Certainly, there are simpler ways to update your App Widget. For example, the App Widget could use its one service to do all the work of downloading the threat level data and updating the App Widget content, but then the application is left to do its own thing. The method described here illustrates how you can bypass some of the update frequency limitations of App Widgets and still share content between App Widgets and their underlying application.

Tip

Updating the `RemoteViews` object need not happen from within the App Widget provider. It can be called directly from the application, too. In this example, the service created for downloading the threat level data is used by the application and App Widget alike. Using a service for downloading online data is a good practice for a number of reasons. However, if there was no download service to leverage, we could have gotten away with just one service. In this service, fully controlled by the App Widget, we would have not only done the download but also then updated the `RemoteViews` object directly. Doing this would have eliminated the need for listening to the shared preferences changes from the App Widget service, too.

Configuring the Android Manifest File for App Widgets

In order for the Android system to know about your application's App Widget, you must include a `<receiver>` tag in the application's Android manifest file to register it as an App Widget provider. App Widgets often use services, and these services must be registered within the Android manifest file with a `<service>` tag like any other service. Here is an excerpt of the Android manifest file from the `SimpleAppWidget` project:

```
<receiver android:name="SimpleAppWidgetProvider"
    android:label="@string/widget_desc"
    android:icon="@drawable/threat_levels_descriptions">
    <intent-filter>
        <action android:name=
            "android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/simple_widget_info" />
</receiver>
<service android:name="SimpleDataUpdateService" />
<service android:name="SimpleAppWidgetProvider$PrefListenerService" />
```

Notice that, unlike a typical `<receiver>` definition, a `<meta-data>` section references an XML file resource. The `<receiver>` tag includes several bits of information about the App Widget configuration, including a label and icon for the App Widget, which is displayed on the App Widget picker (where the user chooses from available App Widgets on the system). The `<receiver>` tag also includes an intent filter to handle the `android.appwidget.action.APPWIDGET_UPDATE` intent action, as well as a `<meta-data>` tag that references the App Widget configuration file stored in the XML resource directory. Finally, the services used to update the App Widget are registered.

Installing an App Widget

After your application has implemented App Widget functionality, a user (who has installed your application) can install it to the Home screen using the following steps:

1. Long-press on the Home Screen.
2. From the menu, choose the Widgets option, as shown in Figure 22.1 (left).

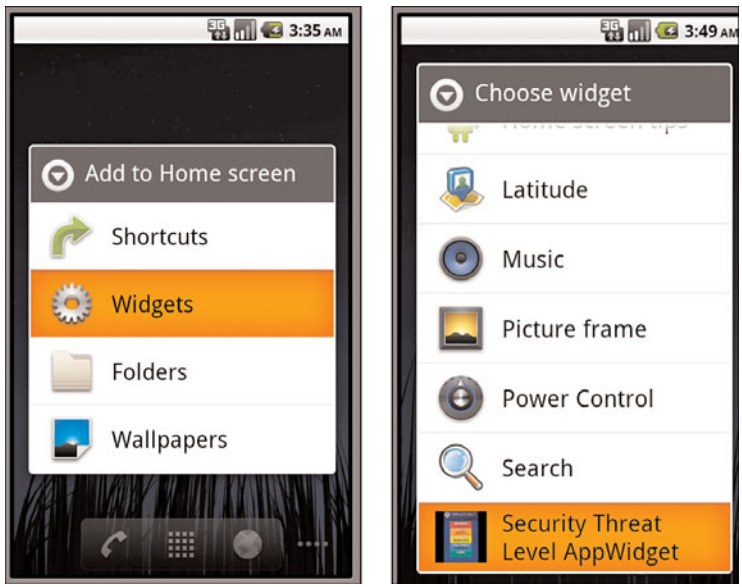

Figure 22.1 Using the Widget picker to install an App Widget on the Home screen.

3. From the Widget menu, choose the App Widget you want to include, as shown in Figure 22.1 (right).
4. Provided there is room for it, the App Widget is placed on the screen, as shown in Figure 22.2. You can move the App Widget around on the screen or remove it by dragging it onto the trash icon at the bottom of the Home screen.

Becoming an App Widget Host

Although somewhat less common, applications might also become App Widget hosts. App Widget hosts (`android.appWidget.AppWidgetHost`) are simply containers that can embed and display App Widgets. The most commonly used App Widget host is the Home screen. For more information on developing an App Widget host, see the Android SDK documentation.

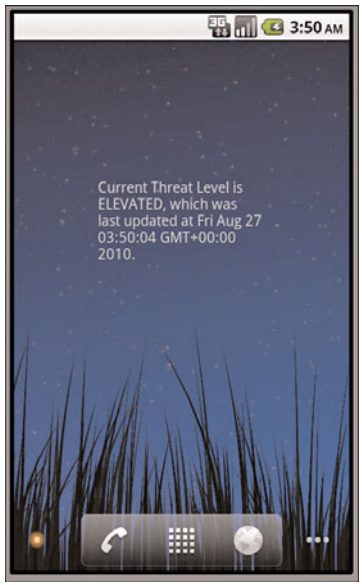

Figure 22.2 A Simple App Widget on the Home screen that displays the security threat level.

Working with Live Wallpapers

In addition to still image wallpapers, Android supports the notion of a live wallpaper. Instead of displaying a static background image on the Home screen, the user can set an interactive, or live, wallpaper that can display anything that can be drawn on a surface, such as graphics and animations. Live wallpapers were introduced in Android 2.1 (API Level 7).

Your applications can provide live wallpapers that use 3D graphics and animations as well as display interesting application content. Some examples of live wallpapers include

- A 3D display showing an animated scene portraying abstract shapes
- A service that animates between images found on an online image-sharing service
- An interactive pond with water that ripples with touch
- Wallpapers that change based on the actual season, weather, and time of day

Tip

Programmatic installation of still image wallpapers is discussed in Chapter 15, "Using Android Multimedia APIs."

Creating a Live Wallpaper

A live wallpaper is similar to an Android Service, but its result is a surface that the host can display. You need to make the following changes to your application in order to support live wallpapers:

- Provide an XML wallpaper configuration.
- Provide a `WallpaperService` implementation.
- Update the application Android manifest file to register the wallpaper service with the appropriate permissions.

Now let's look at some of these requirements in greater detail.

Tip

The code examples provided in this section are taken from the SimpleLiveWallpaper application. The source code for this application is provided for download on the book website.

Creating a Live Wallpaper Service

The guts of the live wallpaper functionality are provided as part of a `WallpaperService` implementation, and most of the live wallpaper functionality is driven by its `WallpaperService.Engine` implementation.

Implementing a Wallpaper Service

Your application needs to extend the `WallpaperService` class. The most important method the class needs to override is the `onCreateEngine()` method. Here is a sample implementation of a wallpaper service called `SimpleDroidWallpaper`:

```
public class SimpleDroidWallpaper extends WallpaperService {
    private final Handler handler = new Handler();

    @Override
    public Engine onCreateEngine() {
        return new SimpleWallpaperEngine();
    }
    class SimpleWallpaperEngine extends WallpaperService.Engine {
        // Your implementation of a wallpaper service engine here...
    }
}
```

There's not much to this wallpaper service. The `onCreateEngine()` method simply returns your application's custom wallpaper engine, which provides all the functionality for a specific live wallpaper. You could also override the other wallpaper service methods, as necessary. A `Handler` object is initialized for posting wallpaper draw operations.

Implementing a Wallpaper Service Engine

Now let's take a closer look at the wallpaper service engine implementation. The wallpaper service engine handles all the details regarding the lifecycle of a specific instance of a live wallpaper. Much like the graphics examples used in Chapter 17, "Using Android 3D

Graphics with OpenGL ES," live wallpaper implementations use a `Surface` object to draw to the screen.

There are a number of callback methods of interest within the wallpaper service engine:

- You can override the `onCreate()` and `onDestroy()` methods to set up and tear down the live wallpaper. The `Surface` object is not valid during these parts of the lifecycle.
- You can override the `onSurfaceCreated()` and `onSurfaceDestroyed()` methods (convenience methods for the `Surface` setup and teardown) to set up and tear down the `Surface` used for live wallpaper drawing.
- You should override the `onVisibilityChanged()` method to handle live wallpaper visibility. When invisible, a live wallpaper must not remain running. This method should be treated much like an `Activity` pause or resume event.
- The `onSurfaceChanged()` method is another convenience method for `Surface` management.
- You can override the `onOffsetsChanged()` method to enable the live wallpaper to react when the user swipes between Home screens.
- You can override the `onTouchEvent()` method to handle touch events. The incoming parameter is a `MotionEvent` object—we talk about the `MotionEvent` class in detail in the gestures section of Chapter 24, "Handling Advanced User Input." You also need to enable touch events (off by default) for the live wallpaper using the `setTouchEventsEnabled()` method.

The implementation details of the live wallpaper are up to the developer. Often, a live wallpaper implementation uses OpenGL ES calls to draw to the screen. For example, the sample live wallpaper project included with this book includes a live wallpaper service that creates a Bitmap graphic of a droid, which floats around the screen, bouncing off the edges of the wallpaper boundaries. It also responds to touch events by changing its drift direction. Its wallpaper engine uses a thread to manage drawing operations, posting them back to the system using the Handler object defined in the wallpaper service.

Tip

Your live wallpaper can respond to user events, such as touch events. It can also listen for events where the user drops items on the screen. For more information, see the documentation for the `WallpaperService.Engine` class.

Note

Unfortunately, the wallpaper engine implementation of the sample application, SimpleLiveWallpaper, is far too lengthy for print due to all the OpenGL ES drawing code. However, you can see its implementation as part of the sample code provided for download. Specifically, check the `SimpleDroidWallpaper` class.

> **Warning**
>
> You should take into account handset responsiveness and battery life when designing live wallpapers.

Creating a Live Wallpaper Configuration

Next, your application must provide an XML wallpaper definition. You can store this definition within the project's resources in the `/res/xml` directory. For example, here is a simple wallpaper definition called `/res/xml/droid_wallpaper.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<wallpaper xmlns:android="http://schemas.android.com/apk/res/android"
    android:thumbnail="@drawable/live_wallpaper_android"
    android:description="@string/wallpaper_desc" />
```

This simple wallpaper definition is encapsulated within the `<wallpaper>` XML tag. The `description` and `thumbnail` attributes are displayed on the wallpaper picker, where the user is prompted to select a specific wallpaper to use.

Configuring the Android Manifest File for Live Wallpapers

Finally, you need to update the application's Android manifest file to expose the live wallpaper service. Specifically, the `WallpaperService` needs to be registered using the `<service>` tag. The `<service>` tag must include several important bits of information:

- The `WallpaperService` class
- The `BIND_WALLPAPER` permission
- An intent filter for the `WallpaperService` action
- Wallpaper metadata to reference the live wallpaper configuration

Let's look at an example. Here is the `<service>` tag implementation for a simple live wallpaper:

```
<service
    android:label="@string/wallpaper_name"
    android:name="SimpleDroidWallpaper"
    android:permission="android.permission.BIND_WALLPAPER">
    <intent-filter>
        <action
            android:name="android.service.wallpaper.WallpaperService" />
    </intent-filter>
    <meta-data
        android:name="android.service.wallpaper"
        android:resource="@xml/droid_wallpaper" />
</service>
```

In addition to the service definition, you also need to limit installation of your application to API Level 7 and higher (where support for live wallpapers exists) using the `<uses-sdk>` manifest tag:

```
<uses-sdk android:minSdkVersion="7" android:targetSdkVersion="8" />
```

Keep in mind that your live wallpaper might use APIs (such as OpenGL ES 2.0 APIs) that require a higher `minSdkVersion` than API Level 7. You might also want to use the `<uses-feature>` tag to specify that your application includes live folder support, for use within Android Market filters:

```
<uses-feature android:name="android.software.live_wallpaper" />
```

Installing a Live Wallpaper

After you've implemented live wallpaper support within your application, you can set a live wallpaper on your Home screen using the following steps:

1. Long-press on the Home Screen.
2. From the menu, choose the Wallpapers option, as shown in Figure 22.3 (left).

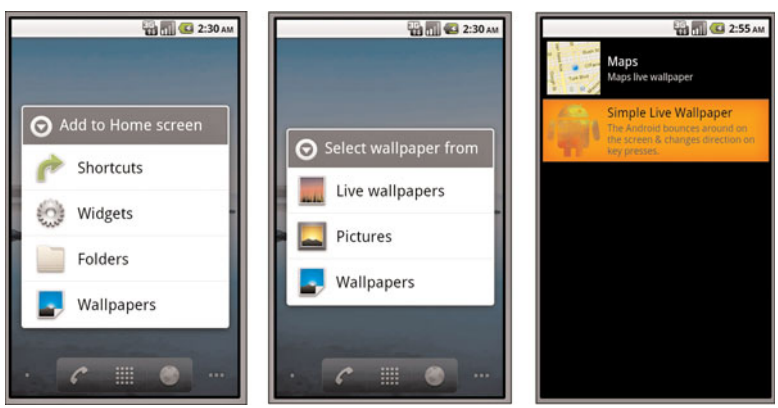

Figure 22.3 Installing a live wallpaper on the Home screen.

3. From the Wallpaper menu, choose the Live wallpapers option, as shown in Figure 22.3 (middle).
4. From the Live Wallpaper menu, choose the live wallpaper you want to include, as shown in Figure 22.3 (right).

5. After you've chosen a wallpaper, it is shown in preview mode. Simply choose the Set Wallpaper button to confirm you want to use that live wallpaper. The live wallpaper is now visible on your Home screen, as shown in Figure 22.4.

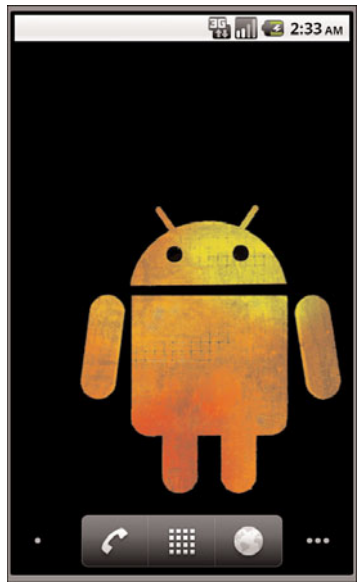

Figure 22.4 A Simple AppWidget on the Home screen that displays the security threat level.

Acting as a Content Type Handler

Your application can act as a content type filter—that is, handle common intent actions such as `VIEW`, `EDIT`, or `SEND` for specific MIME types.

Tip

See the `android.content.Intent` class for a list of standard activity actions.

A photo application might act as a content type handler for `VIEW` actions for any graphic formats, such as JPG, PNG, or RAW image file MIME types. Similarly, a social networking application might want to handle intent `SEND` actions when the underlying data has a MIME type associated with typical social content (for example, text, graphic, or video). This means that any time the user tries to send data (with the MIME types that the social networking application was interested in) from an Android application using an Intent with action `SEND`, the social networking application is listed as a choice for completing the `SEND` action request. If the user chooses to send the content using the social networking application, that application has to launch an Activity to handle the request (for example, an Activity that uploads the content to the social networking website to share).

Finally, content type handlers make it easier to extend the application to act as a content provider, provide search capabilities, or include live folder features. Define data records using custom MIME types, so that no matter how an Intent fires (inside or outside the application), the action is handled by the application in a graceful fashion.

To enable your application to act as a content type handler, you need to make several changes to your application:

- Determine which Intent actions and MIME types your application needs to be able to handle.
- You need to implement an `Activity` that can process the Intent action or actions that you want to handle.
- You need to register that `Activity` in your application's Android Manifest file using the `<activity>` tag as you normally would. You then need to configure an `<intent-filter>` tag for that Activity within your application's Android Manifest file, providing the appropriate intent action and MIME types your application can process.

Determining Intent Actions and MIME Types

Let's look at a simple example. For the remainder of this chapter, we make various modifications to a simple field notes application that uses a content provider to expose African game animal field notes; each note has a title and text body (the content itself comes from field notes on African game animals that we wrote up years ago on our nature blog, which is very popular with grade-schoolers). Throughout these examples, the application acts as a content type handler for `VIEW` requests for data with a custom MIME type:

```
vnd.android.cursor.item/vnd.androidbook.live.fieldnotes
```

Tip

MIME types come in two forms. Most developers are familiar with MIME types, such as `text/plain` or `image/jpeg` (as defined in RFC2045 & RFC2046), which are standards used globally. The Internet Assigned Numbers Authority (IANA, at http://www.iana.org) manages these global MIME types.

Developers frequently need to create their own MIME types, but without the need for them to become global standards. These types must still be sufficiently unique that MIME type namespace collisions do not occur. When you're dealing with Android content providers, there are two well-defined prefixes that you can use for creating MIME types. The `ContentResolver.CURSOR_DIR_BASE_TYPE` prefix ("vnd.android.cursor.dir") is for use with directories or folders of items. The `ContentResolver.CURSOR_ITEM_BASE_TYPE` prefix ("vnd.android.cursor.item") is for use with a single type. The part after the slash must then be unique. It's not uncommon to pattern MIME types after package names or other such unique qualifiers.

Implementing the Activity to Process the Intents

Next the application needs an `Activity` class to handle the Intents it receives. For the sample, we simply need to load a page capable of viewing a field note. Here is a sample implementation of an `Activity` that can parse the `Intent` data and show a screen to displays the field note for a specific animal:

```
public class SimpleViewDetailsActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.details);
        try {
            Intent launchIntent = getIntent();
            Uri launchData = launchIntent.getData();
            String id = launchData.getLastPathSegment();
            Uri dataDetails = Uri.withAppendedPath
                (SimpleFieldnotesContentProvider.CONTENT_URI, id);
            Cursor cursor =
                managedQuery(dataDetails, null, null, null, null);
            cursor.moveToFirst();
            String fieldnoteTitle = cursor.getString(cursor
                .getColumnIndex(SimpleFieldnotesContentProvider
                .FIELDNOTES_TITLE));
            String fieldnoteBody = cursor.getString(cursor
                .getColumnIndex(SimpleFieldnotesContentProvider
                .FIELDNOTES_BODY));
            TextView fieldnoteView = (TextView)
                findViewById(R.id.text_title);
            fieldnoteView.setText(fieldnoteTitle);
            TextView bodyView = (TextView) findViewById(R.id.text_body);
            bodyView.setLinksClickable(true);
            bodyView.setAutoLinkMask(Linkify.ALL);
            bodyView.setText(fieldnoteBody);
        } catch (Exception e) {
            Toast.makeText(this, "Failed.", Toast.LENGTH_LONG).show();
        }
    }
}
```

The `SimpleViewDetailsActivity` class retrieves the Intent that was used to launch the `Activity` using the `getIntent()` method. It then inspects the details of that intent, extracting the specific field note identifier using the `getLastPathSegment()` method. The rest of the code simply involves querying the underlying content provider for the appropriate field note record and displaying it using a layout.

Registering the Intent Filter

Finally, the Activity class must be registered in the application's Android manifest file and the intent filter must be configured so that the application only accepts Intents for specific actions and specific MIME types. For example, the `SimpleViewDetailsActivity` would be registered as follows:

```
<activity
    android:name="SimpleViewDetailsActivity">
    <intent-filter>
       <action
           android:name="android.intent.action.VIEW" />
           <category
               android:name="android.intent.category.DEFAULT" />
           <data android:mimeType =
              "vnd.android.cursor.item/vnd.androidbook.live.fieldnotes" />
    </intent-filter>
</activity>
```

The `<activity>` tag remains the same as any other. The `<intent-filter>` tag is what's interesting here. First, the action that the application wants to handle is defined using an `<action>` tag that specifies the action the application can handle is the `VIEW` action. The `<category>` tag is set to DEFAULT, which is most appropriate, and finally the `<data>` tag is used to filter `VIEW` Intents further to only those of the custom MIME type associated with field notes.

Tip

The rest of the sample applications used in this chapter (SimpleSearchIntegration and SimpleLiveFolder) act as content type handlers for field note content as described in this section. The source code for these applications is provided in full for download on the book website. However, read on for more information regarding the implementation of these applications.

Making Application Content Searchable

If your application is content rich, either with content created by users or with content provided by you, the developer, then integrating with the search capabilities of Android can provide many benefits and add value to the user. The application data becomes part of the overall handset experience, is more accessible, and your application may be presented to the user in more cases than just when they launch it.

Most Android devices share a set of common hardware buttons: Home (), Menu (), Back (), and Search (). Applications can implement powerful search features within their applications using the Android framework. There are two ways that search capabilities are generally added to Android applications:

- Applications implement a search framework that enables their activities to react to the user pressing the Search button and perform searches on data within that application.
- Applications can expose their content for use in global, system-wide searches that include application and web content.

Search framework features include the ability to search for and access application data as search results, as well as the ability to provide suggestions as the user is typing search criteria. Applications can also provide an Intent to launch when a user selects specific search suggestions.

Tip

The code examples provided in this section are taken from the SimpleSearchIntegration application. The source code for this application is provided for download on the book website.

Let's consider the African field notes application we discussed in the previous section. This application uses a simple content provider to supply information about game animals. Enabling search support within this application seems rational; it would enable the user to quickly find information about a specific animal simply by pressing the Search button. When a result is found, the application needs to be able to apply an `Intent` for launching the appropriate screen to view that specific field note—the perfect time to implement a simple content type handler that enables the application to handle "view field note" actions, as shown in Figure 22.5.

Enabling Searches Within Your Application

You need to make a number of changes within your application to enable searches. Although these changes might seem complex, the good news is that if you do it right, enabling global searches later is very simple. Searching content generally necessitates that your application acts as a content provider, or at the very least has some sort of underlying database that can be searched in a systematic fashion.

Note

The search framework provided by the `SearchManager` class (`android.app.SearchManager`) does not actually perform the search queries—that is up to you, the developer. The `SearchManager` class simply manages search services and the search dialog controls. How and what data is searched and which results are returned are implementation details.

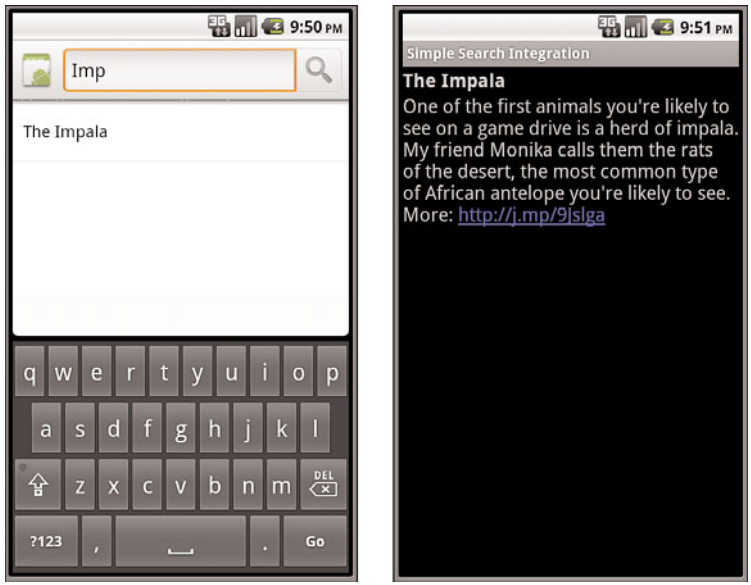

Figure 22.5 Handling in-application searches and search suggestions.

To enable in-application searches, you need to

- Develop an application with data, ideally exposed as a content provider.
- Create an XML search configuration file.
- Implement an `Activity` class to handle searches.
- Configure the application's Android manifest file for searches.

Now let's look at each of these requirements in more detail.

Creating a Search Configuration

Creating a search configuration for your application simply means that you need to create an XML file with special search tags. This search configuration file is normally stored in the xml resource directory (for example, `/res/xml/searchable.xml`) and referenced in the searchable application's Android manifest file.

Enabling Basic Searches

The following is a sample search configuration the field notes application might use, stored as an application resource file called

```
<?xml version="1.0" encoding="utf-8"?>
<searchable
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:label="@string/app_name"
    android:hint="@string/search_hint"
    android:searchSettingsDescription="@string/search_settings_help">
</searchable>
```

The basic attributes of the search configuration are fairly straightforward. The `label` field is generally set to the name of your application (the application providing the search result). The `hint` field is the text that shows in the `EditText` control of the search box when no text has been entered—a prompt. You can further customize the search dialog by customizing the search button text and input method options, if desired.

Enabling Search Suggestions

If your application acts as a content provider and you want to enable search suggestions—those results provided in a list below the search box as the user types in search criteria—then you must include several additional attributes within your search configuration. You need to specify information about the content provider used to supply the search suggestions, including its authority, path information, and the query to use to return search suggestions. You also need to provide information for the `Intent` to trigger when a user clicks on a specific suggestion.

Again, let's go back to the field notes example. Here are the search configuration attributes required in order to support search suggestions that query field note titles:

```
android:searchSuggestAuthority =
    "com.androidbook.simplesearchintegration.SimpleFieldnotesContentProvider"
android:searchSuggestPath="fieldnotes"
android:searchSuggestSelection="fieldnotes_title LIKE ?"
android:searchSuggestIntentAction="android.intent.action.VIEW"
android:searchSuggestIntentData = "content://com.androidbook.simplesearch
integration.SimpleFieldnotesContentProvider/fieldnotes"
```

The first attribute, `searchSuggestAuthority`, sets the content provider to use for the search suggestion query. The second attribute defines the path appended to the `Authority` and right before `SearchManager.SUGGEST_URI_PATH_QUERY` is appended to the `Authority`, as well. The third attribute supplies the SQL WHERE clause of the search query (here, only the field note titles, not their bodies, are queried to keep search suggestion performance reasonably fast). Next, an `Intent` action is provided for when a user clicks a search suggestion and then finally the intent, the `Uri` used to launch the `Intent`, is defined.

You can also set a threshold (`android:searchSuggestThreshold`) on the number of characters the user needs to type before a search suggestion query is performed. Consider setting this value to a reasonable number like 3 or 4 characters to keep queries to a minimum (the default is 0). At a value of zero, even an empty search field shows suggestions—but these are not filtered at all.

Each time the user begins to type in search criteria, the system performs a content provider query to retrieve suggestions. Therefore, the application's content provider interface needs to be updated to handle these queries. In order to make this all work properly,

you need to define a projection in order to map the content provider data columns to those that the search framework expects to use to fill the search suggestion list with content. For example, the following code defines a project to map the field notes unique identifiers and titles to the `_ID`, `SUGGEST_COLUMN_TEXT_1` and `SUGGEST_COLUMN_INTENT_DATA_ID` fields for the search suggestions:

```
private static final HashMap<String, String>
FIELDNOTES_SEARCH_SUGGEST_PROJECTION_MAP;
static {
    FIELDNOTES_SEARCH_SUGGEST_PROJECTION_MAP =
        new HashMap<String, String>();
    FIELDNOTES_SEARCH_SUGGEST_PROJECTION_MAP.put(_ID, _ID);
    FIELDNOTES_SEARCH_SUGGEST_PROJECTION_MAP.put(
        SearchManager.SUGGEST_COLUMN_TEXT_1, FIELDNOTES_TITLE + " AS "
        + SearchManager.SUGGEST_COLUMN_TEXT_1);
    FIELDNOTES_SEARCH_SUGGEST_PROJECTION_MAP.put(
        SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID, _ID + " AS "
        + SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID);
}
```

Each time search suggestions need to be displayed, the system executes a query using the `Uri` provided as part of the search configuration. Don't forget to define this `Uri` and register it in the content provider's `UriMatcher` object (using the `addURI()` method). For example, the field notes application used the following `Uri` for search suggestion queries:

```
content:// com.androidbook.simplesearchintegration.
SimpleFieldnotesContentProvider/fieldnotes/search_suggestion_query
```

By providing a special search suggestion `Uri` for the content provider queries, you can simply update the content provider's `query()` method to handle the specialized query, including building the projection, performing the appropriate query and returning the results. Let's take a closer look at the field notes content provider `query()` method:

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
    String[] selectionArgs, String sortOrder) {
    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
    queryBuilder.setTables(SimpleFieldnotesDatabase.FIELDNOTES_TABLE);
    int match = sURIMatcher.match(uri);
    switch (match) {
    case FIELDNOTES_SEARCH_SUGGEST:
        selectionArgs = new String[] { "%" + selectionArgs[0] + "%" };
        queryBuilder.setProjectionMap(
            FIELDNOTES_SEARCH_SUGGEST_PROJECTION_MAP);
        break;
    case FIELDNOTES:
        break;
```

```
        case FIELDNOTE_ITEM:
            String id = uri.getLastPathSegment();
            queryBuilder.appendWhere(_ID + "=" + id);
            break;
        default:
            throw new IllegalArgumentException("Invalid URI: " + uri);
        }
        SQLiteDatabase sql = database.getReadableDatabase();
        Cursor cursor =
            queryBuilder.query(sql, projection, selection,
            selectionArgs, null, null, sortOrder);
        cursor.setNotificationUri(getContext().getContentResolver(), uri);
        return cursor;
    }
```

This `query()` method implementation handles both regular content queries and special search suggestion queries (those that come in with the search suggestion `Uri`). When the search suggestion query occurs, we wrap the search criteria in wildcards and use the handy `setProjectionMap()` method of the `QueryBuilder` object to set and execute the query as normal. Because we want to return results quickly, we only search for titles matching the search criteria for suggestions, not the full text of the field notes.

> **Tip**
>
> Instead of using wildcards and a slow `LIKE` expression in SQLite, we could have used the SQLite FTS3 extension, which enables fast full-text queries. With a limited number of rows of data, this is not strictly necessary in our case and it requires creating tables in a different and much less relational way. Indices are not supported, so query performance might suffer. See the SQLite FTS3 documentation at http://www.sqlite.org/fts3.html.

Enabling Voice Search

You can also add voice search capabilities to your application. This enables the user to speak the search criteria instead of type it. There are several attributes you can add to your search configuration to enable voice searches. The most important attribute is `voiceSearchMode`, which enables voice searches and sets the appropriate mode: The `showVoiceSearchButton` value enables the little voice recording button to display as part of the search dialog, the `launchRecognizer` value tells the Android system to use voice recording activity, and the `launchWebSearch` value initiates the special voice web search activity.

To add simple voice support to the field notes sample application can be done simply by adding the following line to the search configuration:

```
android:voiceSearchMode="showVoiceSearchButton|launchRecognizer"
```

Other voice search attributes you can set include the voice language model (free form or web search), the voice language, the maximum voice results, and a text prompt for the voice recognition dialog. See the Android SDK documentation regarding Searchable

Configuration for more details: http://developer.android.com/guide/topics/search/searchable-config.html.

Creating a Search Activity

Next, you need to implement an `Activity` class that actually performs the requested searches. This `Activity` is launched whenever your application receives an intent with the action value of `ACTION_SEARCH`.

The search request contains the search string in the extra field called `SearchManager.QUERY`. The `Activity` takes this value, performs the search, and then responds with the results.

Let's look at the search `Activity` from our field notes example. You can implement its search activity, `SimpleSearchableActivity`, as follows:

```
public class SimpleSearchableActivity extends ListActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Intent intent = getIntent();
        checkIntent(intent);
    }

    @Override
    protected void onNewIntent(Intent newIntent) {
        // update the activity launch intent
        setIntent(newIntent);
        // handle it
        checkIntent(newIntent);
    }

    private void checkIntent(Intent intent) {
        String query = "";
        String intentAction = intent.getAction();
        if (Intent.ACTION_SEARCH.equals(intentAction)) {
            query = intent.getStringExtra(SearchManager.QUERY);
            Toast.makeText(this,
                "Search received: " + query, Toast.LENGTH_LONG)
                .show();
        } else if (Intent.ACTION_VIEW.equals(intentAction)) {
            // pass this off to the details view activity
            Uri details = intent.getData();
            Intent detailsIntent =
                new Intent(Intent.ACTION_VIEW, details);
            startActivity(detailsIntent);
            finish();
            return;
        }
```

```
        fillList(query);
    }

    private void fillList(String query) {
        String wildcardQuery = "%" + query + "%";
        Cursor cursor =
            managedQuery(
                SimpleFieldnotesContentProvider.CONTENT_URI,
                null,
                SimpleFieldnotesContentProvider.FIELDNOTES_TITLE
                + " LIKE ? OR "
                + SimpleFieldnotesContentProvider.FIELDNOTES_BODY
                + " LIKE ?",
                new String[] { wildcardQuery, wildcardQuery }, null);
        ListAdapter adapter =
            new SimpleCursorAdapter(
                this,
                android.R.layout.simple_list_item_1,
                cursor,
                new String[] {
                    SimpleFieldnotesContentProvider.FIELDNOTES_TITLE },
                new int[] { android.R.id.text1 });
        setListAdapter(adapter);
    }

    @Override
    protected void onListItemClick(
        ListView l, View v, int position, long id) {
        Uri details = Uri.withAppendedPath(
            SimpleFieldnotesContentProvider.CONTENT_URI, "" + id);
        Intent intent =
            new Intent(Intent.ACTION_VIEW, details);
        startActivity(intent);
    }
}
```

Both the `onCreate()` and `onNewIntent()` methods are implemented because the `Activity` is flagged with a `launchMode` set to `singleTop`. This `Activity` is capable of bringing up the search dialog when the user presses the Search button, like the rest of the activities in this example. When the user performs a search, the system launches the `SimpleSearchableActivity`—the same activity the user was already viewing. We don't want to create a huge stack of search result activities, so we don't let it have more than one instance on top of the stack—thus the `singleTop` setting.

Handling the search is fairly straightforward. We use the search term provided for us to create a query. Using the `managedQuery` call, the results are obtained as a `Cursor` object

that is then used with the `SimpleCursorAdapter` object to fill the `ListView` control of the `Activity` class.

For list item click handling, the implementation here simply creates a new `VIEW` intent and, effectively, lets the system handle the item clicking. In this case, the details activity handles the displaying of the proper field note. Why do this instead of launching the class activity directly? No reason other than it's simple and it's well tested from other uses of this launch style.

When a user clicks on a suggestion in the list, instead of an `ACTION_SEARCH`, this activity receives the usual `ACTION_VIEW`. Instead of handling it here, though, it's passed on to the details view `Activity` as that activity is already designed to handle the drawing of the details for each item—no reason to implement it twice.

Configuring the Android Manifest File for Search

Now it's time to register your searchable Activity class within the application manifest file, including configuring the intent filter associated with the `ACTION_SEARCH` action. You also need to mark your application as searchable using a `<meta-data>` manifest file tag.

Here is the Android manifest file excerpt for the searchable activity registration:

```
<activity
    android:name="SimpleSearchableActivity"
    android:launchMode="singleTop">
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
    </intent-filter>
    <meta-data
        android:name="android.app.searchable"
        android:resource="@xml/searchable" />
</activity>
```

The main difference between this `<activity>` tag configuration and a typical activity is the addition of the intent filter for intents with an action type of `SEARCH`. In addition, some metadata is provided so that the system knows where to find the search configuration details.

Next, let's look at an example of how to enable the Search button for all activities within the application. This `<meta-data>` block needs to be added to the `<application>` tag, outside any `<activity>` tags.

```
<meta-data
    android:name="android.app.default_searchable"
    android:value =
        "com.androidbook.simplesearchintegration.SimpleSearchableActivity" />
```

This `<meta-data>` tag configures the default activity that handles the search results for the entire application. This way, pressing the Search button brings up the search dialog from

any activity within the application. If you don't want this functionality in every activity, you need to add this definition to each activity for which you do want the Search button enabled.

> **Note**
>
> Not all Android devices have a Search button. If you want to guarantee search abilities within the application, consider adding other ways to initiate a search, such as adding a Search button to the application screen or providing the search option on the Option menu.

Enabling Global Search

After you have enabled your application for searches, you can make it part of the global device search features with a few extra steps. Global searches are often invoked using the Quick Search Box. In order to enable your application for global search, you need to

- Begin with an application that already has in-application search abilities as described earlier.
- Update the search configuration file to enable global searches.
- Include your application in global searches by updating the Search settings of the device.

Now let's look at these requirements in a bit more detail. Let's assume we're working with the same sample application—the field notes. Figure 22.6 shows the global search box, as initiated from the Home screen.

Updating a Search Configuration for Global Searches

Updating an existing search configuration is very simple. All you need to do is add the `includeInGlobalSearch` attribute in your configuration and set it to `true` as follows:

```
android:includeInGlobalSearch="true"
```

At this point, you should also ensure that your application is acting as a content type handler for the results you provide as part of global searches (if you haven't already). That way, users can select search suggestions provided by your application. Again, you probably want to leverage the content type handler functionality again, in order to launch the application when a search suggestion is chosen.

> **Tip**
>
> You can initiate a global search using the `SearchManager.INTENT_ACTION_GLOBAL_SEARCH` Intent.

Updating Search Settings for Global Searches

However, the user has ultimate control over what applications are included as part of the global search. Your application is not included in global searches by default. The user must include your application explicitly. In order for your application's content to show up as part of global searches, the user must adjust the device Search settings. The user makes this configuration from the Settings, Search, Searchable Items menu, as shown in Figure 22.7.

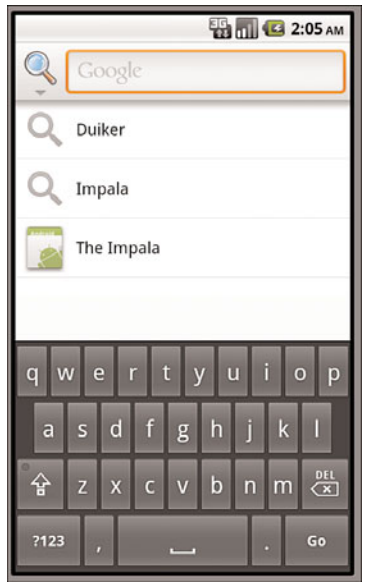

Figure 22.6 Application content is included in global search results, such as when the user presses the search button while on the Home screen.

If your application has content that is appropriate for global searches, you might want to include a shortcut to these settings so that users can easily navigate to them without feeling like they've left your application. The `SearchManager` class has an intent called `INTENT_ACTION_SEARCH_SETTINGS` for this purpose:

```
Intent intent = new Intent(SearchManager.INTENT_ACTION_SEARCH_SETTINGS);
startActivity(intent);
```

This intent launches the Settings application on the Search settings screen, as shown in Figure 22.7 (left).

As you can see, searches—whether they are in-application searches or global searches—allow application content to be exposed in new and interesting ways so that the user's data is always just a few keystrokes (or spoken words) away. But wait! There's more! Check out the Search Dev Guide on the Android developer website to learn more about the sophisticated features available as part of the Android search framework: http://developer.android.com/guide/topics/search/index.html.

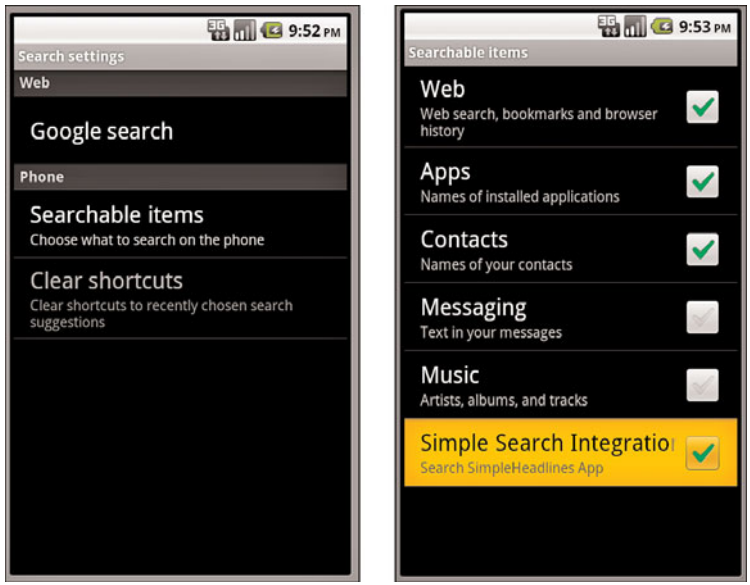

Figure 22.7 Configuring device search settings to include content from your application.

Working with Live Folders

Another way you can make content-rich applications more readily available to users is with the use of live folders. Introduced in Android 1.5 (API Level 3), a live folder is a special type of folder that the user can place in various areas such as the Home screen and, when clicked, displays its content by querying an application that acts as a content provider. Each piece of data in the folder can be paired with an intent. You could also think of it as a folder of shortcuts into your application. For example, a music application might allow the user to create live folders for favorite music. Similarly, a to-do list application might include support for a live folder of the day's tasks. Finally, a game might have a live folder for saved game points. When the user clicks on an item, the application launches to play the appropriate song, show the appropriate to-do list item, or start the game at that save point. Applications can support live folders with different types of content—it all depends on the content the application has to expose.

Let's return to the example of the African field notes application and update it so that users can create live folders with field note titles. Clicking on a specific field note launches the application with an action `VIEW` for the full field note contents (again, by acting as a content type handler, as discussed earlier in this chapter).

Tip

Many of the code examples provided in this section are taken from the SimpleLiveFolder application. The source code for this application is provided for download on the book website.