Developing Firefly Plugins

Introduction

Firefly plugins allow your app to use Firefly technology for content-aware operations. When Firefly identifies an item, users can tap the identified item's label to see actions to take for that item. The available actions for an identified item can be built in to Firefly, or you can provide additional actions through plugins. An example of a built-in action could be purchasing the identified item from Amazon, or sharing that item's information in email or social networks. For plugin actions, you could provide a VOIP plugin that allows users to dial a phone number identified by Firefly using VOIP. Or you could provide a plugin that allows users to purchase concert tickets for a Firefly-identified performer if there is an available performance within a 50-mile radius of the user's current location. All Firefly plugins rely on Firefly's identification capabilities to provide content for their operations.

This document describes how Firefly plugins work so that you can understand how to write your own.

Overview

When Firefly scans an item, it attempts to perform an identification. If Firefly identifies the item, it creates a digital representation of the item called a digital entity. A digital entity is made up of one or more facets, each of which describes an aspect of the identified item. For example, if the identified item is a book, the digital entity includes a book facet, which includes information such as the title, authors, publisher, release date, ISBN-10, and ISBN-13. If the book is sold by Amazon, the digital entity also contains a product details facet, which includes information such as ASIN, customer rating, department, and the product's Amazon.com URL. Similarly, an identified song's digital entity may contain a music facet and product details facet.

When a plugin is first installed, it registers with Firefly to tell the system what types of identifications it is interested in processing. To do this, the plugin defines a set of filters that specifies what types of facets it wants to receive. There are four types of filters available:

- The **required** filter requires that a digital entity contain all of the given types of facets for the item to be processed by the plugin. Firefly passes all the filtered facets to the plugin.
- The **any-of** filter requires that the digital entity contain at least one of the given types of facets for the item to be processed by the plugin. If more than one any-of types are present, Firefly passes all those facets to the plugin.
- The **optional** filter does not require any of the given types of facets to be present in the digital entity in order for the plugin to process the item. If any of the types are present, Firefly passes those types to the plugin.
- The **not-any filter** requires that a digital entity not contain any of the given types of facets to be present for the filter to match.

There are two types of Firefly plugins:

- **Simple** plugins do not perform business logic and always display their actions to the user based solely on the type of facets that are present in the digital entity. The sample plugin FoneVOIP

(available in the SDK) is an example of a simple plugin. The FoneVOIP plugin is invoked when Firefly identifies a phone number. The user is given the option to dial the phone number every time, with no other logic to execute.

- **Resolving** plugins perform business logic and conditionally provide additional actions to the user. Typically a resolving plugin calls on an external service to perform the business logic. The sample plugin Exempli (available in the SDK) is an example of a resolving plugin. The Exempli plugin is invoked when Firefly identifies a musical artist. That plugin calls an external service to determine whether there are any upcoming performances by that artist within a 50-mile radius of the user's current location. Because the business logic is conditional, Firefly may not present any results or actions to the user.

Resolving plugins define a resolver class, which is where the logic happens. Simple plugins do not define a resolver class.

When Firefly identifies one or more items, it creates a user interface element for each of those items. When a user taps one of these items, either immediately or from the Firefly history, a detail page opens to display the complete list of actions that may be performed on that item. Your plugin defines a UI element that includes both a label and an action to be performed. The action is typically an intent that deep-links into the plugin's app to reveal more content or perform an action.

Digital Entities and Facets

When Firefly detects an item, it attempts to perform an identification. If Firefly identifies the item, it creates a digital representation of the item called a digital entity. The `DigitalEntity` class represents digital entities in the Firefly plugin architecture.

Each `DigitalEntity` object includes one or more facets, represented by the `Facet` class. Each facet represents one aspect of the entity. Facets can be generated internally by Firefly when it identifies and creates the digital entity, or provided by a resolving plugin. Note that one plugin cannot access the facets generated by another plugin.

Each `Facet` consists of a facet type and a set of attributes. For a list of valid facet types, see `com.amazon.mw.entity.FacetType`.

A digital entity will contain at most one of any type of facet. The facet's attributes are `String` key–value pairs. Resolving plugins, through their resolver class, add a facet of the type `MY_FACET` to the `DigitalEntity` if they wish to display something in the list of actions on the detail page. Adding a facet of the type `MY_FACET` is a way to store information resulting from the resolving plugin's business logic so that it can be passed back to the plugin when the plugin creates a label for its action and handles its click event. Examples of the types of data that can be stored in `MY_FACET` include a string to display in the user interface, a unique ID number representing a database index needed in order to execute an intent when the item is clicked, or other contextual information such as a timestamp or geolocation data that is used when an item is accessed from Firefly's history and clicked at a time distant from when the initial identification was made.

Filters

Your plugin defines a set of filters that describes which facets your plugin is interested in. The `DigitalEntityFilter` class represents a filter. Firefly uses the plugin's filters to determine when the plugin should be invoked. If the facets in an identified item's digital entity match the set of filters defined in the plugin's `DigitalEntityFilter`, that plugin is invoked. Once the filter set is applied to an incoming `DigitalEntity`, only those facets that were specified by the filters are passed through to the plugin for further processing. The only exception is if a resolving plugin produces a facet, it is automatically added to the `DigitalEntity` without the `DigitalEntityFilter` needing to specify the `MY_FACET` facet type.

There are four types of filters available:

- The **required** filter defines which facet types must be present in the `DigitalEntity`. All required facet types must be present in order for processing to continue.
- The **any-of** filter defines a set of facet types of which at least one must be present. At least one of the any-of facet types must be present in order for processing to continue.
- The **optional** filter defines facet types that may be present. If an optional facet type is present, it will be included in the `DigitalEntity`. If an optional facet type is not present, processing continues as long as the required and any-of filters have been satisfied.
- The **not-any filter** requires that a digital entity not contain any of the given types of facets to be present for the filter to match.

The following `DigitalEntityFilter` methods are used to define the filter criteria:

- Use `addRequiredFacets(FacetType...)` to define required facet types.
- Use `addAnyOfFacets(FacetType...)` to define any-of facet types.
- Use `addOptionalFacets(FacetType...)` to define optional facet types.

A `DigitalEntityFilter` must include at least one required or any-of facet type.

In this example, the plugin whose filter is shown is invoked for both music and video identifications and it also receives Amazon product information if it is available:

```
1 DigitalEntityFilter filter = new DigitalEntityFilter();
2 filter.addAnyOfFacets(FacetType.MUSIC, FacetType.VIDEO);
3 filter.addOptionalFacets(FacetType.PRODUCT);
```

User Interface Elements

When Firefly identities an item, it displays a tappable label on the screen. When users tap the label, a detail page opens. The detail page contains a list of labels that represent actions the user can take on the identified item. For example, if Amazon sells the item, one of the built-in actions is to purchase that item from Amazon. Your plugins can add additional actions to the list.

To add actions to the detail list, your plugin must define a `Label` and implement an `onClick()` method that is invoked by Firefly when your label is tapped. Extend the `DigitalEntityUI` class to do both these things.

The `DigitalEntityUI` class has two abstract methods that you must implement:

- `getLabel()` returns a `Label` that includes information such as a description of the action that will take place when the label is tapped. The label should not contain time-sensitive information because

the user may not interact with your plugin immediately after an identification is made. The label will remain visible in Firefly's history and it is possible the user may interact with your plugin quite some time after the initial identification is made. `getLabel()` should return quickly, otherwise Firefly may abandon the request and issue an error. To ensure `getLabel()` returns within the time allotted, all business logic should be performed in a resolving plugin's `Resolver` class, which is given approximately seven seconds of processing time, and the result of the business logic should be stored in a `MY_FACET` facet for use by `getLabel()`.

- `onClick()` is a listener that is called when your plugin's label is tapped. The normal response is for your plugin to send an intent to a detail page within your app, although this is not strictly necessary and you may send any valid intent. Note that `onClick()` may not be called immediately after an identification is made – actions on identified items may also be picked from Firefly's history.

Plugin Types

There are two types of Firefly plugins. Both types of plugin allow you to add additional actions to an identified item's detail page.

- **Simple** plugins do not perform business logic and always display their actions to the user based solely on the type of facets that are present in the digital entity. The sample plugin FoneVOIP (available in the SDK) is an example of a simple plugin. The FoneVOIP plugin is invoked when Firefly identifies a phone number. The user is given the option to dial the phone number every time, with no other logic to execute.
- **Resolving** plugins perform business logic and conditionally provide additional actions to the user. Typically a resolving plugin calls on an external service to perform the business logic. The sample plugin Exempli (available in the SDK) is an example of a resolving plugin. The Exempli plugin is invoked when Firefly identifies a musical artist. That plugin calls an external service to determine whether there are any upcoming performances by that artist within a 50-mile radius of the user's current location. Because the business logic is conditional, Firefly may not present any results or actions to the user.

To create a simple plugin, extend the `SimplePlugin` class. To create a resolving plugin, extend the `ResolvingPlugin` class. Both of these base classes have abstract methods you must implement. Those methods are:

- `createDigitalEntityUI(DigitalEntity)`, which is a factory method to create and return a new `DigitalEntityUI` that defines the plugin's user interaction.
- `getDigitalEntityFilter()`, which returns a `DigitalEntityFilter` that describes the types of facets that are relevant for this plugin.
- `getPluginDescription()`, which returns a one line description of the plugin's function.
- `onError(PluginError)`, which is called in the event of an error in either the configuration or runtime interaction with a plugin.

Additionally, your resolving plugin must implement `createDigitalEntityResolver(DigitalEntity)`. This is a factory method that returns a new `Resolver` instance that processes a `DigitalEntity` that matches the `DigitalEntityFilter` returned by `getDigitalEntityFilter()`. The `Resolver` instance is where you perform your business logic.

Resolvers for Resolving Plugins

The `Resolver` class allows a resolving plugin to extend a Firefly-identified item's information. For example, if Firefly identifies a song, the Exempli sample concert ticket plugin's `Resolver` could contact a web service and add information about upcoming concerts in the area by that artist to the item's `DigitalEntity`.

When Firefly invokes a resolving plugin, the code in the `Resolver` class's `resolve()` method is executed. This is where the business logic takes place. If the `resolve()` method does not return within seven seconds, Firefly may abandon the request and issue an error. Errors can be caught by your resolving plugin's `onError(PluginError)` method.

If, as a result of the business logic, there is relevant information to add to the `DigitalEntity`, your `resolve()` method should return a `Facet` of the type `MY_FACET`. The returned `Facet` should contain information needed for display by the `DigitalEntityUI`'s `Label`, as well as any information that needs to be passed as an intent extra to the plugin's app via the `DigitalEntityUI`'s `onClick()` method. If there is nothing relevant to add, `resolve()` returns null and a "no result" label is produced.

Method Calls in Context

The following steps outline the sequence of method calls that are made once Firefly identifies an item.

1. Firefly identifies an item and creates a digital entity with one or more facets.
2. Plugins with matching digital entity filters are identified and invoked by the system.
3. If the plugin is a resolving plugin, `createDigitalEntityResolver()` is called to create the resolver, and `Resolver.resolve()` is invoked. If `resolve()` returns null, processing stops.
4. For simple plugins and resolving plugins whose `resolve()` method did not return null, `createDigitalEntityUI()` is called and `DigitalEntityUI.getLabel()` is called to get the display label.
5. The user navigates to the detail page, either immediately after the identification is made, or at some later date from the Firefly history, and clicks on a label.
6. `createDigitalEntityUI()` is called and `DigitalEntityUI.onClick()` is called to perform the action.

Discovering and Registering Plugins

If your app has the appropriate manifest declarations (described in the next section), Firefly automatically registers the plugin and enables it in Firefly's left panel.

Configuring the Android Manifest

There are some changes you need to make to your `AndroidManifest.xml` in order to be able to develop and use Firefly plugins. The changes are the same for both simple and resolving plugins.

First, declare the package name in the `<manifest>` element. The package name becomes the plugin name in Firefly. The plugin will become a new plugin if the package name changes.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.amazon.mw.exempli"
          ...>
```

Second, request permission for this app to be recognized as a Firefly plugin.

```
<!-- declare the Firefly permission -->
<uses-permission android:name="com.amazon.mw.permission.PLUGIN" />
```

Third, within the `<application>` element, declare that you are using the Firefly library, including the required="false" attribute to allow the app to run on devices without Firefly capabilities.

```
<!-- Uses the Firefly library -->
<uses-library android:name="com.amazon.mw.sdk" android:required="false"/>
```

Fourth, also within the `<application>` element, register the Firefly plugin service and declare the `PLUGIN_CONSUMER` permission to protect the plugin by ensuring that only Firefly has permission to communicate with it. Also declare the Firefly intent filters.

```
<!-- Register the Firefly plugin service.

     The PLUGIN_CONSUMER permission protects the plugin by ensuring
     only Firefly has permission to communicate with it.

     exported="true" allows external applications or services (Firefly)
     to interact with this plugin. -->
<service android:name=".ExempliPlugin"
         android:permission="com.amazon.mw.permission.PLUGIN_CONSUMER"
         android:exported="true">
  <!-- Set the intent filter for Firefly plugins -->
  <intent-filter>
    <action android:name="amazon.intent.action.mw.IMPORT_PLUGIN" />
  </intent-filter>
</service>
```

The following example shows the previous code snippets in context.

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.amazon.mw.exempli"
          ...>

  ...

  <!-- declare the Firefly permission -->
  <uses-permission android:name="com.amazon.mw.permission.PLUGIN" />

```

```
  <application ...>

    ...

    <!-- Uses the Firefly library -->
    <uses-library android:name="com.amazon.mw.sdk" android:required="false"/>

    <!-- Register the Firefly plugin service.

         The PLUGIN_CONSUMER permission protects the plugin by ensuring
         only Firefly has permission to communicate with it.

         exported="true" allows external applications or services (Firefly)
         to interact with this plugin. -->
    <service android:name=".ExempliPlugin"
             android:permission="com.amazon.mw.permission.PLUGIN_CONSUMER"
             android:exported="true">

      <!-- Set the intent filter for Firefly plugins -->
      <intent-filter>
        <action android:name="amazon.intent.action.mw.IMPORT_PLUGIN" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

Sample Code

The SDK includes code samples that show how to build simple and resolving plugins.

Tips and Tricks

When developing a plugin you may notice caching behavior where a plugin doesn't get called if Firefly re-identifies the same item. To clear the cache, open the left panel, disable your plugin, then re-enable it.

Related Resources

- Firefly SDK
- Understanding Firefly
- Firefly Plugin Examples