brief contents

PART 1 INTRODUCING GRAILS ................................................................. 1
   1  ■ Grails in a hurry . . . 3
   2  ■ The Groovy essentials 33

PART 2 CORE GRAILS ................................................................. 61
   3  ■ Modeling the domain 63
   4  ■ Creating the initial UI 91
   5  ■ Retrieving the data you need 110
   6  ■ Controlling application flow 136
   7  ■ Services and data binding 155
   8  ■ Developing tasty forms, views, and layouts 189

PART 3 EVERYDAY GRAILS ................................................................. 227
   9  ■ Building reliable applications 229
   10 ■ Using plugins: just add water 261
   11 ■ Protecting your application 297
   12 ■ Exposing your app to other programs 328
13 ■ Single-page web applications (and other UI stuff) 357
14 ■ Understanding Spring and transactions 380

PART 4 ADVANCED GRAILS ........................................................399
15 ■ Understanding events, messaging, and scheduling 401
16 ■ NoSQL and Grails 432
17 ■ Beyond compile, test, run 467
18 ■ Grails in the cloud 496
In this book’s early chapters, you had your first exposure to Grails’s domain classes. Life was simple: you saved, you validated, and you used dynamic finds. You can go a long way with that knowledge—probably all the way to your first few publicly hosted Grails applications. But then you hit the wall. As users flock to your stunning new social networking applications, you wonder how you can tune your queries to run more efficiently. You start to think about query-caching options. You want to refactor your domain classes. And you want to monitor where all your cycles are going. In short, you want to take off the safety harness and get into serious GORM kung fu. When you hit the performance and scalability wall, this is the chapter for you.

If you currently work in enterprise Java, you probably already have legacy databases (perhaps with existing Hibernate mappings). You’ll explore the ins and outs of integrating Grails applications with your existing tables, Java Naming and Directory Interface (JNDI) data sources, and other things your enterprise may have lying
around. But it’s no good understanding the high-level stuff if you don’t have an idea how the underlying engine works, so you’ll peek underneath to see how Grails interacts with Hibernate and how you can tune it to make it all faster.

## 19.1 Domain model kung fu

In chapter 3, you explored the common domain model relationships: 1:1, 1:m, and m:n. But as you develop your Grails applications, you’ll probably come across situations that don’t fit into those three standard relationships. In this section, you explore less common domain modeling options, starting with inheritance.

### 19.1.1 Exploring inheritance options

Relational databases aren’t designed for object-oriented (OO) data, and inheritance relationships don’t map comfortably to a relational design. With that acknowledgment out of the way, what can GORM do about domain classes that use inheritance?

GORM provides two options for handling inheritance: table-per-hierarchy and table-per-subclass approaches.

By default, GORM uses one table per hierarchy, so your base class and its subclasses are stored in the same table.

An example makes this clearer. Let’s modify Hubbub to support a special starred post that will always appear with a star and the reason it was starred. All of a user’s followers will notice these starred posts, marked with a special icon.

You could implement `StarPost` by hacking your existing `Post` object:

```java
boolean starred
String reason
```

But if you did this, starred would be false for nearly every post, and reason would be null, so your constraints would have to be relaxed. Let’s do the proper OO thing, and subclass your `Post` domain class, as shown in the following code.

```java
package com.grailsinaction;

class StarPost extends Post {
  String reason
  static constraints = {
    reason maxSize: 50
  }
}
```

Your `StarPost` is implemented, and it inherits everything from the base `Post` class. You also have to make modifications to your `Tag` class to explicitly mention the new relationship:

```java
static belongsTo = [ User, Post, StarPost ]
```

With your `Tag` class updated and your `StarPost` ready to roll, it’s time to create an integration test to make sure that all the relationships work as expected. The following listing shows a unit test for your new `StarPost`. 
package com.grailsinaction

import grails.plugins.springsecurity.SpringSecurityService
import grails.test.mixin.Mock
import spock.lang.Specification

@Mock([User, Post, StarPost])
class StarPostSpec extends Specification {

    void "Star posts operate like a normal Post"() {
        given: "A basic user"
        def securityService = Mock(SpringSecurityService)
        securityService.encodePassword(_ as String) >> "skfjhasfh"

        def u = new User(loginId: 'glen')
        u.springSecurityService = securityService
        u.setPassword('secret')

        when: "A series of posts and starposts are added"
        u.addToPosts(new Post(content: "First Post"))
        u.addToPosts(new Post(content: "Second Post"))
        u.addToPosts(
            new StarPost(content: "Third Post", reason: "I shine on 3"))
        u.save(failOnError: true)

        then: "Starposts appear in the list of posts"
        Post.countByUser(u) == 3
        StarPost.countByUser(u) == 1
    }
}

When you have domain classes in a hierarchical relationship like this, you can take advantage of polymorphic queries. You can now retrieve all of a user’s posts 1 or their starred posts 2 by taking advantage of the polymorphic query mechanisms that GORM dynamic finders give you for free.

We said that GORM uses one table per hierarchy by default. One of the disadvantages of this approach is that you can’t have a nullable constraint in any of your sub-classes, because base classes do inserts in the same table as your subclasses. But this is a small limitation compared to the power of polymorphic queries.

What if you want to use one table per subclass? You can, but it requires a little GORM DSL magic. Add the following mapping to your root class (Post in your case):

```groovy
static mapping = {
    tablePerHierarchy false
}
```

If you do use one table per subclass, GORM uses outer joins to load instances, so performance can suffer if the class hierarchy is large. But you can use nullable constraints on your domain class fields, so there’s a trade off.
19.1.2 Embedding domain classes

Sometimes you want the convenience of a domain object, but you don’t want the overhead of a separate table. This is particularly likely in legacy scenarios when you have a 1:1 mapping logically, but all the object properties are implemented as separate columns in one big table. For these situations, GORM lets you use embedding. Embedding lets you model one table as two or more related objects.

Let’s take the example of your Profile object. Each user has one Profile object containing their email address and other personal data. But suppose you decide you don’t want the overhead of managing a separate Profile table—you want to embed the profile information directly in the User table. The following code demonstrates how you can rework those classes to make that happen.

```groovy
class User {
    String userId
    String password
    Date signupDate = new Date()
    Profile profile
    static embedded = ['profile']
    // ...
}
class Profile {
    byte[] photo
    String fullName
    // ...
}
```

As you can see, embedding is done with the `embedded` construct. For this to work, you have to define the Profile class in the same Groovy file as the User class (User.groovy), otherwise you end up with an empty Profile table in your database.

Embedding gives you a chance to keep your OO semantics without taking up unnecessary table space in your database.

19.1.3 Using maps for quick and dirty (or cheap and cheerful) tables

Although having a custom domain class manage your data is a logical approach, in certain situations you don’t want the overhead from writing a custom class.

Let’s imagine that Hubbub supports custom plugins. Users can add these custom widgets to their Hubbub home page and configure them with various properties. You could create a domain class called PluginProperty with strings for the property name and value, but it’s a little contrived. All you want is a map to store your key/value pairs.

For these cases, GORM lets you declare properties as maps. The following code shows an example of storing map-style properties.

```groovy
class User {
    String userId
    String password
    Map pluginProperties
    // ... other stuff here
}
```
Behind the scenes, Grails stores pluginProperties as a Map of key/value pairs of type varchar(255), so you're constrained to only storing strings.

Here's how you can test it:

```groovy
user = new User(userId: 'glen', password: 'notlong',
               pluginProperties: ['colour':'green']).save()
user.pluginProperties = [ one: '1', two: '2' ]
```

Storing free-form string data in maps can quickly become a maintenance nightmare, so you should use this capability with caution. Still, it's convenient for quick-and-dirty tables such as pluginProperties.

### 19.1.4 Exploring domain model events

GORM handles much of the drudgery of getting data into and out of the database. But sometimes you need to integrate with GORM's lifecycle to add your own features. In chapter 15 you took advantage of the new Platform Core eventing facility to hook into GORM Events, but that's all new in Grails 2.x. The standard way to accomplish these hooks before Platform Core was to define event methods directly on domain class objects.

Imagine reimplementing your classic audit capability that logs any changes to the domain model each time an object is modified. Standard GORM events provide a mechanism to do that, giving you a way to hook into GORM before and after an object is saved or loaded.

GORM exposes eight main events:

- `beforeInsert`
- `beforeUpdate`
- `beforeDelete`
- `beforeValidate`
- `afterInsert`
- `afterUpdate`
- `afterDelete`
- `onLoad`

If you want to catch any of these, define a closure with the corresponding name such as this:

```groovy
def beforeDelete() {
  log.warn "${id} is toast by ${springSecurityService.currentUser}"
}
```

None of the methods take any arguments or return any values, so if you need a state, it needs to be on the domain class itself or retrievable via the current thread. (In the preceding example, you retrieve the current SpringSecurity user from the current thread.)

GORM makes one more allowance around events to handle the common case of time stamping. One of the most common uses of events is to timestamp the creation or modification of a domain class. For example, you may catch `beforeUpdate` and set
the modification time (or catch beforeInsert and set the creation time). For these situations, GORM offers a convenience convention: if you name two Date fields on your object dateCreated and lastUpdated, GORM automatically sets these fields to the current time:

```java
class User {
    Date dateCreated
    Date lastUpdated
    String userId
    // ...  }
```

This is a special case, but it’s one of the most common uses of events (along with audit logging), so it’s handy having it as a built-in feature. You took advantage of this in your Post object when you created it in chapter 3, and it’s been a real timesaver.

Now that we’ve taken you through the corner cases of domain modeling, it’s time to explore a few tricks and tips to improve your query performance.

19.2 Caching kung fu: moving from 2 users to 2^{10}

In previous chapters, we issued complex queries without much concern for how hard the underlying database may be working to catch up. That’s fine if your site gets only a few hundred hits a day; but when you aim your sights higher, you need to explore caching and performance tuning.

In this section, we look at how GORM handles caching and see how you can determine which parts of your application would benefit from caching and when to tune the knobs. It all starts with understanding GORM’s use of Hibernate second-level caching.

19.2.1 Hibernate settings: should you use the second-level cache?

Whenever you interact with the object model in Hibernate, an internal first-level cache is always in play—the session. For the cases where you change multiple properties on a given object during a request (such as changing the user’s last login time and IP address), Hibernate can batch updates and execute a single SQL `UPDATE`.

Sometimes, the first-level cache isn’t enough. Take the scenario of the 1:m relationship (each user having many posts). Every request in which you do a `user.posts.each { }` results in a requery of the database. If your posts change infrequently (or not at all), that’s wasted querying.

**NOTE** Calling `user.posts.each` twice in the same request won’t requery the database because the collection is cached in the session.

For these scenarios, you can enable the second-level cache. When the second-level cache is enabled for a domain class, Hibernate first searches in this cache before looking in the database. Hibernate also handles evicting objects from the cache when posts are added, edited, or deleted.

The only time you need to be careful about having a second-level cache in play is when you work with clustered application servers. In that scenario, you need to ensure
that none of the servers is working with stale data by configuring your caching provider to be cluster-aware.

As a rule of thumb, you definitely want a second-level cache in play to improve performance. But how do you configure and tune it? Before you start tuning, let's configure the basic cache settings.

### 19.2.2 Configuring the cache

Now that you know what a second-level cache does, it's time to learn how to configure it for your Grails application. The first thing you need to get familiar with is DataSource.groovy. You saw this file when you configured your database connection parameters, but it also includes a hibernate section that you haven’t explored yet:

```groovy
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.region.factory_class='net.sf.ehcache.hibernate.EhCacheRegionFactory'
}
```

Your Hibernate second-level cache is enabled, as is a kind of query cache (which we get to later). A cache region factory class is also specified. This tells Hibernate which underlying caching library out of several to use. Grails uses Ehcache by default, which is a popular and stable option that we’ll stick with.

Each caching provider has its own configuration mechanism. Ehcache uses a single XML file (ehcache.xml) that needs to be in the root of the class path; place it in either the `/src/java` or the `/grails-app/conf` directory. The following listing shows a sample ehcache.xml file to get you started.

```xml
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ehcache.xsd">
    <defaultCache
        maxElementsInMemory="1000"
        eternal="false"
        timeToIdleSeconds="3600"
        timeToLiveSeconds="3600"
        overflowToDisk="false"
        diskPersistent="false"
        diskExpireThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU"
    />
</ehcache>
```

A full discussion of Ehcache is beyond the scope of this book, but these default settings will cache objects for 1 hour (3,600 seconds), after which they're timed out and refreshed from the database when next queried.

Ehcache supports many configuration options, including a distributed cache, the ability to persist the cache to disk to survive restarts, and various eviction policies for
quiet or stale elements. The online documentation (http://ehcache.org/) is excellent, so check it out to learn more.

### 19.2.3 Caching individual domain classes

To enable caching for individual domain classes, a little more work is required. You need to add a `mapping` block to each domain class that you want to cache, such as the following code:

```groovy
static mapping = {
    cache true
}
```

Or, if you need more control over your caching options, you can pass in a map:

```groovy
static mapping = {
    cache usage: "read-write"
}
```

The `cache` setting refers to how Hibernate handles concurrent access to the underlying cache (see table 19.1). Remember that Hibernate writes to that cache, too, every time you update, create, or delete a domain class instance. If you query reference data, and you know there won’t be any updates to the data (if it contains a list of time zones, for example), you can safely set the `cache` setting to `read-only` for better performance.

### Table 19.1 All the standard Hibernate caching strategies are supported by Grails; these are the common ones you’ll use.

<table>
<thead>
<tr>
<th>Cache option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>The same as read-write caching</td>
</tr>
<tr>
<td>read-only</td>
<td>Only useful for static reference data; otherwise concurrency problems are inevitable</td>
</tr>
<tr>
<td>read-write</td>
<td>The default setting; implements concurrency on the cache</td>
</tr>
</tbody>
</table>

For Hubbub, your main caching requirements relate to a user’s posts. In those cases, you want to employ caching strategies when you walk the object graph from `User` to `Post`. For this scenario, you need to modify your `User` class:

```groovy
static mapping = {
    cache: true
    posts cache: true
}
```

**TIP** Remember that you have to enable caching in two places. First, you need to turn caching on globally in `DataSource.groovy` (this is `true` by default). Second, you have to enable caching for your individual domain classes.

But that’s only half of the story. Now that your `Post` object is cached, you may want to tune the timing of that particular cache. Revisit your `ehcache.xml` file and add a new cache element with a `name` that matches the name of the domain class:
<cache
   name="com.grailsinaction.Post"
   maxElementsInMemory="10000"
   eternal="false"
   timeToIdleSeconds="300"
   timeToLiveSeconds="600"
   overflowToDisk="false"
/>
<cache
   name="com.grailsinaction.User"
   maxElementsInMemory="10000"
   eternal="false"
   timeToIdleSeconds="300"
   timeToLiveSeconds="600"
   overflowToDisk="false"
/>

**NOTE** If you put your domain classes in packages, include your package name in the name field, too, as you did in the preceding example (com.grailsinaction.Post).

In the preceding settings, you set the idle time to 300 seconds (so that if no one accesses an instance in the cache in 300 seconds, it will be evicted) and a maximum TTL of 600 seconds (whether or not the object is active, it’s always discarded after 600 seconds). Those are fairly aggressive settings, but it’s a good starting point for your profiling exercise.

**TIP** When it comes to caching, there’s no one-size-fits-all strategy. The strategy you select depends on how objects are used in your system. If you’re in a low-write, high-read environment (like we are for a user’s posts in Hubbub), it makes sense to embrace caching at a certain level. When we cover performance profiling in the next section, you’ll learn about tools you can use to evaluate your caching needs.

Now that you’ve tuned all the settings on your cache, let’s look at profiling to see if you can find any bottlenecks in your application performance.

### 19.2.4 Enough talk, let’s profile: P6Spy

If you’re into a first-principles approach, your profiling journey may start with inspecting the SQL queries that your application generates. To print GORM queries to the console, modify your DataSource.groovy file:

```groovy
development {
   dataSource {
      logSql = "true"
   }
}
```

That gives you insight into query generation, but it won’t tell you how long your database took to satisfy the query. For that, you need to perform query profiling, and the easiest way to do that in Grails is via P6Spy.
P6Spy is a small Java Database Connectivity (JDBC) driver that wraps your real driver to provide a stopwatch. It then logs how many milliseconds the database took to satisfy your query. There’s even a Grails plugin (or two) for P6Spy (you’ll use the more current one: P6Spyui). To install the plugin, edit /grails-app/conf/BuildConfig.groovy:

```groovy
plugins {
    ...  
    compile(':p6spy-ui:0.1') {  
        excludes 'jquery'
    }
}
```

The current version of p6spy-ui (0.1) links to an older version of jQuery, so skip that dependency here. After the plugin is installed, modify your DataSource.groovy file to point to the P6Spy driver instead of your regular database. Update your development dataSource class in /grails-app/conf/DataSource.groovy to use the new P6Spy version:

```groovy
development {  
    dataSource {  
        dbCreate = "update"  
        driverClassName =  
            "com.p6spy.engine.spy.P6SpyDriver"  
        url = "jdbc:h2:mem:devDB"  
    }
}
```

Notice that you don’t change the URL—P6Spy proxies using the username, password, and URL of your database. But you have to tell it the real driver to use, and for that you need to modify your /grails-app/conf/Config.groovy. While the plugin defines most of the defaults, you need to at least tell it the name of your real driver setting, which tells P6Spy which database to proxy:

```groovy
grails.plugin.p6spy.realdriver="org.h2.Driver"
```

You have now set it to your sample H2 database, so you’re in business. P6Spy has a range of tunable options that you can add to your Config.groovy. See the Grails p6spy-ui plugin page for the details.

With your database profiling in place, let’s run Hubbub and see what profiling data you can collect. You now have a P6Spy controller installed at http://localhost:8080/hubbub/p6spy/. Inside, you’ll find the output and timing of every query that went to your database, along with handy charts to monitor performance. Figure 19.1 shows the stream of SQL queries being processed by the profiler. Click the Pause icon at the top of the window to see the results of the analysis.

P6Spy can give you basic database profiling information, and it’s great for working out which queries in your application could benefit most from caching or tuning. But after you have insight into where the bottlenecks are, you’ll want to add indexes. Let’s explore how that’s done in Grails.
Improving performance with indexed fields

Once your profiling efforts give you insight into which retrievals are taking the most time, you should add indexes to your database to make lookups more efficient. Grails handles custom indexing via the versatile mapping closure (you saw this closure when configuring caching).

Suppose your reporting process does a lookup on the User table and creates reports around dateCreated—the date the user signed up for your service. Given the popularity of your website, it makes sense to index on the dateCreated field. Creating the index involves giving the index a name (in your case, date_created_idx to keep the DBAs happy) and then attaching it to the domain class field you want to index on:

```groovy
static mapping = {
    dateCreated index:'date_created_idx'
}
```

Using indexing and profiling with P6Spy can be a great way to identify which queries in your application may benefit from caching.
19.2.6 What about query caching?

We talked about second-level caching and how that can improve performance when navigating the object graph. But what about when you want to cache queries? Options are available for this, but they’re not likely to be helpful. GORM only caches query results if none of the tables included in the query changed since your last query. If GORM changed any of the items in any of the tables, the cached results are discarded, and you hit the database directly. In fact, you also incur the overhead of the cache check.

But if you still want to use query caching, you have a few options. The first is to use a cache argument to dynamic finders, such as

```groovy
def entries = Post.findAllByUser(user, [cache: true])
```

This works great if the query that you need to execute can be expressed in a dynamic finder. Unfortunately, that’s often not the case, so you’re left with two options: criteria queries and Hibernate Query Language (HQL).

Criteria queries are the neatest solution and the one we recommend. Imagine you want to cache a user’s recent posts; specify the `cacheable(true)` setting, and you’re ready to go:

```groovy
def entries = Post.createCriteria().list {
    eq('user', user)
    cacheable(true)
}
```

But remember, if either the `Posts` or `Users` table changes (not only for this user, but for anyone), this query is evicted from the cache.

Criteria queries make this straightforward. But if you already use HQL, you can use Hibernate directly:

```groovy
def recentPosts = {
    def posts = Post.withSession { session ->
        session.createQuery("select p from $Post.name p where p.user.id=:userid")
            .setCacheable(true)
    }
}
```

What if I need distributed caching?
Second-level caches are great, but traditionally they haven’t had the best support for clustering, making them significantly less useful in high-availability settings. Replication technologies were typically chatty and didn’t perform well, but great progress has been made in this area. Commercial-strength distributed caches, such as the open source Terracotta platform, now have excellent Grails support through plugins. Ehcache now has distributed cache options, too. If you run in a cluster, be sure to check them out.

It’s time to explore what facilities Grails offers to accelerate query performance through caching.
As you can see, this technique is more powerful but less elegant than dynamic finders or criteria queries, which we prefer for all caching queries (where practical).

By default, all queries are cached in a single cache called `org.hibernate.cache.StandardQueryCache`. You can define a cache with this name in your ehcache.xml file if you want to control how long the cache should live. If you don’t, it uses your `defaultCache` settings. Alternatively, you can use different `cache regions` for each of your cached queries.

To use cache regions, define a cache element in ehcache.xml with the name of the cache (such as `hourlyCache`). With this cache region defined, pass its name into your criteria:

```groovy
def entries = Post.createCriteria().list {
    eq('user', user)
    cacheable(true)
    cacheRegion('hourlyCache')
}
```

Alternatively, if you use HQL to perform your query, chain a call to `setCache-Region('hourlyCache')` to get the same effect.

In addition to these query performance-enhancement options, there’s a server-level query performance-enhancement option you haven’t explored yet. JNDI data sources, provided by most servers, give you an efficient way to manage database connections. It’s time to explore this and other benefits they offer.

### 19.2.7 JNDI? That’s so old school…

If you always lumped JNDI into the category of “old-school J2EE stuff that’s long past,” it’s time to have a second look. Compelling reasons exist for using JNDI data sources in your next Grails project—particularly for the production data source:

- You don’t need to hardcode usernames, passwords, database names, or even types, making for a source-control-friendly check-in for your project even on public sites such as GitHub, BitBucket, and Google Code.
- It offers efficient pooling of resources by your server, including adding connections to the pool when needed and reconnecting dead instances.
- Developers can easily use their preferred DBMS without custom configuration and library files.

But JNDI has drawbacks, too:

- The JNDI naming format can differ from container to container.
- Configuration for Jetty is problematic and requires a custom jetty-env.xml file in your `/WEB-INF/` directory.
There’s rarely a good reason not to use a JNDI data source for your production data source, and we encourage you to make this your first change after doing grails create-app. We’ve deployed small apps to production accidentally using an embedded HyperSQL Database (HSQLDB), and we don’t want you to suffer the same pain.

The following code example shows how JNDI data sources are configured for the GlassFish application server in DataSource.groovy. This configuration varies from server to server, so check the documentation.

```groovy
// environment specific settings
environments {
    // ... 
    production {
        dataSource {
            dbCreate = "update"
            jndiName = "jdbc/hubbub"
        }
    }
}
```

After you update your DataSource.groovy file, you need to define the JNDI data source itself in your application server. Usually, your server admin console lets you create JNDI data sources and configure the database username, password, connection pool size, and so on.

That completes your survey of query and data source tricks and tips. It’s time now to look at legacy concerns. Integrating with legacy databases is an important part of enterprise development, so we’ll spend the second half of this chapter exploring the common issues you’re likely to encounter when integrating your new Grails applications with historical data.

### 19.3 Legacy integration kung fu

It would be nice if you could start with a clean data model each time, but life is rarely that simple. You often have an existing database that your DBA won’t modify. Like all decent frameworks, Grails is comfortable playing nicely with your existing environment.

In this section, you examine three techniques for migrating an existing database application to Grails:

1. **Recycling mappings**—You’ll look at how to reuse your existing Hibernate mappings and Java domain objects, and build a Grails UI in front.
2. **Using GORM DSL**—You’ll look at the situation where you have an existing legacy database but no existing Hibernate mappings. In that case, you can skip the Hibernate step entirely and use the GORM legacy mapping DSL to map your Grails domain objects directly to your existing database schema.
3. **Dealing with multiple data sources**—You’ll look at how to integrate multiple data sources, something that may come up when working with legacy code.

But first, we introduce you to a poorly constructed legacy database. This is a real data model found in a real enterprise (with the names changed to protect the innocent).
This model has all the nasty surprises you’d expect: string-based natural keys, auto-incrementing keys not based on sequences, many-to-many tables with link attributes, several data types used for table keys (including `char`), and more. The basic layout of the data model is shown in figure 19.2.

Let’s step through the process of writing a simple single-page UI that lets you browse the data model. If you have Hibernate mappings for the database, it would be nice to reuse them. Let’s explore that option first.

### 19.3.1 Recycling Hibernate mappings

If you’re coming to GORM after using Hibernate, you’re familiar with Hibernate annotations and mapping files. You probably have `.hbm.xml` files and matching domain classes. There’s no need to throw them away.

Lurking quietly in the `grails-app/conf/hibernate` directory is a space for you to place your existing mapping files. Drop your domain classes in `/src/java` (or possibly as a JAR file in your `/lib` directory), and you’re in business. If you name your Hibernate configuration file `/grails-app/conf/hibernate/hibernate.cfg.xml`, there’s nothing more to do.

The following code shows a sample `hibernate.cfg.xml` file for the legacy database in figure 19.2.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <mapping resource="org.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Notice the lack of data-source configuration, which you typically see in `hibernate.cfg.xml`—this is already configured in the data-source definitions in `DataSource.groovy`. But you do need to reference your existing `org.hbm.xml` file, which contains your Hibernate domain-class mapping definitions for the sample database. You can use
annotations to specify these mappings, but because this is a legacy setup, let’s assume you’re lost in a world of pointy XML.

Your org.hbm.xml file is a standard Hibernate mapping file. It specifies the mapping from a series of bizarrely named underlying tables to a set of standard Java classes. The following listing shows an extract of your mapping file to give you a feel for it.

Listing 19.3 An extract from a legacy Hibernate mapping file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
 http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping auto-import="true" default-lazy="false"
 package="com.grailsinaction.legacy.db">
 <class name="Branch"
     table="BK_BRANCH" mutable="false">
     <cache usage="read-write"/>
     <id name="name" column="BRANCH_NM">
         <generator class="assigned"/>
     </id>
     </class>
</hibernate-mapping>
```

The syntax is ugly, but it does the heavy lifting. The full set of mapping files for this domain is included in the book’s source code.

**PIMP my DATA MODEL: ADDING AN AJAX INTERFACE**

Suppose you got your legacy model running by importing your existing Hibernate mappings, and you have access to your Java DTOs, too. Let’s write controller logic so you have a basic UI for browsing your domain model.

You don’t have scaffolding options because the standard scaffolds make assumptions about the existence of ID fields, which you don’t have. You could customize the scaffolding templates, but that’s too much work for what you want. Let’s create a quick Ajax-based controller to browse the database.

Here’s your browser controller:

```java
import com.grailsinaction.legacy.db.*
class BrowserController {
    def index = {  redirect(action: 'list') }
}
Legacy integration kung fu

def list = {
    // Pass through to Ajax browser form
}

There’s no rocket science here—you expose a list endpoint to pass through to your Ajax form. In the following listing, you create your list.gsp with your Ajax-populated drop-down list.

Listing 19.4 Implementing a list.gsp with Ajax

```groovy
html
<head>
    <title>Legacy DB Browser</title>
    <g:javascript library="prototype"/>
</head>
<body>
    <fieldset>
        <legend>Browse Branches</legend>
        <g:form action="showDetails">
            <label for="id">Branch:</label>
            <g:select name="id"
                from="${com.grailsinaction.legacy.db.Branch.list()}
                optionKey="name" optionValue="name"/>
            <g:submitToRemote value="Show Details"
                update="details"
                url="[action: 'showDetails']"/>
        </g:form>
    </fieldset>
    <div id="details"></div>
</body>
</html>
```

Not much exciting stuff is happening there. You have a combo box full of available branch names and a Show Details button that submits the request to the controller and replaces a div with the returned value. Figure 19.3 shows the UI so far.

![Figure 19.3 The Branch browser combo box launches an Ajax submit.](image)

Let’s implement your controller logic to handle the submit and return well-formed HTML. Using Groovy’s MarkupBuilder is the simplest way to generate the HTML, so let’s implement code to return the manager for the selected branch. The following listing shows the initial back-end code.

Listing 19.5 Implementing the Ajax back end to show branch details

```groovy
def showDetails() {
    def branch = Branch.findByName(params.id)

    if (branch) {
        return branch
    } else {
        return null
    }
}
```

Queries legacy database to find branch

Populates drop-down box with list of branch objects

Performs Ajax submit of selected branch

Holds DIV for content returned via Ajax
def writer = new StringWriter()
def html = new groovy.xml.MarkupBuilder(writer)
// Could do all this directly in a render() call
// but it's harder to debug
html.div {
    div(id: 'manager') {
        fieldset {
            legend('Manager Details')
            dl {
                dt('Name: ')
                dd(branch.manager.name)
                dt('Rating: ')
                dd(branch.manager.managementRating)
            }
        }
    }
}
render(writer.toString())

One thing to notice in listing 19.5 is that you use a dynamic finder, Branch.findByName(), on your legacy DTO class. Grails decorates your DTO with a proxy that implements the persistence and querying logic that Grails needs, so you get all this for free on your legacy classes.

With your markup builder in place to navigate your data model, let’s take the UI for a spin as shown in figure 19.4.

Figure 19.4  The basic manager details are in place.

Fantastic! You have your Ajax interface in action, but you haven’t exercised all the potential relationships that could be navigated via the model. Let’s fix that right now with a much more complete MarkupBuilder example. The following listing shows code that navigates the entire legacy model.

Listing 19.6  A more complete showDetails that navigates all relationships

def showDetails = {
    def branch = Branch.findByName(params.id)
    def writer = new StringWriter()
    def html = new groovy.xml.MarkupBuilder(writer)
    // Could do all this directly in a render() call
// but it's harder to debug
html.div {
    div(id: 'manager') {
        fieldset {
            legend('Manager Details')
            dl {
                dt('Name: ')
                dd(branch.manager.name)
                dt('Rating: ')
                dd(branch.manager.managementRating)
            }
        }
    }
}
div(id: 'sections') {
    branch.sections.each { section ->
        fieldset {
            legend('Section: ' + section.name)
            dl {
                dt('Start Date: ')
                dd(section.start)
                dt('Files: ')
                dd(section.files.size())
                section.files.each { sectToFile ->
                    dl(style: 'padding: 1em; border: 1px dotted black') {
                        dt('File Name: ')
                        dd(sectToFile.file.name)
                        dt('Type: ')
                        dd(sectToFile.file.resourceType.name)
                        dt('Created: ')
                        dd(sectToFile.start)
                        dt('Owner: ')
                        dd(sectToFile.file.owner.name)
                    }
                }
                dt('Locations: ')
                dd(section.locations.size())
                ul {
                    section.locations.each { sectToLoc ->
                        li(sectToLoc.location.name)
                    }
                }
            }
        }
    }
}
render(writer.toString())

That’s a much bigger builder, but this time you got a full navigation graph to make sure everything works correctly. Check out the updated UI in figure 19.5.

You know that your legacy relationships are all in order, but you don’t want to settle for read-only access to your data. It’s time to explore your options for saving new
Now that your legacy classes are mapped, it’s time to look at options for adding constraints.

**Adding constraints to legacy POJO classes**

Even though your legacy classes are POJOs, Grails still gives you a convention-based facility for specifying legacy mappings. You can’t add `constraints` closures to Java
classes (at least not in JDK7), but Grails lets you place Groovy constraint scripts alongside your Java classes following the convention of \textit{DomainClassName}Constraints.groovy. Here’s a sample BranchConstraints.groovy file:

```groovy
def constraints = {
    name(size: 4..30)
}
```

Note that these classes are placed alongside your Java POJOs in /src/java (and not in /src/groovy, as you might expect). Also note that the \texttt{constraints} closure isn’t static as in GORM classes.

Let’s write a test case to confirm that your constraint works as anticipated. The following code shows a sample test.

```groovy
import com.grailsinaction.legacy.db.*
import grails.test.*
class BranchTests extends GrailsUnitTestCase {
    void testBranchConstraints() {
        def branch = new Branch(name: 'a')
        assertFalse(branch.validate())
        branch.name = 'aaaa'
        assertTrue(branch.validate())
    }
}
```

If you run \texttt{grails test-app}, you find this test passes, telling you that your constraint works fine.

With your exploration of constraints complete, you’re now on top of all the options for Hibernate-based legacy integration. But what if you don’t have legacy Hibernate mappings? It’s time to look at what the GORM DSL offers for doing legacy mappings without all that XML.

\section*{19.3.2 Using GORM DSL to access existing database table structures}

All this work around integrating legacy Hibernate mappings is fantastic if you have legacy Hibernate mappings to work with. But you might need to develop a snazzy UI for a legacy database that powers an old PHP application. For these situations, GORM offers a DSL for working with legacy databases.

By adding a \texttt{mapping} section to your domain classes (which you saw in section 19.2), you can change the name of the tables or columns, specify join tables, use custom key-generation strategies, and more. Behind the scenes, the GORM DSL configures the Hibernate mappings dynamically, so you can work with your domain classes and not get caught up in XML. In this section, you redevelop your legacy database example from the previous section using the GORM DSL.

First, create a set of domain class objects in /grails-app/domain because you’re dealing with first-class Grails domain classes. Keep the same package names because you can reuse the controller code you already developed to browse the new object model. Figure 19.6 shows the reworked domain classes, this time rewritten in Groovy.
With your domain model in place, it’s time to get acquainted with the mapping options available for hooking up legacy tables.

**THE BASICS: CHANGING TABLE AND FIELD NAMES**

DBAs have a long history of creating bizarre naming standards, so the first thing you need to learn is mapping to custom table and field names. You can map to fields named whatever you like by taking advantage of the mapping block.

Let’s tackle something simple, such as the `FileOwner` object, which doesn’t have any relationships. The following listing shows the reworked domain class.

```java
package com.grailsinaction.legacy.db

class FileOwner {
    int id
    String name
    String description

    static mapping = {
        table 'BK_FILE_OWNER'
        version false
        id column: 'FILE_OWNER_ID', generator: 'increment'
        name column: 'OWNER_NM'
        description column: 'OWNER_DESC'
    }
}
```

Listing 19.7 A domain class with a legacy mapping block

Your mapping closure introduces a few new constructs that you haven’t seen before. You can change the name of the backing table ②, and by specifying property names in the block, you can remap them to their corresponding database columns.

Another thing to note is version false ③. Hibernate uses the version column to track object versions for optimistic locking. This isn’t implemented in your legacy tables, so you turn off the feature.

You also define an int-based id field (the default type is long) to make sure everything matches up to your int-based id field in the database ①. Alternatively, you can specify the type on the mapping itself (id column: 'FILE_OWNER_ID', type: 'integer'), which is convenient for tricky timestamp formats or char fields (such as DB2’s yes_no character field to handle Booleans).

The id field uses one of the built-in Hibernate generators for generating key values. In this case, you use the increment strategy ④, which uses the next incrementing number, but you’re free to use any of the generators (including sequences, hilo, or even assigned natural keys, which we cover shortly).
That gets you through the basics of mapping custom field and table names, but what about the more thorny issue of navigating relationships? It’s time to dig a little deeper.

**Adding One-to-Many Relationships**

One of the most common relationship types is the one-to-many (or many-to-one) relationship. Our File object has several many-to-one relationships mapped. The following listing demonstrates mapping this complex class using mapping DSL entries.

### Listing 19.8 Mapping a complex relationship using the mapping DSL

```groovy
class File {
  int id
  FileType resourceType
  String name
  short securityRating
  FileOwner owner
  Date start = new Date()
  Date end = new Date()
  String description
  static hasMany = [sections: SectionToFile]
  static mapping = {
    table 'BK_FILE'
    version false
    id column: 'FILE_ID', generator: 'increment'
    name column: 'FILE_NM'
    start column: 'START_DT', type: 'date'
    end column: 'END_DT', type: 'date'
    description column: 'FILE_DESC'
    securityRating column: 'SECURITY_RATING_VAL'
    owner column: 'FILE_OWNER_ID'
    resourceType column: 'FILE_TYPE_CD'
    sections column: 'SECTION_ID',
    joinTable: 'BK_FILE_SECTION_MAP'
  }
}
```

FileOwner and FileType are the many-to-one relationships in this class. All that’s required is a mapping from the field name to the database column for the foreign key. After your mapping is in place, you’re free to navigate the object graph like a standard Grails relationship.

**Many-to-Many: Handling Join Tables**

In your sample application, you need to model explicit join objects from SectionToFile. That’s because custom attributes are on the join itself (in your case, start and end dates for the file’s ownership lifetime). Often, no attributes exist on the join, and you have a classic many-to-many with a join table containing only the two IDs for each object in the relationship.

You have pure join tables with no other attributes when linking sections to locations. The join table, BK_LOCATION_SECTION_MAP, has IDs for each side and no other attributes. Mapping this in the GORM DSL requires specifying a `hasMany` block.
and then adding an appropriate `joinTable` mapping. The following listing shows a mapping via a join table.

### Listing 19.9 Mapping relationships that use join tables

```java
package com.grailsinaction.legacy.db

class Section {
    int id
    String name
    Date start = new Date()
    Date end = new Date()
    static hasMany = [files: SectionToFile,
                      locations: Location,
                      branches: BranchToSection ]
    static belongsTo = [ Branch ]
    static mapping = {
        table 'BK_SECTION'
        version false
        id column: 'SECTION_ID', generator: 'increment', type:'integer'
        name column: 'SECTION_NM'
        start column: 'START_DT', type: 'date'
        end column: 'END_DT', type: 'date'
        locations column:'SECTION_ID',
        joinTable:'BK_LOCATION_SECTION_MAP'
    }
}
```

Take special note of your `locations` attribute. It uses the `joinTable` attribute to specify the name of the many-to-many link table, and it also specifies the column (`SECTION_ID`) of this side of the relationship.

**Corner Case: Handling Natural and Composite Keys**

You still have key-related territory to cover. Not all tables have the luxury of a surrogate primary key; many depend on either a natural key or a collection of fields that forms a composite key. GORM DSL has support for both, in varying degrees.

In your example application, you have a `SectionToFile` join object that models attributes of the join between `Section` and `File`. Being a join table, it has neither a natural nor a surrogate key, but you can model the composite of both fields as your key. This is shown in the following listing. Note that Hibernate requires these linking objects to implement `Serializable`.

### Listing 19.10 Handling existing link tables in many-to-many relationships

```java
package com.grailsinaction.legacy.db

class SectionToFile implements Serializable {  
    Section section
    File file
    Date start
    Date end
    static mapping = {
        table 'BK_FILE_SECTION_MAP'
    }
}
```
One thing that you lose with composite IDs is the ability to do \texttt{get()}s based on the composite ID. You need to use query by example to get back to your original object.

The case of natural keys is trickier. In your \texttt{Branch} class, the \texttt{name} property forms the primary key for the table. Convincing GORM that this is a good thing involves a sleight of hand, as shown in the following listing.

\begin{Verbatim}
Listing 19.11 Handling natural keys with GORM workarounds
\end{Verbatim}

GORM depends on having an \texttt{id} field, and it insists that it appear in your mapping. In listing 19.11, you have a \texttt{name} field rather than \texttt{id}, so you rely on getters and setters to map your internal \texttt{id} field to a more public \texttt{name} field. It’s not pretty, but it does the job.

Be careful when saving objects mapped in this way because Grails views a non-null \texttt{id} as indicating a persistent instance. You need to use the \texttt{branch.save(insert: true)} option with this approach.

That rounds out your exploration of legacy data sources. We took you through a worst-case database, so you should feel confident that you can tackle any structure that a DBA can throw at you.
When to use GORM DSL and when to use Hibernate mappings

Now that you’ve had a good look at both GORM DSL and Hibernate mappings for the same nasty database, you may wonder what the best option is. Obviously, if you have existing Hibernate mappings, it makes sense to reuse them. But what if you have a brand-new web application that needs to integrate with a crazy legacy database? As with most things, the answer is “it depends.”

If your database makes extensive use of natural (noninteger) keys or join tables with attributes, using Hibernate mappings is usually cleaner. That said, if your team isn’t invested in Hibernate mappings, a GORM DSL solution almost always exists, but it may require compromises in your object model—you may end up introducing join objects with composite keys to map the relationship.

If your legacy database is fairly modern with surrogate keys and not too much special sauce, GORM DSL is definitely the way to go. You end up with a clean set of object classes, and you can live your life without the tedious pointy XML of Hibernate mappings.

19.3.3 Dealing with multiple data sources

The Grails data source mechanism makes it easy to configure a single data source, but what if your application needs more than one? Suppose you need to access a reference data table of countries and their codes in another team’s database.

The good news is that Grails supports multiple data sources out of the box. To add a second data source, create a new entry in your /grails-app/conf/DataSource.groovy with the name `dataSource_yourotherds`. The standard data source properties that you’re used to still apply.

The following listing shows a sample definition of `DataSource.groovy` that loads two reference table classes from a separate PostgreSQL database. Most of the settings will be familiar, because you saw them specified in the standard `DataSource.groovy` file.

```
Listing 19.12 Configuring multiple data sources in /grails-app/conf/DataSource.groovy

dataSource {
    dbCreate = "create-drop"
    url = "jdbc:h2:mem:devDb"
}
dataSource_countries {
    dialect = org.hibernate.dialect.PostgreSQLDialect
    driverClassName = 'org.postgresql.Driver'
    username = 'glen'
    password = 'password'
    url = 'jdbc:postgresql://localhost/reference_db'
    dbCreate = 'update'
}
```

After you configure your data source, you need to tell your domain class which data source it should use to query and persist its data. This is handled via the ever-versatile GORM mapping DSL:
You can configure as many data sources as you like in this file and make appropriate entries in the matching domain classes for that data source. After your mapping block and data source definition is in place, you can then access your domain objects with the usual dynamic finders, such as `CountryCodes.findByCode('au')`. Save and update operations work as well.

Grails's multiple data source support is extensive. There's even support for mapping a given domain class across multiple data sources (so you can query `CountryCodes` from several data sources!) Check out the Grails documentation for the details.

19.4 Summary and best practices

We introduced advanced GORM functionality in this chapter. The less-common relationship types (including inheritance) can prove to be real timesavers (and code neateners), but you have to remember the pros and cons of each choice.

We also took you through the details of query and second-level caching, and we covered cache tuning in detail.

We looked at performance-measurement options for the data tier and explored a profiling tool to help you work out what to tune. Finally, we undertook a detailed range of options for getting Grails working with legacy databases, including reuse of existing Hibernate mapping files and working with the GORM legacy mapping DSL.

Before we move on to explore more about plugins in chapter 20, let's review a few key best practices from this chapter:

- **Understand inheritance.** When evaluating your inheritance options, understand the potential performance cost of using a one-table-per-class strategy, and weigh that against the relaxed validation constraints required when using the one-table-per-hierarchy approach.

- **Use JNDI data sources.** Use JNDI for production data sources (and change your database setting for production to JNDI straight after running `create app`).

- **Don't guess, profile.** The Grails P6Spy UI plugin gives you good insight into your query timings. Combined with a load tool such as Apache JMeter, you can simulate how your app will perform if you get slashdotted.

- **Don't be overzealous with query caching, but do use second-level caches.** Sometimes you're better off rolling your own caching mechanism for high-churn data (such as for summary tables that update regularly but are backed by an expensive query). Always enable second-level caching (although this has caveats, particularly when running in clusters).
- **Reuse Hibernate mappings.** If you have existing Hibernate mappings for your legacy database, use them. GORM DSL requires compromises that aren’t worth it if you’ve already done the hard work of writing real Hibernate mappings.

- **Favor GORM DSL for well-designed databases and Hibernate mappings for evil ones.** For brand-new development on well-designed legacy databases, favor GORM DSL. There’s less to maintain, and your team doesn’t need to learn Hibernate. When working with legacy databases that have bizarre schemas, favor raw Hibernate mappings (particularly if you have existing Hibernate mapping files). Hibernate mappings have much more power than straight GORM DSL, and you end up with a much cleaner solution.

In the next chapter, we venture into a discussion on how to develop plugins, so you can create new ones that add functionality to your growing application.
It may be time for you to stop reconfiguring, rewriting, and recompiling your Java web apps. Grails, a Groovy-powered web framework, hides all that busy work so you can concentrate on what your applications do, not how they’re built. In addition to its famously intuitive dev environment and seamless integration with Spring and Hibernate, the new Grails 2.3 adds improved REST support, better protection against attacks from the web, and better dependency resolution.

**Grails in Action, Second Edition** is a comprehensive introduction to Grails 2. In this totally revised edition you’ll master Grails as you apply TDD techniques to a full-scale example (a Twitter clone). Along the way you’ll learn “single-page web app” techniques, work with NoSQL back ends, integrate with enterprise messaging, and implement a RESTful API.

**What’s Inside**
- Covers Grails 2.3 from the ground up
- Agile delivery and testing using Spock
- How to use and manage plugins
- Tips and tricks from the trenches

No Java or Groovy knowledge is required. Some web development and OOP experience is helpful.

There’s no substitute for experience: **Glen Smith** and **Peter Ledbrook** have been fixtures in the Grails community, contributing code, blogging, and speaking at conferences worldwide, since Grails 0.2.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/GrailsinActionSecondEdition