# Highlighting Text Input with Jetpack Compose

We recently launched a new feature at Buffer, called Ideas. With Ideas, you can store all your best ideas, tweak them until they’re ready, and drop them straight into your Buffer queue. Now that Ideas has launched in our web and mobile apps, we have some time to share some learnings from the development of this feature. In this blog post, we’ll dive into how we added support for URL highlighting to the Ideas Composer on Android, using Jetpack Compose.

---

We started adopting Jetpack Compose into our app in 2021 – using it as standard to build all our new features, while gradually adopting it into existing parts of our application. We built the whole of the Ideas feature using Jetpack Compose – so alongside faster feature development and greater predictability within the state of our UI, we had plenty of opportunities to further explore Compose and learn more about how to achieve certain requirements in our app.

Within the Ideas composer, we support dynamic link highlighting. This means that if you type a URL into the text area, then the link will be highlighted – tapping on this link will then show an “Open link” pop-up, which will launch the link in the browser when clicked.

In this blog post, we’re going to focus on the link highlighting implementation and how this can be achieved in Jetpack Compose using the TextField composable.

For the Ideas composer, we're utilising the TextField composable to support text entry. This composable contains an argument, visualTransformation, which is used to apply visual changes to the entered text.

```
TextField( ... visualTransformation = ...
)
```

This argument requires a VisualTransformation implementation which is used to apply the visual transformation to the entered text. If we look at the source code for this interface, we'll notice a filter function which takes the content of the TextField and returns a TransformedText reference that contains the modified text.

```
@Immutable
fun interface VisualTransformation { fun filter(text: AnnotatedString): TransformedText
}
```

When it comes to this modified text, we are required to provide the implementation that creates a new AnnotatedString reference with our applied changes. This changed content then gets bundled in the TransformedText type and returned back to the TextField for composition.

So that we can define and apply transformations to the content of our TextField, we need to start by creating a new implementation of the VisualTransformation interface for which we'll create a new class, UrlTransformation. This class will implement the VisualTransformation argument, along with taking a single argument in the form of a Color. We define this argument so that we can pass a theme color reference to be applied within our logic, as we are going to be outside of composable scope and won't have access to our composable theme.

```
class UrlTransformation( val color: Color
```

```
) : VisualTransformation { }
```

With this class defined, we now need to implement the filter function from the VisualTransformation interface. Within this function we're going to return an instance of the TransformedText class – we can jump into the source code for this class and see that there are two properties required when instantiating this class.

```
/** * The transformed text with offset offset mapping */
class TransformedText( /** * The transformed text */ val text: AnnotatedString, /** * The map used for bidirectional offset mapping from original to transformed text. */ val offsetMapping: OffsetMapping
)
```

Both of these arguments are required, so we're going to need to provide a value for each when instantiating the TransformedText class.

- **text** – this will be the modified version of the text that is provided to the filter function
- **offsetMapping** – as per the documentation, this is the map used for bidirectional offset mapping from original to transformed text

```
class UrlTransformation( val color: Color
) : VisualTransformation { override fun filter(text: AnnotatedString): TransformedText { return TransformedText( ..., OffsetMapping.Identity ) }
}
```

For the offsetMapping argument, we simply pass the OffsetMapping.Identity value – this is the predefined default value used for the OffsetMapping interface, used for when that can be used for the text transformation that does not change the character count. When it comes to the text argument we'll need to write some logic that will take the current content, apply the highlighting and return it as a new AnnotatedString reference to be passed into our TransformedText reference. For

this logic, we're going to create a new function, buildAnnotatedStringWithUrlHighlighting. This is going to take two arguments – the text that is to be highlighted, along with the color to be used for the highlighting.

```
fun buildAnnotatedStringWithUrlHighlighting( text: String,
color: Color
): AnnotatedString { }
```

From this function, we need to return an AnnotatedString reference, which we'll create using buildAnnotatedString. Within this function, we'll start by using the append operation to set the textual content of the AnnotatedString.

```
fun buildAnnotatedStringWithUrlHighlighting( text: String,
color: Color
): AnnotatedString { return buildAnnotatedString {
append(text) }
}
```

Next, we'll need to take the contents of our string and apply highlighting to any URLs that are present. Before we can do this, we need to identify the URLs in the string. URL detection might vary depending on the use case, so to keep things simple let's write some example code that will find the URLs in a given piece of text. This code will take the given string and filter the URLs, providing a list of URL strings as the result.

```
text?.split("\\s+".toRegex())?.filter { word ->
Patterns.WEB_URL.matcher(word).matches()
}
```

Now that we know what URLs are in the string, we're going to need to apply highlighting to them. This is going to be in the form of an annotated string style, which is applied using the addStyle operation.

```
fun addStyle(style: SpanStyle, start: Int, end: Int)
```

When calling this function, we need to pass the SpanStyle that

we wish to apply, along with the start and end index that this styling should be applied to. We're going to start by calculating this start and end index – to keep things simple, we're going to assume there are only unique URLs in our string.

```
text?.split("\\s+".toRegex())?.filter { word -> Patterns.WEB_URL.matcher(word).matches() }.forEach { val startIndex = text.indexOf(it) val endIndex = startIndex + it.length }
```

Here we locate the start index by using the indexOf function, which will give us the starting index of the given URL. We'll then use this start index and the length of the URL to calculate the end index. We can then pass these values to the corresponding arguments for the addStyle function.

```
text?.split("\\s+".toRegex())?.filter { word -> Patterns.WEB_URL.matcher(word).matches() }.forEach { val startIndex = text.indexOf(it) val endIndex = startIndex + it.length addStyle( start = startIndex, end = endIndex ) }
```

Next, we need to provide the SpanStyle that we want to be applied to the given index range. Here we want to simply highlight the text using the provided color, so we'll pass the color value from our function arguments as the color argument for the SpanStyle function.

```
text?.split("\\s+".toRegex())?.filter { word -> Patterns.WEB_URL.matcher(word).matches() }.forEach { val startIndex = text.indexOf(it) val endIndex = startIndex + it.length addStyle( style = SpanStyle( color = color ), start = startIndex, end = endIndex ) }
```

With this in place, we now have a complete function that will take the provided text and highlight any URLs using the

provided Color reference.

```
fun buildAnnotatedStringWithUrlHighlighting( text: String, color: Color
): AnnotatedString { return buildAnnotatedString { append(text) text?.split("\\s+".toRegex())?.filter { word -> Patterns.WEB_URL.matcher(word).matches() }.forEach { val startIndex = text.indexOf(it) val endIndex = startIndex + it.length addStyle( style = SpanStyle( color = color, textDecoration = TextDecoration.None ), start = startIndex, end = endIndex ) } }
}
```

We'll then need to hop back into our UrlTransformation class and pass the result of the buildAnnotatedStringWithUrlHighlighting function call for the TransformedText argument.

```
class UrlTransformation( val color: Color
) : VisualTransformation { override fun filter(text: AnnotatedString): TransformedText { return TransformedText( buildAnnotatedStringWithUrlHighlighting(text, color), OffsetMapping.Identity ) }
}
```

Now that our UrlTransformation implementation is complete, we can instantiate this and pass the reference for the visualTransformation argument of the TextField composable. Here we are using the desired color from our MaterialTheme reference, which will be used when highlighting the URLs in our TextField content.

```
TextField( ... visualTransformation = UrlTransformation( MaterialTheme.colors.secondary)
)
```

---

With the above in place, we now have dynamic URL highlighting support within our TextField composable. This means that now whenever the user inserts a URL into the composer for an Idea,

we identify this as a URL by highlighting it using a the secondary color from our theme.

In this post, we've learnt how we can apply dynamic URL highlighting to the contents of a TextField composable. In the next post, we'll explore how we added the "Open link" pop-up when a URL is tapped within the composer input area.

https://buffer.com/resources/highlighting-text-input-with-jetpack-compose/