# Chapter 27: New Location APIs

By Marin Todorov

Geocoding has been present in the iOS SDK for a while now, but as with several other frameworks Apple has made some improvements to the API so they are easier to work with.

In iOS 5, forward and reverse geocoding are part of the CoreLocation framework with the new class `CLGeocoder`. Placemarks are now handled by the new class `CLPlacemark`. It is way better having all the relevant classes together and leaving the MapKit framework to handle only functionality purely related to working with maps.

Furthermore testing location aware apps is becoming much easier and more powerful with Xcode 4.2. While going through the tutorial project I'm going to show you the new tools available to test your app with different pre-defined locations and how to test with your own custom locations too. Hooray for Xcode 4.2!

In this tutorial you are going to build an app called BeerAdvisor. When started it will fetch the user's location and show them their current address and location on a map. Once the app detects that the user enters a location known as a good area for drinking beer, it will alert the user that it's beer time!

## Getting Started

Start Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Single View Application** template, and click *Next*. Enter **BeerAdvisor** for the product name, enter **BA** for the class prefix, select iPhone for the Device Family, and make sure that **Use automatic reference counting** and **Use Storyboards** are checked (leave the other checkboxes unchecked). Click *Next* and save the project by clicking *Create*.

Select your project and select the BeerAdvisor target. Open the **Build Phases** tab and open the **Link Binary With Libraries** fold. Click the plus button and double click **CoreLocation.framework** to add Core Location capabilities to the project.

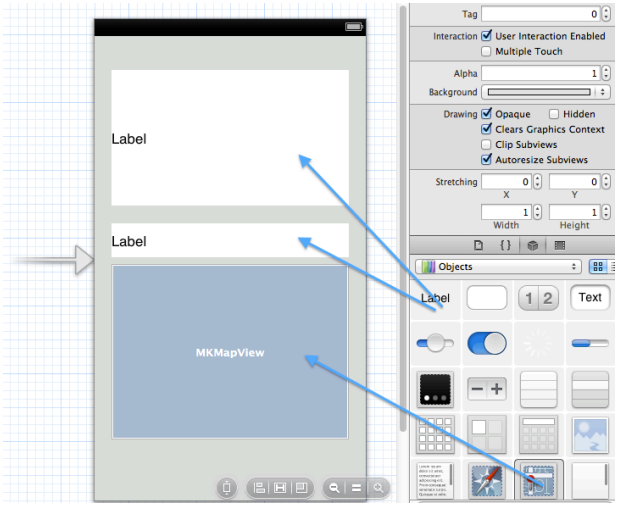




You'll also be using a map to show the user's location, so click again the plus button and double click **MapKit.framework**.

Open up **MainStoryboard.storyboard**. You are going to set up the interface to contain a map, a label to show the current coordinates, and a label to show the current address.

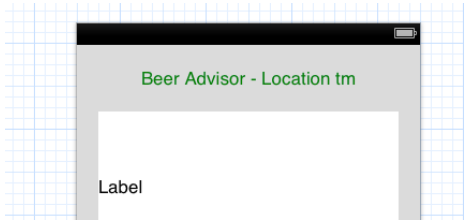
Drag two labels and one map view from the Objects library onto the application's view and position them and resize them as shown below:




In the Attributes Inspector set "Background" for both labels to white and "Color" to black. Select the top label and set the Lines to 5 in the Attributes Inspector. This should be enough for almost all kinds of address formatting.

For the second label set the **Autoshrink** property to "Minimum Font Size". For the minimum font size enter "9".

One final touch - let's add a heading to the application (beautify as you find fit):




You will need few outlets to the UI elements we just created. Open up **BAViewController.m** and replace the interface definition at the top with the following:

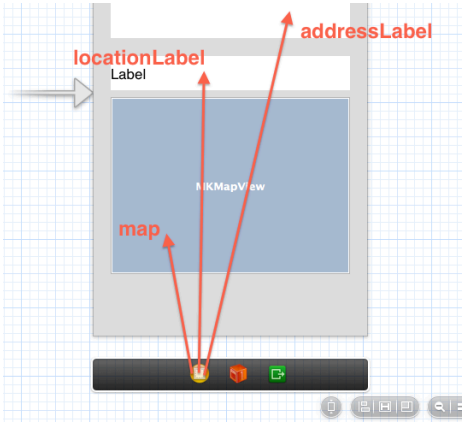
```
#import <MapKit/MapKit.h>
#import <CoreLocation/CoreLocation.h>

@interface BAViewController()<CLLocationManagerDelegate>
{
  IBOutlet MKMapView* map;
  IBOutlet UILabel* locationLabel;
  IBOutlet UILabel* addressLabel;
  CLLocationManager* manager;
}
@end
```

Next, open again **MainStoryboard.storyboard** and make the necessary connections:

- Control-drag from **View Controller** to the **top label**, and choose **addressLabel** from the popup.
- Control-drag from **View Controller** to the **bottom label**, and choose **locationLabel** from the popup.
- Control-drag from **View Controller** to the **map view**, and choose **map** from the popup.








OK - you're all connected! Let's move on to some coding.

## Reverse Geocoding With Core Location

First the application will fetch the user's current location and show them where they are in a human readable form.

Tracking user's location is done the same way as pre-iOS 5.0. You will use the `manager` class variable that you declared in the interface of the `BAViewController` class to track the device location. You also declared in the interface part that your class would conform to the `CLLocationManagerDelegate` protocol so you can set it as a delegate to the location manager.

Let's start by turning the location tracking on.

When the app starts you would like to have changes in location reported to you. So let's hook up to `viewDidLoad` and fire up the location manager in there. Replace the boilerplate declaration of `viewDidLoad` inside the implementation body with this code:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  //start updating the location

  manager = [[CLLocationManager alloc] init];
  manager.delegate = self;
```

```
    [manager startUpdatingLocation];
}
```

You create a new instance of `CLLocationManager` and store it in the manager instance variable. Then you set the delegate to self (the view controller) and finally call `startUpdatingLocation` so the location manager starts reporting location changes to the delegate.

Now to be good iOS developer, add also a `viewDidUnload` method, which will explicitly tell the manager to stop the location updates:

```
-(void)viewDidUnload
{
    [manager stopUpdatingLocation];
}
```

The `CLLocationManagerDelegate` protocol defines a method `locationManager:didUpdateToLocation:fromLocation:` for reporting locations, so you'll need to implement that in your view controller.

When you test this in the iPhone Simulator there's new locations reported constantly, so I'll implement a little condition to make the app react only to new locations.

```
- (void)locationManager:(CLLocationManager *)manager
didUpdateToLocation:(CLLocation *)newLocation
fromLocation:(CLLocation *)oldLocation
{
    if (newLocation.coordinate.latitude !=
        oldLocation.coordinate.latitude) {
        [self revGeocode: newLocation];
    }
}
```

Do you notice that the method takes `newLocation` and `oldLocation` parameters? This fact helps you track how much the user has moved, and in the case above - if the latitude didn't change you just don't fire up the reverse geocoder. You can never save too much CPU and battery! :]

Next let's add the `revGeocode` method where we will finally do some geocoding with the new iOS 5 `CLGeocoder` class.

```
-(void)revGeocode:(CLLocation*)c
{
    //reverse geocoding demo, coordinates to an address
    addressLabel.text = @"reverse geocoding coordinate ...";
```





```
    CLGeocoder* gcrev = [[CLGeocoder alloc] init];

    [gcrev reverseGeocodeLocation:c completionHandler:
      ^(NSArray *placemarks, NSError *error) {

        CLPlacemark* revMark = [placemarks objectAtIndex:0];
        //turn placemark to address text

    }];
}
```

As you see it is incredibly easy to get an address for a location. But there are few more things to talk about.

First of all the `CLGeocoder` class is a one-shot object. What that means is that you are to make only one forward or reverse geocoding call with one instance of `CLGeocoder`. So don't keep `CLGeocoder` instances in class instance variables or do any other type of persisting. You create one by calling `alloc` and `init`, make your call, and leave ARC to destroy it at the proper time.

`reverseGeocodeLocation:completionHandler:` makes a reverse geocoding call to the Apple's servers and invokes the block you pass when there's a response. Check error to see if the call was successful or not (we're skipping that to keep the tutorial shorter). In your block code iterate over the `placemarks` array to get all the possible addresses the server returned for the coordinates you provided it.

You get the first result returned from the server and store it in `revMark`. The new for iOS 5 `CLPlacemark` class provides very extended information about a location. Let's have a quick look at the properties you can access in a `CLPlacemark`:

- **CLLocation* location**: The location including coordinates of the placemark.
- **NSDictionary* addressDictionary**: The address representation of the location.
- **NSString* ISOcountryCode**: The country code - "US", "FR", "DE", etc.
- **NSString* country**: The country name.
- **NSString* postalCode**: The postal code of the area.
- **NSString* locality**: Usually the city or town.
- **NSString* subLocality**: Usually the neighborhood or area.
- **NSString* thoroughfare**: Usually the street name.
- **NSString* subThoroughfare**: Usually the street number.
- **CLRegion* region**: A geographic region associated with the location.
- **NSString* inlandWater**: If the location is on water - the name of the sea, river, or lake.
- **NSString* ocean**: If the location is in an ocean - the name of the ocean.

- **NSArray* areasOfInterest**: A list of points of interest nearby the location. (Say, how cool is that?)

For your project you are going to use only the address dictionary, but getting all sorts of info out of `CLPlaceMark` is just as easy. So, let's get the address and show it in our label. Find the "**turn placemark to address text**" comment and replace with the following code:

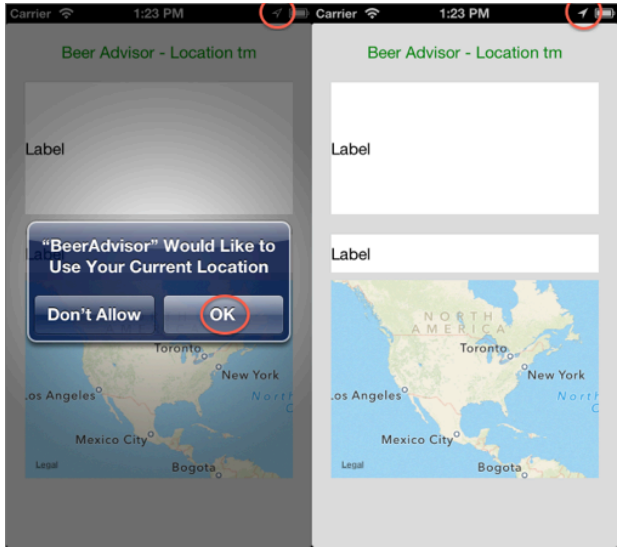
```
NSArray* addressLines =
  revMark.addressDictionary[@"FormattedAddressLines"];

NSString* revAddress =
  [addressLines componentsJoinedByString: @"\n"];

addressLabel.text = [NSString stringWithFormat:
   @"Reverse geocoded address: \n%@", revAddress];

//now turn the address to coordinates
```

In the `addressDictionary` there's a key called `FormattedAddressLines` and it contains the formatted address text. It actually is an `NSArray` instance and for each text line in the address there's a single `NSString` element in the array. You use `componentsJoinedByString:` to join the text lines and store them in `revAddress`.

Last step - show the address on the screen: just set the `text` property of the address label to what you've got from the `CLPlaceMark`. Let's test, hit Run in Xcode:







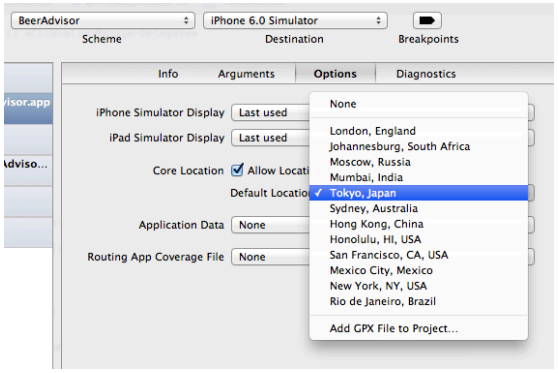

As usual, since the iOS doesn't have permission for the app to use location data, the Simulator asks for permission. Once you tap "OK" the app goes on running. However - that's not exactly what we expected - is it? Well the Simulator does not simulate location changes - this is what is wrong. We can easily fix that with the new tools Xcode 4.2 provides us with. Let's do it!

## Location in the Simulator - Part 1

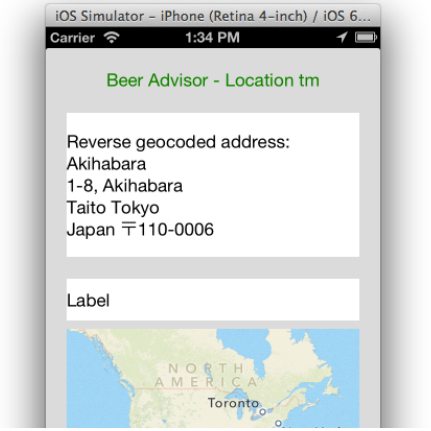
With Xcode 4.2 Apple also brings location simulation to the set of awesome iOS development tools.

One way to control the simulated location is to edit your project scheme. This way every time you run that scheme the set location will be simulated. Let's do that!

From Xcode's menu choose **Product\Edit scheme...** and sure enough the scheme properties dialogue pops up. Now open the tab saying **Options** and make sure **Allow Location Simulation** is checked. Then have a look at **Default location** and choose **Tokyo, Japan** from the pre-defined by Apple list of locations.




Now click "OK" and hit again the Run button:

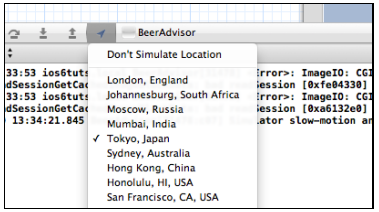



So, the selected location's address shows up inside the address label, and hey! There's the purple arrow next to the battery indicator, which shows the application's receiving location data! Great!





Also another cool feature - look at the debugging toolbar in Xcode (just under the code editor). There's a new button in Xcode 4.2 with the location arrow - when it's blue it means that it simulates location to the application being run. Also when you click it, a list with location pops up - so you can actually change the currently simulated location at runtime. Try that - choose several different location and watch the application react and show the addresses.




## Forward Geocoding

Next you are going to take the current address, convert it to coordinates, and use those to show to the user where he is on the map on the screen.

Open **BAViewController.m**, find the comment "**now turn the address to coordinates**" and replace it with:

```
[self geocode: revAddress];
```

So, once you get the address from the server, you will immediately going to make another call to convert it to coordinates.

And here's the initial method body to add inside the implementation of the class:

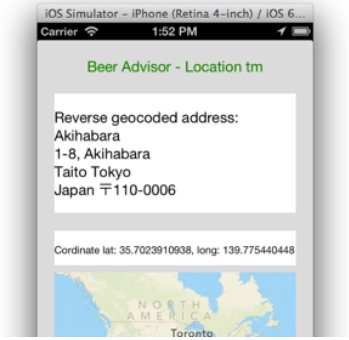
```
-(void)geocode:(NSString*)address
{
  locationLabel.text = @"geocoding address...";
  CLGeocoder* gc  = [[CLGeocoder alloc] init];

  //2
  [gc geocodeAddressString:address completionHandler:
    ^(NSArray *placemarks, NSError *error) {
    //3
    if ([placemarks count]>0) { //4
      CLPlacemark* mark = (CLPlacemark*)placemarks[0];
      double lat = mark.location.coordinate.latitude;
      double lng = mark.location.coordinate.longitude;
```

```
        //5 show the coords text
        locationLabel.text = [NSString stringWithFormat:
          @"Cordinate\nlat: %@, long: %@",
         [NSNumber numberWithDouble: lat],
          [NSNumber numberWithDouble: lng]];
          //show on the map

        }
    }];
}
```

This is pretty similar to what you did before:

1. Lets the user know there's a geocoding call in progress.
2. `geocodeAddressString:completionHandler:` works the same way as the reverse geocoding call, it just takes a string as input. It also returns an array of placemark objects, so handling the result goes the same way as what you did before.
3. Unlike passing coordinates to the geocode call, passing an address might actually not return results - if the address cannot be recognized you won't get any placemarks as a result. (Also that's why is good to check the error parameter).
4. You get the coordinates from the first `CLPlacemark`.
5. Finally you show the coordinates inside the location label in the UI.

You can hit Run and see your current latitude and longitude!




OK - you're almost done, the final touch is to show the returned placemark on the map. There's nothing new in how you do that compared to iOS 4.x, so I'm not





going to discuss it in detail. First of all you need a custom location annotation class, so let's go with the most minimal class to do the job.

Create a new File, select the **iOS\Cocoa Touch\Objective-C class** template, and enter **BAAnnotation** for the class and **NSObject** for the Subclass.

Then open **BAAnnotation.h** and replace its contents with the following:

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>

@interface BAAnnotation : NSObject <MKAnnotation>

@property (nonatomic) CLLocationCoordinate2D coordinate;

-(id)initWithCoordinate:(CLLocationCoordinate2D)c;

@end
```

You define a class, which conforms to the `MKAnnotation` protocol, so you can use it to add placemarks to a MapKit map view. The only required property is the coordinate property. It's also handy to be able to initialize the annotation in one shot, so implement a custom initializer `initWithCoordinate:`. Next open **BAAnnotation.m** and replace its contents with the following:

```
#import "BAAnnotation.h"

@implementation BAAnnotation

-(id)initWithCoordinate:(CLLocationCoordinate2D)c
{
    if (self = [super init]) {
        self.coordinate = c;
    }
    return self;
}
@end
```

This simple annotation implementation just features a custom initializer, which assigns the coordinate to the class property. Now you are ready to show the location on the map.

Open **BAViewController.m** and at the top of the code import the new annotation class:

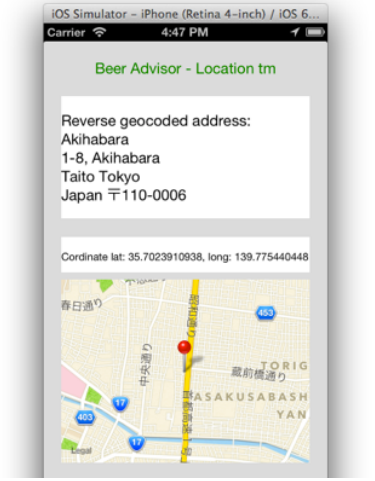
```
#import "BAAnnotation.h"
```

Now find the "**show on the map**" comment and replace it with this code to animate the map to the current location:

```
//1
 CLLocationCoordinate2D coordinate;
 coordinate.latitude = lat;
 coordinate.longitude = lng;
 //2
 [map addAnnotation:[[BAAnnotation alloc]
    initWithCoordinate:coordinate] ];
 //3
 MKCoordinateRegion viewRegion =
   MKCoordinateRegionMakeWithDistance(coordinate, 1000, 1000);
 MKCoordinateRegion adjustedRegion = [map
   regionThatFits:viewRegion];

 [map setRegion:adjustedRegion animated:YES];
```

1. First you build a `CLLocationCoordinate2D` with the latitude and longitude you've just got back from the server.
2. In one shot you create a new `BAAnnotation` and add it to the map.
3. You build a region around the coordinate and set it to the map.

Now you can hit again the Run button and see the result:








At this point you have the app forward and reverse geocode with CoreLocation!

# Monitoring for a Given Area

Monitoring for a given region is supported prior iOS 5.0, but now there's a new API called `startMonitoringForRegion:`.

The "Beer Advisor" app will have only one predefined region. This is the Kreuzberg neighborhood in Berlin - an area known for a lot of bars where tons of foreigners hang out.

1st of May in Kreuzberg by Alexandre Baron

So, at the end of `viewDidLoad` add the following code to start monitoring whether the user enters Kreuzberg:

```
//monitor for the Kreuzberg neighbourhood
//1
CLLocationCoordinate2D kreuzbergCenter;
kreuzbergCenter.latitude = 52.497727;
kreuzbergCenter.longitude = 13.431129;
//2
CLRegion* kreuzberg = [[CLRegion alloc]
  initCircularRegionWithCenter: kreuzbergCenter radius: 1000
  identifier: @"Kreuzberg"];
//3
[manager startMonitoringForRegion: kreuzberg];
```

1. You build a `CLLocationCoordinate2D` with the coordinate of the area's center more or less (I just sampled it by hand in Google maps.)
2. You create kreuzberg `CLRegion` that has the chosen center and a radius of 1000 meters (about 0.6 miles).
3. `startMonitoringForRegion:` will tell the location manager to let you know when the user arrives inside the given region.

Let's also add this code at the end of `viewDidUnload`:

```
for (CLRegion* region in manager.monitoredRegions) {
    [manager stopMonitoringForRegion: region];
}
```

Instead of keeping an instance variable to our region and then stop monitoring for it when the view unloads, we'll just iterate over `monitoredRegions` and disable all. This is ready for when you are going to start adding more regions.

There's one more final thing to do: react when the user enters into Kreuzberg! The location manager will call `locationManager:didEnterRegion:` on its delegate when that happens. Let's just show an alert:

```
- (void)locationManager:(CLLocationManager *)manager
         didEnterRegion:(CLRegion *)region
{
    [[[UIAlertView alloc] initWithTitle:@"Kreuzberg, Berlin"
       message:@"Awesome place for beer, just hop in any bar!"
       delegate:nil
       cancelButtonTitle:@"Close"
       otherButtonTitles: nil] show];
}
```

If you would like to show in the alert which region the user entered - do check out the region parameter. For now we're checking for just one region, so we're just showing an alert with preset text.

At this point a question comes to mind. Apple doesn't seem to be big fans of Kreuzberg - they didn't find necessary to include it in their predefined locations list, so we have a problem how to test this functionality without flying over to Berlin! (Which is also a great idea, but maybe it's better to first finish the tutorial and then head to some Berlin fun! )

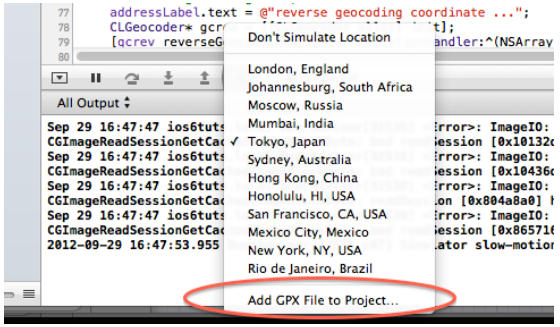
## Location in the Simulator - Part 2

Luckily Apple allows adding your own custom locations to simulate, so beer lovers across the world can rejoice!





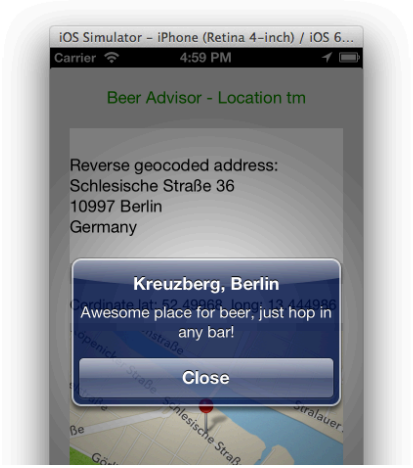
While running the project, click again on the button in the debug toolbar and have a look at the list of locations. Do you notice that there's an option at the bottom called "Add GPX file to project..."?




GPX files are plain text files containing XML markup and contain GPS data. If you're interested in the GPX file format, you can have a look here: http://en.wikipedia.org/wiki/ GPS_eXchange_Format

You can find the GPX file I generated as an asset file of this tutorial, it's called "**cake.gpx**". I used http://gpx-poi.com - a very simple tool that allows you to drop a pin on the world map and get a GPX file with the coordinates. I generated a GPX of a place in Kreuzberg (it's a bar called "Cake", thus the filename :] ) and we're going to use it to showcase custom simulated locations.

Run the application and choose "Add GPX file to project..." from the locations list. In the File Open dialogue locate "**cake.gpx**" from the chapter resources and add it to the project. This will also change the simulated location to Kreuzberg, Berlin, Germany and you should see this on the screen:




Awesome! Now that you added the GPX file to the project it is always available in the simulated locations list.

Furthermore - you can also test locations on the device, just use the locations list as you did with the Simulator - pretty awesome! However ... (there's always a but, isn't there?) monitoring for regions doesn't work on the device with simulated locations. So ... if you test on the Simulator you'll get the Kreuzberg alert, but if you do that on the device you won't.

## Where to Go From Here?

If you want to keep playing around with maps and locations, there's more fun stuff you can try!

- You can extend the demo project with further interesting beer regions (perhaps in your own home town!) If you do, please share them with us ;]
- Pull and visualize more interesting data from `CLPlaceMark`.
- Have a look at the heading information that the location manager supplies.