CHAPTER 11

iOS : Pushing Changes Back to the Server

In this chapter, we're going to continue the work of the Six Bookmarks application by implementing the code that allows the user to modify the bookmarks and then push them back to the server. We'll start by looking at the user interface.

Configuring Bookmarks

We're going to base the user interface for our configuration window on a **Table View** control. The Table View control is perhaps the most common user interface metaphor on iOS devices—it seems to me that every application I ever download and use on iPhone or iPad is based on tables!

We'll jump right in and start designing the user interface. Similar to the Android application, what we're going to do is build a form that shows the current bookmarks. Selecting a bookmark will take us into a "singleton edit" page. We'll also be able to add new bookmarks, edit (i.e., delete) bookmarks, and manually initiate a sync routine.

Add a new **UIViewController Subclass** item to the project called `SBConfigureView`. When the view is created, double-click on the `SBConfigureView.xib` file to start editing the UI. Once there, add the following elements:

- Add a **Navigation Bar** control, changing its text to **Configure**.
- Add a **Toolbar** control. Change the default button text to **Done**, and add two new buttons. When you add the buttons, use the **Attributes Inspector** to change the **Identifier** of one to be **Edit** and the other to be **Refresh**. The **Refresh** button will automatically adopt a built-in image. Also add a **Flexible Space Bar Button Item** to push the **Edit** and **Refresh** buttons to the right of the bar.
- To the navigation bar, add a button and set its **Identifier** button to be **Add**. This, too, will adopt a built-in image.
- Finally, add a **Table View** control to the main area of the form.

You should end up with something like Figure 11-1.

In terms of the code, we need to maintain a property for referencing the table, and a property for holding a list of bookmarks in memory. We also need event handlers to receive notifications of the done, edit, sync/refresh, and add buttons. (We can implement `handleClose` straightaway, as we know what we're doing with this—we just need to go back to the navigator form.) We also need to change the base class to be our special `SBViewController` class and also implement the `UITableViewDelegate` and `UITableViewDataSource` protocols.

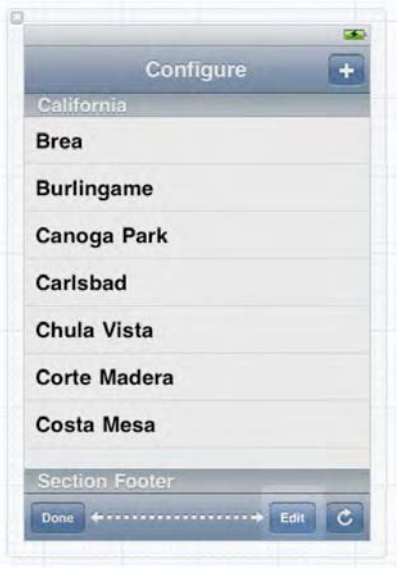

Figure 11-1. *The design of the configuration form*

Here's the definition:

```
// SBConfigureView.h
#import <UIKit/UIKit.h>
#import "SBViewController.h"
#import "SBSync.h"

@interface SBConfigureView : SBViewController <UITableViewDelegate, UITableViewDataSource> {

        NSMutableArray *bookmarks;
        UITableView *table;
        SBSync *syncEngine;

}

@property (nonatomic, retain) NSMutableArray *bookmarks;
@property (nonatomic, retain) IBOutlet UITableView *table;
@property (nonatomic, retain) SBSync *syncEngine;

-(IBAction)handleClose:(id)sender;
-(IBAction)handleSync:(id)sender;
-(IBAction)handleEdit:(id)sender;
-(IBAction)handleAdd:(id)sender;

@end
```

And here's the implementation:

```
// SBConfigureView.m
#import "SBConfigureView.h"
#import "MyClasses.h"

@implementation SBConfigureView

@synthesize bookmarks;
@synthesize table;
@synthesize syncEngine;

-(IBAction)handleClose:(id)sender
{
        // go back to the navigator…
        [self.owner openNavigator];
}

-(IBAction)handleSync:(id)sender
{
        [SBMessageBox show:@"TBD"];
}

-(IBAction)handleEdit:(id)sender
{
        [SBMessageBox show:@"TBD"];
}

-(IBAction)handleAdd:(id)sender
{
        [SBMessageBox show:@"TBD"];
}

// Some wizard-generated code has been ommitted for brevity

-(void)dealloc
{
        [syncEngine release];
        [bookmarks release];
        [table release];
    [super dealloc];
}

@end
```

When we put together the view in the user interface editor, we didn't wire up the controls and events using the **Connections Inspector**. We can do this now to wire up the following:

- **selector** event of **Add** button goes to `handleAdd` method.
- **selector** event of **Done** button goes to `handleClose` method.
- **selector** event of **Edit** button goes to `handleEdit` method.
- **selector** event of **Sync/Refresh** button goes to `handleSync` method.
- **table** property on **File's Owner** goes to the **Table View** control.
- *Very important*, the **dataSource** member of the **Table View** control has to be bound back to **File's Owner**. This is the mechanism through which the view discovers its data.
- *Also very important*, the **delegate** member of the **Table View** control also has to be bound back to **File's Owner**. This is the mechanism through which the view raises user interaction events.

That's essentially it for the user interface basics—we'll look now at how to put data onto the table.

Putting Data on the Table

The approach that Cocoa Touch takes with the table view is that we wire up the **dataSource** member of the Table View control and then wait for events to be called. The minimum two events that we need to support are `tableView:numberOfRowsInSection` and `tableView:cellForRowAtIndexPath`. We'll look at those methods in a moment—our immediate concern is implementing `refreshView` so that our `bookmarks` property gets populated with the visible bookmarks from the SQLite database.

We've already seen how we can use the `getBookmarksForDisplay` method on the navigator. We'll reuse this method here, but we'll capture the results in a property for later use.

The listing follows—one thing to mention beforehand is that if we happen to load a new list of bookmarks, we need to tell the table to reload. We'll keep track of this state in a local variable called `doReload` and call `reloadData` as needed.

```
// Add method to SBConfigureView.m
-(void)refreshView
{
        // do we have bookmarks?
        BOOL doReload = FALSE;
        if(self.bookmarks != nil)
                doReload = TRUE;

        // get our bookmarks...
        NSMutableArray *theBookmarks = nil;
        NSError *err = [SBBookmark getBookmarksForDisplay:&theBookmarks];
        if(err != nil)
                [SBMessageBox showError:err];
        else
        {
                // keep track...
                self.bookmarks = theBookmarks;
        }
```

```
        // reload...
        if(doReload)
                [self.table reloadData];
}
```

`tableView:numberOfRowsInSection` simply needs to return the number of bookmarks for display. `tableView:cellForRowAtIndexPath` is more interesting—this method requires us to return back a cell for it to show on the view. The way it does this is quite interesting. It assumes that memory on the device is limited (which it is), and so if you have a long list of items (many tens up to many thousands and beyond), it will attempt to recycle and reuse cells that have become no-longer-visible—for example, when the user scrolls the view. To do this, each cell is given a key, which is a classifier of type rather than instance—it's very common to find Table View instances in applications that use a single table but multiple types of cells. When we are asked for a cell at a given path (which in our case will be a row ordinal), we can ask the Table View control to recycle one of those cells for us. If it can't, we'll create one and return it. Either way, we'll update the text on the cell that we either got given or created and hand it back. The recycling process works by providing an arbitrary string to the type of cell. (It's common in iOS applications to have multiple types of cell on a table.) Our arbitrary string will be stored in a constant variable called `cellId` and will have value `BookmarkKey`.

Here's the implementation:

```
// Add methods to SBConfigureView.m…
-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
        return [self.bookmarks count];
}

-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:↩
(NSIndexPath *)indexPath
{
        static NSString *cellId = @"BookmarkKey";

        // find one...
        UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:cellId];
        if(cell == nil)
                cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault↩
 reuseIdentifier:cellId] autorelease];

        // set...
        NSUInteger row = [indexPath row];
        cell.textLabel.text = [[self.bookmarks objectAtIndex:row] name];
        return cell;
}
```

We are very nearly at a point where we can run this and get some data on the screen, which is a good indicator as to how simple working with lists is in iOS. To get it working, we first need to add a method called `openConfiguration` to our `SixBookmarksAppDelegate` class, which is patterned on the `openLogon` and `openNavigator` methods that we built earlier. Here's the code:

```
// Add method to SixBookmarksAppDelegate…
// <…> Needs declaration in header
```

```
-(void)openConfiguration
{
        // find a preexisting view...
        SBViewController *view = [self getController:[SBConfigureView class]];
        if(view == nil)
                view = [[SBConfigureView alloc] initWithNibName:@"SBConfigureView"↩
bundle:[NSBundle mainBundle]];

        // show the view...
        [self showView:view];
}
```

Previously, our method on `SBNavigatorView` for handling the **Configure** button click simply displayed a message box. We can change this now to physically open the form:

```
// Modify method in SBNavigatorView…
-(IBAction)handleConfigure:(id)sender
{
        [self.owner openConfiguration];
}
```

Run the application and you will be able to click through, open the new form, and see the bookmarks. Figure 11-2 illustrates.

Figure 11-2. *The configuration form*

Sorting the Bookmarks

If you recall from the Android chapter, when we displayed the bookmarks in its Table View equivalent, we ended up with bookmarks sorted by name rather than by ordinal. This would undoubtedly lead to user confusion, and so we sorted the bookmarks before display. We need to do the same thing.

The way you do a sort in Objective-C is functional, but a little odd. What we have to do is provide a "function pointer" of sorts through to the sort routine. When the sort routine chugs along sorting, it will call this function pointer to compare two items. We're going to sort by ordinal, and so we need to add a method called `ordinalComparer` to `SBBookmark`. Here's the code:

```
// Add method to SBBookmark.m
-(NSComparisonResult)ordinalComparer:(id)otherObject
{
        SBBookmark *other = (SBBookmark *)otherObject;
        if(self.ordinal < other.ordinal)
                return NSOrderedAscending;
        else if(self.ordinal > other.ordinal)
                return NSOrderedDescending;
        else
                return NSOrderedSame;
}
```

As you can see, this works by giving the method another bookmark instance to compare.

To call this method, we use the special Objective-C syntax of `@selector` to create a "function pointer" (a "method pointer," really) to the `ordinalComparer` method which we can then pass into the `sortUsingSelector` method of the `NSMutableArray`. Here's the implementation:

```
// Modify method in SBConfigureView.m…
-(void)refreshView
{
        // do we have bookmarks?
        BOOL doReload = FALSE;
        if(self.bookmarks != nil)
                doReload = TRUE;

        // get our bookmarks...
        NSMutableArray *theBookmarks = nil;
        NSError *err = [Bookmark getBookmarksForDisplay:&theBookmarks];
        if(err != nil)
                [SBMessageBox showError:err];
        else
        {
                // keep track...
                self.bookmarks = theBookmarks;

                // sort by ordinal...
                [theBookmarks sortUsingSelector:@selector(ordinalComparer:)];
        }
```

```
        // reload...
        if(doReload)
                [self.table reloadData];
}
```

Now if you run the code again, you will end up with a list sorted in ordinal order. Figure 11-3 illustrates.

Figure 11-3. *The configuration view, now with items correctly ordered*

Singleton View

We now need a singleton view for editing single bookmarks. To the project, add a new **UIViewController Subclass** item called **SBConfigureSingletonView**. Once added, open the `SBConfigureSingletonView.xib` file in the usual way.

To the form, add the following:

- A **Navigation Bar** control, with the title changed to **Configure**
- A **Bar Button** Item control on the left of the navigation bar with its **Identifier** set to **Cancel**
- Another **Bar Button** Item control on the right of the navigation bar with its **Identifier** set to **Save**
- Two **Text Field** controls—set the **Placeholder** to **Name** and **URL** respectively.

Figure 11-4 illustrates.

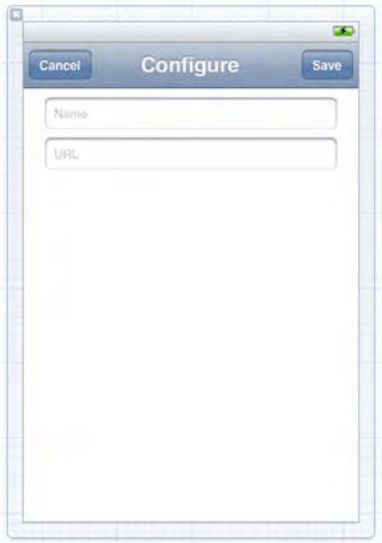

Figure 11-4. *The designer view for "configure singleton"*

Again in the usual way, we need to go back into the code to add properties for the text boxes and methods to receive event notifications for the buttons. In addition, the form will also need to hold a reference to the current `SBBookmark` instance and also have a method called `setOrdinal`. This `setOrdinal` method will be used to initialize the view.

Here's the definition:

```
// SBConfigureSingletonView.h
#import <UIKit/UIKit.h>
#import "SBViewController.h"
#import "SBBookmark.h"

@interface SBConfigureSingletonView : SBViewController {

        SBBookmark *bookmark;
        UITextField *textName;
        UITextField *textUrl;

}

@property (nonatomic, retain) SBBookmark *bookmark;
@property (nonatomic, retain) IBOutlet UITextField *textName;
@property (nonatomic, retain) IBOutlet UITextField *textUrl;
```

```
-(void)setOrdinal:(int)theOrdinal;

-(IBAction)handleCancel:(int)sender;
-(IBAction)handleSave:(int)sender;

@end
```

And then the implementation:

```
// SBConfigureSingletonView.m
#import "SBConfigureSingletonView.h"
#import "MyClasses.h"

@implementation SBConfigureSingletonView

@synthesize bookmark;
@synthesize textName;
@synthesize textUrl;

-(void)setOrdinal:(int)theOrdinal
{
        // we'll do this later…
}

-(IBAction)handleCancel:(int)sender
{
        [self.owner openConfiguration];
}

-(IBAction)handleSave:(int)sender
{
        [SBMessageBox show:@"TBD"];
}

// Some wizard-generated code has been omitted.

-(void)dealloc
{
        [bookmark release];
        [textName release];
        [textUrl release];
    [super dealloc];
}

@end
```

With the class stubbed out in the way, we can go back into the user interface editor and bind up the methods and properties, specifically:

- **selector** event of **Cancel** button goes to `handleCancel` method.
- **selector** event of **Save** button goes to `handleSave` method.
- **textName** property of **File's Owner** goes to the **Name** text box.
- **textUrl** property of **File's Owner** goes to the **URL** text box.

Next we can add an `openBookmarkSingleton` method to the `SixBookmarksAppDelegate` class that will open the view and call the `setOrdinal` method. (We haven't implemented `setOrdinal` properly yet, so all this will do is show a message box.)

Here's the implementation:

```
// Add method to SixBookmarksAppDelegate.m
// <…> Needs declaration in header
-(void)openBookmarkSingleton:(int)theOrdinal
{
        // find a preexisting view...
        SBViewController *view = [self getController:[SBConfigureSingletonView class]];
        if(view == nil)
                view = [[SBConfigureSingletonView alloc] initWithNibName:↩
@"SBConfigureSingletonView" bundle:[NSBundle mainBundle]];

        // set...
        [((SBConfigureSingletonView *)view) setOrdinal:theOrdinal];

        // show the view...
        [self showView:view];
}
```

OK, so at this point, we can view the view and show it, but how do we configure the table so that we can click on it and make all this happen?

Each cell on the table can be configured with a "disclosure indicator." You will have undoubtedly seen this used in iOS applications before—it is a small arrow to the right of each item that tells you that you can "drill into" the item for more information. (There is a variation on this called a "disclosure button," which we're not going to use. A disclosure button lets you do something with an item, e.g., make a phone call or start an email, but it isn't designed to indicate that an item can be drilled into.)

This is very easy to do—all we have to do when we configure our table cell is set the **cellAccessory** property. Here's the code:

```
// Modify method in SBConfigureView.m…
-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:↩
(NSIndexPath *)indexPath
{
        static NSString *cellId = @"BookmarkKey";

        // find one...
        UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:cellId];
        if(cell == nil)
                cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault↩
 reuseIdentifier:cellId] autorelease];
```

```
        // set...
        NSUInteger row = [indexPath row];
        cell.textLabel.text = [[self.bookmarks objectAtIndex:row] name];

        // set the disclosure indicator...
        cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

        // return...
        return cell;
}
```

If you run the code now, you can see the disclosure indicators. Figure 11-5 illustrates.

Figure 11-5. *The configuration view, now with disclosure indicators against each item*

We can now implement the `tableView:didSelectRowAtIndexPath` method and open the singleton editor. Here's the implementation:

```
// Add method to SBConfigureView.m…
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
```

```
        // find it...
        SBBookmark *bookmark = [self.bookmarks objectAtIndex:[indexPath row]];
        [self.owner openBookmarkSingleton:[bookmark ordinal]];
}
```

Run the project again, and you can now click on the cells in the table to bring up the singleton editing UI. (The name and URL fields will not be populated—we're going to do this shortly.) Figure 11-6 illustrates.

Figure 11-6. *The "configure singleton" view being displayed, but with no data*

Editing a Bookmark

The code to edit a bookmark is reasonably straightforward. We have to implement `setOrdinal` so that it will either load a bookmark from disk or create a new one for us, implement `refreshView` so that the text boxes are updated, handle validation of the text contained within when the user clicks the **Save** button, and if saving is OK, commit the changes to the database.

Looking at `setOrdinal` first, the approach here needs to be that we'll create a `SBSqlFilter` instance to try to load a bookmark from disk. (This filter will exclude deleted bookmarks with the given ordinal.) If there isn't a bookmark on disk, we'll create one and set its ordinal so that it goes into the correct location when we do come to save it.

Here's the implementation (we'll build `getByOrdinal` in a moment):

```
// Replace setOrdinal method in SBConfigureSingletonView…
-(void)setOrdinal:(int)theOrdinal
{
        // try and load it...
        SBBookmark *theBookmark = nil;
        NSError *err = [SBBookmark getByOrdinal:theOrdinal bookmark:&theBookmark];
        if(err != nil)
                [SBMessageBox showError:err];
        else
        {
                // create...
                if(theBookmark == nil)
                {
                        theBookmark = [[[SBBookmark alloc] init] autorelease];

                        // set the ordinal (so that we save into the right place)...
                        theBookmark.ordinal = theOrdinal;
                }

                // set...
                self.bookmark = theBookmark;
        }

        // update...
        self.refreshView;
}
```

The `getByOrdinal` method on `SBBookmark` is going to need a new method to be added to `SBSqlFilter`, namely a method that returns a single entity back as opposed to a collection. The logic is that it will fetch a collection and return back the topmost item. (Although this is a bit hacky—it should check that exactly one or exactly zero items are returned and return the solo item if one is found—I know from experience this approach tends to be fine from a practical perspective.)

```
// Add method to SBSqlFilter.m…
// <…> Needs declaration in header
-(NSError *)executeEntity:(SBEntity **)theEntity
{
        // reset...
        *theEntity = nil;

        // run...
        NSMutableArray *entities = nil;
        NSError *err = [self executeEntityCollection:&entities];
```

```
        // get...
        if([entities count] > 0)
                *theEntity = [entities objectAtIndex:0];

        // return...
        return err;
}
```

`getByOrdinal` on `SBBookmark` can then configure a filter and use this method to return back a single bookmark from the database. The filter will be configured to select back the item that has a given ordinal, but that is not marked as locally deleted. Here's the implementation:

```
// Add method to SBBookmark.m…
// <…> Needs declaration in header
+(NSError *)getByOrdinal:(int)theOrdinal bookmark:(SBBookmark **)theBookmark
{
        SBSqlFilter *filter = [[SBSqlFilter alloc] initWithType:[SBBookmark class]];
        [filter.constraints addObject:[[SBSqlFieldConstraint alloc]
initWithFieldAndValue:[filter.entityType getField:@"ordinal"]

value:[NSNumber numberWithInt:theOrdinal]]];
        [filter.constraints addObject:[[SBSqlFieldConstraint alloc]➥
 initWithFieldAndValue:[filter.entityType getField:@"localdeleted"]

value:[NSNumber numberWithBool:FALSE]]];

        // return...
        NSError *err = [filter executeEntity:theBookmark];
        [filter release];
        return err;
}
```

Finally for this section, if we implement `refreshView`, we can see the data on the screen. Here's the implementation:

```
// Add method to SBConfigureSingletonView.m…
-(void)refreshView
{
        // set...
        if(self.bookmark != nil)
        {
                [self.textName setText:[self.bookmark name]];
                [self.textUrl setText:[self.bookmark url]];
        }
```

```
        else
        {
                [self.textName setText:nil];
                [self.textUrl setText:nil];
        }
}
```

Figure 11-7 illustrates a successful result.

Figure 11-7. *The "configure singleton" view, now with data*

Capturing and Committing Changes

The next step is to implement the singleton view so that we can enter a new name and URL and commit it to the database. As part of this work, we'll have to modify the `SBEntityChangeProcessor` so that it is able to handle updates—if you recall, when we built this in the last chapter, it could only handle inserts.

The actual code to handle the save operation is very straightforward. Here's the implementation to add to `SBConfigureSingletonView`:

```
// Modigy handleSave method in SBConfigureSingletonView…
-(IBAction)handleSave:(int)sender
{
        // get the values...
        NSString *name = self.textName.text;
        NSString *url = self.textUrl.text;

        // can we?
        SBErrorBucket *errors = [[SBErrorBucket alloc] init];
        if(name == nil || [name length] == 0)
                [errors addError:@"Name is required."];
        if(url == nil || [url length] == 0)
                [errors addError:@"URL is required."];

        // ok...
        if(!([errors hasErrors]))
        {
                // save the regular bits...
                [self.bookmark setName:name];
                [self.bookmark setUrl:url];

                // set the flags...
                [self.bookmark setLocalModified:TRUE];
                [self.bookmark setLocalDeleted:FALSE];

                // save...
                [self.bookmark saveChanges];

                // ok...
                [self.owner openConfiguration];
        }
        else
                [SBMessageBox show:[errors errorsAsString]];

        // release...
        [errors release];
}
```

You can see from the code that we are reusing the `SBErrorBucket` class that we built before. Also note how when we have a successful validation, we set the `localModified` property to `TRUE` and the `localDeleted` property to `FALSE`. We'll need this when we complete the synchronization routine and commit the changes back to the server.

If you run the application and attempt to modify a bookmark, the application will crash, as we have not yet added the capability to run `UPDATE` queries against the database. (You can discover the cause of the crash by using the debugger's output console.) Let's build the update capability now.

What we're looking to do is build an `UPDATE` query like this one:

```
UPDATE Bookmark SET Name=?, Url=?, LocalModified=?, LocalDeleted=? WHERE BookmarkId=?
```

Recall that the entity is able to tell us which columns have been modified. We'll use this capability in the `handleUpdate` method of `SBEntityChangeProcessor` to build up a SQL query. We'll also use the `SBEntityType` class's ability to return back to us the key field, as we obviously need to constrain the update query to operate only against the bookmark that we wish to update. Here's the implementation:

```
// Replace method in SBEntityChangeProcessor…
-(void)update:(SBEntity *)entity
{
        // create...
        SBEntityType *et = self.entityType;

        // do we have a table?
        SBDBHelper *db = [[SBRuntime current] getDatabase];
        [db ensureTableExists:et];

        // create a statement...
        NSMutableString *builder = [NSMutableString string];
        SBSqlStatement *sql = [[SBSqlStatement alloc] init];

        // header...
        [builder appendString:@"UPDATE "];
        [builder appendString:et.nativeName];
        [builder appendString:@" SET "];
        BOOL first = TRUE;
        for(SBEntityField *field in et.fields)
        {
                if([entity isFieldModified:field])
                {
                        if(first)
                                first = FALSE;
                        else
                                [builder appendString:@", "];

                        // name...
                        [builder appendString:field.nativeName];
                        [builder appendString:@"=?"];

                        // param...
                        [sql addParameter:[entity getValue:field]];
                }
        }

        // constrain by the key field...
        SBEntityField *keyField = [et getKeyField];
        [builder appendString:@" WHERE "];
        [builder appendString:keyField.nativeName];
        [builder appendString:@"=?"];
        [sql addParameter:[entity getValue:keyField]];
```

```
        // attach...
        [sql setCommandText:builder];

        // run...
        @try
        {
                BOOL ok = [db execNonQuery:sql];
                if(!(ok))
                {
                        NSError *err = [db getLastError];
                        @throw [NSException exceptionWithName:[[self class] description]↩
reason:[SBErrorHelper formatError:err] userInfo:nil];
                }
        }
        @finally
        {
                // release...
                [sql release];
        }
}
```

Now if you run the application, you will be able to update a bookmark. Figure 11-8 and Figure 11-9 illustrate.

Figure 11-8. *Editing a bookmark*

Figure 11-9. *The configuration screen showing the changed data*

Implementing the Delete Method

We're not going to be issuing `DELETE` statements against individual bookmarks in this project, but we can delete from the UI. This simply issues an `UPDATE` operation to mark the entity as deleted. However, for completeness, here's the `delete` implementation:

```
// Replace method in SBEntityChangeProcessor.m…
-(void)delete:(SBEntity *)entity
{
        // create...
        SBEntityType *et = [self entityType];

        // do we have a table?
        SBDBHelper *db = [[SBRuntime current] getDatabase];
        [db ensureTableExists:et];

        // create a statement...
        NSMutableString *builder = [NSMutableString string];
        SBSqlStatement *sql = [[SBSqlStatement alloc] init];
```

```
        // create a statement...
        SBEntityField *keyField = [et getKeyField];
        [builder appendString:@"DELETE FROM "];
        [builder appendString:et.nativeName];
        [builder appendString:@" WHERE "];
        [builder appendString:keyField.nativeName];
        [builder appendString:@"=?"];
        [sql addParameter:[entity getValue:keyField]];

        // attach...
        [sql setCommandText:builder];

        // run...
        @try
        {
                BOOL ok = [db execNonQuery:sql];
                if(!(ok))
                {
                        NSError *err = [db getLastError];
                        @throw [NSException exceptionWithName:[[self class] description]➥
 reason:[SBErrorHelper formatError:err] userInfo:nil];
                }
        }
        @finally
        {
                // release...
                [sql release];
        }
}
```

Adding a Bookmark

Now that we can change bookmarks, we need to be able to add or delete them. We'll look at add first.

The "add" operation shouldn't require us to do too much work. When we built the singleton view, the `setOrdinal` method would create a new `SBBookmark` instance if a bookmark with the given ordinal did not exist. When we have a new bookmark and call `saveChanges` on it, the existing functionality of the `SBEntityChangeProcessor` should issue an `INSERT` statement. In fact, the only tricky bit is finding a blank ordinal to set. We'll add a method called `getOrdinalToAdd`, which will create an array of six `BOOL` values, set them all to `FALSE`, and then walk the bookmarks setting values into the array to `TRUE` based on ordinal value. The routine then looks back through the array for the first `FALSE` value, indicating an empty slot. (Personally, this routine is why I'm not mad keen on Objective-C as a language—this is some pretty complicated code for doing something very easy.) If we get an ordinal, we can defer to `SBConfigureSingletonView` to edit it, otherwise we'll display an error. Here's the implementation:

```
// Add and change methods in SBConfigureView.m…
// <…> Needs declaration in header
-(int)getOrdinalToAdd
{
        NSMutableArray *array = [NSMutableArray array];
        for(int index = 0; index < 6; index++)
                [array addObject:[NSNumber numberWithBool:FALSE]];
        for(SBBookmark *bookmark in self.bookmarks)
                [array replaceObjectAtIndex:[bookmark ordinal] withObject:[NSNumber↩
 numberWithBool:TRUE]];

        // walk...
        for(int index = 0; index < 6; index++)
        {
                NSNumber *value = [array objectAtIndex:index];
                if([value boolValue] == FALSE)
                        return index;
        }

        // return...
        return -1;
}

-(IBAction)handleAdd:(id)sender
{
        int ordinal = [self getOrdinalToAdd];
        if(ordinal != -1)
                [self.owner openBookmarkSingleton:ordinal];
        else
                [SBMessageBox show:@"No more bookmarks can be added."];
}
```

You can now run the application and add a new bookmark. Figure 11-10 illustrates.

Figure 11-10. *The configuration screen, having added a new bookmark*

Deleting Bookmarks

To round off the operations, we'll implement the delete functionality. This is going to take advantage of functionality already in the Table View control. You will most likely already have seen this functionality in other iPhone applications—you can put the table in edit mode and see little red "wheels" to the left of each item that allow you to access a delete button. To be truly iOS standards compliant, we need to be able to do more "edit mode," such as reordering the items. However, I'm calling this as "out of scope," and we'll just look at handling the delete operations.

Turning the grid into "edit" mode is easy—all we have to do is call the `setEditing` method. Here's the implementation:

```
// Replace handleEdit method on SBConfigureView.m
-(IBAction)handleEdit:(id)sender
{
        [self.table setEditing:TRUE animated:TRUE];
}
```

If you run the project now, you can access "edit" mode and from there access a **Delete** button by turning the wheel. Figure 11-11 illustrates.

Figure 11-11. *The configuration screen in "edit" mode*

One question that occurs is how do we exit edit mode? iOS applications tend towards having a minimal set of buttons; hence my proposal is that we rework the **Done** button so that if we're in "edit" mode, we'll return to "normal" mode, whereas if we're in "normal" mode, we'll quit. Here's the implementation:

```
// Replace handleClose method in SBConfigureView.m and add stopEditing method…
-(IBAction)handleClose:(id)sender
{
        // are we editing?
        if(self.table.editing)
                [self stopEditing];
    else
                [self.owner openNavigator];
}

// <…> Needs declaration in header…
-(void)stopEditing
{
        [self.table setEditing:FALSE animated:TRUE];
}
```

Now if you run the project you can enter and exit "edit" mode at will.

When the user clicks a **Delete** button, the `tableView:commitEditingStyle` method will be called. When we detect this has happened, all we have to do is set the `localDeleted` property of the entity to `TRUE` and commit the change to the database. When we next retrieve bookmarks from the view, the filter will exclude ones that are marked as locally deleted, and it will appear as having been deleted. Remember that we cannot physically delete the bookmark because we need to keep track of the fact that it has to be deleted from the server.

Here's the implementation of `tableView:commitEditingStyle`:

```
// Add method to SBConfigureView.m…
-(void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)➥
style forRowAtIndexPath:(NSIndexPath *)indexPath
{
        // great - get rid of it...
        SBBookmark *bookmark = [self.bookmarks objectAtIndex:[indexPath row]];
        [bookmark setLocalDeleted:TRUE];
        [bookmark saveChanges];

        // update...
        [self stopEditing];
        [self refreshView];
}
```

If you run the project at this point, you can delete the bookmarks from the local database. Figure 11-12 and Figure 11-13 illustrate.

Figure 11-12. *The configuration view with a deleted bookmark*

Figure 11-13. *The navigator view showing the deleted bookmark*

Manually Syncing

We've almost finished the section covering the user interface—in the next section, we'll move onto physically pushing the changes back up to the server. Before we finish the section, we'll wire up the manual sync operation. All this will do is call the `doSync` method on the `SBSync` class. This will call back to us via `syncOk` and `syncFailed`, which we also need to add. Here's the implementation:

```
// Replace method in SBConfigureView.m and add syncOk and syncFailed methods…
-(IBAction)handleSync:(id)sender
{
        if(self.syncEngine == nil)
                self.syncEngine = [[SBSync alloc] init];
        [self.syncEngine doSync:(SBSyncCallback *)self];
}

-(void)syncOk
{
        [self refreshView];
        [SBMessageBox show:@"Sync OK."];
}
```

```
-(void)syncFailed:(NSError *)err
{
        // nope...
        [SBMessageBox showError:err];
}
```

One thing to note on that operation is that we demand-initialize the `syncEngine` property when we need it.

If we run the project and click the **Sync/Refresh** button, the `doSync` operation will run. As we haven't implemented the code to push changes up the server, when we do this, it will restore the local database to be in-sync with the set of bookmarks on the server—any local changes will be scrapped. Figure 11-14 illustrates a successful sync.

Figure 11-14. *Message box indicating the the sync operation ran*

Pushing Changes to the Server

We can now move onto looking at the code to push changes to the server. This operation essentially involves getting the bookmarks from the server, comparing it to the local set, and working out a set of changes to push.

When we built this operation for the generic web application for Android, we were lucky in that we could get the bookmarks back from the server as a synchronous method. However, on iPhone we have to retrieve the bookmarks back asynchronously through our callback, *and* we have to deal with the fact that

our callback is already being used by the getLatest operation. Recall that when we built our SBSync class, initially we added an enumeration called SBSyncMode with a single member SBSMGetLatest. We now need to add a second member—SBSMPushChanges—to this. Here's the definition:

```
// Modify enumeration in SBSync.h…
typedef enum
{
        SBSMGetLatest = 0,
        SBSMPushChanges = 1

} SBSyncMode;
```

The second thing we need to do is add two properties of type NSMutableArray. One will hold the list of bookmarks that are flagged as locally modified, and a second will hold the list of bookmarks that are flagged as locally deleted. Here are the @property definitions:

```
// Add @property definitions to SBSync.h, also add instance variables, @synthesize↩
declaration and cleanup code in dealloc…
@property (nonatomic, retain) NSMutableArray *updates;
@property (nonatomic, retain) NSMutableArray *deletes;
```

We've alluded to the fact a few times now that we need to be able to get the locally modified and locally deleted methods from the database. Here are the methods to add to SBBookmark that will issue these requests:

```
// Add method to SBBookmark.m…
// <…> Needs declaration in header
+(NSError *)getBookmarksForServerUpdate:(NSMutableArray **)theBookmarks
{
        SBSqlFilter *filter = [[SBSqlFilter alloc] initWithType:[SBBookmark class]];
        [filter.constraints addObject:[[SBSqlFieldConstraint alloc]↩
initWithFieldAndValue:[filter.entityType getField:@"localmodified"]

value:[NSNumber numberWithBool:TRUE]]];
        [filter.constraints addObject:[[SBSqlFieldConstraint alloc]↩
initWithFieldAndValue:[filter.entityType getField:@"localdeleted"]

value:[NSNumber numberWithBool:FALSE]]];
        // return...
        NSError *err = [filter executeEntityCollection:theBookmarks];
        [filter release];
        return err;
}
```

```
// <…> Needs declaration in header
+(NSError *)getBookmarksForServerDelete:(NSMutableArray **)theBookmarks
{
        SBSqlFilter *filter = [[SBSqlFilter alloc] initWithType:[SBBookmark class]];
        [filter.constraints addObject:[[SBSqlFieldConstraint alloc]➥
initWithFieldAndValue:[filter.entityType getField:@"localdeleted"]

value:[NSNumber numberWithBool:TRUE]]];
        // return...
        NSError *err = [filter executeEntityCollection:theBookmarks];
        [filter release];
        return err;
}
```

The sendChanges method is the first "meaty" method to add to SBSync. This will be called first, ahead of getLatest (as we need to update the server prior to getting the canonical set from the server). Its job will be to get the locally modified and locally deleted bookmarks and store them in the updates and deletes properties. If no changes are detected, it will defer immediately to getLatest. If changes are detected, a request to get the bookmark set from the server will be called, and we'll wait until this returns. Here's the implementation:

```
// Add method to SBSync.m…
// <…> Needs declaration in header
-(void)sendChanges
{
        // do we have anything to send?
        NSMutableArray *updates = nil;
        NSMutableArray *deletes = nil;
        NSError *err = [SBBookmark getBookmarksForServerUpdate:&updates];
        if(err != nil)
        {
                [self.callback syncFailed:err];
                return;
        }
        err = [SBBookmark getBookmarksForServerDelete:&deletes];
        if(err != nil)
        {
                [self.callback syncFailed:err];
                return;
        }

        // do we have anything to do?
        if(updates.count == 0 && deletes.count == 0)
        {
                [self getLatest];
        }
        else
        {
```

```
                // store the updates - we're going to come back async later...
                self.updates = updates;
                self.deletes = deletes;

                // ok - we have to create a delta, get the server items...
                self.mode = SBSMPushChanges;
                SBBookmarksService *service = [[SBBookmarksService alloc] init];
                [service getAll:(SBODataFetchCallback *)self];
                [service release];
        }
}
```

One more piece of housekeeping here is that we need to modify the `doSync` method to call `sendChanges` as opposed to `getLatest`. Here's the implementation:

```
// Modify doSync method in SBSync.m…
-(void)doSync:(SBSyncCallback *)theCallback
{
        self.callback = theCallback;

        // start spinning...
        [SBServiceProxy startSpinning];

        // check the database...
        SBDBHelper *db = [[SBRuntime current] getDatabase];
        [db ensureTableExists:[SBEntityType getEntityType:[SBBookmark class]]];
        [db release];

        // run...
        [self sendChanges];
}
```

We can now move on and look at detecting the actual changes and pushing them up.

Work Items

Detecting the changes is straightforward—we have done this already in our Android application. The problem for our iOS implementation is that our OData change requests will be issued asynchronously, which means we need to maintain a queue of items, progressing each one in turn until all have been sent.

To handle this, we will create a class called `SBSyncWorkItem`, which will hold a reference to an entity, the ID of the entity on the server, and the mode of operation (i.e., insert, update, or delete). We can maintain a queue of those in `SBSync` and, like the preceding example, work through the queue in turn until they have all been sent.

Before we can build `SBSyncWorkItem`, we need to create an enumeration that holds the operation type. This can be added anywhere in the project (even in a new file); however, my one will be added to the `SBODataServiceProxy.h` file. Here's the definition:

```
// Add enumeration to SBODataServiceProxy.h…
typedef enum
{
        SBODOInsert = 0,
        SBODOUpdate = 1,
        SBODODelete = 2

} SBODataOperation;
```

As I alluded to previously, `SBSyncWorkItem` is a simple class that holds a set of properties. Here's the definition:

```
// SBSyncWorkItem.h
#import <Foundation/Foundation.h>
#import "SBEntity.h"
#import "SBODataServiceProxy.h"

@interface SBSyncWorkItem : NSObject {
        SBEntity *entity;
        int serverId;
        SBODataOperation mode;
}

@property (nonatomic, retain) SBEntity *entity;
@property (assign) int serverId;
@property (assign) SBODataOperation mode;

-(id)initWithData:(SBEntity *)theEntity serverId:(int)theServerId↩
mode:(SBODataOperation)theMode;

@end
```

And here's the implementation:

```
// SBSyncWorkItem.m
#import "SBSyncWorkItem.h"
#import "MyClasses.h"

@implementation SBSyncWorkItem

        @synthesize entity;
        @synthesize serverId;
        @synthesize mode;

        -(id)initWithData:(SBEntity *)theEntity serverId:(int)theServerId↩
mode:(SBODataOperation)theMode
        {
                if(self = [super init])
```

```
                {
                        self.entity = theEntity;
                        self.serverId = theServerId;
                        self.mode = theMode;
                }

                // return...
                return self;
        }

        -(void)dealloc
        {
                [entity release];
                [super dealloc];
        }

@end
```

The next job is to build up a list of work items.

The odataFetchOk method will be called on SBSync when entities have been loaded from the server. At the moment, this is being used to service the getLatest method. The mode property allows us to vary its function depending on how it is being called. In this case, we now need to make this call a method called receiveServerItemsForPushChanges when we have the appropriate mode. Here's the code—we'll build receiveServerItemsForPushChanges in a moment.

```
// Modify method in SBSync.m…
-(void)odataFetchOk:(SBEntityXmlBucket *)entities opCode:(int)theOpCode
{
        if(theOpCode == OPCODE_ODATAFETCHALL)
        {
                // mode?
                if(self.mode == SBSMGetLatest)
                        [self receiveServerItemsForGetLatest:bucket];
                else if(self.mode == SBSMPushChanges)
                        [self receiveServerItemsForPushChanges:bucket];
                else
                        @throw [NSException exceptionWithName:[[self class] description]↩
reason:@"Mode was unhandled" userInfo:nil];
        }
        else
                @throw [NSException exceptionWithName:[[self class] description]↩
reason:@"Operation code was unhandled" userInfo:nil];

        // stop spinning...
        [SBServiceProxy stopSpinning];
}
```

We've seen the code for calculating the change delta a few times now. Let's recap the logic:

- We start by walking a list of locally modified entities, and for each we look in the server's set to find a match by ordinal.
- If we find one that matches, this is an `UPDATE`. We override the fields in the in-memory copy of the server's bookmark with the local data and queue an update. We create a work unit based on the server's bookmark.
- If we have one locally that is not referenced on the server, this is an `INSERT`. We create a work unit based on the local bookmark.
- If we have one in our deleted set that is matched by one in the server set, this is a `DELETE`. We create a work unit accordingly.

The `receiveServerItemsForPushChanges` does that logic, plus a little bit more. On `SBSync` we will hold two properties—`workItems` (an `NSMutableArray` instance) and `workItemIndex` (an integer). We'll add these now so that we don't forget.

```
// Add properties to SBSync.h…
// <…> Needs instance variable, @synthesize declaration and cleanup code in dealloc↩
 for workItems.
@property (nonatomic, retain) NSMutableArray *workItems;
@property (assign) int workItemIndex;
```

Next here's the implementation of `receiveServerItemsForPushChanges` that implements the algorithm that we just discussed:

```
// Add method to SBSync.m…
// <…> Needs declaration in header
-(void)receiveServerItemsForPushChanges:(SBEntityXmlBucket *)entities
{
        SBEntityType *et = [SBEntityType getEntityType:[SBBookmark class]];

        // good - we got this far... build a set of work to do...
        NSMutableArray *theWorkItems = [NSMutableArray array];
        for(SBBookmark *local in self.updates)
        {
                // find it in our set...
                SBBookmark *toUpdate = nil;
                for(SBBookmark *server in entities.entities)
                {
                        if(local.ordinal == server.ordinal)
                        {
                                toUpdate = server;
                                break;
                        }
                }
```

```
                // did we have one to change?
                if(toUpdate != nil)
                {
                        // walk the fields...
                        int serverId = 0;
                        for(SBEntityField *field in et.fields)
                        {
                                if(!(field.isKey))
                                        [toUpdate setValue:field value:[local↩
getValue:field] reason:SBESRUserSet];
                                else
                                        serverId = toUpdate.bookmarkId;
                        }

                        // send that up...
                        [theWorkItems addObject:[[SBSyncWorkItem alloc]↩
initWithData:toUpdate serverId:serverId mode:SBODOUpdate]];
                }
                else
                {
                        // we need to insert it...
                        [theWorkItems addObject:[[SBSyncWorkItem alloc] initWithData:local↩
serverId:0 mode:SBODOInsert]];
                }
        }

        // what about ones to delete?
        for(SBBookmark *local in self.deletes)
        {
                // find a matching ordinal on the server...
                for(SBBookmark *server in entities.entities)
                {
                        if(local.ordinal ==  server.ordinal)
                        {
                                int serverId = server.bookmarkId;
                                [theWorkItems addObject:[[SBSyncWorkItem alloc]↩
initWithData:server serverId:serverId mode:SBODODelete]];
                        }
                }
        }

        // now we have a list of work, we have to process it... asynchronously...
        self.workItems = theWorkItems;
        self.workItemIndex = 0;
        [self processWorkItems];
}
```

At the bottom of the method, we set the `workItems` property and also set the `workItemIndex` to be `0`. `workItemIndex` will act as a pointer into the queue. Once `workItemIndex` reaches the length of the `workItems` array, we know that we are finished.

So that we know we're on the right track, I'm going to propose that we display a message box that shows the number of discovered work items. Here's the code:

```
// Add to SBSync.m
// <…> Needs declaration in header
-(void)processWorkItems
{
        [SBMessageBox show:[NSString stringWithFormat:@"We have %d work item(s) to do.",↩
 self.workItems.count]];
}
```

If you run the project, go into the configuration screen and change the bookmarks, and then run a manual sync, you will see something like Figure 11-15.

Figure 11-15. *The sync operation reporting that we have two items to sync*

Now that we know our work items are being populated, we can turn our attention to actually issuing the OData requests to the server.

Issuing OData Change Requests

When we needed to create XML documents on Android, we used the XML Pull Library and its `XmlSerializer` class to manually create the document. You may also recall that my preferred method for building XML documents is to use a DOM approach, such as the `XmlDocument` class in .NET. However, just as Android does not give us a DOM-style API for creating documents, neither does iOS. Instead we have to use the **Libxml2** library, which is an open-source library that is part of Gnome. It essentially works in the same way as the `XmlSerializer` on Android, apart from the fact that the API is a flat, C-style API (like SQLite) rather than an object-orientated API.

To access Libxml2 in our project, we need to add a reference to it. Follow the steps that you undertook to add the SQLite library to the project, but this time add `libxml2.dylib`. Figure 11-16 illustrates.

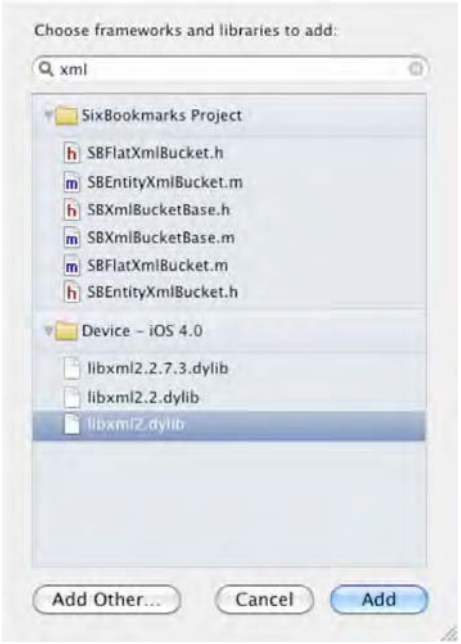

Figure 11-16. *Adding the* `libxml2.dylib` *library*

The other thing you need to do when adding the library is change the search path of the project to include the header files for Libxml2. Specifically, these are `libxml/encoding.h` and `libxml/xmlwriter.h`.

To do this, bring up the project settings and select the **SixBookmarks** *project* (not the target). At the top of the window, select **Build Settings**. This will offer you a *lot* of options. Into the little search box, enter **search**—this will limit the options to those that relate to headers and will show you an option called **Header Search Paths**.

To this **Header Search Paths** value, add a reference to this path:

```
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS4.0.sdk/usr/include/libxml2
```

Figure 11-17 illustrates.

Figure 11-17. *Project settings showing the "Header Search Paths" option*

■ **NOTE** Your mileage may vary on that path if you are using a later version of the SDK. One option is to use Spotlight to find the file on disk and add a reference to that. Check the support forums at `http://forums.multimobiledevelopment.com/` for updated information if you are stuck.

Once you've added the path, accept the various dialogs we have opened and return to the project.

Flagging Fields As "Not on Server"

Recall that when we built the sync operation in Android, we had to indicate that certain fields were not available on the server, otherwise the OData operation would crash. We need to repeat this now, and it's easily enough done—in fact, we added an `isOnServer` property when we built `SBEntityField` in the last chapter. Thus all that remains is when we declare the entity type in `SBRuntime`, we need to set the `isOnServer` property to `FALSE` for the `localModified` and `localDeleted` fields. Here's the implementation:

```
// Modify start method on SBRuntime.m…
+(void)start
{
        // create...
        SBRuntime *rt = [[SBRuntime alloc] init];
        [SBRuntime setCurrent:rt];

        // type...
        SBEntityType *et = [[SBEntityType alloc] initWithInstanceTypeNativeName:↩
[SBBookmark class] nativeName:@"Bookmark"];
        [[et addField:BOOKMARK_IDKEY nativeName:BOOKMARK_IDKEY type:SBDT_INT32 size:-1]↩
setIsKey:TRUE];
        [et addField:BOOKMARK_ORDINALKEY nativeName:BOOKMARK_ORDINALKEY type:SBDT_INT32↩
size:-1];
```

```
        [et addField:BOOKMARK_NAMEKEY nativeName:BOOKMARK_NAMEKEY type:SBDT_STRING↩
size:128];
        [et addField:BOOKMARK_URLKEY nativeName:BOOKMARK_URLKEY type:SBDT_STRING size:256];
        [[et addField:BOOKMARK_LOCALMODIFIEDKEY nativeName:BOOKMARK_LOCALMODIFIEDKEY↩
type:SBDT_INT32 size:-1] setIsOnServer:FALSE];
        [[et addField:BOOKMARK_LOCALDELETEDKEY nativeName:BOOKMARK_LOCALDELETEDKEY↩
type:SBDT_INT32 size:-1] setIsOnServer:FALSE];
        [SBEntityType registerEntityType:et];
        [et release];
}
```

Issuing Requests

I won't repeat the discussion from the Android chapter that has a quick look at the OData protocol, but I will repeat the presentation of the XML that we need to issue. This is an example of an OData XML document used to create a bookmark on the server:

```
POST /services/bookmarks.svc/Bookmark HTTP/1.1
Host: services.multimobiledevelopment.com
accept: application/atom+xml
content-type: application/atom+xml
content-encoding: UTF-8
content-length: 384
x-amx-apiusername: amxmobile
x-amx-token: 961c8c1b9d4ddd5799e7f0a7b4a5ee8b

<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
        xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
        xmlns="http://www.w3.org/2005/Atom">
    <content type="application/xml">
        <m:properties>
            <d:Name>Apress</d:Name>
            <d:Url>http://www.apress.com/</d:Url>
            <d:Ordinal>0</d:Ordinal>
        </m:properties>
    </content>
</entry>
```

We might as well jump straight in and build the methods to issue the insert, update, and delete instructions to the server.

We're going to add the calls that require Libxml2 to SBODataServiceProxy, and so we need to add the header files that we mentioned previously to this project. Add these to the top of the implementation, like so:

```
// Add header files to SBODataServiceProxy.m…
#import <libxml/encoding.h>
#import <libxml/xmlwriter.h>
```

```
#import "SBODataServiceProxy.h"
#import "MyClasses.h"

@implementation SBODataServiceProxy

// reminder of implementation ommitted for brevity…
```

pushUpdate is the complicated one to do, so we'll look at this first. pushUpdate is going to do a lot for us—it's going to build the XML document and then call a method called executeODataOperation, passing in the URL and the XML. (We'll build the executeODataOperation method and its companion getEntityUrlForPush method in a moment.) Here is the implementation of pushUpdate:

```
// Add method to SBODataServiceProxy…
// <…> Needs declaration in header
-(void)pushUpdate:(SBEntity *)entity serverId:(int)theServerId➥
callback:(SBODataFetchCallback *)theCallback
{
        // create...
        xmlBufferPtr buffer = xmlBufferCreate();
    xmlTextWriterPtr writer = xmlNewTextWriterMemory(buffer, 0);

        // start the document...
        xmlTextWriterStartDocument(writer, "1.0", "UTF-8", NULL);

        // bring forward...
        const char *atomUri = [[SBEntityXmlBucket atomNamespace]➥
cStringUsingEncoding:NSUTF8StringEncoding];
        const char *atomPrefix = nil;
        const char *metadataUri = [[SBEntityXmlBucket msMetadataNamespace]➥
cStringUsingEncoding:NSUTF8StringEncoding];
        const char *metadataPrefix = "m";
        const char *dataUri = [[SBEntityXmlBucket msDataNamespace]➥
cStringUsingEncoding:NSUTF8StringEncoding];
        const char *dataPrefix = "d";

        // start entry and content and properties...
        xmlTextWriterStartElementNS(writer, BAD_CAST atomPrefix, BAD_CAST "entry",➥
BAD_CAST atomUri);
        xmlTextWriterStartElementNS(writer, BAD_CAST atomPrefix, BAD_CAST "content",➥
BAD_CAST atomUri);
        xmlTextWriterWriteAttribute(writer, BAD_CAST "type", BAD_CAST "application/xml");
        xmlTextWriterStartElementNS(writer, BAD_CAST metadataPrefix, BAD_CAST➥
"properties", BAD_CAST metadataUri);

        // fields...
        SBEntityType *et = entity.entityType;
        for(SBEntityField *field in et.fields)
        {
```

```
                if(!(field.isKey) && field.isOnServer)
                {
                        xmlTextWriterStartElementNS(writer, BAD_CAST dataPrefix,↩
BAD_CAST [field.nativeName cStringUsingEncoding:NSUTF8StringEncoding], BAD_CAST dataUri);
                        NSObject *value = [entity getValue:field];
                        if(field.type == SBDT_STRING)
                                xmlTextWriterWriteString(writer, BAD_CAST↩
[(NSString *)value cStringUsingEncoding:NSUTF8StringEncoding]);
                        else if(field.type == SBDT_INT32)
                                xmlTextWriterWriteString(writer, BAD_CAST↩
[[NSString stringWithFormat:@"%d", [(NSNumber *)value intValue]]↩
cStringUsingEncoding:NSUTF8StringEncoding]);
                        else
                                @throw [NSException exceptionWithName:[[self class]↩
description] reason:@"Unhandled data type." userInfo:nil];
                        xmlTextWriterEndElement(writer);
                }
        }

        // end content and entry...
        xmlTextWriterEndElement(writer);
        xmlTextWriterEndElement(writer);
        xmlTextWriterEndElement(writer);

        // end the document...
        xmlTextWriterEndDocument(writer);

        // get the data out...
    xmlFreeTextWriter(writer);
    NSData *xmlData = [NSData dataWithBytes:(buffer->content) length:(buffer->use)];
    xmlBufferFree(buffer);
        NSString *xml = [[NSString alloc] initWithData:xmlData↩
encoding:NSUTF8StringEncoding];

        // dump the data...
        NSLog(@"%@", xml);

        // now we can send it...
        NSString *url = nil;
        SBODataOperation opType;
        if(theServerId == 0)
        {
                url = [self getServiceUrl:et];
                opType = SBODOInsert;
        }
```

```
        else
        {
                url = [self getEntityUrlForPush:entity serverId:theServerId];
                opType = SBODOUpdate;
        }

        // call...
        [self executeODataOperation:opType url:url xml:xml callback:theCallback];
}
```

Although that method is quite long, I personally think that it is reasonably straightforward. A lot of the code relates to marshalling data from the "managed" Objective-C world to the flat, C-style API of Libxml2. When we call the method, we pass in an `SBODataFetchCallback` instance. We've already used this to retrieve items—we're going to reuse it with a different op code. Ultimately within `SBSync`, when our `odataFetchOk` method is called, we'll look at this op code and know that the callback relates to an OData change operation.

However, it's worth mentioning the `BAD_CAST` calls. This is a macro defined in the Libxml2 headers that is needed to cast strings. I'm not entirely sure why it's called a "bad cast"—seems to me that anything that makes it work is a "good cast"!

When we issue an OData change request to the server, the URL will refer either to a specific entity (for updates and deletes) or to the class (for inserts). We already have a method called `getServiceUrl`, which returns back a URL to the class based on an entity type. We can now extend this to return a URL for a specific entity. Here's the implementation:

```
// Add method to SBODataServiceProxy.m…
// <…> Needs declaration in header
-(NSString *)getEntityUrlForPush:(SBEntity *)theEntity serverId:(int)theServerId
{
        return [NSString stringWithFormat:@"%@(%d)", [self↩
getServiceUrl:theEntity.entityType], theServerId];
}
```

Before we look at the `executeODataOperation` implementation, we'll add `pushInsert` and `pushDelete`, as these are easy. `pushInsert` defers to pushUpdate, but uses a different URL. `pushDelete` just defers to `executeODataOperation`. Here they are:

```
// Add methods to SBODataServiceProxy.m…
// <…> Needs declaration in header
-(void)pushDelete:(SBEntity *)entity serverId:(int)theServerId↩
callback:(SBODataFetchCallback *)theCallback
{
        // get...
        NSString *url = [self getEntityUrlForPush:entity serverId:theServerId];
        [self executeODataOperation:SBODODelete url:url xml:nil callback:theCallback];
}
```

```
// <…> Needs declaration in header
-(void)pushInsert:(SBEntity *)entity callback:(SBODataFetchCallback *)theCallback
{
        // an insert is an update but with a different url...
        [self pushUpdate:entity serverId:0 callback:theCallback];
}
```

Implementing executeODataOperation

In the Android application, when we looked at the equivalent method, we learned that each of the three options requires different HTTP methods. Specifically, we learned that insert requires a `POST`, update requires a `MERGE`, and delete requires a `DELETE`. We already know that all of the HTTP requests to the service require our special `x-amx-apiusername` and `x-amx-token` headers provided through our `getDownloadSettings` method. Hence all we have to do is package up an appropriate request and send it.

The only wrinkle in this process is that the request will run in an asynchronous fashion, and hence we have to handle the callback. The callback gets passed into the `pushInsert`, `pushUpdate`, and `pushDelete` methods—we pass this all the way through to `executeODataOperation` and store it in the `callback` property that we built before. However, when the request is done, we want to capture the result and then decide whether we want to call the method that indicates success or the method that indicates failure on callback. Hence the callback that we pass into the `SBDownloadBucket` that we prepare to hold the result of the HTTP download is a reference to `self`.

Before we build `executeODataOperation`, we need to change the `SBOpCodes` enumeration to include `OPCODE_ODATACHANGE`. Here's the revised definition:

```
// Modify SBOpCodes in SBServiceProxy.h…
typedef enum
{
        OPCODE_APILOGON = 1000,
        OPCODE_USERSLOGON = 2000,
        OPCODE_ODATAFETCHALL = 3000,
        OPCODE_ODATACHANGE = 3001

} SBOpCodes;
```

Here's the implementation of `executeODataOperation`:

```
// Add method to SBODataServiceProxy.m
// <…> Needs declaration in header
-(void)executeODataOperation:(SBODataOperation)opType url:(NSString*)theUrl xml:↩
(NSString *)theXml callback:(SBODataFetchCallback *)theCallback
{
        // store the callback...
        self.callback = theCallback;

        // create a request...
        NSMutableURLRequest * request = [NSMutableURLRequest requestWithURL:↩
[NSURL URLWithString:theUrl]];
        [request setValue:@"application/atom+xml" forHTTPHeaderField:@"Content-type"];
```

```
        // what method?
        if(opType == SBODOInsert)
                [request setHTTPMethod:@"POST"];
        else if(opType == SBODOUpdate)
                [request setHTTPMethod:@"MERGE"];
        else if(opType == SBODODelete)
                [request setHTTPMethod:@"DELETE"];
        else
                @throw [NSException exceptionWithName:[[self class]➥
 description] reason:@"Unhandled operation type." userInfo:nil];

        // get the settings...
        SBDownloadSettings *settings = [self getDownloadSettings];
        for(NSString *name in settings.extraHeaders)
        {
                NSString * value = [settings.extraHeaders objectForKey:name];
                [request addValue:value forHTTPHeaderField:name];
        }
        [settings release];

        // set the body...
        if(theXml != nil && theXml.length > 0)
                [request setHTTPBody:[theXml dataUsingEncoding:NSUTF8StringEncoding]];

        // create the connection with the request and start loading the data
        SBDownloadBucket *bucket = [[SBDownloadBucket alloc]➥
initWithCallback:(SBDownloadCallback *)self opCode:OPCODE_ODATACHANGE];
        NSURLConnection *connection = [[NSURLConnection alloc] initWithRequest:➥
request delegate:bucket];
        if(connection != nil)
                NSLog(@"Connection started...");
}
```

Once the HTTP operation has completed, we'll receive notification into our `downloadComplete` method. This is currently coded up to understand `OPCODE_ODATAFETCHALL`. We need to modify this so that it additionally understands `OPCODE_ODATACHANGE` and defer to a new method called `handleODataChangeComplete`. Here is the revised implementation of `downloadComplete`:

```
// Modify downloadComplete method in SBODataServiceProxy…
-(void)downloadComplete:(SBDownloadBucket *)bucket
{
        // did we fail - i.e. not 200, not 204 (OK, but nothing to say) and not 201
("created")...
        if(bucket.statusCode != 200 && bucket.statusCode != 204 && bucket.statusCode != 201)
        {
                // create an error...
                NSString *message = [NSString stringWithFormat:@"An OData request➥
returned HTTP status '%d'.", bucket.statusCode];
```

```
                NSString *html = bucket.dataAsString;
                NSLog(@"%@ --> %@", message, html);
                NSError *err = [SBErrorHelper error:self message:message];

                // flip back...
                [self.callback odataFetchFailed:err opCode:bucket.opCode];
        }
        else
        {
                // ok...
                if(bucket.opCode == OPCODE_ODATAFETCHALL)
                        [self handleFetchAllComplete:bucket];
                else if(bucket.opCode == OPCODE_ODATACHANGE)
                        [self handleODataChangeComplete:bucket];
                else
                        @throw [NSException exceptionWithName:[[self class]↩
description] reason:[NSString stringWithFormat:@"An op code of '%d' was not↩
recognised.", [bucket opCode]] userInfo:nil];
        }
}
```

As previously, we haven't implemented `handleODataChangeComplete`, but it's very easy—it simply defers to our callback. Here's the implementation:

```
// Add method to SBODataServiceProxy.m…
// <…> Needs declaration in header
-(void)handleODataChangeComplete:(SBDownloadBucket *)bucket
{
        // good, tell the callback that we did it...
        [self.callback odataFetchOk:nil opCode:bucket.opCode];
}
```

We're now at a point where once we kick off an OData request, we can send it to the server, receive a callback into the proxy and then hand that callback back to the originator of the request. The next stage is to modify the `processWorkItems` method that we stubbed earlier so that it does something meaningful.

Modifying processWorkItems

Recall that we have an array of work items stored in our `workItems` property and that `processWorkItems` is called as soon as the queue has been initialized. The first operation that `processWorkItems` will undertake will be to look at the current `workItemIndex` in relation to the list of work items. If `workItemIndex` is the same as the length of `workItems`, there is no more work to be done and `getLatest` can be called. If there is work to be done, the bookmark referenced from `workItemIndex` can be handed over the service proxy. Here's the implementation:

```
// Replace method in SBSync.m…
-(void)processWorkItems
{
```

```
        // are we at the end of the list?  if so... time to get latest...
        if(self.workItemIndex == [self.workItems count])
        {
                // return...
                [self getLatest];
                return;
        }

        // get the work item...
        SBSyncWorkItem *item = [self.workItems objectAtIndex:self.workItemIndex];
        NSLog(@"Syncing: %d, %d, %@", item.mode, item.serverId, [[item.entity class]↩
description]);

        // call the service...
        SBBookmarksService *service = [[SBBookmarksService alloc] init];
        if(item.mode == SBODOInsert)
                [service pushInsert:item.entity callback:(SBODataFetchCallback *)self];
        else if(item.mode == SBODOUpdate)
                [service pushUpdate:item.entity serverId:item.serverId↩
callback:(SBODataFetchCallback *)self];
        else if(item.mode == SBODODelete)
                [service pushDelete:item.entity serverId:item.serverId↩
callback:(SBODataFetchCallback *)self];

        // we now need to wait for something to happen...
}
```

When an OData change operation succeeds, the `odataFetchOk` method will be called on `SBSync`. We need to do something similar now in the callback depending on the op code as we did in `SBODataServiceProxy`—specifically, if we receive an `OPCODE_ODATACHANGE` notification, we need to increment the `workItemIndex` and call `processWorkItems`. The logic of `processWorkItems` dictates whether to run `getLatest` or do the next in the queue. Here's the revised implementation of `odataFetchOk`:

```
// Modify odataFetchOk method in SBSync.m…
-(void)odataFetchOk:(SBEntityXmlBucket *)entities opCode:(int)theOpCode
{
        if(theOpCode == OPCODE_ODATAFETCHALL)
        {
                // mode?
                if(self.mode == SBSMPushChanges)
                        [self receiveServerItemsForPushChanges:entities];
                else if(self.mode == SBSMGetLatest)
                        [self receiveServerItemsForGetLatest:entities];
                else
                        @throw [NSException exceptionWithName:[[self class]↩
description] reason:@"Mode was unhandled" userInfo:nil];
        }
```

```
        else if(theOpCode == OPCODE_ODATACHANGE)
        {
                // fire the next one in the queue...
                self.workItemIndex++;
                [self processWorkItems];
        }
        else
                @throw [NSException exceptionWithName:[[self class] description]↩
 reason:@"Operation code was unhandled" userInfo:nil];
}
```

We're now at a point where we can run the application and successfully synchronize some changes up to the server. Figure 11-18 illustrates a change to the text of my **Topaz Filer** bookmark example being propagated all the way up to the server and back down again into the generic web application, the source code to which can be downloaded from the web site.

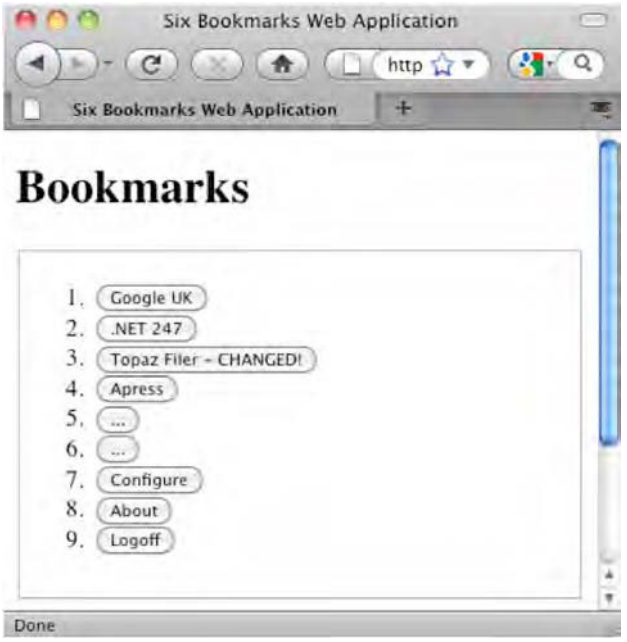

Figure 11-18: *The modified data represented on the Web application.*

Conclusion

In this chapter, we have completed the work to make our Six Bookmarks application fully functional. We can now synchronize our bookmarks with the server and present a user interface for modification.