Identity Services Programming Guide

Networking, Internet, & Web: Services & Discovery

2008-10-15

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Identity Services Programming Guide

Identity Services is a new technology in Mac OS X v10.5 that allows developers to access users and groups on a system in order to create customized access controls. Identity Services also introduces a new type of user, known as a sharing user. Sharing users are similar to standard users but do not have login access or a home directory. They are designed for users who only need access to network services such as file sharing or screen sharing.

Identity Services provides access to users and groups through two APIs. The Core Services Identity API supports user and group creation, enumeration, attribute inspection, credential management, and group membership management. The Collaboration framework is an Objective-C API providing access to identities, as well as managing a user interface element for selecting identities. All of these features can be combined for use in managing access control lists (ACLs).

This book describes the Identity Services architecture and explains how to leverage that architecture in new and existing Cocoa and Carbon applications. It is intended both for developers who want to use the Identity Services API and for system administrators who want to understand the infrastructure for users, groups, and access control lists.

Organization of This Document

This book contains the following chapters:

- "Identity Services Overview" (page 9) describes the underlying structure of Identity Services.
- "Using the Identity Picker" (page 15) explains how to select and create identities in a GUI-based application.
- "Finding and Monitoring Identities" (page 19) explains how to search for identities using the `CSIdentityQuery` and `CBIdentity` classes.
- "Working with Access Control Lists" (page 25) explains how to create, store, and load an ACL.

See Also

Refer to the following reference documents for Identity Services:

- *Identity Services Reference Collection*
- *Collaboration Framework Reference*
- *Core Services Identity Reference*

Identity Services Overview

To use the Identity Services APIs, it is important to know how identities work, and how they can be used. The following chapter describes the different types of identities and explains how to use them in Mac OS X. It also explains the APIs available for using identities in your applications.

What Is an Identity?

When most people think about a user account, they think about a home directory. However, a user account is much more complicated than that.

A user account, or **user**, works in a similar manner to a debit card. You normally think of your debit card as a method to purchase items. When you go to a store, you find what you want to buy and hand it to the cashier along with your debit card. The cashier scans the card, and makes sure your PIN number matches what the bank has on file. Assuming it does, you receive your items and walk out.

A user account on a computer works in a similar fashion. When you log into the computer, the computer asks for a user name and password. The user name is like the debit card number, it specifies who you are. The password is like your PIN number, because it proves you are who you say you are. After you're approved, the computer can give you access to your files.

But what is really happening when you use your debit card? When the cashier scans the card, the debit card number is sent to the bank. The bank has information about you, such as your name, birthday, social security number, and balance. The bank then tells the cashier whether the card is valid, so he knows if you can buy the item. The basis of the account is really the information stored on the computers at your bank, not the card itself.

In much the same way, a user is defined by its **identity**, which is a record that contains information about the user. Each identity is stored in a trusted directory known as an **identity authority**. Your user name is sent to the identity authority and your record is found. Then, if your password matches that on the record, you can get access to any file that your identity has access to.

Each identity contains the following required attributes:

- A universally unique identifier (UUID)
- A full name (must be unique throughout the system)
- A POSIX ID (UID for a user, GID for a group)
- A systemwide unique name (POSIX name)

Additionally, identities can contain the following optional attributes:

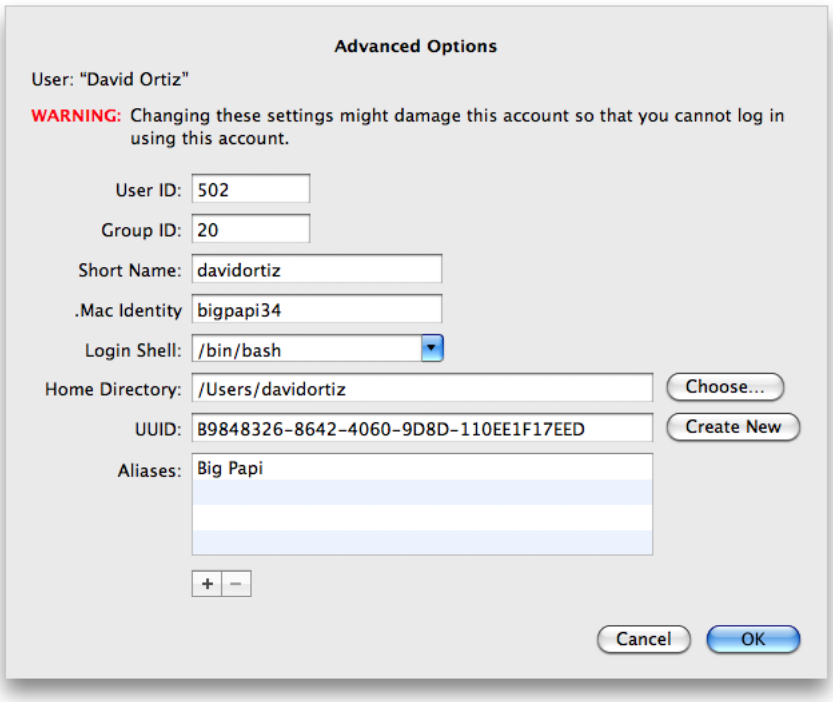
- A home directory (only for users)
- Alternate names, known as aliases

- An email address
- An image

You can edit these attributes using the Accounts preference pane in System Preferences by Control-clicking on the user. See Figure 1-1.

Warning: Use extreme caution when changing account attributes, as the changes might damage the account.

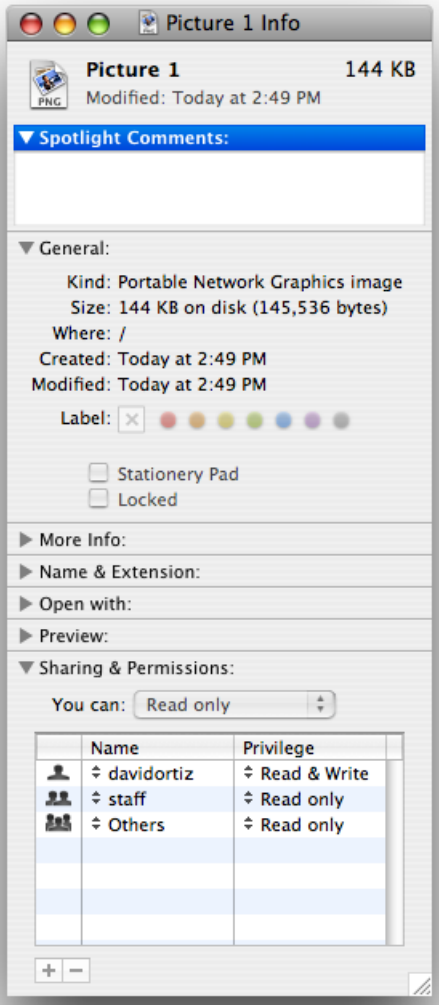
Figure 1-1 Advanced user options



Access Control Lists

Fundamentally, identities are used to control who has access to files and services. If you view the Get Info panel on a file in the Finder, you've seen the Ownership and Permissions section (Figure 1-2). This section displays what access users and groups have to the file or folder. These permissions (no access, read only, or read and write) are stored in an **access control list** (ACL). Every single folder and file on your computer has an associated ACL. Thus you can create very specific, customized access permissions for the files on your computer.

Figure 1-2 Ownership and permissions panel



Each sharing service (for example, SSH or AFP) on your computer also has an ACL. An ACL associated with a service is referred to as a **service access control list**, or SACL. The ability to create and customize ACLs and SACLs with users and groups helps make very secure, very adjustable applications.

Types of Identities

An identity can represent either a user or group:

A **user** is an identity that has **authentication credentials**. There are two forms of credentials: a traditional password, and a certificate. A user can be added to ACLs to access files or services. There are four different types of users: administrators, standard, child, and sharing. The list of user types and their features is available in Figure 1-3.

Figure 1-3 User properties

User	Administrator	Standard	Sharing	Child
Home Directory	✓	✓	✗	✓
Appears in Login Window	✓	✓	✗	✓
Parental Controls	✗	✗	✗	✓
Appears in ACL	✓	✓	✓	✓
Administrative Privileges	✓	✗	✗	✗

A **group** is an identity that has other identities as members. Groups can be added to ACLs and provide a powerful way to manage permissions. A group can contain an unlimited number of members.

API Architecture

There are two levels to the Identity Services APIs: the C-based Core Services Identity API and the Objective-C-based Collaboration framework. The Core Services Identity API is comprised of three Core Foundation-based objects that allow you to create, manipulate, and remove users and groups: an identity object, an identity query object, and an identity authority object. The Collaboration framework is a set of wrapper classes for the Core Services Identity API objects. Additionally, it provides user interface support for applications.

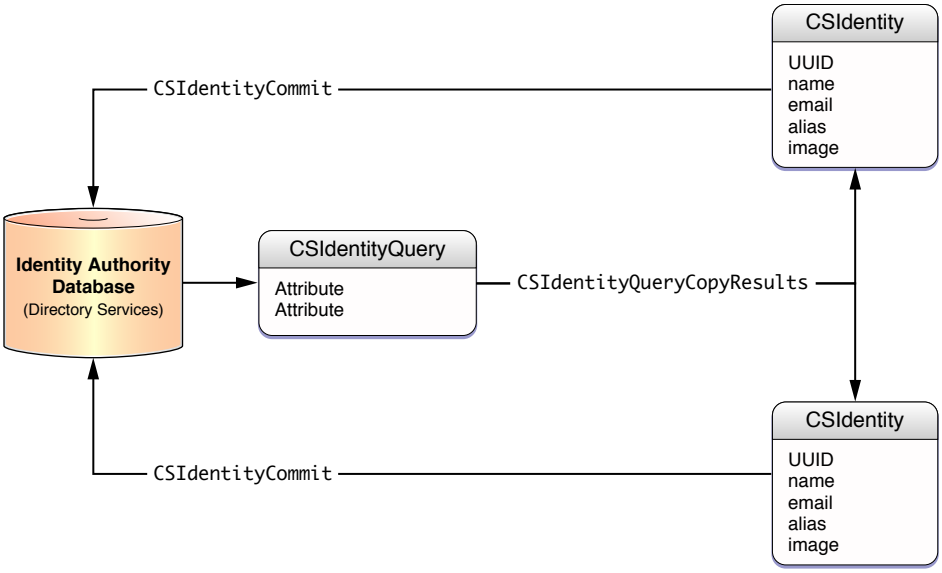
An **identity object** (`CSIdentity`) represents an identity in an identity authority and contains the attributes of that user or group. The required attributes are a full name, a POSIX name, a UUID, and a POSIX ID. Of these attributes, the POSIX name, the UUID and the POSIX ID can all be autogenerated by the API. User identities also include authentication information (password and/or certificate), and group identities have a list of members. These attributes can be set and retrieved using the appropriate methods, and the user or group represented by the identity object can be stored in the identity's authority database using a commit method. Each instantiation of the identity class represents one user or group. Identity objects are not limited to sharing users but can also be used to access standard users.

An **identity query** object (`CSIdentityQuery`) allows an application to search an identity authority database for users and groups and monitor them for changes. It provides methods for searching for identities by name, UUID, POSIX ID, reference data, or current user. In addition, an identity query object can register to be notified for changes to an identity. For example, if you have an identity object that represents a group, you can use an identity query object to notify you when the group membership changes.

An **identity authority object** (`CSIdentityAuthority`) represents a repository of identities. Every Mac OS X system contains a local identity authority that stores the identities local to the system. When you first create a user on a brand new Mac OS X system, the user identity is stored in the local identity authority. An identity authority can also exist on a network directory server. An identity authority object can represent either the local authority, the network-bound authorities, or both.

For a visual representation of how `CSIdentity` objects and `CSIdentityQuery` objects interact with an identity authority, see Figure 1-4.

Figure 1-4 Identity class interaction

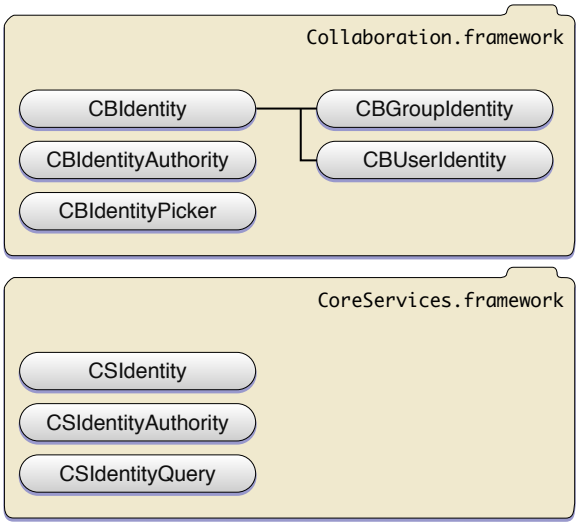


The Collaboration framework allows you to access identities with Objective-C APIs. It also houses the the Identity Picker user interface element. The `CBIdentity` and `CBIdentityAuthority` APIs provide the read-only access to attributes of the `CSIdentity` and `CSIdentityAuthority` objects.

The Collaboration framework also has methods to convert between the `CSIdentity` objects and `CBIdentity` objects. If you have a `CSIdentity` object and want to use it in Cocoa, use the `identityWithCSIdentity:` class factory method to create a `CBIdentity` object from the same identity. Similarly, if you have a `CBIdentity` object and want to use it in C, use the `CSIdentity` method on the identity object to return a `CSIdentity` object from the same identity.

The Identity Picker's relationship with Identity Services is analogous to the People Picker's relationship with the Address Book. Just as the People Picker is a user interface element for selecting a person from the Address Book, the Identity Picker is a user interface element for selecting identities. The Identity Picker is a sheet or a modal dialog that allows someone to select users on the system and promote Address Book entries to sharing users. It is managed by a `CBIdentityPicker` object. More information about the Identity Picker is available in "Using the Identity Picker" (page 15). The class hierarchy of the Collaboration framework can be seen in Figure 1-5.

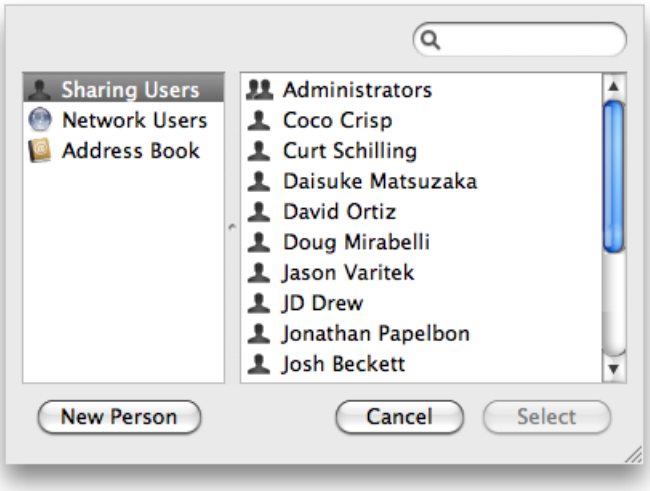
Figure 1-5 Identity Services class hierarchy



With a strong understanding of the structure of identities, you'll find the APIs much easier to use, and more powerful.

Using the Identity Picker

Most applications that use identities require someone to select users to be added to an access control list (ACL). Enter the Identity Picker. The Identity Picker is managed by the `CBIdentityPicker` class in the Collaboration framework. It allows users an easy interface for not only selecting identities but also for promoting entries in the Address Book to sharing users. As an application developer, no extra work is needed to allow users to create identities; this capability comes free from invoking the Identity Picker. This chapter describes how to customize the Identity Picker for your application and how to invoke the correct implementation of the Identity Picker.



Creating and Customizing the Identity Picker

For every Identity Picker your application requires, you should create a separate instance of the `CBIdentityPicker` class.

The ability of the Identity Picker to select multiple entries at once is a customizable feature. By default, the Identity Picker does not allow multiple selections. To change the default setting, call the `setAllowsMultipleSelection:` method and pass the value `YES`. See Listing 2-1 for an example of how to use these methods.

Listing 2-1 Customizing an `CBIdentityPicker` instance

```
// Instantiate an ABIdentityPicker object.
CBIdentityPicker *picker = [[CBIdentityPicker alloc] init];

// Allow the Identity Picker to select multiple entries.
[picker setAllowsMultipleSelection:YES];
```

Invoking the Identity Picker Sheet

The `beginSheetModalForWindow:modalDelegate:didEndSelector:contextInfo:` method invokes the Identity Picker as a sheet.

The first argument is the window in which the sheet should open. In most cases, this is `[sender window]`. The `modalDelegate` argument is the delegate object. The `didEndSelector` argument is the method that will be called when the user selects identities and chooses either the OK or Cancel button. It should be a method that contains three arguments of its own: a `CBIdentityPicker` object, an array of the identities selected, and a context. Finally, the `contextInfo` argument is any object you want sent to the delegate method.

Listing 2-2 Invoking the Identity Picker sheet

```
- (IBAction)plusButton:(id)sender
{
    [picker beginSheetModalForWindow:[sender window]
            modalDelegate:self
          didEndSelector:@selector(identityPickerDidEnd:identities:contextInfo:)
            contextInfo:nil];
}
```

When the user closes the Identity Picker, the delegate method (`identityPickerDidEnd:identities:contextInfo:`) is called. This method passes the selected identities as an array of `CBIdentity` objects and the context defined by the method `beginSheetModalForWindow:modalDelegate:didEndSelector:contextInfo:`. See Listing 2-3.

Listing 2-3 Retrieving identities from the Identity Picker

```
- (void)identityPickerDidEnd:(CBIdentityPicker *)identityPickerController
identities:(NSArray*)identities contextInfo:(void *)contextInfo
{
    NSEnumerator *enumerator = [identities objectEnumerator];
    CBIdentity *nextIdentity;

    while (nextIdentity = [enumerator nextObject]) {

        //Do something interesting with the nextIdentity object

    }
}
```

Invoking the Identity Picker Modal Dialog

To use the Identity Picker modal dialog, call the method `runModal`. This method invokes the modal identity picker. If the user selects the OK button in the Identity Picker window, you can return the selected identities with the `identities` method. The `identities` method returns an array of `CBIdentity` objects. See Listing 2-4.

Listing 2-4 Invoking the modal Identity Picker

```
- (IBAction)plusButton:(id)sender
{
    NSArray *identities;
    NSEnumerator *enumerator;
    CBIdentity *nextIdentity;

    if ([picker runModal] == NSOKButton) {
        identities = [picker identities];
        enumerator = [identities objectEnumerator];

        // Enumerate over the returned identities
        while ((nextIdentity = [enumerator nextObject])) {

            // Do something interesting with the identity object
        }
    }
}
```

Finding and Monitoring Identities

One of the most common uses of the Identity Services API is to look for an identity stored in an identity authority. You may also need to observe changes to users and groups that occur outside the scope of your application. For example, if another application removes a user from a group your application is using, you want to be notified of this change. In the Core Services Identity API, the `CSIdentityQuery` class provides synchronous and asynchronous access to find and monitor identities from an identity authority's database. In the Collaboration framework, these methods are part of the `CBIdentity` class.

This chapter explains how to search for identities using both `CSIdentityQuery` objects and `CBIdentity` objects.

Find an Identity

You can find an identity using either the Objective-C based Collaboration framework or the Core Services Identity API.

Using the Collaboration Framework

To find a user or group with the Collaboration framework, use one of the CBIdentity class factory methods. There are three methods that allow you to search based on different properties of an identity:

- If you want to search by full names, short names, or aliases, use the `identityWithName:authority:` method.
- If you want to search by UUID, use the `identityWithUUIDString:authority:` method.
- If you want to search using a persistent reference, use the `identityWithPersistentReference:` method. (For more information about persistent references, see "Loading an ACL" (page 26)).

To complete your search, pass a search term and an identity authority object to either the `identityWithName:authority:` and the `identityWithUUIDString:authority:` methods. There are a number of class factory methods in `CBIdentityAuthority` that allow you to create an identity authority object based on the identity authorities you want to search. Listing 3-1 (page 19) shows how to search for all identities named "David Ortiz" in a local identity authority.

Listing 3-1 Finding an identity in Objective-C

```
CBIdentityAuthority *localAuthority =
    [CBIdentityAuthority localIdentityAuthority];
CBIdentity *user =
    [CBIdentity identityWithName:@"David Ortiz" authority:localAuthority];
```

You can also search specifically for a user identity or a group identity by using the `CBUserIdentity` and `CBGroupIdentity` classes, respectively. By default, the `CBIdentity` class factory methods search for a user identities first, and if none are located then it looks for group identities.

Using the Core Services Identity API

To find a user or group with the Core Services Identity API you need to create a `CSIdentityQuery` object. A `CSIdentityQuery` object contains methods to search the identities database. It is important to use the appropriate method to create the identity query object based on how you want to search the database. The following methods are provided for you:

- If you want to search by full names, short names, or aliases, use the `CSIdentityQueryCreateForName` method.
- If you want to search by UUID, use the `CSIdentityQueryCreateForUUID` method.
- If you want to search by POSIX ID, use the `CSIdentityQueryCreateForPosixID` method.
- If you want to search by reference data (generated by the `CSIdentityCreatePersistentReference` method), use the `CSIdentityQueryCreateForPersistentReference` method.
- If you want to search for the current user's identity, use the `CSIdentityQueryCreateForCurrentUser` method.

There are two ways to execute the search, synchronously and asynchronously. It is highly recommended that you run any process that could block as a result of network delays asynchronously.

Search Identities Synchronously

To perform a `CSIdentityQuery` search synchronously, call the method `CSIdentityQueryExecute` on your identity query object. The method returns only when it has completed the search. If the query is executed successfully, `CSIdentityQueryExecute` returns `TRUE`; otherwise, it returns `FALSE`. Assuming the query was successful, run the `CSIdentityQueryCopyResults` method to return an array of identity objects. When you have finished retrieving the identities, make sure to release the `CSIdentityQuery` object. Listing 3-2 shows an example of this.

Listing 3-2 Finding identities synchronously

```
CSIdentityQueryRef query;
CFErrorRef error;
CFArrayRef identityArray;

// create the identity query based on name
query = CSIdentityQueryCreateForName(kCFAllocatorDefault,
                                     CFSTR("David"),
                                     kCSIdentityQueryStringBeginsWith,
                                     kCSIdentityClassUser,
                                     CSGetDefaultIdentityAuthority());

// execute the query
if (CSIdentityQueryExecute(query, kCSIdentityQueryGenerateUpdateEvents, &error))
{
    // retrieve the results of the identity query
    identityArray = CSIdentityQueryCopyResults(query);
```

```
    // do something with identityArray


}
CFRelease(query);
```

Search Identities Asynchronously

Performing an identity query asynchronously is similar to performing a query synchronously but differs in an important way. With a synchronous query, you execute the query, wait for it to complete, and then ask for the results. In contrast, with an asynchronous query, you start the query and your callback function will be passed the results as they become available. The process for setting up asynchronous callbacks is similar in theory and in practice to other Core Services callbacks.

To search for an identity asynchronously requires two main steps: setting up a callback function and adding the identity query object to a run loop. First, set up your callback function. The callback function must be a `void` function and must accept five arguments:

- `CSIdentityQueryRef query`, the identity query object
- `CSIdentityQueryEvent event`, the event that caused the callback function to be run
- `CFArrayRef identities`, the results of the query as an array identities
- `CFErrorRef error`, the error that occurred as a result of the identity query, if applicable
- `void *info`, any data placed in the `CSIdentityQueryClientContext`, to be sent to the callback function

A callback function might look like Listing 3-3.

Listing 3-3 Identity query callback function

```
void myIdentityQueryCallback (CSIdentityQueryRef query,
                              CSIdentityQueryEvent event,
                              CFArrayRef identities,
                              CFErrorRef error,
                              void *info) {

    // See what event triggered the callback
    switch ( event ) {

        case kCSIdentityQueryEventResultsAdded:
        // An identity was added to the list of results
        break;

        case kCSIdentityQueryEventSearchPhaseFinished:
        // The query was completed
        break;
    }
}
```

To add the identity query object to a run loop, first create the object. Then create a `CSIdentityQueryClientContext` structure. In the `CSIdentityQueryClientContext` structure, define the name of the callback function to be run. With the `CSIdentityQueryClientContext` structure set up, call the `CSIdentityQueryExecuteAsynchronously` method to add the query to a run loop.

`CSIdentityQueryExecuteAsynchronously` requires five arguments: the identity query object to be executed, the execution options (from `CSIdentityQueryFlags`), a pointer to your `CSIdentityQueryClientContext` structure, the run loop on which to schedule callbacks, and the run loop mode. If the query is added to the run loop, the function returns `TRUE` and any errors as a result of the query are sent to your callback function. See Listing 3-4 to see how this looks in code.

Listing 3-4 Adding an identity query object to a run loop

```
CSIdentityQueryRef query;
CSIdentityQueryClientContext queryclient =
    {0, NULL, NULL, NULL, NULL, myIdentityQueryCallback};

// create the identity query based on name
query = CSIdentityQueryCreateForName(kCFAllocatorDefault,
                                     CFSTR("David"),
                                     kCSIdentityQueryStringBeginsWith,
                                     kCSIdentityClassUser,
                                     CSGetDefaultIdentityAuthority());

// add the identity query object to the current run loop
if (!CSIdentityQueryExecuteAsynchronously(query,
                                          kCSIdentityQueryGenerateUpdateEvents,
                                          &queryclient,
                                          CFRunLoopGetCurrent(),
                                          kCFRunLoopCommonModes))
{
    // query was not added to the run loop
}
```

When an event is triggered based on your query, your callback function is run. The event that causes your callback function to run is passed to the callback function along with an array of identities. If identities from the query are added, removed, or modified, the array contains only those identities that have been affected. If the search is completed, then the array is `NULL`. In this case, use `CSIdentityQueryCopyResults` to get the full list of identities.

After the query has completed, you need to remove the identity query object from the run loop. To do this, call the `CSIdentityQueryStop` method and pass it the identity query object. Then, release the identity query object. This should look like the code in Listing 3-5.

Listing 3-5 Invalidating an identity query object

```
CSIdentityQueryStop(query);
CFRelease(query);
```

Continually Monitor Identities

Monitoring an identity is very similar to searching for one. If you are searching for an identity asynchronously, then all you need to do is to not call `CSIdentityQueryStop` when the query is completed. As long as your identity query object is registered on the run loop, it continues to notify you when the contents of your query change. So if you are searching for all Identities with the name "Chris", and a new user is created with the name "Chris Jones" after your original search finished, your callback function will be notified of this new user. When you are done monitoring the identities, make sure to call `CSIdentityQueryStop`.

To monitor identities synchronously, you need to poll an identity query object. Each identity query object can only be executed once, so after running `CSIdentityQueryExecute` and checking the results with `CSIdentityQueryCopyResults`, you will need to create an identical identity query object to execute again. Each time you run `CSIdentityQueryCopyResults` it will return an array with the full results of your query, not just what has changed. This is another reason why it is recommended that you search for and monitor identities asynchronously, rather than synchronously.

Working with Access Control Lists

A number of applications that take advantage of identities use them to control access to a file or service. The identities and their access levels are often stored in an access control list (ACL). The following chapter explains how to create an ACL, store it, and load it for use later.

Creating an ACL

An ACL can be maintained in any data object you want. This data object should be tailored based on your application's needs. For example, if your application needs to store a list of only those users and groups who can access your service, then a simple array or set may suffice. However, if your application needs to store a list of identities and their access level, you might want to use a dictionary. Listing 4-1 (page 25) shows how to create a dictionary object for you ACL and then run the identity picker. The identities returned from the picker are stored in the dictionary object as a key-value pair, where the key is the identity and the value is the permissions.

Listing 4-1 Creating an ACL with the Collaboration framework

```
// Create a dictionary to use as your ACL
NSMutableDictionary *accessControlList = [[NSMutableDictionary alloc] init];

// Run the identity picker
if ([picker runModal] == NSOKButton) {
    NSArray *identities = [picker identities];
    NSEnumerator *enumerator = [identities objectEnumerator];


    // Enumerate over the returned identities,
    // and add each one to the ACL
    while ((nextIdentity = [enumerator nextObject])) {

        // Make sure to use the identity object as the key, and the permissions
 level as the value
        [accessControlList setObject:@"read-only" forKey:nextIdentity];
    }
}
```

If you are using the Core Services Identity API, you can create a similar ACL with a `CFDictionary` object, using `CSIdentity` objects as the keys.

Writing an ACL to a File

When your application quits, it needs to store the ACL as a file. If your ACL is stored as an `NSDictionary` object, simply use the method `writeToFile:atomically:` to write the ACL to a plist file. When the file is written, each identity is stored as a persistent reference. The persistent reference of an identity is an opaque data object that is a faster, more reliable way to retrieve the identity from an identity authority than a UUID.

If you are not using a dictionary to house your ACL, use a persistent reference when you write the ACL to a file. In the Collaboration framework, use the method `persistentReference` to create a persistent reference for an identity or use the `NSCoding` protocol methods. In the Core Services Identity API, use the `CSIdentityCreatePersistentReference` method.

Loading an ACL

After you write your ACL to a file, your application needs to restore the ACL to a form it can access quickly. If you are using an `NSDictionary` object, instantiate a new object with the `dictionaryWithContentsOfFile:` method. Similarly, if you used the `NSCoding` protocol methods to archive the ACL, use the appropriate methods to unarchive the ACL. Either approach will also convert the persistent reference to an identity object automatically, so you can begin using the new dictionary object immediately. If an identity in the ACL has been removed, no conversion will take place.

If you wrote the ACL to a custom file format, you need to load the file back into memory, and load each persistent reference. You can instantiate the identity object from the persistent reference with the `identityWithPersistentReference:` method.

Retrieving an identity object in the Core Services API is a bit more involved. Once you have the presistent reference, create an identity query object using the `CSIdentityQueryCreateForPersistentReference` method. Then execute the query, and retrieve the returned identity objects. (For more information about querying an identity authority, see "Finding and Monitoring Identities" (page 19)).

Document Revision History

This table describes the changes to *Identity Services Programming Guide*.

Date	Notes
2008-10-15	Fixed typographical errors.
2007-05-15	New document that describes how to create, maintain, and search for users and groups.

Document Revision History