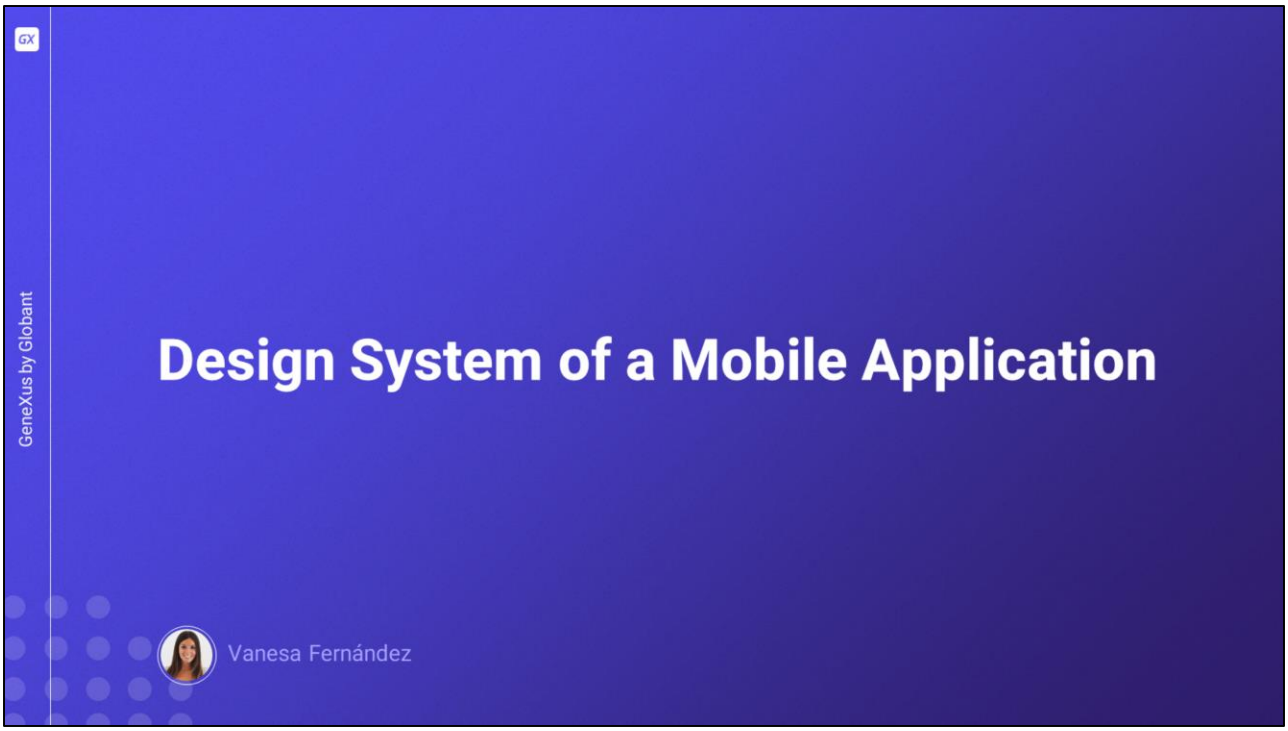
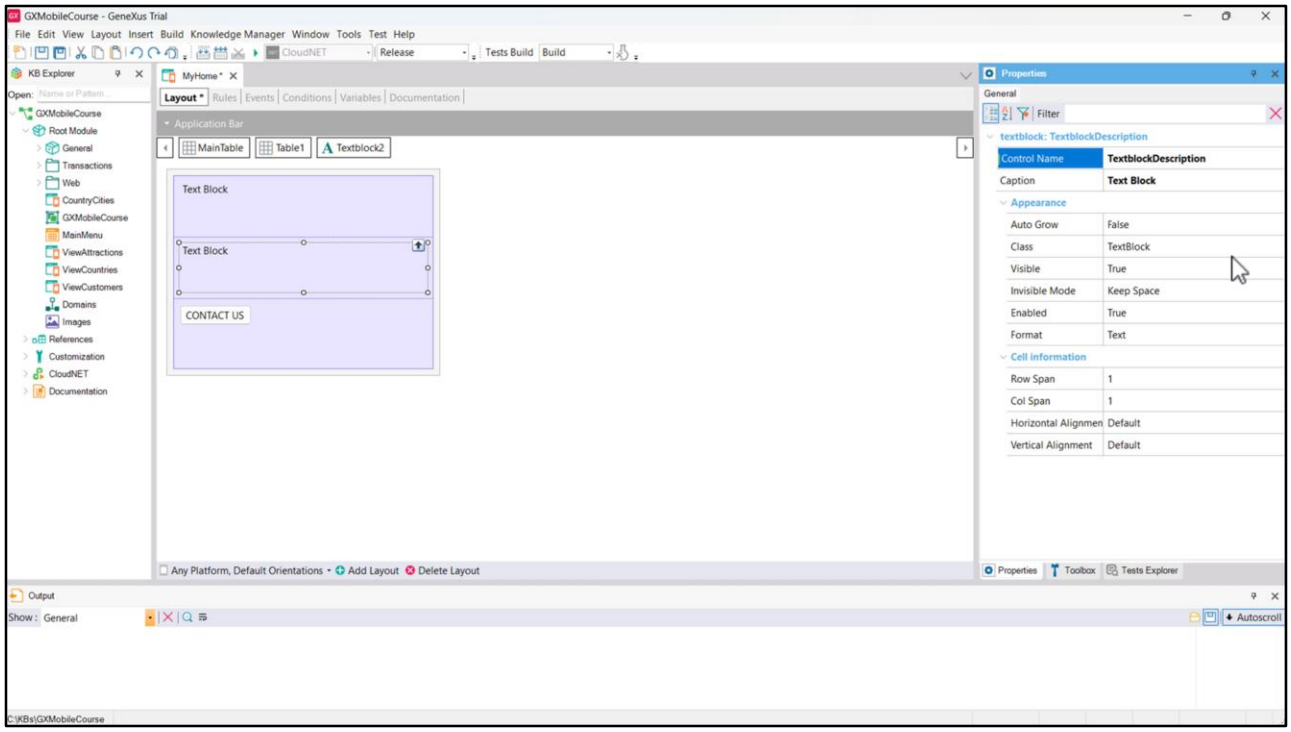
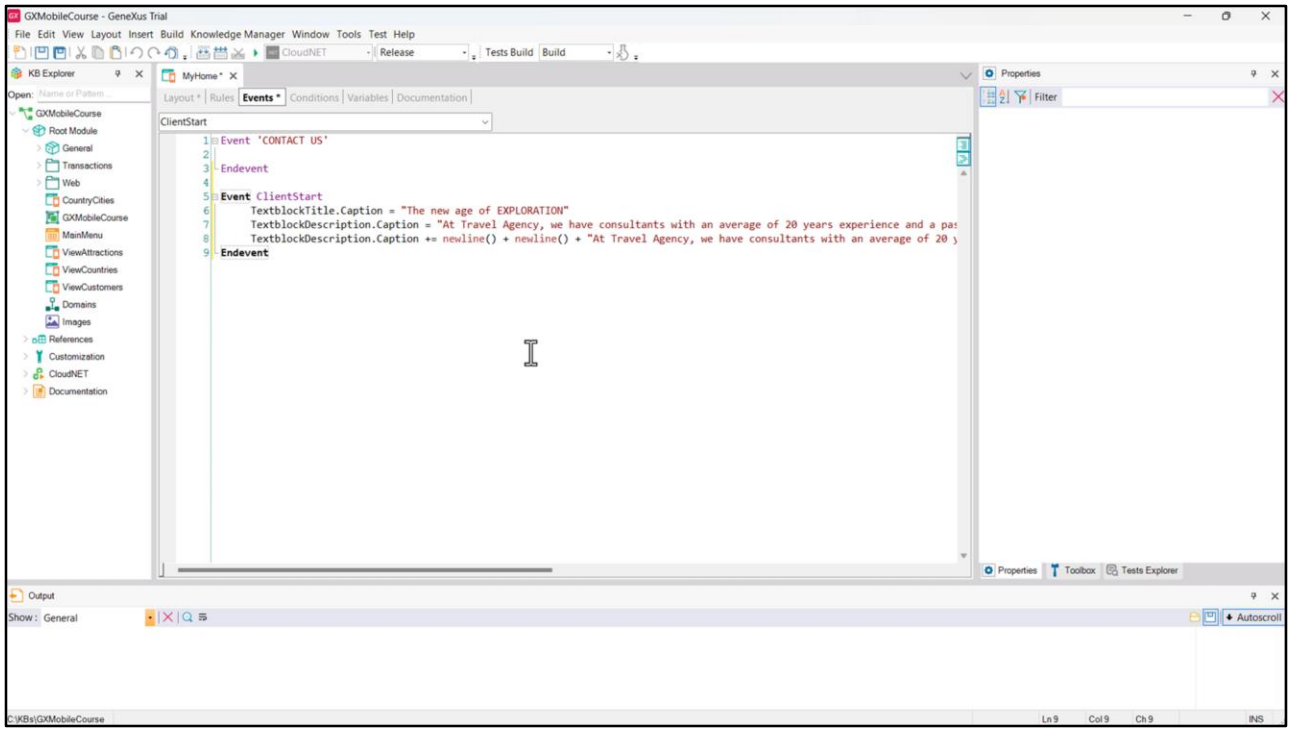


In this video, we will study how to work with the Design System object to add a design to our application. We will see, among other things, how to use tokens, styles, and properties, as well as how to incorporate fonts and work with light and dark modes.

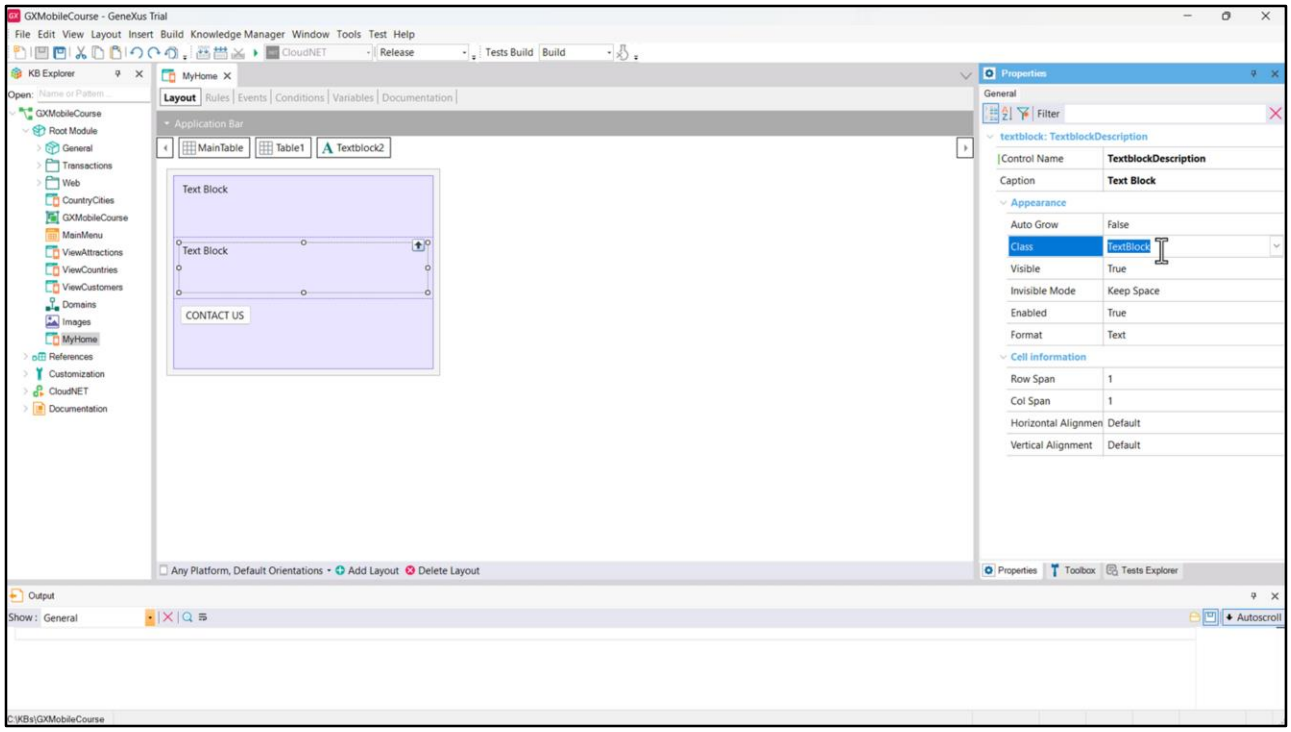


Just to do some tests, let's create a Panel named MyHome, and set the Main program property to True in order to run it easily as it won't have any dependencies.

Let's add a table, and 3 elements inside it: 2 textblocks, and a button labeled "CONTACT US". Let's change the name of the control of the first textblock to TextblockTitle... and the second one to TextblockDescription.

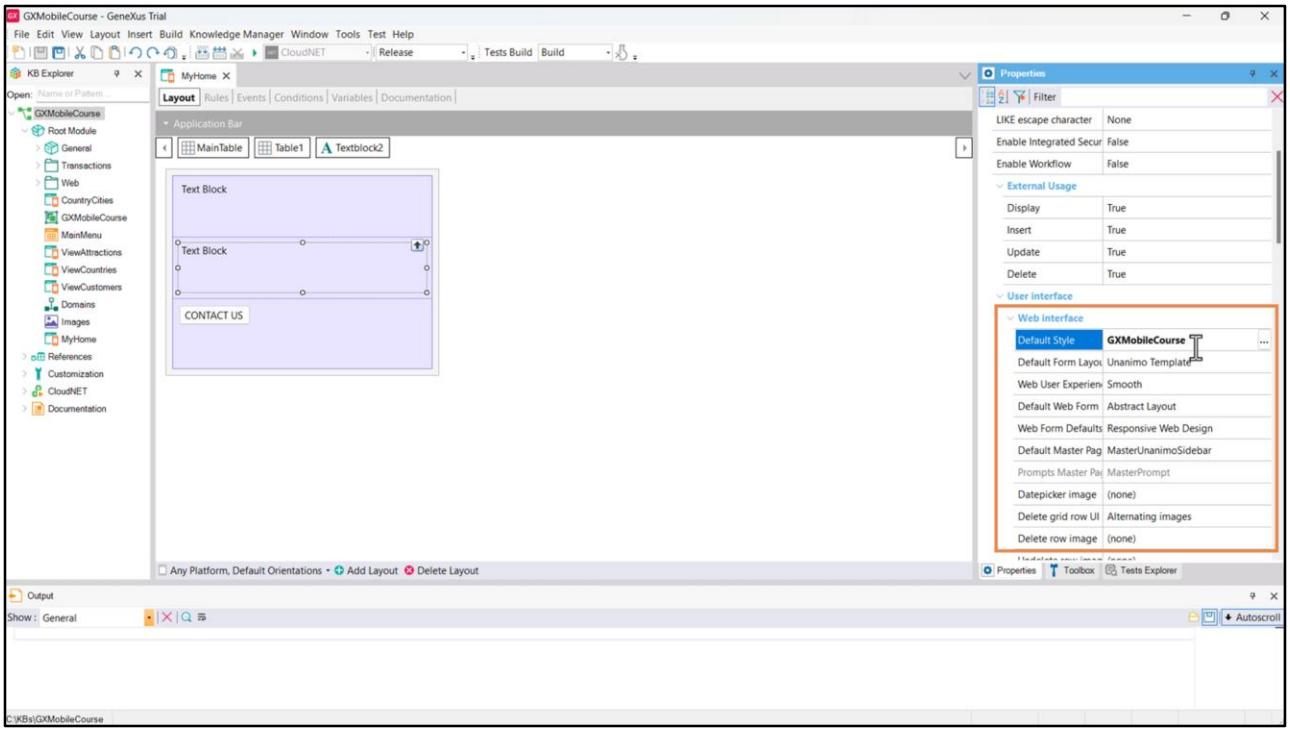


Let's go to the events and add ClientStart, which is going to be executed when the Panel opens. Let's load the paragraphs separately so we can add the new line, which allows us to set the spacing between them.

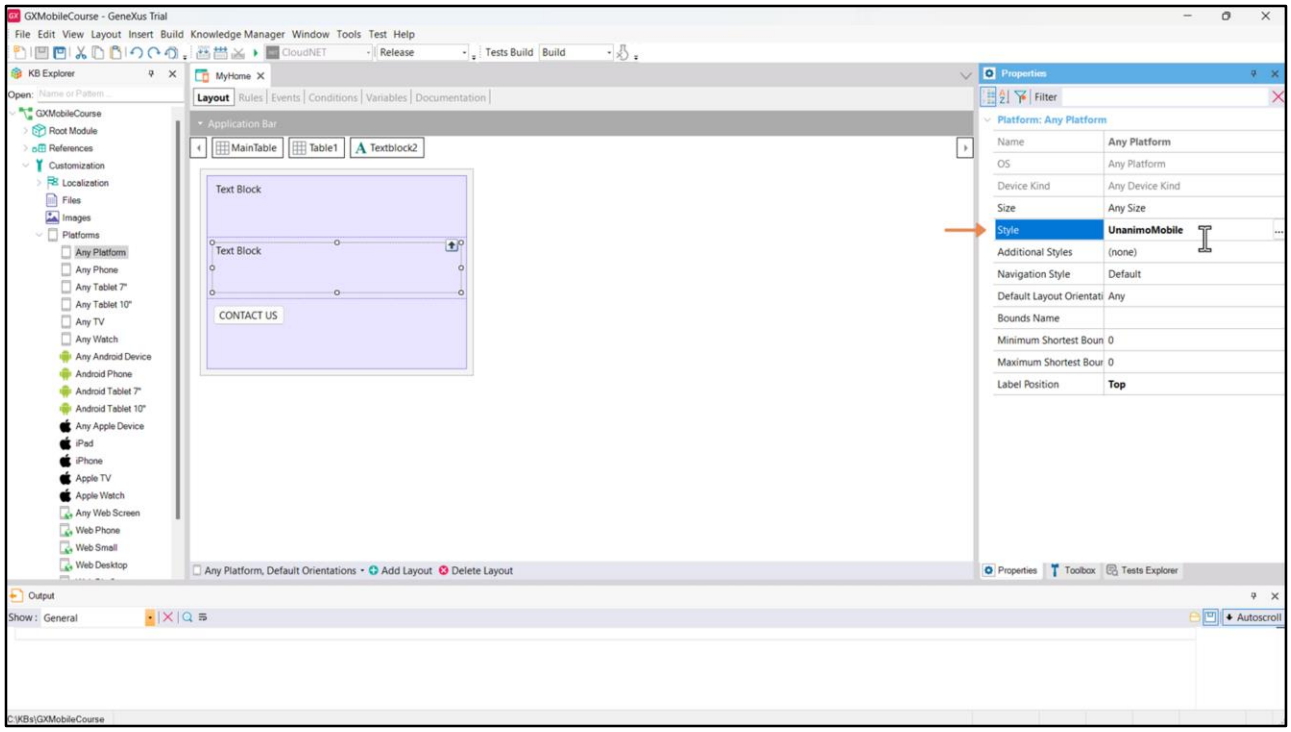


If we edit one of the TextBlock controls we added, we can see that it has the Class property, which by default has the TextBlock value.

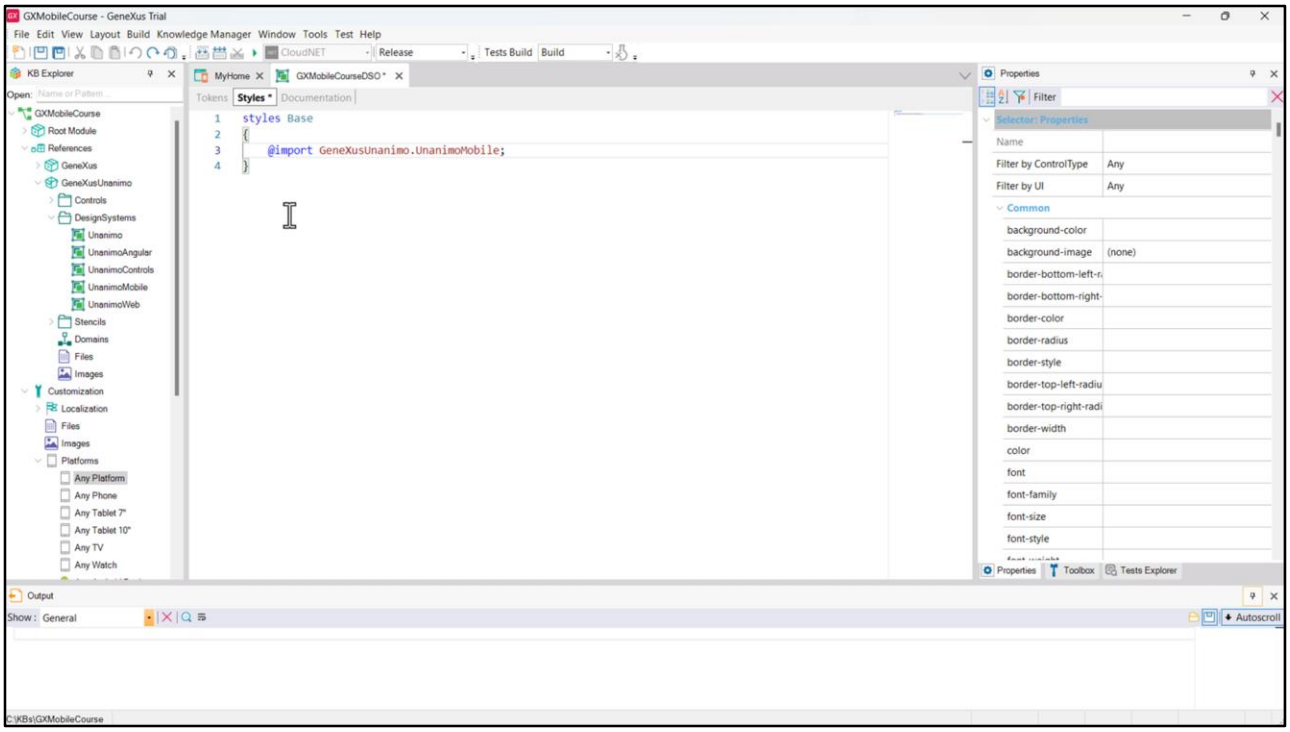
These class definitions, which will contain the characteristics of each layout element, will be defined and centralized in one object: the Design System.



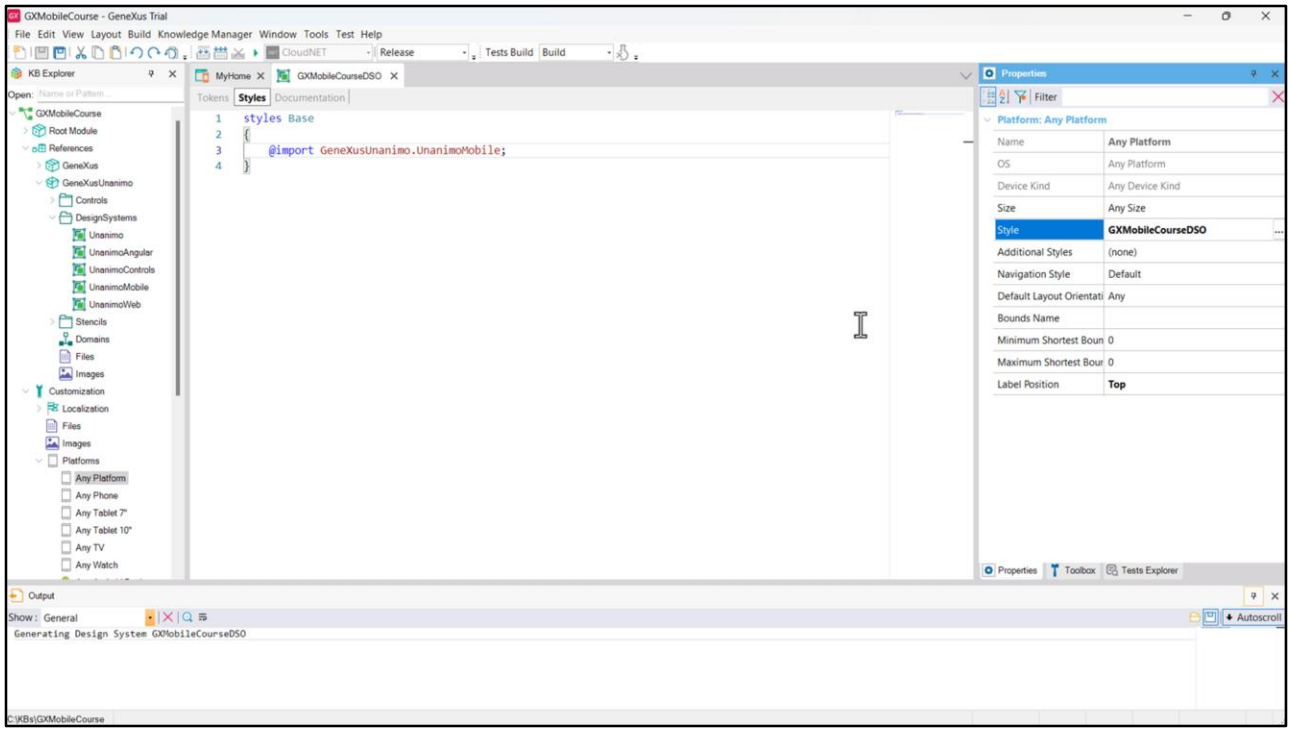
Every KB is created by default with a predefined design system that can be seen when we edit the version properties, in the Default Style. The style corresponds to the Design System object containing all these definitions. By default, this Design System object is created in the KB and takes the same name as the KB. This DSO is configured below the Web interface property group: this means it controls application development when Web Panels are used. Since we are working on a native mobile application, this is not our case, because we are using Panels instead of Web Panels.



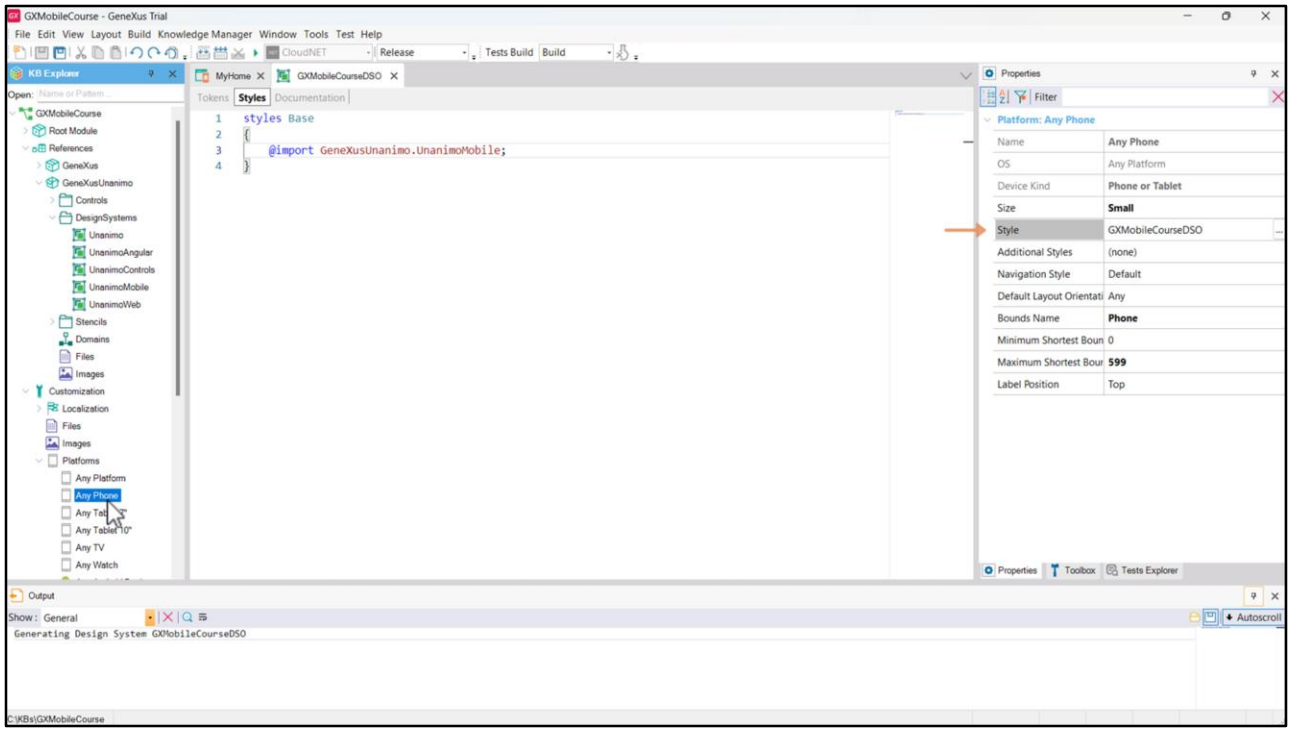
To take control of the design when we are in a native environment, we must make changes where the platform definitions are located, which is below the Customization node. In the Any Platform node, we can see that the Style property is taking the Design System object named UnanimoMobile. Unanimo is already a Design System –the one predefined by GeneXus– created in the GeneXusUnanimo node. It is the one that by default will take control of the design of the entire application if we don't define something else.



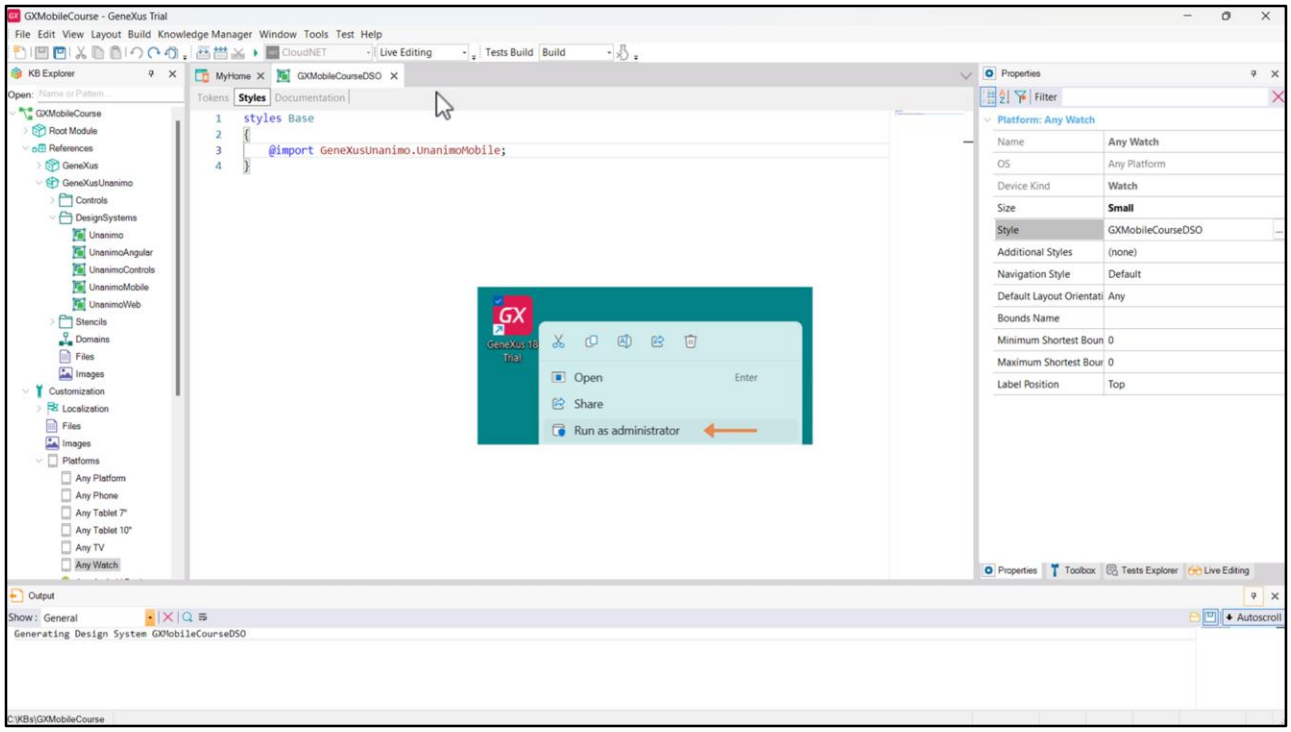
As we are going to develop from scratch, we will create a new object of Design System type that will apply the styles to our native application. We will call it GXMobileCourseDSO, and we will indicate here that this Design System must inherit from the definitions of the UnanimoMobile Design System. To do so, in the Styles tab we write the following:



We will use the Tokens and Styles tabs to make our definitions, but first, let's indicate GeneXus that we want our KB to use the Design System we've just created. Let's go to Any Platforms and change the Style property.



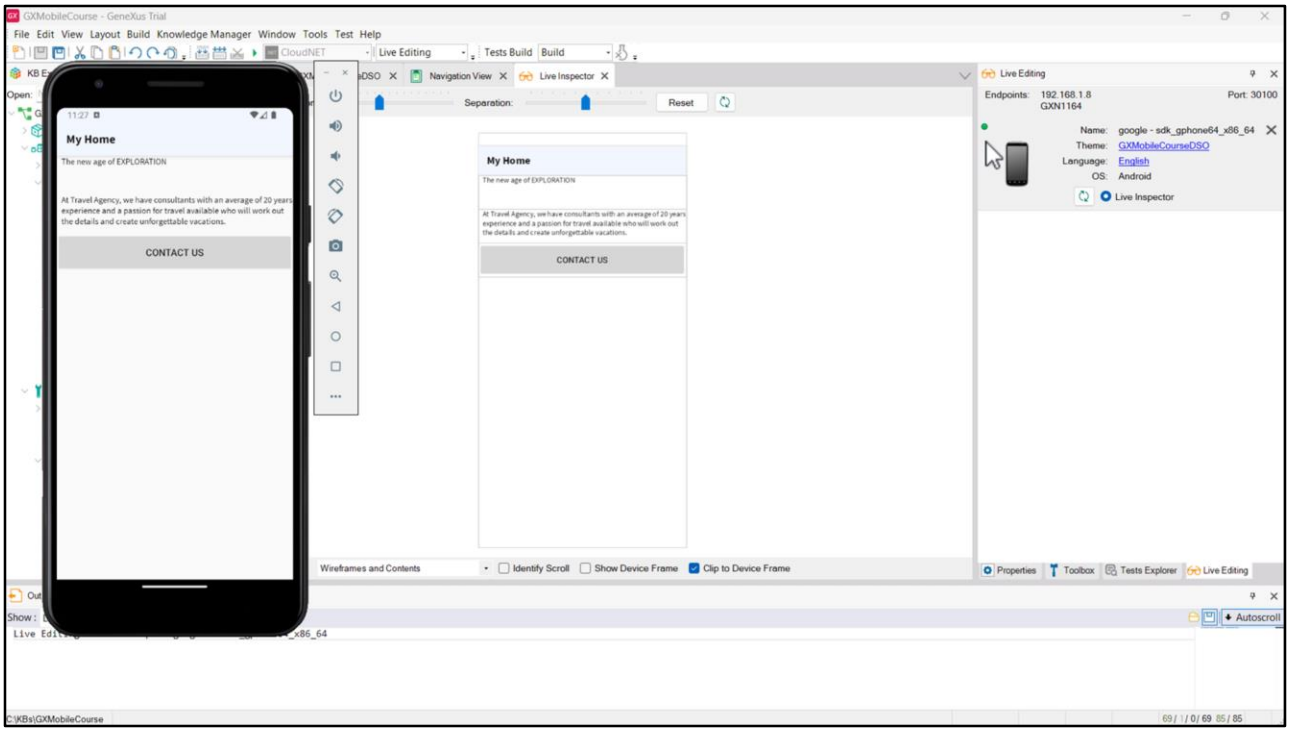
If we look at the different sizes and platforms, we can see that they inherit from the one we've just configured, but we can change this using a Design System object for each screen size and platform.



Before running our Panel, to see how it looks before we start changing its style, let's configure the use of the Live Editing tool. But... what is Live Editing? When prototyping an application, one of the most time-consuming tasks is polishing the Look & Feel (or User Interface, UI) and the User Experience (UX). To simplify this task, GeneXus has a feature that allows you to change the state of the application in real time from the IDE without saving the modified objects.

To prototype in Live Editing mode, we simply change the Release value in this ComboBox to Live Editing and run the application again. If you are using the Trial version of GeneXus, you will have to run it with Administrator permissions.

In this case, since it is the first time that the Live Editing mode is enabled, it will be necessary to compile the application again, but this won't be necessary for the next changes, which will simply be reflected automatically in the emulator and in the GeneXus IDE.

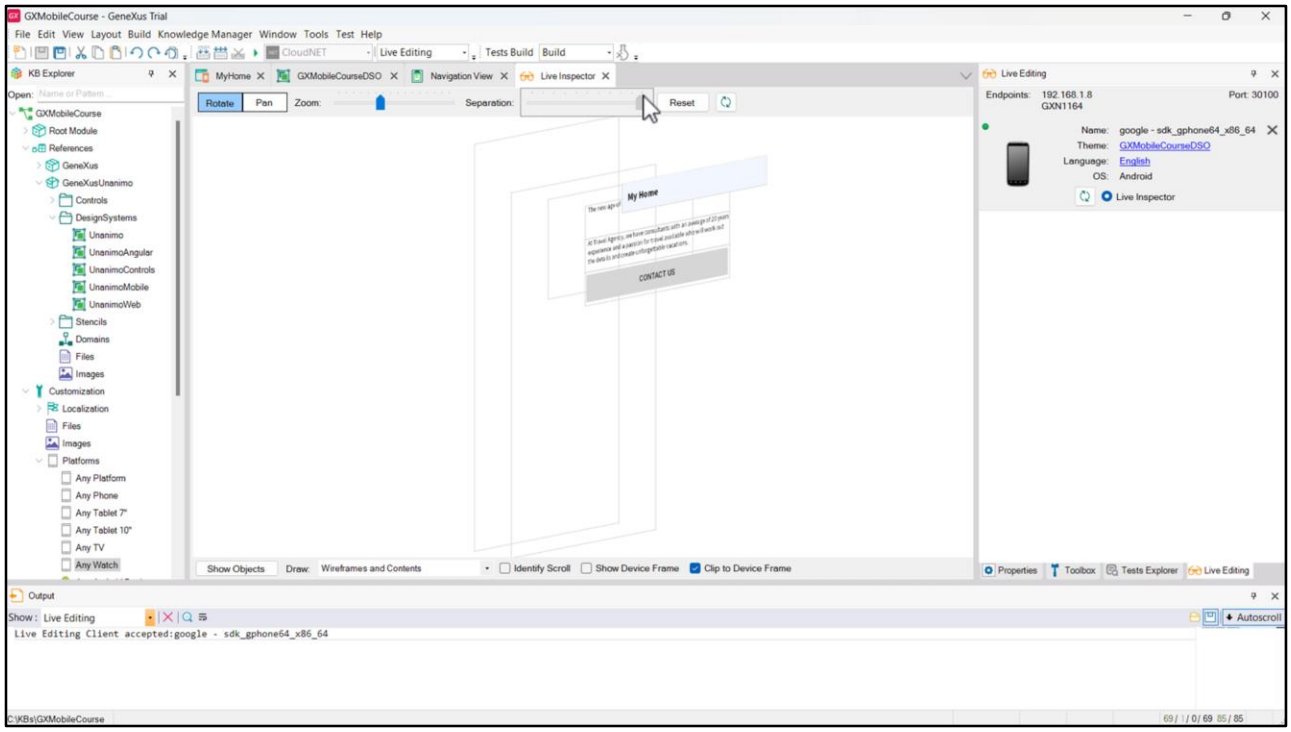


We run it.

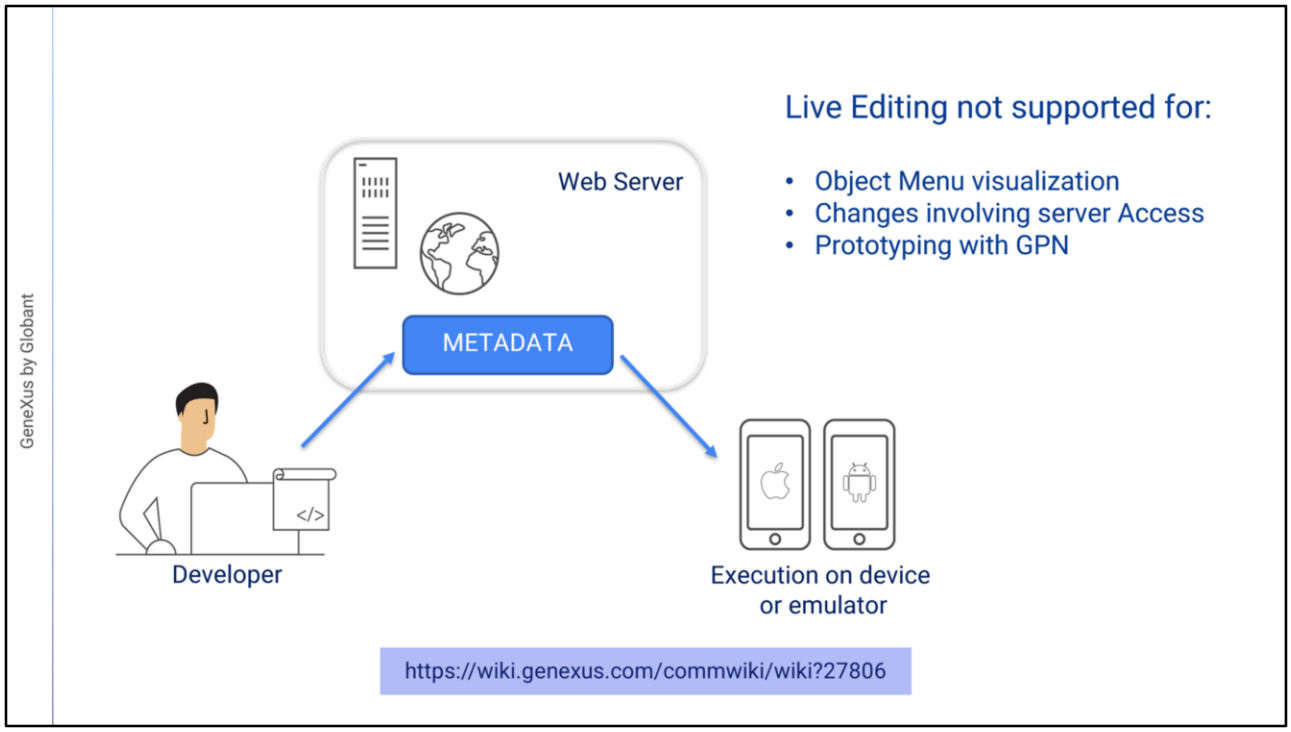
We already have the application ready again in the emulator, only that now we enable Live Editing.

In the IDE we can see that a Live Editing tab has been enabled, containing information of the devices that are connected; in this case, the emulator we had running. We can see the device data, such as the name, the Design System that is being used in the application and the language. The green dot indicates that the device is connected to Live Editing.

We can also see that a new window called Live Inspector has been opened. This is one of the main advantages of Live Editing. Although there is a restriction and we can't see when a menu object is running, the rest of the panels are visible. We just hover the mouse over the on-screen controls to view their name and click on them to see their main data.



Moving the mouse we can rotate the image in any direction, and see layers that show where each control is located. These layers allow us to select them in a much simpler way, since in general the sections or tables are superimposed. With the Separation slider we can increase or decrease the space between layers; with Zoom we can zoom in or out, and if we select Pan instead of Rotate, we can move the content with the mouse, without Rotating it. With Reset we return to the default display.

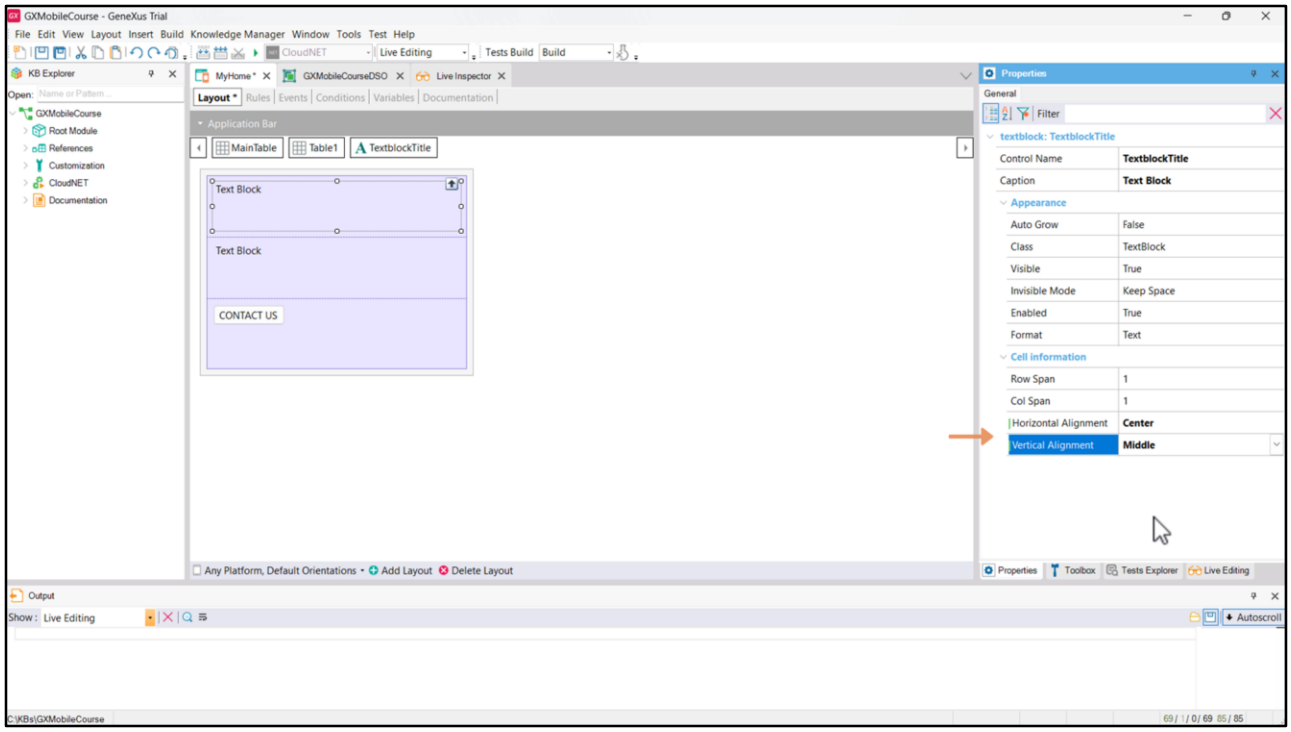


But, how does Live Editing work? When we use Live Editing, the Server will be “listening” to the changes we make and will replicate these changes on the Metadata that is accessed by the mobile device (in our case, the emulator) and used to draw the screen on the device and define its behavior.

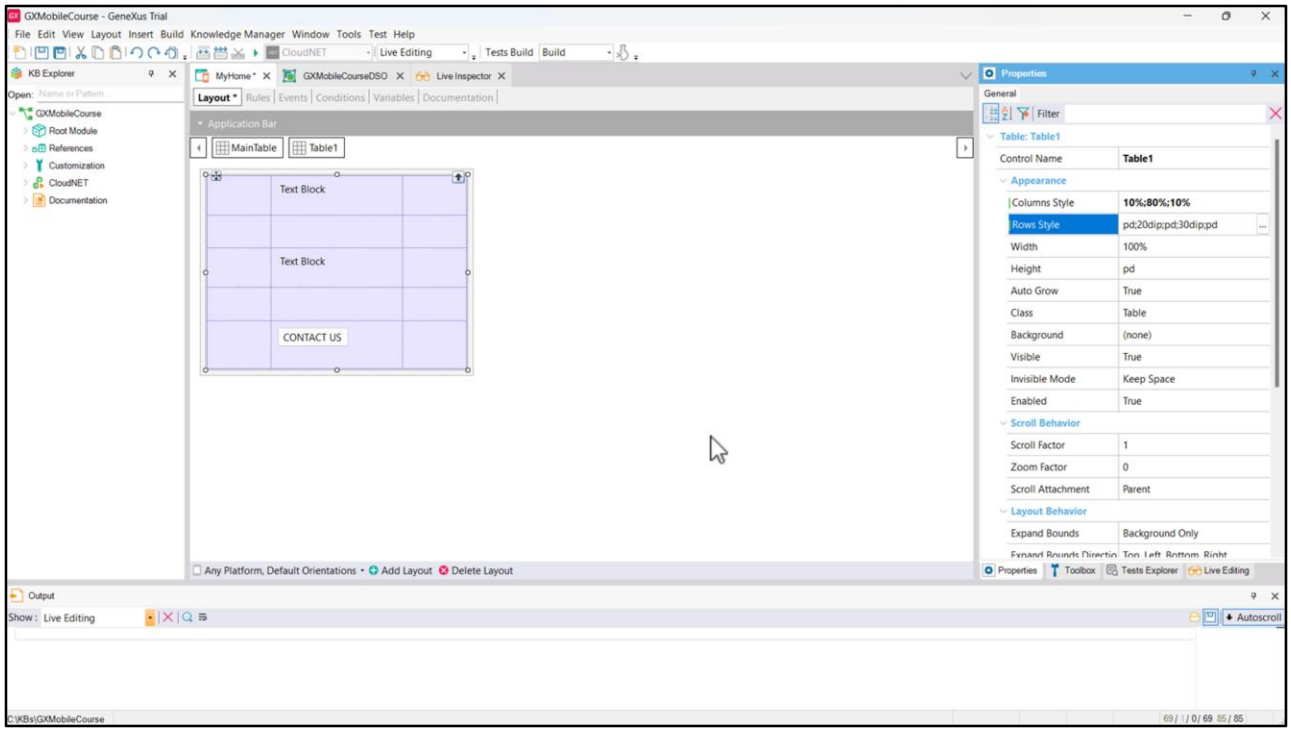
Live Editing has some use restrictions, for example:

- It doesn’t work for menu objects.
- Changes that involve server access cannot be displayed. For example, if we add an attribute to the form, the value must be retrieved from the database in a server event (Start, Refresh, or Load), so it will be necessary to compile the application again.
- Since live editing only works with the compiled application, it doesn't work if you are prototyping with the GPN (GeneXus Projects Navigator).

For more details on the use of Live Editing and Live Inspector, visit the GeneXus wiki: https://wiki.genexus.com/commwiki/wiki?27806

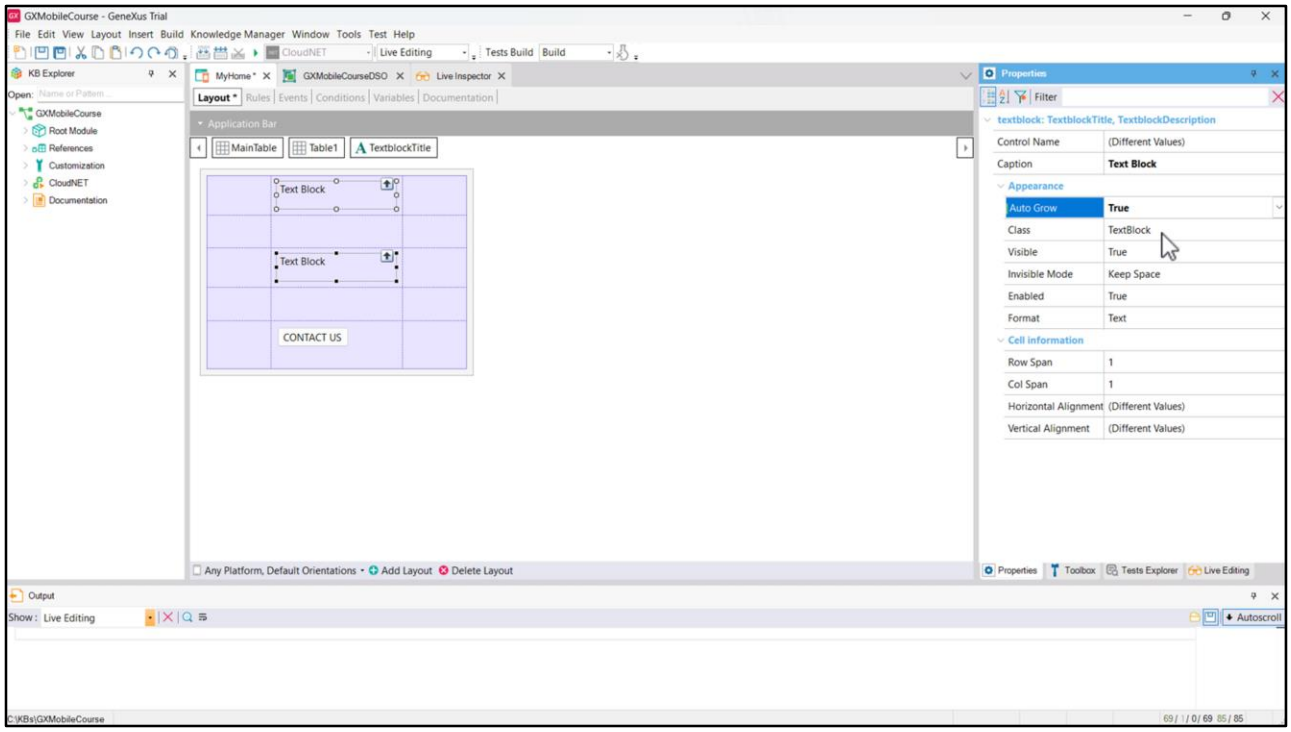


Let's go back to our example. We want the title to be centered in the cell that contains it, so we can use the Horizontal Alignment and Vertical Alignment properties with the values Center and Middle, respectively.

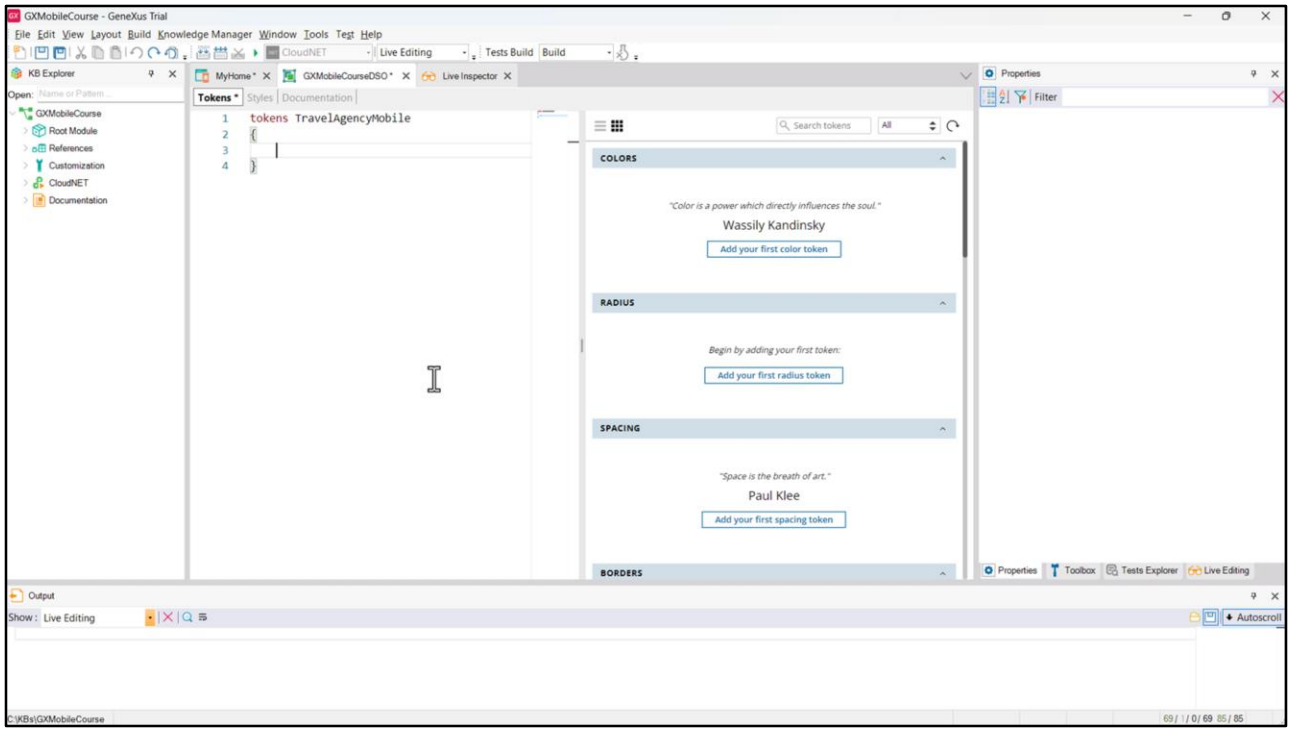


Let's also change the Horizontal Alignment property of the TextblockDescription so that the text is justified.

Let's add two more columns to the table, one to the left and one to the right of the content, and two more rows, one below the title and one above the button, in order to generate spaces that make the information look better. In the Columns Style property, we can see that each column takes up 33% of the total width of the table. Let's change the values so that the first column takes up 10% of the total width, the second column (which contains the content) takes up 80%, and the last column takes up the remaining 10%. Now, let's change the values of the Rows Style property, so that the second and fourth columns –which are the separators– take up 20 and 30 dips, respectively.

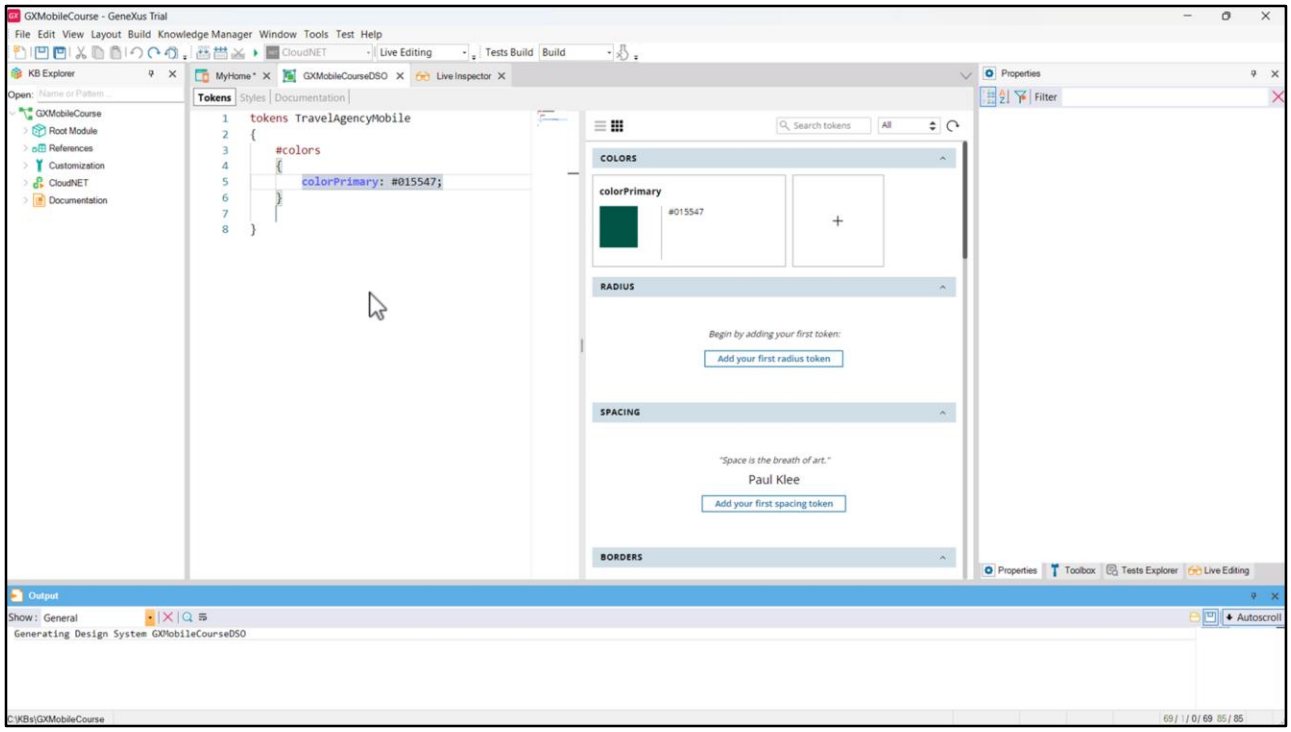


Let's also set the Auto Grow property of the cells containing the Textblocks to True. In this way, if the predefined space for each cell is not enough, the cells will expand so that all the content is visible.



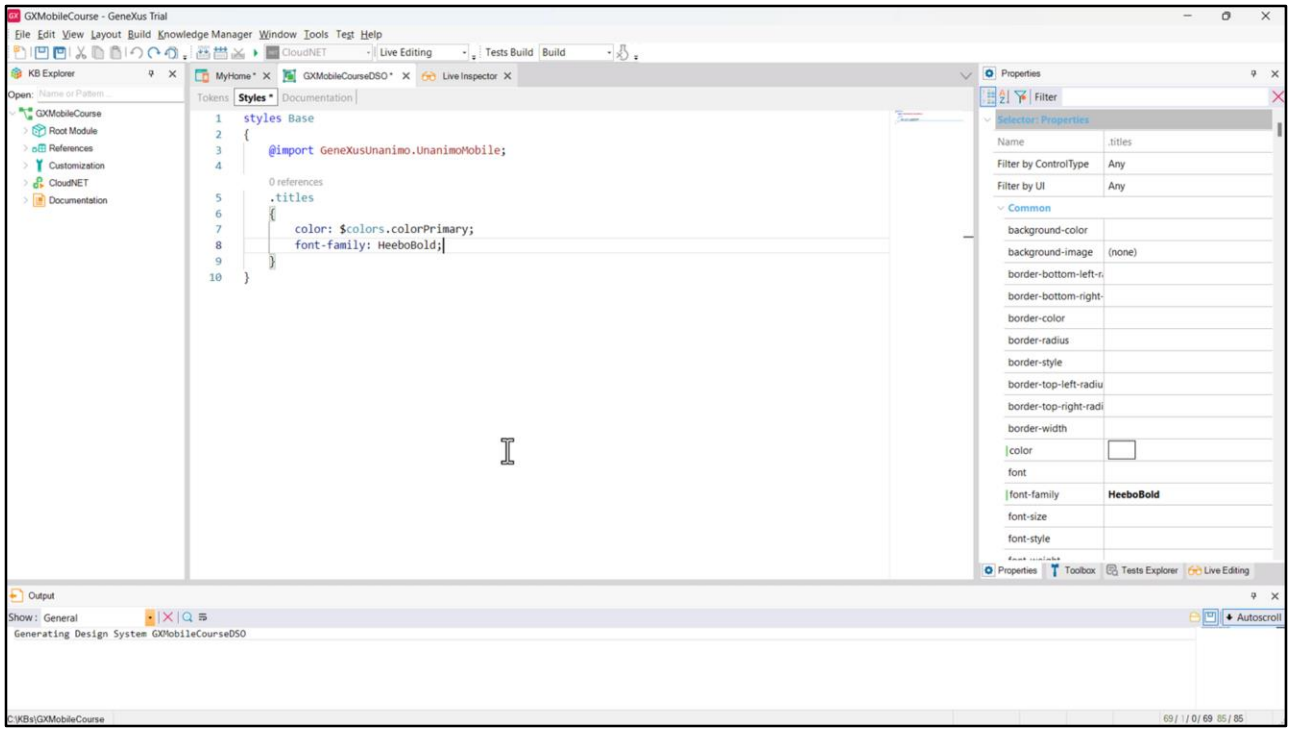
Now let's start to change the style of the Panel controls. First, we change the color and font type of the title. We could also set a centered alignment in the DSO, but it won't be necessary because we've set it in the properties and they belong to a hierarchical order above the DSO: that is to say, if we set the centered alignment in the DSO, the values of the properties would override them.

Let's give a name to the set of tokens we will create in order to keep the DSO organized: let's call it TravelAgencyMobile.



Let's start by defining a color constant: we will give it an identifying name, for example, Primary. For all the colors to be easily identifiable just by looking at the code, we will keep the same naming convention for every color constant we create, and the name will start with the word *color*. If we already know the hexadecimal value of the color (for example, because the design team gave it to us in a Figma file) we enter it here. We will use this dark green for the title.

On the right there is another editor that is synchronized with the one on the left; that is to say, we can work in one or the other and what we do in one will be reflected in the other.



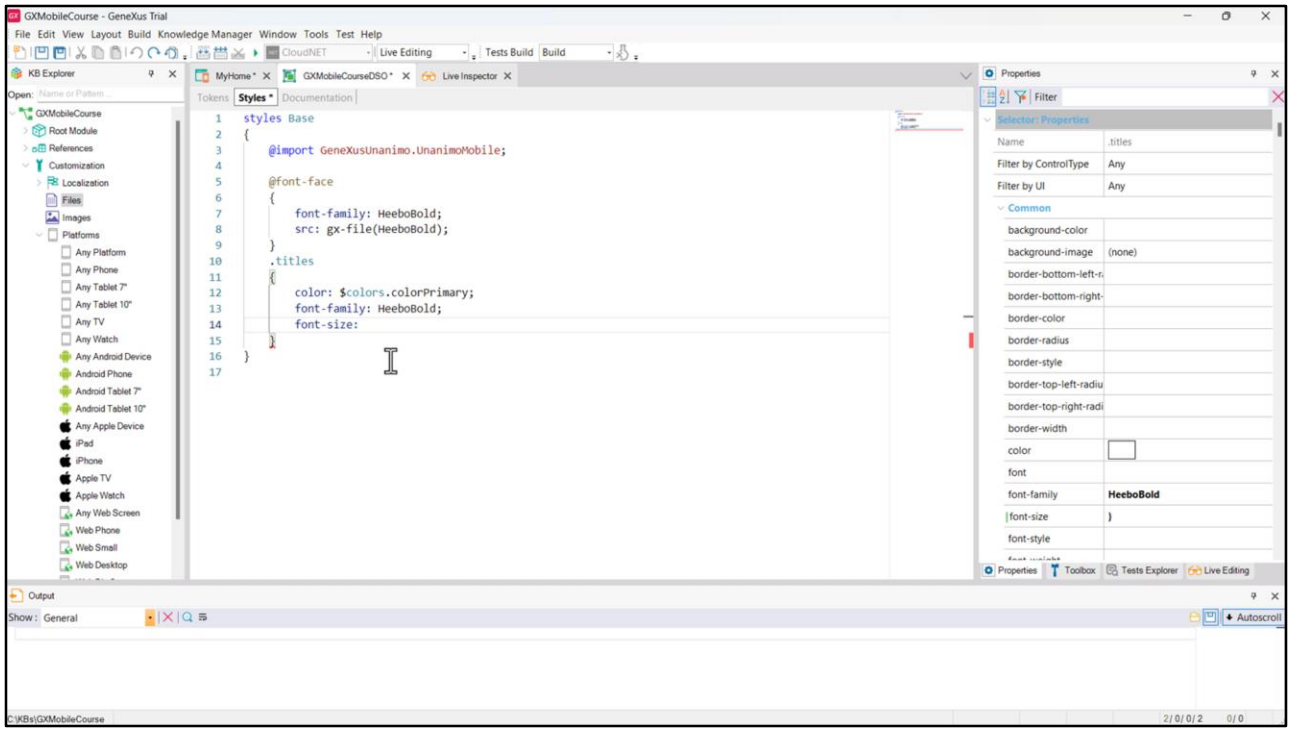
In the Styles tab, we will create the class that will style the content.

We will name it *titles*, and we will add the necessary CSS properties to establish the characteristics of this class.

The color to be applied is that of the token that we have just defined, so we write:
color: $ (which allows us to reference a token) colors (because it is going to be a color token).colorPrimary;
In this way, we indicate that every control that has this class associated with it will have this color.

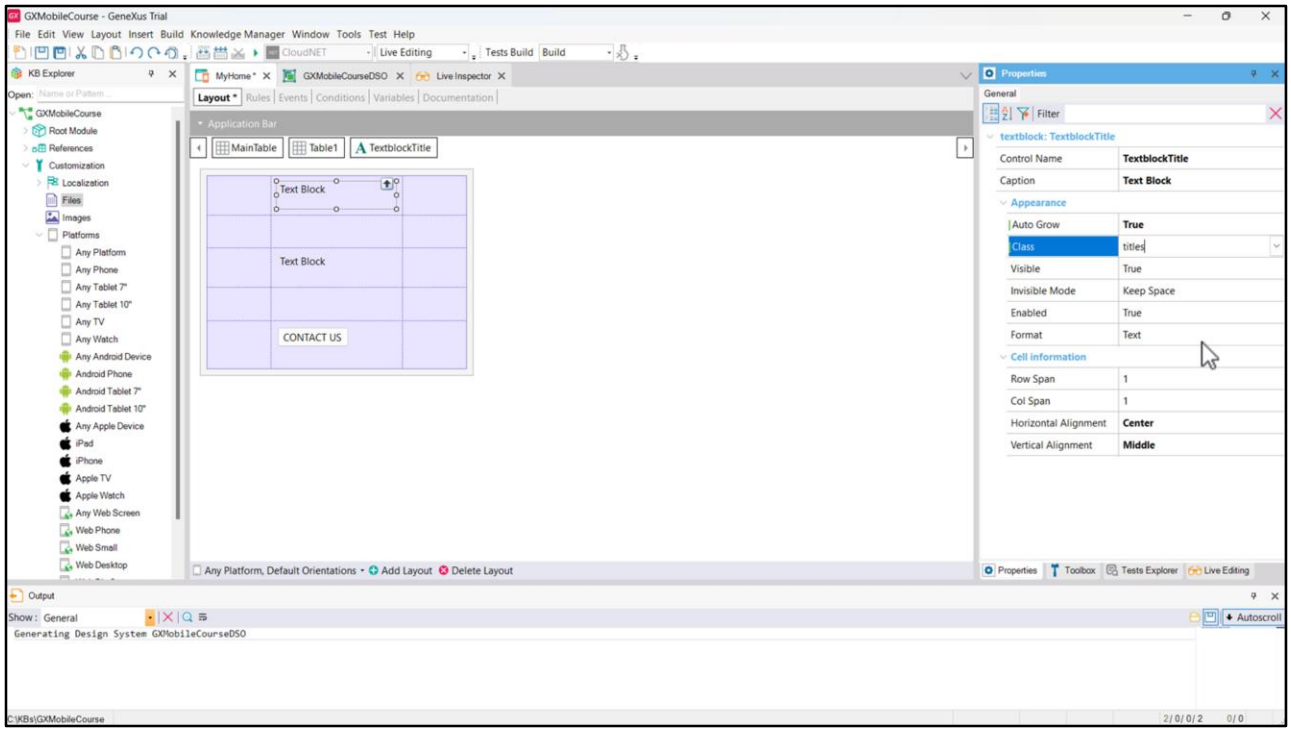
Let's specify now the font family to be used:
font-family: HeeboBold;

This font family is not part of the predefined ones; that is, those understood by default by browsers. This means that we must define in the DSO what that font is and where it is obtained from. To include it, we will have to import the font file in the KB to be able to reference it in the font-family property of the DSO.

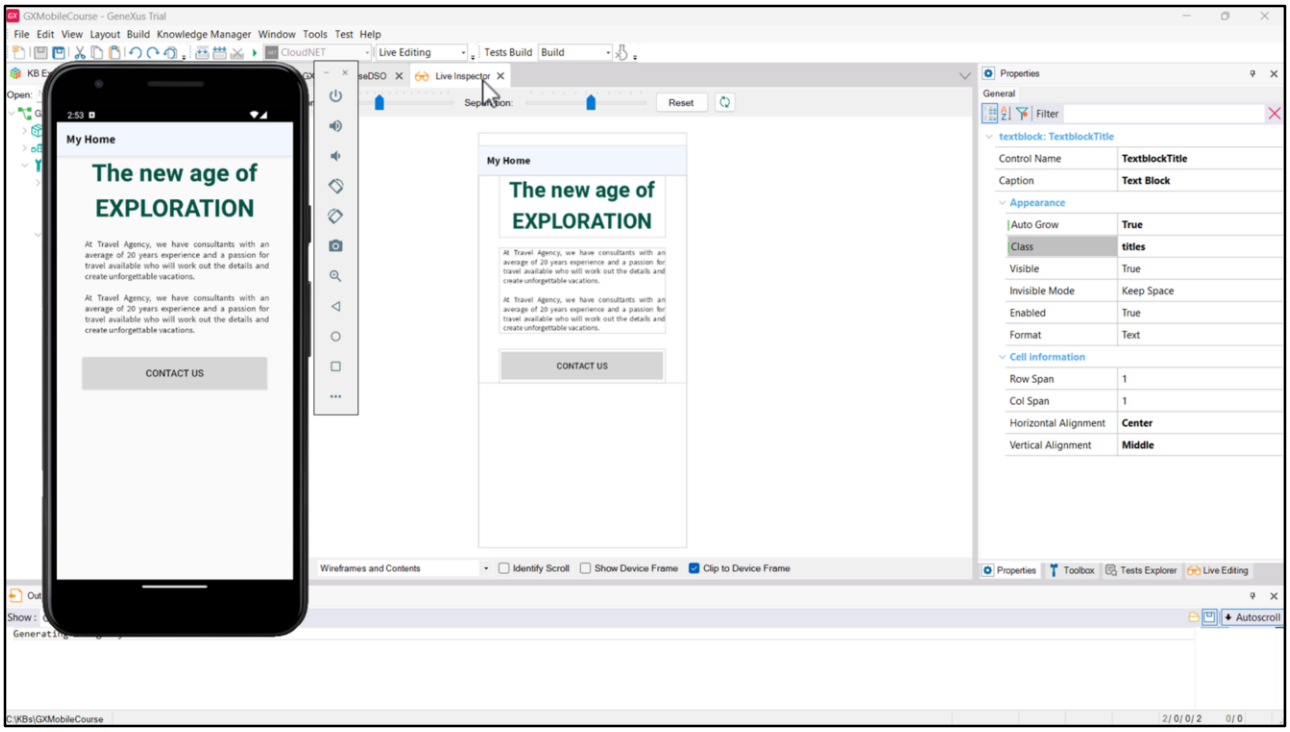


We go to Customization → Files → New file, create the file called HeeboBold and choose it from its location. Here we enter the name that the font family will have, and in the src property we indicate where the file is taken from. With the gx-file() function we retrieve the file accessible in our KB. This indicates the DSO to include this font, but then it is used in the class reference through the font-family property that we defined for our titles class.
font-family: HeeboBold;

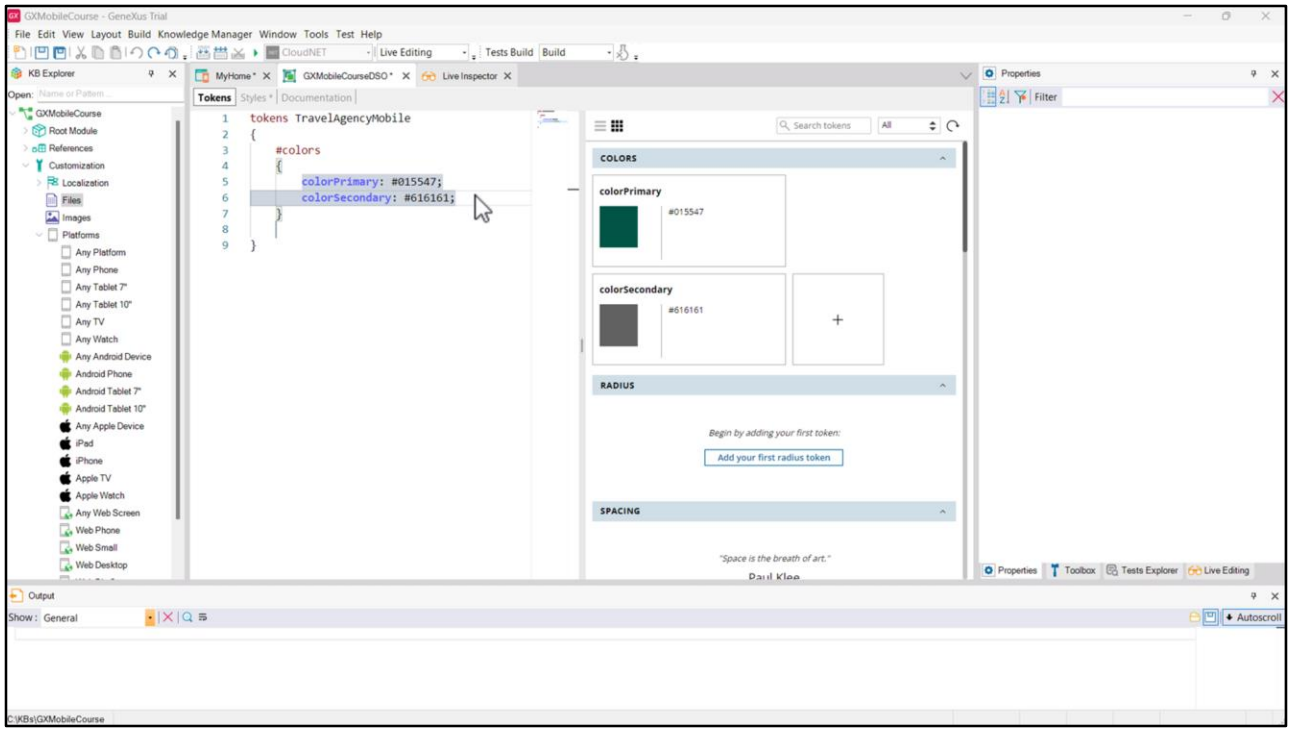
With the font-size property we indicate the size we want for the title.



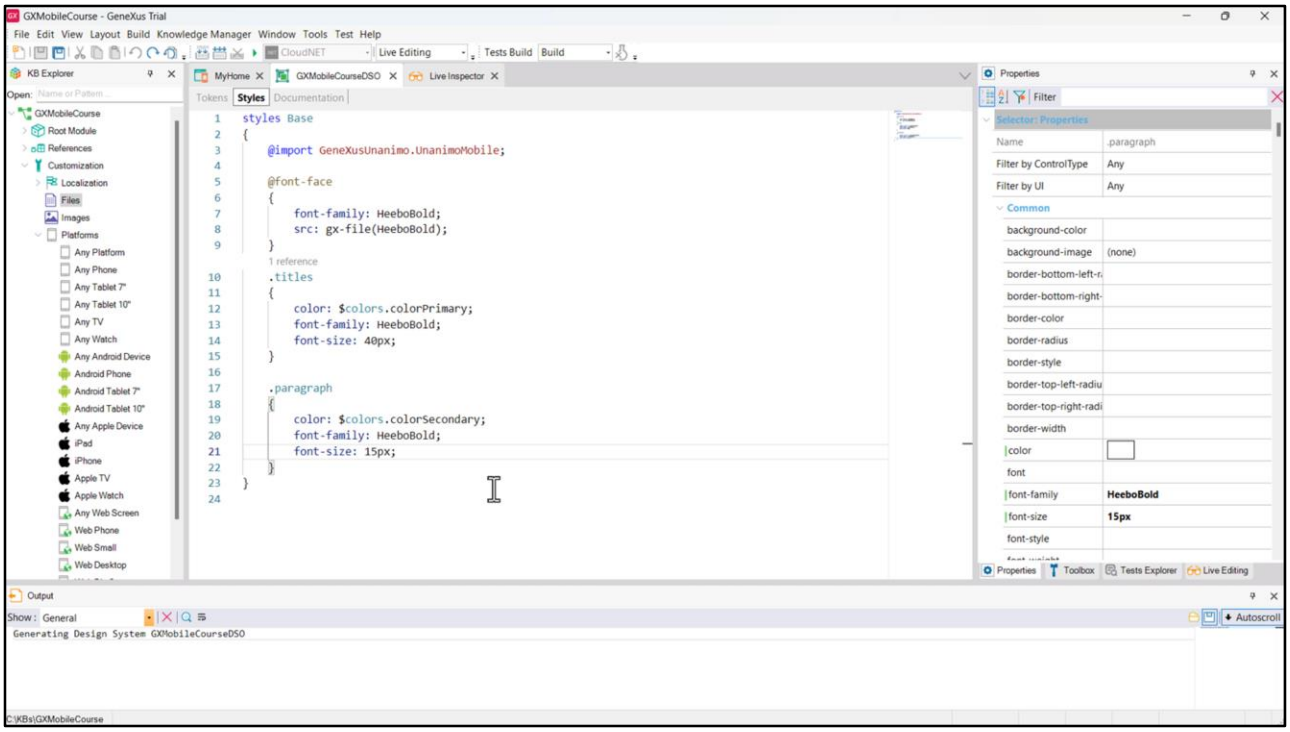
The next step is to set the Textblock control to take the *titles* class we've just defined.



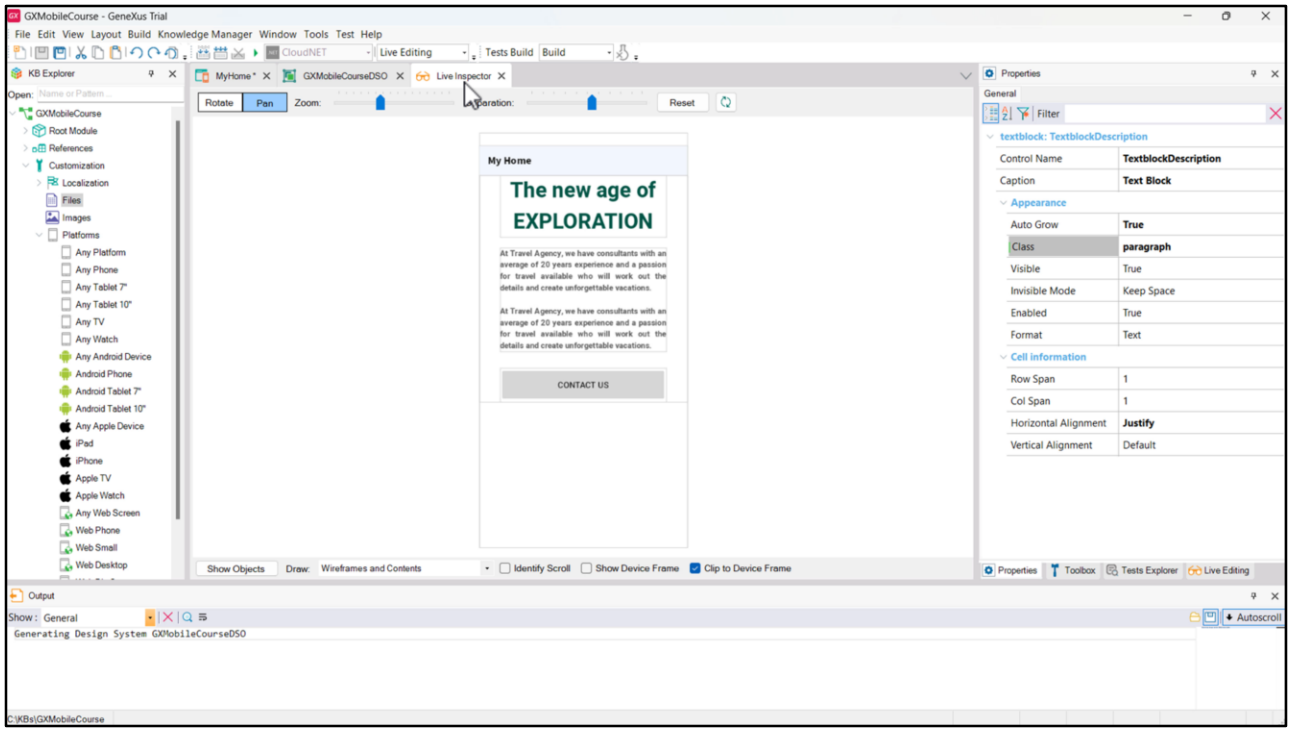
Live Editing already shows what we did in our DSO.



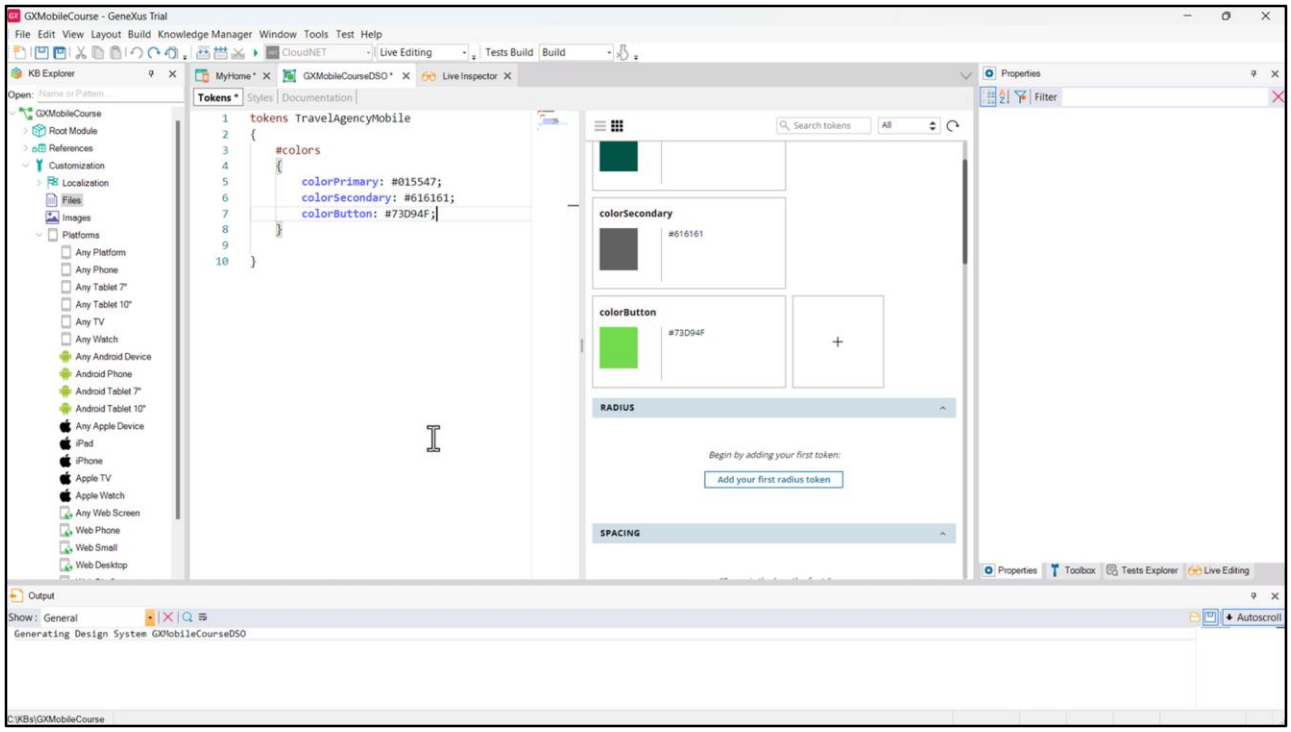
Let's do the same for the information text. Let's create another color token and name it colorSecondary; this time it is dark grey...



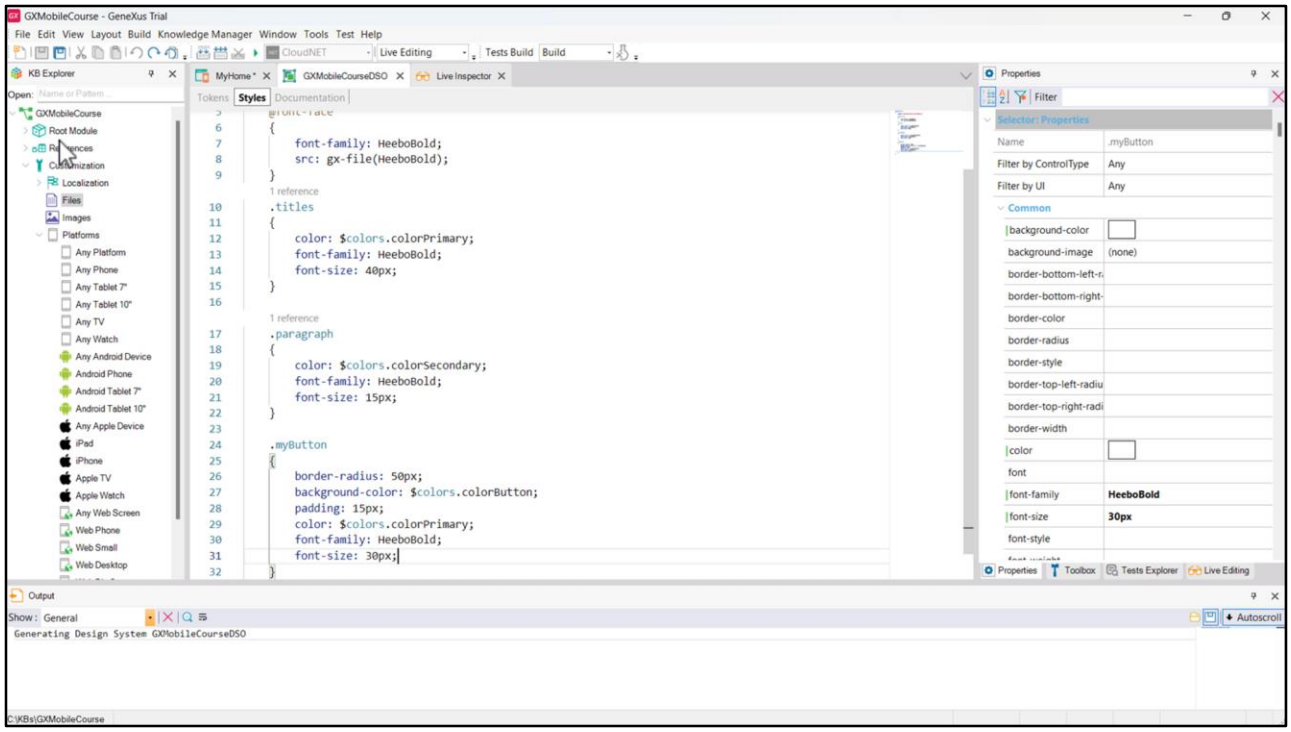
Also, a class named *paragraph* to which we set the following properties:



Let's assign the *paragraph* class to the TextblockDescription control, and see how the changes look.



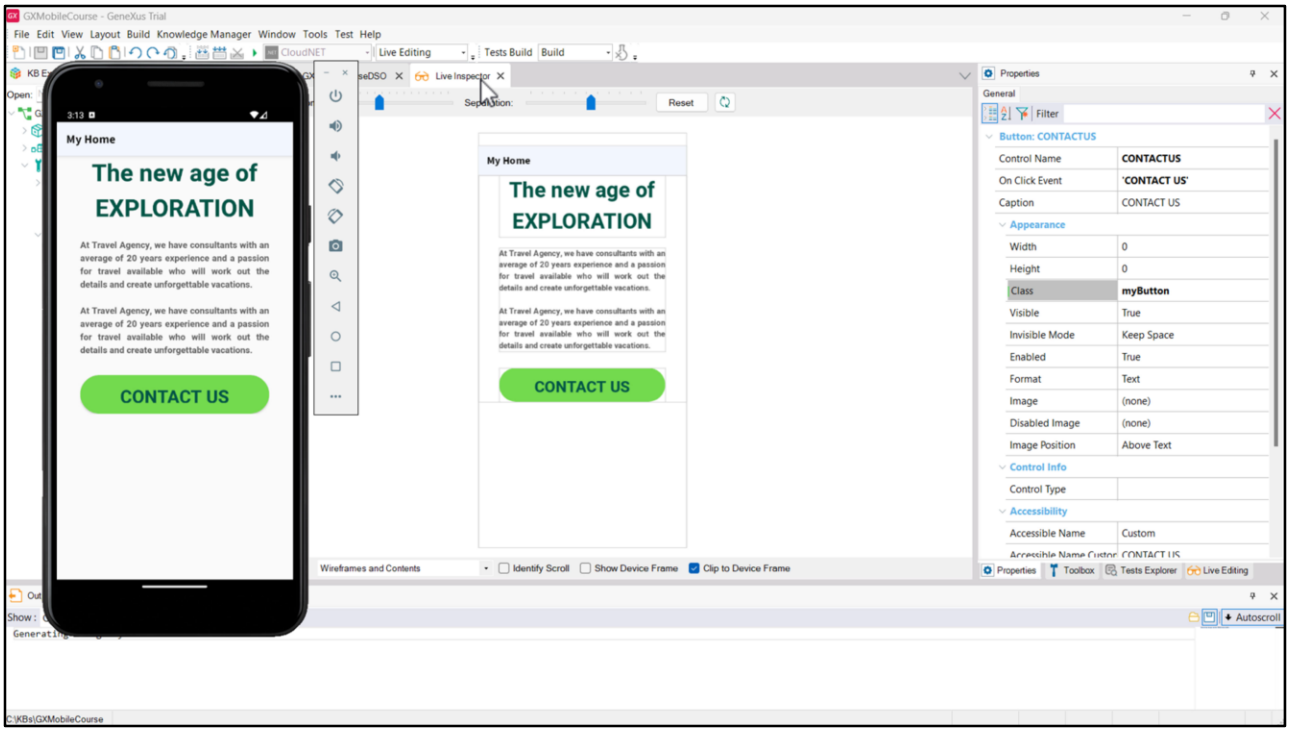
Now, we have to style the button. For that, let's create a new color token called colorButton to associate this shade of green…



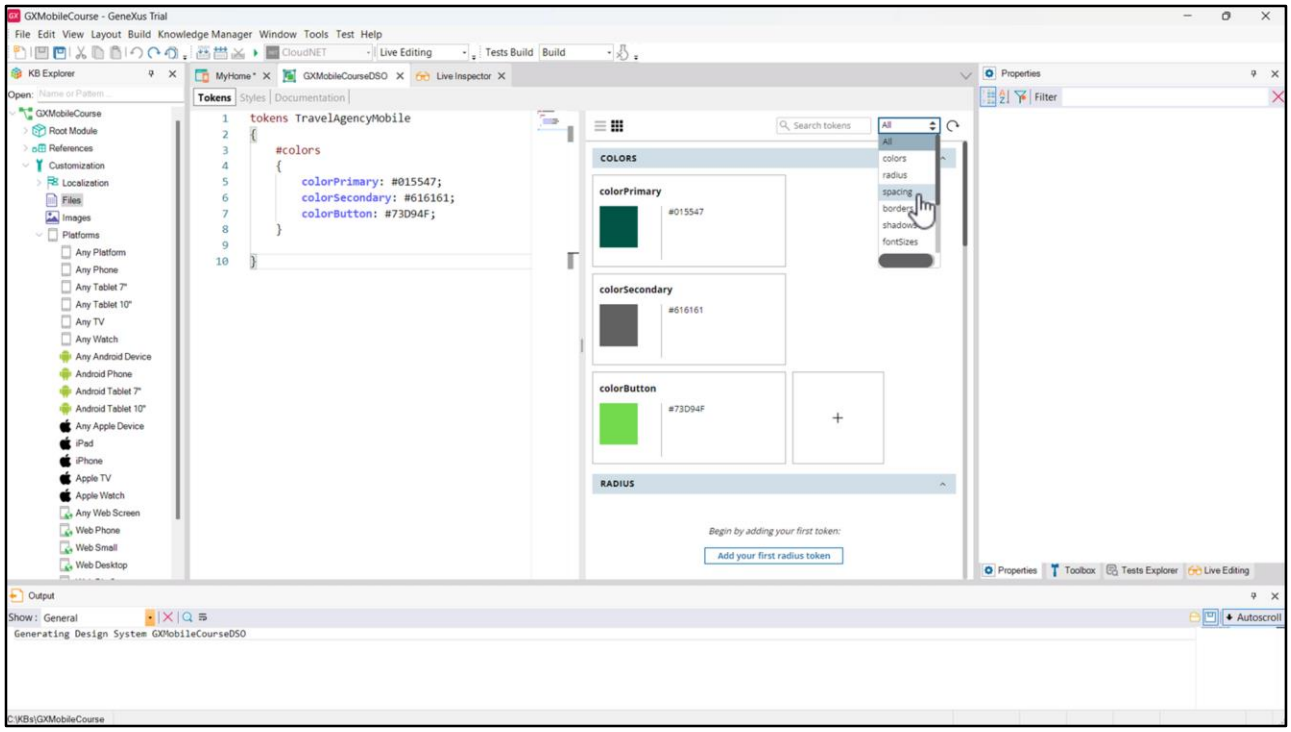
…and let's start defining the properties of the *myButton* class.

The border-radius property is used to round the edges; the background-color property allows us to set the background color; the padding property creates a space between the button's text and its edges.

Now let's complete it with the text style, which is the same as we did before...

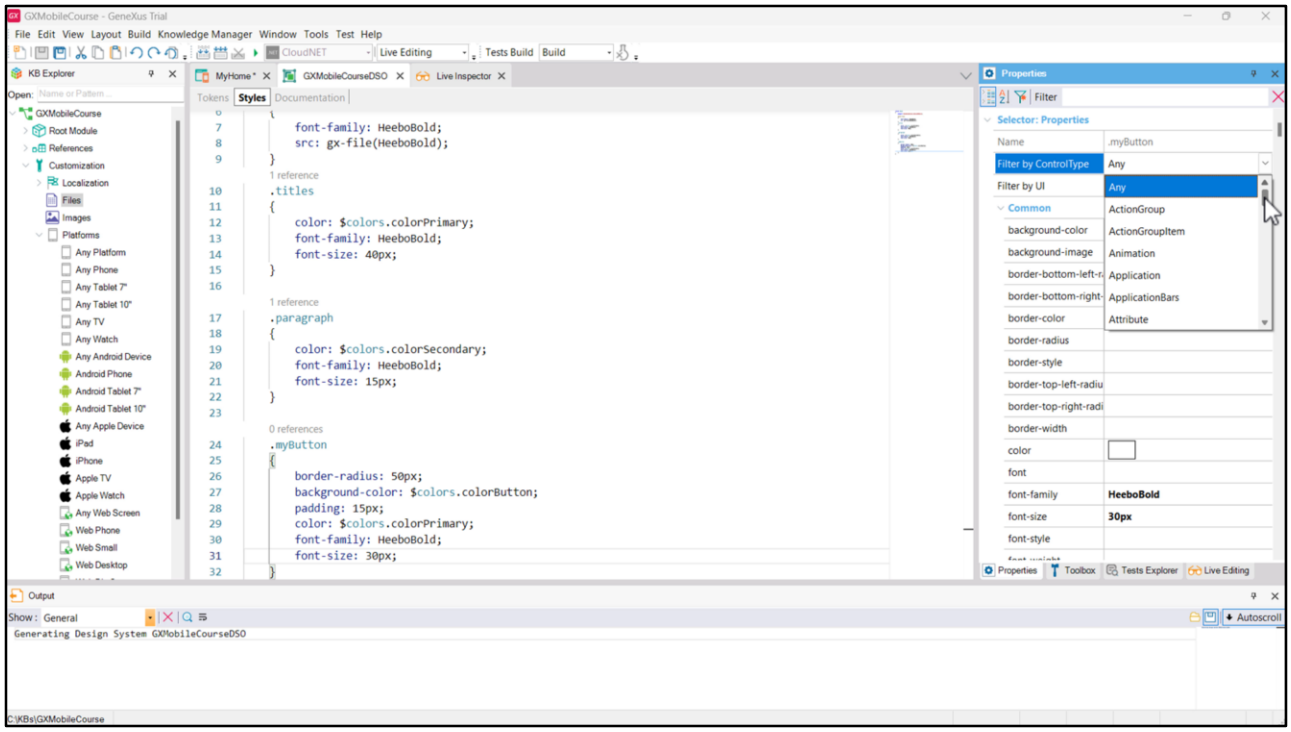


...and associate our class with the Panel button.

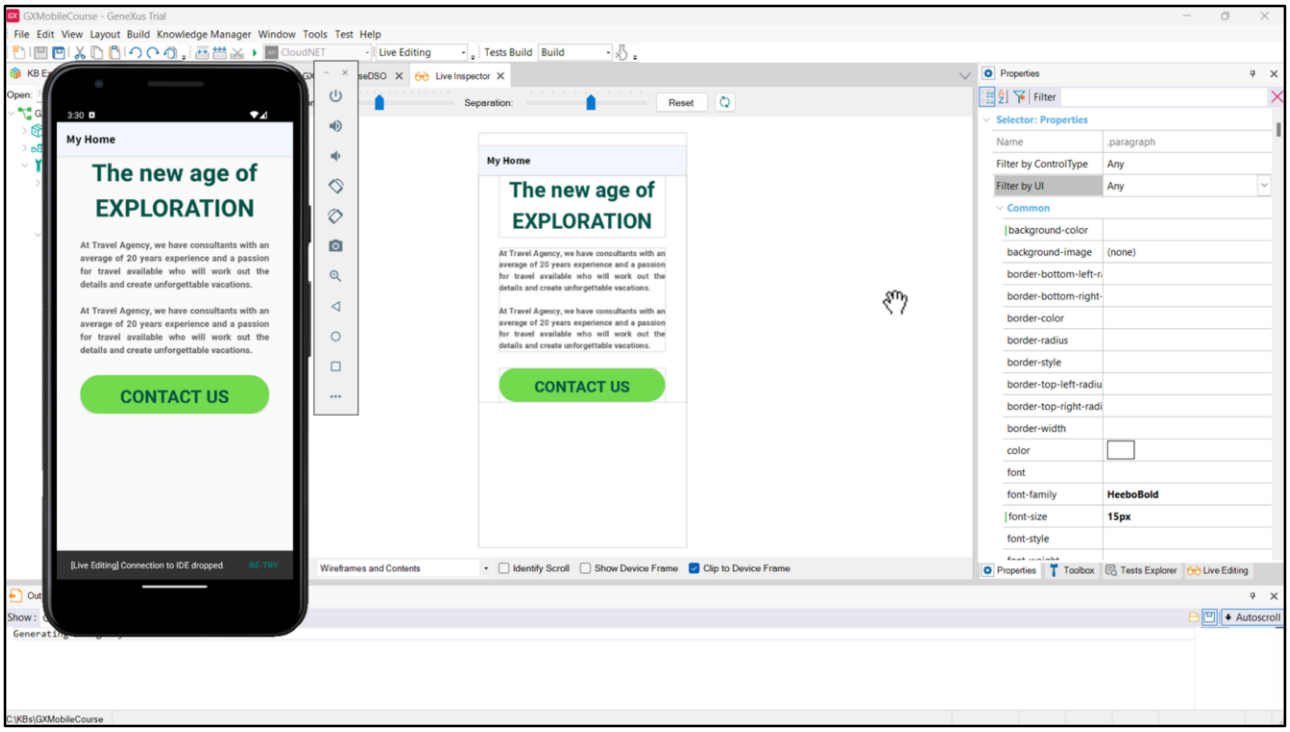


We can use not only color tokens, which are the most noticeable ones, but we can also define them for other kinds of things, for example: to give spacing, to define font sizes, etc.

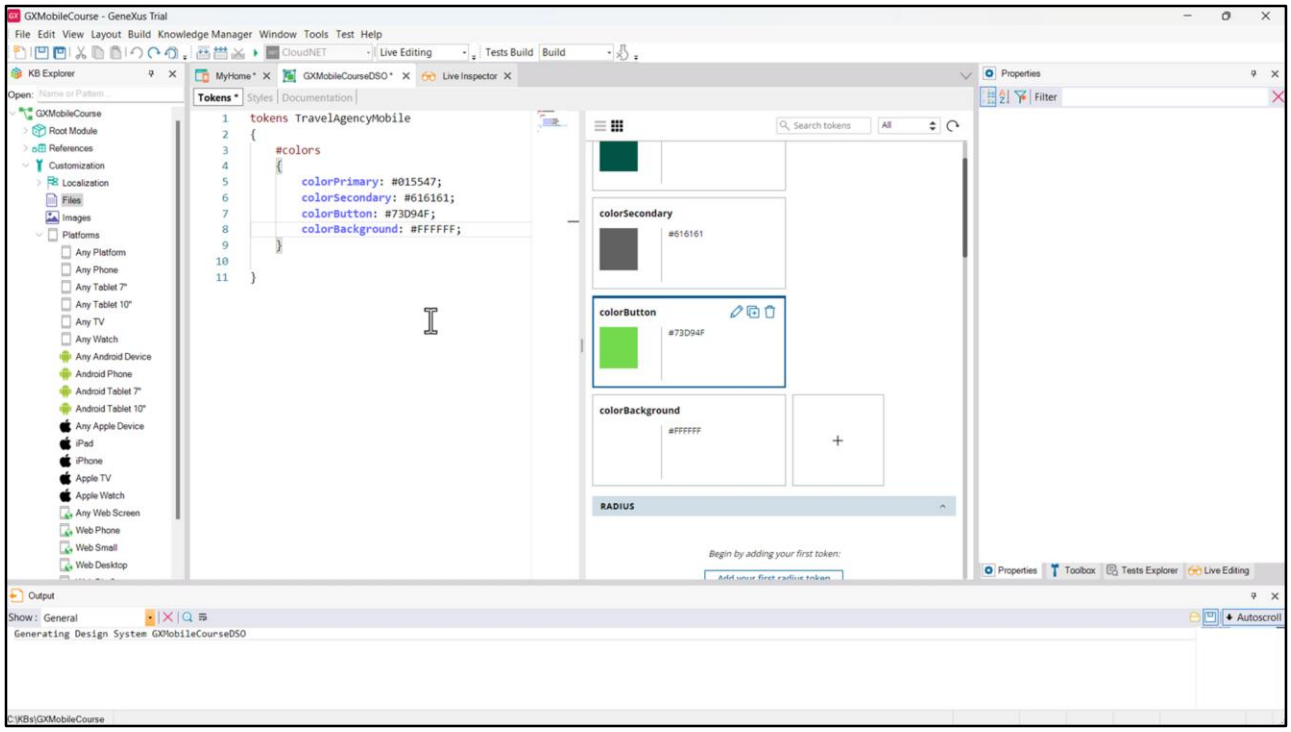
There are properties that are common to several controls (such as border-radius, background-color, color, font-family, font-size, among others).



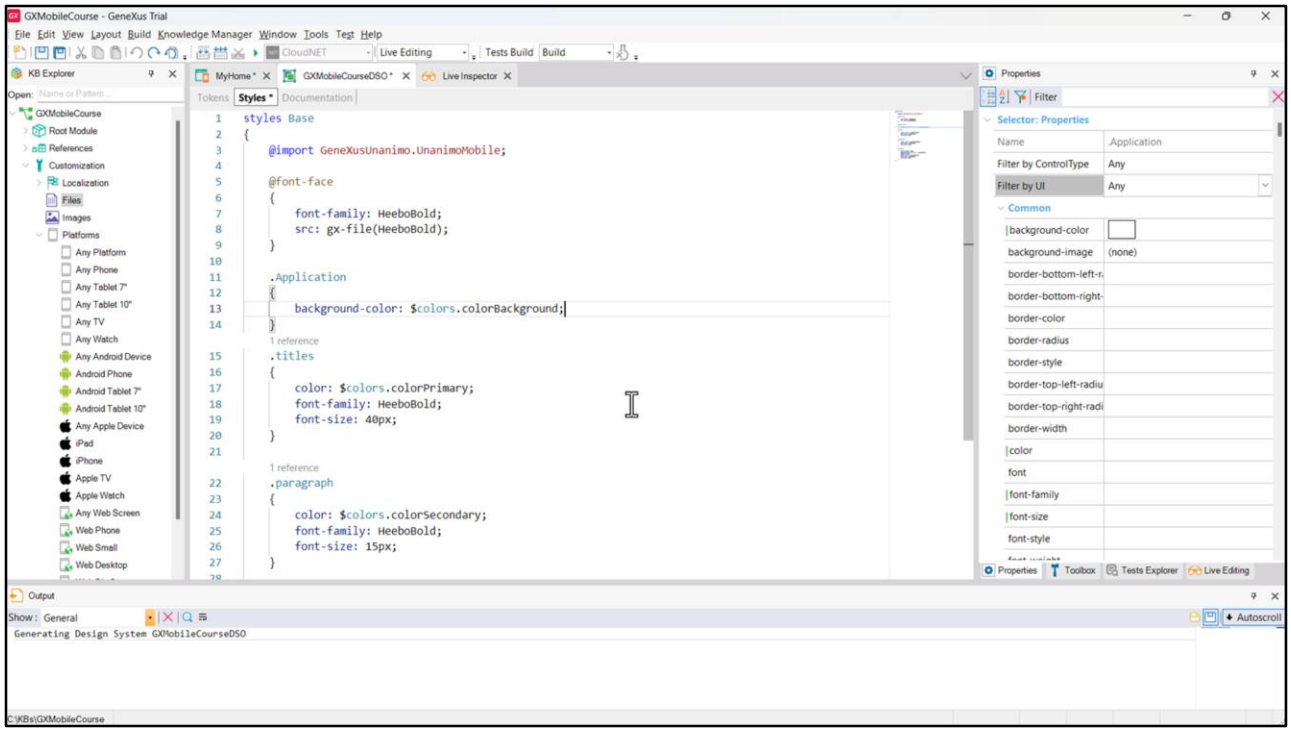
When we are working in the Styles tab, the properties window gives the option to filter by control type, where we will see and configure only those properties that apply to the selected control, and filter by UI, where we can choose the type.



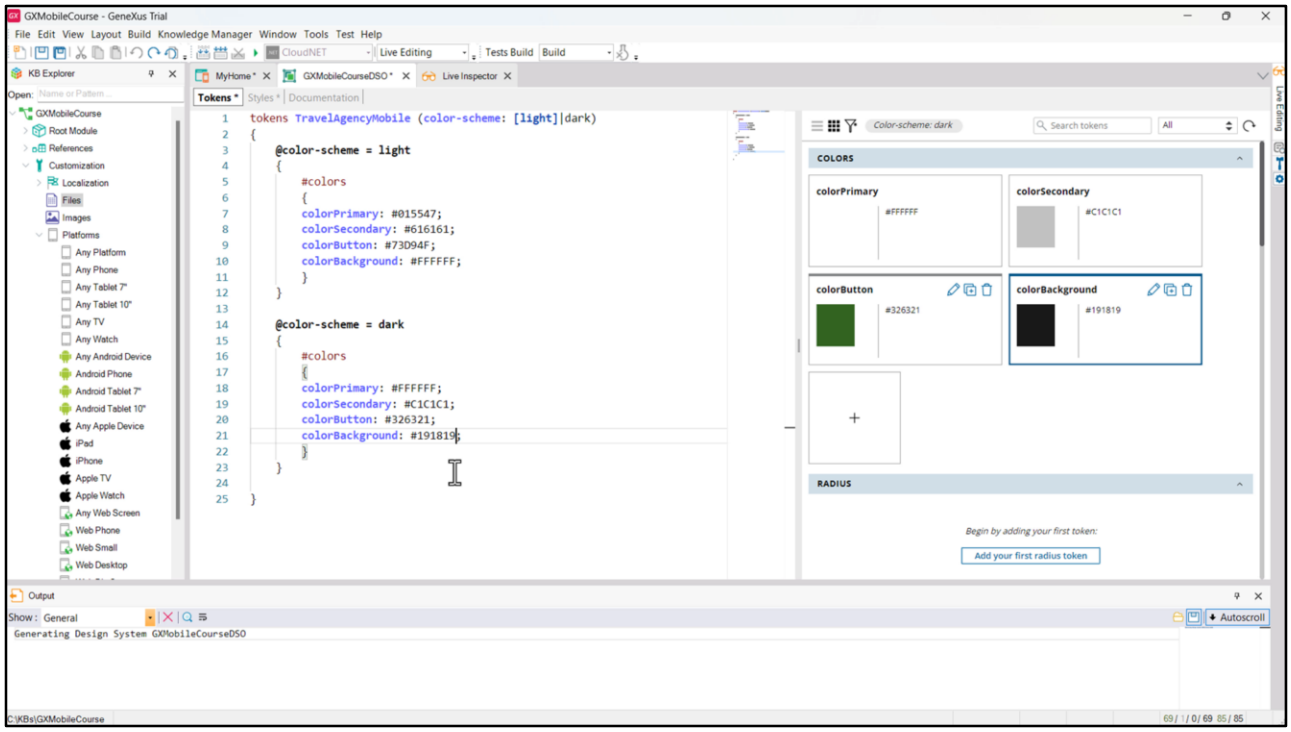
So far, we have designed the *light* mode of our application, but we can also design the *dark* mode. For that, we will have to think about varying the background color, the color of the texts and also the color of the button. In the case of the background, in *light* mode it will take the white color that we already have, and in *dark* mode it will take a very dark gray. For the title, in *light* mode, it will be the dark green that we defined as the primary color, and in *dark* mode it will be white, to contrast with the dark background. Similarly, we will define the colors for the description text and for the button in *dark* mode.



Let's specify the background color of the application: let's create the *Application* class and define the color, which we will retrieve from a token named colorBackground.

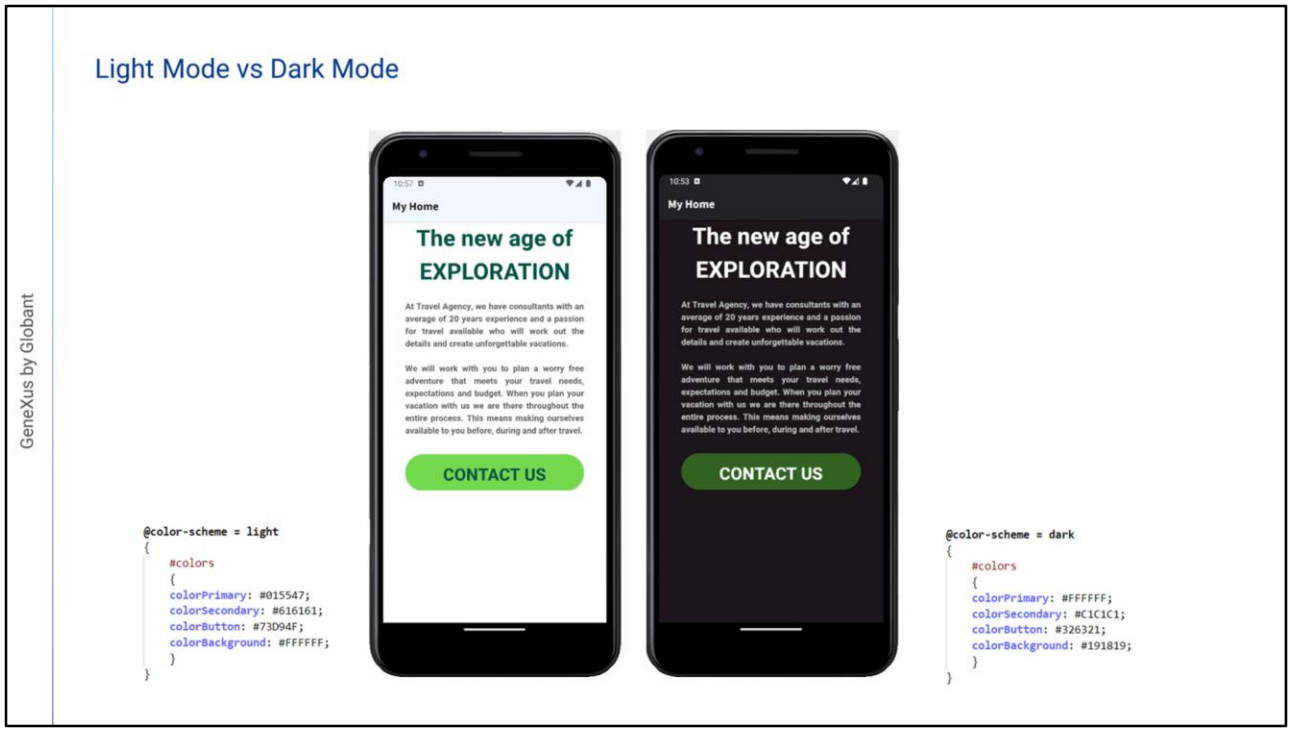


By naming the class like this, it will be the one applied in the body tag of all HTML automatically.



To specify the modes, we must use the options: color-scheme can be *light* or *dark*. By typing *light* in square brackets, we are indicating that we want this to be the default mode. Next, we vary the color tokens according to the color-scheme. For the *light* mode we will use these colors, and for the *dark* mode we copy those of the *light* mode and simply change the colors to be applied: the title will be white, the description will be a light gray, the button will be a different green than the one we used in the light mode, and the background will be a very dark gray.

Before running to see the changes, set the 'Enable Preferred Color Scheme' property of the MyHome Panel to True and do Build All.



In this way, our application will be adapted to the needs of all users.

We have seen an introduction of everything we can do with the Design System Object. In a future video, we will learn how to import into our KB a Figma design created by the designers.


GeneXus by Globant
GeneXus™
by Globant
training.genexus.com