- **Youtube**
- **Twitter**
- **Email**
- **RSS**

**Jump**

- **Home**
- **Blog**
- **Jobs**
- **Our Apps**
- **Advertise**

# Customizing UILabel Hyperlinks

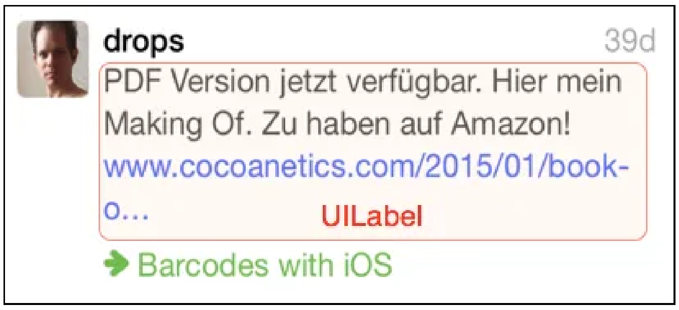
**Mar 03, 2015**

For the ProductLayer **prod.ly app** I wanted to show a timeline of user opines. Opines are – in spirit – similar to tweets and as such can also contain hyperlinks. In this post I am discussing how to customize hyperlink drawing for **UILabel**s. In a second article on this subject I will then show how to implement tappable hyperlinks for **UILabel** as well.

## Ad

UILabel is able to display attributed strings since iOS 6. Ranges which have an **NSLinkAttributeName** attribute get displayed in a different color and underlined. There are two problems with that: 1) UILabel does not implement **user touch handling** and 2) you have no way to **customize the hyperlink style**.

Consider the following style for example from prod.ly. Not only is the hyperlink tappable, I've also customized the look by removing the underline. Underlined hyperlinks are so … 80s.


drops 39d
PDF Version jetzt verfügbar. Hier mein Making Of. Zu haben auf Amazon! www.cocoanetics.com/2015/01/book-o…

**This link** opens in an in-app browser and while the users finger is down on the link it shows a gray background highlight. Similar to Tweetbot I am also shortening the URL it is too long, replacing the rest of the URL with an ellipsis.

Let's begin with something easy first.

## Customizing Link Drawing

NSLayoutManager, besides managing the layout as its name suggests, also draws the individual parts of the attributed string. The methods with which it does that can be found in the Drawing Support section of NSLayoutManager.h. Our attributed strings will not ever show any underlines, neither for hyperlinks nor for regular text. I created a **PRYLayoutManager** subclass and overwrote the appropriate method to *not do anything*:

```
- (void)drawUnderlineForGlyphRange:(NSRange)glyphRange
                     underlineType:(NSUnderlineStyle)underlineVal
                    baselineOffset:(CGFloat)baselineOffset
```

```
{
        // ignore underlines
}
```

NSLayoutManager has a internal color hard-coded for glyph ranges belonging to a hyperlink. It ignores the **NSForegroundColorAttributeName**, we believe wrongfully so. All drawing functions are called with the CGContext already configured by layout manager. This includes setting the fill color for the glyphs. Since we want the foreground color to be used instead, we change it:

```
- (void)showCGGlyphs:(const CGGlyph *)glyphs
           positions:(const CGPoint *)positions
               count:(NSUInteger)glyphCount
                font:(UIFont *)font
              matrix:(CGAffineTransform)textMatrix
          attributes:(NSDictionary *)attributes
           inContext:(CGContextRef)graphicsContext
{
   UIColor *foregroundColor = attributes[NSForegroundColorAttributeName];

   if (foregroundColor)
   {
      CGContextSetFillColorWithColor(graphicsContext, foregroundColor.CGColor);
   }

   [super showCGGlyphs:glyphs
             positions:positions
                 count:glyphCount
                  font:font
                matrix:textMatrix
            attributes:attributes
             inContext:graphicsContext];
}
```

If there is a foreground color attribute for the attributes of this glyph run, then it is set as the CGContext's fill color. This causes the glyphs belonging to the hyperlink to show in the correct color.

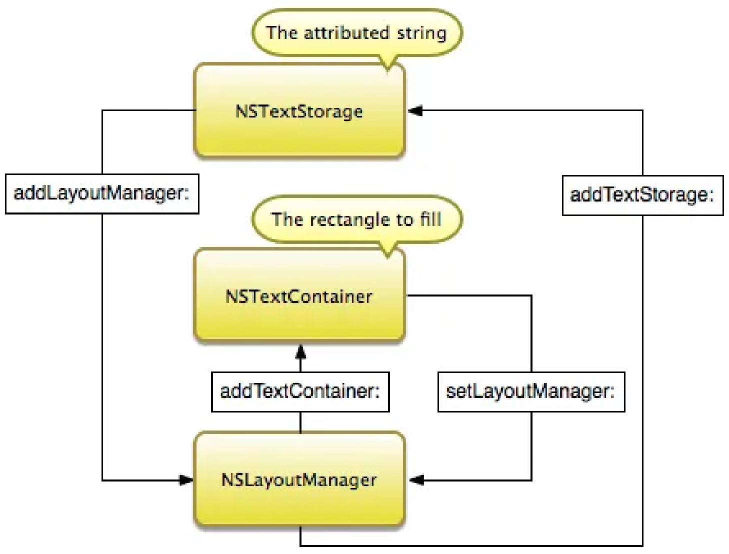
Next we need to get **UILabel** to use our custom layout manager instead of its own internal one.

## NSLayoutManager Surgery on UILabel

**UILabel** uses an internal **NSLayoutManager** for laying out the text and drawing it. Unfortunately Apple does not provide a property for us to access or customize it. For touch handling, the layout manager gives us the index in the attributed string. The layout manager is also in charge of sizing the label (intrinsic content size), layout out the text contents (line wrapping and truncation) and finally, drawing the text contents. This leaves us no choice but to replace the internal layout manager of UILabel with out own so that we can do all of the above mentioned things.

Th
*st*
ra

The *container* is nothing more than something that contains text, in our example a simple rectangle. This figure shows how the 3 classes work together.



We do not know for certain, but it is highly likely that UILabel has these instances of these 3 classes for internal use. The add* methods suggest that you can have multiple layout managers for each storage and multiple text containers for each layout manager. However for grafting our own stack on UILabel we will only need one of each. A UILabel can only have one attributed string and one rectangle, that is the frame of the label itself.

Since we cannot access these we are going to implement 3 lazy properties which instantiate our own versions:

```
- (NSTextStorage *)textStorage
{
   if (!_textStorage)
   {
      _textStorage = [[NSTextStorage alloc] init];
      [_textStorage addLayoutManager:self.layoutManager];
      [self.layoutManager setTextStorage:_textStorage];
   }

   return _textStorage;
}

- (NSTextContainer *)textContainer
{
   if (!_textContainer)
   {
      _textContainer = [[NSTextContainer alloc] init];
      _textContainer.lineFragmentPadding = 0;
      _textContainer.maximumNumberOfLines = self.numberOfLines;
      _textContainer.lineBreakMode = self.lineBreakMode;
      _textContainer.widthTracksTextView = YES;
      _textContainer.size = self.frame.size;

      [_textContainer setLayoutManager:self.layoutManager];
   }

   return _textContainer;
}

-
{
   if (!_layoutManager)
   {
      // Create a layout manager for rendering
      _layoutManager = [[PRYLayoutManager alloc] init];
      _layoutManager.delegate = self;
      [_layoutManager addTextContainer:self.textContainer];
   }

   return _layoutManager;
}
```

One tricky part is that you have to synchronize the properties on the text container with the **UILabel** properties. I've implemented only the most important ones. Every time one of these gets set our custom layout manager needs to be updated as well. In particular, necessary for determine the opine cell size heights, I am making use of the preferredMaxLayoutWidth property.

```
- (void)layoutSubviews
{
   [super layoutSubviews];

   // Update our container size when the view frame changes
   self.textContainer.size = self.bounds.size;
}

- (void)setFrame:(CGRect)frame
{
   [super setFrame:frame];

   CGSize size = frame.size;
   size.width = MIN(size.width, self.preferredMaxLayoutWidth);
   size.height = 0;
   self.textContainer.size = size;
}

- (void)setBounds:(CGRect)bounds
{
   [super setBounds:bounds];

   CGSize size = bounds.size;
   size.width = MIN(size.width, self.preferredMaxLayoutWidth);
   size.height = 0;
   self.textContainer.size = size;
}

-
{
   CGSize size = self.bounds.size;
   size.width = MIN(size.width, self.preferredMaxLayoutWidth);
   self.textContainer.size = size;
}
```

A key method for determining the size of a label is the following. I found this implementation in **KILabel.m** and – sorry to say – don't fully understand it, in particular the need for a @try/@catch. But it works, so that's good enough:

```
- (CGRect)textRectForBounds:(CGRect)bounds limitedToNumberOfLines:(NSInteger)numberOfLines
{
  // Use our text container to calculate the bounds required. First save our
  // current text container setup
  CGSize savedTextContainerSize = self.textContainer.size;
  NSInteger savedTextContainerNumberOfLines = self.textContainer.maximumNumberOfLines;

  // Apply the new potential bounds and number of lines
  self.textContainer.size = bounds.size;
  self.textContainer.maximumNumberOfLines = numberOfLines;

  // Measure the text with the new state
  CGRect textBounds;
  @try
  {
      NSRange glyphRange = [self.layoutManager
                            glyphRangeForTextContainer:self.textContainer];
      textBounds = [self.layoutManager boundingRectForGlyphRange:glyphRange
                                       inTextContainer:self.textContainer];

      // Position the bounds and round up the size for good measure
      textBounds.origin = bounds.origin;
      textBounds.size.width = ceilf(textBounds.size.width);
      textBounds.size.height = ceilf(textBounds.size.height);
  }
  @finally
  {
      // Restore the old container state before we exit under any circumstances
      self.textContainer.size = savedTextContainerSize;
      self.textContainer.maximumNumberOfLines = savedTextContainerNumberOfLines;
   }

   return textBounds;
}
```

Autolayout appears to be calling to this method from an internal method having to do with the intrinsic content size.

Fc If

we

kn

construct the appropriate attributes dictionary for turning the plain text string into an attributed one that matches the label properties. But for the sake of this tutorial we only care about this:

```
- (void)setAttributedText:(NSAttributedString *)attributedText
{
    // Pass the text to the super class first
    [super setAttributedText:attributedText];

    [self.textStorage setAttributedString:attributedText];
}
```

Passing through the attributed string to the super implementation causes the internal layout manager to work the same as our custom one. I think that this is necessary to keep auto layout working. In my implementation I am sizing the opine cells via auto layout.

The last part is to replace the text drawing with our own. When drawing the contents the contents is always vertically centered. To calculate the necessary offset we have the following helper function:

```
- (CGPoint)_textOffsetForGlyphRange:(NSRange)glyphRange
{
   CGPoint textOffset = CGPointZero;

   CGRect textBounds = [self.layoutManager boundingRectForGlyphRange:glyphRange
                                                     inTextContainer:self.textContainer];
   CGFloat paddingHeight = (self.bounds.size.height - textBounds.size.height) / 2.0f;
   if (paddingHeight &gt; 0)
   {
       textOffset.y = paddingHeight;
   }

   return textOffset;
}
```

Finally, we get to the actual drawing:

```
- (void)drawTextInRect:(CGRect)rect
{
   // Calculate the offset of the text in the view
   CGPoint textOffset;
   NSRange glyphRange = [self.layoutManager glyphRangeForTextContainer:self.textContainer];
   textOffset = [self _textOffsetForGlyphRange:glyphRange];

   // Drawing code
```

```
    //[super drawTextInRect:rect];
}
```

This first draws all backgrounds for all glyph ranges of the entire attributed string, since we only have one string, one container, one layout manager. Then we draw the actual glyphs. For debugging you can call the super's implementation of -drawTextInRect: and you should see the two outputs line up (except for the look of the hyperlinks).

## Conclusion

**Matthew Styles** deserves most of the credit for figuring out this technique. I based my implementation in large part on his **KILabel** albeit with many improvements.

At this point we have bent **UILabel** to our will insofar as we have customized the appearance of hyperlinks. The custom drawing of glyph decorations, backgrounds and the glyphs itself can be modified – based on the glyph's attributes – to suit your particular needs.

In the next blog article we will implement a gesture recognizer for interaction with the hyperlinks. We want to be able to highlight the link while it is being touched and also perform an action if the link was tapped. All of this is now possible and easy to do that we have our own layout manager instance.

**Sharing:**

More

**Like this:**

Loading...

**Related**

| | | |
|---|---|---|
| Tappable UILabel Hyperlinks | Implementing an In-App App Store | Rich Text Update 1.4 |
| March 4, 2015 | March 13, 2015 | April 5, 2013 |
| In "Recipes" | In "Recipes" | In "Updates" |

**Categories: Recipes**

**← Years of Barcodes**
**Tappable UILabel Hyperlinks →**

## 7 Comments »

1.
**_Matt Styles_**
**April 30, 2015 | 10:12 am**

Nice article. TextKit is a powerful tool and I agree its a shame that Apple don't expose more of the layout engine to make it easier to subclass the UIKit classes. I guess its intentional though.

bia

2. *AliSoftware*
   **May 23, 2015 | 4:43 pm**

   Nice article, thanks!

   Note that another solution is to use UITextView instead of UILabel, which is way more customizable.

   So one can switch to UITextView, and then :
   * customize it so that it looks like an UILabel (editable = NO, textContainerInsets = UIEdgeInsetsZero, …)
   * set the linkTextAttributes to whatever you want instead of the default blue+underline
   * access the layoutManager, textContainer and textStorage properties (which are readonly, but obviously you might override them in a subclass) to make your own (especially useful if you want each link to have a different color/underline style instead of having the same lintTextAttributes for all links)

3.
   ***Ryan McLeod***
   **July 17, 2015 | 12:14 am**

   Thank you so much for this!

   (You have a HTML entity typo! Search for “>”)

4.
   ***Ryan McLeod***
   **July 17, 2015 | 12:14 am**

   errr “& gt;” without the space

5. *OTL*
   **July 13, 2016 | 4:58 am**

   ThankYou So Much

## Trackbacks

1. ***Tappable UILabel Hyperlinks | Cocoanetics***
2. ***Implementing an In-App App Store | Cocoanetics***