Apple Push Notification Services in iOS 6 Tutorial: Part 2/2

Ali Hafizji on May 23, 2013

Hi, what's up?
SteveJ @ Today 10:06 PM
I just wrote this cool app that uses push notifications!
You @ Today 10:07 PM
I know, I'm using it right now to talk to you. :-)
SteveJ @ Today 10:08 PM

Create a simple chat app with Apple Push Notification Services!

Update 4/12/2013: Fully updated for iOS 6 (original post byMatthijs Hollemans, update by Ali Hafizji).

This is the second part of a 2-part tutorial series on integrating Apple Push Notification Services (APNS) into an iPhone app.

In the first part of the tutorial series, you learned how to enable your iPhone app to receive push notifications, and how to send a test push notification using a PHP script.

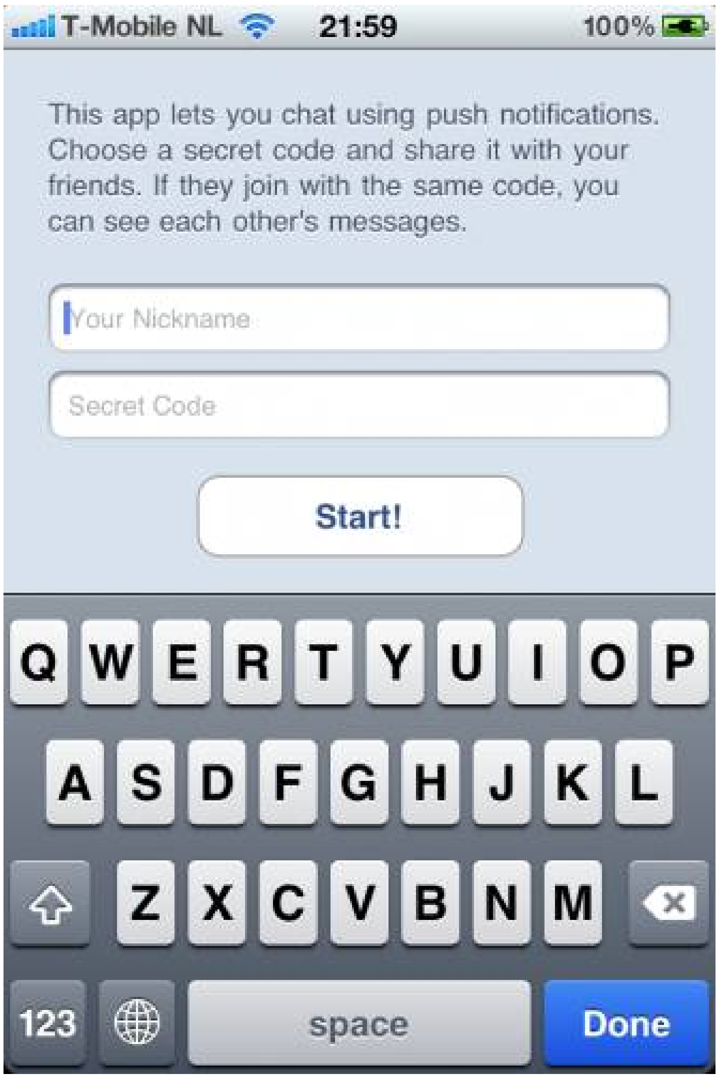
In this second and final part of the tutorial series, you'll learn how to make a simple app using APNS, and a simple PHP web service to power it!

Note: This tutorial is on the long side, so make sure you set aside a nice chunk of time (and some munchies!) to go through it. But it's well worth it – once you finish you'll have a functional app and a web service using push notifications!

Getting Started: Introducing PushChat

In this tutorial, you're going to make a simple direct messaging app named

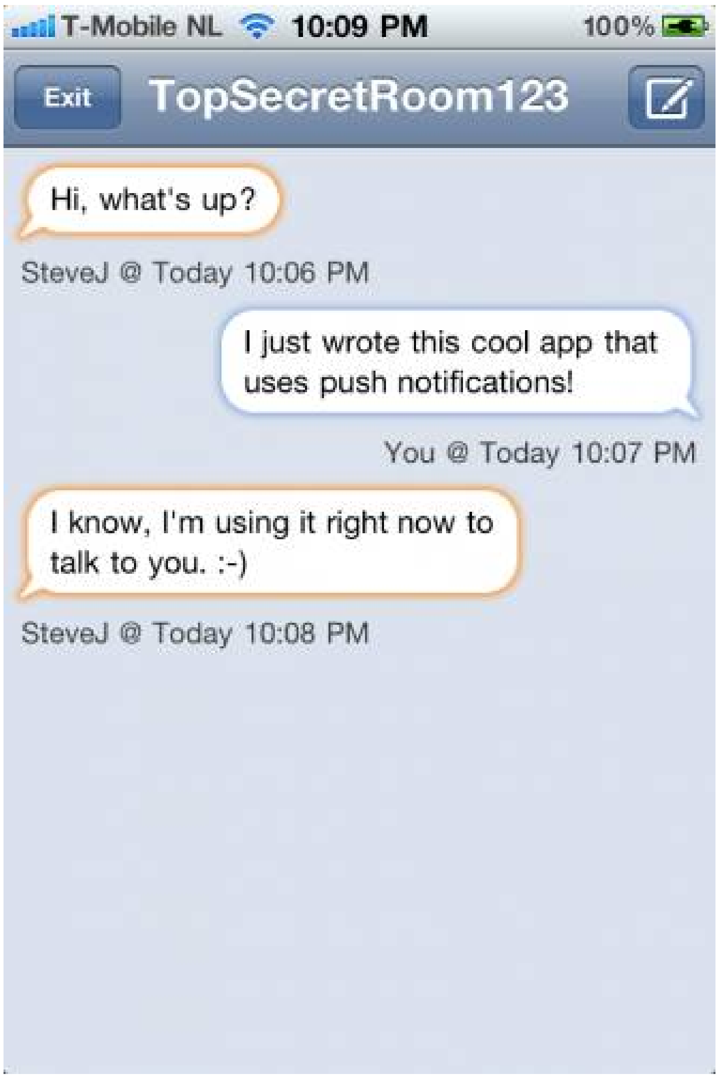
PushChat that uses push notifications to deliver the messages. This is what it will look like:


T-Mobile NL
21:59
100%
This app lets you chat using push notifications. Choose a secret code and share it with your friends. If they join with the same code, you can see each other's messages.
Your Nickname
Secret Code
Start!
space
Done

The first screen that a user sees is the Login screen. Here the user enters their nickname and a “secret code”. The idea is that the user shares this secret code with his/her friends.

Everyone using the same secret code will see each other’s messages. So in effect, the secret code functions as the name of a chat room. Of course, if you can guess someone’s code you can see their messages, that’s why it’s supposed to be a **secret** ;-)

When the user presses the Start! button, the app sends a message to your server to register this user for that chat room. The Login screen then makes way for the Chat screen:



Hi, what's up?

SteveJ @ Today 10:06 PM

I just wrote this cool app that uses push notifications!

You @ Today 10:07 PM

I know, I'm using it right now to talk to you. :-)

SteveJ @ Today 10:08 PM

The secret code is shown in the navigation bar. Messages that are sent by this user are displayed on the right-hand side of the screen, messages that you received are shown on the left. So in this example, both the user and someone named SteveJ (I wonder who that is?) signed up with secret code "TopSecretRoom123".

The third and final screen of the app is the Compose screen:

Nothing special going on here. It's just a text view with a keyboard. Messages are limited to 190 bytes and the top of the screen shows how many are left.

I put in that limitation because the payload of a push notification has a maximum size of 256 bytes, and that includes the JSON overhead (see "Anatomy of a push notification" in part 1 of this tutorial series).

When the user presses the Save button, the message is sent to your server and pushed to all other users who signed up with the same secret code.

The Server API

In the above explanation of how PushChat works, I mentioned "server" several times. You need to use a server because you have to coordinate stuff between different devices. It’s not much of a chat when you’re just talking to yourself!

I wrote a simple web service that uses PHP and MySQL. The iPhone app will send the following commands to this server:

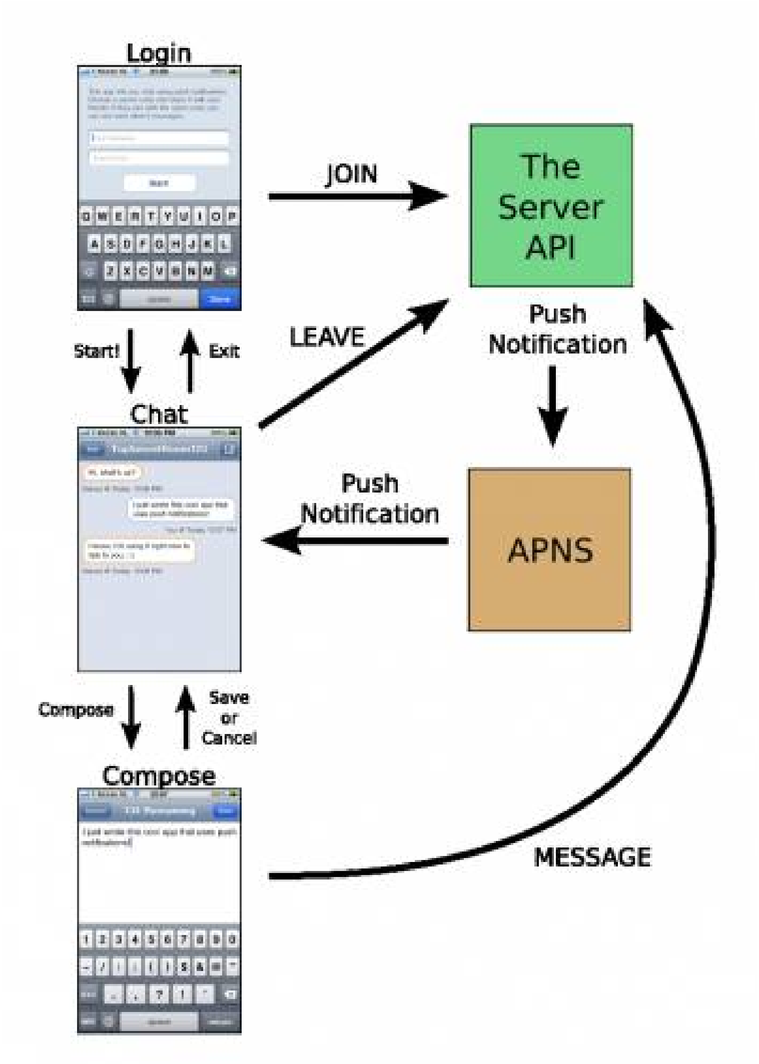
JOIN. When a user signs up in the Login screen, you send his nickname, his secret code, and his device token to the server. The server adds a record for this user to the list of active users. From then on, any messages sent by members of that same chat room will also be pushed to this new user.

LEAVE. This is the opposite of JOIN. The user can press the Exit button in the Chat screen to leave the chat. A LEAVE command is sent to the server which removes the user from the list of active users. From then on, no more notifications will be pushed to this user.

MESSAGE. When a user presses the Save button on the Compose screen, you will send the new text message to the server. The server will then convert these messages into push notifications, one for each member of that chat room, and passes them on to APNS. A few seconds later, APNS pushes these messages to those devices.

UPDATE. This lets the server know that a user has received a new device token. The token may change from time to time and if that happens you need to let the server know. More about this later.

In a picture it looks like this:



At various times the app sends data to the server to let it know that the user has joined or left a chat, or that he/she is sending a new message. The server in turn creates push notifications that it sends to APNS.

The Apple servers are responsible for delivering those notifications to the app. When the app receives a new push notification, it adds a speech bubble to the Chat screen with the contents of that message.

Setting Up the Server

If you're new to web services, you may want to check outRay's tutorial on PHP and MySQL first.

This tutorial uses MAMP to set up a server and database. This is an excellent solution for when your app is still in the development stage. MAMP is very easy to set up and you don't need to pay for a separate server.

The only requirement is that your Mac and your iPhone share the same local network, otherwise the app will not be able to communicate with the server. Most people will have WiFi at home, so this shouldn't be too much of a problem.

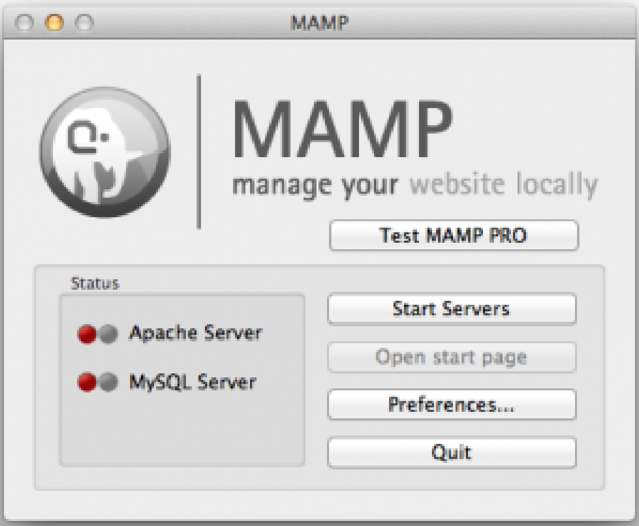
Of course, once your app is ready for submission to the App Store, you should set up a real server that is connected to the internet. ;-)

You can download MAMP for free. (There is also a paid PRO version, but the free one is good enough for our purposes.)

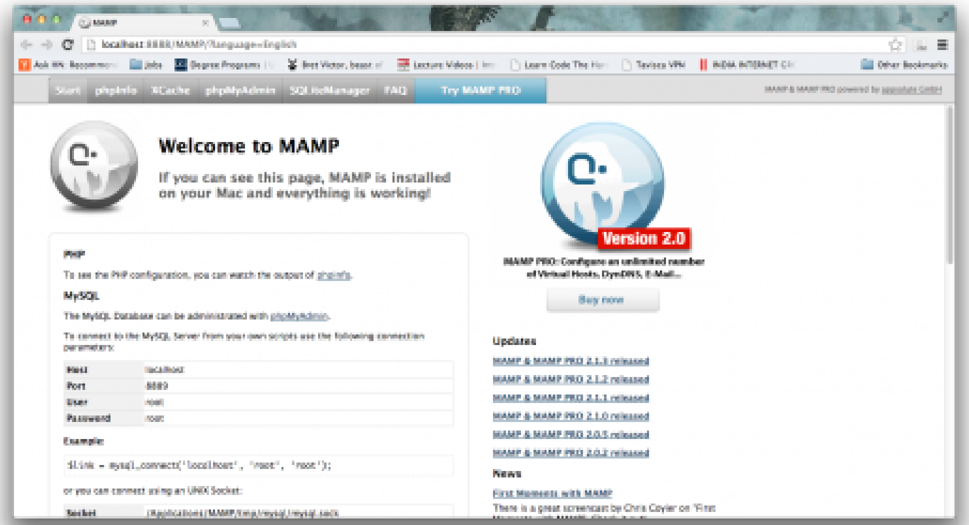
MAMP contains the Apache web server, the PHP scripting language, and the MySQL database. You will be using all three of these. (If you want to use the server code from this tutorial on a PHP installation other than MAMP, you will have to make sure the following extensions are installed and enabled: PDO, pdo_mysql, mbstring, OpenSSL.)

Installing MAMP is easy. Unzip the download and open the DMG file. Accept the license agreement and drag the MAMP folder into your Applications folder. Done!

To start MAMP, go to **Applications/MAMP** and click on the MAMP icon (the one with the elephant). This should open the MAMP window:



Click the **Start Servers**, this will start the Apache server and the MySQL server. Once both the light turn to green select the **Open start page** button. Your favorite web browser should open and you should see a welcome page:



Welcome to MAMP

If you can see this page, MAMP is installed on your Mac and everything is working!

PHP

To see the PHP configuration, you can watch the output of phpinfo.

MySQL

The MySQL Database can be administrated with phpMyAdmin.

To connect to the MySQL Server from your own scripts use the following connection parameters:

Host	localhost
Port	8889
User	root
Password	root

Example:

```
$link = mysql_connect('localhost', 'root', 'root');
```

or you can connect using an UNIX Socket:

Socket	/Applications/MAMP/tmp/mysql/mysql.sock

MAMP PRO: Configure an unlimited number of Virtual Hosts, DynDNS, E-Mail...

Updates

MAMP & MAMP PRO 2.1.3 released
MAMP & MAMP PRO 2.1.2 released
MAMP & MAMP PRO 2.1.1 released
MAMP & MAMP PRO 2.1.0 released
MAMP & MAMP PRO 2.0.5 released
MAMP & MAMP PRO 2.0.2 released

News

First Moments with MAMP

There is a great screencast by Chris Coyier on "First

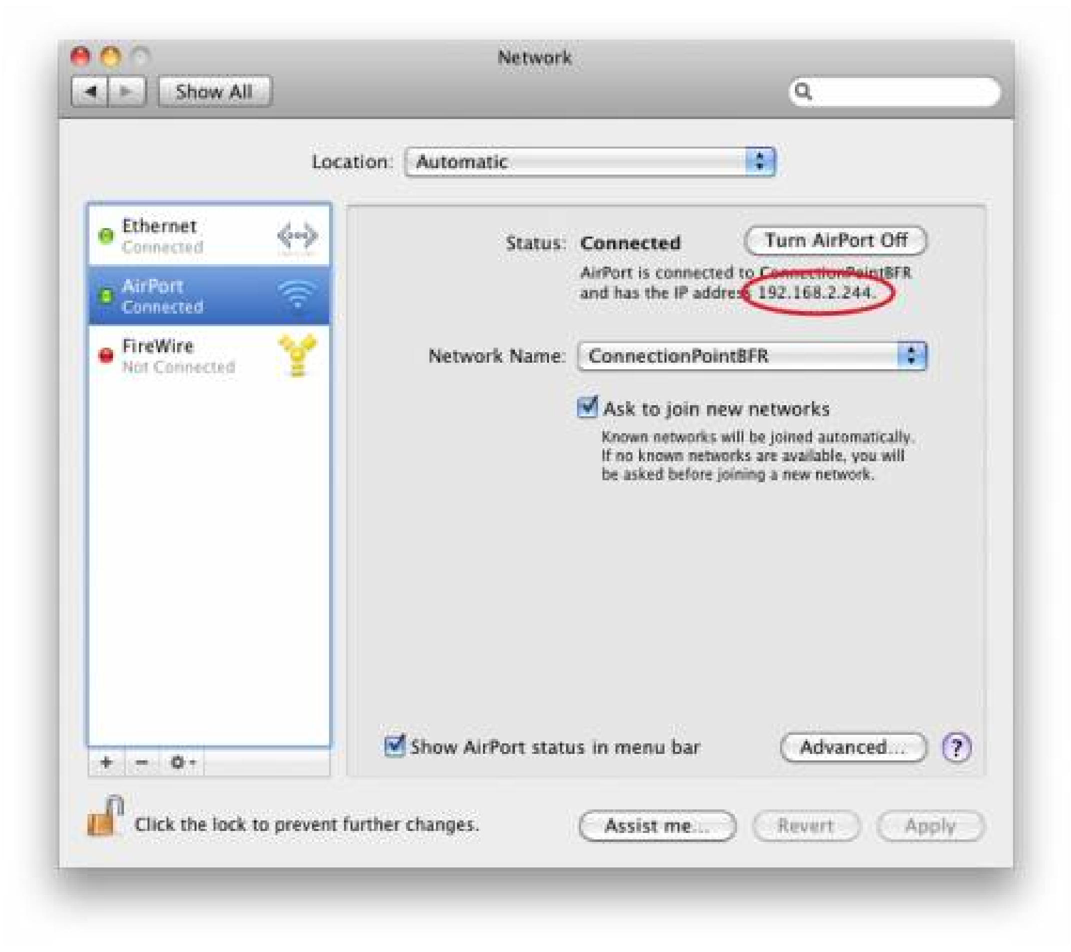
Great! Now download the PushChatServer code and unzip it. I'm going to assume you unzipped it on your Desktop. The reason I say this is because you need to put the path to these files into the configuration for the Apache web server.

Open **Applications/MAMP/conf/apache/httpd.conf** and add the following lines:

```
Listen 44447

<VirtualHost *:44447>
        DocumentRoot
"/Users/matthijs/Desktop/PushChatServer/api"
        ServerName 192.168.2.244:44447
        ServerAlias pushchat.local
        CustomLog
"/Users/matthijs/Desktop/PushChatServer/log/apache_access.log
combined
        ErrorLog
"/Users/matthijs/Desktop/PushChatServer/log/apache_error.log"

        SetEnv APPLICATION_ENV development
        php_flag magic_quotes_gpc off

        <Directory
"/Users/matthijs/Desktop/PushChatServer/api">
                Options Indexes MultiViews FollowSymLinks
                AllowOverride All
                Order allow,deny
                Allow from all
        </Directory>
</VirtualHost>
```

There are a few lines you will need to change. The path to where I installed the PushChatServer files is "/Users/matthijs/Desktop". Change those to where you unzipped the files.

Change the IP address in the "ServerName" line to the IP address of your Mac. If you don't know how to find the IP address, open System Preferences and go to the Network panel.



Status: Connected

AirPort is connected to ConnectionPointBFR and has the IP address 192.168.2.244.

Network Name: ConnectionPointBFR

✓ Ask to join new networks

Known networks will be joined automatically. If no known networks are available, you will be asked before joining a new network.

✓ Show AirPort status in menu bar

I used the IP address of my MacBook's WiFi but the Ethernet port's IP address would have been fine too. Note that you should still keep the port number 44447 in ServerName:

```
ServerName <your IP address>:44447
```

The web service runs on port 44447. This is an arbitrary number. Web sites usually run on port 80 and the default MAMP web page runs on port 8888. I just picked something that wouldn't conflict with either of these.

I also defined **pushchat.local** as an alias for the server name. This is like a domain name except that it only works on the local network. You have to bind that name to an IP address. The easiest way to do this is to edit the file**/etc/hosts**. Add the following line to the bottom of this file and save it:

```
127.0.0.1        pushchat.local
```

In the MAMP window, click Stop Servers. When the lights have turned red, click Start Servers. If your changes to httpd.conf contain no errors, then both server lights should turn green again.

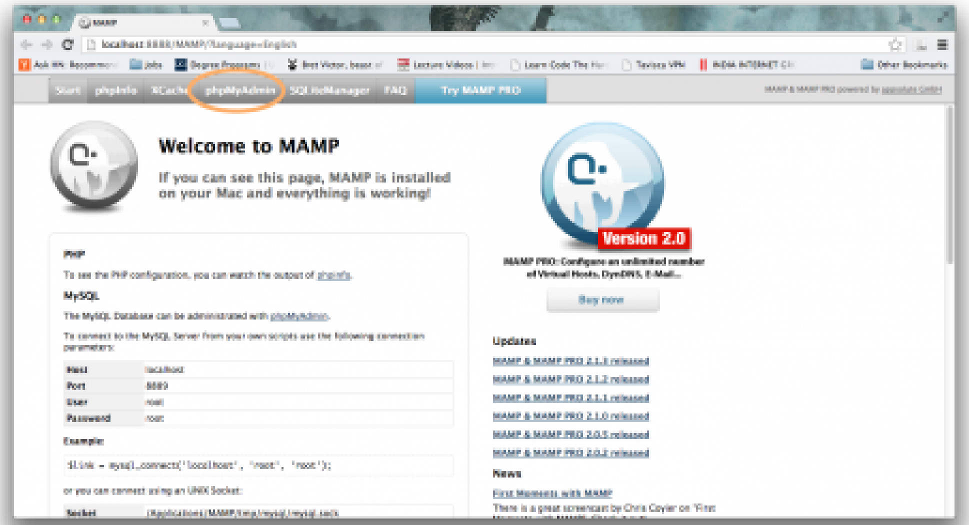
Now point your favorite web browser tohttp://pushchat.local:44447. You should see the message:

```
If you can see this, it works!
```

Awesome. This means Apache and the PHP code for the server API were installed properly. Now you have to configure the database.

Setting up The Database

Return to the MAMP start page (you can do this by clicking Open start page in the MAMP window) and click the**phpMyAdmin** button at the top.



Welcome to MAMP

If you can see this page, MAMP is installed on your Mac and everything is working!

PHP

To see the PHP configuration, you can watch the output of phpinfo.

MySQL

The MySQL Database can be administrated with phpMyAdmin.

To connect to the MySQL Server from your own scripts use the following connection parameters:

Host	localhost
Port	8889
User	root
Password	root

Example:

```
$link = mysql_connect('localhost', 'root', 'root');
```

or you can connect using an UNIX Socket:

Socket	/Applications/MAMP/tmp/mysql/mysql.sock

MAMP PRO: Configure an unlimited number of Virtual Hosts, DynDNS, E-Mail...

Updates

MAMP & MAMP PRO 2.1.3 released
MAMP & MAMP PRO 2.1.2 released
MAMP & MAMP PRO 2.1.1 released
MAMP & MAMP PRO 2.1.0 released
MAMP & MAMP PRO 2.0.5 released
MAMP & MAMP PRO 2.0.2 released

News

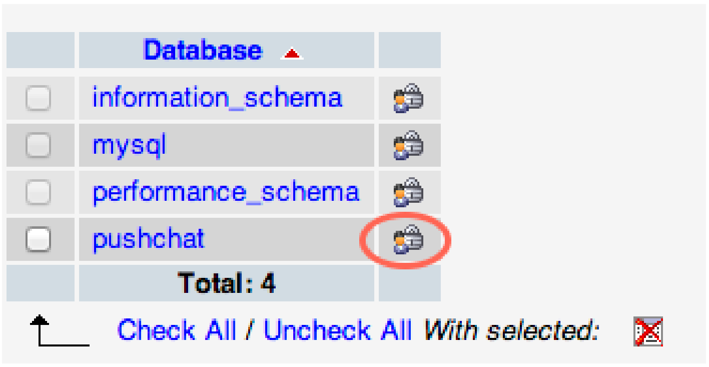
First Moments with MAMP

There is a great screencast by Chris Coyier on "First

You will be presented with a screen that looks like the one below:

Select the **Databases** tab and type **pushchat** in the "Create Database" field and choose "utf8_general_ci" for Collation. Then click **Create** to create the database. You should see a message that the database was successfully created.

Press the user privileges button on the database as shown below and select **add user**.



	Database ▲	
☐	information_schema	
☐	mysql	
☐	performance_schema	
☐	pushchat	
	Total: 4	

Check All / Uncheck All *With selected:*

I filled in the fields as follows:

- **User name**: pushchat
- **Host**: localhost
- **Password**: d]682\#%yI1nb3
- **Privileges**: Check "Grant all privileges on database "pushchat""

Click the **Add User** button to add the user. It's OK if you choose another password but then you will have to remember to update the password in the various PHP scripts as well.

That's the database and the user. Now you need to add the tables to the database. Click the **SQL** tab at the top of the screen and paste the following into the text box:

```
USE pushchat;

SET NAMES utf8;

DROP TABLE IF EXISTS active_users;

CREATE TABLE active_users
(
        user_id varchar(40) NOT NULL PRIMARY KEY,
        device_token varchar(64) NOT NULL,
        nickname varchar(255) NOT NULL,
        secret_code varchar(255) NOT NULL,
        ip_address varchar(32) NOT NULL
)
ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

(You can also find these statements in the file PushChatServer/database/api.sql.)

Click the Go button to execute these statements. This adds a single table, active_users, to the database. When the server API receives a JOIN command, it will add a new record for that user to this table. Conversely, when a LEAVE command is issued, it will remove the user's record from this table.

There is another table you need to add. Repeat the above procedure with these statements:

```
USE pushchat;

SET NAMES utf8;

DROP TABLE IF EXISTS push_queue;

CREATE TABLE push_queue
(
        message_id integer NOT NULL AUTO_INCREMENT,
        device_token varchar(64) NOT NULL,
        payload varchar(256) NOT NULL,
        time_queued datetime NOT NULL,
        time_sent datetime,
        PRIMARY KEY (message_id)
)
ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

(You can find these statements in the file PushChatServer/database/push.sql.)

Whenever the server receives a MESSAGE command, it adds the push notifications to this table.

Configuring the Server API

I don't have much room in this tutorial to describe how the server API works but I have extensively commented the**api.php** source, so even if you know only little PHP you should be able to follow what is going on. So please take a peek!

Of particular interest to us at this point is the file**api_config.php**. This file contains the configuration options for the server API.

Currently there are two sets of configuration options, one for development mode and one for production mode. I find it useful to have two sets of settings that you can easily switch between.

But how do you tell the API whether it runs in development mode or production mode? The answer is in the Apache VirtualHost configuration. Recall that you added a <VirtualHost> block to httpd.conf:

```
<VirtualHost *:44447>
        …

        SetEnv APPLICATION_ENV development

        ...
</VirtualHost>
```

The directive "SetEnv APPLICATION_ENV development" creates an environment variable that determines in which mode the script runs. If you want to switch to the production configuration, either remove this line or change the word "development" to "production".

If you've used the same value for the database name, username and password that I have, then you shouldn't have to change anything in api_config.php. However, if you chose different values then you'll have to enter them here, otherwise the API won't be able to connect to the database.

Surf with your favorite browser to:http://pushchat.local:44447/test/database.php. You should see the following message:

```
Database connection successful!
```

Tip: If you get unexplained errors, then check the PHP error log in **Applications/MAMP/logs/php_error.log** and the Apache log in **PushChatServer/log/apache_error.log**. These log files may provide useful hints as to what is wrong.

That should complete the setup for the server API.

Let's Do Some Programming Already!

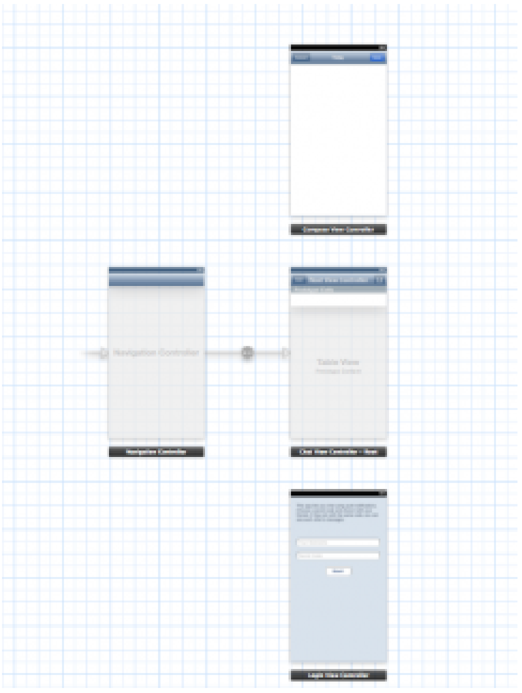
Download the PushChatStarter code and unzip it. This is the PushChat app without the networking and push code. I will quickly explain how the app works and then you'll add in the code that lets it talk to the server and receive push notifications. After all, that's what you came here for. ;-)

You can run PushChat in the simulator for now because it doesn't have any push functionality yet.

The project uses Cocoapods so instead of opening the .xcodeproj file open the .xcworkspace file in Xcode and go to the Target Settings. You have to change the bundle ID from "com.hollance.PushChat" to your own bundle ID that you chose all the way back when you made the App ID in the iOS Dev Portal (com.yoursite.PushChat). This is important because the app needs to be signed with your provisioning profile.

Build & Run the app and play with it for a bit. Because this new app uses the same bundle ID as the very basic test app that you made earlier, it's a good idea to uninstall that app before you Build & Run this one, just to make sure your iPhone doesn't get confused.

Open up **MainStoryboard.storyboard** to take a peek at how the app is structured:



The entry point to the app is a navigation controller that has the ChatViewController at its root. That view controller shows the sent and received messages. As you can see in the storyboard, it's a regular UITableView controller.

The **speech bubbles** are drawn by **MessageTableViewCell**, which is a subclass of UITableViewCell, and SpeechBubbleView, which is a subclass of UIView that does some custom drawing. This is plain old table view programming and you've no doubt seen it before, so I won't waste any time on explaining how this works.

The other two screens of the app, the **Login screen and the Compose screen**, are modal view controllers that are displayed on top of the ChatViewController. You can probably guess, but the Login screen is implemented in LoginViewController and the Compose screen in ComposeViewController. Both these view controllers have ids in the storyboard so that you can refer to them while initialisation. These are very simple screens; check out the source code for more details.

That leaves the **two data model classes**, DataModel and Message. Message describes the content of a single message that was either sent by this user or received from another user. Each Message has a sender name, a date, and the actual message text.

The DataModel class is responsible for the list of these Message objects. When a new message is sent or received, DataModel adds it to the list and then saves this list to a file in the app's Documents directory. When the app starts up, DataModel loads the messages from this file.

That concludes our very brief overview of this app. I suggest you play a little with the source code before continuing. It has plenty of comments that should explain how everything works.

Talking With the Server

You will now go through the various screens in the app and add the code that lets it communicate with the server. The server expects the app to send HTTP POST requests, so I am going to use AFNetworking to send data to the server. All the required classes have already been added to the PushChat starter code, so there is no need to install more stuff.

If you're new to using web services from iOS apps, I suggest you take a small detour and read Ray's tutorial on this topic.

Add the following line to **defs.h**:

```
#define ServerApiURL @"http://192.168.2.244:44447"
```

This is the URL of the server. You should change the IP address to point to your server. Also make sure that MAMP is currently active, so that the server is actually available, and that your iPhone is on the same network as the server.

The Login screen seems like a good place to start. When the user presses the Start! Button, you will send a JOIN command to the server. This lets the server know that there is a new user. The server will then add this user to its active_users table.

To do this first open the **ChatViewController.m** file and add the following:

```
//add above the implementation
@interface ChatViewController() {
    AFHTTPClient *_client;
}
@end
```

In the -initWithCoder: method add the following line below the call to -loadMessages

```
_client = [AFHTTPClient clientWithBaseURL:[NSURL
URLWithString:ServerApiURL]];
```

This instantiates a AFHTTPClient object which has a base URL of the server you setup.

Note: The reason you're creating the AFHTTPClient in the ChatViewController class is because the other 2 view controllers are going to launch modally. Since the ChatViewController will launch these modal view controller it will be the one holding the AFHTTPClient object.

Next open up **LoginViewController.h** and add a property to it.

```
@property (nonatomic, strong) AFHTTPClient *client;
```

Now you need to pass the AFHTTPClient from the ChatViewController to the LoginViewController, to do this open **ChatViewController.m** and modify the -showLoginViewController method as below:

```
- (void) showLoginViewController {
    LoginViewController *loginController = (LoginViewControll
*) [ApplicationDelegate.storyBoard
instantiateViewControllerWithIdentifier:@"LoginViewController
    loginController.dataModel = _dataModel;
    loginController.client = _client;
    [self presentViewController:loginController animated:YES
completion:nil];
}
```

Add the following chunk of code between -userDidJoin and -loginAction methods of LoginViewController.m:

```
- (void)postJoinRequest {
        MBProgressHUD *hud = [MBProgressHUD
showHUDAddedTo:self.view animated:YES];
        hud.labelText = NSLocalizedString(@"Connecting",
nil);
```

```
        NSDictionary *params = @{@"cmd":@"join",
                                 @"user_id":[_dataModel
userId],
                                 @"token":[_dataModel
deviceToken],
                                 @"name":[_dataModel
nickname],
                                 @"code":[_dataModel
secretCode]};

        [_client postPath:@"/api.php"
                                parameters:params

success:^(AFHTTPRequestOperation *operation, id
responseObject) {

        if ([self isViewLoaded]) {
            [MBProgressHUD hideHUDForView:self.view
animated:YES];
            if([operation.response statusCode] != 200) {
                ShowErrorAlert(NSLocalizedString(@"There was
an error communicating with the server", nil));
            } else {
                [self userDidJoin];
            }
        }
    } failure:^(AFHTTPRequestOperation *operation, NSError
*error) {
        if ([self isViewLoaded]) {
            [MBProgressHUD hideHUDForView:self.view
animated:YES];
            ShowErrorAlert([error localizedDescription]);
        }
    }];
}
```

Yikes, what is all of this? Let’s go through it line by line.

```
MBProgressHUD *hud = [MBProgressHUD showHUDAddedTo:self.view
animated:YES];
hud.labelText = NSLocalizedString(@"Connecting", nil);
```

This shows an activity spinner that blocks the whole screen. You can read more about MBProgressHUD in Ray’s article.

```
NSDictionary *params = @{@"cmd":@"join",
                         @"user_id":[_dataModel userId],
                         @"token":[_dataModel deviceToken],
                         @"name":[_dataModel nickname],
                         @"code":[_dataModel secretCode]};
```

Here you create a dictionary which contains the parameters to be posted. I designed the server API to look for a field named “cmd” in the POST data. The value of this field determines which command the API will execute. In this case, it is the “join” command.

Join takes four parameters. The user’s nickname and the secret code are obvious. These are what the user typed into the input fields on the Login screen. But what are “user_id” and “token”? As you might guess, token is the push device token. This is where you let the server know what address to put on the push notifications that it wants to send to this user.

The user_id as the name suggest is used to uniquely identify the user. I chose to identify users in the server's database using the user_id. I could have used the nickname instead but then no two persons could have the same nickname and you would need some way to let the app know that a nickname was already taken.

I could also have used the device token to uniquely identify a user, but the device token may change from time to time. I'm not saying that this approach should be used for all web services, but for our purposes it is sufficient.

> **Note:** It is important to note that the user_id and device token are different from one another. The user_id is fixed and does not change where as the device token may change from time to time. Also the device token is only used for push notifications

OK, now it becomes interesting:

```
[_client postPath:@"/api.php"
                             parameters:params
                             success:^(AFHTTPRequestOperation
*operation, id responseObject) {
        //success block
} failure:^(AFHTTPRequestOperation *operation, NSError
*error) {
       //failure block
}];
```

Blocks! In OS 4.0, blocks were introduced as an alternative for delegates. Just for the fun of it, I decided to use blocks in this tutorial. Here you set the success block. The code in between the ^{ } brackets will not be executed right now, but when the request was successfully completed.

Inside the block you do:

```
if ([self isViewLoaded])
{
      [MBProgressHUD hideHUDForView:self.view animated:YES];
```

First, you check if the view is still loaded. Because the request will be performed asynchronously in the background, it might take a while to complete. Theoretically it is possible that our view is unloaded in the mean time, for example because of a low memory warning. In that case, you will simply ignore the fact that this request ever happened.

This is probably a little overkill for an app as simple as this one, but it's a bit of defensive programming on my part. It's always a good idea to keep these kinds of strange situations in mind, especially when you're doing things in the background.

The rest of the completion block looks like this:

```
if([operation.response statusCode] != 200) {
     ShowErrorAlert(NSLocalizedString(@"There was an error 
communicating with the server", nil));
} else {
     [self userDidJoin];
}
```

You check the status code of the response. This is the HTTP status code that the server API sends back to the app. If everything went right, as it normally does, this status code is "200 OK". However, computers and the internet are fickle and sometimes things go wrong. For example if the MySQL database goes offline, the server API will return "500 Server Error". You shown an alert view in such cases.

The ShowErrorAlert() function is a helper function that creates a UIAlertView and shows it. Also note that I use NSLocalizedString() whenever I create a string for showing on the screen. That is a habit I got into, which makes it a lot easier later to localize your apps. See also Sean Berry's tutorial on localization.

If there were no errors, you call the userDidJoin method. This method tells the DataModel that the user has successfully joined the chat and then it closes the modal view controller, which returns the user to the main screen.

Because who knows what might happen on the internet, you also set a block that is executed when the request fails for some reason. Often this happens because the server could not be reached, or the request took too long (time out).

```
failure:^(AFHTTPRequestOperation *operation, NSError *error)
{
        if ([self isViewLoaded]) {
            [MBProgressHUD hideHUDForView:self.view 
animated:YES];
            ShowErrorAlert([operation.error 
localizedDescription]);
        }
```

Here you simply hide the spinner and show an error alert view.

Still in **LoginViewController.m**, make the following change to loginAction:

```
- (IBAction)loginAction
{
        ...

        // REPLACE THIS LINE:
        [self userDidJoin];

        // WITH:
        [self postJoinRequest];
}
```

Previously, the login was faked by calling userDidJoin directly when the Start! button was pressed. That will no longer fly, and you will only call userDidJoin when the request to the server was successfully completed.

If you were paying close attention, you might have noticed that you called two methods on DataModel that aren't defined yet: user_id and deviceToken. Let's add these and get rid of those compiler errors.

In **DataModel.h**, add:

```
- (NSString*)userId;
- (NSString*)deviceToken;
- (void)setDeviceToken:(NSString*)token;
```

In **DataModel.m**, add the following code:

```
//add the following above the implementation block
static NSString * const UserId = @"UserId";

//add after setJoinedChat: method
- (NSString*)userId
{
    NSString *userId = [[NSUserDefaults
standardUserDefaults] stringForKey:UserId];
    if (userId == nil || userId.length == 0) {
        userId = [[[NSUUID UUID] UUIDString]
stringByReplacingOccurrencesOfString:@"-" withString:@""];
        [[NSUserDefaults standardUserDefaults]
setObject:userId forKey:UserId];
    }
    return userId;
}
```

This method obtains a unique identifier for the logged in user, it uses the NSUUID api to generate a UUID (Universally Unique IDentifier, sometimes also called a GUID) and stores it in the NSUserDefaults. By default the UUID has dashes in it but you just want to have it as a string of 40 characters, so you strip out the dashes.

Note: Since Apple deprecated the use of the UDID api, you're using a UUID for this app. Say what? The UDID is an alphanumeric string unique to each device based on various hardware details. That is what a previous version of this tutorial used to identify the user. However, Apple is now rejecting any apps that use the UDID. A UUID, on the other hand, is simply an ID that is virtually guaranteed to be unique but it is not tied to the device in any way.

It's not a mandate to use a UUID — all you need is a way to uniquely identify a user. For many apps, a username/password pair will do the trick.

By the way, don't confuse the device token with the now-deprecated UDID. The device token is not the same thing as the device ID. The token is used only with push notifications and, unlike the UDID, it can change over time.

Still in DataModel.m, add the following line to the top of the file, above the @implementation block:

```
static NSString * const DeviceTokenKey = @"DeviceToken";
```

And the following functions inside the @implementation:

```
- (NSString*)deviceToken
{
        return [[NSUserDefaults standardUserDefaults]
stringForKey:DeviceTokenKey];
}

- (void)setDeviceToken:(NSString*)token
{
        [[NSUserDefaults standardUserDefaults]
setObject:token forKey:DeviceTokenKey];
}
```

There is one more small change to make to the initialize method:

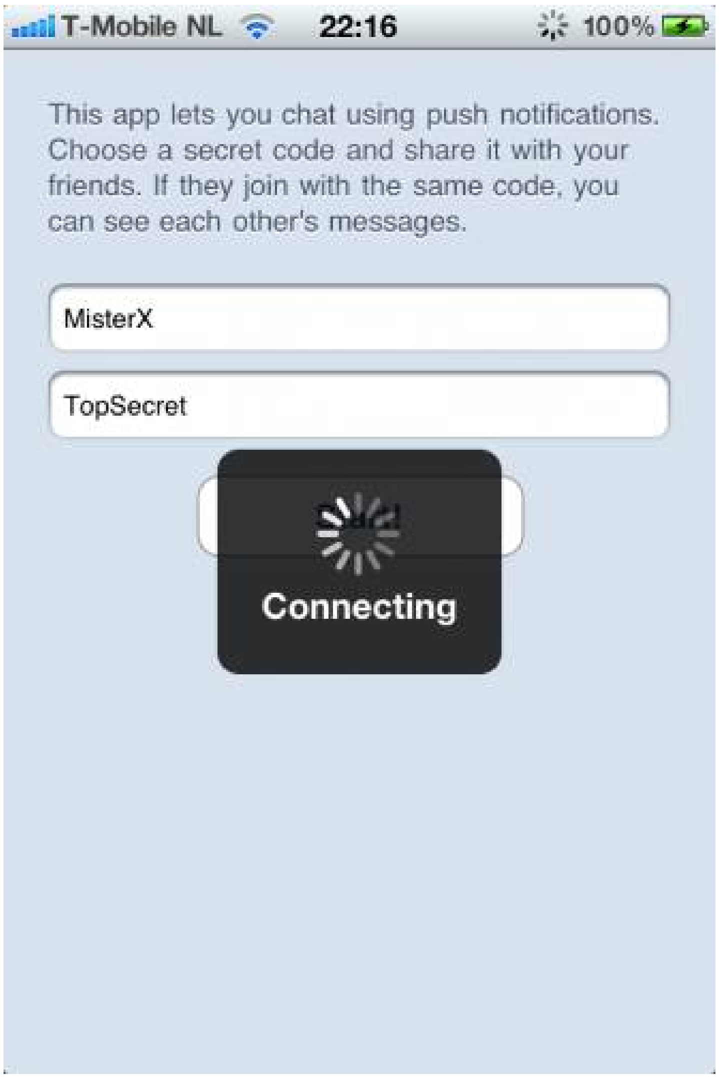
```
+ (void)initialize
{
        if (self == [DataModel class])
        {
                // Register default values for our settings
                [[NSUserDefaults standardUserDefaults]
registerDefaults:
                        @{NicknameKey: @"",
                          SecretCodeKey: @"",
                          JoinedChatKey: @0,

                          //ADD THESE LINE
                          DeviceTokenKey: @"0",
                          UserId:@""}];
        }
}
```

What did you do here? In setDeviceToken, you store the token in the NSUserDefaults dictionary. The deviceToken you read it from NSUserDefaults again. If you're not familiar with NSUserDefaults, it's a simple but very useful class that lets you store settings for your app.

In the initialize method, which is called by the Objective-C runtime when the DataModel class is used for the very first time, you set default values for all the settings you store in NSUserDefaults.

The line you added sets the default value for the device token to the string @"0". I will explain later why that is necessary. For now, it means that when you send the JOIN command to your server, it passes along a device token that is @"0" instead of the real token.

Phew, that was a lot to take in. Let's Build & Run the sucker and see if it actually works.



This app lets you chat using push notifications. Choose a secret code and share it with your friends. If they join with the same code, you can see each other's messages.

Enter a nickname and a secret code and press Start! I used “MisterX” and “TopSecret”. The “Connecting” spinner should appear briefly and if the communication with the server went well, the Login screen disappears and the main screen shows up.

If instead you got an error message, then here are some troubleshooting tips: Make sure MAMP is running. There should be green lights in front of both Apache Server and MySQL Server. Make sure you can access the server’s IP address using your web browser. Also verify you put the proper server IP address in defs.h. Finally, your iPhone should be able to reach your server, which requires them to be part of the same local network.

Wouldn’t it be nice if there were a way to verify that the server actually received and processed the message. Well, today’s your lucky day I’m just about to show you how. After receiving a JOIN request, the server API adds a record for this user to the active_users database table. You can use phpMyAdmin to look at the contents of this table and verify that this really took place.

Click on MAMP’s Open start page button and click the**phpMyAdmin** tab at the top of the screen. Navigate to the pushchat database, pick the **active_users** table, and click the Browse tab. You should see something like this:

The table now contains one row for MisterX, with secret code TopSecret, and device token “0”. The server API also logs the IP address of the device that made the request.

I kept it simple for this tutorial but for a real web service I would log as much data as possible, such as the version of the app that made the request, the OS version and type of the user’s device, the time the request was made, and so on. This kind of analytics data is invaluable to track down problems, to prevent abuse, and to get insight into who your users are and how they use your app.

Finishing Up the Server Communication

If you understood the changes you made to LoginViewController, then you should be able to follow along with the rest of the changes without a problem, because they are basically the same.

Add the following method between userDidLeave and exitAction in the **ChatViewController.m**:

```
- (void)postLeaveRequest
{
    MBProgressHUD *hud = [MBProgressHUD
showHUDAddedTo:self.navigationController.view animated:YES];
    hud.labelText = NSLocalizedString(@"Signing Out", nil);

    NSDictionary *params = @{@"cmd":@"leave",
                             @"user_id":[_dataModel
userId]};
    [ApplicationDelegate.client
            postPath:@"/api.php"
            parameters:params
            success:^(AFHTTPRequestOperation *operation, id
responseObject) {
                if ([self isViewLoaded]) {
                    [MBProgressHUD
hideHUDForView:self.navigationController.view animated:YES];
                    if (operation.response.statusCode !=
200) {

ShowErrorAlert(NSLocalizedString(@"There was an error

communicating with the server", nil));
                    } else {
                        [self userDidLeave];
                    }
                }
            } failure:^(AFHTTPRequestOperation *operation,
NSError *error) {
                if ([self isViewLoaded]) {
                    [MBProgressHUD
hideHUDForView:self.navigationController.view animated:YES];
                    ShowErrorAlert([error
localizedDescription]);
                }
            }];
}
```

This should look familiar. First you create the post parameters, this contains the "cmd" key with a value "leave". The only other parameter is the user_id, which is how you identify which user wants to sign out. The rest of the code is nearly identical to what you did before, except now you call userDidLeave when the request successfully completes.

Replace exitAction with:

```
- (IBAction)exitAction
{
    [self postLeaveRequest];
}
```

Build & Run and tap the Exit button on the main screen. Now go into **phpMyAdmin** and refresh the active_users table. It should be empty. When the server API received the LEAVE command, it removed the user's record from the table.

Our final changes are to the ComposeViewController. Just as you did with the LoginViewController, add the following property to the **ComposeViewController.h** file.

```
@property (nonatomic, strong) AFHTTPClient *client;
```

Next open up the **ChatViewController.m** file and change the -composeAction method to the following:

```
- (IBAction)composeAction
{
        // Show the Compose screen
        ComposeViewController *composeController =
(ComposeViewController *) [ApplicationDelegate.storyBoard
instantiateViewControllerWithIdentifier:@"ComposeViewControll
        composeController.dataModel = _dataModel;
        composeController.delegate = self;
    composeController.client = _client;
        [self presentViewController:composeController animate
completion:nil];
}
```

Open the **ComposeViewController.m** file and between userDidCompose and cancelAction, add:

```
- (void)postMessageRequest
{
    [_messageTextView resignFirstResponder];

    MBProgressHUD *hud = [MBProgressHUD
showHUDAddedTo:self.view animated:YES];
    hud.labelText = NSLocalizedString(@"Sending", nil);

    NSString *text = self.messageTextView.text;
    NSDictionary *params = @{@"cmd":@"message",
                             @"user_id":[_dataModel userId],
                             @"text":text};

    [_client
             postPath:@"/api.php"
             parameters:params
             success:^(AFHTTPRequestOperation *operation, id
responseObject) {
                 [MBProgressHUD hideHUDForView:self.view
animated:YES];
                 if (operation.response.statusCode != 200) {

ShowErrorAlert(NSLocalizedString(@"Could not send the
message to the server", nil));
                 } else {
                     [self userDidCompose:text];
                 }
             } failure:^(AFHTTPRequestOperation *operation,
NSError *error) {
                 if ([self isViewLoaded]) {
                     [MBProgressHUD hideHUDForView:self.view
animated:YES];
                     ShowErrorAlert([error
localizedDescription]);
                 }
             }];
}
```

More of the same. The only difference is that you hide the keyboard first. Then you send the “message” command to the API, along with the user_id and the message text. If all goes well, you call userDidCompose, which adds a new Message object to the DataModel and shows it on the main screen.

Replace saveAction with:

```
- (IBAction)saveAction
{
    [self postMessageRequest];
}
```

Build & Run once more. Login and tap the compose button at the top-right of the screen. Write some excellent prose and tap Save. After a few seconds of network activity, your message should show up in a new speech bubble.

Preparing for Push

It took a while, but you’re finally ready to add push notifications to the app. You’ve already seen how to register for push notifications and how to obtain the device token. Let’s add that again to **AppDelegate.m**.

In application:didFinishLaunchingWithOptions:, add the following just before the return statement:

```
- (BOOL)application:(UIApplication*)application
didFinishLaunchingWithOptions:(NSDictionary*)launchOptions
{
        ...

        [[UIApplication sharedApplication]
registerForRemoteNotificationTypes:
                (UIRemoteNotificationTypeSound |
UIRemoteNotificationTypeAlert)];

        return YES;
}
```

You register for sound and alert messages. The app doesn't put a badge on its icon, so there's no point in registering for that kind of notification.

Modify the **AppDelegate.m** file as follows:

```
//add at the top
#import "ChatViewController.h"
#import "DataModel.h"

//add at the bottom
- (void)application:(UIApplication*)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)dev
{
    UINavigationController *navigationController =
(UINavigationController*)_window.rootViewController;
    ChatViewController *chatViewController = (ChatViewControl
[navigationController.viewControllers objectAtIndex:0];

    DataModel *dataModel = chatViewController.dataModel;
        NSString *oldToken = [dataModel deviceToken];

        NSString *newToken = [deviceToken description];
        newToken = [newToken stringByTrimmingCharactersInSet:
[NSCharacterSet characterSetWithCharactersInString:@"<>"]];
        newToken = [newToken stringByReplacingOccurrencesOfSt
" withString:@""];

        NSLog(@"My token is: %@", newToken);

        [dataModel setDeviceToken:newToken];

        if ([dataModel joinedChat] && ![newToken
isEqualToString:oldToken])
        {
                [self postUpdateRequest];
        }
}

- (void)application:(UIApplication*)application
didFailToRegisterForRemoteNotificationsWithError:(NSError*)er
{
        NSLog(@"Failed to get token, error: %@", error);
}
```

You've seen these methods before; this is how you obtain the device token. Let's look at -didRegisterForRemoteNotificationsWithDeviceToken: in a little more detail:

```
        NSString *newToken = [deviceToken description];
        newToken = [newToken
stringByTrimmingCharactersInSet:[NSCharacterSet
characterSetWithCharactersInString:@"<>"]];
        newToken = [newToken
stringByReplacingOccurrencesOfString:@" " withString:@""];
```

You've seen before that a device token looks like this:

```
<0f744707 bebcf74f 9b7c25d4 8e335894 5f6aa01d a5ddb387
462c7eaf 61bbad78>
```

But it's more convenient to deal with in this format:

```
0f744707bebcf74f9b7c25d48e3358945f6aa01da5ddb387462c7eaf61bba
```

The code snippet above converts the former into the latter.

The following bit of code requires some explanation:

```
	if ([dataModel joinedChat] && ![newToken
isEqualToString:oldToken])
	{
		[self postUpdateRequest];
	}
```

The didRegisterForRemoteNotificationsWithDeviceToken method is usually called right away after you've registered for push notifications. Usually, but not always. Obtaining the device token happens asynchronously and it could take up to a few seconds, especially if your app hasn't tried to obtain a device token before (i.e. the very first time it is run).

It is theoretically possible that the user has already clicked the Start! button on the Login screen before you have received a valid device token. In that case, the app will send the default value for the device token, @"0", to the server. That is obviously not a valid token and you cannot send push notifications to it.

If you receive the device token after the user has already joined a chat, then you have to let the server know about this as soon as you get it. That is what the "update" API command is for. It ensures that the server API always has the most up-to-date token for this user's device. (You only send the UPDATE command when the device token has actually changed, not every time the app starts.)

Add the following code above didRegisterForRemoteNotificationsWithDeviceToken:

```
- (void)postUpdateRequest
{
    UINavigationController *navigationController =
(UINavigationController*)_window.rootViewController;
    ChatViewController *chatViewController =
(ChatViewController*)[navigationController.viewControllers
objectAtIndex:0];

    DataModel *dataModel = chatViewController.dataModel;

    NSDictionary *params = @{@"cmd":@"update",
                             @"user_id":[dataModel userId],
                             @"token":[dataModel
deviceToken]};
    AFHTTPClient *client = [AFHTTPClient clientWithBaseURL:
[NSURL URLWithString:ServerApiURL]];
```

```
    [client
     postPath:@"/api.php"
     parameters:params
     success:nil failure:nil];
}
```

We don’t care about what happens to this Request. It is silently performed in the background. So if it fails, you don’t show an error message.

This completes all the communication that the app sends to the server API. Now let’s take a look at what the server does when it receives a message and how it turns that message into a push notification.

Push on the Server

In the PushChatServer directory there is a push folder that contains the PHP scripts you need to send out push requests. You should put these files in a directory on the server that is not accessible from the web, in other words outside of your DocumentRoot. This is important because you don’t want visitors to your website to download your private key! (In our MAMP setup, this is already taken care of.)

The most important script in the push folder is **push.php**. This script should be run as a background process on your server. Every few seconds it checks if there are new push notifications to be sent out. If so, it sends them to the Apple Push Notification Service.

First, you need to edit the file **push_config.php**, which contains the configuration options for push.php. You may need to change the passphrase for the private key and possibly the database password.

As with the server API, the push script can run in either development mode or production mode. In development mode, it talks to the APNS sandbox server and it uses your development SSL certificate. You should use development mode in combination with Debug builds of your app. Production mode should be used for Ad Hoc and App Store builds of your app.

There is a file named **ck_development.pem** in the push folder. Replace it with your own PEM file that you created way back in the beginning of this tutorial.

Now open a new Terminal window and execute the following command:

```
$ /Applications/MAMP/bin/php/php5.2.17/bin/php push.php
development
```

This starts the push.php script in development mode. Note that you are now using MAMP’s version of PHP, not the PHP that comes pre-installed with your Mac. This is important because otherwise the script cannot connect to the MySQL database.

The push.php script should not exit. If it does, then something went wrong. Take a peek at the push_development.log file in the log folder. Mine looks like this:

```
2011-05-06T16:32:19+02:00 Push script started (development
mode)
2011-05-06T16:32:19+02:00 Connecting to
gateway.sandbox.push.apple.com:2195
2011-05-06T16:32:21+02:00 Connection OK
```

Because the push.php script should be run as a background process, it cannot write its output to the console. Instead, it writes to a log file. Every time it sends out a push notification, a line is added to the log.

Note that currently you're not actually running push.php as a background process. I don't like to do that for development purposes. (If you want to stop push.php, press Ctrl+C.) However, on your production server you should start the script as follows:

```
$ /Applications/MAMP/bin/php/php5.2.17/bin/php push.php
production &
```

The "&" will detach the script from the shell and put it in the background.

So what does push.php do? First, it opens a secure connection to the APNS server. It keeps that connection open all the time. Many of the examples for push that I've seen create a new connection to APNS for each push message they deliver. That approach is discouraged by Apple. Setting up a new secure connection over and over is very costly in terms of processing power and network resources. It's better to just keep the connection open.

Once the connection is established, the script goes into an endless loop. Each time through the loop it checks the push_queue database table for new entries. It can recognize new entries by their "time_sent" field. If the value of time_sent is NULL, then this notification hasn't been sent yet. The script creates a binary package that contains the device token and the JSON payload and sends it to the APNS server.

If you're interested in learning about this binary format, I suggest you take a look at the The Binary Interface and Notification Formats chapter in the Local and Push Notification Programming Guide. It's a good idea to read that guide anyway!

Once push.php has sent out the new notifications, it fills in their time_sent fields with the current time. Then the script goes to sleep for a few seconds, and the loop repeats.

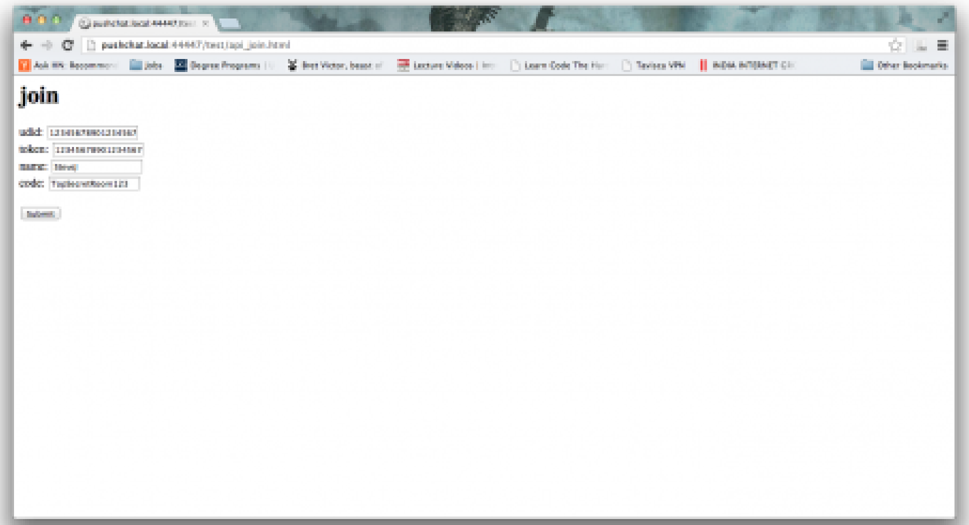
This means that to send a push notification all you have to do is place a new record in the push_queue table. That is exactly what the server API script does when it receives a MESSAGE request from the iPhone app. A few seconds later, push.php wakes up, sees the new record and sends out the notification.

Can We Finally Send Some Push Notifications!?

If you have two iPhones, you could install the app on both devices and have both users join with the same secret code. When one user sends a message to the server, the API should send a push notification to the other user.

But what if you don't have two devices to test with? Well, in that case you just have to pretend you have another user to chat with. Point your browser to:http://pushchat.local:44447/test/api_join.html

This shows a very basic HTML form:


pushchat.local:44447/test/api_join.html

join

udid: 12345678901234567

token: 12345678901234567

name: Steve

code: TopSecretRoom123

You can use this form to send a POST request to the server API, just like our app does. The server can't really tell the difference, so this is a useful tool to fake (and test) the API commands.

Fill in a 40-character user_id, a 64-character device token, and a nickname. The secret code should be the same as the one the app is signed into. It doesn't really matter what the unique ID and device token are, as long as they are different from the ones the app uses. Otherwise the server API obviously won't recognize this as a different user.

Press the Submit button. You can verify that a new user signed in by browsing the active_users table with phpMyAdmin.

Now point your browser to:http://pushchat.local:44447/test/api_message.html

Make sure the UUID is the one from the fake user, type a message text, and press Submit. Tadaa… within a few seconds your iPhone should beep and display the push notification. Congrats!

If you don't see anything, close the app first and try again. You haven't actually added any code in the app to handle incoming push notifications, so the notification will only show up when the app isn't currently running.

If you get no notification at all, then make sure push.php is still running and check the push_development.log. It should say something like:

```
2011-05-06T23:57:29+02:00 Sending message 1 to
 '0f744707bebcf74f9b7c25d48e3358945f6aa01da5ddb387462c7eaf61b
bad78',
payload: '{"aps":{"alert":"SteveJ: Hello,
world!","sound":"default"}}'
2011-05-06T23:57:29+02:00 Message successfully delivered
```

You can also verify with phpMyAdmin that a new record was added to the push_queue database table.

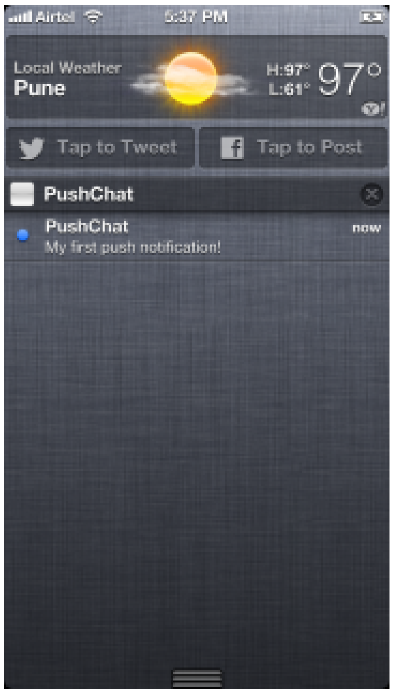
It may take a while for push notifications to be delivered, and as I mentioned before, they sometimes may not be delivered at all. Keep trying it a few more times and see if you get lucky.

Receiving Push Messages in the App

So what happens when your iPhone receives a push notification? There are three possible situations:

- **The app is currently running in the foreground**. Nothing happens on the screen and no sound is played, but your app delegate gets a notification. It is up to the app to do something in response to the notification.
- **The app is closed, either the iPhone's home screen is active or some other app is running**. An alert view/banner pops up with the message text and a sound is played. The user can press Close to dismiss the notification or View to open your app, in case of the banner the user can just tap on the banner to open your app.
- **The iPhone is locked**. Now also an alert view pops up and a sound is played but the alert does not have Close/View buttons. Instead, unlocking the phone opens the app.

In devices running iOS 5 and above the notifications can also be viewed in the notification center by swiping down from the top of the screen. The notification center can only be viewed if the device in unlocked.



PushChat
My first push notification!

The final modifications to our app all take place in**AppDelegate.m**, as the application delegate is the one who receives the push notifications. There are two places for this:

1. **application:didFinishLaunchingWithOptions:**. If your app wasn't running when the notification came in, it is launched and the notification is passed as part of the launchOptions dictionary.
2. **application:didReceiveRemoteNotification:**. This method is invoked if your app is active when a notification comes in. On iOS 4.0 or later, if your app was suspended in the background it is woken up and this method is also called. You can use UIApplication's applicationState property to find out whether your app was suspended or not.

Both methods receive a dictionary that contains the JSON payload. You don't have to parse the JSON data youself, that is already done for you by the OS.

Add the following lines to didFinishLaunchingWithOptions:, just before the return statement:

```
        if (launchOptions != nil)
        {
                NSDictionary *dictionary = [launchOptions
objectForKey:UIApplicationLaunchOptionsRemoteNotificationKey]
                if (dictionary != nil)
                {
                        NSLog(@"Launched from push
notification: %@", dictionary);
                        [self
addMessageFromRemoteNotification:dictionary updateUI:NO];
                }
        }
```

This checks to see if there are launch options and whether they contain a push notification. If so, you call the addMessageFromRemoteNotification:updateUI: method to handle the notification.

Add the following method to **AppDelegate.m**:

```
- (void)application:(UIApplication*)application
didReceiveRemoteNotification:(NSDictionary*)userInfo
{
        NSLog(@"Received notification: %@", userInfo);
        [self addMessageFromRemoteNotification:userInfo
updateUI:YES];
}
```

Pretty simple. This method also depends on addMessageFromRemoteNotification:updateUI: to do all the work.

Copy-paste the following method above didFinishLaunchingWithOptions:

```
- (void)addMessageFromRemoteNotification:
(NSDictionary*)userInfo updateUI:(BOOL)updateUI
{
    UINavigationController *navigationController =
(UINavigationController*)_window.rootViewController;
    ChatViewController *chatViewController =
                        (ChatViewController*)
[navigationController.viewControllers  objectAtIndex:0];

    DataModel *dataModel = chatViewController.dataModel;

        Message *message = [[Message alloc] init];
        message.date = [NSDate date];

        NSString *alertValue = [[userInfo valueForKey:@"aps"]
valueForKey:@"alert"];

        NSMutableArray *parts = [NSMutableArray
arrayWithArray:[alertValue componentsSeparatedByString:@":
"]];
        message.senderName = [parts objectAtIndex:0];
        [parts removeObjectAtIndex:0];
        message.text = [parts componentsJoinedByString:@": "]

        int index = [dataModel addMessage:message];

        if (updateUI)
                [chatViewController didSaveMessage:message
atIndex:index];
}
```

Oh! and dont forget to include Message.h.

This is the last bit of code, I promise! Let’s go through it bit by bit.

```
        Message *message = [[Message alloc] init];
        message.date = [NSDate date];
```

First you create a new Message object. You will fill it up with the contents of the push notification and add it to the DataModel momentarily.

```
        NSString *alertValue = [[userInfo
valueForKey:@"aps"] valueForKey:@"alert"];
```

This obtains the alert text from the push notification. The JSON payload of our push notifications looks like this:

```
{
        "aps":
        {
                "alert": "SENDER_NAME: MESSAGE_TEXT",
                "sound": "default"
        },
}
```

The server has put the message text and the sender name into the “alert” field. You’re not really interested in any of the other fields from this dictionary.

```
        NSMutableArray *parts = [NSMutableArray
arrayWithArray:[alertValue componentsSeparatedByString:@":
"]];
        message.senderName = [parts objectAtIndex:0];
        [parts removeObjectAtIndex:0];
        message.text = [parts componentsJoinedByString:@":
"];
```

This code extracts the sender name and the message text and puts them into the Message object. The sender name is the text until the first colon followed by a space.

```
        int index = [dataModel addMessage:message];
```

Now that you’ve set the properties of the Message object, you can add it to the DataModel.

```
        if (updateUI)
                [chatViewController didSaveMessage:message
atIndex:index];
```

Finally, you tell the ChatViewController to insert a new row in its table view. However, you shouldn’t do this when the notification is received in didFinishLaunchingWithOption:s. At that point, the table view isn’t loaded yet. If you try to insert a row it will get confused and the application will crash.

That’s it. Build & Run once more. Use the test_message.html form to send a push notification and now a speech bubble should be added on the left-hand side of the screen whenever the app receives a new message. Woot!

Customizing the Notifications

You may recall from the earlier discussion on the anatomy of a push notification that you can do more than send text. It is also possible to change the sound that the phone makes when it receives a new notification. I have added a short sound file, beep.caf, to the app's resources.

Go to api.php and change the following line in the makePayload() function from this:

```
$payload = '{"aps":{"alert":"' . $nameJson . ': ' .
$textJson . '","sound":"default"}}';
```

Into this:

```
$payload = '{"aps":{"alert":"' . $nameJson . ': ' .
$textJson . '","sound":"beep.caf"}}';
```

You don't have to change anything in the app or recompile it. Do close the app on the device, though. When notifications are received while the app is running no sound is played, which will sort of spoil this demonstration. Now use test_message.html to send another push notification. When the alert pops up on your phone, the sound should be different.

Play with the other parameters too! Try to localize the buttons, or set a badge on the app icon. (Note: if you want to play with badges, don't forget to make the app also register for badge notifications. Currently it only listens for sounds and alerts.)

Have fun!

The Feedback Server

APNS is happy to give you feedback!

You're probably ready to fall asleep by now (feel free to take a short break!), but there is one more piece of code that a proper push server needs.

Imagine the following scenario: Your server has a database table with the device tokens of many users. At some point, some of these users will remove your app from their devices. A sad, but unavoidable fact of life.

However, your server will not know about this as no one tells it about these uninstalls. It will happily continue to send push notifications that can no longer be delivered as your app is not present to receive them.

This is clearly an undesirable situation and that is where the Feedback Service comes in. You are supposed to connect to this service periodically and download a list of device tokens that are no longer valid. You should stop sending push notifications to these device tokens.

In the **PushChatServer/push** folder you will find the files**feedback.php**, which is the script that connects to the Feedback Service and downloads the tokens, and feedback_config.php, which contains the configuration options. This script also uses your PEM file with the SSL certificate and private key to authenticate with the Apple servers.

Unlike push.php, the feedback script should not be run continuously as a background process. Instead, you should set up a cron job on the server that launches it every hour or so.

The script sets up a connection to the Feedback Service, downloads the list of expired device tokens, and closes the connection. Then it simply deletes the records from active_users that correspond to these tokens.

Improvements and Limitations

PushChat uses push notifications as the sole mechanism of delivering data to the app. That works OK for this tutorial, but it has a big problem. The delivery of push notifications is not guaranteed, so the user may miss a piece of data if one of the notifications goes missing. In addition, if the user presses the Close button on the push alert, the app will never see the notification. In the case of PushChat, no speech bubble will be added for such messages.

A better approach is to keep the data on the server. Our API could put each message into a “messages” database table. When the app is opened, either by the user directly or in response to a push notification, it should send a “give me all new messages” request to the server and then download these messages. This way the app doesn’t depend on the push notifications alone to receive the data.

The push.php script works reasonably well but I wouldn’t use it if your app has lots of users. PHP isn’t the tool of choice for heavy-duty networking software. If you have ambitious plans it may be worth switching to a more robust implementation in C or

C++. It’s worth checking out the following code for an alternative PHP implementation:http://code.google.com/p/php-apns/.

Troubleshooting

Development to production A lot of developers face the issue of push notifications not working when they move from development to production. The usual suspect for this is that the device token is cached. If you’re using the same device for development and production the device supplies the cached token to be used for production. Remember that the production environment does not hit the sandbox endpoints.

A good workaround is to use a different app id for development and production mode. There are several ways you could do this, one way would be to keep the bundle id in a variable that you change depending upon the selected scheme, another way would be to use a good source control system :p. For example if you’re using git, then the production branch would use the production bundle id and the development one would use the development bundle id. This will ensure that the device token generated is correct according to the environment.

Where To Go From Here?

Here is a sample project with all of the code you’ve developed in the above tutorial.

Even though this tutorial is already the length of a short novel, there is still much more to be said about push notifications. If you’re serious about adding push to your app, I suggest you study the following resources for the do’s and don’ts:

- Apple’s Local and Push Notification Programming Guide
- Section 5 of the App Store Approval Guidelines
- Technical Note TN2265
- Session 129 of the WWDC 2010 videos. Most of this talk is about local notifications but there’s some discussion on push in the beginning.

If you have any questions, comments, or suggestions, please join in the forum discussion below!