FREE eBook

LEARNING PubNub

Free unaffiliated eBook created from
Stack Overflow contributors.

#pubnub

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: pubnub

It is an unofficial and free PubNub ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official PubNub.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with PubNub

Remarks

This is an simple, yet thorough, example of initializing PubNub, subscribing to a channel and publishing to that channel.

- Once you init PUBNUB, you can subscribe to a channel.
- The `connect` callback indicates that subscription to the channel was successful, so we call our `pub` function which performs a `publish` to the channel we just subscribed to.
- This published message will be sent to the PubNub network which will send the message to all active subscribers. In this case, it is just us so we will receive that message in our `message` callback where we are displaying the various attributes of the received message to our browser's Console.

In a real world use case, you would update your web page UI to display the received message.

See also: latest/official PubNub JavaScript SDK Docs

Versions

Version	Release Date
3.15.x	2016-04-01

Examples

Publish on Subscribe Success (connect)

This example show how to subscribe, and once that is successful, publishing a message to that channel. It also demonstrates the full set of parameters that can be included in the `subscribe`'s `message` callback function.

```
pubnub = PUBNUB({
    publish_key   : 'your_pub_key',
    subscribe_key : 'your_sub_key'
});

pubnub.subscribe({
    channel : "channel-1",
    message : function (message, envelope, channelOrGroup, time, channel) {
        console.log(
        "Message Received." + "\n" +
        "Channel or Group: " + JSON.stringify(channelOrGroup) + "\n" +
        "Channel: " + JSON.stringify(channel) + "\n" +
        "Message: " + JSON.stringify(message) + "\n" +
        "Time: " + time + "\n" +
```

```
        "Raw Envelope: " + JSON.stringify(envelope)
    )},
    connect:    pub,
    disconnect: function(m) {console.log("DISCONNECT: " + m)},
    reconnect:  function(m) {console.log("RECONNECT: " + m)},
    error:      function(m) {console.log("ERROR: " + m)}
});

function pub() {
   pubnub.publish({
        channel : "channel-1",
        message : {"msg": "I'm Puuumped!"},
        callback: function(m) {console.log("Publish SUCCESS: " + m)},
        error: function(m) {console.log("Publish ERROR: " + m)}
   })
};
```

Read Getting started with PubNub online: https://riptutorial.com/pubnub/topic/327/getting-started-with-pubnub

Chapter 2: Access Manager

Examples

Wildcard Channel Group Manage Grant - Java SDK v4

When it comes to adding/removing channels to/from your channel groups, you need to have must have the `manage` permission for those channel groups. But you should never grant clients the permission to `manage` the channel groups that they will subscribe to. If they did, then they could add any channel they wanted to their channel group and instantly have read access to that channel.

So this is why your server must be the only entity that has the `manage` permission. But your server will need to have the `manage` permission for every single channel group so that it can add/remove channels to/from channel groups on behalf of all of the clients.

But granting `manage` to each and every channel group can be a bit tedious. Instead, you can grant `manage` to all channel groups (existing and to be created) in one *wildcard* grant.

```
// init PubNub instance using PNConfiguration with the secret-key
PNConfiguration pnConfiguration = new PNConfiguration();
pnConfiguration.setSubscribeKey("my_subkey")
pnConfiguration.setPublishKey("my_pubkey");
// secret key allows server to `grant` permissions
pnConfiguration.setSecretKey("my_secretkey");
pnConfiguration.setSecure(true);
// set the the server's auth key
pnConfiguration.setAuthKey("server_authkey");
PubNub pubnub = new PubNub(pnConfiguration);

// grant read and manage using the channel group wildcard - ":"
// with forever ttl (0)
pubNub.grant()
    .channelGroups(Arrays.asList(":")) // colon (:) is channel group wildcard
    .manage(true) // add/remove channels to/from channel groups
    .read(true) // in case server needs to subscribe or do here-now on channel groups
    .ttl(0) // 0 = forever grant
    .async(new PNCallback<PNAccessManagerGrantResult>() {
        @Override
        public void onResponse(PNAccessManagerGrantResult result, PNStatus status) {
            // check status for success or failure of grant
        }
    });
```

From here on, your server will be able to add/remove channels to/from any channel group your app creates.

pubnub access manager at server side

PubNub Access Manager (PAM) extends PubNub's existing security framework by allowing developers to create and enforce secure access to channels throughout the PubNub Real Time Network.

Access Manager allows you to manage granular permissions for your realtime apps and data streams, create multiple permission levels, grant and revoke access, and audit user access.

To use Access Manager, you need to enable Access Manager in the Admin Dashboard. Once you enable Access Manager, you must grant permissions before any data can be sent or received.

```
PAM Server side Configuration
```

In order to client side working correctly, at server side must first issue the appropriate permissions for a given PAM channel or channel-group and auth token combination.

for granting these permission you must initialize pubnub instance at least with your subscribe and secret keys.

Example :

Step 1. Make Pubnub Configuration : -

```
PNConfiguration pnConfiguration = new PNConfiguration();
        pnConfiguration.setSubscribeKey(SUBSCRIBE_KEY);
        pnConfiguration.setPublishKey(PUBLISH_KEY);;
        pnConfiguration.setSecretKey(SECRET_KEY);
        pnConfiguration.setSecure(true);
        pnConfiguration.setLogVerbosity(PNLogVerbosity.BODY);
```

Step 2. Initialize PubNub with pnConfiguration

```
PubNub pubnub = new PubNub(pnConfiguration);
```

PAM Operation occurs at three level

```
1. A global level (no auth key, and no channel/channel group is defined)

2. A channel/channel group level (only a channel/channel group is defined)

3. A channel/channel group and key level (where both the channel/channel     group and key are
defined)
```

At all these levels we can grant , revoke and audit permissions. Here we do the same on channel/channel group and auth key level.

```
PAM  Grant
```

we can grant a read/write permission to auth_key on specific channels or channel groups

Example:

Synchronously:

```
try {
```

```
    pubnub.grant().authKeys(Arrays.asList("auth1,auth2"))
    .channels(Arrays.asList("channel1,channel2")).read(true).write(true    ).ttl(0).sync();

} catch (PubNubException e) {
    e.printStackTrace();
}
```

Asynchronously:

pubNub.grant().channels(channels).authKeys(Arrays.asList(authKey)).read(true).write(true).manage(false
.async(new PNCallback() {

```
@Override
public void onResponse(PNAccessManagerGrantResult result,
     PNStatus status) {
}});
```

`PAM REVOKE:` we can revoke a permission to auth_key from a specific channel or channel groups.

Syntax for revoking permission same as granting . Just we need to change the permission true to false.

```
try {

    pubnub.grant().authKeys(Arrays.asList("auth1,auth2"))
    .channels(Arrays.asList("channel1,channel2")).read(false).write( false    ).ttl(0).sync();

} catch (PubNubException e) {
    e.printStackTrace();
}
```

`PAM Audit:` we can audit a given permission to specific channel/channel group or to a given auth_key on specific channel or channel group

Example:

```
pubnub.audit().channel("mycha").authKeys(Arrays.asList("a1")).async(new
PNCallback<PNAccessManagerAuditResult>(){

    @Override
    public void onResponse(PNAccessManagerAuditResult result,
            PNStatus status) {

            }
    });
```

`PAM Add Channels into groups:` we can also add channels into channel groups

Example:

```
pubnub.addChannelsToChannelGroup().channelGroup("my_channel").channels(Arrays.asList("my_channel5"))

    .async(new PNCallback<PNChannelGroupsAddChannelResult>() {
```

```
            @Override
            public void onResponse(PNChannelGroupsAddChannelResult
result,PNStatus status) {


            }
});
```

```
Authentication Isue at Client Side (403 Forbidden):
```

If there is an error performing PAM operations, you may receive a 403 error. If you do, be sure you have set the correct secret_key, and the issuing computer's clock is synced with NTP.

```
NTP  Setup
```

Network Time Protocol (NTP) is a protocol that is used to synchronize computer clock times in a network of computers. NTP uses Coordinated Universal Time (UTC) to synchronize computer clock times to a millisecond, and sometimes to a fraction of a millisecond.

Here we need to scyn server time with pubnub. Follow the step for doing so

Step 1 Intallation NTP

```
$ sudo apt-get update
$ sudo apt-get install ntp
```

Step 2 Edit ntp.conf

```
Replace these four with pubnub server

server 0.ubuntu.pool.ntp.org
server 1.ubuntu.pool.ntp.org
server 2.ubuntu.pool.ntp.org
server 3.ubuntu.pool.ntp.org
```

to

```
server 0.pubsub.pubnub.com
server 1.pubsub.pubnub.com
server 2.pubsub.pubnub.com
server 3.pubsub.pubnub.com
```

Step 3 Restart NTP Service

$ sudo service ntp restart

Ref :

[https://www.pubnub.com/docs/web-javascript/pam-security][1]

https://www.pubnub.com/docs/java/pubnub-java-sdk-v4

Read Access Manager online: https://riptutorial.com/pubnub/topic/330/access-manager

Chapter 3: Channel Specific Callbacks for v4 SDKs

Examples

JavaScript v4 SDK Channel Specific Callbacks

In PubNub JavaScript v3, you could implement a unique callback for every channel that you subscribed to as long as you called the subscribe function for each channel and implemented the callback in that subscribe like this:

```
var pubnub = new PubNub({
    publishKey: "your-pub-key",
    subscribeKey: "your-sub-key"
});

pubnub.subscribe({
    channel: 'ch1',
    message: function (m) {
        console.log(m + " ch1 callback");
    }
});

pubnub.subscribe({
    channel: 'ch2',
    message: function (m) {
        console.log(m + " ch2 callback: ");
    }
});
```

So in the above code, if you publish the message `"hello"` to `ch1`:

```
publish({channel: "ch1", message: "hello"});
```

...then you would get the output, `hello ch1 callback`. And of course if you published the same message to `ch2`, you would get the output `hello ch2 callback`.

The ability to provide a custom callback for each channel, or a group of channels, is useful and often more desirable than creating a monolithic callback with a long `if-then-else` of channel names and code to be executed for each condition. And there are better practices to use than the above simple example but I wanted to make it easy to compare and contrast with how PubNub JavaScript SDK v4 (and any v4 PubNub SDK) is designed.

PubNub v4 SDKs use a single listener to receive all `message`, `presence` and `status` events. This means that when you `subscribe` to a channel, you only provide the channel name(s) rather than a companion `message callback` function. In JavaScript SDK v4, the listener looks like this:

```
pubnub.addListener({
```

```
    message: function(m) {
        console.log(JSON.stringify(m));
    },
    presence: function(p) {
        console.log(JSON.stringify(p));
    },
    status: function(s) {
        console.log(JSON.stringify(s));
    }
});
```

I know many developers will be wondering how to migrate the `subscribe` code with unique `message callbacks` from SDK v3 with this sort of design without resorting to the never ending channel name conditional code that I mentioned above, like this:

```
pubnub.addListener({
    message: function(m) {
        if (m.subscribedChannel == 'ch1') {
            console.log(m.message + "ch1 callback");
        }
        else if (m.subscribedChannel == 'ch2') {
            console.log(m.message + "ch2 callback");
        }
        else {
            console.log(m.message + "default callback");
        }
    }
    // removed the other two callbacks for brevity purposes
});
```

The parameter `m` that is passed into the message listener above has the following structure which is a different design than the multiple parameter design of JavaScript SDK v3.

```
{
    "channel": "ch1",
    "subscription": <undefined>,
    "timetoken": "14721821326909151",
    "message": "hello"
}
```

That `timetoken` is the actual `publish timetoken`. For experienced PubNub developers, you should be excited to see that this is now available to the subscriber, but let's not get into why this is important and powerful right now.

Now I wouldn't expect any experienced JavaScript developer to write code as represented above and many advance developers might already know what to do. But for those developers that are at the beginner to intermediate level with JavaScript, the solution may not be immediately obvious. However, I know once you see this simple design approach below, it will open your eyes to the unlimited possibilities of the JavaScript language - here we go.

Let's restate the requirement:

> For every channel I subscribe to, I want to be able to provide a unique function to in invoke when a message is received on that channel. And I want to avoid the monolithic

conditional channel name approach.

So what we need to do first is create a function lookup table. A `hashtable` to be exact. This table will have channel names as keys and functions (the code to invoke when a message is received on that channel) as values. If you are somewhat new to JavaScript or have been coding with the language for awhile but haven't really dove into the language features yet, this might sound odd and impossible, but it's really how JavaScript works and you've been doing it all along and didn't really know it. Let's define our function lookup table:

```
ftbl = {};
```

That's it - you have an object that will hold your channels and functions. Pretty simple, right? But how do you add the channels and functions? Just like any other key/value.

```
ftbl.ch1 = function(m){console.log(m.message + " ch1 callback")};
ftbl.ch2 = function(m){console.log(m.message + " ch1 callback")};
```

...and so on with each channel and function you want to define. And you don't have to create all of your channel/function keys in one spot of your code. You can add each channel/function to the `ftbl` as you subscribe to a channel.

```
ftbl.ch10 = function(m){console.log(m.message + " ch10 callback")};
pubnub.pubnub.subscribe({
    channels: ['ch10']
});
```

OK, that's simple enough, and you can get fancier and more advanced with how you do this but just keeping it basic. But how do you invoke this function for the channel it is linked to? This is why JavaScript is so cool, powerful and easy especially if you come from a rigid and structured language like Java - check it out.

```
pubnub.addListener({
    message: function(m) {
        // use the channel name to get the function
        // from ftbl and invoke it
        ftbl[m.subscribedChannel](m);
    },
    presence: function(p) {
        console.log(JSON.stringify(p));
    },
    status: function(statusEvent) {
        console.log(JSON.stringify(s));
    }
});
```

That's all there is to it. Just get the function from `ftbl` using the channel name that is passed into the listener's `message` callback function and add `(m)` to the end of it and boom, it runs your function.

If the channel is `ch10`, `ftbl[m.subscribedChannel](m)` just invokes `function(m){console.log(m.message + "ch10 callback")}` passing in the `m` parameter which your function can parse and exploit as it needs to.

So calling the following `publish` function:

```
pubnub.publish(
    {
        channel : "ch10",
        message : "hello"
    },
    function(status, response) {
        console.log(status, response);
    }
);
```

...will result in the following message getting displayed: `hello ch10 callback`. And the equivalent for publishing to other channels that you have defined in your function lookup table. Don't forget to provide a default for unknown channels.

And don't forget the `presence` and `status` callbacks in the listener. This could be just two more function lookup tables or just a slightly more complex `ftbl`:

```
ftbl.message.ch1 = function(m){console.log(m.message + " ch1 message cb")};
ftbl.presence.ch1 = function(m){console.log(m.message + " ch1 presence cb")};
ftbl.status.ch1 = function(m){console.log(m.message + " ch1 status cb")};
```

or

```
ftbl.ch1.message = ...
ftbl.ch1.presence = ...
ftbl.ch1.status = ...
```

I like the former better than the latter but it's really up to you. And you probably want some generic `status` event handling code anyway but it will depend on your specific requirements.

And this can get even more complicated and robust with optional functions per channel that can be invoked depending on some additional data that you embed in the message payload.

So there you go, unique callbacks for each channel. No more excuses for not wanting to migrate from 3x to 4x. But if you do have some doubts about migrating, don't hesitate to reach out to PubNub Support and we'll get you moving forward. And don't forget to review the PubNub JavaScript SDK v3 to v4 Migration Guide.

Read Channel Specific Callbacks for v4 SDKs online:
https://riptutorial.com/pubnub/topic/6037/channel-specific-callbacks-for-v4-sdks

Chapter 4: Hello World

Remarks

`<init>` (i.e. `require ("pubnub")`)

Initialize an instance of PubNub to invoke operations.

Parameter	Details
publish_key	String - your publish key from your PubNub Admin Dashboard account
subscribe_key	String - your publish key from your PubNub Admin Dashboard account

`subscribe`

Subscribe to a channel(s) and provide a means to receive messages published to the channel(s).

Parameter	Details
channel	String - channel name or comma-delimited list of channel names
message	function - the callback function that will receive messages published on the subscribe channels
connect	function - the callback function that will be called when the subscription to the channels is successful

`publish`

Publish a message to a channel which will be received by subscribers on that channel.

Parameter	Details
channel	String - channel name on which to send the message
message	String - The message to publish on the channel. JSON format is recommended (do not stringify; use the JSON object)

Examples

Publish and Subscribe for Node.JS SDK

Publish and Subscribe for Node.JS

Install PubNub NPM Package.

```
npm install pubnub@3.15.2
```

Example Publish Subscribe with Node.JS

```
var channel = "hello_world";
var pubnub = require("pubnub")({
    publish_key   : "your_pub_key"
,   subscribe_key : "your_sub_key"
});

pubnub.subscribe({
    channel : channel
,   message : receive  // print message
,   connect : send     // send message after subscribe connected
});

function receive(message) { console.log(message) }

function send(message) {
    pubnub.publish({
        channel : channel
    ,   message : message
    });
}
```

Read Hello World online: https://riptutorial.com/pubnub/topic/329/hello-world

Chapter 5: Message Filtering

Remarks

Stream Filter provides the ability to filter messages on the server before they are sent to a subscriber is a popular request. With the introduction of our v4.x SDKs, you now have the ability to do so using message *meta data*.

Examples

Prevent Receiving Your Own Messages Using Objective-C

Setting a filter applies to all channels that you will subscribe to from that particular client. This client filter *excludes* messages that have this subscriber's UUID set at the sender's UUID:

```
NSString *expression = [NSString stringWithFormat:@"(uuid != '%@'",
                        self.client.currentConfiguration.uuid];
[self.client setFilterExpression:expression];
```

When publishing messages, you need to include the sender's UUID if you want the subscriber side client filter to work:

```
[self.client publish:@"message" toChannel:@"group-chat"
        withMetadata:@{@"uuid": self.client.currentConfiguration.uuid}
          completion:^(PNPublishStatus *status) {

    // Check whether request successfully completed or not.
    if (!status.isError) {

        // Message successfully published to specified channel.
    }
    else {

        // Request processing failed. Handle message publish error.
        // Check 'category' property to find out possible issue
        // publish can be attempted again using: [status retry];
    }
}];
```

See also:

- PubNub Objective-C SDK Stream Filter Documentation

Read Message Filtering online: https://riptutorial.com/pubnub/topic/593/message-filtering

Chapter 6: Presence

Examples

Presence in JavaScript

Presence works by sending messages when a user joins, leaves, or times out from a particular channel. You can listen for these messages to track who is in a channel, and how long since they did anything.

First, make sure each user as a UUID. Set this when you initialize PubNub:

```
var pubnub = PUBNUB({
     publish_key: 'my_pub_key',
     subscribe_key: 'my_sub_key',
     uuid: '1234_some_uuid'
 });
```

Now when you connect to a channel, add an extra listener for `join` events.

```
pubnub.subscribe({
    channel: "channel-1",
    message: function(m){console.log(m)}
    presence: onPresenceEvent,
});

onPresenceEvent = function(message, envelope, channel){
    if (!message.action) {
        // presence interval mode happens
        // when occupancy > presence announce max
        // there is no action key
        console.log("Presence Interval Mode: occupancy = " + m.occupancy);
        return;
    }

    console.log(
        "Action:    " + message.action + "\n" +
        "UUID:      " + message.uuid + "\n" +
        "Channel:   " + JSON.stringify(channel) + "\n" +
        "Occupancy: " + message.occupancy + "\n" +
        "Timestamp: " + message.timestamp);

    else if (m.action == 'join') {
        // new subscriber to channel
        // add the user to your buddy list
    }
    else if (m.action == 'leave') {
        // subscriber explicitly unsubscribed channel
        // remove user from your buddy list
    }
    else if (m.action == 'timeout') {
        // subscriber implicitly unsubscribed channel (did not unsubscribe)
        // remove user from your buddy list
    }
```

```
    else if (m.action == 'state-change') {
        // subscriber changed state
        // update the attributes about the user in the buddy list
        // i.e. - is typing, online status, etc.
        console.log("State Data: " + JSON.stringify(message.data));
    }
};
```

The message object sent to the `presence` callback will include the action taken (join, leave, timeout or state-change) and the UUID of the user who did the action, as well as timestamp and some other metadata.

When state is set, a `state-change` event is sent which will include the new state in the `data` key of the `message` key.

Note: If you have the Access Manager enabled, you *must* ensure that your grants cover both the regular channel as well as the presence channel. Otherwise, when you attempt to subscribe to the channel with a presence callback, the SDK will also subscribe you to the presence channel which will fail if you have not applied the grants. The presence channel name is the regular channel name with a "-pnpres" suffix; meaning a channel named "pubnub-sensor-array" will have a presence channel named "pubnub-sensor-array-pnpres". See the Access Manager examples for more information.

Setting State Upon Subscribe

When a user subscribes to a channel, you may want to set state for that newly subscribed user. While there is a *subscribe with state* API, there are some scenarios where this is not the most optimal/reliable technique (like during a disconnect/reconnect situation - the state will be lost and not reinstated).

It is better to explicitly set state once the channel is successfully subscribed. This means you use the `subscribe`'s `connect` callback to set the state.

```
var pubnub = PUBNUB({
    publish_key: 'my_pub_key',
    subscribe_key: 'my_sub_key',
    uuid: 'users_uuid'
});

pubnub.subscribe({
    channel: 'channel-1',
    message: function(msg, env, ch){console.log(msg)},
    connect: function(m) {
        console.log('CONNECT: ' + m);
        pubnub.state({
            channel  : 'channel-1', // use the channel param from the subscribe
            state    : {'nickname': 'Bandit', 'mood': 'Pumped!'},
            callback : function(m){console.log(m)},
            error    : function(m){console.log(m)}
        });
    },
    disconnect : function(m){console.log('DISCONNECT: ' + m)},
    reconnect  : function(m){console.log('RECONNECT: ' + m)},
```

```
    error      : function(m){console.log('CONNECT: ' + m)}
});
```

Read Presence online: https://riptutorial.com/pubnub/topic/331/presence

Chapter 7: Storage & Playback

Parameters

Parameter	`Type` / `Required` / `Default` Description
channel	`String` / `Yes` Specifies channel to return history messages from.
reverse	`Boolean` / `No` / `false` Setting to true will traverse the time line in reverse starting with the oldest message first. Default is false. If both start and end arguments are provided, reverse is ignored and messages are returned starting with the newest message.
limit	`Number` / `No` / `100` Specifies the number of historical messages to return.
start	`Number` / `No` Time token delimiting the start of time slice (exclusive) to pull messages from.
end	`Number` / `No` Time token delimiting the end of time slice (inclusive) to pull messages from.
includeTimetoken	`Boolean` / `No` / `false` If true the message post timestamps will be included in the history response.

Remarks

Simply enabling *Storage & Playback* add-on for your keys in the PubNub Admin Dashboard will result in all published messages on all channels to be stored. You can prevent a message from being stored by passing the `storeInHistory` parameter as `false` when the message is published, like this:

```
pubnub.publish(
    {
        message: {
            'price': 8.07
        },
        channel: 'channel1',
        storeInHistory: false // override default storage options
    },
    function (status, response) {
        // log status & response to browser console
        console.log("STATUS  : " + console.log(JSON.stringify(status));
        console.log("RESPONSE: " + console.log(JSON.stringify(response));
    }
);
```

Otherwise, just omit the `storeInHistory` or set to `true` to store the message when it is published.

Examples

JavaScript SDK v4 - History

```
// initialize pubnub object
var pubnub = new PubNub({
    subscribeKey: "yourSubscribeKey",
    publishKey: "myPublishKey" // optional
})

// get up to the last 100 messages
// published to the channel
pubnub.history(
    {
        channel: 'channel1'
    },
    function (status, response) {
        // log status & response to browser console
        console.log("STATUS  : " + console.log(JSON.stringify(status));
        console.log("RESPONSE: " + console.log(JSON.stringify(response));
    }
);

// this is the format of the response
[
    [array of returned messages],
    "firstMessageTimetoken",
    "lastMessageTimetoken"
]

// example of response
[
    [{'price':10.02}, {'price':10.12}, {'price':10.08}, {'price':10.10}],
    "14691304969408991",
    "14691307326690522"
]
```

Read Storage & Playback online: https://riptutorial.com/pubnub/topic/3714/storage---playback

Chapter 8: Stream Controller: Channel Groups

Remarks

When using Channel Groups, you should not add or remove channels in your client side applications. This example shows adding channels to a channel group and subscribing to that channel group for simplicity sake. But in a real world scenario, you should have your server do all the add/remove of channels to/from channel groups. When you enable Access Manager, you will need the `manage` permission to add/remove channels to/from channel groups and you should never grant the `manage` permission to clients for security reasons. Only your server should be granted the `manage` permission.

Examples

Subscribe with Channel Groups in JavaScript

With Stream Controller add-on enabled, you can use Channel Groups to subscribe to a 1000's of channels from a single client. You do this by creating a channel group and adding channels to the channel group. We'll assume `pubnub` variable has been initialized properly with your keys.

Create a generic callback handler function:

```
function displayCallback(m,e,c,d,f){
    console.log(JSON.stringify(m, null, 4));
}
```

Create channel group and add channels to it:

```
pubnub.channel_group_add_channel({
  callback: displayCallback,
  error: displayCallback,
  channel_group: "sports",
  channel: "football,baseball,basketball,lacrosse,cricket"
});
```

Now, subscribe to the channel group and you will be subscribed to all channels in that group:

```
pubnub.subscribe({
  callback: displayCallback,
  error: displayCallback,
  channel_group: "sports"
});
```

Any messages published to the channels in the channel group will be received in the `displayCallback` function.

Read Stream Controller: Channel Groups online: https://riptutorial.com/pubnub/topic/580/stream-controller--channel-groups

Chapter 9: UUIDs

Examples

JavaScript/Web SDK

For JavaScript, here is the code we recommend for generating, persisting and retrieving a UUID. This could be wrapped in a function can called directly from the PUBNUB.init function rather than the two step inline solution below.

```
// get/create/store UUID
var UUID = PUBNUB.db.get('session') || (function(){
    var uuid = PUBNUB.uuid();
    PUBNUB.db.set('session', uuid);
    return uuid;
})();

// init PUBNUB object with UUID value
var pubnub = PUBNUB.init({
    publish_key: pubKey,
    subscribe_key: subKey,
    uuid: UUID
});
```

Android/Java SDK

For Android, here is the code we recommend for generating, persisting and retrieving a UUID. There is not constructor that accepts the UUID as a parameter, so you must instantiate `Pubnub` object first then use the setter to provide the UUID.

```
// creating the Pubnub connection object with minimal args
Pubnub pubnub = new Pubnub(pubKey, subKey);

// get the SharedPreferences object using private mode
// so that this uuid is only used/updated by this app
SharedPreferences sharedPrefs = getActivity().getPreferences(Context.MODE_PRIVATE);

// get the current pn_uuid value (first time, it will be null)
String uuid = getResources().getString(R.string.pn_uuid);

// if uuid hasn't been created & persisted, then create
// and persist to use for subsequent app loads/connections
if (uuid == null || uuid.length == 0) {
    // PubNub provides a uuid generator method but you could
    // use your own custom uuid, if required
    uuid = pubnub.uuid();
    SharedPreferences.Editor editor = sharedPrefs.edit();
    editor.putString(getString(R.string.pn_uuid), uuid);
    editor.commit();
}

// set the uuid for the pubnub object
pubnub.setUUID(uuid);
```

Read UUIDs online: https://riptutorial.com/pubnub/topic/561/uuids

Chapter 10: webhook

Examples

pubnub webhook

```
Webhook Overview
```

A WebHook is an HTTP callback: an HTTP POST that occurs when something happens; a simple event-notification via HTTP POST. A web application implementing WebHooks will POST a message to a URL when certain things happen.

```
PubNub Presence
```

Pubnub presence all about the user presence at pubnub platform. it provides the presence of user when they are joining, leaving a channel or when there is a user's state changes. Presence Webhooks provide a means for your server to be notified whenever presence events occur on any channel for your keys. This provides an easy to scale solution for your server side application to monitor the presence events.

```
How it would reduce the overhead
```

Without Presence Webhooks, your server would have to subscribe to all the channels' -pnpres channels. So this can be a tedious task to control overs channels if your app has thousands of channels or more.

Pubnub Webhooks would help us in this scenario and it is easier to implement and scale with traditional, well-known web infrastructure(load balancers, web and app servers provided by your application service providers like Heroku, Rackspace, Azure, Amazon and others).

```
PubNub Presence Webhooks
```

PubNub Presence Webhooks are a means for the PubNub network to invoke a REST endpoint your server directly as presence events occur. It would also help in load balancing. So here you need to create REST Endpoint URL of your server at which pubnub would sends the presence data.

```
User Presence Events
```

There are four user events at pubnunb platform

1. join
2. leave
3. timeout
4. state-change

And two channel level events : active and inactive. please refer pubnub-doc for detailed information.

Each event has it's own Webhook for which you can provide a REST endpoint to your server to handle the event. Or you can provide one REST endpoint for all of them and just implement conditional logic on the action attribute on your server to handle each individual event.

Whatever you choose, you need to provide the sub-key and the REST URIs to PubNub Support to configure this for you. You likely will have more than one sub-key with different endpoints for different server environments (dev, test, production, for example).

Once your server's REST endpoints are implemented and the PubNub key configuration is in place, you are ready to go. But before you implement the REST endpoints, it might be helpful to know what the events' data looks like.

Here is an example of a join :

```
HTTP POST
Content-Type: application/json

{
    'action': 'join',
    'sub_key': 'sub-c-...',
    'channel': 'lacrosse'
    'uuid': '1234-5678-90ab-cdef',
    'timestamp': 1440568311,
    'occupancy': 1,
    'data': {'foo': 'bar'}
}
```

This would be the same for leave and timeout and state-change except for the action value, of course.

```
Webhook Response Status
```

It is important that your REST endpoint implementation should return a status code (200 OK) immediately upon receiving the Webhook from PubNub.

```
Pubnub Re-try
```

If pubnub does not recieve 200 From Rest-Endpoint then it will send duplicate events because PubNub assumes no response means your server did not receive the event. PubNub will wait five seconds for the 200 response before trying again. After a third retry (four total attempts), PubNub will no longer attempt to send that particular event to your server.

Read webhook online: https://riptutorial.com/pubnub/topic/3715/webhook

Credits

S. No	Chapters	Contributors
1	Getting started with PubNub	Community, Craig Conover, Dara Kong, Josh Marinacci
2	Access Manager	Craig Conover, Girish Kumar, Josh Marinacci
3	Channel Specific Callbacks for v4 SDKs	Craig Conover
4	Hello World	Craig Conover, girlie_mac, PubNub
5	Message Filtering	Craig Conover, Serhii Mamontov
6	Presence	Alex Vidal, Craig Conover, Dara Kong, Josh Marinacci
7	Storage & Playback	Craig Conover
8	Stream Controller: Channel Groups	Craig Conover
9	UUIDs	Craig Conover
10	webhook	Girish Kumar