new ModelAndView("history", model);
  }

  public WeatherDAO getWeatherDAO() {
    return weatherDAO;
  }

  public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
  }

  public LocationDAO getLocationDAO() {
    return locationDAO;
  }

  public void setLocationDAO(LocationDAO locationDAO) {
    this.locationDAO = locationDAO;
  }
}
```

The `HistoryController` is wired to two DAO objects defined in `simple-persist`. The DAOs are bean properties of the `HistoryController`: `WeatherDAO` and `LocationDAO`. The goal of the `HistoryController` is to retrieve a `List` of `Weather` objects which correspond to the `zip` parameter. When the `WeatherDAO` saves the `Weather` object to the database, it doesn't just store the zip code, it stores a `Location` object which is related to the `Weather` object in the `simple-model`. To retrieve a `List` of `Weather` objects, the `HistoryController` first retrieves the `Location` object that corresponds to the `zip` parameter. It does this by invoking the `findByZip()` method on `LocationDAO`.

Once the `Location` object has been retrieved, the `HistoryController` will then attempt to retrieve recent `Weather` objects that match the given `Location`. Once the `List<Weather>` has been retrieved, a `HashMap` is created to hold two variables for the `history.vm` Velocity template shown in history.vm Rendered by the HistoryController.

history.vm Rendered by the HistoryController

```
<b>
Weather History for: ${location.city}, ${location.region}, ${location. ↩
    country}
</b>
<br/>

#foreach( $weather in $weathers )
<ul>
<li>Temperature: $weather.condition.temp</li>
<li>Condition: $weather.condition.text</li>
<li>Humidity: $weather.atmosphere.humidity</li>
<li>Wind Chill: $weather.wind.chill</li>
<li>Date: $weather.date</li>
</ul>
#end
```

The `history.vm` template in `src/main/webapp/WEB-INF/vm` references the `location` variable to print out information about the location of the forecasts retrieved from the `WeatherDAO`. This template then uses a Velocity control structure, `#foreach`, to loop through each element in the `weathers` variable. Each element in `weathers` is assigned to a variable named `weather` and the template between `#foreach` and `#end` is rendered for each observation.

You’ve seen these `Controller` implementations, and you’ve seen that they reference other beans defined in `simple-weather` and `simple-persist`, they respond to HTTP requests, and they yield control to some mysterious templating system that knows how to render Velocity templates. All of this magic is configured in a Spring application context in `src/main/webapp/WEB-INF/weather-servlet.xml`. This XML configures the controllers and references other Spring-managed beans. It is loaded by a `ServletContextListener` which is also configured to load the `applicationContext-weather.xml` and `applicationContext-persist.xml` from the classpath. Let’s take a closer look at the `weather-servlet.xml` shown in Spring Controller Configuration weather-servlet.xml.

Spring Controller Configuration weather-servlet.xml

```
<beans>
  <bean id="weatherController" ❶
        class="org.sonatype.mavenbook.web.WeatherController">
    <property name="weatherService" ref="weatherService"/>
    <property name="weatherDAO" ref="weatherDAO"/>
  </bean>

  <bean id="historyController"
        class="org.sonatype.mavenbook.web.HistoryController">
    <property name="weatherDAO" ref="weatherDAO"/>
    <property name="locationDAO" ref="locationDAO"/>
  </bean>

  <!-- you can have more than one handler defined -->
  <bean id="urlMapping"
        class="org.springframework.web.servlet.handler.
          SimpleUrlHandlerMapping">
    <property name="urlMap">
      <map>
        <entry key="/weather.x"> ❷
          <ref bean="weatherController" />
        </entry>
        <entry key="/history.x">
          <ref bean="historyController" />
        </entry>
      </map>
    </property>
  </bean>

  <bean id="velocityConfig" ❸
    class="org.springframework.web.servlet.view.velocity.
      VelocityConfigurer">
    <property name="resourceLoaderPath" value="/WEB-INF/vm/"/>
  </bean>

  <bean id="viewResolver" ❹
```

```
    class="org.springframework.web.servlet.view.velocity.
      VelocityViewResolver">
    <property name="cache" value="true"/>
    <property name="prefix" value=""/>
    <property name="suffix" value=".vm"/>
    <property name="exposeSpringMacroHelpers" value="true"/>
  </bean>
</beans>
```

1. The `weather-servlet.xml` defines the two controllers as Spring-managed beans. `weatherController` has two properties which are references to `weatherService` and `weatherDAO`. `historyController` references the beans `weatherDAO` and `locationDAO`. When this `ApplicationContext` is created, it is created in an environment that has access to the `ApplicationContexts` defined in both `simple-persist` and `simple-weather`. In web.xml for simple-webapp you will see how Spring is configured to merge components from multiple Spring configuration files.

2. The `urlMapping` bean defines the URL patterns which invoke the `WeatherController` and the `HistoryController`. In this example, we are using the `SimpleUrlHandlerMapping` and mapping `/weather.x` to `WeatherController` and `/history.x` to `HistoryController`.

3. Since we are using the Velocity templating engine, we will need to pass in some configuration options. In the `velocityConfig` bean, we are telling Velocity to look for all templates in the `/WEB-INF/vm` directory.

4. Last, the `viewResolver` is configured with the class `VelocityViewResolver`. There are a number of `ViewResolver` implementations in Spring from a standard ViewResolver to render JSP or JSTL pages to a resolver which can render Freemarker templates. In this example, we're configuring the Velocity templating engine and setting the default prefix and suffix which will be automatically appended to the names of the template passed to `ModelAndView`.

Finally, the `simple-webapp` project was a `web.xml` which provides the basic configuration for the web application. The `web.xml` file is shown in web.xml for simple-webapp.

web.xml for simple-webapp

```
<web-app id="simple-webapp" version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
               http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>Simple Web Application</display-name>
  <context-param> ❶
    <param-name>contextConfigLocation</param-name>
      <param-value>
        classpath:applicationContext-weather.xml
        classpath:applicationContext-persist.xml
      </param-value>
  </context-param>

  <context-param> ❷
    <param-name>log4jConfigLocation</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
  </context-param>

  <listener> ❸
    <listener-class>
      org.springframework.web.util.Log4jConfigListener
    </listener-class>
  </listener>

  <listener>
    <listener-class> ❹
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet> ❺
    <servlet-name>weather</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping> ❻
    <servlet-name>weather</servlet-name>
    <url-pattern>*.x</url-pattern>
  </servlet-mapping>
</web-app>
```

❶ Here's a bit of magic which allows us to reuse the `applicationContext-weather.xml` and `applicationContext-persist.xml` in this project. The `contextConfigLocation` is used by the `ContextLoaderListener` to create an `ApplicationContext`. When the weather servlet is created, the `weather-servlet.xml` from Spring Controller Configuration weather-servlet.xml is going to be evaluated with the `ApplicationContext` created from this `contextConfigLocation`. In this way, you can define a set of beans in another project and you can reference these beans via the classpath. Since the `simple-persist` and `simple-`

`weather` JARs are going to be in `WEB-INF/lib`, all we do is use the `classpath:` prefix to reference these files. (Another option would have been to copy these files to `/WEB-INF` and reference them with something like `/WEB-INF/applicationContext-persist.xml`.)

❷ The `log4jConfigLocation` is used to tell the `Log4JConfigListener` where to look for Log4J logging configuration. In this example, we tell Log4J to look in `/WEB-INF/log4j.properties`.

❸ This makes sure that the Log4J system is configured when the web application starts. It is important to put this `Log4JConfigListener` before the `ContextLoaderListener`; otherwise, you may miss important logging messages which point to a problem preventing application startup. If you have a particularly large set of beans managed by Spring, and one of them happens to blow up on application startup, your application will fail. If you have logging initialized before Spring starts, you might have a chance to catch a warning or an error. If you don't have logging initialized before Spring starts up, you'll have no idea why your application refuses to start.

❹ The `ContextLoaderListener` is essentially the Spring container. When the application starts, this listener will build an `ApplicationContext` from the `contextConfigLocation` parameter.

❺ We define a Spring MVC `DispatcherServlet` with a name of `weather`. This will cause Spring to look for a Spring configuration file in `/WEB-INF/weather-servlet.xml`. You can have as many `DispatcherServlets` as you need; a `DispatcherServlet` can contain one or more Spring MVC `Controller` implementations.

❻ All requests ending in `.x` will be routed to the `weather` servlet. Note that the `.x` extension has no particular meaning; it is an arbitrary choice and you can use whatever URL pattern you like.

7.7 Running the Web Application

To run the web application, you'll first need to build the entire multi-module project and then build the database using the Hibernate3 plugin. First, from the top-level `simple-parent` project directory, run `mvn clean install`:

```
$ mvn clean install
```

Running `mvn clean install` at the top-level of your multi-module project will install all of modules into your local Maven repository. You need to do this before building the database from the `simple-webapp` project.

Warning
This plugin version requires Java 6 to work.

To build the database from the `simple-webapp` project, run the following from the `simple-webapp` project's directory:

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] -----------------------------------------------------
[INFO] Building Multi-Spring Chapter Simple Web Application
[INFO]task-segment: [hibernate3:hbm2ddl]
[INFO] -----------------------------------------------------
[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151  INFO org.hibernate.tool.hbm2ddl.SchemaExport - export  ↩
    complete
[INFO] -----------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] -----------------------------------------------------
```

Once you've done this, there should be a `${basedir}/data` directory which will contain the HSQLDB database. You can then start the web application with:

```
$ mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] -----------------------------------------------------
[INFO] Building Multi-Spring Chapter Simple Web Application
[INFO]task-segment: [jetty:run]
[INFO] -----------------------------------------------------
[INFO] Preparing jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project:
Multi-Spring Chapter Simple Web Application
...
[INFO] Context path = /simple-webapp
[INFO] Tmp directory =  determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] Starting jetty 6.1.7 ...
2008-03-25 10:28:03.639::INFO:  jetty-6.1.7
```

```
...
2147 INFO  DispatcherServlet  - FrameworkServlet ’weather’: \
initialization completed in 1654 ms
2008-03-25 10:28:06.341::INFO:  Started SelectChannelConnector@0 ↩
    .0.0.0:8080
[INFO] Started Jetty Server
```

Once Jetty is started, you can load http://localhost:8080/simple-webapp/weather.x?zip=60202 and you should see the weather for Evanston, IL in your web browser. Change the ZIP code and you should be able to get your own weather report.

```
Current Weather Conditions for: Evanston, IL, US

* Temperature: 42
* Condition: Partly Cloudy
* Humidity: 55
* Wind Chill: 34
* Date: Tue Mar 25 10:29:45 CDT 2008
```

7.8 The Simple Command Module

The `simple-command` project is a command-line version of the `simple-webapp`. It is a utility that relies on the same dependencies: `simple-persist` and `simple-weather`. Instead of interacting with this application via a web browser, you would run the `simple-command` utility from the command line.

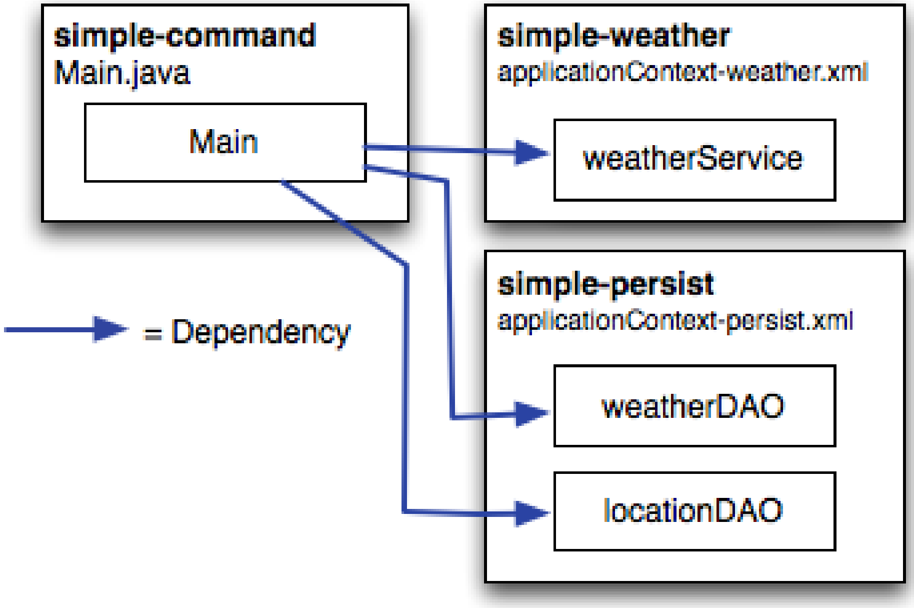

Figure 7.4: Command Line Application Referencing simple-weather and simple-persist

POM for simple-command

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-command</artifactId>
  <packaging>jar</packaging>
  <name>Simple Command Line Tool</name>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
```

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <testFailureIgnore>true</testFailureIgnore>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation>
            </component>
          </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-persist</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>1.8.0.7</version>
    </dependency>
  </dependencies>
</project>
```

This POM creates a JAR file which will contain the `org.sonatype.mavenbook.weather.Main` class shown in The Main Class for simple-command. In this POM we configure the Maven Assembly plugin to use a built-in assembly descriptor named `jar-with-dependencies` which creates a single JAR file containing all the bytecode a project needs to execute, including the bytecode from the project you are building and all the bytecode from libraries your project depends upons.

The Main Class for simple-command

```
package org.sonatype.mavenbook.weather;

import java.util.List;

import org.apache.log4j.PropertyConfigurator;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.LocationDAO;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;

public class Main {

    private WeatherService weatherService;
    private WeatherDAO weatherDAO;
    private LocationDAO locationDAO;
    public static void main(String[] args) throws Exception {
        // Configure Log4J
        PropertyConfigurator.configure(
          Main.class.getClassLoader().getResource("log4j.properties"));

        // Read the zip code from the Command-line
        // (if none supplied, use 60202)
        String zipcode = "60202";
        try {
            zipcode = args[0];
        } catch (Exception e) {
        }

        // Read the Operation from the Command-line
        // (if none supplied use weather)
        String operation = "weather";
        try {
            operation = args[1];
        } catch (Exception e) {
        }

        // Start the program
        Main main = new Main(zipcode);

        ApplicationContext context =
          new ClassPathXmlApplicationContext(
            new String[] { "classpath:applicationContext-weather.xml",
              "classpath:applicationContext-persist.xml" });
        main.weatherService =
          (WeatherService) context.getBean("weatherService");
        main.locationDAO = (LocationDAO) context.getBean("locationDAO");
        main.weatherDAO = (WeatherDAO) context.getBean("weatherDAO");
        if( operation.equals("weather")) {
            main.getWeather();
        } else {
            main.getHistory();
        }
    }

    private String zip;

    public Main(String zip) {
        this.zip = zip;
    }

    public void getWeather() throws Exception {
        Weather weather = weatherService.retrieveForecast(zip);
        weatherDAO.save( weather );
        System.out.print(new WeatherFormatter().formatWeather(weather));
    }

    public void getHistory() throws Exception {
        Location location = locationDAO.findByZip(zip);
        List<Weather> weathers = weatherDAO.recentForLocation(location);
        System.out.print(
          new WeatherFormatter().formatHistory(location, weathers));
    }
}
```

The `Main` class has a reference to `WeatherDAO`, `LocationDAO`, and `WeatherService`. The static `main()` method in this class:

- Reads the zip code from the first command line argument
- Reads the operation from the second command line argument. If the operation is "weather", the latest weather will be retrieved from the web service. If the operation is "history", the program will fetch historical weather records from the local database.
- Loads a Spring `ApplicationContext` using two XML files loaded from `simple-persist` and `simple-weather`
- Creates an instance of `Main`
- Populates the `weatherService`, `weatherDAO`, and `locationDAO` with beans from the Spring `ApplicationContext`
- Runs the appropriate method `getWeather()` or `getHistory()`, depending on the specified operation

In the web application we use Spring `VelocityViewResolver` to render a Velocity template. In the stand-alone implementation, we need to write a simple class which renders our weather data with a Velocity template. WeatherFormatter Renders Weather Data using a Velocity Template is a listing of the `WeatherFormatter`, a class with two methods that render the weather report and the weather history.

WeatherFormatter Renders Weather Data using a Velocity Template

```
package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;
import java.util.List;
import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherFormatter {

    private static Logger log = Logger.getLogger(WeatherFormatter.class);

    public String formatWeather(Weather weather) throws Exception {
        log.info( "Formatting Weather Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader().
                                   getResourceAsStream("weather.vm"));
        VelocityContext context = new VelocityContext();
        context.put("weather", weather );
        StringWriter writer = new StringWriter();
        Velocity.evaluate(context, writer, "", reader);
        return writer.toString();
    }

    public String formatHistory(Location location, List<Weather> weathers)
        throws Exception {
        log.info( "Formatting History Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader().
                                   getResourceAsStream("history.vm"));
        VelocityContext context = new VelocityContext();
        context.put("location", location );
        context.put("weathers", weathers );
        StringWriter writer = new StringWriter();
        Velocity.evaluate(context, writer, "", reader);
        return writer.toString();
    }
}
```

The `weather.vm` template simply prints the zip code's city, country, and region as well as the current temperature. The `history.vm` template prints the location and then iterates through the weather records stored in the local database. Both of these templates are in `${basedir}/src/main/resources`.

The weather.vm Velocity Template

```
****************************************
Current Weather Conditions for:
${weather.location.city},
${weather.location.region},
${weather.location.country}
****************************************

* Temperature: ${weather.condition.temp}
* Condition: ${weather.condition.text}
* Humidity: ${weather.atmosphere.humidity}
* Wind Chill: ${weather.wind.chill}
* Date: ${weather.date}
```

The history.vm Velocity Template

```
Weather History for:
${location.city},
${location.region},
${location.country}


#foreach( $weather in $weathers )
****************************************
* Temperature: $weather.condition.temp
* Condition: $weather.condition.text
* Humidity: $weather.atmosphere.humidity
* Wind Chill: $weather.wind.chill
* Date: $weather.date
#end
```

7.9 Running the Simple Command

The `simple-command` project is configured to create a single JAR containing the bytecode of the project and all of the bytecode from the dependencies. To create this assembly, run the `assembly` goal of the Maven Assembly plugin from the `simple-command` project directory:

```
$ mvn assembly:assembly
[INFO] -----------------------------------------------------
[INFO] Building Multi-spring Chapter Simple Command Line Tool
[INFO]task-segment: [assembly:assembly] (aggregator-style)
[INFO] -----------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
```

```
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
...
[INFO] [jar:jar]
[INFO] Building jar: .../simple-parent/simple-command/target/simple-command.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Building jar: .../simple-parent/simple-command/target
/simple-command-jar-with-dependencies.jar
```

The build progresses through the lifecycle compiling bytecode, running tests, and finally building a JAR for the project. Then the `assembly:assembly` goal creates a JAR with dependencies by unpacking all of the dependencies to temporary directories and then collecting all of the bytecode into a single JAR in `target/` named `simple-command-jar-with-dependencies.jar`. This "uber" JAR weighs in at 15 MB.

Before you run the command-line tool, you will need to invoke the `hbm2ddl` goal of the Hibernate3 plugin to create the HSQLDB database. Do this by running the following command from the `simple-command` directory:

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] ------------------------------------------------------
[INFO] Building Multi-spring Chapter Simple Command Line Tool
[INFO]task-segment: [hibernate3:hbm2ddl]
[INFO] ------------------------------------------------------
[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151  INFO org.hibernate.tool.hbm2ddl.SchemaExport - export  complete
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------
```

Once you run this, you should see a `data` directory under `simple-command`. This `data` directory holds the HSQLDB database. To run the command-line weather forecaster, run the following from the `simple-command` project directory:

```
$ java -cp target/simple-command-jar-with-dependencies.jar \
       org.sonatype.mavenbook.weather.Main 60202
2321 INFO  YahooRetriever  - Retrieving Weather Data
```

```
2489 INFO  YahooParser  - Creating XML Reader
2581 INFO  YahooParser  - Parsing XML Response
2875 INFO  WeatherFormatter  - Formatting Weather Data
****************************************
Current Weather Conditions for:
Evanston,
IL,
US
****************************************

* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: Wed Aug 06 09:35:30 CDT 2008
```

To run a history query, execute the following command:

```
$ java -cp target/simple-command-jar-with-dependencies.jar \
       org.sonatype.mavenbook.weather.Main 60202 history
2470 INFO  WeatherFormatter  - Formatting History Data
Weather History for:
Evanston, IL, US

****************************************
* Temperature: 39
* Condition: Heavy Rain
* Humidity: 93
* Wind Chill: 36
* Date: 2007-12-02 13:45:27.187
****************************************
* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: 2008-08-06 09:24:11.725
****************************************
* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: 2008-08-06 09:27:28.475
```

7.10 Conclusion

We've spent a great deal of time on topics not directly related to Maven to get this far. We've done this to present a complete and meaningful example project which you can use to implement real-world systems. We didn't take any shortcuts to produce slick, canned results quickly, and we're not going to dazzle you with some Ruby on Rails-esque wizardry and lead you to believe that you can create a finished Java Enterprise application in "10 easy minutes!" There's too much of this in the market; there are too many people trying to sell you the easiest framework that requires zero investment of time or attention. What we're trying to do in this chapter is present the entire picture, the entire ecosystem of a multi-module build. What we've done is present Maven in the context of a application which resembles something you could see in the wild—not the fast-food, 10 minute screen-cast that slings mud at Apache Ant and tries to convince you to adopt Apache Maven.

If you walk away from this chapter wondering what it has to do with Maven, we've succeeded. We present a complex set of projects, using popular frameworks, and we tie them together using declarative builds. The fact that more than 60% of this chapter was spent explaining Spring and Hibernate should tell you that Maven, for the most part, stepped out of the way. It worked. It allowed us to focus on the application itself, not on the build process. Instead of spending time discussing Maven, and the work you would have to do to "build a build" that integrated with Spring and Hibernate, we talked almost exclusively about the technologies used in this contrived project. If you start to use Maven, and you take the time to learn it, you really do start to benefit from the fact that you don't have to spend time coding up some procedural build script. You don't have to spend your time worrying about mundane aspects of your build.

You can use the skeleton project introduced in this chapter as the foundation for your own, and chances are that when you do, you'll find yourself creating more and more modules as you need them. For example, the project on which this chapter was based has two distinct model projects, two persistence projects which persist to dramatically different databases, several web applications, and a Java mobile application. In total, the real world system I based this on contains at least 15 interrelated modules. The point is that you've seen the most complex multi-module example we're going to include in this book, but you should also know that this example just scratches the surface of what is possible with Maven.

7.10.1 Programming to Interface Projects

This chapter explored a multi-module project which was more complex than the simple example presented in Chapter 6, yet it was still a simplification of a real-world project. In a larger project, you might find yourself building a system resembling Figure 7.5.

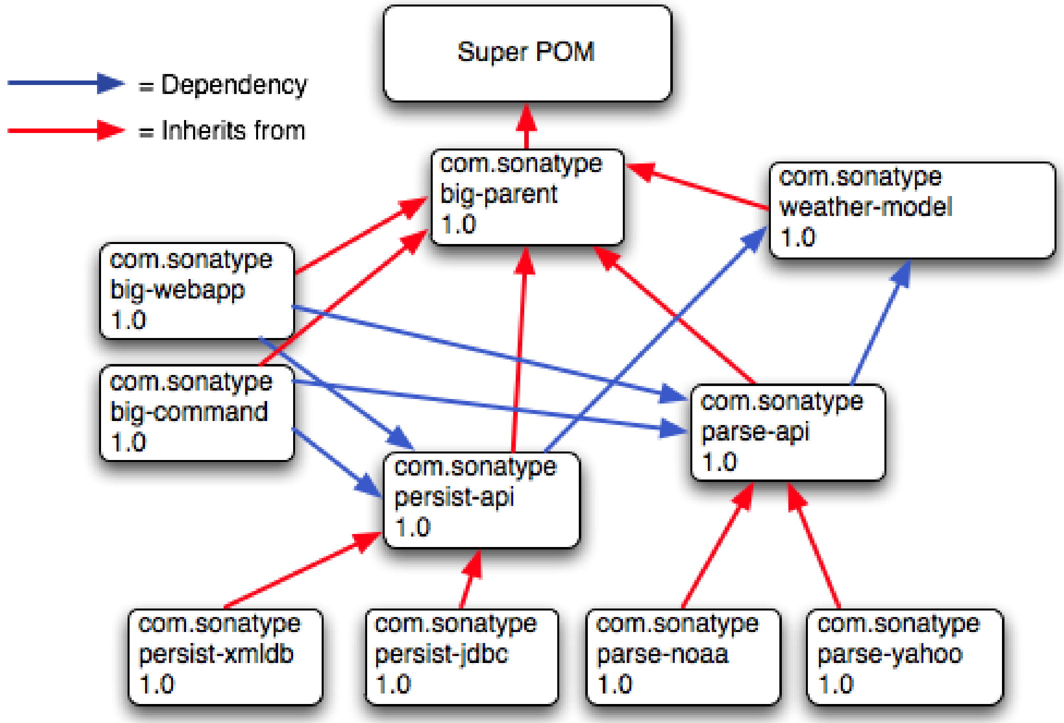
= Dependency
= Inherits from
Super POM
com.sonatype big-parent 1.0
com.sonatype weather-model 1.0
com.sonatype big-webapp 1.0
com.sonatype big-command 1.0
com.sonatype parse-api 1.0
com.sonatype persist-api 1.0
com.sonatype persist-xmldb 1.0
com.sonatype persist-jdbc 1.0
com.sonatype parse-noaa 1.0
com.sonatype parse-yahoo 1.0

Figure 7.5: Programming to Interface Projects

When we use the term *interface project* we are referring to a Maven project which contains interfaces and constants only. In Figure 7.5 the interface projects would be `persist-api` and `parse-api`. If `big-command` and `big-webapp` are written to the interfaces defined in `persist-api`, then it is very easy to just swap in another implementation of the persistence library. This particular diagram shows two implementations of the `persist-api` project, one which stores data in an XML database, and the other which stores data in a relational database. If you use some of the concepts in this chapter, you can see how you could just pass in a flag to the program that swaps in a different Spring application context XML file to swap out data sources of persistence implementations. Just like the OO design of the application itself, it is often wise to separate the interfaces of an API from the implementation of the API into separate Maven projects.

Chapter 8

Optimizing and Refactoring POMs

8.1 Introduction

In Chapter 7, we showed how many pieces of Maven come together to produce a fully functional multi-module build. Although the example from that chapter suggests a real application—one that interacts with a database, a web service, and that itself presents two interfaces: one in a web application, and one on the command line—that example project is still contrived. To present the complexity of a real project would require a book far larger than the one you are now reading. Real-life applications evolve over years and are often maintained by large, diverse groups of developers, each with a different focus. In a real-world project, you are often evaluating decisions and designs made and created by others. In this chapter, we take a step back from the examples you've seen in the previous chapters, and we ask ourselves if there are any optimizations that might make more sense given what we now know about Maven. Maven is a very capable tool that can be as simple or as complex as you need it to be. Because of this, there are often a million ways to accomplish the same task, and there is often no one "right" way to configure your Maven project.

Don't misinterpret that last sentence as a license to go off and ask Maven to do something it wasn't designed for. While Maven allows for a diversity of approach, there is certainly "A Maven Way", and you'll be more productive using Maven as it was designed to be used. All this chapter is trying to do is communicate some of the optimizations you can perform on an existing Maven project. Why didn't we just introduce an optimized POM in the first place? Designing POMs for pedagogy is a very different requirement from designing POMs for efficiency. While it is certainly much easier to define a certain setting in your `~/.m2/settings.xml` than to declare a profile in a `pom.xml`, writing a book, and reading a book is mostly about pacing and making sure we're not introducing concepts before you are ready. In the previous chapters, we've made an effort not to overwhelm the reader with too much information, and, in doing so, we've skipped some core concepts like the `dependencyManagement` element introduced in this chapter.

There are many instances in the previous chapters when the authors of this book took a shortcut or glossed over an important detail to shuffle you along to the main point of a specific chapter. You learned how to create a Maven project, and you compiled and installed it without having to wade through hundreds of pages introducing every last switch and dial available to you. We've done this because we believe it is important to deliver the new Maven user to a result faster rather than meandering our way through a very long, seemingly interminable story. Once you've started to use Maven, you should know how to analyze your own projects and POMs. In this chapter, we take a step back and look at what we are left with after the example from Chapter 7.

8.2 POM Cleanup

Optimizing a multimodule project's POM is best done in several passes, as there are many areas to focus on. In general, we are looking for repetition within a POM and across the sibling POMs. When you are starting out, or when a project is still evolving rapidly, it is acceptable to duplicate some dependencies and plugin configurations here and there, but as the project matures and as the number of modules increases, you will want to take some time to refactor common dependencies and configuration points. Making your POMs more efficient will go a long way to helping you manage complexity as your project grows. Whenever there is duplication of some piece of information, there is usually a better way.

8.3 Optimizing Dependencies

If you look through the various POMs you notice a lot of duplication that you can remove by moving parts into a parent POM.

Just as in your project's source code, any time you have duplication in your POMs, you open the door a bit for trouble down the road. Duplicated dependency declarations make it difficult to ensure consistent versions across a large project. When you only have two or three modules, this might not be a primary issue, but when your organization is using a large, multimodule Maven build to manage hundreds of components across multiple departments, one single mismatch between dependencies can cause chaos and confusion. A simple version mismatch in a project's dependency on a bytecode manipulation package called ASM three levels deep in the project hierarchy could throw a wrench into a web application maintained by a completely different group of developers who depend on that particular module. Unit tests could pass because they are being run with one version of a dependency, but they could fail disastrously in production where the bundle (WAR, in this case) was packaged up with a different version. If you have tens of projects using something like Hibernate Annotations, each repeating and duplicating the dependencies and exclusions, the mean time between someone screwing up a build is going to be very short. As your Maven projects become more complex, your dependency lists are going to grow, and you are going to want to consolidate versions and dependency declarations in parent POMs.

The duplication of the sibling module versions can introduce a particularly nasty problem that is not directly caused by Maven and is learned only after you've been bitten by this bug a few times. If you use the Maven Release plugin to perform your releases, all these sibling dependency versions will be updated automatically for you, so maintaining them is not the concern. If `simple-web` version `1.3-SNAPSHOT` depends on `simple-persist` version `1.3-SNAPSHOT`, and if you are performing a release of the 1.3 version of both projects, the Maven Release plugin is smart enough to change the versions throughout your multimodule project's POMs automatically. Running the release with the Release plugin will automatically increment all of the versions in your build to `1.4-SNAPSHOT`, and the release plugin will commit the code change to the repository. Releasing a huge multimodule project couldn't be easier, until...

Problems occur when developers merge changes to the POM and interfere with a release that is in progress. Often a developer merges and occasionally mishandles the conflict on the sibling dependency, inadvertently reverting that version to a previous release. Since the consecutive versions of the dependency are often compatible, it does not show up when the developer builds, and won't show up in any continuous integration build system as a failed build. Imagine a very complex build where the trunk is full of components at `1.4-SNAPSHOT`, and now imagine that Developer A has updated Component A deep within the project's hierarchy to depend on version `1.3-SNAPSHOT` of Component B. Even though most developers have `1.4-SNAPSHOT`, the build succeeds if version `1.3-SNAPSHOT` and `1.4-SNAPSHOT` of Component B are compatible. Maven continues to build the project using the `1.3-SNAPSHOT` version of Component B from the developer's local repositories. Everything seems to be going quite smoothly—the project builds, the continuous integration build works fine, and so on. Someone might have a mystifying bug related to Component B, but she chalks it up to malevolent gremlins and moves on. Meanwhile, a pump in the reactor room is steadily building up pressure, until something blows....

Someone, let's call them Mr. Inadvertent, had a merge conflict in component A, and mistakenly pegged component A's dependency on component B to `1.3-SNAPSHOT` while the rest of the project marches on. A bunch of developers have been trying to fix a bug in component B all this time and they've been mystified as to why they can't seem to fix the bug in production. Eventually someone looks at component A and realizes that the dependency is pointing to the wrong version. Hopefully, the bug wasn't large enough to cost money or lives, but Mr. Inadvertent feels stupid and people tend to trust him a little less than they did before the whole sibling dependency screw-up. (Hopefully, Mr. Inadvertent realizes that this was user error and not Maven's fault, but more than likely he starts an awful blog and complains about Maven endlessly to make himself feel better.)

Fortunately, dependency duplication and sibling dependency mismatch are easily preventable if you make some small changes. The first thing we're going to do is find all the dependencies used in more than one project and move them up to the parent POM's dependencyManagement section. We'll leave out the sibling dependencies for now. The `simple-parent` pom now contains the following:

```
<project>
    ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring</artifactId>
                <version>2.0.7</version>
            </dependency>
            <dependency>
                <groupId>org.apache.velocity</groupId>
                <artifactId>velocity</artifactId>
                <version>1.5</version>
            </dependency>
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-annotations</artifactId>
                <version>3.3.0.ga</version>
            </dependency>
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-commons-annotations</artifactId>
                <version>3.3.0.ga</version>
            </dependency>
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate</artifactId>
                <version>3.2.5.ga</version>
                <exclusions>
                    <exclusion>
                        <groupId>javax.transaction</groupId>
                        <artifactId>jta</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
        </dependencies>
    </dependencyManagement>
    ...
</project>
```

Once these are moved up, we need to remove the versions for these dependencies from each of the POMs; otherwise, they will override the dependencyManagement defined in the parent project. Let's look at only `simple-model` for brevity's sake:

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-annotations</artifactId>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate</artifactId>
        </dependency>
    </dependencies>
    ...
</project>
```

The next thing we should do is fix the replication of the `hibernate-annotations` and `hibernate-commons-annotations` version since these should match. We'll do this by creating a property called `hibernate.annotations.version`. The resulting `simple-parent` section looks like this:

```
<project>
    ...
  <properties>
    <hibernate.annotations.version>3.3.0.ga
      </hibernate.annotations.version>
  </properties>

  <dependencyManagement>
    ...
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    ...
  </dependencyManagement>
  ...
</project>
```

The last issue we have to resolve is with the sibling dependencies and define the versions of sibling projects in the top-level parent project. This is certainly a valid approach, but we can also solve the version problem just by using two built-in properties — `${project.groupId}` and `${project.version}`. Since they are sibling dependencies, there is not much value to be gained by enumerating them in the parent, so we'll rely on the built-in `${project.version}` property. Because they all share the same group, we can further future-proof these declarations by referring to the current POM's group using the built-in `${project.groupId}` property. The `simple-command` dependency section now looks like this:

```
<project>
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>simple-weather</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>simple-persist</artifactId>
            <version>${project.version}</version>
        </dependency>
        ...
    </dependencies>
    ...
</project>
```

Here's a summary of the two optimizations we completed that reduce duplication of dependencies:

Pull-up common dependencies to `dependencyManagement`

If more than one project depends on a specific dependency, you can list the dependency in `dependencyManagement`. The parent POM can contain a version and a set of exclusions; all the child POM needs to do to reference this dependency is use the `groupId` and `artifactId`. Child projects can omit the version and exclusions if the dependency is listed in `dependencyManagement`.

Use built-in project `version` and `groupId` for sibling projects

Use `${project.version}` and `${project.groupId}` when referring to a sibling project. Sibling projects almost always share the same `groupId`, and they almost always share the same release version. Using `${project.version}` will help you avoid the sibling version mismatch problem discussed previously.

8.4 Optimizing Plugins

If we take a look at the various plugin configurations, we can see the HSQLDB dependencies duplicated in several places. Unfortunately, dependencyManagement doesn't apply to plugin dependencies, but we can still use a property to consolidate the versions. Most complex Maven multimodule projects tend to define all versions in the top-level POM. This top-level POM then becomes a focal point for changes that affect the entire project. Think of version numbers as string literals in a Java class; if you are constantly repeating a literal, you'll likely want to make it a variable so that when it needs to be changed, you have to change it in only one place. Rolling up the version of HSQLDB into a property in the top-level POM yields the following properties element:

```
<project>
  ...
  <properties>
    <hibernate.annotations.version>3.3.0.ga
      </hibernate.annotations.version>
    <hsqldb.version>1.8.0.7</hsqldb.version>
  </properties>
  ...
</project>
```

The next thing we notice is that the hibernate3-maven-plugin configuration is duplicated in the simple-webapp and simple-command modules. We can manage the plugin configuration in the top-level POM just as we managed the dependencies in the top-level POM with the dependencyManagement section. To do this, we use the pluginManagement element in the top-level POM's build element:

```
<project>
  ...
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.5</source>
            <target>1.5</target>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>hibernate3-maven-plugin</artifactId>
          <version>2.1</version>
          <configuration>
            <components>
```

```
              <component>
                <name>hbm2ddl</name>
                <implementation>annotationconfiguration</implementation>
              </component>
            </components>
          </configuration>
          <dependencies>
            <dependency>
              <groupId>hsqldb</groupId>
              <artifactId>hsqldb</artifactId>
              <version>${hsqldb.version}</version>
            </dependency>
          </dependencies>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
  ...
</project>
```

8.5 Optimizing with the Maven Dependency Plugin

On larger projects, additional dependencies often tend to creep into a POM as the number of dependencies grow. As dependencies change, you are often left with dependencies that are not being used, and just as often, you may forget to declare explicit dependencies for libraries you require. Because Maven 2.x includes transitive dependencies in the compile scope, your project may compile properly but fail to run in production. Consider a case where a project uses classes from a widely used project such as Jakarta Commons BeanUtils. Instead of declaring an explicit dependency on BeanUtils, your project simply relies on a project like Hibernate that references BeanUtils as a transitive dependency. Your project may compile successfully and run just fine, but if you upgrade to a new version of Hibernate that doesn't depend on BeanUtils, you'll start to get compile and runtime errors, and it won't be immediately obvious why your project stopped compiling. Also, because you haven't explicitly listed a dependency version, Maven cannot resolve any version conflicts that may arise.

A good rule of thumb in Maven is to always declare explicit dependencies for classes referenced in your code. If you are going to be importing Commons BeanUtils classes, you should also be declaring a direct dependency on Commons BeanUtils. Fortunately, via bytecode analysis, the Maven Dependency plugin is able to assist you in uncovering direct references to dependencies. Using the updated POMs we previously optimized, let's look to see if any errors pop up:

```
$ mvn dependency:analyze
[INFO] Scanning for projects...
```

```
[INFO] Reactor build order:
[INFO]   Chapter 8 Simple Parent Project
[INFO]   Chapter 8 Simple Object Model
[INFO]   Chapter 8 Simple Weather API
[INFO]   Chapter 8 Simple Persistence API
[INFO]   Chapter 8 Simple Command Line Tool
[INFO]   Chapter 8 Simple Web Application
[INFO]   Chapter 8 Parent Project
[INFO] Searching repository for plugin with prefix: ’dependency’.

...

[INFO] -----------------------------------------------------
[INFO] Building Chapter 8 Simple Object Model
[INFO]task-segment: [dependency:analyze]
[INFO] -----------------------------------------------------
[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]javax.persistence:persistence-api:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[WARNING]org.hibernate:hibernate:jar:3.2.5.ga:compile
[WARNING]junit:junit:jar:3.8.1:test

...

[INFO] -----------------------------------------------------
[INFO] Building Chapter 8 Simple Web Application
[INFO]task-segment: [dependency:analyze]
[INFO] -----------------------------------------------------
[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [dependency:analyze]
```

```
[WARNING] Used undeclared dependencies found:
[WARNING]org.sonatype.mavenbook.optimize:simple-model:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]org.apache.velocity:velocity:jar:1.5:compile
[WARNING]javax.servlet:jstl:jar:1.1.2:compile
[WARNING]taglibs:standard:jar:1.1.2:compile
[WARNING]junit:junit:jar:3.8.1:test
```

In the truncated output just shown, you can see the output of the `dependency:analyze` goal. This goal analyzes the project to see whether there are any indirect dependencies, or dependencies that are being referenced but are not directly declared. In the `simple-model` project, the Dependency plugin indicates a “used undeclared dependency” on `javax.persistence:persistence-api`. To investigate further, go to the `simple-model` directory and run the `dependency:tree` goal, which will list all of the project’s direct and transitive dependencies:

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: ’dependency’.
[INFO] -----------------------------------------------------
[INFO] Building Chapter 8 Simple Object Model
[INFO]task-segment: [dependency:tree]
[INFO] -----------------------------------------------------
[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.optimize:simple-model:jar:1.0
[INFO] +- org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[INFO] |  \- javax.persistence:persistence-api:jar:1.0:compile
[INFO] +- org.hibernate:hibernate:jar:3.2.5.ga:compile
[INFO] |  +- net.sf.ehcache:ehcache:jar:1.2.3:compile
[INFO] |  +- commons-logging:commons-logging:jar:1.0.4:compile
[INFO] |  +- asm:asm-attrs:jar:1.5.3:compile
[INFO] |  +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  +- antlr:antlr:jar:2.7.6:compile
[INFO] |  +- cglib:cglib:jar:2.1_3:compile
[INFO] |  +- asm:asm:jar:1.5.3:compile
[INFO] |  \- commons-collections:commons-collections:jar:2.1.1:compile
[INFO] \- junit:junit:jar:3.8.1:test
[INFO] -----------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] -----------------------------------------------------
```

From this output, we can see that the `persistence-api` dependency is coming from `hibernate`. A cursory scan of the source in this module will reveal many `javax.persistence` import statements confirming that we are, indeed, directly referencing this dependency. The simple fix is to add a direct reference to the dependency. In this example, we put the dependency version in `simple-parent`’s `dependencyManagement` section because the dependency is linked to Hibernate, and the Hibernate version is declared here. Eventually you are going to want to upgrade your project’s version of Hibernate.

Listing the persistence-api dependency version near the Hibernate dependency version will make it more obvious later when your team modifies the parent POM to upgrade the Hibernate version.

If you look at the dependency:analyze output from the simple-web module, you will see that we also need to add a direct reference to the simple-model dependency. The code in simple-webapp directly references the model objects in simple-model, and the simple-model is exposed to simple-webapp as a transitive dependency via simple-persist. Since this is a sibling dependency that shares both the version and groupId, the dependency can be defined in simple-webapp's pom.xml using the ${project.groupId} and ${project.version}.

How did the Maven Dependency plugin uncover these issues? How does dependency:analyze know which classes and dependencies are directly referenced by your project's bytecode? The Dependency plugin uses the ObjectWeb ASM (http://asm.objectweb.org/) library to produce a list of "used, undeclared dependencies" dependencies

In contrast, the list of unused, declared dependencies is a little trickier to validate, and less useful than the "used, undeclared dependencies." For one, some dependencies are used only at runtime or for tests, and they won't be found in the bytecode. These are pretty obvious when you see them in the output; for example, JUnit appears in this list, but this is expected because it is used only for unit tests. You'll also notice that the Velocity and Servlet API dependencies are listed in this list for the simple-web module. This is also expected because, although the project doesn't have any direct references to the classes of these artifacts, they are still essential during runtime.

Be careful when removing any unused, declared dependencies unless you have very good test coverage, or you might introduce a runtime error. A more sinister issue pops up with bytecode optimization. For example, it is legal for a compiler to substitute the value of a constant and optimize away the reference. Removing this dependency will cause the compile to fail, yet the tool shows it as unused. Future versions of the Maven Dependency plugin will provide better techniques for detecting and/or ignoring these types of issues.

You should use the dependency:analyze tool periodically to detect these common errors in your projects. It can be configured to fail the build if certain conditions are found, and it is also available as a report.

8.6 Final POMs

As an overview, the final POM files are listed as a reference for this chapter. Final POM for simple-parent shows the top-level POM for simple-parent.

Final POM for simple-parent

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.optimize</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 8 Simple Parent Project</name>

  <modules>
    <module>simple-command</module>
    <module>simple-model</module>
    <module>simple-weather</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.5</source>
            <target>1.5</target>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>hibernate3-maven-plugin</artifactId>
          <version>2.1</version>
          <configuration>
            <components>
              <component>
                <name>hbm2ddl</name>
                <implementation>annotationconfiguration</implementation>
              </component>
            </components>
          </configuration>
          <dependencies>
            <dependency>
              <groupId>hsqldb</groupId>
              <artifactId>hsqldb</artifactId>
              <version>${hsqldb.version}</version>
            </dependency>
          </dependencies>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <properties>
    <hibernate.annotations.version>3.3.0.ga
      </hibernate.annotations.version>
    <hsqldb.version>1.8.0.7</hsqldb.version>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
        <version>2.0.7</version>
      </dependency>
      <dependency>
        <groupId>org.apache.velocity</groupId>
        <artifactId>velocity</artifactId>
        <version>1.5</version>
      </dependency>
      <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>persistence-api</artifactId>
        <version>1.0</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
        <version>${hibernate.annotations.version}</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-commons-annotations</artifactId>
        <version>${hibernate.annotations.version}</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate</artifactId>
        <version>3.2.5.ga</version>
        <exclusions>
          <exclusion>
            <groupId>javax.transaction</groupId>
            <artifactId>jta</artifactId>
          </exclusion>
        </exclusions>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The POM shown in Final POM for simple-command captures the POM for `simple-command`, the command-line version of the tool.

Final POM for simple-command

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
      <groupId>org.sonatype.mavenbook.optimize</groupId>
      <artifactId>simple-parent</artifactId>
      <version>1.0</version>
  </parent>

  <artifactId>simple-command</artifactId>
  <packaging>jar</packaging>
  <name>Chapter 8 Simple Command Line Tool</name>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <configuration>
            <archive>
              <manifest>
                <mainClass>org.sonatype.mavenbook.weather.Main</mainClass>
                <addClasspath>true</addClasspath>
              </manifest>
            </archive>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <configuration>
            <testFailureIgnore>true</testFailureIgnore>
          </configuration>
        </plugin>
        <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
            <descriptorRefs>
              <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-weather</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-persist</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
    </dependency>
  </dependencies>
</project>
```

The POM shown in Final POM for simple-model is the `simple-model` project's POM. The `simple-model` project contains all of the model objects used throughout the application.

Final POM for simple-model

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-model</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Object Model</name>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>persistence-api</artifactId>
    </dependency>
  </dependencies>
</project>
```

The POM shown in Final POM for simple-persist is the `simple-persist` project's POM. The `simple-persist` project contains all of the persistence logic that is implemented using Hibernate.

Final POM for simple-persist

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-persist</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Persistence API</name>

  <dependencies>
      <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>simple-model</artifactId>
        <version>${project.version}</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate</artifactId>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-commons-annotations</artifactId>
      </dependency>
      <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.4</version>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
      </dependency>
  </dependencies>
</project>
```

The POM shown in Final POM for simple-weather is the `simple-weather` project's POM. The `simple-weather` project is the project that contains all of the logic to parse the Yahoo Weather RSS feed. This project depends on the `simple-model` project.

Final POM for simple-weather

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Weather API</name>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.3.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Finally, the POM shown in Final POM for simple-webapp is the `simple-webapp` project's POM. The `simple-webapp` project contains a web application that stores retrieved weather forecasts in an HSQLDB database and that also interacts with the libraries generated by the `simple-weather` project.

Final POM for simple-webapp

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>Chapter 8 Simple Web Application</name>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-weather</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-persist</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.1.2</version>
    </dependency>
    <dependency>
      <groupId>taglibs</groupId>
      <artifactId>standard</artifactId>
      <version>1.1.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <version>6.1.9</version>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>${hsqldb.version}</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

8.7 Conclusion

This chapter has shown you several techniques for improving the control of your dependencies and plugins to ease future maintenance of your builds. We recommend periodically reviewing your builds in this way to ensure that duplication is reduced and kept at a minimum. This will ensure that your build performance does not degrade and you produce high quality outputs.

Chapter 9

Creative Commons License

This work is licensed under a Creative Commons Attribution, Non-commercial, No Derivative Works 3.0 United States license. For more information about this license, see http://creativecommons.org/licenses/by-nc-nd/3.0/us/. You are free to share, copy, distribute, display, and perform the work under the following conditions:

- You must attribute the work to Sonatype, Inc. with a link to http://www.sonatype.com.

If you redistribute this work on a web page, you must include the following link with the URL in the about attribute listed on a single line (remove the backslashes and join all URL parameters):

```
<div xmlns:cc="http://creativecommons.org/ns#"
     about="http://creativecommons.org/license/results-one?q_1=2&q_1=1\
            &field_commercial=n&field_derivatives=n&field_jurisdiction=us\
            &field_format=StillImage&field_worktitle=Repository%3A+\ ↩
                Management\
            &field_attribute_to_name=Sonatype%2C+Inc.\
            &field_attribute_to_url=http%3A%2F%2Fwww.sonatype.com\
            &field_sourceurl=http%3A%2F%2Fwww.sonatype.com%2Fbook\
            &lang=en_US&language=en_US&n_questions=3">
    <a rel="cc:attributionURL" property="cc:attributionName"
       href="http://www.sonatype.com">Sonatype, Inc.</a> /
    <a rel="license"
       href="http://creativecommons.org/licenses/by/3.0/us/">
        CC BY 3.0</a>
</div>
```

When downloaded or distributed in a jurisdiction other than the United States of America, this work shall be covered by the appropriate ported version of Creative Commons Attribution, NC, ND Works 3.0 license for the specific jurisdiction. If the Creative Commons Attribution, NC, ND version 3.0 license is not available for a specific jurisdiction, this work shall be covered under the Creative Commons Attribution, NC, ND version 2.5 license for the jurisdiction in which the work was downloaded or distributed. A comprehensive list of jurisdictions for which a Creative Commons license is available can be found on the Creative Commons International web site at http://creativecommons.org/international.

If no ported version of the Creative Commons license exists for a particular jurisdiction, this work shall be covered by the generic, unported Creative Commons Attribution, NC, ND version 3.0 license available from http://creativecommons.org/licenses/by/3.0/.

Chapter 10

Copyright

Copyright © 2011 Sonatype, Inc.

Online version published by Sonatype, Inc.

This work is licensed under a Creative Commons Attribution, Non-Commercial, No Derivative Works 3.0 United States license. For more information about this license, see http://creativecommons.org/licenses/-by-nc-nd/3.0/us/.

Nexus™, Nexus Professional™, Matrix™, Matrix Professional™, Sonatype Professional™, all Matrix-related logos, and all Nexus-related logos are trademarks or registered trademarks of Sonatype, Inc., in the United States and other countries.

Java™, all Java-based trademarks and logos, Hudson™, and all Hudson-based trademarks and logos are trademarks or registered trademarks of Oracle, Inc., in the United States and other countries.

IBM® and WebSphere® are trademarks or registered trademarks of International Business Machines, Inc., in the United States and other countries.

Eclipse™ is a trademark of the Eclipse Foundation, Inc., in the United States and other countries.

Apache and the Apache feather logo are trademarks of The Apache Software Foundation.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Sonatype, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.