



LoopBack - Component Documentation

Copyright 2016 StrongLoop, an IBM company

1. LoopBack components	3
1.1 Creating components	4
1.2 Push notifications	6
1.2.1 Push notifications for iOS apps	14
1.2.2 Push notifications for Android apps	16
1.2.3 Push notification API	25
1.2.3.1 Installation API	25
1.2.3.2 Notification API	26
1.2.3.3 PushManager API	27
1.2.3.4 Installation REST API	29
1.2.3.5 Push Notification REST API	31
1.2.4 Tutorial: Push notifications	32
1.2.4.1 Tutorial: push notifications - LoopBack app	32
1.2.4.2 Tutorial: push notifications - Android client	36
1.2.4.3 Tutorial: push notifications - iOS client	45
1.2.4.4 Tutorial: push notifications - putting it all together	48
1.3 Storage component	52
1.3.1 Storage component API	56
1.3.2 Storage component REST API	59
1.4 Third-party login (Passport)	61
1.4.1 Configuring providers.json	66
1.4.2 Tutorial: third-party login	70
1.5 Synchronization	72
1.5.1 Sync example app	81
1.5.2 Advanced topics - sync	83
1.5.3 Tutorial: Offline Synchronization	84
1.6 OAuth 2.0	86
1.7 API Explorer	89

LoopBack components

- Overview
 - Component contract
- Pre-defined LoopBack components
 - API Explorer
 - OAuth2
 - Push Notifications
 - Storage
 - Third-party login (Passport)

Overview

LoopBack *components* are predefined packages that extend a basic LoopBack application. Fundamentally, a component is related code bundled together as a unit to enable LoopBack applications for easy reuse. You can configure components declaratively in `component-config.json`.

The bare minimum to meet the LoopBack component "contract" is to export a `function(app, options)` as the main module export.

A LoopBack application itself is nothing more than a grouping of components with elements to tie them all together.

Component contract

To be a LoopBack component, a module must export a function with the following signature as the main module export:

```
function(app, options)
```

Compare that with Express middleware, which exports `function(options)` that is supposed to return `function(req, res, next)` or `function(err, req, res, next)`.

Pre-defined LoopBack components

LoopBack provides several pre-defined components, as described in the table below.

The sections below describe the configuration settings for each component in `component-config.json`.

Component	Description	Module
API Explorer	Enables the Swagger UI for the API. See Use API Explorer for an example.	<code>loopback-component-explorer</code>
OAuth 2.0	Enables LoopBack applications to function as oAuth 2.0 providers to authenticate and authorize client applications and users to access protected API endpoints.	<code>loopback-component-oauth2</code>
Push Notifications	Adds push notification capabilities to your LoopBack application as a mobile back end service.	<code>loopback-component-push</code>
Storage component	Adds an interface to abstract storage providers like S3, filesystem into general containers and files.	<code>loopback-component-storage</code>
Synchronization	Adds replication capability between LoopBack running in a browser or between LoopBack back-end instances to enable offline synchronization and server-to-server data synchronization.	Built into LoopBack; will be refactored into <code>loopback-component-sync</code>
Third-party login (Passport)	Adds third-party login capabilities to your LoopBack application like Facebook, GitHub etc.	<code>loopback-component-passport</code>

API Explorer

The `slc loopback application generator` will scaffold an app with `component-config.json` containing the default entry for LoopBack API Explorer:

server/component-config.json

```
{
  "loopback-explorer": {
    "mountPath": "/explorer"
  }
}
```

OAuth2

Example:

server/component-config.json

```
{
  "loopback-component-oauth2": {
    "dataSource": "db",
    "loginPage": "/login",
    "loginPath": "/login",
    "addHttpHeaders": "X-"
  }
}
```

Push Notifications



This component does not yet meet the "contract" to be a LoopBack component.

Storage



This component does not yet meet the "contract" to be a LoopBack component.

Third-party login (Passport)



This component does not yet meet the "contract" to be a LoopBack component.

Creating components

- [Overview](#)
- [How to create components](#)
- [Registering components](#)

Overview

LoopBack *components* are predefined packages that extend a basic LoopBack application. Fundamentally, a component is related code bundled together as a unit to enable LoopBack applications to easily reuse it. They are registered to the app using `component-config.json`.

Components can be official [StrongLoop npm packages](#) or custom components. This page is about custom components.

How to create components

A LoopBack component is basically a node module which accepts a LoopBack application instance and an accompanying configuration object. Using the available LoopBack application instance and the configuration object, the functionality of the component can be programmed according to the requirements.



If a component is not published on npm, the convention is to put the component file in the `server/components` directory.

The following are some examples to help you understand the components API.

In this example, the component mounts a middleware on the path specified in the `path`, property of the `options` object, which prints a message "Your Component", when a request is made to the path.

server/components/my-component.js

```
module.exports = function (loopbackApplication, options) {
  loopbackApplication.use(options.path, function (req, res, next) {
    res.send('Your Component');
  });
};
```

In this example, the component mounts a middleware which prints the `options` object to the console.

server/components/show-options.js

```
module.exports = function (loopbackApplication, options) {
  loopbackApplication.use(function (req, res, next) {
    console.log(options);
    next();
  });
};
```



To better understand how routing works in LoopBack and how it affects components, refer [Routing](#).

While a lot of components mount a middleware of some kind, all components need not do so. For example, this component prints the `loopback` version on initialization, and does nothing more than that.

server/components/version.js

```
module.exports = function (loopbackApplication) {
  var version = loopbackApplication.loopback.version;
  console.log('LoopBack v%s', version);
};
```

The `options` parameter passed to the component function is the value specified in the `component-config.json` file for the component. The value can be any valid JavaScript data type. The next section explains how to register components using the `component-config.json` file, and specify its `options` property.

Registering components

An application will load all components that have an entry in the `component-config.json` that does not evaluate to false. If the component is installed using `npm`, use its package name as the key; if the component is a local component, specify its relative path from the `server` directory.

Here is an example of a `component-config.json` file with entries for a component installed via `npm`, and the three local components described earlier.

```
{
  "loopback-component-explorer": {
    "mountPath": "/explorer"
  },
  "./components/my-component": {
    "path": "/my-component"
  },
  "./components/show-options": {},
  "./components/version": true
}
```

Push notifications



StrongLoop Labs

This project provides early access to advanced or experimental functionality. It may lack usability, completeness, documentation, and robustness, and may be outdated.

However, StrongLoop supports this project: Paying customers can open issues using the StrongLoop customer support system (Zendesk). Community users, please report bugs on GitHub.

For more information, see [StrongLoop Labs](#).

- [Overview](#)

- [Installation](#)
- [Use the LoopBack push notification sample application](#)
 - [Set up messaging credentials for Android apps](#)
 - [Set up messaging credentials for iOS apps](#)
 - [Run the sample server application](#)
- [Set up your LoopBack application to send push notifications](#)
 - [Create a push model](#)
 - [Configure the application with push settings](#)
 - [Register a mobile application](#)
 - [Register a mobile device](#)
 - [Send push notifications](#)
 - [Send out the push notification immediately](#)
 - [Schedule the push notification request](#)
 - [Error handling](#)
- [Architecture](#)

See also:

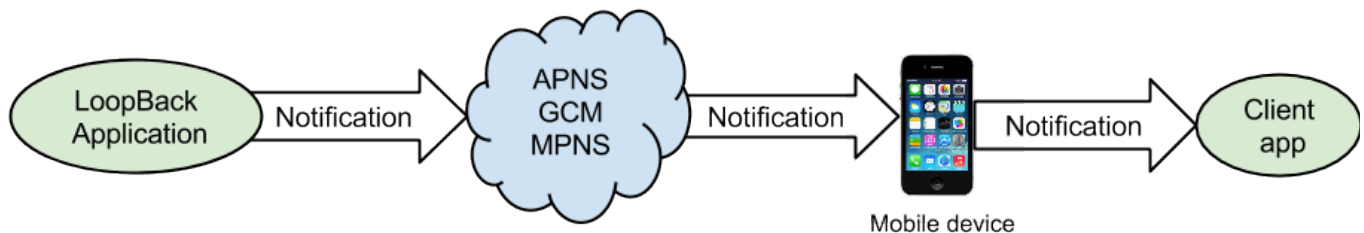
- [API reference](#)
- [Example server application](#)
- [Example iOS app](#)
- [Example Android app](#)

Overview

Push notifications enable server applications (known as *providers* in push parlance) to send information to mobile apps even when the app isn't in use. The device displays the information using a "badge," alert, or pop up message. A push notification uses the service provided by the device's operating system:

- **iOS** - Apple Push Notification service (APNS)
- **Android** - Google Cloud Messaging (GCM)

The following diagram illustrates how it works.



To send push notifications, you'll need to create a LoopBack server application, then configure your iOS and Android client apps accordingly. You can either use the example LoopBack push application or create your own. Using the example application is a good way to learn about LoopBack push notifications.

For more information on the architecture of the LoopBack Push Notification component, see [Architecture](#) below.



For information on how to implement real-time "push-like" notifications to web clients, see [Realtime server-sent events](#).

Installation

Install the LoopBack push component as usual for a Node package:

```
$ npm install loopback-component-push
```

Use the LoopBack push notification sample application



If you are creating your own LoopBack server application, skip this section and go to [Set up your LoopBack application to send push notifications](#).

First, download the sample app:

```
$ git clone https://github.com/strongloop/loopback-example-push.git
```

Set up messaging credentials for Android apps

First, if you haven't already done so, [get your Google Cloud Messaging \(GCM\) credentials](#) for Android apps. After following the instructions, you will have a GCM API key. Then edit your application's `config.js` to add them; for example, in the sample app, `loopback-2.x/server/config.js`:

```
exports.gcmServerApiKey = 'Your-server-api-key';
```

Replace `Your-server-api-key` with your GCM server API key. For example:

```
exports.gcmServerApiKey = 'AIzaSyDEPWYN9Dxf3xD0qbQluCwuHsGfK4aJehc';
```

Set up messaging credentials for iOS apps

If you have not already done so, [create your APNS certificates](#) for iOS apps. After following the instructions, you will have APNS certificates on your system. Then edit your application's `config.js` to add them; for example, in the sample app, `loopback-2.x/server/config.js`:

```
exports.apnsCertData = readCredentialsFile('apns_cert_dev.pem');
exports.apnsKeyData = readCredentialsFile('apns_key_dev.pem');
```

Replace the file names with the names of the files containing your APNS certificates. By default, `readCredentialsFile()` looks in the `/credentials` sub-directory for your APNS certificates.

If you don't have a client app yet, leave the default `appName` in `config.js` for now. Once you have created your client app, update the `appName`.

Now follow the instructions in:

- [Push notifications for Android apps](#) to set up Android client apps.

- [Push notifications for iOS apps](#) to set up iOS client apps

Run the sample server application

First install all dependencies, then run the Node application as follows:

```
$ cd loopback-example-push/loopback
$ npm install
$ bower install
...
$ node .
```

The server application will be available at <http://localhost:3000/>.

Set up your LoopBack application to send push notifications

Follow the directions in this section to configure your own LoopBack application to send push notifications. It may be helpful to refer to the [example LoopBack application](#).

Create a push model

The code below illustrates how to set up a push model with a MongoDB database as the data source, as shown in the push example app.

server/model-config.json

```
{
  ...
  "push": {
    "public": true,
    "dataSource": "push"
  }
  ...
}
```

To send push notifications, you must create a push model. The database is used to load and store the corresponding application/user/installation models.

server/datasources.json

```
{
  "db": {
    "connector": "mongodb",
    "url": "mongodb://demo:L00pBack@demo.strongloop.com/demo"
  },
  "push": {
    "name": "push",
    "connector": "loopback-component-push",
    "installation": "installation",
    "notification": "notification",
    "application": "application"
  }
}
```

Configure the application with push settings

Register a mobile application

The mobile application needs to register with LoopBack so it can have an identity for the application and corresponding settings for push services. Use the Application model's `register()` function for sign-up.

For information on getting API keys, see:

- [Get your Google Cloud Messaging credentials](#) for Android.
- [Set up iOS clients](#) for iOS.

For example, here is how the sample app registers a mobile client:

loopback-2.x/server/push-demo.js

› [Expand](#)

[source](#)

```
module.exports = function (app) {
  var Notification = app.models.notification;
  var Application = app.models.application;
  var PushModel = app.models.push;

  function startPushServer() {
    // Add our custom routes
    var badge = 1;
    app.post('/notify/:id', function (req, res, next) {
      var note = new Notification({
        expirationInterval: 3600, // Expires 1 hour from now.
        badge: badge++,
        sound: 'ping.aiff',
        alert: '\uD83D\uDCE7 \u2709 ' + 'Hello',
        messageFrom: 'Ray'
      });

      PushModel.notifyById(req.params.id, note, function (err) {
        if (err) {
          console.error('Cannot notify %j: %s', req.params.id, err.stack);
          next(err);
          return;
        }
        console.log('pushing notification to %j', req.params.id);
        res.send(200, 'OK');
      });
    });

    PushModel.on('error', function (err) {
      console.error('Push Notification error: ', err.stack);
    });
  }

  // Pre-register an application that is ready to be used for testing.
  // You should tweak config options in ./config.js

  var config = require('./config');

  var demoApp = {
    id: 'loopback-component-push-app',
    userId: 'strongloop',
    name: config.appName,

    description: 'LoopBack Push Notification Demo Application',
    pushSettings: {
      apns: {
```

```

    certData: config.apnsCertData,
    keyData: config.apnsKeyData,
    pushOptions: {
      // Extra options can go here for APN
    },
    feedbackOptions: {
      batchFeedback: true,
      interval: 300
    }
  },
  gcm: {
    serverApiKey: config.gcmServerApiKey
  }
}
};

updateOrCreateApp(function (err, appModel) {
  if (err) {
    throw err;
  }
  console.log('Application id: %j', appModel.id);
});

//--- Helper functions ---
function updateOrCreateApp(cb) {
  Application.findOne({
    where: { id: demoApp.id }
  },
  function (err, result) {
    if (err) cb(err);
    if (result) {
      console.log('Updating application: ' + result.id);
      delete demoApp.id;
      result.updateAttributes(demoApp, cb);
    } else {
      return registerApp(cb);
    }
  });
}

function registerApp(cb) {
  console.log('Registering a new Application...');
  // Hack to set the app id to a fixed value so that we don't have to change
  // the client settings
  Application.beforeSave = function (next) {
    if (this.name === demoApp.name) {
      this.id = 'loopback-component-push-app';
    }
    next();
  };
  Application.register(
    demoApp.userId,
    demoApp.name,
    {
      description: demoApp.description,
      pushSettings: demoApp.pushSettings
    },
    function (err, app) {
      if (err) {

```

```
        return cb(err);
    }
    return cb(null, app);
}
);
}
```

```
startPushServer();
};
```

Register a mobile device

The mobile device also needs to register itself with the backend using the Installation model and APIs. To register a device from the server side, call the `Installation.create()` function, as shown in the following example:

```
Installation.create({
  appId: 'MyLoopBackApp',
  userId: 'raymond',
  deviceToken: '756244503c9f95b49d7ff82120dc193cae3a7cb56f60c2ef2a19241e8f33305',
  deviceType: 'ios',
  created: new Date(),
  modified: new Date(),
  status: 'Active'
}, function (err, result) {
  console.log('Registration record is created: ', result);
});
```

Most likely, the mobile application registers the device with LoopBack using REST APIs or SDKs from the client side, for example:

```
POST http://localhost:3010/api/installations
{
  "appId": "MyLoopBackApp",
  "userId": "raymond",
  "deviceToken":
"756244503c9f95b49d7ff82120dc193cae3a7cb56f60c2ef2a19241e8f33305",
  "deviceType": "ios"
}
```

Send push notifications

Send out the push notification immediately

LoopBack provides two Node.js methods to select devices and send notifications to them:

- `notifyById()`: Select a device by registration ID and send a notification to it.
- `notifyByQuery()`: Get a list of devices using a query (same as the **where** property for `Installation.find()`) and send a notification to all of them.

For example, the code below creates a custom endpoint to send out a dummy notification for the selected device:

```

var Notification = app.models.notification;
var Application = app.models.application;
var PushModel = app.models.push;
function startPushServer() {
// Add our custom routes
var badge = 1;
app.post('/notify/:id', function (req, res, next) {
  var note = new Notification({
    expirationInterval: 3600, // Expires 1 hour from now.
    badge: badge++,
    sound: 'ping.aiff',
    alert: '\uD83D\uDCE7 \u2709 ' + 'Hello',
    messageFrom: 'Ray'
  });
  PushModel.notifyById(req.params.id, note, function (err) {
    if (err) {
      console.error('Cannot notify %j: %s', req.params.id, err.stack);
      next(err);
      return;
    }
    console.log('pushing notification to %j', req.params.id);
    res.send(200, 'OK');
  });
});
PushModel.on('error', function (err) {
  console.error('Push Notification error: ', err.stack);
});

```

To select a list of devices by query, use the `PushManager.notifyByQuery()`, for example:

```

PushManager.notifyByQuery({userId: {inq: selectedUserIds}}, note, function(err) {
  console.log('pushing notification to %j', selectedUserIds);
});

```

Schedule the push notification request



This feature is not yet available. When you are ready to deploy your app, [contact StrongLoop](#) for more information.

Error handling

LoopBack has two mechanisms for reporting push notification errors:

- Most configuration-related errors are reported to the callback argument of notification functions. These errors should be reported back to the caller (HTTP client) as can be seen in the `notifyById()` code example above.
- Transport-related errors are reported via "error" events on the push connector. The application should listen for these events and report errors in the same way as other errors are reported (typically via `console.error`, `bunyan`, and so forth.).

```

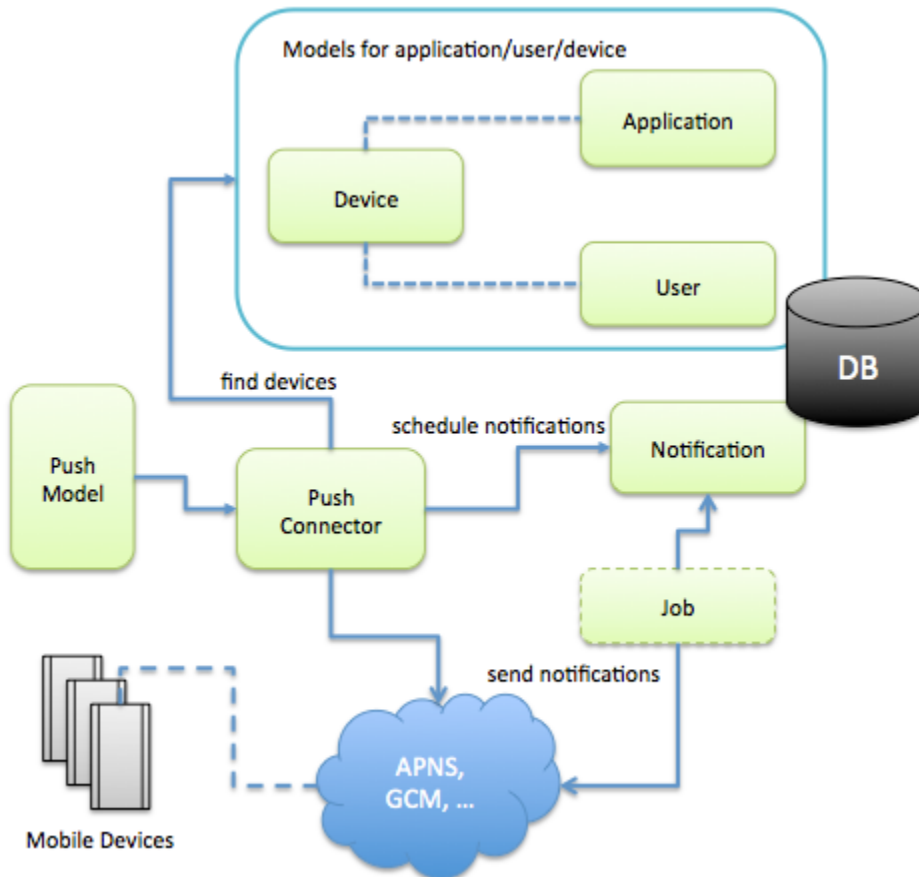
PushModel.on('error', function(err) {
  console.error('Push Notification error: ', err.stack);
});

```

Architecture

The following diagram illustrates the LoopBack push notification system architecture. The components are:

- Device model and APIs to manage devices with applications and users.
- Application model to provide push settings for device types such as iOS and Android.
- Notification model to capture notification messages and persist scheduled notifications.
- Optional job to take scheduled notification requests.
- Push connector that interacts with device registration records and push providers APNS for iOS apps and GCM for Android apps.
- Push model to provide high-level APIs for device-independent push notifications.



Push notifications for iOS apps

- [Overview](#)
- [Configure APN push settings in your server application](#)
- [Add LoopBack iOS SDK as a framework](#)
- [Initialize LBRESTAdapter](#)
- [Register the device](#)
- [Handle received notifications](#)

Related articles:

See also:

- [Android SDK API docs](#)

Overview

This article provides information on creating iOS apps that can get push notifications from a LoopBack application. See [Push notifications](#) for information on creating the corresponding LoopBack server application.

The basic steps to set up push notifications for iOS clients are:

1. Provision an application with Apple and configure it to enable push notifications.
2. Provide a hook to receive the device token when the application launches and register it with the LoopBack server using the `LBInstallation` class.
3. Provide code to receive notifications, under three different application modes: foreground, background, and offline.
4. Process notifications.

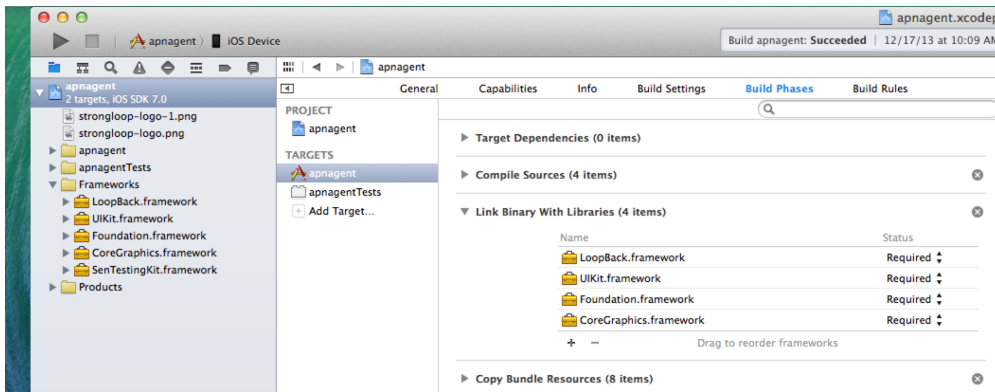
For general information on the Apple push notifications, see [Apple iOS Local and Push Notification Programming Guide](#). For additional useful information, see [Delivering iOS Push Notifications with Node.js](#).

Configure APN push settings in your server application

Please see [Register a mobile application](#).

Add LoopBack iOS SDK as a framework

Open your XCode project, select targets, under build phases unfold **Link Binary with Libraries**, and click on '+' to add LoopBack framework.



The LoopBack iOS SDK provides two classes to simplify push notification programming:

- [LBInstallation](#) - enables the iOS application to register mobile devices with LoopBack.
- [LBPushNotification](#) - provides a set of helper methods to handle common tasks for push notifications.

Initialize LBRESTAdapter

The following code instantiates the shared `LBRESTAdapter`. In most circumstances, you do this only once; putting the reference in a singleton is recommended for the sake of simplicity. However, some applications will need to talk to more than one server; in this case, create as many adapters as you need.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.settings = [self loadSettings];
    self.adapter = [LBRESTAdapter adapterWithURL:[NSURL
URLWithString:self.settings[@"RootPath"]]];

    // Reference to Push notifs List VC
    self.pnListVC = (NotificationListVC *)[(UINavigationController
*)self.window.rootViewController viewControllers]
                                objectAtIndex:0];

    LBPushNotification* notification = [LBPushNotification application:application
                                didFinishLaunchingWithOptions:launchOptions];

    // Handle APN on Terminated state, app launched because of APN
    if (notification) {
        NSLog(@"Payload from notification: %@", notification.userInfo);
        [self.pnListVC addPushNotification:notification];
    }

    return YES;
}
```

Register the device

```

- (void)application:(UIApplication*)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
{
    __unsafe_unretained typeof(self) weakSelf = self;

    // Register the device token with the LoopBack push notification service
    [LBPushNotification application:application
didRegisterForRemoteNotificationsWithDeviceToken:deviceToken
    adapter:self.adapter
    userId:@"anonymous"
    subscriptions:@[@"all"]
    success:^(id model) {
        LBInstallation *device = (LBInstallation *)model;
        weakSelf.registrationId = device._id;
    }
    failure:^(NSError *err) {
        NSLog(@"Failed to register device, error: %@", err);
    }

];

...
}

- (void)application:(UIApplication*)application
didFailToRegisterForRemoteNotificationsWithError:(NSError*)error {
    // Handle errors if it fails to receive the device token
    [LBPushNotification application:application
didFailToRegisterForRemoteNotificationsWithError:error];
}

```

Handle received notifications

```

- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo {
    // Receive push notifications
    LBPushNotification* notification = [LBPushNotification application:application
didReceiveRemoteNotification:userInfo];

    [self.pnListVC addPushNotification:notification];
}

```

Push notifications for Android apps

- [Overview](#)
- [Prerequisites](#)
 - [Configure Android Development Tools](#)
 - [Get your Google Cloud Messaging credentials](#)
- [Install and run LoopBack Push Notification app](#)
- [Configure GCM push settings in your server application](#)
- [Prepare your own Android project](#)
- [Check for Google Play Services APK](#)
- [Create LocalInstallation](#)
- [Register with GCM if needed](#)
- [Register with LoopBack server](#)
- [Handle received notifications](#)
- [Troubleshooting](#)

Related articles:

See also:

- [Android SDK API docs](#)

Overview

This article provides information on creating Android apps that can get push notifications from a LoopBack application. See [Push notifications](#) for information on creating the corresponding LoopBack server application.

To enable an Android app to receive LoopBack push notifications:

1. Setup your app to use Google Play Services.
2. On app startup, register with GCM servers to obtain a device registration ID (device token) and register the device with the LoopBack server application.
3. Configure your LoopBack application to receive incoming messages from GCM.
4. Process the notifications received.

Prerequisites

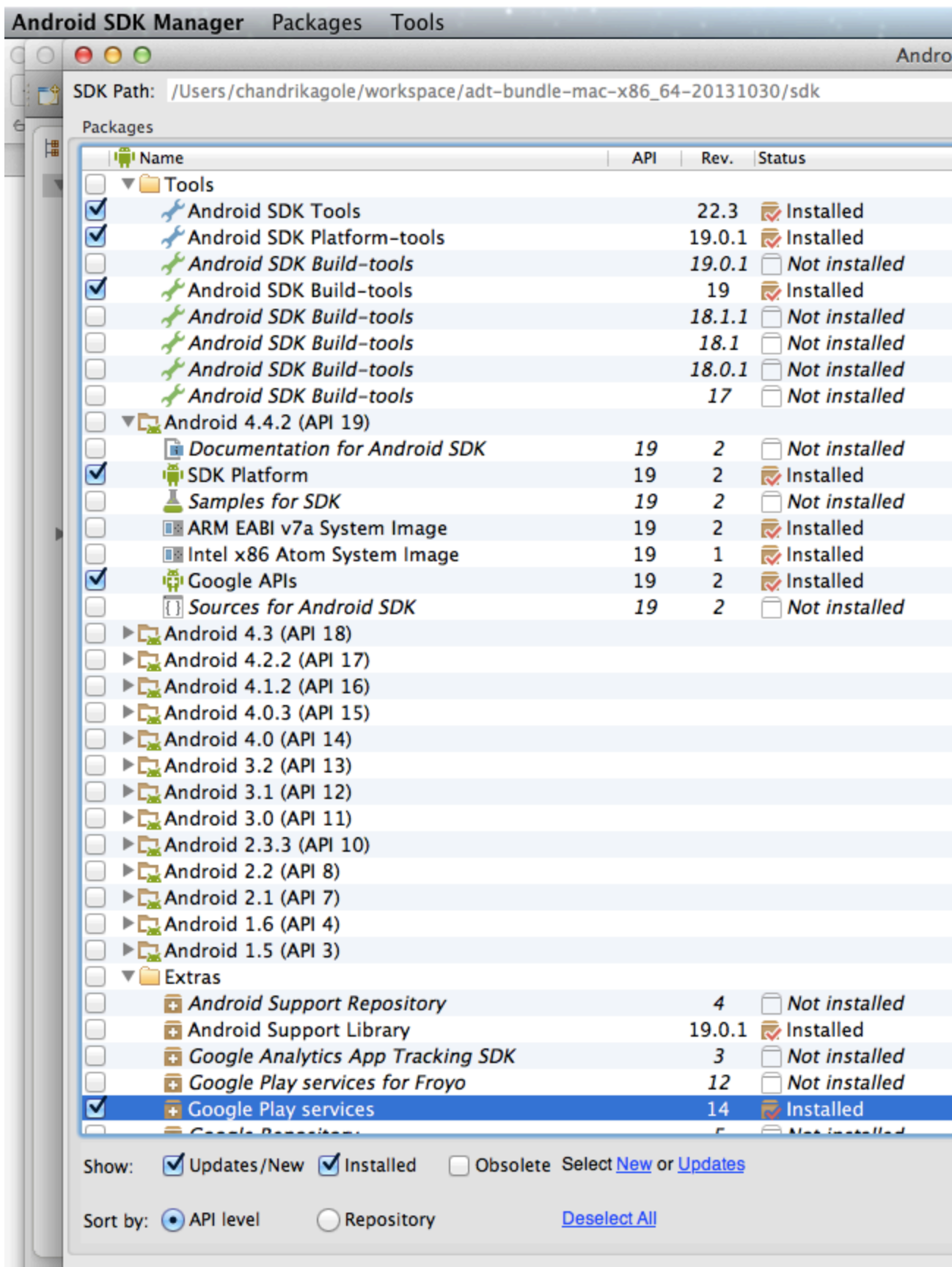
Before you start developing your application make sure you've performed all the prerequisite steps outlined in this section.

- **Download the [LoopBack Android SDK](#)**
- **Install [Eclipse development tools \(ADT\)](#)**

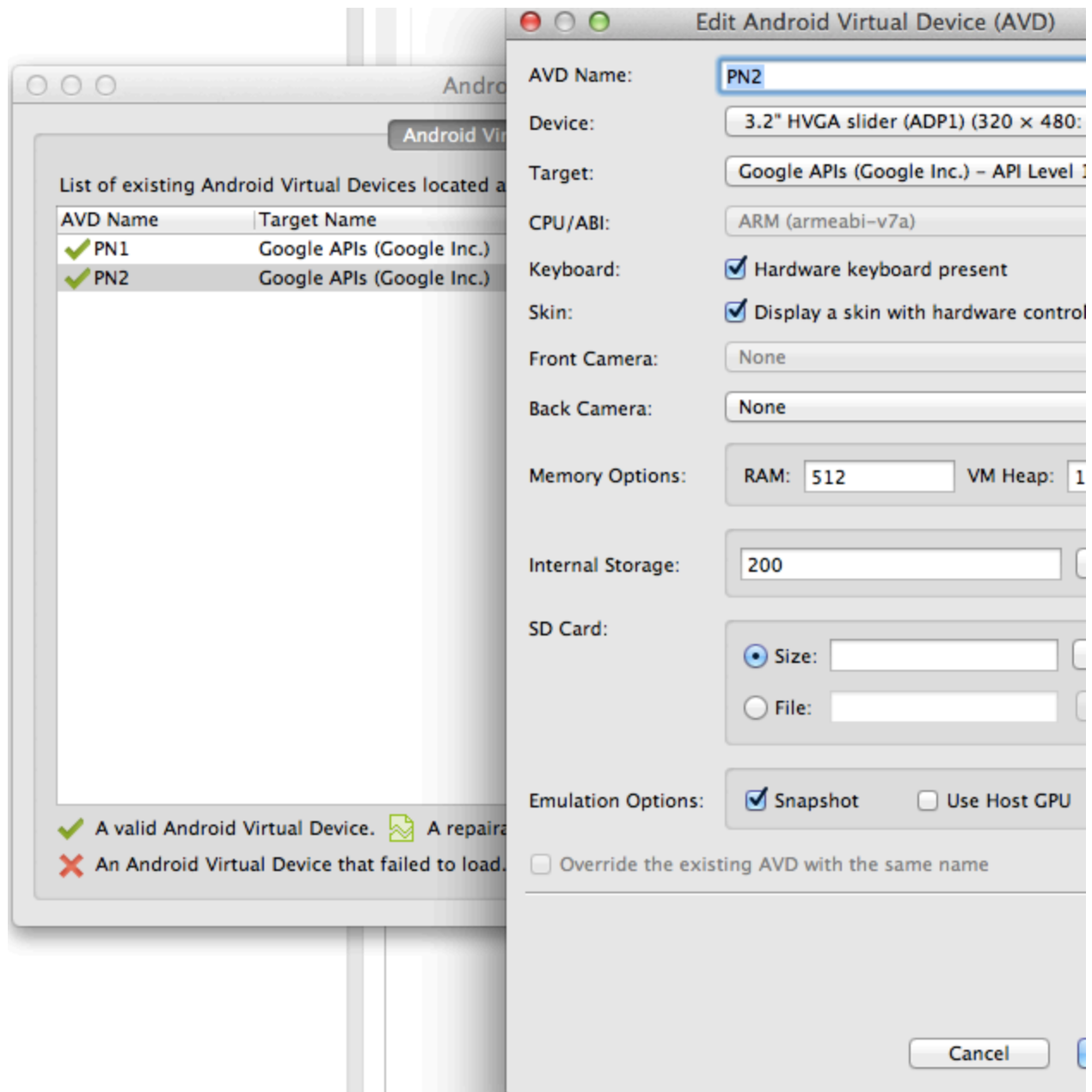
Configure Android Development Tools

Now configure Eclipse ADT as follows:

1. Open Eclipse from the downloaded ADT bundle.
2. In ADT, choose **Window > Android SDK Manager**.
3. Install the following if they are not already installed:
 - Tools:
 - Android SDK Platform-tools 18 or newer
 - Android SDK Build-tools 18 or newer
 - Android 4.3 (API 18):
 - SDK Platform.
 - Google APIs
 - Extras:
 - Google Play Services
 - Intel x86 Emulator Accelerator (HAXM)



- Before you start, make sure you have set up at least one Android virtual device: Choose **Window > Android Virtual Device Manager**.
- Configure the target virtual device as shown in the screenshot below. See [AVD Manager](#) for more information.



If you are using the virtual device suggested above, you must also install the ARM EABI v7a System Image SDK.

Get your Google Cloud Messaging credentials

[Open the Android Developer's Guide](#) To send push notifications to your Android app, you need to setup a Google API project and enable the Google Cloud Messaging (GCM) service.

Follow the instructions to get your GCM credentials:

1. Follow steps to create a Google API project and enable the GCM service.
2. Create an Android API key
 - a. In the sidebar on the left, select **APIs & auth > Credentials**.
 - b. Click **Create new key**.
 - c. Select **Android key**.
 - d. Enter the SHA-1 fingerprint followed by the package name, for example
 45:B5:E4:6F:36:AD:0A:98:94:B4:02:66:2B:12:17:F2:56:26:A0:E0;com.example
 NOTE: Leave the package name as "com.example" for the time being.
3. You also have to create a new server API key that will be used by the LoopBack server:

- a. Click **Create new key**.
- b. Select **Server key**.
- c. Leave the list of allowed IP addresses empty for now.
- d. Click **Create**.
- e. Copy down the API key. Later you will use this when you configure the LoopBack server application.

Install and run LoopBack Push Notification app

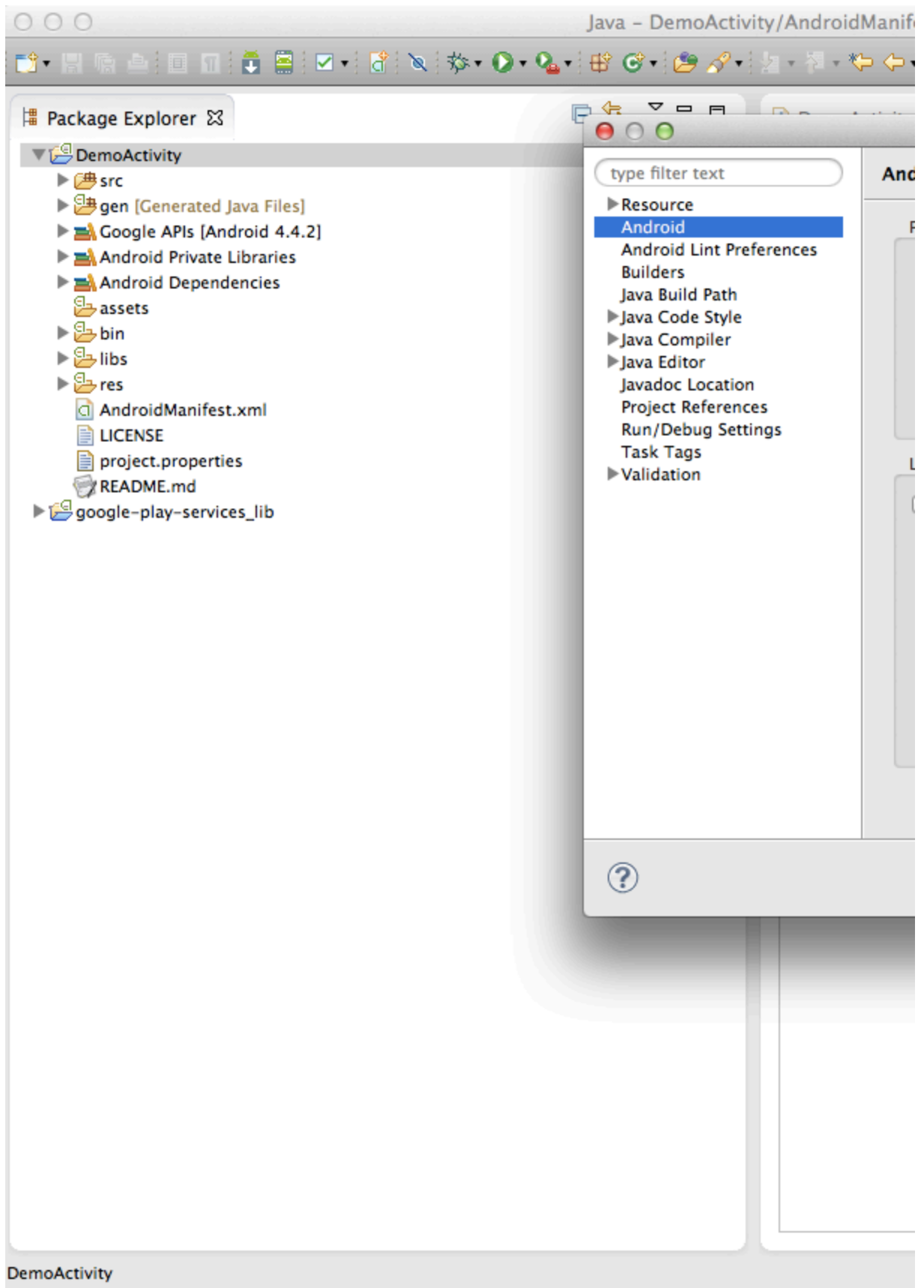
If you want to use the sample Android client app, download the [Push Notification Example Android app](#). Then follow these steps to run the app:

1. Open ADT Eclipse.
2. Import the push notification application to your workspace:
 - a. Choose **File > Import**.
 - b. Choose **Android > Existing Android Code into Workspace**.
 - c. Click **Next**.
 - d. Browse to the example Android app you just downloaded.
 - e. Click **Finish**.



ADT does not take long to import the guide app. Don't be misguided by the progress bar at the bottom of the IDE window: it indicates memory use, not loading status.

3. Import Google Play Services library project into your workspace. The project is located inside the directory where you have installed the Android SDK.
 - a. Choose **File > Import**.
 - b. Choose **Android > Existing Android Code into Workspace**.
 - c. Click **Next**.
 - d. Browse to the <android-sdk>/extras/google/google_play_services/libproject/google-play-services_lib directory.
 - e. Check **Copy projects into workspace**
 - f. Click **Finish**.
- See [Google Play Services SDK](#) for more details.
4. Add the imported google-play-services_lib as an Android build dependency of the push notification application.
 - a. In the Package Explorer frame in Eclipse, select the push notification application.
 - b. Choose **File > Properties**.
 - c. Select **Android**.
 - d. In the Library frame, click on **Add...** and select google-play-services_lib.
 - e. Also under Project Build Target, set the target as Google APIs.



5. Edit `src/com/google/android/gcm/demo/app/DemoActivity.java`.
 - Set `SENDER_ID` to the project number from the Google Developers Console you created earlier in [Get your Google Cloud Messaging credentials](#).
6. Go back to the <https://cloud.google.com/console/project> and edit the Android Key to reflect your unique application ID. Set the value of **A**

ndroid applications to something like this:

Android applications	XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX LOOPBACK_APP_ID X:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX;com.google.android.gcm.demo.app.DemoApplication
-----------------------------	--

- Run the [LoopBack server application](#) you set up earlier. If you didn't set the `appName` in the server application's `config.js` earlier, do it now.
Set it to **"com.google.android.gcm.demo.app.DemoActivity"**.
- Click the green **Run** button in the toolbar to run the application. Run it as an Android application. You will be prompted to select the target on which to run the application. Select the AVD you created earlier.



It may take several minutes to launch your application and the Android virtual device the first time.



Due to a [known issue with Google Play Services](#), you must download and import an older version of Google Play services.

- Download https://dl-ssl.google.com/android/repository/google_play_services_3225130_r10.zip
- Extract the zip file.
- In Eclipse ADT, right-click on your project and choose **Import...**
- Choose **Existing Android Code into Workspace** then click Next.
- Click **Browse...**
- Browse to the `google-play-services/libproject/google-play-services_lib/` directory created when you extracted the zip file and select it in the dialog box.
- Click **Finish**.

You must also update `AndroidManifest.xml` as follows:

- In Eclipse ADT, browse to `DemoActivity/AndroidManifest.xml`.
- Change the line


```
<meta-data android:name="com.google.android.gms.version"
  android:value="@integer/google_play_services_version" />
```

 to


```
<meta-data android:name="com.google.android.gms.version" android:value="4030500" />
```
- Save the file.

Configure GCM push settings in your server application

Add the following key and value to the push settings of your application:

```
{
  gcm: {
    serverApiKey: "server-api-key"
  }
}
```

Replace `server-api-key` with the API key you obtained in [Get your Google Cloud Messaging credentials](#).

Prepare your own Android project

Follow the instructions in [Android SDK documentation](#) to add LoopBack Android SDK to your Android project.

Follow the instructions in Google's [Implementing GCM Client guide](#) for setting up Google Play Services in your project.



To use push notifications, you must install a compatible version of the Google APIs platform. To test your app on the emulator, expand the directory for Android 4.2.2 (API 17) or a higher version, select **Google APIs**, and install it. Then create a new AVD with Google APIs as the platform target. You must install the package from the SDK manager. For more information, see [Set Up Google Play Services](#).

Check for Google Play Services APK

Applications that rely on the Google Play Services SDK should always check the device for a compatible Google Play services APK before using Google Cloud Messaging.

For example, the following code checks the device for Google Play Services APK by calling `checkPlayServices()` if this method returns true, it proceeds with GCM registration. The `checkPlayServices()` method checks whether the device has the Google Play Services APK. If it doesn't, it displays a dialog that allows users to download the APK from the Google Play Store or enables it in the device's system settings.

```
@Override
public void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    if (checkPlayServices()) {
        updateRegistration();
    } else {
        Log.i(TAG, "No valid Google Play Services APK found.");
    }
}

private boolean checkPlayServices() {
    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
    if (resultCode != ConnectionResult.SUCCESS) {
        if (GooglePlayServicesUtil.isUserRecoverableError(resultCode)) {
            GooglePlayServicesUtil.getErrorDialog(resultCode, this,
                PLAY_SERVICES_RESOLUTION_REQUEST).show();
        } else {
            Log.i(TAG, "This device is not supported.");
            finish();
        }
        return false;
    }
    return true;
}
```

Create LocalInstallation

Once you have ensured the device provides Google Play Services, the app can register with GCM and LoopBack (for example, by calling a method such as `updateRegistration()` as shown below). Rather than register with GCM every time the app starts, simply store and retrieve the registration ID (device token). The `LocalInstallation` class in the LoopBack SDK handles these details for you.

For more information on `LocalInstallation`, see [Working with the LocalInstallation class](#).

The example `updateRegistration()` method does the following:

- Lines 3 - 4: get a reference to the shared `RestAdapter` instance.
- Line 5: Create an instance of `LocalInstallation`.
- Line 13: Subscribe to topics.
- Lines 15-19: Check if there is a valid GCM registration ID. If so, then save the installation to the server; if not, get one from GCM and then save the installation.

```

private void updateRegistration() {

    final DemoApplication app = (DemoApplication) getApplication();
    final RestAdapter adapter = app.getLoopBackAdapter();
    final LocalInstallation installation = new LocalInstallation(context, adapter);

    // Substitute the real ID of the LoopBack application as created by the server
    installation.setAppId("loopback-app-id");

    // Substitute a real ID of the user logged in to the application
    installation.setUserId("loopback-android");

    installation.setSubscriptions(new String[] { "all" });

    if (installation.getDeviceToken() != null) {
        saveInstallation(installation);
    } else {
        registerInBackground(installation);
    }
}

```

Register with GCM if needed

In the following code, the application obtains a new registration ID from GCM. Because the `register()` method is blocking, you must call it on a background thread.

```

private void registerInBackground(final LocalInstallation installation) {
    new AsyncTask<Void, Void, Exception>() {
        @Override
        protected Exception doInBackground(final Void... params) {
            try {
                GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);
                // substitute 12345 with the real Google API Project number
                final String regid = gcm.register("12345");
                installation.setDeviceToken(regid);
                return null;
            } catch (final IOException ex) {
                return ex;
                // If there is an error, don't just keep trying to
                // register.
                // Require the user to click a button again, or perform
                // exponential back-off.
            }
        }
        @Override
        protected void onPostExecute(final Exception error) {
            if (err != null) {
                Log.e(TAG, "GCM Registration failed.", error);
            } else {
                saveInstallation(installation);
            }
        }
    }.execute(null, null, null);
}

```


Register with LoopBack server

Once you have all Installation properties set, you can register with the LoopBack server. The first run of the application should create a new Installation record, subsequent runs should update this existing record. The LoopBack Android SDK handles the details. Your code just needs to call `save()`.

```
void saveInstallation(final LocalInstallation installation) {
    installation.save(new Model.Callback() {
        @Override
        public void onSuccess() {
            // Installation was saved.
            // You can access the id assigned by the server via
            // installation.getId();
        }
        @Override
        public void onError(final Throwable t) {
            Log.e(TAG, "Cannot save Installation", t);
        }
    });
}
```

Handle received notifications

Android apps handle incoming notifications in the standard way; LoopBack does not require any special changes. For more information, see the section "Receive a message" of Google's [Implementing GCM Client guide](#).

Troubleshooting

When running your app in the Eclipse device emulator, you may encounter the following error:

```
Google Play services, which some of your applications rely on, is not supported by your device. Please
contact the manufacturer for assistance.
```

To resolve this, install a compatible version of the Google APIs platform. See [Prepare your Android project](#) for more information.

Push notification API

- [Installation API](#)
- [Notification API](#)
- [PushManager API](#)
- [Installation REST API](#)
- [Push Notification REST API](#)

Installation API

- [Class: Installation](#)
 - [Installation.findByIdApp](#)
 - [Installation.findByIdUser](#)
 - [Installation.findBySubscriptions](#)

Module: loopback-component-push

undefined

Installation

Installation

Installation Model connects a mobile application to a device, the user and other information for the server side to locate devices using application id/version, user id, device type, and subscriptions. Users may have many devices, or Installations, which can receive Notifications from the Application.

Class Properties

Name	Type	Description
appld	String	The unique identifier for the application this {{Installation}} is registered for.
appVersion	String	Version of the application currently running on the client (optional).

badge	Number	The number of the last displayed badge icon (iOS only).
created	Date	The date the Installation was created.
deviceToken	String	The device token as provided by GCM or APN.
deviceType	String	The device type such as ios.
modified	Date	The date the Installation was last updated.
status	String	Status of the installation, provider dependent, like 'Active'.
subscriptions	Array	The type of notifications the device is subscribed to.
timeZone	String	The Time Zone ID of the device, ex: America/Vancouver.
userId	String	The user id that reported by the device.

Installation.findByApp(deviceType, appld, [appVersion], cb)

Find installations by application id/version

Arguments

Name	Type	Description
deviceType	String	The type of device (android, ios)
appld	String	The application id
[appVersion]	String	The application version
cb	function(Error=, Installation[])	Callback function passed to find() with cb(err, obj[]) signature.

Installation.findBySubscriptions(deviceType, subscriptions, cb)

Find installations by subscriptions

Arguments

Name	Type	Description
deviceType	String	The type of device (android, ios)
subscriptions	String or String[]	Either a comma/space delimited string or array of subscriptions
cb	function(Error=, Installation[])	Callback function passed to find() with cb(err, obj[]) signature.

Installation.findByUser(deviceType, userId, cb)

Find installations by user id

Arguments

Name	Type	Description
deviceType	String	The type of device (android, ios)
userId	String	The user id
cb	function(Error=, Installation[])	Callback function passed to find() with cb(err, obj[]) signature.

Notification API

- [Class: Notification](#)

undefined

Module: loopback-component-push

Notification

Notification

Notification Model

See the official documentation for more details on provider-specific properties.

[Android - GCM](#)[iOS - APN](#)**Class Properties**

Name	Type	Description
alert	Any	The notification's message (iOS only - use <code>message</code> on Android).
badge	Number	The value indicated in the top right corner of the app icon.
category	String	The category for the push notification action (iOS8+ only).
collapseKey	String	An arbitrary string (such as "Updates Available") used to collapse a group of like messages when the device is offline, so only the last message gets sent to the client (Android only).
created	Date	The date that the notification is created.
delayWhileIdle	Boolean	If the device is idle, do not send the message immediately. The server will wait for the device to become active, and then only the last message for each <code>collapse_key</code> value will be sent (Android only).
deviceToken	String	The device token as provided by GCM or APN.
deviceType	String	The device type such as <code>ios</code> . Set to -1 to increment the current value by one (iOS only).
expirationInterval	Number	The expiration interval in seconds. The interval starts at the time when the notification is sent to the push notification provider.
expirationTime	Date	The time that the notification expires.
message	String	The notification's message (Android only - use <code>alert</code> on iOS).
modified	Date	The date that the notification is modified.
scheduledTime	Date	The time that the notification should be sent (not supported yet).
sound	String	The name of a sound file in the application bundle (iOS only).
status	String	Status of the notification (not supported yet).

PushManager API

- [Class: PushManager](#)
 - [PushManager.providers](#)
 - [pushManager.configureProvider](#)
 - [pushManager.configureApplication](#)
 - [pushManager.notifyById](#)
 - [pushManager.notifyByQuery](#)
 - [pushManager.notify](#)
 - [pushManager.notifyMany](#)

Module: loopback-component-push

undefined

PushManager(settings)The PushManager class. See the [options](#) for more information on the settings parameters.**Arguments**

Name	Type	Description
settings	Object	The push settings.

settings

Name	Type	Description

ttlInSeconds	Number	Time-to-live, in seconds.
checkPeriodInSeconds	Number	A number in seconds for the automatic delete check interval

pushManager.configureApplication(appld, deviceType, cb)

Lookup or set up push notification service for the given appld

Arguments

Name	Type	Description
appld	String	The application id
deviceType	String	The type of device (android, ios)
cb	function(Error=, Application)	Callback function called with <code>cb(err, obj)</code> signature.

pushManager.configureProvider(deviceType, pushSettings)

Configure push notification for a given device type. Return null when no provider is registered for the device type.

Arguments

Name	Type	Description
deviceType	String	The type of device (android, ios)
pushSettings	Object	The push settings

Returns

Name	Type	Description
result	Provider or null	A provider from <code>PushManager.providers</code> (<code>GcmProvider</code> , <code>ApnsProvider</code>) matching the <code>deviceType</code> (android, ios)

pushManager.notify(installation, notification, cb)

Push a notification to the given installation. This is a low-level function used by the other higher-level APIs like `notifyById` and `notifyByQuery`.

Arguments

Name	Type	Description
installation	Installation	Installation instance - the recipient.
notification	Notification	The notification to send.
cb	function(Error=)	

pushManager.notifyById(installationId, notification, cb)

Push a notification to the device with the given registration id.

Arguments

Name	Type	Description
installationId	Object	Registration id created by call to <code>Installation.create()</code> .
notification	Notification	The notification to send.
cb	function(Error=)	

pushManager.notifyByQuery(installationQuery, notification, cb)

Push a notification to all installations matching the given query.

Arguments

Name	Type	Description
installationQuery	Object	Installation query, e.g. { appId: 'iCarsAppId', userId: 'jane.smith.id' }
notification	Notification	The notification to send.
cb	function(Error=)	

pushManager.notifyMany(appId, deviceType, deviceTokens, notification, cb)

Push notification to installations for given devices tokens, device type and app.

Arguments

Name	Type	Description
appId	String	application id
deviceType	String	type of device (android, ios)
deviceTokens	String[]	device tokens of recipients.
notification	Notification	The Notification object to send.
cb	function(Error=)	

Installation REST API

All of the endpoints in the table below are inherited from [PersistedModel REST API](#), except for the following:

- [Find installations by app ID](#)
- [Find installations by user ID](#)

This model is provided by the [Push notifications](#) component, loopback-component-push. For more information, see the [LoopBack Push Component API documentation](#).

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
/Installations/byApp	GET		Find installations by app ID	Query parameters: <ul style="list-style-type: none"> • deviceType • appId • appVersion
/Installations/byUser	GET		Find installations by user ID	Query parameters: <ul style="list-style-type: none"> • deviceType • userId
/Installations	POST		Add installation instance and persist to data source. Inherited from generic model API .	JSON object (in request body)
/Installations	GET		Find all instances of installations that match specified filter. Inherited from generic model API .	One or more filters in query parameters: <ul style="list-style-type: none"> • where • include • order • limit • skip / offset • fields
/Installations	PUT		Update / insert installation instance and persist to data source. Inherited from generic model API .	JSON object (in request body)

/Installations/ <i>id</i>	GET		Find installation by ID : Return data for the specified instance ID. Inherited from generic model API .	<i>id</i> , the installation ID (in URI path)
/Installations/ <i>id</i>	PUT		Update installation attributes for specified installation ID and persist. Inherited from generic model API .	Query parameters: <ul style="list-style-type: none"> • <i>data</i> An object containing property name/value pairs • <i>id</i> - The installation ID
/Installations/ <i>id</i>	DELETE		Delete installation with specified instance ID. Inherited from generic model API .	<i>id</i> , installation ID (in URI path)
/Installations/count	GET		Return number of installation instances that match specified where clause. Inherited from generic model API .	Query parameter: "where" filter.
/Installations/ <i>id</i> /exists	GET		Check instance existence : Return true if specified user ID exists. Inherited from generic model API .	URI path: <i>id</i> installation ID
/Installations/findOne	GET		Find first installation instance that matches specified filter. Inherited from generic model API .	Same as Find matching instances .

Find installations by app ID

Return JSON array of installations of specified app ID that also match the additional specified arguments (if any).

```
GET /Installations/byApp
```

Arguments

All arguments are in query string:

- *deviceType*
- *appId*
- *appVersion*

Example

Request:

```
curl -X GET
http://localhost:3000/Installation/byApp?appId=KrushedKandy&deviceType=ios
```

Response:

```
[
  {
    "id": "1",
    "appId": "KrushedKandy",
    "userId": "raymond",
    "deviceType": "ios",
    "deviceToken": "756244503c9f95b49d7ff82120dc193cae3a7cb56f60c2ef2a19241e8f33305",
    "subscriptions": [],
    "created": "2014-01-09T23:18:57.194Z",
    "modified": "2014-01-09T23:18:57.194Z"
  },
  ...
]
```

Errors

Find installations by user ID

Return JSON array of installations by specified user ID that also match the additional specified argument (if provided).

```
GET /Installations/byUser
```

Arguments

Arguments are in query string:

- deviceType
- userId

Example

Request:

```
curl -X GET
http://localhost:3000/Installations/byUser?userId=raymond
```

Response:

```
[
  {
    "id": "1",
    "appId": "MyLoopBackApp",
    "userId": "raymond",
    "deviceType": "ios",
    "deviceToken": "756244503c9f95b49d7ff82120dc193cae3a7cb56f60c2ef2a19241e8f33305",
    "subscriptions": [],
    "created": "2014-01-09T23:18:57.194Z",
    "modified": "2014-01-09T23:18:57.194Z"
  }
]
```

Errors

Push Notification REST API

All of the endpoints in the table below are inherited from [PersistedModel](#), except for [Send push notification](#).

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
/push	POST	Allow / Deny	Send a push notification by installation query	Query parameters: deviceQuery Request body: notification

Send push notification

Send a push notification by installation query.

POST /push

Arguments

- deviceQuery - Object; query parameter.
- notification - Object; request body.

Example

Request:

```
curl -X -g POST -H "Content-Type:application/json"
-d '{"badge" : 5, "sound": "ping.aiff", "alert": "Hello", "messageFrom": "Ray"}'
http://localhost:3000/push?deviceQuery[userId]=1
```

Response code: 200

Response body:

```
{}
```

Errors**Tutorial: Push notifications**

This four-part tutorial explains how to create a mobile application on Amazon EC2 that can send push notification to iOS and Android client applications:

- [Part one](#) shows how to use the `slc` command-line tool to create a LoopBack application running on Amazon EC2 to send push notifications.
- [Part two](#) explains how to setup and create an Android app to receive push notifications.
- [Part three](#) explains how to setup and create an iOS app to receive push notifications.
- [Part four](#) explains how to use LoopBack's swagger REST api and send/receive push notifications on your Android and iOS devices.

Related articles:**See also:**

- [Android SDK API docs](#)

You can use the StrongLoop AMI as a starting point for building a backend node application using the `slc` command line tool. Using this app, mobile developers can dynamically manage 'mobile object' schemas directly from the LoopBack SDK as they are building their mobile app. The pre-built StrongLoop AMI makes it easy for Node developers who've chosen Amazon as their infrastructure provider to quickly get up and running.

Now dive in and start on [part one](#)!

Tutorial: push notifications - LoopBack app

This is part one of a four-part tutorial on setting up a mobile application on Amazon EC2 that can send push notification to iOS and Android client applications. This article (part one) shows how to use the `slc` command-line tool to create a LoopBack application running on Amazon EC2 to send push notifications.

- [Overview](#)
- [Prerequisites](#)
- [Set up mobile backend server on Amazon](#)
 - [Launch the instance](#)
 - [Create the application](#)
- [Next steps](#)

Overview

Push notifications enable server applications (known as *providers* in push parlance) to send information to mobile apps even when the app isn't in use. The device displays the information using a "badge," alert, or pop up message. A push notification uses the service provided by the device's operating system:

- **iOS** - Apple Push Notification service (APNS)
- **Android** - Google Cloud Messaging (GCM)

For an overview of the LoopBack Push Notification component, see [Push notifications](#).

Prerequisites

Before starting this tutorial:

- Make sure you have an [Amazon AWS](#) account.
- To set up push notifications for an iOS app:
 - [Download the LoopBack iOS SDK](#)
 - [Install Xcode](#)
- To set up push notifications for an Android app:
 - [Download the LoopBack Android SDK](#)
 - [Install Eclipse development tools \(ADT\)](#)

Set up mobile backend server on Amazon

Launch the instance

1. To follow along you will need to have an [Amazon AWS](#) account.
2. To find the StrongLoop AMI, simply log into the [AWS Console](#) and select "EC2." Browse images by selecting "AMIs" under the "Images" drop down. Make sure the filtering shows "Public Images", "All Images" and "All Platforms". From here you can simply search "StrongLoop" and select the latest version. Current - StrongLoop-slc v2.5.2 (node v0.10.26)

Image: ami-bbf88e8b

AMI ID	AMI Name	Source
ami-bbf88e8b	StrongLoop-slc v2.5.2 (node v0.10.26)	257586017729/StrongLc
ami-14d5bf24	StrongLoop-slc v2.5.0 (node v0.10.26)	257586017729/...
ami-14422724	StrongLoop-node-v0.10.22	257586017729/...
ami-1649d626	StrongLoop Suite 1.0	257586017729/...
ami-5c9dfe6c	StrongLoop-node-v0.10.22	257586017729/...
ami-d479e1e4	StrongLoop Suite 1.1.0	257586017729/...

Details

AMI ID	ami-bbf88e8b	AMI Name	StrongLoop-slc v2.5.2 (node v0.10.26)
Owner	257586017729	Source	257586017729/StrongLc
Status	pending	State Reason	-
Platform	Other Linux	Architecture	x86_64

3. Launch a new instance from the console using this AMI. Once the instance is up and running, you can remote ssh into your newly created server instance using the same ec2-keypair and the machine instance ID.

```
$ ssh -i ec2-keypair ec2-user@ec2-54-222-22-59.us-west-1.compute.amazonaws.com
```

Create the application



StrongLoop Controller (slc) and MongoDB are pre-installed in the StrongLoop AML. To run MongoDB, use `~/mongodb/bin/mongod` &. You may need to use `sudo`.

1. Use the `slc loopback` command to create a LoopBack application:

```
$ slc loopback
[?] Enter a directory name where to create the project: push
[?] What's the name of your application? push
```

2. Add the `loopback-component-push` and `loopback-connector-mongodb` modules as dependencies in `package.json`:

```
"dependencies": {
  "loopback": "~1.7.0",
  "loopback-component-push": "~1.2.0",
  "loopback-connector-mongodb": "~1.2.0"
},
```

3. Install dependencies:

```
npm install
```

4. Add the mongo connection string in `/server/datasources.json`:

```
"db": {
  "defaultForType": "db",
  "connector": "mongodb",
  "url": "mongodb://localhost/demo"
},
```



Try it with an example

To try out an example, you can replace the `server.js` file in your application with [this](#). You will also need the `model-config.js` file to save your configurations. Alternatively, to enable push notifications for your own application using the `loopback-component-push` module, follow the steps below (5-8)

5. Create a push model
 - a. To send push notifications, you must create a push model. The code below illustrates how to do this with a database as the data source. The database is used to load and store the corresponding application/user/installation models.

```
var loopback = require('loopback');
var app = loopback();
...
var Notification = app.models.notification;
var Application = app.models.application;
var PushModel = app.models.push;
```

6. Register a mobile(client) application

- a. The mobile application needs to register with LoopBack so it can have an identity for the application and corresponding settings for push services. Use the Application model's `register()` function for sign-up.

For information on getting API keys, see:

- [Get your Google Cloud Messaging credentials](#) for Android.
- [Set up iOS clients](#) for iOS.

```

Application.register('put your developer id here',
  'put your unique application name here',
  {
    description: 'LoopBack Push Notification Demo Application',
    pushSettings: {
      apns: {
        certData: readCredentialsFile('apns_cert_dev.pem'),
        keyData: readCredentialsFile('apns_key_dev.pem'),

        pushOptions: {
        },
        feedbackOptions: {
          batchFeedback: true,
          interval: 300
        }
      },
      gcm: {
        serverApiKey: 'your GCM server API Key'
      }
    }
  },
  function(err, app) {
    if (err) return cb(err);
    return cb(null, app);
  }
);

function readCredentialsFile(name) {
  return fs.readFileSync(
    path.resolve(__dirname, 'credentials', name),
    'UTF-8'
  );
}

```

7. Register a mobile device

- a. The mobile device also needs to register itself with the backend using the Installation model and APIs. To register a device from the server side, call the `Installation.create()` function, as shown in the following example:

```

Installation.create({
  appId: 'MyLoopBackApp',
  userId: 'raymond',
  deviceToken:
    '756244503c9f95b49d7ff82120dc193cale3a7cb56f60c2ef2a19241e8f33305',
  deviceType: 'ios',
  created: new Date(),
  modified: new Date(),
  status: 'Active'
}, function (err, result) {
  console.log('Registration record is created: ', result);
});

```

Most likely, the mobile application registers the device with LoopBack using REST APIs or SDKs from the client side, for example

```
POST http://localhost:3010/api/installations
{
  "appId": "MyLoopBackApp",
  "userId": "raymond",
  "deviceToken":
"756244503c9f95b49d7ff82120dc193cae3a7cb56f60c2ef2a19241e8f33305",
  "deviceType": "ios"
}
```

8. Send push notifications

a. Send out the push notification

LoopBack provides two Node.js methods to select devices and send notifications to them:

- `notifyById()`: Select a device by registration ID and send a notification to it.
- `notifyByQuery()`: Get a list of devices using a query (same as the **where** property for `Installation.find()`) and send a notification to all of them.

For example, the code below creates a custom endpoint to send out a dummy notification for the selected device:

```
var badge = 1; app.post('/notify/:id', function (req, res, next) {
  var note = new Notification({
    expirationInterval: 3600, // Expires 1 hour from now.
    badge: badge++,
    sound: 'ping.aiff',
    alert: '\uD83D\uDCE7 \u2709 ' + 'Hello',
    messageFrom: 'Ray'
  });

  PushModel.notifyById(req.params.id, note, function(err) {
    if (err) {
      // let the default error handling middleware
      // report the error in an appropriate way
      return next(err);
    }
    console.log('pushing notification to %j', req.params.id);
    res.send(200, 'OK');
  });
});
```

To select a list of devices by query, use the `PushModel.notifyByQuery()`, for example:

```
PushModel.notifyByQuery({userId: {inq: selectedUserIds}}, note,
function(err) { console.log('pushing notification to %j',
selectedUserIds);
});
```

Next steps

- To setup and create an Android app to receive push notifications go to [Part two](#)
- To setup and create an iOS app to receive push notifications go to [Part three](#)
- To use LoopBack's swagger REST API and send/receive push notifications on your Android and iOS devices go to [Part four](#)

Tutorial: push notifications - Android client

This is the second of a four-part tutorial on setting up a mobile backend as a service on Amazon and setting up iOS and Android client applications to enable push notification. If you want to setup Push Notifications for an iOS app refer to Part 3.

- [Overview](#)
- [Prerequisites](#)
 - [Configure Android Development Tools](#)
 - [Get your Google Cloud Messaging credentials](#)
- [Configure GCM push settings in your server application](#)
- [Install and run LoopBack Push Notification app](#)
- [Prepare your own Android project](#)
- [Check for Google Play Services APK](#)
- [Create LocalInstallation](#)
- [Register with GCM if needed](#)
- [Register with LoopBack server](#)
- [Handle received notifications](#)
- [Troubleshooting](#)

Overview

This article provides information on creating Android apps that can get push notifications from a LoopBack application.

To enable an Android app to receive LoopBack push notifications:

1. Setup your android client app to use Google Play Services.
2. On app startup, register with GCM servers to obtain a device registration ID (device token) and register the device with the LoopBack server application.
3. Configure your LoopBack application to receive incoming messages from GCM.
4. Process the notifications received.

Prerequisites

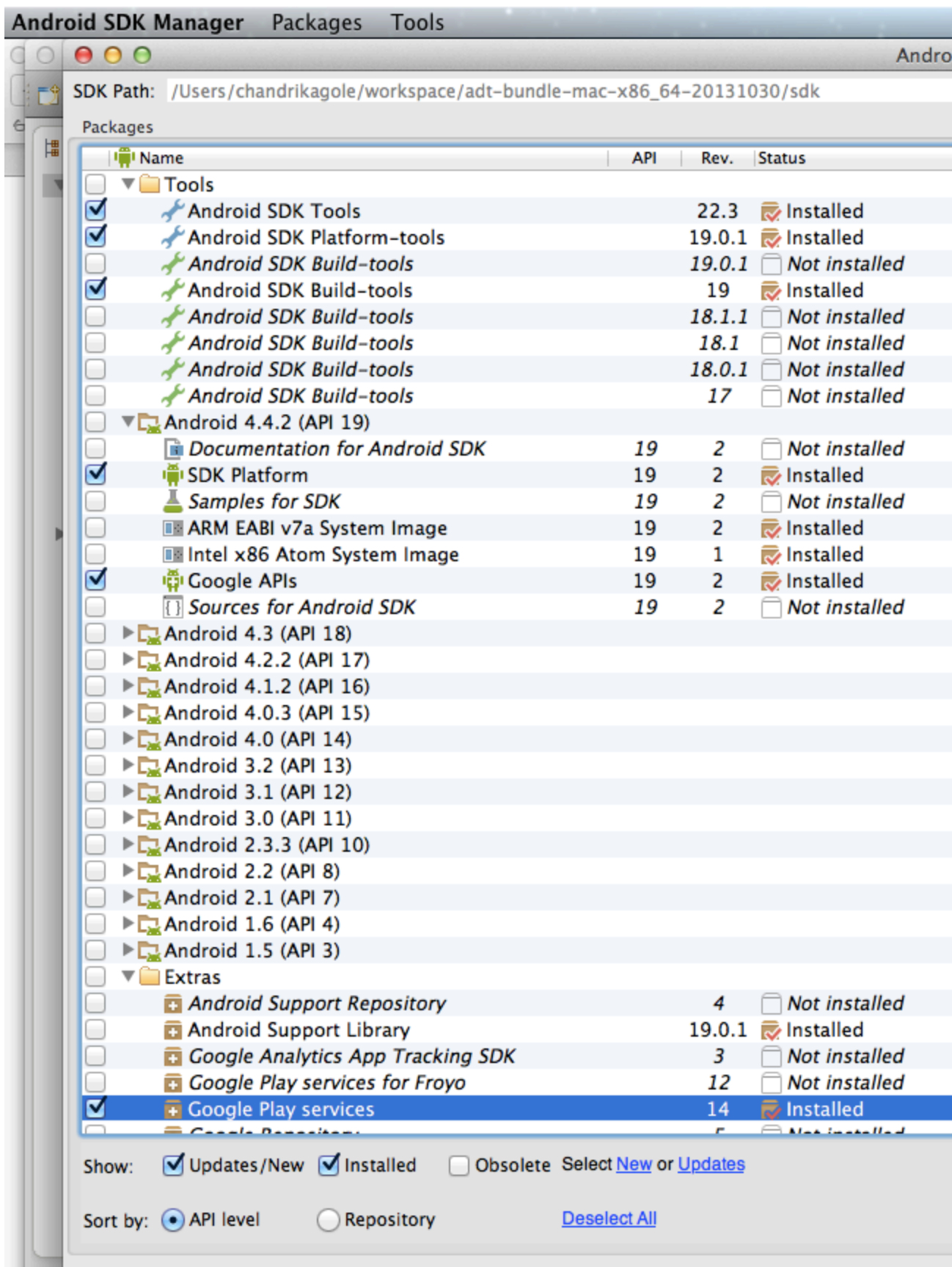
Before you start developing your application make sure you've performed all the prerequisite steps outlined in this section.

- [Download the LoopBack Android SDK](#)
- [Install Eclipse development tools \(ADT\)](#)

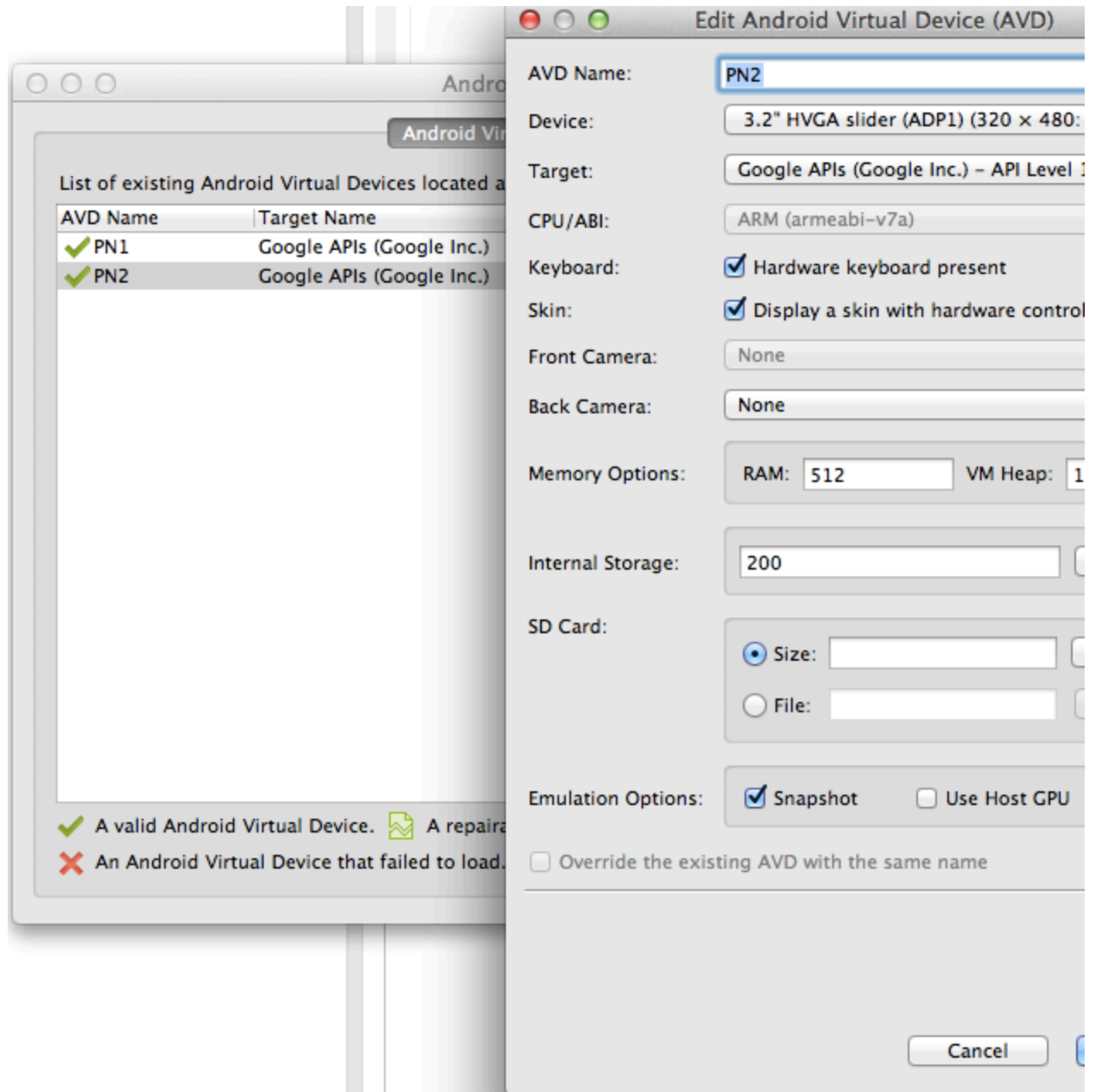
Configure Android Development Tools

Now configure Eclipse ADT as follows:

1. Open Eclipse from the downloaded ADT bundle.
2. In ADT, choose **Window > Android SDK Manager**.
3. Install the following if they are not already installed:
 - Tools:
 - Android SDK Platform-tools 18 or newer
 - Android SDK Build-tools 18 or newer
 - Android 4.3 (API 18):
 - SDK Platform.
 - Google APIs
 - Extras:
 - Google Play Services
 - Intel x86 Emulator Accelerator (HAXM)



- Before you start, make sure you have set up at least one Android virtual device: Choose **Window > Android Virtual Device Manager**.
- Configure the target virtual device as shown in the screenshot below. See [AVD Manager](#) for more information.



If you are using the virtual device suggested above, you must also install the ARM EABI v7a System Image SDK.

Get your Google Cloud Messaging credentials

To send push notifications to your Android app, you need to setup a Google API project and enable the Google Cloud Messaging (GCM) service.

- [Open the Android Developer's Guide](#)

Follow the instructions to get your GCM credentials:

1. Follow steps to create a Google API project and enable the GCM service.
2. Create an Android API key
 - a. In the sidebar on the left, select **APIs & auth > Credentials**.
 - b. Click **Create new key**.
 - c. Select **Android key**.
 - d. Enter the SHA-1 fingerprint followed by the package name, for example
 45:B5:E4:6F:36:AD:0A:98:94:B4:02:66:2B:12:17:F2:56:26:A0:E0;com.example
 NOTE: Leave the package name as "com.example" for the time being.

3. You also have to create a new server API key that will be used by the LoopBack server:
 - a. Click **Create new key**.
 - b. Select **Server key**.
 - c. Leave the list of allowed IP addresses empty for now.
 - d. Click **Create**.
 - e. Copy down the API key. Later you will use this when you configure the LoopBack server application.

Configure GCM push settings in your server application

Add the following key and value to the push settings of your application in the config.js file -

```
{
  gcm: {
    serverApiKey: "server-api-key"
  }
}
```

Replace `server-api-key` with the API key you obtained in the section Get your Google Cloud Messaging credentials.



If you want to try a sample client application follow steps in "Install and run LoopBack Push Notification app" OR if you want to enable push notifications for your own android application using the LoopBack SDK follow steps in "Prepare your own Android project"

Install and run LoopBack Push Notification app

If you want to use the sample Android client app, download the [Push Notification Example Android app](#) . Then follow these steps to run the app:

1. Open ADT Eclipse.
2. Import the push notification application to your workspace:
 - a. Choose **File > Import**.
 - b. Choose **Android > Existing Android Code into Workspace**.
 - c. Click **Next**.
 - d. Browse to the example Android app you just downloaded.
 - e. Click **Finish**.

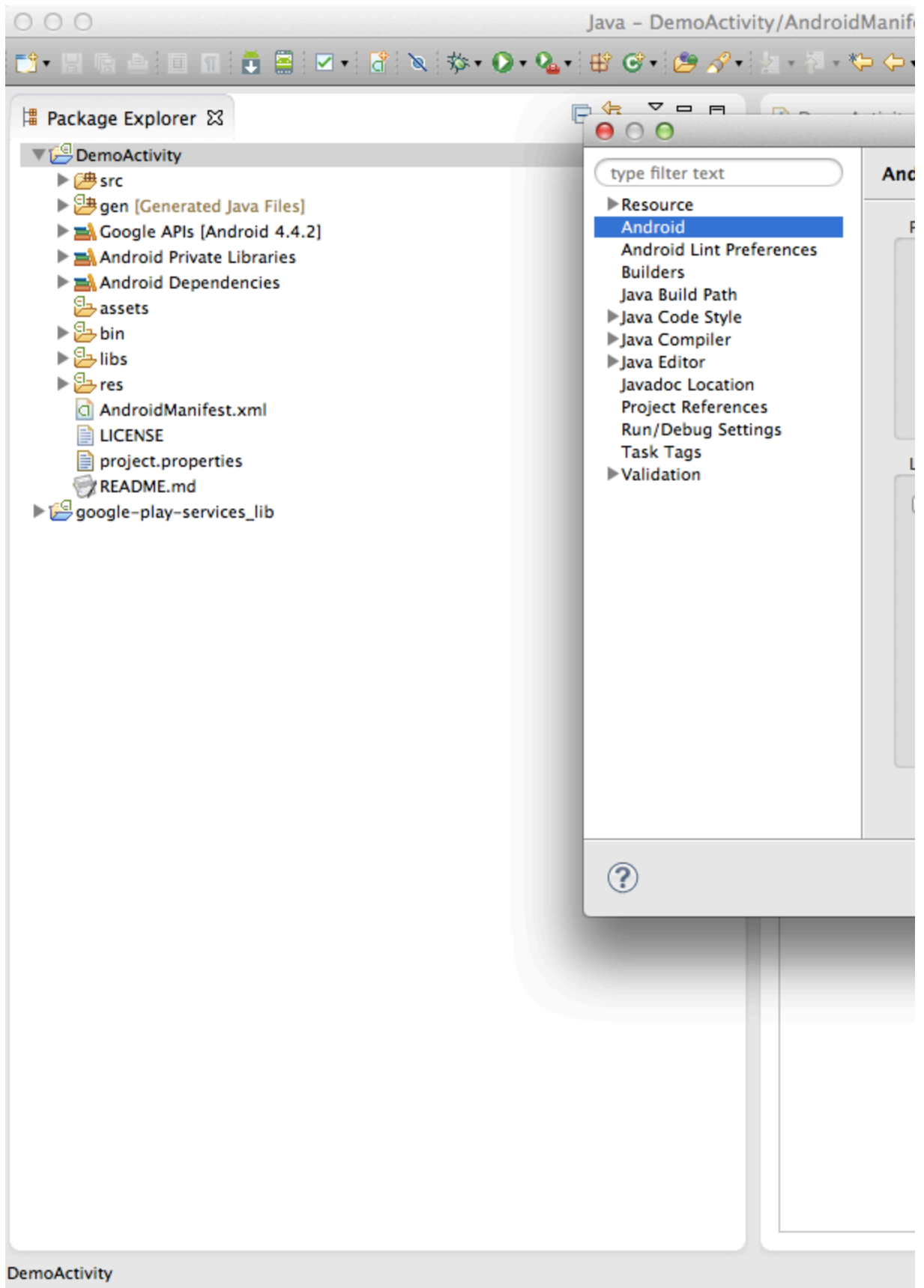


ADT does not take long to import the guide app. Don't be misguided by the progress bar at the bottom of the IDE window: it indicates memory use, not loading status.

3. Import Google Play Services library project into your workspace. The project is located inside the directory where you have installed the Android SDK.
 - a. Choose **File > Import**.
 - b. Choose **Android > Existing Android Code into Workspace**.
 - c. Click **Next**.
 - d. Browse to the `<android-sdk>/extras/google/google_play_services/libproject/google-play-services_lib` directory.
 - e. Check **Copy projects into workspace**
 - f. Click **Finish**.

See [Google Play Services SDK](#) for more details.

4. Add the imported google-play-services_lib as an Android build dependency of the push notification application.
 - a. In the Package Explorer frame in Eclipse, select the push notification application.
 - b. Choose **File > Properties**.
 - c. Select **Android**.
 - d. In the Library frame, click on **Add...** and select `google-play-services_lib`.
 - e. Also under Project Build Target, set the target as Google APIs.



5. Edit `src/com/google/android/gcm/demo/app/DemoActivity.java`.
 - Set `SENDER_ID` to the project number from the Google Developers Console you created earlier in Get your Google Cloud Messaging credentials.
6. Go back to the <https://cloud.google.com/console/project> and edit the Android Key to reflect your unique application ID. Set the value of **A**

ndroid applications to something like this:

Android applications	XX:XX:XX:XX:XX:XX:XX:XX:XX:XX
	LOOPBACK_APP_ID
	X:XX:XX:XX:XX:XX:XX:XX:XX:XX;com.google.android.gcm.demo.app.DemoApplication

- Set the appName in the server application's `config.js` to "**com.google.android.gcm.demo.app.DemoActivity**".
- Edit `src/com/google/android/gcm/demo/app/DemoApplication.java`
 - Set adaptor to our server ip. In my case it is

DemoApplication.java

```
adapter = new RestAdapter(
    getApplicationContext(),

    "http://ec2-54-184-36-164.us-west-2.compute.amazonaws.com:3000/api/");
}
```

- Click the green **Run** button in the toolbar to run the application. Run it as an Android application. You will be prompted to select the target on which to run the application. Select the AVD you created earlier.



It may take several minutes to launch your application and the Android virtual device the first time.



Due to a [known issue with Google Play Services](#), you must download and import an older version of Google Play services.

- Download https://dl-ssl.google.com/android/repository/google_play_services_3225130_r10.zip
- Extract the zip file.
- In Eclipse ADT, right-click on your project and choose **Import...**
- Choose **Existing Android Code into Workspace** then click Next.
- Click **Browse...**
- Browse to the `google-play-services/libproject/google-play-services_lib/` directory created when you extracted the zip file and select it in the dialog box.
- Click **Finish**.

You must also update `AndroidManifest.xml` as follows:

- In Eclipse ADT, browse to `DemoActivity/AndroidManifest.xml`.
- Change the line


```
<meta-data android:name="com.google.android.gms.version" android:value="@integer/google_play_services_version"/>
```

 to


```
<meta-data android:name="com.google.android.gms.version" android:value="4030500"/>
```
- Save the file.

Prepare your own Android project

Follow the instructions in [Android SDK documentation](#) to add LoopBack Android SDK to your Android project.

Follow the instructions in Google's [Implementing GCM Client guide](#) for setting up Google Play Services in your project.



To use push notifications, you must install a compatible version of the Google APIs platform. To test your app on the emulator, expand the directory for Android 4.2.2 (API 17) or a higher version, select **Google APIs**, and install it. Then create a new AVD with Google APIs as the platform target. You must install the package from the SDK manager. For more information, see [Set Up Google Play Services](#).

Check for Google Play Services APK

Applications that rely on the Google Play Services SDK should always check the device for a compatible Google Play services APK before using Google Cloud Messaging.

For example, the following code checks the device for Google Play Services APK by calling `checkPlayServices()` if this method returns true, it proceeds with GCM registration. The `checkPlayServices()` method checks whether the device has the Google Play Services APK. If it doesn't, it displays a dialog that allows users to download the APK from the Google Play Store or enables it in the device's system settings.

```
@Override
public void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    if (checkPlayServices()) {
        updateRegistration();
    } else {
        Log.i(TAG, "No valid Google Play Services APK found.");
    }
}

private boolean checkPlayServices() {
    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
    if (resultCode != ConnectionResult.SUCCESS) {
        if (GooglePlayServicesUtil.isUserRecoverableError(resultCode)) {
            GooglePlayServicesUtil.getErrorDialog(resultCode, this,
                PLAY_SERVICES_RESOLUTION_REQUEST).show();
        } else {
            Log.i(TAG, "This device is not supported.");
            finish();
        }
        return false;
    }
    return true;
}
```

Create LocalInstallation

Once you have ensured the device provides Google Play Services, the app can register with GCM and LoopBack (for example, by calling a method such as `updateRegistration()` as shown below). Rather than register with GCM every time the app starts, simply store and retrieve the registration ID (device token). The `LocalInstallation` class in the LoopBack SDK handles these details for you.

The example `updateRegistration()` method does the following:

- Lines 3 - 4: get a reference to the shared `RestAdapter` instance.
- Line 5: Create an instance of `LocalInstallation`.
- Line 13: Subscribe to topics.
- Lines 15-19: Check if there is a valid GCM registration ID. If so, then save the installation to the server; if not, get one from GCM and then save the installation.

```

private void updateRegistration() {

    final DemoApplication app = (DemoApplication) getApplication();
    final RestAdapter adapter = app.getLoopBackAdapter();
    final LocalInstallation installation = new LocalInstallation(context, adapter);

    // Substitute the real ID of the LoopBack application as created by the server
    installation.setAppId("loopback-app-id");

    // Substitute a real ID of the user logged in to the application
    installation.setUserId("loopback-android");

    installation.setSubscriptions(new String[] { "all" });

    if (installation.getDeviceToken() != null) {
        saveInstallation(installation);
    } else {
        registerInBackground(installation);
    }
}

```

Register with GCM if needed

In the following code, the application obtains a new registration ID from GCM. Because the `register()` method is blocking, you must call it on a background thread.

```

private void registerInBackground(final LocalInstallation installation) {
    new AsyncTask<Void, Void, Exception>() {
        @Override
        protected Exception doInBackground(final Void... params) {
            try {
                GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);
                // substitute 12345 with the real Google API Project number
                final String regid = gcm.register("12345");
                installation.setDeviceToken(regid);
                return null;
            } catch (final IOException ex) {
                return ex;
                // If there is an error, don't just keep trying to
                // register.
                // Require the user to click a button again, or perform
                // exponential back-off.
            }
        }
        @Override
        protected void onPostExecute(final Exception error) {
            if (error != null) {
                Log.e(TAG, "GCM Registration failed.", error);
            } else {
                saveInstallation(installation);
            }
        }
    }.execute(null, null, null);
}

```

Register with LoopBack server

Once you have all Installation properties set, you can register with the LoopBack server. The first run of the application should create a new Installation record, subsequent runs should update this existing record. The LoopBack Android SDK handles the details. Your code just needs to call `save()`.

```
void saveInstallation(final LocalInstallation installation) {
    installation.save(new Model.Callback() {
        @Override
        public void onSuccess() {
            // Installation was saved.
            // You can access the id assigned by the server via
            // installation.getId();
        }
        @Override
        public void onError(final Throwable t) {
            Log.e(TAG, "Cannot save Installation", t);
        }
    });
}
```

Handle received notifications

Android apps handle incoming notifications in the standard way; LoopBack does not require any special changes. For more information, see the section "Receive a message" of Google's [Implementing GCM Client guide](#).

Troubleshooting

When running your app in the Eclipse device emulator, you may encounter the following error:

```
Google Play services, which some of your applications rely on, is not supported by your device. Please
contact the manufacturer for assistance.
```

To resolve this, install a compatible version of the Google APIs platform.

Tutorial: push notifications - iOS client

This is the third of a four-part tutorial on setting up a mobile backend as a service on Amazon and setting up iOS and Android client applications to enable push notification. See [Tutorial: push notifications - LoopBack app](#) for information on setting up the server application.

- [Overview](#)
- [Prerequisites](#)
- [Configure APN push settings in your server application](#)
- [Install and run LoopBack Push Notification app](#)
- [Add LoopBack iOS SDK as a framework to your own iOS Project](#)
- [Initialize LBRESTAdapter](#)
- [Register the device](#)
- [Handle received notifications](#)

Overview

This article provides information on creating iOS apps that can get push notifications from a LoopBack application. See [Push notifications](#) for information on creating the corresponding LoopBack server application.

The basic steps to set up push notifications for iOS clients are:

1. Provision an application with Apple and configure it to enable push notifications.
2. Provide a hook to receive the device token when the application launches and register it with the LoopBack server using the `LBInstallation` class.
3. Provide code to receive notifications, under three different application modes: foreground, background, and offline.
4. Process notifications.

For general information on the Apple push notifications, see [Apple iOS Local and Push Notification Programming Guide](#). For additional useful information, see [Delivering iOS Push Notifications with Node.js](#).

Prerequisites

Before you start developing your application make sure you've performed all the prerequisite steps outlined in this section.

- [Download the LoopBack iOS SDK](#)
- [Install Xcode](#)

Configure APN push settings in your server application

Set up messaging credentials for iOS apps

If you have not already done so, [create your APNS certificates](#) for iOS apps. After following the instructions, you will have APNS certificates on your system. Then edit `config.js` in your backend application and look for these lines:

```
exports.apnsCertData = readCredentialsFile('apns_cert_dev.pem');
exports.apnsKeyData = readCredentialsFile('apns_key_dev.pem');
```

Replace the file names with the names of the files containing your APNS certificates. By default, `readCredentialsFile()` looks in the `/credentials` sub-directory for your APNS certificates. You can create a credentials directory and copy your pem files into that directory.



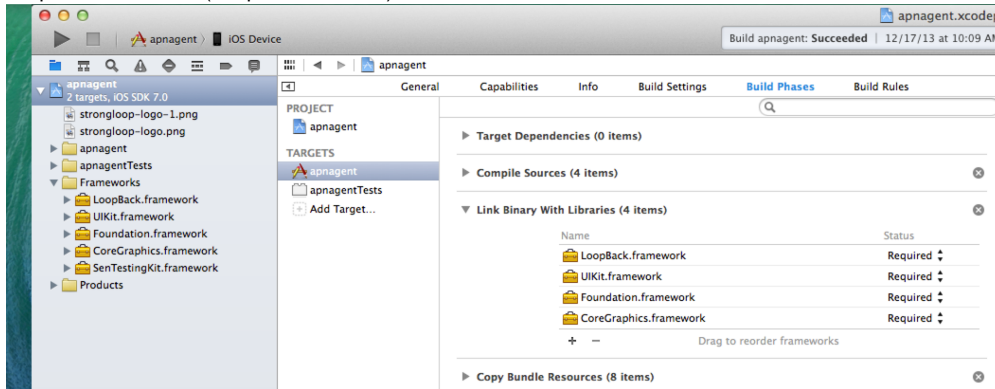
To try a sample client application, follow [Install and run LoopBack Push Notification app](#).

To enable push notifications for your own Android app see [Add LoopBack iOS SDK as a framework to your own iOS Project](#).

Install and run LoopBack Push Notification app

If you want to use the sample iOS client app, download the [Push Notification Example iOS app](#). Then follow these steps to run the app

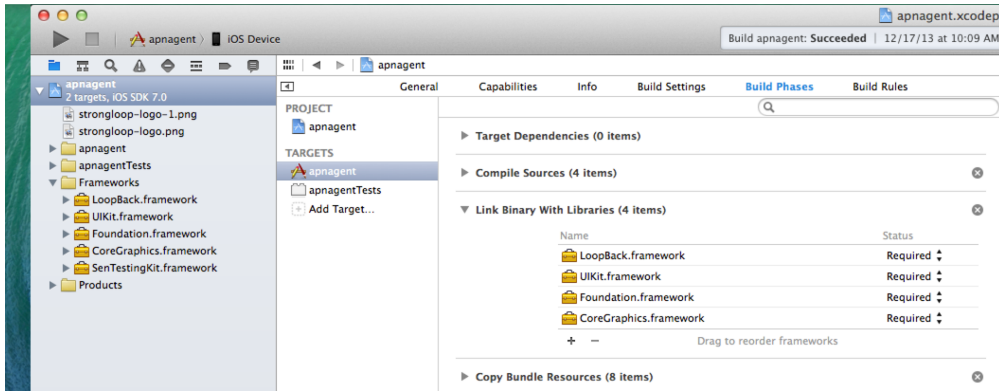
1. Open the `apnagent.xcodeproj` in Xcode, select targets, under build phases unfold **Link Binary with Libraries**, and click on '+' to add LoopBack framework(LoopBack iOS sdk).



2. Edit `Settings.plist` and update the `RootPath` to your instance ip. In my case it is - <http://ec2-54-184-36-164.us-west-2.compute.amazonaws.com:3000/api>

Add LoopBack iOS SDK as a framework to your own iOS Project

Open your XCode project, select targets, under build phases unfold **Link Binary with Libraries**, and click on '+' to add LoopBack framework.



The LoopBack iOS SDK provides two classes to simplify push notification programming:

- [LBInstallation](#) - enables the iOS application to register mobile devices with LoopBack.
- [LBPushNotification](#) - provides a set of helper methods to handle common tasks for push notifications.

Initialize LBRESTAdapter

The following code instantiates the shared `LBRESTAdapter`. In most circumstances, you do this only once; putting the reference in a singleton is recommended for the sake of simplicity. However, some applications will need to talk to more than one server; in this case, create as many adapters as you need.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.settings = [self loadSettings];
    self.adapter = [LBRESTAdapter adapterWithURL:[NSURL
    URLWithString:self.settings[@"RootPath"]]];

    // Reference to Push notifs List VC
    self.pnListVC = (NotificationListVC *)[[UINavigationController
    *)self.window.rootViewController viewControllers]
    objectAtIndex:0];

    LBPushNotification* notification = [LBPushNotification application:application
    didFinishLaunchingWithOptions:launchOptions];

    // Handle APN on Terminated state, app launched because of APN
    if (notification) {
        NSLog(@"Payload from notification: %@", notification.userInfo);
        [self.pnListVC addPushNotification:notification];
    }

    return YES;
}
```

Register the device

```

- (void)application:(UIApplication*)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
{
    __unsafe_unretained typeof(self) weakSelf = self;

    // Register the device token with the LoopBack push notification service
    [LBPushNotification application:application
didRegisterForRemoteNotificationsWithDeviceToken:deviceToken
    adapter:self.adapter
    userId:@"anonymous"
    subscriptions:@[@"all"]
    success:^(id model) {
        LBInstallation *device = (LBInstallation *)model;
        weakSelf.registrationId = device._id;
    }
    failure:^(NSError *err) {
        NSLog(@"Failed to register device, error: %@", err);
    }

];

...
}

- (void)application:(UIApplication*)application
didFailToRegisterForRemoteNotificationsWithError:(NSError*)error {
    // Handle errors if it fails to receive the device token
    [LBPushNotification application:application
didFailToRegisterForRemoteNotificationsWithError:error];
}

```

Handle received notifications

```

- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo {
    // Receive push notifications
    LBPushNotification* notification = [LBPushNotification application:application
    didReceiveRemoteNotification:userInfo];

    [self.pnListVC addPushNotification:notification];
}

```

Tutorial: push notifications - putting it all together

This is the final part of a four-part tutorial on setting up a mobile backend as a service on Amazon and setting up iOS and Android client applications to enable push notifications.

Run the backend sever

1. Make sure you have followed the steps in part 1 to create your back-end application and you have setup the credentials and API keys for your iOS and Android applications.
2. Run the application with the command `node app.js`. If the app runs successfully, you will see this:


```
[ec2-user@ip-10-252-201-3 push]$ node app.js
connect.multipart() will be removed in connect 3.0
visit https://github.com/senchalabs/connect/wiki/Connect-3.0 for alternatives
connect.limit() will be removed in connect 3.0
Browse your REST API at http://0.0.0.0:3000/explorer
LoopBack server listening @ http://0.0.0.0:3000/
Registering a new Application...
Application id: "loopback-push-notification-app"
```

3. You can access the LoopBack API Explorer at http://<server_ip>:3000/explorer:

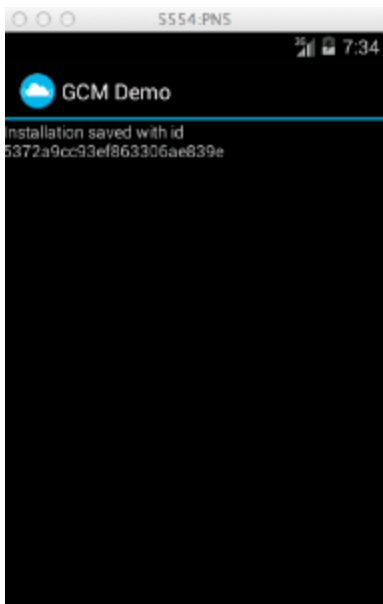
StrongLoop API Explorer Explore

/users	Show/Hide	List Operations	Expand Operations	Raw
/accessTokens	Show/Hide	List Operations	Expand Operations	Raw
/applications	Show/Hide	List Operations	Expand Operations	Raw
/push	Show/Hide	List Operations	Expand Operations	Raw
/installations	Show/Hide	List Operations	Expand Operations	Raw
/notifications	Show/Hide	List Operations	Expand Operations	Raw

[BASE URL: <http://localhost:3010/api>]

Receive push notifications for Android client application

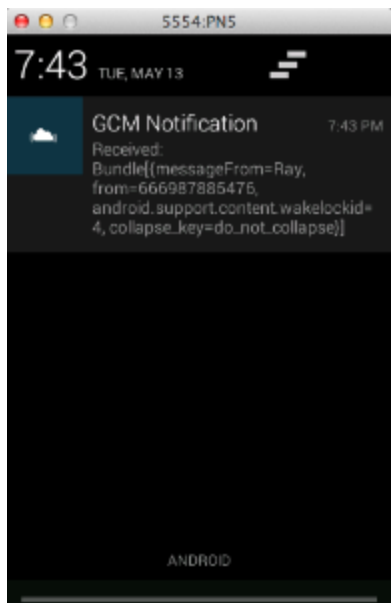
1. Make sure you have set the rest adapter in `src/com/google/android/gcm/demo/app/DemoApplication.java` to your server's IP address.
2. Click the green **Run** button in the toolbar to run the application. Run it as an Android application. You will be prompted to select the target on which to run the application. Select the AVD you created earlier in Part 2
3. Register your application with Loopback:
 - a. Since you have set **gcmServerApiKey** and **appName** in `config.js`, your application will be registered with LoopBack when you run the server.
 - b. Verify this using this GET request: `/api/applications`.
 - c. You can register an application using this POST request: `/api/applications`.
4. Register your device:
 - a. If the AVD launches and the app gets installed successfully, the device will be registered with the back-end and you can verify the installation with the GET request `/api/installations`.
 - b. You can see the GCM application on your Android emulator. You will need the "id" from the to send the push notification to the device. You can also get the id from the `/api/installations`



5. To send a push notification (for example):

```
curl -X POST
http://ec2-54-184-36-164.us-west-2.compute.amazonaws.com:3000/notify/<installatio
n id>
OK
```

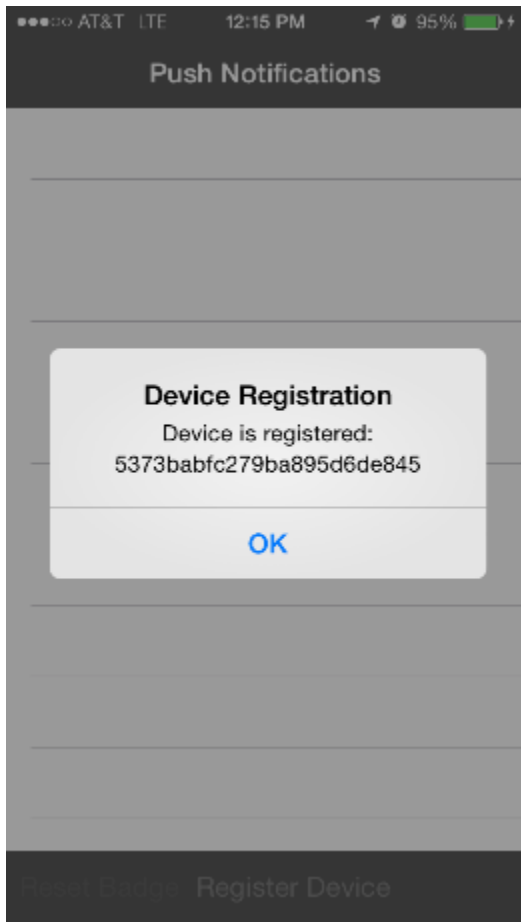
6. The notification received on your emulator will look like this:



Receive push notifications for iOS client application

1. Make sure you have set the RootPath in apnagent/Settings.plist set to your server's ip address.
2. Your phone should be connected to your laptop.
3. Register your application with LoopBack:
 - a. Register your application using this POST request: `/api/applications`.
 - b. Verify the app is registered using this GET request: `/api/applications`.
 - c. You need the "id" from the response to identify the client application. In your client app, edit Settings.plist and set Appld to the id from the previous step.
4. Run your XCode application and remember to connect your registered device. Select your device while running the app.
5. Register your device -

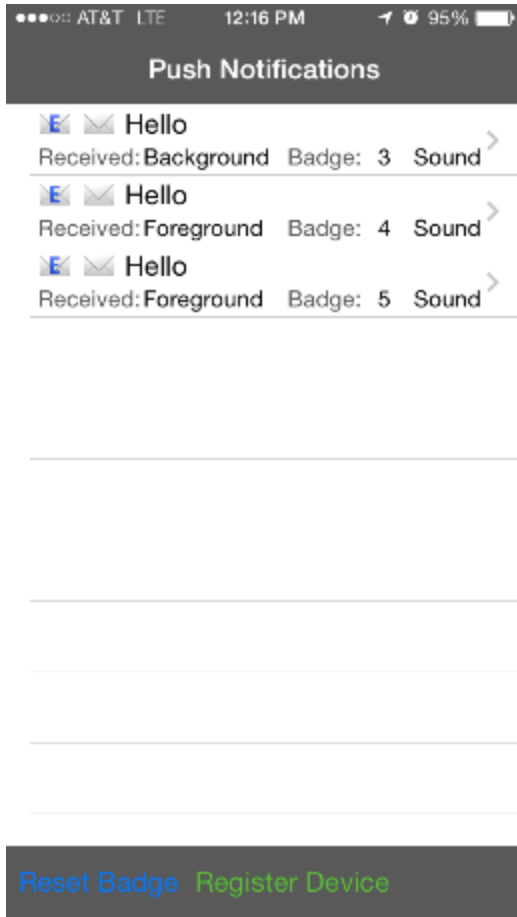
- a. If the app runs successfully, the iOS device will be registered with the back-end and you should be able to verify the installation using GET request `/api/installations`.
- b. You can also see the apnagent application on your phone. Click on Register Device and you should get an alert with the registration number.



6. To send a push notification using REST API (for example):

```
curl -X POST
http://ec2-54-184-36-164.us-west-2.compute.amazonaws.com:3000/notify/<device_registration>
OK
```

7. You should receive a notification on your device that looks like this:



Storage component

- [Overview](#)
- [Installation](#)
 - [Example](#)
- [Containers and files](#)
- [Creating a storage component data source](#)
 - [Using slc and JSON](#)
 - [Using JavaScript](#)
 - [Provider credentials](#)

Related articles:

Overview

The [LoopBack storage component](#) makes it easy to upload and download files to cloud storage providers and the local (server) file system. It has Node.js and REST APIs for managing binary content in cloud providers, including:

- Amazon
- Rackspace
- Openstack
- Azure

You use the storage component like any other LoopBack data source such as a database. Like other data sources, it supports create, read, update, and delete (CRUD) operations with exactly the same LoopBack and REST APIs.



This component does not yet provide metadata management "out of the box." For an example of how to store metadata along with files, see [How to store files with metadata in LoopBack?](#)

Installation

Install the storage component as usual for a Node package:

```
$ npm install loopback-component-storage
```

Example

For an example of using the storage component, see the [LoopBack Storage component GitHub repository](#).

Follow these steps to run the example:

```
$ git clone http://github.com/strongloop/loopback-component-storage.git
$ cd loopback-component-storage/example-2.0
$ npm install
$ node .
```

Then load <http://localhost:3000> in your browser.

Containers and files

The storage component organizes content as *containers* and *files*. A container holds a collection of files, and each file belongs to one container.

- **Container** groups files, similar to a directory or folder. A container defines the namespace for objects and is uniquely identified by its name, typically within a user account. NOTE: A container cannot have child containers.
- **File** stores the data, such as a document or image. A file is always in one (and only one) container. Within a container, each file has a unique name. Files in different containers can have the same name.

Creating a storage component data source

You can create a storage component data source either using slc and the `/server/datasources.json` file or programmatically in JavaScript.

Using slc and JSON

Create a new data source as follows:

```
$ slc loopback:datasource
[?] Enter the data-source name: myfile
[?] Select the connector for myfile: other
[?] Enter the connector name without the loopback-connector- prefix:
loopback-component-storage
```

Then edit `/server/datasources.json` and manually add the properties of the data source (properties other than "name" and "connector"; for example:

```
"myfile": {
  "name": "myfile",
  "connector": "loopback-component-storage",
  "provider": "amazon",
  "key": "your amazon key",
  "keyId": "your amazon key id"
}
```

Using JavaScript

You can also create a storage component data source programmatically with the `loopback.createDataSource()` method, putting code in `/server/server.js`. For example, using local file system storage:

server/server.js

```
var ds = loopback.createDataSource({
  connector: require('loopback-component-storage'),
  provider: 'filesystem',
  root: path.join(__dirname, 'storage')
});

var container = ds.createModel('container');
```

Here's another example, this time for Amazon:

server/server.js

```
var ds = loopback.createDataSource({
  connector: require('loopback-component-storage'),
  provider: 'amazon',
  key: 'your amazon key',
  keyId: 'your amazon key id'
});
var container = ds.createModel('container');
app.model(container);
```

You can also put this code in the `/server/boot` directory, as an exported function:

```
module.exports = function(app) {
  // code to set up data source as shown above
};
```

Provider credentials

Each cloud storage provider requires different credentials to authenticate. Provide these credentials as properties of the JSON object argument to `createDataSource()`, in addition to the `connector` property, as shown in the following table.

Provider	Property	Description	Example
Amazon	provider: 'amazon'		<pre>{ provider: 'amazon', key: '...', keyId: '...' }</pre>
	key	Amazon key	
	keyId	Amazon key ID	
Rackspace	provider: 'rackspace'		<pre>{ provider: 'rackspace', username: '...', apiKey: '...' }</pre>
	username	Your username	
	apiKey	Your API key	

Azure	provider: 'azure'		<pre>{ provider: 'azure', storageAccount: '...', storageAccessKey: '...' }</pre>
	storageAccount	Name of your storage account	
	storageAccessKey	Access key for storage account	
OpenStack	provider: 'openstack'		<pre>{ provider: 'openstack', username: '...', password: '...', authUrl: 'https://your-identity-service' }</pre>
	username	Your username	
	password	Your password	
	authUrl	Your identity service	
Local File System	provider: 'filesystem'		<pre>{ provider: 'filesystem', root: '/tmp/storage' }</pre>
	root	File path to storage root directory.	

API

Once you create a container, it will provide both a REST and Node API, as described in the following table. For details, see the complete [API documentation](#).

Description	Container Model Method	REST URI
List all containers.	getContainers(cb)	GET /api/containers
Get information about specified container.	getContainer(container, cb)	GET /api/containers/:container
Create a new container.	createContainer(options, cb)	POST /api/containers
Delete specified container.	destroyContainer(container, cb)	DELETE /api/containers/:container
List all files within specified container.	getFiles(container, download, cb)	GET /api/containers/:container/files
Get information for specified file within specified container.	getFile(container, file, cb)	GET /api/containers/:container/files/:file
Delete a file within a given container by name.	removeFile(container, file, cb)	DELETE /api/containers/:container/files/:file
Upload one or more files into the specified container. The request body must use multipart/form-data which the file input type for HTML uses.	upload(req, res, cb)	POST /api/containers/:container/upload
Download a file within specified container.	download(container, file, res, cb)	GET /api/containers/:container/download/:file
Get a stream for uploading.	uploadStream(container, file, options, cb)	

Get a stream for downloading.

downloadStream(container,
file, options, cb)

Storage component API

- **Class: StorageService**
 - storageService.getContainers
 - storageService.createContainer
 - storageService.destroyContainer
 - storageService.getContainer
 - storageService.uploadStream
 - storageService.downloadStream
 - storageService.GetFiles
 - storageService.getFile
 - storageService.removeFile
 - storageService.upload
 - storageService.download

Module: loopback-component-storage

undefined

storageService = new StorageService(options)

Storage service constructor. Properties of options object depend on the storage service provider.

Arguments

Name	Type	Description
options	Object	Options to create a provider; see below.

options

Name	Type	Description
provider	String	Storage service provider. Must be one of: <ul style="list-style-type: none"> • 'filesystem' - local file system. • 'amazon' • 'rackspace' • 'azure' • 'openstack' Other supported values depend on the provider. See the documentation for more information.

storageService.createContainer(options, cb)

Create a new storage service container.

Arguments

Name	Type	Description
options	Object	Options to create a container. Option properties depend on the provider.
cb	Function	Callback function

options

Name	Type	Description
name	String	Container name

cb

Name	Type	Description
err	Object or String	Error string or object

container	Object	Container metadata object
------------------	--------	---------------------------

storageService.destroyContainer(container, callback)

Destroy an existing storage service container.

Arguments

Name	Type	Description
container	String	Container name.
callback	Function	Callback function.

callback

Name	Type	Description
err	Object or String	Error string or object

storageService.download(container, file, res, cb)

Download middleware

Arguments

Name	Type	Description
container	String	Container name
file	String	File name
res	Response	HTTP response
cb	Function	Callback function

storageService.downloadStream(container, file, options, callback)

Get the stream for downloading.

Arguments

Name	Type	Description
container	String	Container name.
file	String	File name.
options	Object	Options for downloading
callback	Function	Callback function

callback

Name	Type	Description
err	String or Object	Error string or object

Returns

Name	Type	Description
result	Stream	Stream for downloading

storageService.getContainer(container, callback)

Look up a container metadata object by name.

Arguments

--	--	--

Name	Type	Description
container	String	Container name.
callback	Function	Callback function.

callback

Name	Type	Description
err	Object or String	Error string or object
container	Object	Container metadata object

storageService.getContainers(callback)

List all storage service containers.

Arguments

Name	Type	Description
callback	Function	Callback function

callback

Name	Type	Description
err	Object or String	Error string or object
containers	Object[]	An array of container metadata objects

storageService.getFile(container, file, cb)

Look up the metadata object for a file by name

Arguments

Name	Type	Description
container	String	Container name
file	String	File name
cb	Function	Callback function

cb

Name	Type	Description
err	Object or String	Error string or object
file	Object	File metadata object

storageService.getFiles(container, [options], cb)

List all files within the given container.

Arguments

Name	Type	Description
container	String	Container name.
[options]	Object	Options for download
cb	Function	Callback function

cb

Name	Type	Description
------	------	-------------

err	Object or String	Error string or object
files	Object[]	An array of file metadata objects

storageService.removeFile(container, file, cb)

Remove an existing file

Arguments

Name	Type	Description
container	String	Container name
file	String	File name
cb	Function	Callback function

cb

Name	Type	Description
err	Object or String	Error string or object

storageService.upload(req, res, [options], cb)

Upload middleware for the HTTP request/response

Arguments

Name	Type	Description
req	Request	Request object
res	Response	Response object
[options]	Object	Options for upload
cb	Function	Callback function

storageService.uploadStream(container, file, [options], Callback)

Get the stream for uploading

Arguments

Name	Type	Description
container	String	Container name
file	String	File name
[options]	Object	Options for uploading
Callback	callback	function

Callback

Name	Type	Description
err	String or Object	Error string or object

Returns

Name	Type	Description
result	Stream	Stream for uploading

Storage component REST API

- [List containers](#)
- [Get container information](#)
- [Create container](#)
- [Delete container](#)
- [List files in container](#)
- [Get file information](#)
- [Delete file](#)
- [Upload files](#)
- [Download file](#)

List containers

List all containers for the current storage provider.

```
GET /api/containers
```

Arguments

None.

Get container information

Get information about specified container.

```
GET /api/containers/container-name
```

Arguments

- *container-name* - name of container for which to return information.

Create container

Create a new container with the current storage provider.

```
POST /api/containers
```

Arguments

Container specification in POST body.

Delete container

Delete the specified container.

```
DELETE /api/containers/container-name
```

Arguments

- *container-name* - name of container to delete.

List files in container

List all files within a given container by name.

```
GET /api/containers/container-name/files
```

Arguments

- *container-name* - name of container for which to list files.

Get file information

Get information for a file within a given container by name

```
GET /api/containers/container-name/files/file-name
```

Arguments

- *container-name* - name of container.
- *file-name* - name of file for which to get information.

Delete file

Delete specified file within specified container.

```
DELETE /api/containers/container-name/files/file-name
```

Arguments

- *container-name* - name of container.
- *file-name* - name of file to delete.

Upload files

Upload one or more files into the given container by name. The request body should use [multipart/form-data](#) which the file input type for HTML uses.

```
POST /api/containers/container-name/upload
```

Arguments

- *container-name* - name of container to which to upload files.

Download file

Download specified file within specified container.

```
GET /api/containers/container-name/download/file-name
```

Arguments

- *container-name* - name of container from which to download file.
- *file-name* - name of file to download.

Third-party login (Passport)

- [Overview](#)
- [Installation](#)
- [Models](#)
 - [UserIdentity model](#)
 - [UserCredential model](#)
 - [ApplicationCredential model](#)
- [PassportConfigurator](#)
- [Login and account linking](#)
 - [Third party login](#)
 - [Linking third-party accounts](#)

See also:

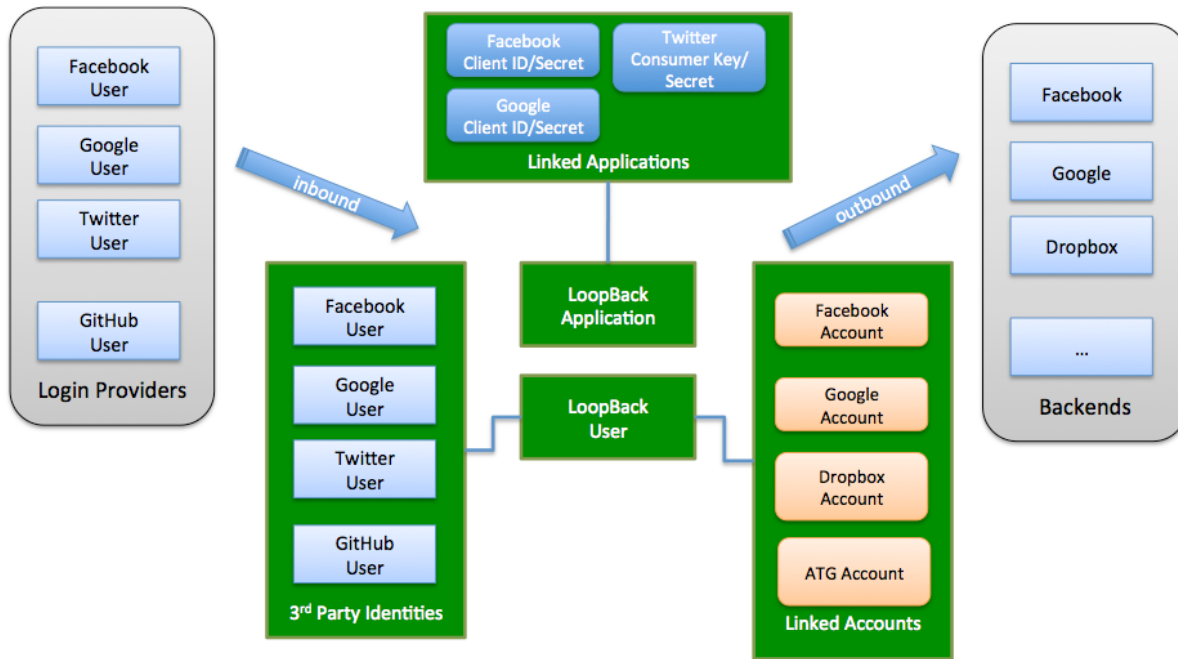
- [Example application](#)
- [API documentation for loopback-component-passport](#)
- [Passport documentation](#)

- **Configuring third-party providers**
 - Configuration in providers.json
 - Set up in application code

Overview

The `loopback-component-passport` module integrates [Passport](#) and supports:

- **Third-party login**, so LoopBack apps can allow users to login using existing accounts on Facebook, Google, Twitter, Github, and others.
- **Integration with enterprise security services** so that users can login with them instead of username and password-based authentication.
- **Linking an account** authorized through one of the above services with a LoopBack application user record. This enables you to manage and control your own user account records while still integrating with third-party services.



This module includes:

- `UserIdentity` model - keeps track of third-party login profiles.
- `UserCredential` model - stores credentials from a third-party provider to represent users' permissions and authorizations.
- `ApplicationCredential` model - stores credentials associated with a client application.
- `PassportConfigurator` - the bridge between LoopBack and Passport.

The [example application](#) demonstrates how to implement third-party login with a LoopBack application.

Installation

Install the third-party login (Passport) component as usual for a Node package:

```
$ npm install loopback-component-passport
```

Models

UserIdentity model

The `UserIdentity` model keeps track of third-party login profiles. Each user identity is uniquely identified by provider and `externalId`. The `UserIdentity` model has a `belongsTo` relation to the `User` model.

The following table describes the properties of the UserIdentity model.

Property	Type	Description
provider	String	Auth provider name; for example facebook, google, twitter, or linkedin.
authScheme	String	auth scheme, such as OAuth, OAuth 2.0, OpenID, OpenID Connect
externalId	String	Provider specific user ID.
profile	Object	User profile, see http://passportjs.org/guide/profile
credentials	Object	<ul style="list-style-type: none"> User credentials <ul style="list-style-type: none"> OAuth: token, tokenSecret OAuth 2.0: accessToken, refreshToken OpenID: openId OpenID Connect: accessToken, refreshToken, profile
userId	Any	LoopBack user ID
created	Date	Created date
modified	Date	Last modified date

UserCredential model

UserCredential has the same set of properties as UserIdentity. It's used to store the credentials from a third party authentication/authorization provider to represent the permissions and authorizations of a user in the third-party system.

ApplicationCredential model

Interacting with third-party systems often requires client application level credentials. For example, you need OAuth 2.0 client ID and client secret to call Facebook APIs. Such credentials can be supplied from a configuration file to your server globally. But if your server accepts API requests from multiple client applications, each client application needs its own credentials. The ApplicationCredential model stores credentials associated with the client application, to support multi-tenancy.

The ApplicationCredential model has a [belongsTo](#) relation to the Application model.

The following table describes the properties of ApplicationCredential model

Property	Type	Description
provider	String	Auth provider name, such as facebook, google, twitter, linkedin
authScheme	String	Auth scheme, such as OAuth, OAuth 2.0, OpenID, OpenID Connect
credentials	Object	Provider-specific credentials <ul style="list-style-type: none"> openId: {returnURL: String, realm: String} OAuth2: {clientId: String, clientSecret: String, callbackURL: String} OAuth: {consumerKey: String, consumerSecret: String, callbackURL: String}
created	Date	Created date
modified	Date	Last modified date

PassportConfigurator

PassportConfigurator is the bridge between LoopBack and Passport. It:

- Sets up models with LoopBack
- Initializes passport
- Creates Passport strategies from provider configurations
- Sets up routes for authorization and callback

Login and account linking

Third party login

The following steps use Facebook OAuth 2.0 login as an example. The basic procedure is:

1. A visitor requests to log in using Facebook by clicking on a link or button backed by LoopBack to initiate OAuth 2.0 authorization.
2. LoopBack redirects the browser to Facebook's authorization endpoint so the user can log into Facebook and grant permissions to LoopBack
3. Facebook redirects the browser to a callback URL hosted by LoopBack with the OAuth 2.0 authorization code
4. LoopBack makes a request to the Facebook token endpoint to get an access token using the authorization code
5. LoopBack uses the access token to retrieve the user's Facebook profile
6. LoopBack searches the UserIdentity model by (provider, externalId) to see there is an existing LoopBack user for the given Facebook id
7. If yes, set the LoopBack user to the current context
8. If not, create a LoopBack user from the profile and create a corresponding record in UserIdentity to track the 3rd party login. Set the newly created user to the current context.

Linking third-party accounts

Follow these steps to link LoopBack accounts using Facebook OAuth 2.0:

1. The user logs into LoopBack first directly or through third-party login.
2. The user clicks on a link or button to start the OAuth 2.0 authorization so that the user can grant permissions to LoopBack.
3. Perform steps 2-5 in third-party login, above.
4. LoopBack searches the UserCredential model by (provider, externalId) to see if there is an existing LoopBack user for the given Facebook ID.
5. Link the Facebook account to the current user by creating a record in the UserCredential model to store the Facebook credentials, such as access token.
6. Now the LoopBack user wants to get a list of pictures from the linked Facebook account(s). LoopBack can look up the Facebook credentials associated with the current user and use them to call Facebook APIs to retrieve the pictures.

Configuring third-party providers



You must register with Facebook and Google to get your own client ID and client secret.

- Facebook: <https://developers.facebook.com/apps>
- Google: <https://console.developers.google.com/project>

The following example illustrates using two providers:

- **facebook-login** for login with Facebook
- **google-link** for linking Google accounts with the current LoopBack user.
- **ms-ad** for linking LDAP records using Microsoft Active Directory. NOTE: This is an early release of support this provider.

Configuration in providers.json

The JSON file providers.json sets up the providers:

providers.json

```

{
  "facebook-login": {
    "provider": "facebook",
    "module": "passport-facebook",
    "clientID": "{facebook-client-id-1}",
    "clientSecret": "{facebook-client-secret-1}",
    "callbackURL": "http://localhost:3000/auth/facebook/callback",
    "authPath": "/auth/facebook",
    "callbackPath": "/auth/facebook/callback",
    "successRedirect": "/auth/account",
    "scope": ["email"]
  },
  ...
  "google-link": {
    "provider": "google",
    "module": "passport-google-oauth",
    "strategy": "OAuth2Strategy",
    "clientID": "{google-client-id-2}",
    "clientSecret": "{google-client-secret-2}",
    "callbackURL": "http://localhost:3000/link/google/callback",
    "authPath": "/link/google",
    "callbackPath": "/link/google/callback",
    "successRedirect": "/link/account",
    "scope": ["email", "profile"],
    "link": true
  },
  ...
  "ms-ad": {
    "provider": "ms-ad",
    "authScheme": "ldap",
    "module": "passport-ldapauth",
    "authPath": "/auth/msad",
    "successRedirect": "/auth/account",
    "failureRedirect": "/msad",
    "failureFlash": true,
    "session": true,
    "ldapAttributeForLogin": "mail",
    "ldapAttributeForUsername": "mail",
    "ldapAttributeForMail": "mail",
    "server": {
      "url": "ldap://ldap.example.org:389/dc=example,dc=org",
      "bindDn": "bindUsername",
      "bindCredentials": "bindPassword",
      "searchBase": "ou=people,dc=example,dc=org",
      "searchAttributes": ["cn", "mail", "uid", "givenname"],
      "searchFilter":
        "(&(objectcategory=person)(objectclass=user)(|(samaccountname={{username}})(mail={{use
rname}})))"
    }
  },
  ...
}

```

Set up in application code

Add code such as the following to `server/server.js` to load provider configurations, as illustrated in [loopback-example-passport](#). The following example code:

- Creates an instance of `PassportConfigurator`.
- Loads the provider configurations shown above.
- Initializes Passport.
- Sets up related models.
- Configures Passport strategies for third-party auth providers.

```
var loopback = require('loopback');
var path = require('path');
var app = module.exports = loopback();
// Create an instance of PassportConfigurator with the app instance
var PassportConfigurator =
  require('loopback-component-passport').PassportConfigurator;
var passportConfigurator = new PassportConfigurator(app);

app.boot(__dirname);
...
// Enable http session
app.use(loopback.session({ secret: 'keyboard cat' }));

// Load the provider configurations
var config = {};
try {
  config = require('./providers.json');
} catch(err) {
  console.error('Please configure your passport strategy in `providers.json`.');
  console.error('Copy `providers.json.template` to `providers.json` and replace the
  clientID/clientSecret values with your own.');
```

```
  process.exit(1);
}
// Initialize passport
passportConfigurator.init();

// Set up related models
passportConfigurator.setupModels({
  userModel: app.models.user,
  userIdentityModel: app.models.userIdentity,
  userCredentialModel: app.models.userCredential
});
// Configure passport strategies for third party auth providers
for(var s in config) {
  var c = config[s];
  c.session = c.session !== false;
  passportConfigurator.configureProvider(s, c);
}
```

Configuring providers.json

- [Overview](#)
- [Example providers.json](#)
- [Provider property reference](#)
 - [Common properties](#)
 - [OAuth 1.0](#)
 - [OAuth 2](#)
 - [Local](#)

Overview

Use the `providers.json` file to configure third-party login using `loopback-component-passport`. This file contains settings for each third-party authorization provider, in `provider` and `provider-link` objects (for example, `google-login` and `google-link`).

To load the configuration, add code such as the following to `server.js`:

/server/server.js

```
var loopbackPassport = require('loopback-component-passport');
var PassportConfigurator = loopbackPassport.PassportConfigurator;
var passportConfigurator = new PassportConfigurator(app);

// Build the providers/passport config
var config = {};
try {
  config = require('../providers.json');
} catch (err) {
  console.trace(err);
  process.exit(1); // fatal
}
```

Example providers.json

Below is the `providers.template.json` file provided with `loopback-example-passport`.

providers.template.json

> [Expand](#)

[source](#)

```
{
  "local": {
    "provider": "local",
    "module": "passport-local",
    "usernameField": "username",
    "passwordField": "password",
    "authPath": "/auth/local",
    "successRedirect": "/auth/account"
  },
  "facebook-login": {
    "provider": "facebook",
    "module": "passport-facebook",
    "clientID": "{facebook-client-id-1}",
    "clientSecret": "{facebook-client-secret-1}",
    "callbackURL": "http://localhost:3000/auth/facebook/callback",
    "authPath": "/auth/facebook",
    "callbackPath": "/auth/facebook/callback",
    "successRedirect": "/auth/account",
    "scope": ["email"],
    "authOptions": {"display": "popup"}
  },
  "google-login": {
    "provider": "google",
    "module": "passport-google-oauth",
    "strategy": "OAuth2Strategy",
    "clientID": "{google-client-id-1}",
    "clientSecret": "{google-client-secret-1}",
    "callbackURL": "http://localhost:3000/auth/google/callback",
    "authPath": "/auth/google",
```

```

    "callbackPath": "/auth/google/callback",
    "successRedirect": "/auth/account",
    "scope": ["email", "profile"]
  },
  "twitter-login": {
    "provider": "twitter",
    "authScheme": "oauth",
    "module": "passport-twitter",
    "callbackURL": "http://localhost:3000/auth/twitter/callback",
    "authPath": "/auth/twitter",
    "callbackPath": "/auth/twitter/callback",
    "successRedirect": "/auth/account",
    "consumerKey": "{twitter-consumer-key}",
    "consumerSecret": "{twitter-consumer-secret}"
  },
  "facebook-link": {
    "provider": "facebook",
    "module": "passport-facebook",
    "clientID": "{facebook-client-id-2}",
    "clientSecret": "{facebook-client-secret-2}",
    "callbackURL": "http://localhost:3000/link/facebook/callback",
    "authPath": "/link/facebook",
    "callbackPath": "/link/facebook/callback",
    "successRedirect": "/link/account",
    "scope": ["email", "user_likes"],
    "link": true
  },
  "google-link": {
    "provider": "google",
    "module": "passport-google-oauth",
    "strategy": "OAuth2Strategy",
    "clientID": "{google-client-id-2}",
    "clientSecret": "{google-client-secret-2}",
    "callbackURL": "http://localhost:3000/link/google/callback",
    "authPath": "/link/google",
    "callbackPath": "/link/google/callback",
    "successRedirect": "/link/account",
    "scope": ["email", "profile"],

```

```

    "link": true
  }
}

```

Provider property reference

Common properties

Property	Type	Description	Example	Default
authPath	String	The local URL for authentication	"/auth/facebook"	/auth/<provider>
authScheme	String	Default is OAuth 2.0	"oauth"	oAuth 2.0
link	Boolean	True if you want to link accounts.	true	false
module	String	Node module to use	"passport-facebook"	
provider	String	Identifies the provider; can be any identifier string.	"facebook"	
strategy	String	The name of passport strategy	"OAuth2Strategy"	

OAuth 1.0

Twitter

Property	Type	Description	Example
callbackPath	String	A local URL to mount the callback page	"/auth/facebook/callback"
callbackURL	String	A URL the Service Provider will use to redirect the User back to the Consumer when Obtaining User Authorization is complete	"http://localhost:3000/auth/facebook/callback"
consumerKey	String	A value used by the Consumer to identify itself to the Service Provider	
consumerSecret	String	A secret used by the Consumer to establish ownership of the Consumer Key	
scope	Array of String	An array of OAuth 1.0 scopes	["email"]
successRedirect	String	A local URL for the success login	"/auth/account"

OAuth 2

Google and Facebook


Property	Type	Description	Example
authOptions	Object	In general, the properties map to those of the Facebook login dialog , but property names and other details depend on the Passport provider's implementation; for example passport-facebook .	"authOptions": {"display": "popup"}
callbackPath	String	A local URL to mount the callback page	"/auth/facebook/callback"
callbackURL	String	OAuth 2.0 callback URL	"http://localhost:3000/auth/facebook/callback"
clientId	String	The client identifier issued to the client during the registration process	
clientSecret	String	The client secret	
scope	Array of String	An array of OAuth 2.0 scopes	["email"]

successRedirect	String	A local URL for the success login	"/auth/account"
-----------------	--------	-----------------------------------	-----------------

Local

Property	Type	Description	Example	Default
usernameField	String	The field name for username on the login form	"user"	username
passwordField	String	The field name for password on the login form	"pass"	password
successRedirect	String	A local URL for the success login	"/auth/account"	

Tutorial: third-party login

 The following is duplicated from [loopback-example-passport](#)

loopback-example-passport

A tutorial for setting up a basic passport example.

- [Overview](#)
- [Prerequisites](#)
- [Client ids/secrets from third party](#)
- [Tutorial](#)

Overview

LoopBack example for [loopback-passport](#) module. It demonstrates how to use LoopBack's `user/userIdentity/userCredential` models and [passport](#) to interact with other auth providers.

- Log in or sign up to LoopBack using third party providers (aka social logins)
- Link third party accounts with a LoopBack user (for example, a LoopBack user can have associated facebook/google accounts to retrieve pictures).

Prerequisites

Before starting this tutorial, make sure you have the following installed:

- Node
- NPM
- [StrongLoop Controller](#)

Client ids/secrets from third party

- [facebook](#)
- [google](#)
- [twitter](#)

Tutorial - Facebook

1. Clone the application

```
$ git clone git@github.com:strongloop/loopback-example-passport.git
$ cd loopback-example-passport
$ npm install
```

2. Get your client ids/secrets from third party(social logins)

- To get your app info: [facebook](#)

- Click on My Apps, then on Add a new App
- Pick the platform [iOS, Android, Facebook Canvas, Website]
- Select proper category for your app.
- Write your app name and "Site URL".
- Skip the quick start to get your "App ID" and "App Secret", which is in "Settings"
- Your app may not work if the settings are missing a contact email and/or "Site URL".
- if you are testing locally, you can simply use `localhost:[port#]` as your "Site URL".

3. Create providers.json

- Copy `providers.json.template` to `providers.json`
- Update `providers.json` with your own values for `clientID/clientSecret`.

```
"facebook-login": {
  "provider": "facebook",
  "module": "passport-facebook",
  "clientID": "xxxxxxxxxxxxxxxx",
  "clientSecret": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
  "callbackURL": "/auth/facebook/callback",
  "authPath": "/auth/facebook",
  "callbackPath": "/auth/facebook/callback",
  "successRedirect": "/auth/account",
  "failureRedirect": "/login",
  "scope": ["email"],
  "failureFlash": true
},
"facebook-link": {
  "provider": "facebook",
  "module": "passport-facebook",
  "clientID": "xxxxxxxxxxxxxxxx",
  "clientSecret": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
  "callbackURL": "/link/facebook/callback",
  "authPath": "/link/facebook",
  "callbackPath": "/link/facebook/callback",
  "successRedirect": "/auth/account",
  "failureRedirect": "/login",
  "scope": ["email", "user_likes"],
  "link": true,
  "failureFlash": true
}
```

4. Facebook profile info

If you require additional information from a Facebook profile such as a name or a gender, you can obtain it by updating `node_modules\passport-facebook\lib\strategy.js` and replacing:

```
this._profileURL = options.profileURL || 'https://graph.facebook.com/me';
```

with

```
this._profileURL = options.profileURL ||
'https://graph.facebook.com/v2.2/me?fields=first_name,gender,last_name,lin
k,locale,name,timezone,verified,email,updated_time';
```

5. Data file

- If you need to see your account info for testing purposes, in `server\datasources.json`, add:

```
"file": "db.json"
```

after

```
"connector": "memory",
```

- The account info will be saved into this file.

6. Run the application

```
$ node .
```

- Open your browser to `http://localhost:3000`
- Click on 'Login with Facebook'.
- Sign up using a local account, then link to your Facebook account.

Synchronization



StrongLoop Labs

This project provides early access to advanced or experimental functionality. It may lack usability, completeness, documentation, and robustness, and may be outdated.

However, StrongLoop supports this project: Paying customers can open issues using the StrongLoop customer support system (Zendesk). Community users, please report bugs on GitHub.

For more information, see [StrongLoop Labs](#).

- Overview
 - LoopBack in the browser
 - Terminology
- Setup
 - Enable change tracking
 - Create a client app
 - Run the client app in the browser
- Access control
- Understanding replication
 - Change model
 - Checkpoints
 - Replication algorithm
 - Conflict detection
 - Conflict resolution
 - Bulk update

See also:

- [loopback-example-full-stack](#)
- [loopback-example-offline-sync](#)

- [Sync methods](#)
- [Frequently asked questions](#)
 - [Does LoopBack support continuous replication?](#)
 - [How do you trigger immediate replication?](#)
- [Known issues](#)



Synchronization is not currently a component, but will be refactored in the future to be a LoopBack component.

Overview

In general, mobile applications need to be able to operate without constant network connectivity. This means the client app must synchronize data with the server application after a disconnected period. To do this:

- The client (browser) app replicates changes made in the server application.
- The server application replicates changes made in the client (browser) app.

This process is called *synchronization* (abbreviated as *sync*). Sync replicates data from the *source* to the *target*, and the target calls the LoopBack replication API.



The LoopBack replication API is a JavaScript API, and thus (currently, at least) works only with a JavaScript client.

Replication means intelligently copying data from one location to another. LoopBack copies data that has changed from source to target, but does not overwrite data that was modified on the target since the last replication. So, sync is just bi-directional replication.

In general there may be conflicts when performing replication. So, for example, while disconnected, a user may make changes on the client that conflict with changes made on the server; what happens when an object or field is modified both locally and remotely? LoopBack handles conflict resolution for you, and enables you to easily present a user interface to allow the end user to make informed decisions to resolve conflicts when they occur. See [Resolving conflicts](#) below.



Currently synchronization is built-in to LoopBack, but will be refactored into a component in the future.

LoopBack in the browser

LoopBack implements synchronization using the LoopBack browser API, that provides the same client JavaScript API as for Node. Thus, LoopBack in the browser is sometimes referred to as *isomorphic*, because you can call exactly the same APIs on client and server.

LoopBack in the browser uses [Browserify](#) to handle dependencies. If you wish, you can use build tools such as [Gulp](#) or [Grunt](#) to generate the client API based on the back-end models and REST API. For example, [loopback-example-full-stack](#) uses Grunt.

Synchronization as described above to handle offline operation is called *offline sync*. LoopBack also provides the ability to consolidate (or "batch") data changes the user makes on the device and send them to the server in a single HTTP request. This is called *online sync*.

Terminology

In addition to [standard terminology](#), conflict resolution uses a number of specific terms.

Change list

A list of the current and previous revisions of all models. Each data source has a unique change list.

Checkpoint

An order-able identifier for tracking the last time a source completed replication. Used for filtering the change list during replication.

Checkpoint list

An ordered list of replication checkpoints used by clients to filter out old changes.

Conflict

When replicating a change made to a source model, a conflict occurs when the source's previous revision differs from the target's current revision.

Rebasing

Conflicts can only be resolved by changing the revision they are based on. Once a source model is "rebased" on the current target version of a model, it is no longer a conflict and can be replicated normally.

Revision

A string that uniquely identifies the state of a model.

Setup

Setup involves three steps:

1. Enable change tracking in the LoopBack app.
2. Create a client app that uses the LoopBack API.
3. Run the client app in the browser

Enable change tracking

You must enable change tracking for each model that you want to be able to access offline. Make the following change to the [Model definition JSON file](#):

- Set `trackChanges` to `true`.
- Change the `id` property to an auto-generated GUID; for information on GUIDs, see [Model definition JSON file reference](#).
- Set `strict` property to `validate`.
- Set the `persistUndefinedAsNull` property to `true`.

For example:

common/models/todo.json

```
{
  "name": "Todo",
  "base": "PersistedModel",
  "strict": "validate",
  "trackChanges": true,
  "persistUndefinedAsNull": true,
  "id": {
    "id": true,
    "type": "string",
    "defaultFn": "guid"
  },
  "title": {
    "type": "string",
    "required": true
  },
  "description": {
    "type": "string"
  }
}
```

For each change-tracked model, a new model (database table) is created to contain change-tracking records. In the example above, a `Todo-Change` model will be created. The change model is attached to the same data source as the model being tracked. Therefore, you will need to migrate your database schema after you have enabled change tracking, for example using StrongLoop Arc.

The change-tracking records are updated in background. Any errors are reported via the static model method `handleChangeError`. It is recommended to provide a custom error handler in your models, as the default behavior is to throw an error.

common/models/todo.js

```
module.exports = function(Todo) {
  Todo.handleChangeError = function(err) {
    console.warn('Cannot update change records for Todo:', err);
  };
}
```

Create a client app

The next step is to create client-side LoopBack app. For each replicated model, create two new client-only subclasses:

- A local model that will use local storage to persist the changes offline
- A remote model that will be connected to the server and used as a target for replication. This model will have change tracking disabled (because the server is already handling it) and enable only the replication REST API.

For example, for the To Do example, here is the JSON file that defines the client local model:

client/models/local-todo.json

```
{
  "name": "LocalTodo",
  "base": "Todo"
}
```

Here is the JSON file that defines the client remote local model:

client/models/remote-todo.json

```
{
  "name": "RemoteTodo",
  "base": "Todo",
  "plural": "Todos",
  "trackChanges": false,
  "enableRemoteReplication": true
}
```

And here is the client model configuration JSON file:

client/model-config.json

```
{
  "_meta": {
    "sources": ["../../common/models", "../models"]
  },
  "RemoteTodo": {
    "dataSource": "remote"
  },
  "LocalTodo": {
    "dataSource": "local"
  }
}
```

Here is the JSON file that defines the client datasources:

client/datasources.json

```
{
  "remote": {
    "connector": "remote",
    "url": "/api"
  },
  "local": {
    "connector": "memory",
    "localStorage": "todo-db"
  }
}
```

Now that you have all models in place, you can set up bi-directional replication between `LocalTodo` and `RemoteTodo`, for example in a client boot script:

client/boot/replication.js

```

module.exports = function(client) {
  var LocalTodo = client.models.LocalTodo;
  var RemoteTodo = client.models.RemoteTodo;

  var since = { push: -1, pull: -1 };

  function sync() {
    // It is important to push local changes first,
    // that way any conflicts are resolved at the client
    LocalTodo.replicate(
      RemoteTodo,
      since.push,
      function pushed(err, conflicts, cps) {
        // TODO: handle err
        if (conflicts.length)
          handleConflicts(conflicts);

        since.push = cps;

        RemoteTodo.replicate(
          LocalTodo,
          since.pull,
          function pulled(err, conflicts, cps) {
            // TODO: handle err
            if (conflicts)
              handleConflicts(conflicts.map(function(c) { return c.swapParties(); }));
            since.pull = cps;
          });
      });
  }

  LocalTodo.observe('after save', function(ctx, next) {
    next();
    sync(); // in background
  });

  LocalTodo.observe('after delete', function(ctx, next) {
    next();
    sync(); // in background
  });

  function handleConflicts(conflicts) {
    // TODO notify user about the conflicts
  }
};

```

Run the client app in the browser

The loopback-boot module provides a build tool for adding all application metadata and model files to a Browserify bundle. [Browserify](#) is a tool that packages Node.js scripts into a single file that runs in a browser.

Below is a simplified example packaging the client application into a browser "module" that can be loaded via `require('lbclient')`. Consult [build.js](#) in loopback-example-full-stack for a full implementation that includes source-maps and error handling.

client/build.js

```

var b = browserify({ basedir: __dirname });
b.require('./client.js', { expose: 'lbclient' });

boot.compileToBrowserify({ appRootDir: __dirname }, b);

var bundlePath = path.resolve(__dirname, 'browser.bundle.js');
b.pipe(fs.createWriteStream(bundlePath));

```

Access control

Because the sync algorithm calls the REST API, it honors model access control settings.

However, when replicating changes only from the server (read-only replication), the client needs to create a new checkpoint value, which requires write permissions. The "REPLICATE" permission type supports this use case: it grants limited write access to the checkpoint-related methods only. For a certain user (a role, a group) to be able to pull changes from the server, they need both READ and REPLICATE permissions. Users with WRITE permissions are automatically granted REPLICATE permission too.

Example ACL configuration:

common/models/car.json

```

{
  "acls": [
    // disable anonymous access
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "DENY"
    },
    // allow all authenticated users to read data
    {
      "principalType": "ROLE",
      "principalId": "$authenticated",
      "permission": "ALLOW",
      "accessType": "READ"
    },
    // allow all authenticated users to pull changes
    {
      "principalType": "ROLE",
      "principalId": "$authenticated",
      "permission": "ALLOW",
      "accessType": "REPLICATE"
    },
    // allow the user with id 0 to perform full sync
    {
      "principalType": "USER",
      "principalId": 0,
      "permission": "ALLOW",
      "accessType": "WRITE"
    }
  ]
}

```

Understanding replication

Offline data access and synchronization has three components:

- Change tracking
- Replication of changes
- Browser version of LoopBack

Change model

As explained above, a new change model is created for each change-tracked model, e.g. `Todo-Change`. This model can be accessed using the method `getChangeModel`, for example, `Todo.getChangeModel()`.

The change model has several properties:

- `modelId` links a change instance (record) with a tracked model instance
- `prev` and `rev` are hash values generated from the model class the Change model is representing. The `rev` property stands for Revision, while `prev` is the hash of the previous revision. When a model instance is deleted, the value `null` is used instead of a hash.
- `checkpoint` associates a change record with a Checkpoint, more on this later.

Additionally, there is a method `type()` that can be used to determine the kind of change being made: `Change.CREATE`, `Change.UPDATE`, `Change.DELETE` or `Change.UNKNOWN`.

The current implementation of the change tracking algorithm keeps only one change record for each model instance - the last change made.

Checkpoints

A checkpoint represents a point in time that you can use to filter the changes to only those made after the checkpoint was created. A checkpoint is typically created whenever a replication is performed, this allows subsequent replication runs to ignore changes that were already replicated.

While in theory the replication algorithm should work without checkpoints, in practice it's important to use correct checkpoint values because the current implementation keeps the last change only.

If you don't pass correct values in the `since` argument of `replicate` method, then you may

- Get false conflicts if the "since" value is omitted or points to an older, already replicated checkpoint.
- Incorrectly override newer changes with old data if the "since" value points to a future checkpoint that was not replicated yet.

Replication algorithm

A single iteration of the replication algorithm consists of the following steps:

1. Create new checkpoints (both source and target)
2. Get list of changes made at the source since the given source checkpoint
3. Find out differences between source and target changes since the given target checkpoint, detect any conflicts.
4. Create a set of instructions - what to change at target
5. Perform a "bulk update" operation using these instructions
6. Return the new checkpoints to the callback

It is important to create the new checkpoints as the first step of the replication algorithm. Otherwise any changes made while the replication is in progress would be associated with the checkpoint being replicated, and thus they would not be picked up by the next replication run.

The consequence is that the "bulk update" operation will associate replicated changes with the new checkpoint, and thus these changes will be considered during the next replication run, which may cause false conflicts.

In order to prevent this problem, the method `replicate` runs several iterations of the replication algorithm, until either there is nothing left to replicate, or a maximum number of iterations is reached.

Conflict detection

Conflicts are detected in the third step. The list of source changes are sent to the target model, which compares them to change made to target model instances. Whenever both source and target modified the same model instance (the same model id), the algorithm checks the current and previous revision of both source and target models to decide whether there is a conflict.

A conflict is reported when both of these conditions are met:

- The current revisions are different, i.e. the model instances have different property values.
- The *current* target revision is different from the *previous* source revision. In other words, if the source change is in sequence after the target change, then there is no conflict.

Conflict resolution

Conflict resolution can be complex. Fortunately, LoopBack handles the complexity for you, and provides an API to resolve conflicts intelligently.

The callback of `Model.replicate()` takes `err` and `conflict[]`. Each `conflict` represents a change that was not replicated and must be manually resolved. You can fetch the current versions of the local and remote models by calling `conflict.models()`. You can manually merge the conflict by modifying both models.

Calling `conflict.resolve()` will set the source change's previous revision to the current revision of the (conflicting) target change. Since the changes are no longer conflicting and appear as if the source change was based on the target, they will be replicated normally as part of the next `replicate()` call.

The conflict class provides methods implementing three most common resolution scenarios, consider using these methods instead of `conflict.resolve()`:

- `conflict.resolveUsingSource()`
- `conflict.resolveUsingTarget()`
- `conflict.resolveManually()`

Bulk update

The bulk update operation expects a list of instructions - changes to perform. Each instructions contains a `Change` instance describing the change, a change type, and model data to use.

In order to prevent race conditions when third parties are modifying the replicated instances while the replication is in progress, the `bulkUpdate` function is implementing a robust checks to ensure it modifies only those model instances that have their expected revision.

The "diff" step returns the current target revision of each model instances that needs an update, this revision is stored as the `change.rev` property.

The "bulkUpdate" method loads the model instance from the database, verifies that the current revision matched the expected revision in the instruction, and then performs a conditional update/delete specifying all model properties as the condition.

```
// Example: apply an update of an existing instance
var current = findById(data.id);
if (revisionOf(current) !== expectedRev)
  return conflict();
var c = Model.updateAll(current, data);
if (c !== 1) conflict();
```

Sync methods

The LoopBack Model object provides a number of methods to support sync, mixed in via the `DataModel` object:

- **bulkUpdate** - Apply an update list.
- **changes** - Get the changes to a model since a given checkpoint. Provide a filter object to reduce the number of results returned.
- **checkpoint** - Create a checkpoint.
- **createUpdates** - Create an update list for `Model.bulkUpdate()` from a delta list from `Change.diff()`.
- **currentCheckpoint** - Get the current checkpoint ID.
- **diff** - Get a set of deltas and conflicts since the given checkpoint.
- **enableChangeTracking** - Start tracking changes made to the model.
- **getChangeModel** - Get the `Change` model.
- **getSourceId** - Get the source identifier for this model / `dataSource`.
- **handleChangeError** - Handle a change error. Override this method in a subclassing model to customize change error handling.
- **rectifyChange** - Tell LoopBack that a change to the model with the given ID has occurred.
- **replicate** - Replicate changes since the given checkpoint to the given target model.
- **findLastChange** - Get the last (current) `Change` object for a given model instance.
- **updateLastChange** - Update the last (current) `Change` object associated with the given model instance.

Frequently asked questions

Does LoopBack support continuous replication?

Yes: with continuous replication, the client immediately triggers a replication when local data changes and the server pushes changes when then occur.

Here is a basic example that relies on a `socket.io` style `EventEmitter`.


```
// psuedo-server.js
MyModel.on('changed', function(obj) {
  socket.emit('changed');
});

// psuedo-client.js
socket.on('changed', function(obj) {
  LocalModel.replicate(RemoteModel);
});
```

How do you trigger immediate replication?

Call `Model.replicate()` to trigger immediate replication.

Known issues

- The size of the browser bundle is over 1.4MB, which is too large for mobile clients. See <https://github.com/strongloop/loopback/issues/989>.
- It's not possible to set a model property to `undefined` via the replication. When a property is undefined at the source but defined at the target, "bulk update" will not set it to undefined at the target. This can be mitigated by using `strict` model and enabling `persistUndefinedAsNull`.
- Browser's `localStorage` limits the size of stored data to about 5MB (depending on the browser). If your application needs to store more data in offline mode, then you need to use `IndexedDB` instead of `localStorage`. LoopBack does not provide a connector for `IndexedDB` yet. See <https://github.com/strongloop/loopback/issues/858>.
- Not all connectors were updated to report the number of rows affected by `updateAll` and `deleteAll`, which is needed by "bulkUpdate". As a result, the replication fails when the target model is persisted using one of these unsupported connectors.
- LoopBack does not fully support fine-grained access control to a selected subset of model instances, therefore it is not possible to replicate models where the user can access only a subset of instances (for example only the instances the user has created).

Sync example app

This example application is a simple "To Do" list that illustrates LoopBack synchronization using isomorphic LoopBack. It is based on the well-known [TodoMVC](#) example.

Prerequisites

You must install the following:

- Node.js. See [Getting started with LoopBack](#).
- [Git](#)
- [MongoDB](#) - if you want to be able to preserve state between application restarts.
- The application uses [Grunt](#) to generate the isomorphic client-side JavaScript API. If you have not already done so, install Grunt:

```
$ npm install grunt-cli -g
```

Run the application

Follow these steps to build and run the example application.

1. Clone the repository and change your current directory:

```
$ git clone https://github.com/strongloop/loopback-example-full-stack.git
$ cd loopback-example-full-stack
```

2. Install the root package dependencies:

```
$ npm install
```

3. Run Bower to install the client scripts:

```
$ bower install
```

4. **Optional:** If you want to run MongoDB, then:

```
$ mongod
```



Make sure you have set the environment variable `NODE_ENV=production`.

5. Build and run the entire project in development mode

```
$ grunt serve
```

6. Open <http://localhost:3000> in your browser to view the running app.
7. Click [View your todo list](#) to see the main screen:



Try it out

First, add a couple of "To Do" items: click [View your todo list](#) then in the "What needs to be done" field enter a short text item and hit RETURN.

Then you can use the API Explorer to check out the LoopBack API: <http://localhost:3000/explorer/>.

Follow these steps to see the synchronization API in action.

1. Create a new todo item.
2. Find the ID of the todo: In the API explorer, go to http://localhost:3000/explorer/#!/Todos/findOne_get_5, then click **Try it out!** You'll see a JSON response that looks something like this:

```
{
  "id": "t-2562",
  "title": "Spackle the den",
  "completed": false,
  "created": 1404245971346
}
```

- Update the To Do item: In the API explorer, go to http://localhost:3000/explorer#!/Todos/ToDo_upsert_put_1.
- In the data field, enter JSON such as the following, but change the value of the id field to the value returned in step 2.

```
{"id": "t-2562", "title": "zarflag"}
```

Where the id is the actual ID of the To Do item.

- Make a change in the browser (against the stale local data / without syncing).
- The conflict GUI pops up and you can choose how to resolve the conflict.

Advanced topics - sync

- [Replication algorithm](#)
- [Conflict resolution](#)
- [Change and revision semantics](#)
 - [Existing data](#)
 - [Revision tracking](#)
 - [Change list / change model](#)
- [Basic performance optimizations](#)

Replication algorithm

- Assign an unique identifier to the source. Most of the time it will be a GUID.
- Save this identifier on the target. This will track a source's replication checkpoint.
- Get a subset of the source change list since the last known replication checkpoint.
- Send the subset change list, containing model ids, current revisions, and previous revisions to the target.
- The target will compare this change list with its own and send back a further subset change list containing revisions that differ and conflict.
- The source then sends a set of bulk updates based on the revisions that differ. Conflicting changes must be resolved by the user before being sent.
- Once the bulk update is complete, the source should save the new checkpoint, provided as the result of the bulk update.
- The target adds a new checkpoint for the source identifier from the first step.

Conflict resolution

To resolve a conflict, a model must be rebased on the target's current revision. There are two basic rebase approaches that will resolve a conflict:

- Fetch the target model's revision
 - Merge the source and target models
 - Rectify the local change list, effectively rebasing the source model on the current revision of the target model
 - The source model may now be replicated
- Forcibly rebase the source model on the current target model's revision
 - Rectify the local change list by modifying the previous version for the source model's entry
 - The source model may now be replicated

Change and revision semantics

Changes are determined using the following logic:

- **Created** - an entry in the change list without a previous revision
 - ...or if an entry does not exist and the given model **does exist**
- **Modified** - an entry with a current and previous revision
- **Deleted** - an entry with only a previous revision
 - ...or if an entry does not exist and the given model also **does not exist**

Existing data

The change list and replication can be used against existing data sources with large sets of data. This is possible using the revision semantics above.

Note: changes made without the loopback API will be treated as "created" or "deleted" change list entries.

Revision tracking

Changes made to a data source that supports replication will be tracked as revisions in that data source's change list.

Change list / change model

A data source's change list is stored like any other model data. Each Model has its own "Change" model. The "Change" model may be attached to any data source. This allows you to store your change lists in any data source.

Basic performance optimizations

Filtering

During replication you may supply a filter. The less data you are trying to replicate, the faster and more memory efficient the replication will be.

Parallel replication

If you are manually implementing the replication algorithm, you should run as many steps in the replication process in parallel as possible. This is usually the "bulk update" step(s).

Tutorial: Offline Synchronization



This article is reproduced from <https://github.com/strongloop/loopback-example-offline-sync>

loopback-example-offline-sync

Note: This example uses loopback@2.0.0 and loopback-boot@2.0.0!

An example running LoopBack in the browser and server, demonstrating the following features:

- offline data access and synchronization
- routes shared between the AngularJS app and the HTTP server

Install and Run

1. You must have `node` and `git` installed. It's recommended to have `mongod` installed too, so that the data is preserved across application restarts.
2. Clone the repo.
3. `cd loopback-example-offline-sync`
4. `npm install` - install the root package dependencies.
5. `npm install grunt-cli -g` - skip if you have Grunt CLI already installed.
6. `npm install bower -g` - skip if you already have Bower installed.
7. `bower install` - install front-end scripts
8. `mongod` - make sure mongodb is running if you want to run with `NODE_ENV=production`.
9. `grunt serve` - build and run the entire project in development mode.
10. open `http://localhost:3000` - point a browser at the running application.

Project layout

The project is composed from multiple components.

- `common/models/` contains definition of models that are shared by both the server and the client.
- `client/lbclient/` provides an isomorphic loopback client with offline synchronization. The client needs some client-only models for data synchronization. These models are defined in `client/lbclient/models/`.

- `client/ngapp/` is a single-page AngularJS application scaffolded using `yo angular`, with a few modifications to make it work better in the full-stack project.
- `server/` is the main HTTP server that brings together all other components. Also contains the REST API server; it exposes the shared models via REST API.

Build

This project uses [Grunt](#) for the build, since that's what `yo angular` creates.

There are three major changes from the generic Gruntfile required for this full-stack example:

- `grunt serve` uses the `server/` component instead of `grunt connect`.
- `lbclient` component provides a custom build script (`lbclient/build.js`) which runs `browserify` to produce a single js file to be used in the browser. The Gruntfile contains a custom task to run this build.
- The definition of Angular routes is kept in a standalone JSON file that is used by the `server/` component too. To make this JSON file available in the browser, there is a custom task that builds `ngapp/config/bundle.js`.

Targets

- `grunt serve` starts the application in development mode, watching for file changes and automatically reloading the application.
- `grunt test` runs automated tests (only the front-end has tests at the moment).
- `grunt build` creates the bundle for deploying to production.
- `grunt serve:dist` starts the application serving the production bundle of the front-end SPA.
- `grunt jshint` checks consistency of the coding style.

Adding more features

Define a new shared model

The instructions assume the name of the new model is 'MyModel'.

1. Create a file `models/my-model.json`, put the model definition there. Use `models/todo.json` as an example, see [loopback-boot docs](#) for more details about the file format.
2. (Optional) Add `models/my-model.js` and implement your custom model methods. See `models/todo.js` for an example.
3. Add an entry to `rest/models.json` to configure the new model in the REST server:

4. Define a client-only model to represent the remote server model in the client - create `lbclient/models/my-model.json` with the following content:

5. Add two entries to `lbclient/models.json` to configure the new models for the client:

6. Register the local model with Angular's injector in `ngapp/scripts/services/lbclient.js`:

Create a new Angular route

Since the full-stack example project shares the routes between the client and the server, the new route cannot be added using the yeoman generator.

1. (Optional) Create a new angular controller using yeoman, for example,

2. (Optional) Create a new angular view using yeoman, for example,

3. Add a route entry to `ngapp/config/routes.json`, for example,

OAuth 2.0

- [Overview](#)
 - [Key elements](#)
- [Installation](#)
- [Using the OAuth2 component](#)
 - [Server types](#)
 - [Authorization server options](#)
 - [Supported grant types](#)
 - [Custom functions for token generation](#)
- [Protect endpoints with OAuth 2.0](#)
- [Examples](#)

Overview

The [LoopBack oAuth 2.0 component](#) enables LoopBack applications to function as oAuth 2.0 providers to authenticate and authorize client applications and users to access protected API endpoints.

The oAuth 2.0 protocol implementation is based on [OAuth2orize](#) and [Passport](#).

For more information on oAuth 2.0, see:

- [An Introduction to OAuth 2 \(Digital Ocean\)](#)
- [Understanding OAuth2 \(BubbleCode blog\)](#)

Internet Engineering Taskforce (IETF) technical specifications (Request for Comments or RFC):

- [OAuth 2.0 specification](#)
- [OAuth 2.0 bearer token](#)
- [OAuth 2.0 JWT token](#)

Key elements

The LoopBack OAuth 2.0 component has the following key elements:

- **Authorization server:** Issues access tokens to the client after successfully authenticating the resource owner and obtaining authorization. The component implements the OAuth 2.0 protocol endpoints, including [authorization endpoint](#) and [token endpoint](#).
- **Resource server:** Hosts the protected resources, and is capable of accepting and responding to protected resource requests using access tokens. The component provides middleware to protect API endpoints so that only requests with valid OAuth 2.0 access tokens are accepted. It also establishes identities such as client application ID and user ID for further access control and personalization.

The authorization server may be the same server as the resource server or separate. A single authorization server may issue access tokens accepted by multiple resource servers.

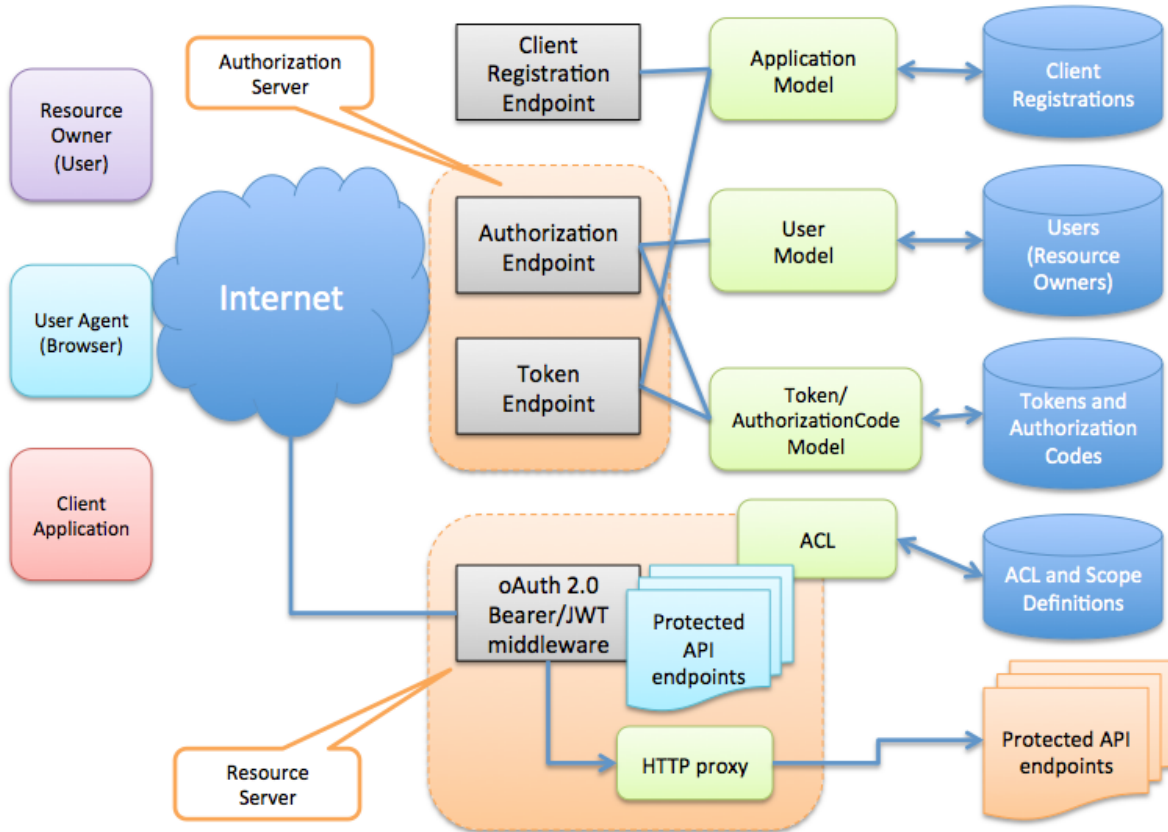
The component defines the following models to manage OAuth 2.0 metadata such as access tokens, authorization codes, clients (applications), and resource owners (users):

- `OAuthAccessToken` (persisting access tokens)
- `OAuthAuthorizationCode` (persisting authorization codes)

It also uses the user and application model from the loopback module:

- `User` (managing resource owners)

- Application (managing client applications)



Installation

Install the component as usual:

```
$ npm install loopback-component-oauth2
```

Using the OAuth2 component

Use in an application as follows:

```
var oauth2 = require('loopback-component-oauth2');

var options = {
  dataSource: app.dataSources.db, // Data source for OAuth2 metadata persistence
  loginPage: '/login', // The login page URL
  loginPath: '/login' // The login form processing URL
};

oauth2.oAuth2Provider(
  app, // The app instance
  options // The options
);
```

The app instance will be used to set up middleware and routes. The data source provides persistence for the OAuth 2.0 metadata models.

Server types

Two option properties indicate if you want to set up the OAuth 2.0 provider as an authorization server, a resource server, or both.

Property	Type	Description
authorizationServer	Boolean	If true, set up the authorization server.
resourceServer	Boolean	If true, set up the resource server.

Authorization server options

The following options are available for an authorization server:

Property	Type	Description	Default
authorizePath	String or Boolean (only false)	If String, specifies the path to mount the authorization endpoint. If false, do not set up the authorization endpoint.	/oauth/authorize
tokenPath	String or Boolean (only false)	If String, specifies the path to mount the token endpoint. If false, do not set up the token endpoint	POST /oauth/token
decisionPath	String or Boolean (only false)	If String, specifies the path to mount the decision endpoint. If false, do not set up the decision endpoint	POST /oauth/authorize/decision
decisionView	String	Server-side view name to render the decision dialog. The input for the view is: <ul style="list-style-type: none"> transactionId: An internal token to prevent forging user: user/resource owner object client: client application object scope: OAuth 2.0 scope(s) redirectURI: redirect URI after the decision is made 	
decisionPage	String	URL to the decision dialog page. Overrides decisionView. The query parameters are: <ul style="list-style-type: none"> transactionId: An internal token to prevent forging userId: user/resource owner ID clientId: client application ID scope: OAuth 2.0 scope(s) redirectURI: redirect uri after the decision is made 	
loginPath	String or Boolean (only false)	Path to mount the user login endpoint.	POST /login
loginPage	String	URL to the login dialog page.	/login

Supported grant types

The `supportedGrantTypes` option controls what grant types should be enabled:

- supportedGrantTypes (string[])
 - default to ['authorizationCode', 'implicit', 'clientCredentials', 'resourceOwnerPasswordCredentials', 'refreshToken', 'jwt'];

Custom functions for token generation

- generateToken: function(options) returns a token string
- getTTL: function(grantType, clientId, resourceOwner, scopes) returns a ttl number in seconds

Protect endpoints with OAuth 2.0


```
oauth2.authenticate(['/protected', '/api', '/me'],  
  {session: false, scope: 'email'});
```

Examples

See [strong-gateway](#) for an example of using OAuth2.0 with the StrongLoop API Gateway.

API Explorer

LoopBack API Explorer is a component that enables the [API Explorer](#) for an application. API Explorer is very useful during development, but in general you probably want to disable it in production.



API Explorer adds "filter" to the query string, but you must enter [Stringified JSON](#) in the **filter** field. Also make sure that the quotes you use are proper straight quotes ("), not curved or typographic quotation marks (" or "). These can often be hard to distinguish visually.