Mastering Grails: Many-to-many relationships with a dollop of Ajax

Skill Level: Introductory

Scott Davis (scott@aboutgroovy.com)
Editor in Chief
AboutGroovy.com

15 Apr 2008

Many-to-many (m:m) relationships can be tricky to deal with in a Web application. In this installment of *Mastering Grails*, Scott Davis shows you how to implement m:m relationships in Grails successfully. See how they're handled by the Grails Object Relational Mapping (GORM) API and the back-end database. Also find out how a bit of Ajax (Asynchronous JavaScript + XML) can streamline the user interface.

Software development is about modeling the real world in code. For example, books have authors and publishers. In a Grails application, you'd create a domain class for each. GORM creates a corresponding database table for each class, and the scaffolding gives you a basic Create/Retrieve/Update/Delete (CRUD) Web interface for free.

The next step in the process is to define the relationships among these classes. A typical publisher publishes more than one book, so the relationship between a publisher and its books is a straightforward one-to-many (1:m) relationship: one `Publisher` publishes many `Books`. You create the 1:m relationship by putting `static hasMany = [books:Book]` in the `Publisher` class. Putting `static belongsTo = Publisher` in the `Book` class adds another dimension to the relationship — cascading updates and deletes. If you delete a `Publisher`, all of the corresponding `Books` are deleted too.

A 1:m relationship is easy to model in the underlying database. Each table has an `id` field that serves as the primary key. When GORM adds a `publisher_id` field to the book table, you have a 1:m relationship between the two tables. On the front end, Grails handles 1:m relationships with aplomb as well. When you create a new `Book`, the automatically generated (scaffolded) HTML form offers a drop-down

combo box, limiting your choices to a list of existing `Publishers`. You've seen examples of 1:m relationships since the first article in this series.

Now it's time to focus on a slightly more sophisticated relationship — the many-to-many (m:m) relationship. The relationship between `Book` and `Author` isn't as easy to model as the one between `Book` and `Publisher`. One book can have many authors, and one author can have many books. This is a classic m:m relationship. In terms of modeling the real world, m:m relationships are quite common. One person can have many checking accounts, and one checking account can be managed by many people. One consultant can work on many projects, and one project can have many consultants. This article shows you how to implement a m:m relationship in Grails, building on the trip-planner application I've been developing throughout the series. Before I turn to the trip planner, though, I'll stick with the book example just a little longer to help you understand an important point.

The third class

In a database, three tables represent m:m relationships: the two tables you'd expect (`Book` and `Author`), and a third join table (`BookAuthor`). Rather than adding a foreign key to either the `Book` or `Author` table, GORM adds `book_id` and `author_id` to the `BookAuthor` join table. The join table allows you to persist books with a single author as well as books with multiple authors. It also lets you represent authors who have written many books. Each unique combination of `Author` and `Book` foreign keys gets its own record in the join table. You gain truly infinite flexibility: one book can have an unlimited number of authors, and one author can have an unlimited number of books.

Dierk Koenig once told me, "If you think that two objects share a simple many-to-many relationship, you haven't looked closely enough at the domain. There is a third object waiting to be discovered with attributes and a life cycle all its own." Indeed, the relationship between `Book` and `Author` goes beyond the simple join table. For example, Dierk is the primary author of *Groovy in Action* (Manning Publications, January 2007). Primary authorship should be represented as a field in the relationship between `Author` and `Book`. So should various other facts: the authors are listed on the cover in a particular order; each author contributed specific chapters to the book; and it's likely that each author was compensated differently based on his contribution. As you can see, the relationship between `Author` and `Book` is a bit more nuanced than originally planned. In the real world, each author signed a contract detailing his relationship with the book in explicit terms. Perhaps a first-class `Contract` class should be created to represent the relationship between `Book` and `Author` better.

In simple terms, this means that what looks like a m:m relationship is in reality two 1:m relationships. If two classes look like they share a m:m relationship, you should dig deeper to identify the third class that holds the two 1:m relationships and

explicitly define it.

Modeling airlines and airports

Going back to the trip planner now, let's revisit the domain model and see if any m:m relationships are lurking around. In the first article, I created a `Trip` class, shown in Listing 1:

Listing 1. The Trip class

```
class Trip {
  String name
  String city
  Date startDate
  Date endDate
}
```

In the second article, I added an `Airline` class, shown in Listing 2, to the mix to demonstrate a simple 1:m relationship:

Listing 2. The Airline class

```
class Airline {
  static hasMany = [trip:Trip]
  String name
  String frequentFlier
}
```

These toy classes served their purpose at the time — simple placeholders to illustrate a point — but they don't hold up as a rigorous domain model. It's time to fix up the original classes and build out something that is a bit more robust.

I created the `Trip` class the way I did because it sounded right at the time. I said things like, "I'm planning a trip to Chicago," or "I'm going to be in New York City from the 15th to the 20th of next month." Fields like `city`, `startDate`, and `endDate` seemed to be natural attributes of my `Trip`. On second look, however, a `Trip` is probably a bit more involved.

I live in Denver, CO — a hub city for United Airlines. That means that I can usually fly directly to my final destination, but sometimes it takes two or more hops to get there. Other times, a trip might involve more than one city by design: "I'm flying out to Boston to teach a class Monday through Friday. While I'm out on the East Coast, I need to swing by Washington, D.C. to speak at a conference on Saturday. I'll fly back home on Sunday afternoon." Even if I'm lucky enough to find a direct flight to a specific city, and I'm not flying to any other cities, my trip will still involve more than one flight — the flight out and the return flight. One `Trip` can involve many `Flights`. Listing 3 defines the relationship between `Trip` and `Flight`:

Listing 3. The 1:m relationship between Trip and Flight

```
class Trip{
  static hasMany = [flights:Flight]
  String name
}

class Flight{
  static belongsTo = Trip
  String flightNumber
  Date departureDate
  Date arrivalDate
}
```

Remember that setting up the relationship with a `belongsTo` field means that deleting a `Trip` also deletes all related `Flights`. If I were building a system for air traffic controllers, I'd probably want to make a different architectural decision. Or if I were trying to build a system for multiple passengers to share a common flight (one `Flight` can have many `Passengers`, one `Passenger` can have many `Flights`), tying a flight to a specific passenger's trip might be a problem. But I'm not trying to model the thousands of flights that occur every day across the world for millions of passengers. In my simple case, all a `Flight` does is further describe a `Trip`. If a `Trip` ceases to be important to me, so does each accompanying `Flight`.

What should I do with the `Airline` class now? One `Trip` can involve many different `Airlines`, and one `Airline` can be used on many different `Trips`. There is definitely a m:m relationship between these two classes, but `Flight` seems to be the appropriate place to add the `Airline`, as shown in Listing 4. One `Airline` can have many `Flights`, while a single `Flight` never has more than one `Airline`.

Listing 4. Relating Airline to Flight

```
class Airline{
  static hasMany = [flights:Flight]
  String name
  String iata
  String frequentFlier
}

class Flight{
  static belongsTo = [trip:Trip, airline:Airline]
  String flightNumber
  Date departureDate
  Date arrivalDate
}
```

You should notice a couple of things. First of all, the `belongsTo` field in `Flight` switched from a single value to a hashmap of values. One `Trip` can have many `Flights`, and one `Airline` can have many `Flights` as well.

Next, I added a new `iata` field to `Airline`. This is for the International Air Transport Association (IATA) code. The IATA gives each airline a unique code —

UAL for United Airlines, COA for Continental, DAL for Delta, and so on. (See Resources for a full list of IATA codes.)

Finally, you should notice another architectural decision I made, this time involving the relationship between `Airlines` and frequent-flier numbers. Because I'm assuming a single user for this system, it's perfectly valid for `FrequentFlier` to be an attribute of the `Airline` class. I can't have more than one frequent-flier number per airline, so this is the simplest possible solution. If the requirements for this trip planner change and I need to support multiple users, I see another m:m relationship emerging. One passenger can have many frequent-flier numbers, and one airline can have many frequent-flier numbers. Creating a join table to manage this relationship would be the right thing to do. I'm going to stick with the simple solution for now, but I'll mentally flag the `FrequentFlier` field as a future refactoring point if requirements change.

City or airport?

Now it's time to add the `City` back into the mix — or maybe not. Although you might say, "I'm flying into Chicago," technically it's an airport you fly into. Am I flying into the Chicago O'Hare or Midway airport? When I fly into New York, is it to LaGuardia or JFK? Clearly I need an `Airport` class instead of a simple `City` field. Listing 5 shows the `Airport` class:

Listing 5. The Airport class

```
class Airport{
  static hasMany = [flights:Flight]
  String name
  String iata
  String city
  String state
  String country
}
```

You can see in Listing 5 that the `iata` field is back. This time DEN is Denver International Airport, ORD is Chicago O'Hare, MDW is Chicago Midway, and so on. You might want to create a `State` class and set up a simple 1:m relationship, or even go so far as to create a `Location` class that encapsulates city, state, and country. I'll leave that for you to finish as a rainy-day project.

Now I'll add `Airports` to the `Flight` class, as shown in Listing 6:

Listing 6. Relating Airports to Flight

```
class Flight{
  static belongsTo = [trip:Trip, airline:Airline]
  String flightNumber
```

```
  Date departureDate
  Airport departureAirport
  Date arrivalDate
  Airport arrivalAirport
}
```

This time, however, I create `departureAirport` and `arrivalAirport` fields explicitly rather than implicitly using the `belongsTo` field. The user interface won't look any different — the fields will all be displayed using combo boxes — but the relationship between the classes is subtly but importantly different. Deleting an `Airport` won't cascade down to the associated `Flight`, whereas deleting a `Trip` or an `Airline` will. I present both methods to you here to illustrate the various ways to relate the various classes together. In reality, it's up to you to decide whether you want your classes to maintain strict referential integrity (in other words, have all deletes cascade) or to allow a looser relationship.

Seeing many-to-many relationships in action

The object model in place now does a reasonably good job of modeling the real world. I take many trips a year, using many different airlines, flying into many different airports. What ties all of these many relationships together is a `Flight`.

Looking at the underlying database, I see nothing but the tables I'd expect to see, as shown by the output of the MySQL `show tables` command in Listing 7:

Listing 7. Tables behind the scenes

```
mysql> show tables;
+----------------+
| Tables_in_trip |
+----------------+
| airline        |
| airport        |
| flight         |
| trip           |
+----------------+
```

The columns in the airline, airport, and trip tables all match the fields in the corresponding domain classes. The `flight` table is the join table, representing the complex relationship among the other tables. Listing 8 shows the fields in the `Flight` table:

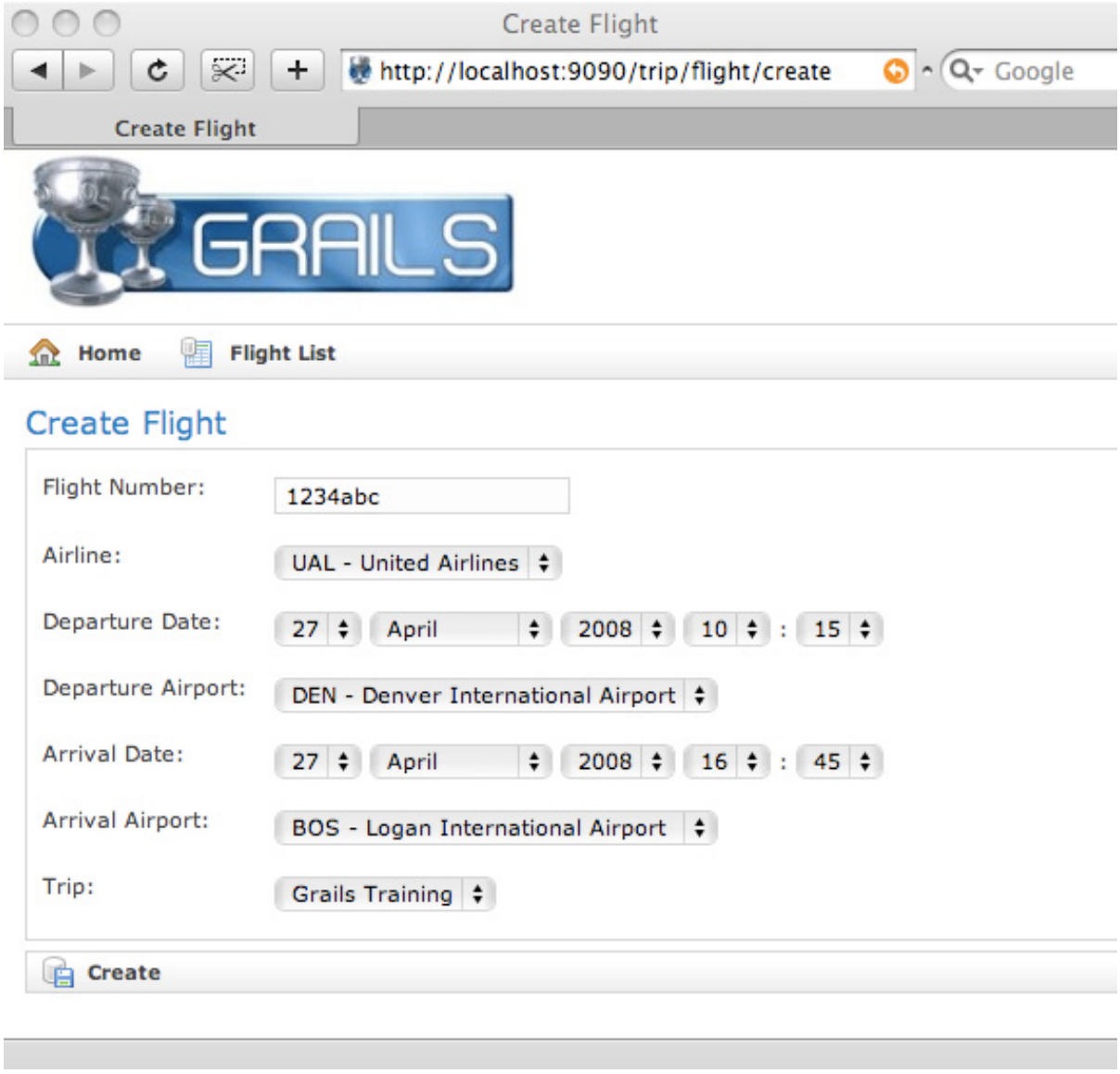
Listing 8. The fields in the Flight table

```
mysql> desc flight;
+----------------------+--------------+------+-----+
| Field                | Type         | Null | Key |
+----------------------+--------------+------+-----+
| id                   | bigint(20)   | NO   | PRI |
```

```
| version               | bigint(20)   | NO   |     |
| airline_id            | bigint(20)   | YES  | MUL |
| arrival_airport_id    | bigint(20)   | NO   | MUL |
| arrival_date          | datetime     | NO   |     |
| departure_airport_id  | bigint(20)   | NO   | MUL |
| departure_date        | datetime     | NO   |     |
| flight_number         | varchar(255) | NO   |     |
| trip_id               | bigint(20)   | YES  | MUL |
+-----------------------+--------------+------+-----+
```

The scaffolded HTML page for creating a new `Flight`, shown in Figure 1, offers combo boxes for all of the related tables:

Figure 1. Scaffolded HTML page for adding flights



Fine-tuning the user interface

Up to this point, the focus of the m:m discussion has been on how to model the relationship with classes and database tables. I hope you can see that it is as much an art as a science. As a Grails developer, you can take advantage of many subtleties to refine the relationship's behavior and side-effects. Turning now to the user interface, you'll see subtle ways that you can tweak the display of m:m relationships as well.

As I demonstrated in the preceding section, Grails uses select fields by default to display 1:m relationships. This isn't a bad place to start, but you might want to use other HTML controls in different circumstances. Select fields display only the current value; you must drop down the list to see all of the possible values. Although this is the best choice if screen real estate is a scarce commodity, you might decide that making all of the choices visible is a better solution. Radio buttons are good for displaying all possible choices and limiting the selection to a single value. Check boxes display all possible choices and allow for multiselection.

Any of those controls is good for displaying a limited number of choices, but they don't scale well to hundreds or thousands of possible values. For example, if I need to present all of the world's approximately 650 airlines to the end user, none of the standard HTML controls are geared for handling that kind of volume. But here's where the developer's judgment comes into play. For this application, I don't want to display all 650 airlines. I've probably flown fewer than a dozen different airlines in my lifetime. Using a select field to display airline choices will most likely be sufficient for some time.

To see how Grails creates the select field for `Airlines`, type `grails generate-views Flight`. Take a look at grails-app/views/flight/create.gsp. The select field is generated in a single line of code using the `<g:select>` tag. (For a refresher on Grails TagLibs, see last month's article.) Listing 9 shows `<g:select>` fields in action:

Listing 9. <g:select> fields in action

```
<g:select optionKey="id"
          from="${Airline.list()}"
          name="airline.id"
          value="${flight?.airline?.id}" ></g:select>
```

Select **View** > **Source** in your Web browser to see how this gets rendered, as shown in Listing 10:

Listing 10. Rendered select field

```
<select name="airline.id" id="airline.id" >
<option value="1" >UAL - United Airlines</option>
<option value="2" >DAL - Delta</option>
<option value="3" >COA - Continental</option>
</select>
```

The `<g:select>` tag's `optionKey` attribute specifies which field of the *one* class will get stored in the value of the *many* field on the other side of the relationship. The primary key of the Airline table (`airline.id`) shows up as the foreign key in the Flight table. In the select field, notice that `airline.id` is the option value. (The `Airline.toString()` method is called for the display values.) If you'd like to change the sort order of the options, you can change the GORM call from `Airline.list()` to `Airline.listOrderByIata()`, `Airline.listOrderByName()`, or any other field you'd like.

Using Ajax to handle large numbers of options

The default select control was a reasonable choice for displaying a realistic number of airlines. Unfortunately, airports aren't quite as cooperative. I might travel to 40 or 50 different airports in a given year. In my experience, offering more than 15 or 20 choices in a select field begins to get a bit cumbersome.

Luckily, the IATA codes for airports are used extensively in the industry. They show up when I am researching flights. They show up on the receipt when I book the flight. They even show up on the ticket itself. Asking users to type in the IATA code is a reasonable alternative to forcing them to scroll through hundreds of possible airports.

Think back to the `Book` example I introduced at the beginning of this article. Does Amazon.com provide one big select field on the home page that displays all of the books it has in stock? No — it provides a text field where you can type in the book's title, author, or even the International Standard Book Number (ISBN) if you are so inclined. I'll use the same technique here for dealing with airports in my trip planner.

Changing the control from a select field to a text field is easy enough. But before I continue with the mechanics of the solution, I want to spend a moment dealing with its semantics. The `iata` field is a free-form text field, yet I can't simply accept any value the user types in. (The application won't scold you if misspell your name, but it needs to warn you if you mistype an IATA code.) I want this feedback to happen immediately, because nothing's more frustrating than repeatedly submitting an entire HTML form and getting chastised each time for typing in an invalid value.

So I don't want to round-trip the entire form to server just to validate a single field, or download the thousands of possible airport IATA codes to the client each time. The solution is to leave the data on the server and make a fine-grained HTTP request for

the individual field rather than a coarse-grained request for the entire form. This technique is called making an Ajax (Asynchronous JavaScript + XML) request. (See Resources for an introduction to Ajax.)

To Ajax-enable my Grails application, I need to tweak the `AirportController` to accept an Ajax request, and tweak the view to make the Ajax request. I'll start with the `AirportController`.

The `AirportController` already has scaffolded closures for returning a list of `Airports`, as well as showing an individual `Airport`. These existing closures, though, return the values as HTML. I'll add a new closure that returns the raw data. One option would be to serialize the POGO down the wire, but my client is a Web browser. Sadly, JavaScript — not Groovy — is the language that the Web browser speaks. (Mozilla Foundation, are you listening?)

As the *x* in Ajax suggests, I could return XML. If you import the `grails.converters` package in the `AirportController`, returning XML is a one-line affair, as shown in Listing 11:

Listing 11. Returning XML from a controller

```
import grails.converters.*

class AirportController {
  def scaffold = Airport

  def getXml = {
    render Airport.findByIata(params.iata) as XML
  }
}
```

The only problem with this solution is that XML is no more native to JavaScript than it is to Groovy. The benefit of an object-relational mapper such as GORM is that it seamlessly transforms data from a nonnative format (stored in a relational database) into Groovy. The JavaScript equivalent of this exercise is transforming Groovy data into JavaScript Object Notation (JSON) (see Resources). Thankfully, you can do the same one-line conversion into JSON that you can into XML. In Listing 12, I add a bit of error handling to the `getJson` closure, but otherwise it's identical to the `getXml` closure:

Listing 12. Returning JSON from a controller

```
def getJson = {
  def airport = Airport.findByIata(params.iata)

  if(!airport){
    airport = new Airport(iata:params.iata, name:"Not found")
  }

  render airport as JSON
```

```
}
```

To verify that the JSON transformation is working, type `http://localhost:9090/trip/airport/getJson?iata=den` into a Web browser. You should get the response shown in Listing 13. (You might need to select **View** > **Source** in the browser to see the JSON response.)

Listing 13. The JSON response

```
{"id":1,"class":"Airport","city":
   "Denver","country":"US","iata":
"DEN","name":"Denver International Airport","state":"CO"}
```

To return a list of airlines, the process is just as easy: `render Airline.list() as JSON`.

Now that I'm generating JSON, it's time to put it to use. I'll comment out the existing `<g:select>` for `departureAirport` and replace it with the four lines of code in Listing 14:

Listing 14. Replacing the select field with a text field

```
<div id="departureAirportText">[Type an Airport IATA Code]</div>
<input type="hidden" name="departureAirport.id" value="-1"
   id="departureAirport.id"/>
<input type="text" name="departureAirportIata" id="departureAirportIata"/>
<input type="button" value="Find" onClick="get('departureAirport')"/>
```

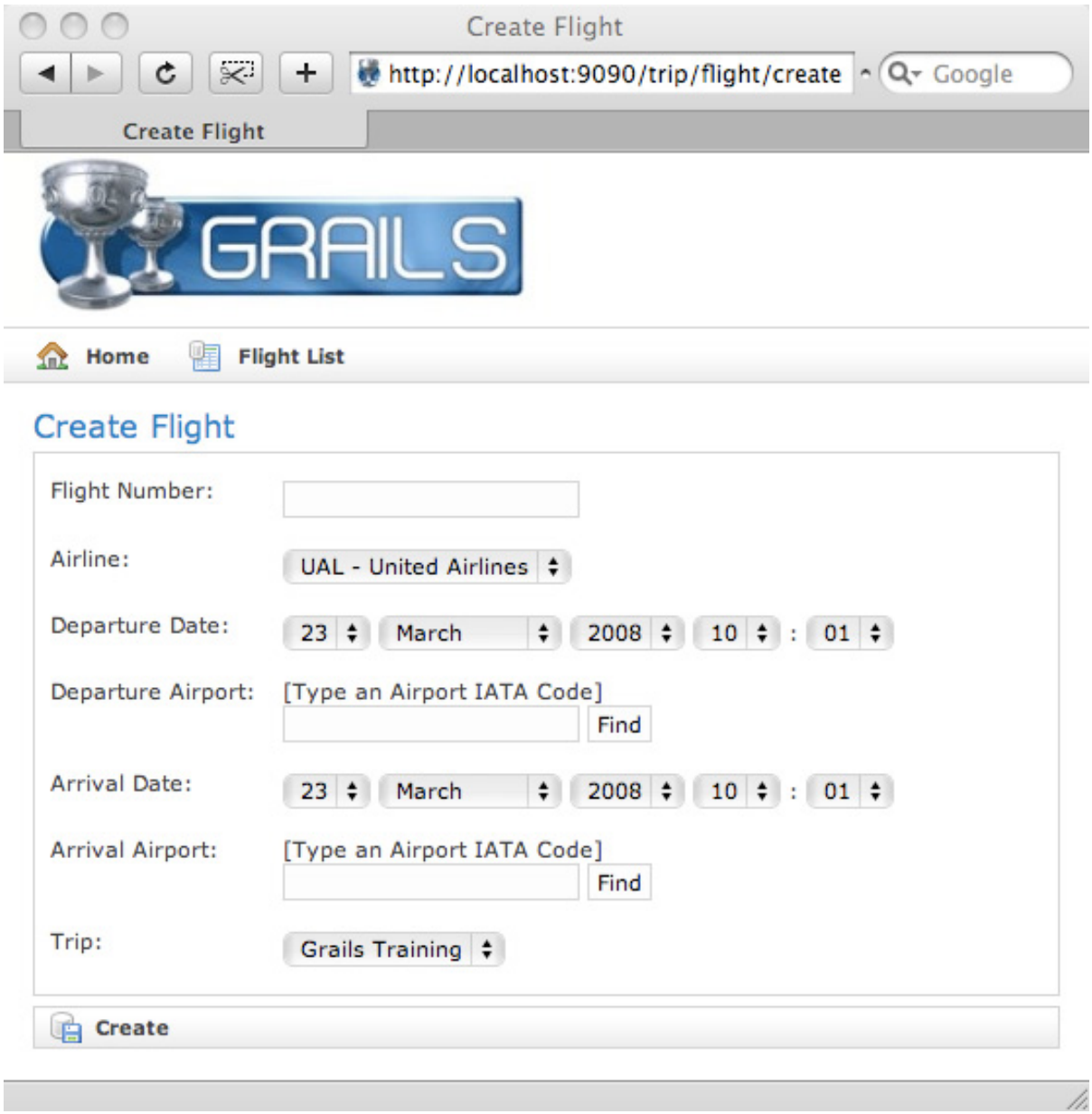
The first line is a read-only display area. Notice that it has an `id`. IDs must be unique across the entire HTML Document Object Model (DOM). I'll use the handle `departureAirportText` to write out the result of the JSON call in just a moment.

`<div>`s aren't sent back to the server when the form is submitted; form controls such as inputs and selects are. The hidden text field gives me a place to store the `id` of the `Airport` when the entire form is submitted back to the server.

The text field named `departureAirportIata` is where the user will type in the IATA code. Supplying identical values for both the name and the ID might not seem very DRY, but the mechanics of HTML require it. The name is what goes back to the server when the form is submitted. The ID is what I'll use for criteria when I call the `getJson` closure.

Finally, the last line is the button that, when clicked, calls a JavaScript function named `get`. I'll show you the implementation of the `get` function in just a moment. For now, Figure 2 shows what the new form looks like:

Figure 2. The revised form



Using Prototype for Ajax calls

Grails ships with a JavaScript library called Prototype (see Resources). Prototype provides a common way to make Ajax calls that is compatible across all of the major browsers. The `get` function simply builds up the URL that you typed into the browser earlier, and then makes the asynchronous call back to the server. If the call is successful (returning an HTTP 200), the `update` function is called. Listing 15 uses Prototype for the Ajax call:

Listing 15. Using Prototype for the Ajax call

```
<g:javascript library="prototype" />
<script type="text/javascript">
  function get(airportField){
    var baseUrl = "${createLink(controller:'airport', action:'getJson')}"
    var url = baseUrl + "?iata=" + $F(airportField + "Iata")
    new Ajax.Request(url, {
      method: 'get',
      asynchronous: true,
      onSuccess: function(req) {update(req.responseText, airportField)}
    })
  }

  ...
</script>
```

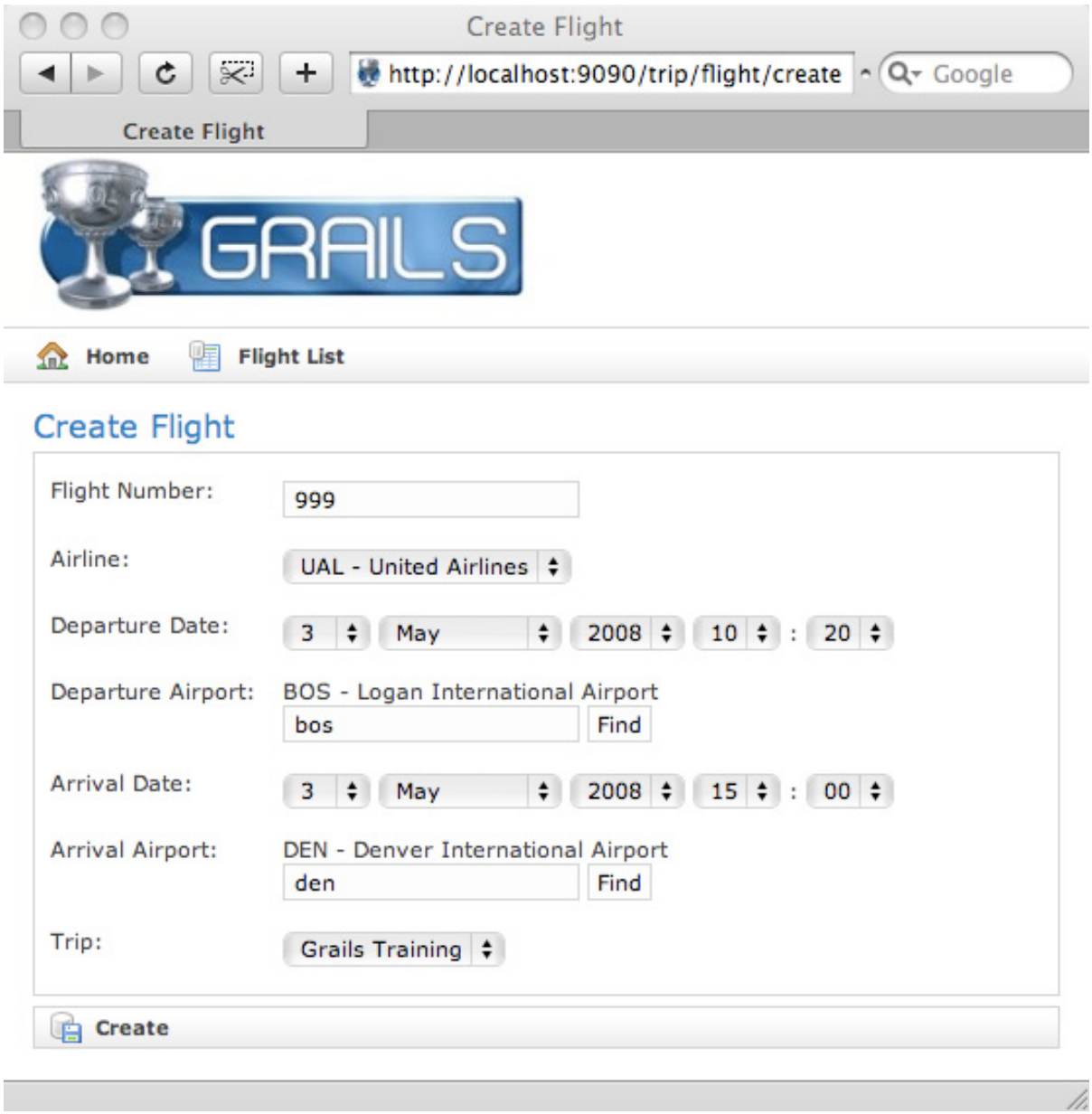
The `update` function reads the result of the JSON call, updates the display of the `<div>`, and changes the value of the hidden field to the airport's primary key if it is found, or -1 if it cannot be found. Listing 16 shows the `update` function:

Listing 16. Updating the fields with JSON data

```
function update(json, airportField){
  var airport = eval( "(" + json + ")" )
  var output = $(airportField + "Text")
  output.innerHTML = airport.iata + " - " + airport.name
  var hiddenField = $(airportField + ".id")
  airport.id == null ? hiddenField.value = -1 : hiddenField.value = airport.id
}
```

Figure 3 shows what the `Flight` form looks like after a couple of successful Ajax calls:

Figure 3. Form populated by Ajax calls



Client-side validation

Finally, a little bit of client-side validation should be done to ensure that bad values for `departureAirport` and `arrivalAirport` aren't submitted back to the server. (It's literally impossible to enter bad values when you present the user with a select field, a set of radio buttons, or a set of check boxes. Because I'm allowing users to type in free-form text, I need to be a bit more wary about the quality of their input.)

Add `onSubmit` to the `g:form` tag:

```
<g:form action="save" method="post" onsubmit="return validate()" >
```

If `validate` returns `true`, the form is submitted to the server. If it returns `false`, the submission is cancelled. Listing 17 shows the `validate` function:

Listing 17. The validate function

```
function validate(){
  if( $F("departureAirport.id") == -1 ){
    alert("Please supply a valid Departure Airport")
    return false
  }

  if( $F("arrivalAirport.id") == -1 ){
    alert("Please supply a valid Arrival Airport")
    return false
  }

  return true
}
```

If you are thinking that moving from a select field to a text field requires a little more work, I agree with you. I didn't make this change to make it easier on myself — I was trying to make it easier on the end user. But consider this: the scaffolding that Grails provides did so much of the initial work for me that I don't mind doing a bit of fine-tuning here and there. The scaffolding isn't meant to be a finished product. It is meant to get all of the mind-numbingly boring stuff out of the way so that you can focus on the more interesting parts of the puzzle.

Conclusion

More than the simple mechanics of many-to-many relationships, I hope that you picked up on some of the subtleties of creating them as well. Sometimes you want the deletes to cascade; other times you don't. Any time you think that you have a simple m:m relationship between two classes, a third class is probably waiting to be discovered.

On the presentation side, select fields are the default choice for m:m relationships, but they aren't the only choice. Radio buttons and check boxes are options to consider if you have fewer choices. When it comes to many choices, a text field and Ajax can do the trick.

Next month I'll show you how to put these flights on an interactive Google map. I'll also talk about Grails services. Until then, have fun mastering Grails.

Resources

Learn

- *Mastering Grails*: Read more in this series to gain a further understanding of Grails and all you can do with it.
- Grails: Visit the Grails Web site.
- Grails Framework Reference Documentation: Direct from the source.
- *Groovy Recipes* (Pragmatic Bookshelf, March 2008): Learn more about Groovy and Grails in Scott Davis' latest book.
- "Ajax overhaul, Part 1: Retrofit existing sites with Ajax and jQuery" (Brian J. Dillard, developerWorks, March 2008): Read the first article in a series on overhauling existing sites with Ajax.
- "Mastering Ajax, Part 10: Using JSON for data transfer" (Brett McLaughlin, developerWorks, March 2007): Find out how JSON can makes moving data and objects around in your applications easier.
- Prototype: Learn more about the Prototype JavaScript framework.
- IATA airport codes and airline codes: The three-letter codes the airline industry uses to identify airports and airlines.
- *Practically Groovy*: This developerWorks series is dedicated to exploring the practical uses of Groovy and teaching you when and how to apply them successfully.
- Groovy: Learn more about Groovy at the project Web site.
- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- Technology bookstore: Browse for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Grails: Download the latest Grails release.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Scott Davis
Scott Davis is an internationally recognized author, speaker, and software developer. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At Work*.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.