**Got You In Project Documentation**

[Project Overview](#h.21mf5nxna2qb)

[Mobile specific issues](#h.tzrtshzc8j3w)

[Sessions](#h.h2zwl98xlc8v)

[Custom Code](#h.9usgb9kx13zd)

[Templates](#h.285iv2kd141m)

[Content Types](#h.22mdoox9wi02)

[Views](#h.4zrwr44nexy0)

[Custom PHP Code](#h.2k5exhmfrr9p)

[CRON](#h.7c83bmayj2uw)

[Barber Services](#h.6wswo895tt53)

[Paypal Integration](#h.m70ueqvaur2d)

[User Export](#h.cbxslqcgn2hm)

## Project Overview

Got You In has both a traditional desktop implemenation (responsive), and mobile apps for Android and iPhone.

Got You In has some interesting complexities which Drupal seems to handle reasonably well using custom content types, custom user profiles (profile 2 module), and Drupal views to connnect all the data dots.   
   
After some experimentation and once the scope of the full project started to come into focus, and the budget was found to be on the scant side, it was decided that the most practical way to integrated 2 mobile apps and a website, was to use Titanium for the app part, with a liberal use of webviews to share content / data between Drupal on one end, and the mobile app stuff on the other. This cuts way down on development time as it allows for a sharing of significant portions of the full application. While not ideal in some respects, it meets the budget and deadline constraints.

## Mobile specific issues

Most of the mobile application uses webviews to display data. This is mostly the same data a desktop user sees, but the layout and styling are changed to better adapt to a mobile situation. (This is important for testing since there are many similarities in code, but they are not identical).

In some cases, hidden webviews are used by the mobile apps, especially to detect account details (eg logged in or not). Knowing who the mobile user is (barber, owner, customer), is important for the mobile app navigation (which is done natively).

There is “native code” to handle:

* title / nav bar
* Slide out menu
  + This is user type dependent
* Sign Up screen
* Log Out screen
* And then to initialize the webviews within the mobile app
* Syncing the user login data so that in app navigation can be handled.

Initially all urls referenced from mobile included ?app=1 query string (this is continued for legacy reasons). We are also now using a different domain name for mobile apps: gotyouin.dbsclients.com.

Problems:

The biggest mobile problem was Android 2.3. This seems to choke on a number of JavaScript references that looked very commonplace and were handled by iPhone and Android 4.\* just fine. Much troubleshooting resulted.

Ideally, we’d like to use caching with Drupal sites. We don’t / can’t here, because Drupal does not see the distinction between the desktop layouts and the special layouts done for mobile apps. The caching gets scrambled. This could probably be addressed with third party modules. The larger issue is that almost the entire app really needs authenticated users for the core functionality. And caching will not play well with so many screens requiring a logged in user. Also, there are a number of highly peronalized screens (search results, etc), that would probably not be cachable anyway. So caching is turned off.

## Sessions

We have attempted to make session lifetime 180 days so that anyone who is using the app does not have to keep logging in. This seems desirable.  
  
Mobile sessions are complicated by the fact that Drupal uses http-only cookies for sessions. These are not accessible by webviews (anything in the html/javascript is accessible including document.cookie). Session data can be grabbed directly using the Drupal API, but co-ordinating raw API data with webviews seems difficult and cumbersome at best. For that reason, all data and Drupal interaction is via webviews. (The services module was evaluated as a possible solution to this, but the http-only cookies presented a hurdle.)

## Custom Code

### Templates

### Content Types

### Views

### Custom PHP Code

Outside of a number of custom templates, and the dbs\_booking module, there is custom code in /gotyouin\_php.

### CRON

See /etc/cron.d/dbs\_clients, for various crons used by the site. All related php code should be in /gotyouin\_php folder.

* email reminders
* notifications (in same script as email reminders)
* recurring appointments
* paypal ?

### Barber Services

I (Jenny, in case you have questions) am giving this hybrid mutant beast a section all by itself to try to minimize confusion because to create it required a combination of a custom content type, a taxonomy, and a custom template with both php and javascript/jQuery code.

Background: the client wanted to have the services limited to a short list of titles that are apparently standard in the industry. However, at the same time, they also wanted each barber to be able to add these services to their list of available services, AND assign their own prices/time periods for those services. When I was first added to the project, there had already been a custom content type called “services” created which had all the necessary fields for the concept of an individual service. HOWEVER, the problem is that you literally need to create an instance of the service content type for each time/price level for each barber - since each barber can assign different values to these fields, we were going to end up with a bazillion versions of a service called “Regular Cut,” for instance, differing mainly by which barber set up the service.

In order to make this work as desired, the barber would need to be able to create a service node for each service offered which contained the other information about that barber’s specific performance of the service - his time and price. So the fixed list of named services I created as a taxonomy called Service Types (machine name service\_types) since those values are supposed to be fixed and finite. Then, I altered the content type Service (machine name service) to have a custom field Service Type (field\_service\_type) which is a term reference for the taxonomy Service Types.

One of the first problems with having to set it up this way was that default back end interfaces won’t generate these items logically. In order to allow the adding/editing of services that would limit the barber to the taxonomy list, yet allow that barber to assign personalized info, I created an add/edit node (73) and made a template for it (page--node--73.tpl.php). This would render a dynamic html form that would allow the adding/deleting of service “instances” using jQuery and php.

The first step in the html form was to generate a “master” div encapsulating all necessary parts of an individual service - including all inputs named consistently and a self-destruct type “remove” button - and use that as a template for any additional divs needed dynamically while filling out the form. This master div had an ID of “service-master” and a style of display:none.

Now, since this master block should always stay pristine and untouched by data, it stays hidden, and we will need at least one visible “copy” for the person to start entering services. So, within the jQuery section at the bottom of the template, there is a function called “clone\_and\_rename” to systemically handle the creation of copies - this holds a static variable called clone\_and\_rename.counter which is used to make sure no two div copies have the same name and that all child items within the div are named using the same “key.” Each child element in the cloned div is named in the pattern of “field-x” where field is the ID of the input, and x is the key - the current value of the static iterator. Examples: ID “type-7” belongs with IDs “price-7”and “time-7,” and all three “live” in a div with the ID “service-7.”

In the php that gets executed before the page is created, it checks for any existing services for this barber. To accomplish that, it first determines whether this is any type of barber, then pulls the profile2 profile fields related to this user if so. If the user is not a barber, it redirects to the user’s profile page (so it wouldn’t show up for customers, for instance). This profile2 contains a field called “field\_services” which is designed to hold multiple node references - a “link” to each service content type “instance” that this barber “owns.” If any services exist in this field, we iterate with a foreach, and build a multidimensional array (jMultiArray) to hold all pertinent info about each listed service. Upon the triggering of the document.ready function, this array is pulled into jQuery using json\_encode, then iterated through to clone\_and\_rename - to create a div for each entry. An important part of creating the div for an existing service is to put its node id in the hidden “nid” input.

Next, a blank copy of the master div is added so a barber has somewhere to begin adding services (this occurs whether or not there were other services). Importantly, this new copy of the fields has no value in “nid,” which is how we know it’s an unsaved service. There is jQuery code attached to the generic “Add Another Service” button at the bottom which checks to see if there are any empty or incomplete service divs already on the page, and nags the user to fill them in. If there are no such issues, it simply creates a fresh copy of the entry div from the master, using the counter to construct ids/names.

jQuery code was added to handle the action of the “remove” button in each div. This was done by giving all the copies of the remove buttons the same class, and simply telling them to destroy their parent (the div) when pressed. If the person has never saved the service div in question to their profile, it deletes with no further prompt because the data isn’t saved anywhere at this point. The way the code can tell whether a service was previously saved to the profile is to check to see if there is a non-zero value in the hidden “nid” input for that div. Is the service has been saved previously, the user is asked to confirm the deletion. The node id followed by a | character is then added to a hidden field “delete” on the form, which will be used on submit to find and delete the actual node.

All interaction with the form from when it’s created until when it is submitted takes place client-side using jQuery. Once the barber is satisfied with the services chosen, clicking the “ALL FINISHED” button at the bottom submits the form for php server-side processing.

A foreach loop checks the POST values, since the number of values cannot be known ahead of time - new fields were created/destroyed dynamically before the form was submitted. The “master” fields are ignored. The contents (if any) of the “delete” input are exploded into an array using the | character as a delimiter. If the array is not empty, these nodes are deleted using a foreach loop which verifies appropriate integer values, then deletes those nodes.

All service div related fields are stuffed as an entry into a holding associative array. For example, if the POST field is type-3 (meaning this is the “type” input within the div iterated as number 3), An entry into the associative array is made in the form of array[‘3’][‘type’] = value, array[‘3’][‘time’] = value, array[‘3’][‘price’] = value, array[‘3’][‘nid’] = value, etc. Seems complex, but the end result of this structure is that you can iterate through the multidimensional associative array, and each div “number” will have as its children the set of fields associated with that div. And that is what we do - foreach through the array and process it. Any entries that have a non-zero nid value are recognized as updates to existing nodes, so the node is loaded, and data submitted is used to replace the existing node’s data. All other entries are created as nodes from scratch using the data.

There are two ways that these services are then associated with the correct barber. Firstly, this barber will be considered by Drupal to be the “author” of each service node created in this way. Secondly, we keep the profile2 field\_services up-to-date with all nodes that currently belong to the barber, since this is the way that a lot of the other custom code was designed to “find” services that belong to a barber. This services code completely replaces the contents of this field whenever the form is processed, to make sure that only the correct node ids are in there - and deleted ones are taken out.

Now, one unfortunate side effect of the fact that we have to allow each barber to create multiple “service” nodes which then belong only to them is that the number of service nodes that exist grows exponentially. Thus, we cannot use the simple “get all the service nodes” type query in a view or template to display the correct services - which is exactly what happens by default, because the custom field keyed to that entity reference pulls all possible versions of a reference. Any template or view which shows services had to have its query altered to only include services which “belong” to a given barber whose profile is being viewed. Although this is perfectly logical, it required a lot of search and destroy of code written before anyone fully thought through the services logic. The previous logic just kind of assumed that all possible services should be queried.

Due to the delicate nature of this section of code, edits to it obviously must be made with extreme caution.

### Paypal Integration

Also a little too complicated to pigeonhole under an above category. Used for paypal processing are the hidden profile2 fields field\_paypalprofile, field\_subscription\_type, field\_subscription\_status, field\_payment\_status, and field\_bill\_date. These fields are present in both the Independent Barber and Shop Owner profiles. The only real “gotcha” I can think of is that this needs to have php 5.3 or later in order for the date manipulation functions to work right.

The client has a Paypal Advanced account which we access using the Payflow API. A Paypal Advanced account allows the use of “hosted templates” on Paypal, which can be customized. The one we are using is referred to as “Layout C,” and it is simply a credit card/payment info type form meant to be iframed into the site. Thus, even though it appears the customer has never left our site, the actual credit card details do not go through our servers. This shifts PCI compliance issues almost entirely to Paypal’s arena.

Part One: the payment page, controlled by page--node--5.tpl.php. Can be reached by “/subscribe”. Since this same template is used for all three “steps” in processing the payment, we use a couple of SESSION variables to keep track of persistent info, such as the code for the subscription type the person has chosen. Step one is with a simple html form allowing the person to choose what level of service subscription they want as is appropriate to their profile type. They can choose per month, per six months, or per year, with the price being increasingly discounted the longer the time period. The person is also prompted to enter their billing address before continuing.

Step two - from this form submission, we use the entered data to construct what Paypal refers to in its API as a “secure token” - a unique ID submitted to Paypal servers which will be used to tie the details we just collected to this sales transaction. In this API call, we specify all pertinent details, including prices, billing address, and whether to use the desktop or mobile version of the template, along with a unique id “key” (in the form of uniqid('DBSSecTokenID-')) to generate the secure token value. Paypal “answers” with the resulting secure token value, and with a payment form tied to it which it feeds back to us in an iframe.

Step three - when the customer enters their cc info and submits this form within the iframe, first Paypal charges the subscription amount as a “sale” (TRXTYPE = S) and returns the PNREF of the transaction. Using the PNREF, we are able to send an API call to Paypal to create a recurring profile (a Paypal account of sorts) based on all of this person’s payment details, with a first bill date set to the calculated next period of recurrence. The official recurring “date” is adjusted in code so if the day of the original payment is > 28, the day will be considered the first day of the next month instead (to accommodate good old February) as used to generate future dates. Paypal returns a recurring profile ID, which is saved to the hidden field field\_paypalprofile in that person’s appropriate profile2 profile. The code for the subscription chosen is saved in field\_subscription\_type, the generated next billing date is saved in field\_bill\_date, and field\_subscription\_status is set to ACTIVE.

Part Two: Paypal cron, paypal\_cron.php. This cron is meant to be run daily, and handles maintenance tasks concerning recurring payments. There are 3 main functions in this file: evaluateSubscriptions, deactivateExpiredTrials and warnExpiringTrials. They are always executed in this order to avoid logic collisions - the most severe case is processed first, and the lesser cases processed afterwards to make sure no account accidentally gets a reprieve without payment.

evaluateSubscriptions queries ACTIVE profile2 profiles with a billing date of 3 days earlier than the current date - in other words, accounts which should have paid 3 days ago. Any such accounts are then checked against the status Paypal has associated with them, which will be ACTIVE if the payment due was successfully debited. All which have a status other than ACTIVE are cancelled immediately (the client didn’t even want to give them a 3 day grace period, but this at least allows for Paypal to “retry” a failed transaction a couple of times before cutting them off, so someone doesn’t get cut off just because their bank’s server is down or something stupid like that). They are sent an email telling them to call if they need to reinstate.

deactivateExpiredTrials queries all PENDING profile2 profiles that are 31 days old or older. PENDING is the value that indicates a trial subscription has been flagged as “time to pay up” (basically), and now the person has not paid. These are cancelled immediately, as above, and sent emails telling them to call if they change their minds.

warnExpiringTrials queries all NEW profile2 profiles (the status a profile gets assigned when the account is first created) that are 28 or more days old, changes the status to PENDING, and sends out an email letting the person know their trial is over, and they need to go pay for a subscription. If this payment isn’t made in 3 days, the account gets caught by deactivateExpiredTrials above.

Part Three - Cancelation. There is a red “Cancel My Account” at the bottom of the screen of user profiles of paid subscriptions. This leads to a node with a custom template. Since we didn’t implement this until after a time, the node number on live is much higher than on staging. So the template on staging is page--node--469.tpl.php - but on live, it’s page--node--1178.tpl.php. It does a basic confirm, and also checks to see if this is an admin account (because it would be very easy to accidentally block the admin account from here). Then blocks the Drupal user account if it’s NOT an admin, and uses API calls to cancel the recurring subscription profile regardless of whether it is an admin (so admins can test creating and cancelling of subscriptions). They receive an email confirming the cancellation, which tells them to call if they need to reinstate.

### Appointment Cancellation Logic - regular and recurring

Basic flow: customers can cancel appointments from their profiles - although actually what they are doing is requesting that the barber cancel the appointment because customers don’t have permissions to actually delete content associated with a barber - the cancellation action taken by a customer is forwarded by email to the barber, who then can actually delete the appointment. Barbers have parallel abilities to the customer, with the difference that their actions DO delete the appointment(s).

The theme file sites/all/themes/bh\_bootstrap/html.tpl.php contains a hidden div at the very bottom with id=”dialog” in order to provide an elegant jQuery dialog box to confirm deletions in the case of recurring appointments, since there are 3 distinct choices a person can make when trying to delete a recurring appointment: delete all, delete this one only, and cancel. For safety on all deletions, a person is asked to confirm the deletion they have requested, even after the clarifying 3 option dialog box.

Customers: the page that shows customers their appointments is a web view - Views->Appointments (Content)->Page 2. The trash can icons interact with layout.js. In turn, once the final choice has been made about deletion, there is a file gotyouin\_php/customerDelete.php, which actually handles the creation of an email requesting the barber to delete an appointment. Again, this action does not actually delete anything.

Barbers: barbers can access the edit screen of any given appointment to cancel it. The template sites/all/themes/bh\_bootstrap/page--node--edit--booking.tpl.php controls this. Originally, the deletion action caused a series of modal “confirm” dialog boxes which, when answered correctly, would allow the deletion action to proceed. However, when it was decided that the elegant-looking 3 choice jQuery box (not modal, just by design) needed to be used, we had to switch to a callback method which would only send a deletion action upon correct choices.

## User Export

Exports a list of users with various details as csv file.

User doing the export must be logged in as an administrator.