Pass Kit Programming Guide

Developer

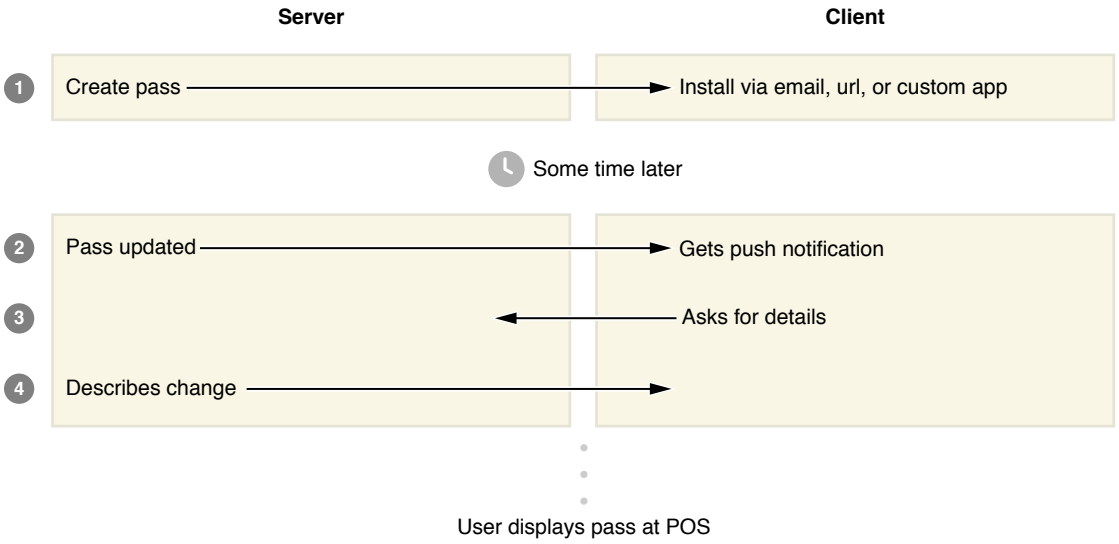
Contents

Figures and Listings

About Pass Kit

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

Passes are a digital representation of information that might otherwise be printed on small pieces of paper or plastic. They let users take an action in the physical world, in the same way as boarding passes, membership cards, and coupons. The pass library contains the user's passes. Users view and manage their passes using the Passbook app.



There are three major parts of this technology:

- A package format for creating passes.
- A web service API for updating passes, implemented on your server.
- An Objective-C API used by your apps to interact with the user's pass library.

At a Glance

This document covers the key concepts of the Pass Kit technology and explains the ways you can use ita.

Creating and Distributing Passes

You create a pass by writing a `pass.json` file and providing the appropriate images and localization data. All of these files are located inside a pass package, which is signed and compressed.

Passes can be distributed via email, websites, or your app.

Relevant Chapters "Creating a Pass" (page 6), "Building and Distributing a Pass" (page 14)

Updating Passes

Passes are updated using a push notification and a web service. When the user's device gets a push notification telling it that the pass has been updated, it queries your server to get the latest version of the passes that have changed.

Relevant Chapters "Updating a Pass" (page 17)

Interacting with the User's Passes

Your app can use the Objective-C API to add, update, and remove passes to provide a richer experience.

Relevant Chapters "Interacting with the User's Passes" (page 20)

See Also

Pass Kit Package Format Reference describes the package format for defining passes.

Pass Kit Web Service Reference describes the web service protocol for updating passes.

Pass Kit Framework Reference describes the Objective-C API for interacting with the user's pass library.

The Passbook support materials available in the developer downloads area contain fully-worked example passes, a script to help you sign passes, and a reference implementation of the web service.

Creating a Pass

Passes are created as a package which contains the following parts:

- The `pass.json` file defines the pass. This file is where you do most of your work to create a pass.
- Images such as your logo and the pass's icon.
- Localization projects (`.lproj` folders) contain strings files and localized images.
- The `manifest.json` file contains a hash of every file in the package.
- The `signature` file contains a signed hash of all the files, used to establish the pass's origin.

Setting Up your Development Environment

To create and distribute passes, you need a pass type identifier and signing certificate.

Go to the Developer Portal and request a pass type identifier and your certificate. This certificate is used in "Building and Distributing a Pass" (page 14) to sign passes, and in "Updating a Pass" (page 17) to update passes via push notifications.

The Top Level of the pass.json File

The `pass.json` file is a JSON dictionary. Listing 1-1 shows part of a pass:

Listing 1-1 Top-level keys of a pass

```
{
  "passTypeIdentifier" : "pass.com.example.myExamplePass",
  "formatVersion" : 1,
  "organizationName" : "Example Company",
  "serialNumber" : "123456",
  "teamIdentifier" : "A1B2C3D4E5",
  "description" : "Example pass showing top-level keys",
```

```
  ...

}
```

The pass type identifier, team identifier, and organization name match what you set up in the Developer Portal. The pass type identifier is conceptually similar to a bundle identifier or a class name. It defines a class or category of passes. For example, coupons for 5 or 10 percent off would use the same pass type identifier, but gift cards might use a different pass type identifier than a coupon. The exact meaning of each pass type identifier is determined by you and your apps. You specify which pass types the app is allowed to interact with in your app's entitlements file. This allows you to create a certain amount of separation between your apps when appropriate.

The serial number is a string that uniquely identifies your pass. A monotonically-increasing integer or is often a convenient way to assign the serial number, but you are free to assign them in any way that makes sense to you. The serial number is treated as an opaque piece of data by the PassKit framework. Two passes that have the same pass type identifier and serial number are understood to represent the same pass—any other data in the pass can be changed by updating the pass.

The description is a string that briefly describes the pass, used by the iOS accessibility technologies. The description should begin with a high-level term such as "Membership card", "Weekly coupon", or "Bus ticket" followed by one or two additional bits of information. Don't try to include all of the data on the pass in its description, just include enough detail to distinguish passes of the same type.

Setting the Pass Style

Different kinds of passes are displayed with a different visual style and lets you provide different fields. Unlike the pass type identifier which you define, you choose the pass style from a fixed list that is determined by the API. You indicate which style a pass uses by specifying one of the keys listed in "Style-Specific Dictionary Keys" in *Pass Kit Package Format Reference* . The value of this key is a dictionary containing fields specific to that style of pass. The contents of this dictionary is largely the same for all pass styles, except that a boarding pass requires the `transitType` key. For example, Listing 1-2 shows a boarding pass for a flight from San Francisco to London.

Listing 1-2 Boarding pass with formatted fields

```
{
  "passTypeIdentifier" : "pass.com.example.myExamplePass",
  "formatVersion" : 1,
```

```
"serialNumber" : "123456",
"description" : "Boarding pass for April 4, San Francisco to London",
"boardingPass" : {
  "primaryFields" : [
    {
      "key" : "origin",
      "label" : "San Francisco",
      "value" : "SFO"
    },
    {
      "key" : "destination",
      "label" : "London",
      "value" : "LHR"
    }
  ],
  "secondaryFields" : [
    {
      "key" : "board-gate",
      "changeMessage" : "Gate changed to %@.",
      "label" : "Gate",
      "value" : "F12"
    },
    {
      "key" : "board-time",
      "changeMessage" : "Boarding time changed to %@.",
      "dateStyle" : "PKDateStyleFull",
      "timeStyle" : "PKTimeStyleFull",
      "label" : "Boards",
      "value" : "2012-04-01T0700-8"
    },
  ],
  "auxilaryFields" : [
    {
      "key" : "seat",
```

```
        "label" : "Seat",
        "value" : "7A"
      },
      {
        "key" : "passenger-name",
        "label" : "Passenger",
        "value" : "John Appleseed"
      }
    ],
    "backFields" : [
      {
        "key" : "freq-flier-num",
        "label" : "Frequent flier number",
        "value" : "1234-5678"
      },
    ]
  "transitType" : "PKTransitTypeAir"
  }
}
```

Declaring a Pass's Fields

Passes have **fields** which contain information that is displayed to the user in a specific way. Most of the information in a pass is part of a field. Some examples of fields are the name and venue of an event, the departure time for an airline boarding pass, and the current balance on a gift card. Some fields are always required, and some are required for a given pass style, and some are always optional.

Every field has a unique key. The Objective-C API gives apps access to pass data using the `localizedValueForFieldKey:` method.

Fields contain a value and a label (which are displayed in the UI), a unique key, and information about how they should be presented. This information is stored in a field dictionary. Listing 1-3 shows an event ticket with three fields: the band's name, the venue, and the concert time. The `event-name` field in the listing has the bare minimum for a field: just a key and its value. The `start-time` field in the listing gives the event's starting time and date, and it also gives information about how the field should be displayed and what message should be shown to the user if the field's value changes.

To provide a message when the field's value changes, use the `changeMessage` key. If the change message is not provided, the value is updated silently.

Listing 1-3 Example of an event ticket

```
{
  "passTypeIdentifier" : "pass.com.example.myExamplePass",
  "formatVersion" : 1,
  "serialNumber" : "123456",
  "description" : "Concert ticket to see The Hectic Glow",

  ...

  "eventTicket" : {
    "primaryFields" : {
      "key": "event-name",
      "value" : "The Hectic Glow in concert",
    },
    {
      "key": "venue-name",
      "label" : "Venue",
      "value" : "Joe\u2019s Coffee Shop",
    },
    {
      "changeMessage" : "Concert time changed to %@.",
      "dateStyle" : "PKDateStyleMedium",
      "timeStyle" : "PKTimeStyleMedium",
      "isRelative" : true,
      "label" : "Starts in",
      "key" : "start-time",
      "value" : "2012-12-31T20:03Z"
    }
  }
}
```

Formatting a Pass's Fields

Fields with a number or a date can specify how they should be formatted.

To specify the format for a number, provide a value for the `numberStyle` key. For a list of the supported number styles, see "Number Style Keys" in *Pass Kit Package Format Reference* . For decimal numbers that represent money, provide the ISO currency code as the value of the `currencyCode` key.

To specify a the format for a date and time, provide a value for both the `dateStyle` and the `timeStyle` keys. For a list of the supported date and time formats, see "Date Style Keys" in *Pass Kit Package Format Reference* .To display it as a relative date, set the value of the `isRelative` key to `true`.

To insert a newline in the value of a field, use `\n`.

Adding Relevance Information

To make you pass accessible directly from the lock screen, provide the time and location that it is relevant using the `relevantDate` and `locations` keys. This lock screen integration makes it easier for users to find the passes they need quickly and easily. Listing 1-4 shows an part of an example pass which provides relevance information.

Listing 1-4 Pass with relevance information

```
{
  "passTypeIdentifier" : "pass.com.example.myExamplePass",
  "formatVersion" : 1,
  "serialNumber" : "123456",
  "description" : "Example pass showing relevance information",

  ...

  "locations" : [
    {"latitude" : 37.296, "longitude" : -122.038 },
    {"latitude" : 37.302, "longitude" : -122.103},
    {"altitude" : 67.0, "latitude" : 37.331, "longitude" : -122.029 }
  ],
  "relevantDate" : "2010-02-05T09:00-08"
}
```

The location is an array of location dictionaries, each of which contains a latitude longitude and optional altitude, that describe locations where users can take some action with the pass. A pass can have up to ten locations specified. For example, a store card could include the locations for your stores that are closest to the user, or the stores that the user visits most frequently. Relevance information can be updated, just like any other data in the pass, as described in "Updating a Pass" (page 17). For example, if a flight time changes you can push an update with the new time, or if the user updates their profile on your website you can update the locations to match their current city.

Adding a Barcode

A pass can contain a barcode, which is convenient way to quickly and accurately enter information in a wide variety of applications—for example, at a checkout or before boarding a train. To add a barcode, specify the barcode type and its contents. The pass can display the barcode in several formats: QR, PDF417, Aztec, and plain text. Listing 1-5 shows an example pass that contains a barcode. You must specify the encoding of the message. You should typically use the ISO 8859-1 encoding (also known as Latin-1), which tends to be the expected encoding for barcode scanners and software, but you can use another encoding if your equipment supports it.

> **Note** Laser scanners don't typically work well for reading barcodes from the screen of an iOS device. Use an optical barcode scanner instead. All of the supported barcode formats are two dimensional.

Listing 1-5 Barcode information

```
{
  "passTypeIdentifier" : "pass.com.example.myExamplePass",
  "formatVersion" : 1,
  "serialNumber" : "123456",
  "description" : "Example pass showing barcode information",

  ...

  "barcode" : {
    "format" : PKBarcodeFormatQR,
    "message" : "Hello world!",
    "messageEncoding" : "iso-8859-1"
  }
```

```
}
```

Localizing a Pass

To localize a pass, provide a strings file and localized images for each locale, as described in "Localized Resources in Bundles" in *Bundle Programming Guide* , and in "Localizing String Resources" in *Internationalization Programming Topics* .

For example, given the following pass fragment:

```
{
  "formatVersion" : 1,
  "organizationName" : "Example Company",
  ...
}
```

As a contrived example, the strings file to localize it into Backward English is as follows:

```
/* Organization name */
"Example Company" = "ynapmoC elpmaxE";
```

In addition to the strings file, localizing into Backward English would require looking at the images in the pass and providing a Backward English version of anything that needs to be displayed differently in that language.

Building and Distributing a Pass

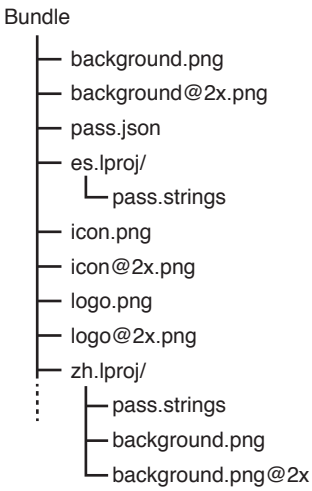
There are a few things you need to do to your pass before it is ready to distribute:

- Create the manifest file.
- Create the signature by cryptographically signing the manifest.
- Compress the pass.

Making the Manifest

The manifest is a JSON dictionary in a file named `manifest.json`. The keys are paths to files in the pass package, relative to the pass. The value of each key is the SHA-1 hash of that file's contents. For example, Figure 2-1 shows a directory structure of a pass package and Listing 2-1 shows the corresponding manifest:

Figure 2-1 Directory structure of a sample pass



Listing 2-1 Example manifest file

```
{
  "background.png" : "844a6063e4192f4f4f34b2cf36996b6b06a6f355",
  "background@2x.png" : "56c66001a5edb87c2b58180daa3e443dcac887e4",
  "pass.json" : "a4f8506e362888755ddf744365cc3cf615e4e6b1",
```

```
  "es.lproj/pass.strings" : "b698506e362888755ddf744365cc3cf615e4e6b1",
  "icon.png" : "105d0f906f633c378d738477fef0d51e0ccec2d2",
  "icon@2x.png" : "f5c3db953176da14d6d1c3c27de12e14119173da",
  "logo.png" : "78a778accde869cea3364bb828074d7a8f0067ce",
  "logo@2x.png" : "af77501cac762637bdb4545b3b758ae4b4632422",
  "zh.lproj/pass.strings" : "a4f8506e362888755ddf744365cc3cf615e4e6b1",
  "zh.lproj/background.png" : "2888755ddfa4f8506e36744365cc3cf615e4e6b1",
  "zh.lproj/background.png@2x" : "f8506e362a4888755ddf744365cc3cf615e4e6b1"
}
```

Making the Signature

The signature is a detached PKCS#7 signature of the manifest. For information about obtaining and installing your certificate, see "Setting Up your Development Environment" (page 6). Listing 2-2 illustrates the process using the `openssl` command, as you might do during development.

Listing 2-2 Terminal commands to create the signature

```
openssl pkcs12 -in "My PassKit Cert.p12" -clcerts -nokeys -out certificate.pem
openssl pkcs12 -in "My PassKit Cert.p12" -nocerts -out key.pem
# You can keep certificate.pem and key.pem to use again later,
# or you can delete them after signing.

openssl smime -binary -sign \
    -signer certificate.pem -inkey key.pem \
    -in manifest.json -out signature \
    -outform DER
```

The signature establishes that the pass came from you. This provides protection against an attacker distributing passes that appear to originate from you because the attacker would not be able to put your signature on a pass.

However, you should handle information displayed on a pass the same way you would handle information on a paper card. You wouldn't trust the balance written on a paper gift card because an attacker might change it. Instead, you give each gift card a unique number and keep a list of their current balance. You can control

your list of current balances, so you can ensure that the list trustworthy and accurate. Likewise, providing the current balance on a pass is very convenient for users, but you should verify the balance against a trusted source at the point of sale.

Compressing the Pass

Use ZIP to compress the contents of the pass package into a single file with the extension `.pkpass` and distribute the resulting file. For example, execute the following command in Terminal:

```
zip -r example.pkpass path/to/pass_package/* -x '*.DS_Store'
```

Getting the Pass to the User

There are three ways to distribute passes to your users:

- Attach the pass to an email, taking advantage of the UI in Mail for adding a pass to the library.
- Host the pass on a web server and provide the URL to your users—for example, as a link on your website or in an email—taking advantage of the UI in Safari for adding a pass to the library.
- Write an app that interacts with the pass library, as described in "Interacting with the User's Passes" (page 20).

If you are attaching the pass to an email or hosting it on the web, declare the MIME type as `application/vnd.apple.pkpass` so that Mail or Safari handle it as a pass.

Updating a Pass

Updating passes is optional, but adds significant value for your users. Updates let you change the contents of a pass after it is added to the user's pass library. For example, if a flight is delayed, sending an update lets you make the user aware of the new departure time. Any data in the pass except for the pass type identifier and serial number can be changed by pushing an update.

To update passes, you need to be able to send push notifications and you need to set up a web server that implements a REST-style web service protocol.

To indicate that a pass can receive updates, add the `webServiceURL` and `authenticationToken` keys to the pass, as shown in Listing 3-1. The **authentication token** is a shared secret between the user's device and your server. It shows that the request for an update to a pass is actually coming from the user who has the pass, and not from a third party. For more information about shared secrets and other topics in cryptography, see *Security Overview*.

Listing 3-1 Pass with authentication token and web service URL

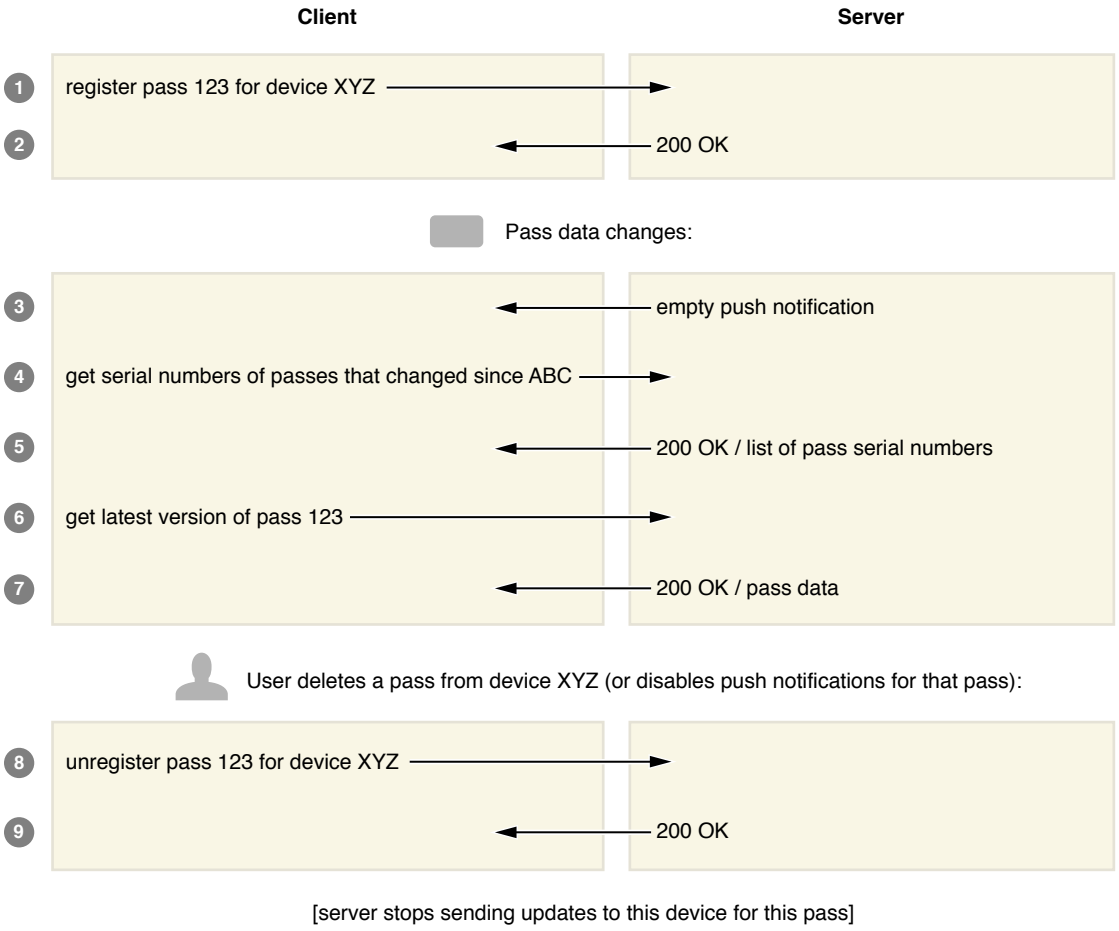
```
{
  "passTypeIdentifier" : "pass.com.example.myExamplePass",
  "formatVersion" : 1,
  "serialNumber" : "123456",
  "description" : "Example pass showing web service information",

  ...

  "authenticationToken" : "e45ffb6c383742ccb992f5e42d45adf",
  "webServiceURL" : "https://example.com/pass-updates/"
}
```

There are five endpoints that your web service needs to respond to. Some parts of the URL are defined by the protocol and tell your server what is being requested, and some parts are defined in the pass. Figure 3-1 illustrates several typical interactions with your server; note that in the case of an updated pass, there is more than one round trip.

Figure 3-1 Interaction between the client and your server



Starting Updates

When the user adds a pass to the pass library and enables updates on that pass, a request is sent to your server.

The **device identifier** is determined by the device, and is used as a shared secret between the user's device and your server. It uniquely identifies a device *and* indicates that the entity making the request is authorized to make such a request. For example, in the request for all passes, the entity making the request doesn't supply an authorization token. Authorization tokens are specified per pass, so there is no appropriate token—the device identifier is sufficient to prove that the request is valid.

Treat the device ID as an opaque piece of data that might change. It's not intended for you to do aggregation or correlation of requests.

Updating a Pass

To update a pass, first send an empty push notification to the device with a the pass type ID as the push topic. When the user's device receives the push notification, it requests a list of serial numbers that have changed, and then requests the latest version of passes.

Stopping Updates

When the user deletes a pass or disables push notifications for that pass, a request is sent to your server. When your server receives this request, it stops sending updates to that device about that pass.

Interacting with the User's Passes

The Pass Kit framework provides an Objective-C API for interacting with the user's pass library. There are two main kinds of apps that use the Pass Kit framework: apps that add passes to the pass library like Mail and Safari do, and apps that integrate tightly with the pass library and interact with its contents to provide a richer experience. For example, a FidoNet mail client just needs to act as a conduit and add passes to the library. An app for a cafe could let customers place an order and add a pass to let them claim their order at the counter.

The `PKPassLibrary` class represents the pass library, and the `PKPass` class represents individual passes. The framework also provides a view controller, the `PKAddPassesViewController` class, which displays a pass and lets users add it to their pass library. These classes give you model-level access to the pass library. Your app is responsible for the presentation of the data.

Adding Passes to the Library

To add a pass to the library:

1. Use the `isPassLibraryAvailable` class method of the `PKPassLibrary` class to check that the pass library is available.
2. Create an instance of the `PKPass` class for the pass, initializing it with the pass's data.
3. Use the `containsPass:` method of the `PKPassLibrary` class to check whether the pass is in the library. Your app can use this method to detect the presence of a pass, even if it doesn't have the entitlements to read passes in the library.
4. If the pass isn't in the library, use an instance of the `PKAddPassesViewController` class to let the user add it.

Accessing a Pass

You can access the description using the `localizedDescription` method.

The `PKPass` class provides properties for some data. You can use the `localizedValueForFieldKey:` method to get the value of arbitrary fields.

If created the pass, you defined the fields and chose the key, so you should know what key you want. In contrast, if the pass was is from another source and your app is just acting as a conduit, you don't know the keys; such apps can just use the properties.

Changing Passes

To find a pass, use the `passWithPassTypeIdentifier:serialNumber:` method of the `PKPassLibrary` class to find a pass.

To replace a pass, use the `replacePassWithPass:` method of the `PKPassLibrary` class. The pass type identifier and serial number of the existing pass must match the pass that is replacing it.

To remove a pass from the pass library, use the `removePass:` method of the `PKPassLibrary` class.

Document Revision History

This table describes the changes to *Pass Kit Programming Guide*.

Date	Notes
2012-07-11	New document that explains how to define passes, update them, and prompt users to add them to their library.

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