#9 Customize Markdown Styling

Introduction

Markdown is a popular way to format text in digital documents, and the `flutter_markdown` package provides an easy way to render Markdown in Flutter applications. However, the default styling options in `flutter_markdown` may only sometimes meet the specific needs of an application. In this document, we will explore how to customize the styling of Markdown using a code snippet as an example. We will cover the creation of a custom element builder, using `SyntaxView` and `HtmlWidget` for code snippets and iframes respectively, and integrating `AnyLinkPreview` for link previews. We will also explain implementing custom inline syntax using the `EmbedSyntax` and `LinksPreview` classes.

Overview of `flutter_markdown` package

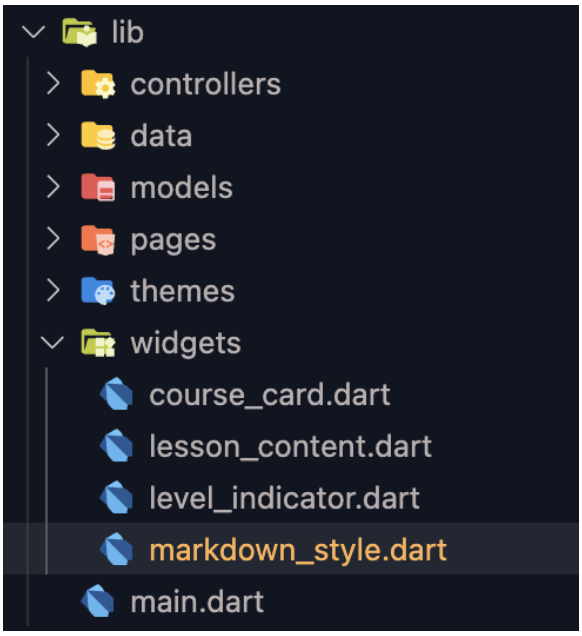
The `flutter_markdown` package is a Flutter implementation of the Markdown markup language. It allows users to render Markdown in their applications using a simple and easy-to-use interface. By default, the package provides a set of styling options that can be customized using a `MarkdownStyleSheet` object and can create a custom element builder with extends the `MarkdownElementBuilder` class.

Implementation - Step-by-step guide

To further expand on the step-by-step guide for creating a lesson page with the Markdown package in a Flutter app, let's break down each step in more detail.

Step 1 - Create a `markdown_style.dart` file

Inside `lib/widgets` directory, create a new file called `markdown_style.dart`



Step 2 - Customized the Markdown Style Sheet

In this step, we will customize the Markdown Style Sheet. Inside the `lib/widgets/markdown_style.dart` file, add the following code:

```
MarkdownStyleSheet markdownStyleSheet(BuildContext context) {
  return MarkdownStyleSheet(
    p: MyTypography.body,
    a: MyTypography.body.copyWith(
      color: Colors.blue[700],
      decoration: TextDecoration.underline,
    ),
    h1: MyTypography.title,
    h2: MyTypography.titleMedium,
    h3: MyTypography.titleSmall,
    h1Padding: const EdgeInsets.only(bottom: 5, top: 5),
    h2Padding: const EdgeInsets.only(bottom: 0, top: 10),
    h3Padding: const EdgeInsets.only(bottom: 0, top: 5),
    codeblockPadding: const EdgeInsets.all(20),
    code: GoogleFonts.firaCode(
      backgroundColor: Colors.grey[200],
      color: Colors.pink[700],
      fontSize: 14,
      fontWeight: FontWeight.w400,
    ),
    codeblockDecoration: BoxDecoration(
      color: Colors.grey[200],
```

```
      borderRadius: BorderRadius.circular(5),
    ),
    listBullet: MyTypography.body.copyWith(
      color: Colors.grey[700],
      fontSize: 14,
    ),
    listIndent: 16,
  );
}
```

This code defines a `MarkdownStyleSheet` object, which is used to style Markdown content. The `MarkdownStyleSheet` object has a number of properties that can be used to control the appearance of different Markdown elements, such as paragraphs, headings, code blocks, and lists.

The following table lists the properties of the `MarkdownStyleSheet` object and their descriptions:

Property	Description
`p`	This property is used to define the text style for paragraphs (<p>).
`a`	This property is used to define the text style for links (<a>).
`h1` `h2` `h3`	These properties are used to define text styles for level 1 (<h1>), level 2 (<h2>), and level 3 (<h3>) headings.
`h1Padding` `h2Padding` `h3Padding`	These properties are used to determine the top and bottom padding spacing for level 1, level 2, and level 3 headings.
`codeblockPadding`	This property is used to specify padding spacing for code blocks (<code>).
`code`	This property is used to define the text style for the code text within a code block (<code>).
`codeblockDecoration`	This property is used to define the display style for the entire code block.
`listBullet`	This property is used to define the text style for bulleted lists ( ).
`listIndent`	This property is used to specify the indentation spacing for each item level in the list ( ).

Using the MarkdownStyleSheet object defined with these styles, we can set the appearance of Markdown-formatted text in our Flutter application.

Afterward, we need to import the necessary file into the `markdown_style.dart` file. Add the following code at the very top of the file:

```
import 'package:flutter/material.dart';
import 'package:flutter_markdown/flutter_markdown.dart';
import 'package:google_fonts/google_fonts.dart';

import '../themes/typography.dart';
```

Now, we can use the `markdownStyleSheet` function inside the `lib/widgets/lesson_content.dart` file. Just uncomment the `styleSheet` property of `MarkdownBody` widget that we comment on before.

So far, the `MarkdownBody` widget at the `lesson_content.dart` file will look something like this:

```
                // Lesson Body with Markdown
                MarkdownBody(
                  softLineBreak: true,
                  fitContent: true,
                  shrinkWrap: true,
                  selectable: true,
                  data: snapshot.data.toString(),
                  styleSheet: markdownStyleSheet(context),  // Add this line
                  // builders: markdownBuilders(context),
                  // inlineSyntaxes: markdownInlineSyntaxes,
                  imageBuilder: (uri, title, alt) {
                    return Padding(
                      padding: const EdgeInsets.only(bottom: 10, top: 5),
                      child: GestureDetector(
                        onTap: () {
                          debugPrint('Link tapped: $uri, $title, $alt');
                          launchUrl(Uri.parse(alt!));
                        },
                        child: ClipRRect(
                          borderRadius: BorderRadius.circular(5),
                          child: Image.network(
                            uri.toString(),
                            fit: BoxFit.cover,
                          ),
                        ),
                      ),
                    );
                  },
                  onTapLink: (text, href, title) {
                    debugPrint('Link tapped: $text, $href, $title');
                    launchUrl(Uri.parse(href!));
```

```
                  },
                ),
```

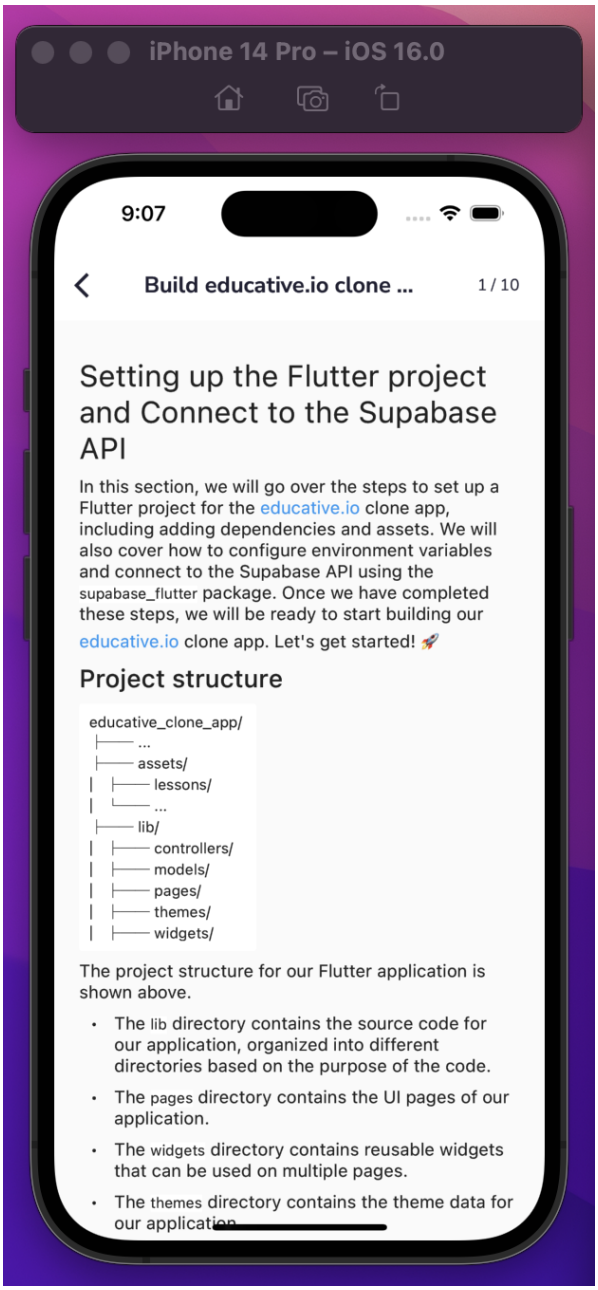
Make sure, we have imported the `markdown_style.dart` file into the `lesson_content.dart` file at the top of the file along with the other necessary imports.

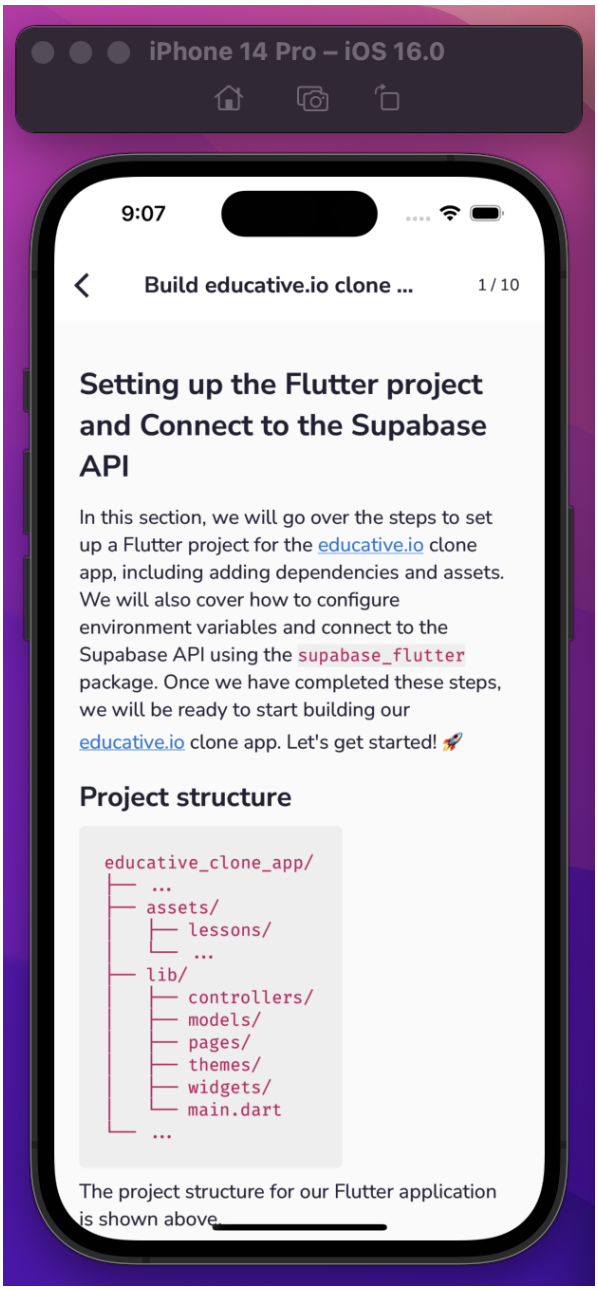
```
import 'markdown_style.dart';
```

If you successfully following so far, we can run the app and see the different results before and after we set `styleSheet`

Before:

After:





Next step, we will customize the rendering of specific Markdown elements which we need, we can create a custom element builder that extends the `MarkdownElementBuilder` class. In our case, we create a `CustomElementBuilder` class that handles the rendering of code snippets, iframe, and link previews.

Step 3 - Creation of custom code tag element

In Step 3, we create a custom element builder to render code snippets. We define the `CustomElementBuilder` class, which extends the `MarkdownElementBuilder` class, and override its `visitElementAfter` method to customize the rendering of specific Markdown elements. We will also define the `markdownBuilders` function to register our custom element builder with the `flutter_markdown` package.

And also, we will use the `SyntaxView` widget from the `flutter_syntax_view` package to custom code tag element. Make sure, you have installed the `flutter_syntax_view` package in your Flutter project. You can check it inside the `pubspec.yml` file.

Okay, first we need to define the `CustomElementBuilder` class that extends the `MarkdownElementBuilder` class.

Add the following code snippet into the `lib/widgets/markdown_style.dart` file:

```
// Custom Element Builder
class CustomElementBuilder extends MarkdownElementBuilder {
  CustomElementBuilder(this._context);

  final BuildContext _context;

  @override
  Widget? visitElementAfter(md.Element element, TextStyle? preferredStyle) {
    return null;
  }
}
```

In the previous code snippet, `_context` is a private final variable of type `BuildContext` that is passed to the `CustomElementBuilder` constructor. It is used to access the `BuildContext` of the current widget tree, which is necessary to create certain types of widgets, for example, we need it to create a custom code tag element. By passing `_context` to the constructor of `CustomElementBuilder`, we can ensure that it is available to all methods of the class that may need it.

Afterward, add the following code snippet to handle rendering the custom code tag element into the `visitElementAfter` method, specifically add that code before `return null;` code:

```
    String text = element.textContent;

    if (element.tag == 'code') {
      String language = '';
```

```
      if (element.attributes['class'] != null) {
        String lg = element.attributes['class'] as String;
        language = lg.substring(9);

        return Container(
          color: Theme.of(_context).scaffoldBackgroundColor,
          padding: const EdgeInsets.only(top: 5, bottom: 10),
          child: ClipRRect(
            borderRadius: BorderRadius.circular(5),
            child: SyntaxView(
              code: text,
              syntax: Syntax.DART,
              syntaxTheme: SyntaxTheme.ayuDark(),
              fontSize: 14,
              withZoom: false,
              withLinesCount: false,
            ),
          ),
        );
      }
    }
```

The previous code snippet checks if the current Markdown element is a `code` tag. If it is, it extracts the text content of the tag and uses it to create a `SyntaxView` widget from the `flutter_syntax_view` package. The `SyntaxView` widget is used to display the code snippet in a customizable code editor.

The `SyntaxView` widget takes several arguments, including the code to be displayed, the syntax (language) of the code, the syntax theme to be used, and the font size of the code text.

After the `SyntaxView` widget is created, it is wrapped in a `Container` widget that provides some additional styling, such as a background color and padding. The resulting widget is then returned by the `visitElementAfter` method and will be used to render any `code` tag elements in the rendered Markdown content.

So far, the `CustomElementBuilder` class will look something like this:

```
// Custom Element Builder
class CustomElementBuilder extends MarkdownElementBuilder {
  CustomElementBuilder(this._context);

  final BuildContext _context;

  @override
  Widget? visitElementAfter(md.Element element, TextStyle? preferredStyle) {
```

```
    String text = element.textContent;

    if (element.tag == 'code') {
      String language = '';

      if (element.attributes['class'] != null) {
        String lg = element.attributes['class'] as String;
        language = lg.substring(9);

        return Container(
          color: Theme.of(_context).scaffoldBackgroundColor,
          padding: const EdgeInsets.only(top: 5, bottom: 10),
          child: ClipRRect(
            borderRadius: BorderRadius.circular(5),
            child: SyntaxView(
              code: text,
              syntax: Syntax.DART,
              syntaxTheme: SyntaxTheme.ayuDark(),
              fontSize: 14,
              withZoom: false,
              withLinesCount: false,
            ),
          ),
        );
      }
    }

    return null;
  }
}
```

Don’t forget to import the `flutter_syntax_view` package into the `lib/widgets/markdown_style.dart` file, so that we can use the `SyntaxView` widget in the file. Also, import the `markdown` package and name it `md` . The `markdown` package is required for the `visitElementAfter` function and other code that custom tags and element builders use.

That should be imported at the top of the file along with the other necessary imports.

```
import 'package:flutter_syntax_view/flutter_syntax_view.dart';
import 'package:markdown/markdown.dart' as md;
```

Next, we need to define the `markdownBuilders` function to register our `CustomElementBuilder` class with `flutter_markdown` package.

Inside the `markdown_style.dart` file, add the following code snippet:

```
// Markdown Builders function
Map<String, MarkdownElementBuilder> markdownBuilders(BuildContext context) {
  return {
    'code': CustomElementBuilder(context),
  };
}
```

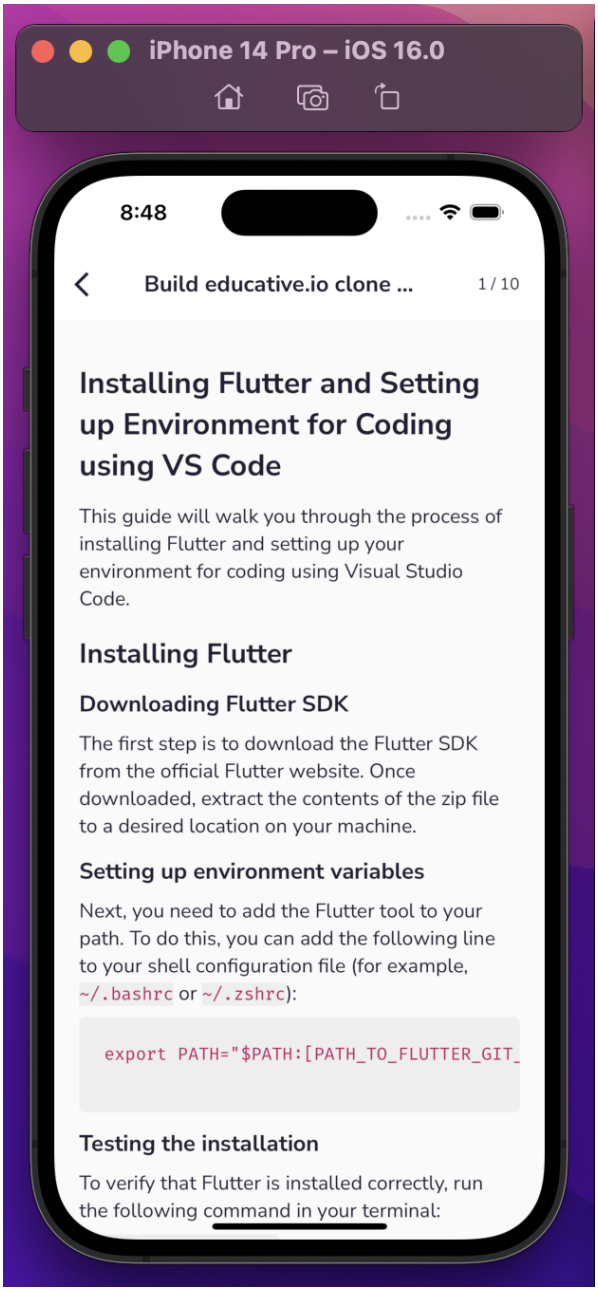
After defining the `markdownBuilder` function, we can now use or call it in the `builders` property of `MarkdownBody` widget inside the `lib/widgets/lesson_content.dart` file. Just uncomment the `builders` property of `MarkdownBody` widget that we comment on before.

So far, the `MarkdownBody` widget at the `lesson_content.dart` file will look something like this:

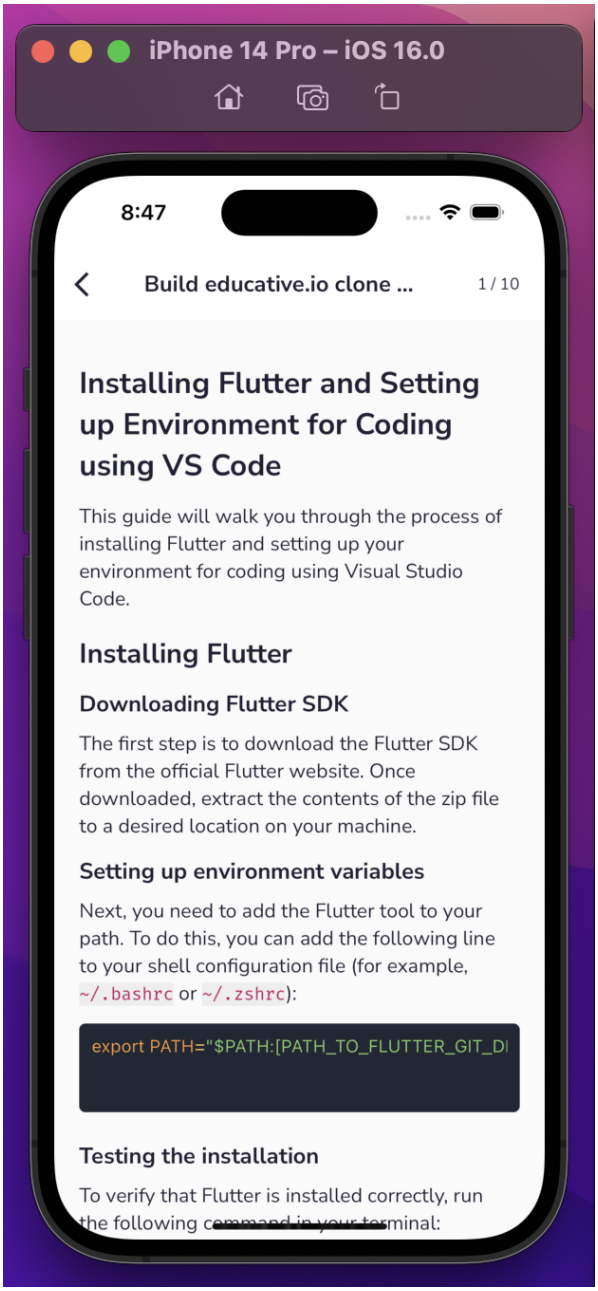
```
              // Lesson Body with Markdown
              MarkdownBody(
                softLineBreak: true,
                fitContent: true,
                shrinkWrap: true,
                selectable: true,
                data: snapshot.data.toString(),
                styleSheet: markdownStyleSheet(context),
                builders: markdownBuilders(context),     // Add this line
                // inlineSyntaxes: markdownInlineSyntaxes,
                imageBuilder: (uri, title, alt) {
                  return Padding(
                    padding: const EdgeInsets.only(bottom: 10, top: 5),
                    child: GestureDetector(
                      onTap: () {
                        debugPrint('Link tapped: $uri, $title, $alt');
                        launchUrl(Uri.parse(alt!));
                      },
                      child: ClipRRect(
                        borderRadius: BorderRadius.circular(5),
                        child: Image.network(
                          uri.toString(),
                          fit: BoxFit.cover,
                        ),
                      ),
                    ),
                  );
                },
                onTapLink: (text, href, title) {
                  debugPrint('Link tapped: $text, $href, $title');
                  launchUrl(Uri.parse(href!));
                },
              ),
```

If you successfully follow step 3, we can run the app and see the different results before and after we defined the custom code tag element with `CustomElementBuilder` class and set `builders` property of `MarkdownBody` widget.

Before:



After:



Step 4 - Create the custom link preview tag and element

In Step 4, we will create a custom rendering to display a link preview. To achieve this, we need to create a custom tag. In the markdown file, it will look something like this:

```
<link-preview>https://google.com</link-preview>
```

You can see the example implementation inside the `assets/lessons/setting-up-supabase.md` file in our Flutter project.

Next, We need to define the `LinksPreview` class, which extends the `md.InlineSyntax` class, and override its `onMatch` method to customize the rendering of specific tags. We will also define the `markdownInlineSyntaxes` variable to register our custom tags with the `flutter_markdown` package.

And also, we need to add more code to the `CustomElementBuilder` class that we created in Step 3. We will use the `AnyLinkPreview` widget from the `any_link_preview` package to render link previews. Make sure to add the relevant packages to your application by adding them to the `pubspec.yaml` file and running `flutter pub get`.

To start with, add the following code snippet into the `markdown_style.dart` file to define the `LinksPreview` class to render the link-preview tag:

```
// Custom Inline Syntax for link preview
class LinksPreview extends md.InlineSyntax {
  LinksPreview() : super(r'''<link-preview>(.*)</link-preview>''');

  @override
  bool onMatch(md.InlineParser parser, Match match) {
    final tag = md.Element.text('link-preview', match[1]!.trim());
    parser.addNode(tag);
    return true;
  }
}
```

The `LinksPreview` class has a constructor that calls the constructor of its superclass with a regular expression that matches the link-preview tag. The `onMatch` method of the `LinksPreview` class is called by the `flutter_markdown` package when the inline parser encounters a match for the regular expression. The method creates a new `md.Element.text` object with the tag name and its text content, and adds it to the parser's nodes. Finally, it returns `true` to indicate that it has handled the match.

This class is necessary to define how the `flutter_markdown` package should render the custom link-preview tag defined in the `assets/lessons/setting-up-supabase.md` file.

Next, add the following code snippet into the `markdown_style.dart` file too:

```
// Register custom inline syntax for link preview in markdown
List<md.InlineSyntax> markdownInlineSyntaxes = [
  LinksPreview(),
];
```

This list of `InlineSyntax` objects is passed to the `Markdown` widget or `MarkdownBody` widget via the `inlineSyntaxes` property and is used to parse the inline Markdown syntax in the text content of the widget.

Afterward, add the following code snippet to the `CustomElementBuilder` class to handle rendering the custom link preview widget:

```
    // link preview
    if (element.tag == 'link-preview') {
      return Padding(
        padding: const EdgeInsets.only(top: 5, bottom: 10),
        child: AnyLinkPreview(
          link: text,
          borderRadius: 10,
          errorBody: 'Oops! Something went wrong.',
          errorTitle: 'Error',
          removeElevation: true,
          titleStyle: MyTypography.body,
          bodyStyle: MyTypography.bodySmall.copyWith(
            color: Colors.grey,
          ),
          cache: const Duration(days: 1),
          backgroundColor: Colors.white,
          placeholderWidget: Container(
            height: 100,
            width: 100,
            color: Colors.grey[200],
          ),
          errorWidget: Container(
            height: 100,
            width: 100,
            color: Colors.grey[200],
          ),
        ),
      );
    }
```

This code block adds a new element to the `CustomElementBuilder` class that enables the rendering of the link preview widget. It checks if the current Markdown element is a `link-preview` tag, and if it is, it creates an `AnyLinkPreview` widget from the `any_link_preview` package. The `AnyLinkPreview` widget takes several arguments, including the link to preview, the border radius of the preview, the styling of the title and body text, and the duration to cache the preview.

The `AnyLinkPreview` widget also has several optional arguments, including the color and elevation of the preview, widgets to display when an error occurs or when the preview is loading and the height and width of the preview.

The resulting widget is then wrapped in a `Padding` widget to provide additional styling, such as padding. The resulting widget is then returned by the `visitElementAfter` method and will be used to render any `link-preview` tag elements in the rendered Markdown content.

So far, the `CustomElementBuilder` class will look something like this:

```
// Custom Element Builder
class CustomElementBuilder extends MarkdownElementBuilder {
  CustomElementBuilder(this._context);

  final BuildContext _context;

  @override
  Widget? visitElementAfter(md.Element element, TextStyle? preferredStyle) {
    String text = element.textContent;

    if (element.tag == 'code') {
      ...
    }

    // link preview
    if (element.tag == 'link-preview') {
      return Padding(
        padding: const EdgeInsets.only(top: 5, bottom: 10),
        child: AnyLinkPreview(
          link: text,
          borderRadius: 10,
          errorBody: 'Oops! Something went wrong.',
          errorTitle: 'Error',
          removeElevation: true,
          titleStyle: MyTypography.body,
          bodyStyle: MyTypography.bodySmall.copyWith(
            color: Colors.grey,
          ),
          cache: const Duration(days: 1),
```

```
          backgroundColor: Colors.white,
          placeholderWidget: Container(
            height: 100,
            width: 100,
            color: Colors.grey[200],
          ),
          errorWidget: Container(
            height: 100,
            width: 100,
            color: Colors.grey[200],
          ),
        ),
      );
    }

    return null;
  }
}
```

To use the `AnyLinkPreview` widget in the `markdown_style.dart` file, we need to import the `any_link_preview` package. This should be added at the top of the file along with the other necessary imports.

```
import 'package:any_link_preview/any_link_preview.dart';
```

Afterward, we update the `markdownBuilders` function inside the `markdown_style.dart` file to register the `link-preview` tag with the `flutter_markdown` package.

The code snippet of the `markdownBuilders` function will look something like this:

```
// Markdown Builders function
Map<String, MarkdownElementBuilder> markdownBuilders(BuildContext context) {
  return {
    'code': CustomElementBuilder(context),
    'link-preview': CustomElementBuilder(context), // Add this line
  };
}
```

Okay, so far we already define the code that is needed to render the custom link preview tag and element.

Now inside the `MarkdownBody` widget of the `lib/widgets/lesson_content.dart` file, we can set the `inlineSyntaxes` property with put the `markdownInlineSyntaxes` that we define in the

`markdown_style.dart` file before.

So far, the `MarkdownBody` widget at the `lesson_content.dart` file will look something like this:

```
// Lesson Body with Markdown
MarkdownBody(
  softLineBreak: true,
  fitContent: true,
  shrinkWrap: true,
  selectable: true,
  data: snapshot.data.toString(),
  styleSheet: markdownStyleSheet(context),
  builders: markdownBuilders(context),
  inlineSyntaxes: markdownInlineSyntaxes,   // Add this line
  imageBuilder: (uri, title, alt) {
    return Padding(
      padding: const EdgeInsets.only(bottom: 10, top: 5),
      child: GestureDetector(
        onTap: () {
          debugPrint('Link tapped: $uri, $title, $alt');
          launchUrl(Uri.parse(alt!));
        },
        child: ClipRRect(
          borderRadius: BorderRadius.circular(5),
          child: Image.network(
            uri.toString(),
            fit: BoxFit.cover,
          ),
        ),
      ),
    );
  },
  onTapLink: (text, href, title) {
    debugPrint('Link tapped: $text, $href, $title');
    launchUrl(Uri.parse(href!));
  },
),
```

And code in `markdown_style.dart` file will look something like this so far:

```
import 'package:any_link_preview/any_link_preview.dart';
import 'package:flutter/material.dart';
import 'package:flutter_markdown/flutter_markdown.dart';
import 'package:flutter_syntax_view/flutter_syntax_view.dart';
import 'package:google_fonts/google_fonts.dart';
import 'package:markdown/markdown.dart' as md;

import '../themes/typography.dart';
```

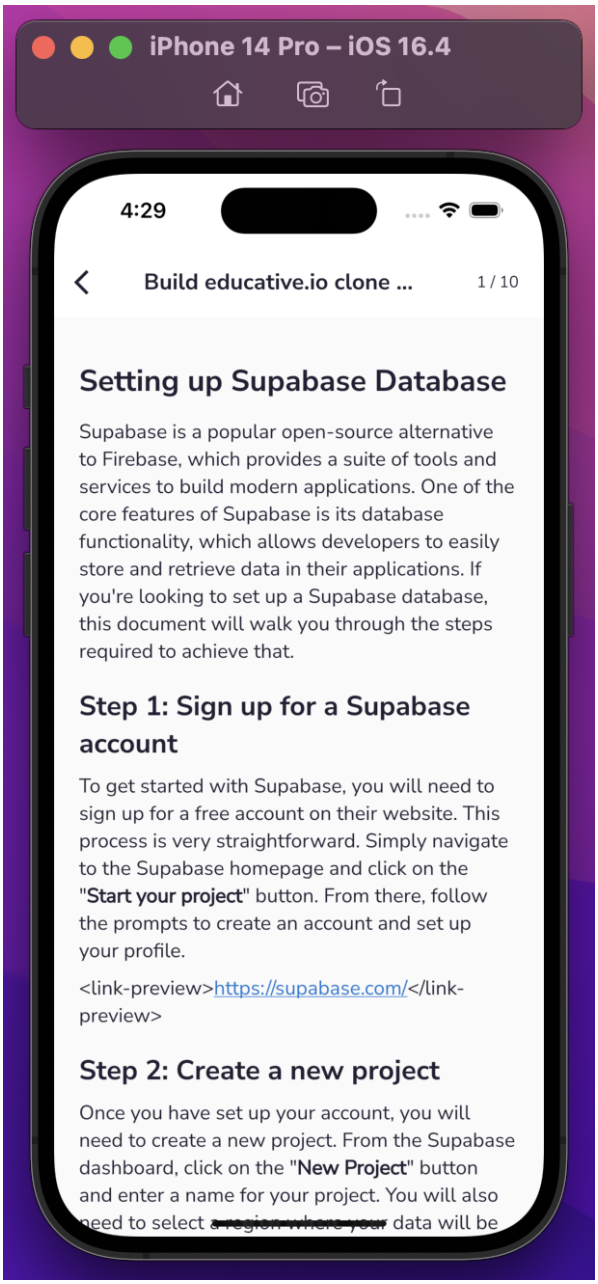
```
// Markdown Style Sheet
MarkdownStyleSheet markdownStyleSheet(BuildContext context) {
  return MarkdownStyleSheet(
    p: MyTypography.body,
    a: MyTypography.body.copyWith(
      color: Colors.blue[700],
      decoration: TextDecoration.underline,
    ),
    h1: MyTypography.title,
    h2: MyTypography.titleMedium,
    h3: MyTypography.titleSmall,
    h1Padding: const EdgeInsets.only(bottom: 5, top: 5),
    h2Padding: const EdgeInsets.only(bottom: 0, top: 10),
    h3Padding: const EdgeInsets.only(bottom: 0, top: 5),
    codeblockPadding: const EdgeInsets.all(20),
    code: GoogleFonts.firaCode(
      backgroundColor: Colors.grey[200],
      color: Colors.pink[700],
      fontSize: 14,
      fontWeight: FontWeight.w400,
    ),
    codeblockDecoration: BoxDecoration(
      color: Colors.grey[200],
      borderRadius: BorderRadius.circular(5),
    ),
    listBullet: MyTypography.body.copyWith(
      color: Colors.grey[700],
      fontSize: 14,
    ),
    listIndent: 16,
  );
}

// Markdown Builders function
Map<String, MarkdownElementBuilder> markdownBuilders(BuildContext context) {
  return {
    'code': CustomElementBuilder(context),
    'link-preview': CustomElementBuilder(context),
  };
}

// Custom Element Builder
class CustomElementBuilder extends MarkdownElementBuilder {
  CustomElementBuilder(this._context);

  final BuildContext _context;

  @override
  Widget? visitElementAfter(md.Element element, TextStyle? preferredStyle) {
    String text = element.textContent;

    if (element.tag == 'code') {
      String language = '';
```

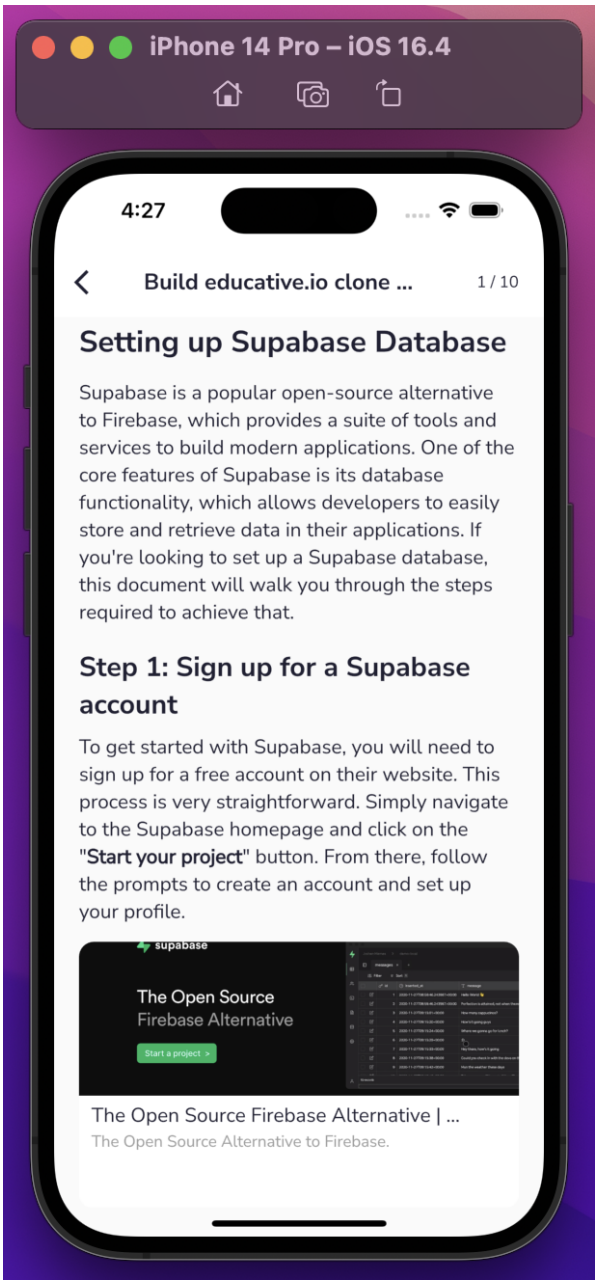
```
      if (element.attributes['class'] != null) {
        String lg = element.attributes['class'] as String;
        language = lg.substring(9);

        return Container(
          color: Theme.of(_context).scaffoldBackgroundColor,
          padding: const EdgeInsets.only(top: 5, bottom: 10),
          child: ClipRRect(
            borderRadius: BorderRadius.circular(5),
            child: SyntaxView(
              code: text,
              syntax: Syntax.DART,
              syntaxTheme: SyntaxTheme.ayuDark(),
              fontSize: 14,
              withZoom: false,
              withLinesCount: false,
            ),
          ),
        );
      }
    }

    // link preview
    if (element.tag == 'link-preview') {
      return Padding(
        padding: const EdgeInsets.only(top: 5, bottom: 10),
        child: AnyLinkPreview(
          link: text,
          borderRadius: 10,
          errorBody: 'Oops! Something went wrong.',
          errorTitle: 'Error',
          removeElevation: true,
          titleStyle: MyTypography.body,
          bodyStyle: MyTypography.bodySmall.copyWith(
            color: Colors.grey,
          ),
          cache: const Duration(days: 1),
          backgroundColor: Colors.white,
          placeholderWidget: Container(
            height: 100,
            width: 100,
            color: Colors.grey[200],
          ),
          errorWidget: Container(
            height: 100,
            width: 100,
            color: Colors.grey[200],
          ),
        ),
      );
    }

    return null;
```

```
  }
}

// Register custom inline syntax for link preview in markdown
List<md.InlineSyntax> markdownInlineSyntaxes = [
  LinksPreview(),
];

// Custom Inline Syntax for link preview
class LinksPreview extends md.InlineSyntax {
  LinksPreview() : super(r'''<link-preview>(.*)</link-preview>''');

  @override
  bool onMatch(md.InlineParser parser, Match match) {
    final tag = md.Element.text('link-preview', match[1]!.trim());
    parser.addNode(tag);
    return true;
  }
}
```

If you successfully follow step 4, we can run the app and see the different results before and after we custom inline syntax link preview.

Before: After:





Step 5 - Create the custom embed tag element to show the embed link

To achieve this, the steps will be the same as in step 4. For more detail just follow the following steps.

- Define the custom rendering to display an embed tag.

- Create a custom tag in the markdown file, it will look something like this: `<embed>https://www.loom.com/embed/cf976c5010e841ab98ac94faedd8cae6</embed>`.
- Define the `EmbedSyntax` class, which extends the `md.InlineSyntax` class, and override its `onMatch` method to customize the rendering of specific tags.
- Add the `EmbedSyntax` class to the `markdownInlineSyntaxes` variable to register our custom tags with the `flutter_markdown` package.
- Add a new element to the `CustomElementBuilder` class that enables the rendering of the embed widget. It checks if the current Markdown element is an `embed` tag, and if it is, it creates an `HtmlWidget` widget from the `flutter_widget_from_html` package.
- Update the `MarkdownBody` widget of the `lib/widgets/lesson_content.dart` file to set the `inlineSyntaxes` property with `markdownInlineSyntaxes` that we define in the `markdown_style.dart` file.

To define the custom rendering to display an embed tag, we need to create a custom tag in the markdown file. In this case, let's use the `<embed>` tag with the text content set to an embed link. Here's an example of what the custom tag might look like:

```
<embed>https://www.loom.com/embed/cf976c5010e841ab98ac94faedd8cae6</embed>
```

You can see the example implementation inside the `assets/lessons/introduction.md` file in our Flutter project.

Next, we need to define the `EmbedSyntax` class inside the `markdown_style.dart` file, which extends the `md.InlineSyntax` class, and override its `onMatch` method to customize the rendering of specific tags. Here's what the `EmbedSyntax` class might look like:

```
class EmbedSyntax extends md.InlineSyntax {
  EmbedSyntax() : super(r'''<embed>(.*)</embed>''');

  @override
  bool onMatch(md.InlineParser parser, Match match) {
    final tag = md.Element.text('embed', match[1]!.trim());
    parser.addNode(tag);
    return true;
  }
}
```

The `EmbedSyntax` class has a constructor that calls the constructor of its superclass with a regular expression that matches the embed tag. The `onMatch` method of the `LinksPreview` class is called by the `flutter_markdown` package when the inline parser encounters a match for the regular expression. The method creates a new `md.Element.text` object with the tag name and its text content, and adds it to the parser's nodes. Finally, it returns `true` to indicate that it has handled the match.

After defining the `EmbedSyntax` class, we need to add it to the `markdownInlineSyntaxes` variable to register our custom `embed` tag with the `flutter_markdown` package. Here's what the updated `markdownInlineSyntaxes` variable might look like:

```
List<md.InlineSyntax> markdownInlineSyntaxes = [
  LinksPreview(),
  EmbedSyntax(),
];
```

Next, we need to add a new element to the `CustomElementBuilder` class that enables the rendering of the embed widget. It checks if the current Markdown element is an `embed` tag, and if it is, it creates an `HtmlWidget` widget from the `flutter_widget_from_html` package. Here's what the updated `CustomElementBuilder` class might look like:

```
class CustomElementBuilder extends MarkdownElementBuilder {
  CustomElementBuilder(this._context);

  final BuildContext _context;

  @override
  Widget? visitElementAfter(md.Element element, TextStyle? preferredStyle) {
    String text = element.textContent;

    if (element.tag == 'code') {
      ...
    }

    // link preview
    if (element.tag == 'link-preview') {
      ...
    }

    // embed
    if (element.tag == 'embed') {
      return Padding(
        padding: const EdgeInsets.only(top: 5, bottom: 10),
```

```
        child: ClipRRect(
          borderRadius: BorderRadius.circular(5),
          child: HtmlWidget(
            '<iframe src="$text"></iframe>',
            customStylesBuilder: (element) {
              if (element.classes.contains('name')) {
                return {'color': 'red'};
              }
              return null;
            },
            onErrorBuilder: (context, element, error) =>
                Text('$element error: $error'),
            onLoadingBuilder: (context, element, loadingProgress) =>
                const CircularProgressIndicator(),
            onTapUrl: (url) => launchUrl(Uri.parse(url)),
            renderMode: RenderMode.column,
            buildAsync: true,
            textStyle: const TextStyle(fontSize: 14),
            enableCaching: true,
          ),
        ),
      );
    }

    return null;
  }
}
```

To use the `HtmlWidget` widget in the `markdown_style.dart` file, we need to import the `flutter_widget_from_html` package. Additionally, we need to import the `url_launcher` package, which is used in the `onTapUrl` property of the `HtmlWidget` widget. These should be added at the top of the file, along with any other necessary imports.

```
import 'package:flutter_widget_from_html/flutter_widget_from_html.dart';
import 'package:url_launcher/url_launcher.dart';
```

Afterward, we update the `markdownBuilders` function inside the `markdown_style.dart` file to register the `embed` tag with the `flutter_markdown` package.

The code snippet of the `markdownBuilders` function will look something like this:

```
// Markdown Builders function
Map<String, MarkdownElementBuilder> markdownBuilders(BuildContext context) {
  return {
    'code': CustomElementBuilder(context),
```

```
    'link-preview': CustomElementBuilder(context),
    'embed': CustomElementBuilder(context),          // Add this line
  };
}
```

Okay, so far we already define the code that is needed to render the custom embed tag element.

Finally, populate the `markdown_style.dart` file will look something like this:

```
import 'package:any_link_preview/any_link_preview.dart';
import 'package:flutter/material.dart';
import 'package:flutter_markdown/flutter_markdown.dart';
import 'package:flutter_syntax_view/flutter_syntax_view.dart';
import 'package:flutter_widget_from_html/flutter_widget_from_html.dart';
import 'package:google_fonts/google_fonts.dart';
import 'package:markdown/markdown.dart' as md;
import 'package:url_launcher/url_launcher.dart';

import '../themes/typography.dart';

// Markdown Style Sheet
MarkdownStyleSheet markdownStyleSheet(BuildContext context) {
  return MarkdownStyleSheet(
    p: MyTypography.body,
    a: MyTypography.body.copyWith(
      color: Colors.blue[700],
      decoration: TextDecoration.underline,
    ),
    h1: MyTypography.title,
    h2: MyTypography.titleMedium,
    h3: MyTypography.titleSmall,
    h1Padding: const EdgeInsets.only(bottom: 5, top: 5),
    h2Padding: const EdgeInsets.only(bottom: 0, top: 10),
    h3Padding: const EdgeInsets.only(bottom: 0, top: 5),
    codeblockPadding: const EdgeInsets.all(20),
    code: GoogleFonts.firaCode(
      backgroundColor: Colors.grey[200],
      color: Colors.pink[700],
      fontSize: 14,
      fontWeight: FontWeight.w400,
    ),
    codeblockDecoration: BoxDecoration(
      color: Colors.grey[200],
      borderRadius: BorderRadius.circular(5),
    ),
    listBullet: MyTypography.body.copyWith(
      color: Colors.grey[700],
      fontSize: 14,
    ),
```

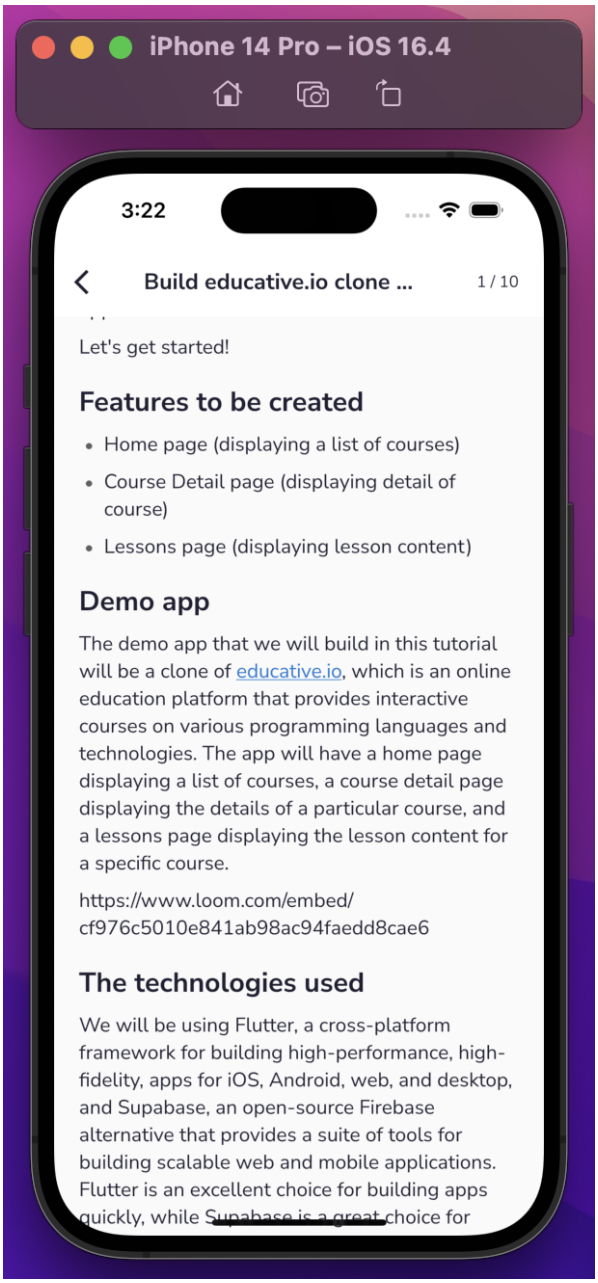
```
    listIndent: 16,
  );
}

// Markdown Builders function
Map<String, MarkdownElementBuilder> markdownBuilders(BuildContext context) {
  return {
    'code': CustomElementBuilder(context),
    'link-preview': CustomElementBuilder(context),
    'embed': CustomElementBuilder(context),
  };
}

// Custom Element Builder
class CustomElementBuilder extends MarkdownElementBuilder {
  CustomElementBuilder(this._context);

  final BuildContext _context;

  @override
  Widget? visitElementAfter(md.Element element, TextStyle? preferredStyle) {
    String text = element.textContent;

    if (element.tag == 'code') {
      String language = '';

      if (element.attributes['class'] != null) {
        String lg = element.attributes['class'] as String;
        language = lg.substring(9);

        return Container(
          color: Theme.of(_context).scaffoldBackgroundColor,
          padding: const EdgeInsets.only(top: 5, bottom: 10),
          child: ClipRRect(
            borderRadius: BorderRadius.circular(5),
            child: SyntaxView(
              code: text,
              syntax: Syntax.DART,
              syntaxTheme: SyntaxTheme.ayuDark(),
              fontSize: 14,
              withZoom: false,
              withLinesCount: false,
            ),
          ),
        );
      }
    }

    // link preview
    if (element.tag == 'link-preview') {
      return Padding(
        padding: const EdgeInsets.only(top: 5, bottom: 10),
        child: AnyLinkPreview(
          link: text,
```

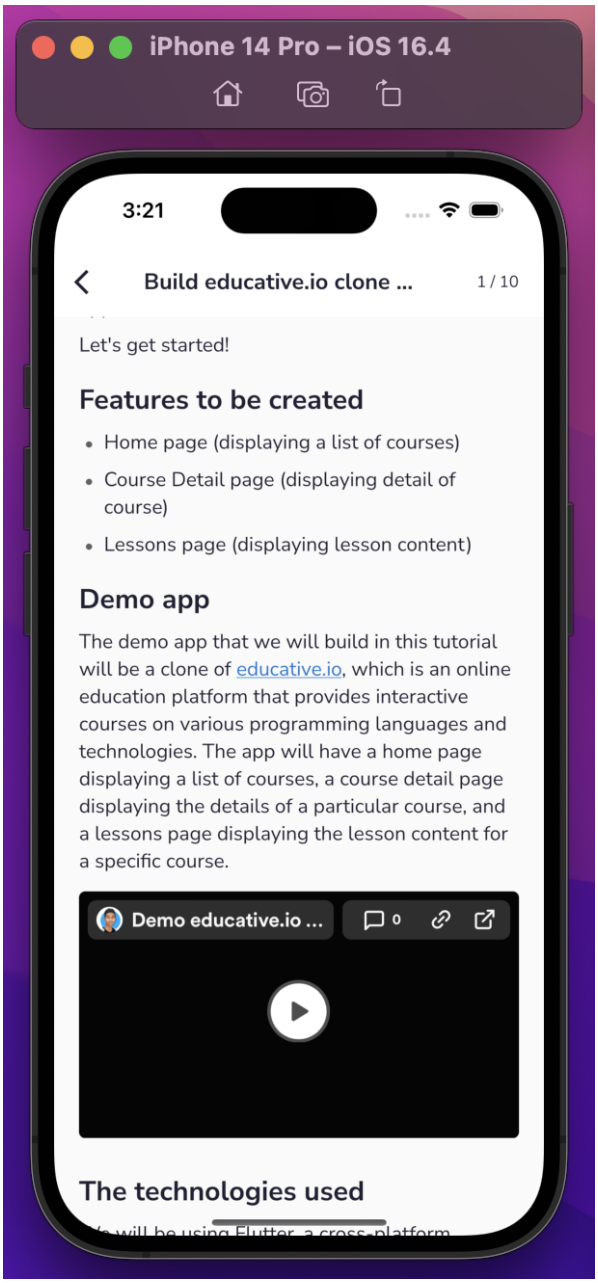
```
          borderRadius: 10,
          errorBody: 'Oops! Something went wrong.',
          errorTitle: 'Error',
          removeElevation: true,
          titleStyle: MyTypography.body,
          bodyStyle: MyTypography.bodySmall.copyWith(
            color: Colors.grey,
          ),
          cache: const Duration(days: 1),
          backgroundColor: Colors.white,
          placeholderWidget: Container(
            height: 100,
            width: 100,
            color: Colors.grey[200],
          ),
          errorWidget: Container(
            height: 100,
            width: 100,
            color: Colors.grey[200],
          ),
        ),
      );
    }

    if (element.tag == 'embed') {
      return Padding(
        padding: const EdgeInsets.only(top: 5, bottom: 10),
        child: ClipRRect(
          borderRadius: BorderRadius.circular(5),
          child: HtmlWidget(
            '<iframe src="$text"></iframe>',
            customStylesBuilder: (element) {
              if (element.classes.contains('name')) {
                return {'color': 'red'};
              }
              return null;
            },
            onErrorBuilder: (context, element, error) =>
                Text('$element error: $error'),
            onLoadingBuilder: (context, element, loadingProgress) =>
                const CircularProgressIndicator(),
            onTapUrl: (url) => launchUrl(Uri.parse(url)),
            renderMode: RenderMode.column,
            buildAsync: true,
            textStyle: const TextStyle(fontSize: 14),
            enableCaching: true,
          ),
        ),
      );
    }

    return null;
  }
}
```

```
// Register custom inline syntax for link preview in markdown
List<md.InlineSyntax> markdownInlineSyntaxes = [
  LinksPreview(),
  EmbedSyntax(),
];

// Custom Inline Syntax for link preview
class LinksPreview extends md.InlineSyntax {
  LinksPreview() : super(r'''<link-preview>(.*)</link-preview>''');

  @override
  bool onMatch(md.InlineParser parser, Match match) {
    final tag = md.Element.text('link-preview', match[1]!.trim());
    parser.addNode(tag);
    return true;
  }
}

// Custom Inline Syntax for embed links
class EmbedSyntax extends md.InlineSyntax {
  EmbedSyntax() : super(r'''<embed>(.*)</embed>''');

  @override
  bool onMatch(md.InlineParser parser, Match match) {
    final tag = md.Element.text('embed', match[1]!.trim());
    parser.addNode(tag);
    return true;
  }
}
```

If you successfully follow step 5, we can run the app and see the different results before and after we custom inline syntax embed link.

Before: After:





Testing the App

If you successfully follow step-by-step this guide, now let's test our app. Run the application on an emulator or device using the terminal command. The command `flutter run` will build the app and install it on your device. The result will resemble the video below:

https://www.loom.com/share/eafce802fb324cf792231de4b29c3d07?sid=d60aa653-e6be-424f-9560-818d91b83946

Conclusion

In conclusion, we have successfully customized the styling of our Markdown texts in Flutter. We have added custom inline syntax for link previews and embed links, as well as defined custom rendering to display them. With these changes, our Markdown texts are more dynamic and user-friendly, making them a great addition to any Flutter project.