

# TAdvResponsiveManager

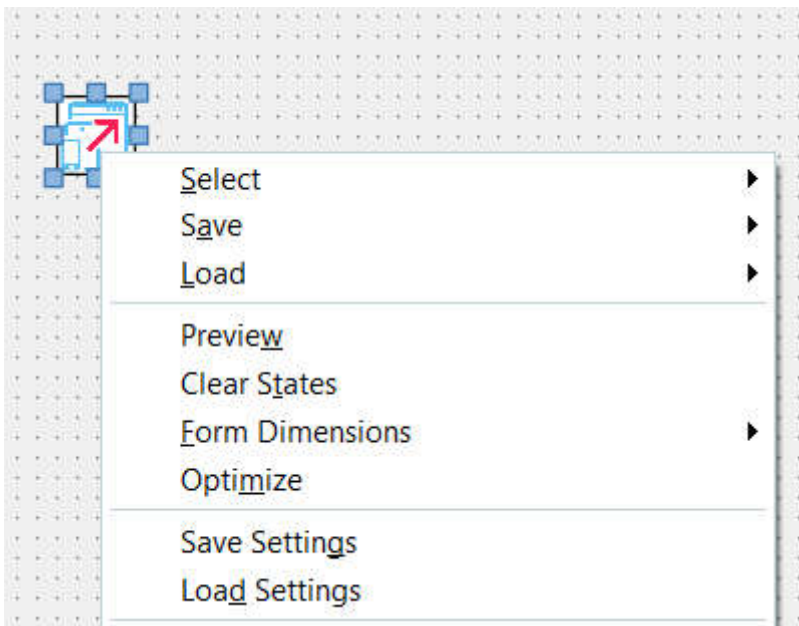
---

Included in TMS VCL UI Pack is the component TAdvResponsiveManager. This component is capable of designing forms in a responsive way. For those not familiar with the term "responsive", this means that the layout of the GUI can adapt to the form factor of the screen where the GUI is used. The component integrates with the form designer in such a way that you only need a single form for multiple states as they are managed by the component. At runtime, resizing events will be captured and handled automatically, whilst detecting and loading the appropriate state. After designing the various responsive states in your application, based on the width and height of the chosen form or control, simply run (or preview) the application to see the result.

## Getting Started

---

To design a form and add responsive behavior, drop an instance of TAdvResponsiveManager (further referred to as "responsive manager") on the form. This is a non-visual component. Right-clicking on the component will provide a set of options to choose from as seen in the screenshot below. The various options will be explained in different topics.



### Select

The select option will show which components or forms are available for responsive design. By default, the form on which the responsive manager lives is preset. Other controls which have been added to the form will popup in the select menu. Note that switching between different controls will prompt to clear existing states. States are tied to the control that is selected in the responsive manager.

Programmatically, selecting a control can be done with the following code.

```
AdvResponsiveManager1.Control := Panel1;
```

## Save

The save option will show an option to create a new state based on the current design of the form, and additionally show the already created states. To create a new state, click on "Save To New State".



After creating a new state, changing the design will automatically be saved when switching between states (see the AutoSave property), or by manually clicking on "Save to [State Name]" as shown in the context menu.

Programmatically, saving to a new state can be done with

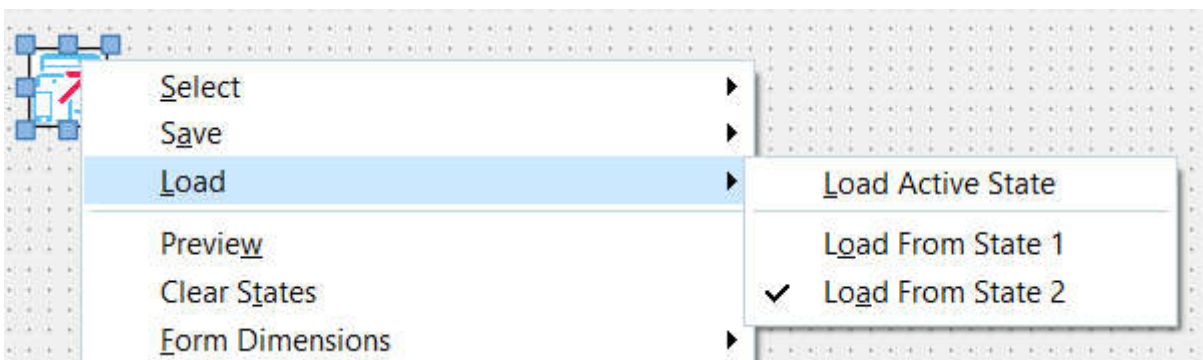
```
AdvResponsiveManager1.SaveToNewState;
```

Saving to an existing state can be done with

```
AdvResponsiveManager1.SaveToState([State Name]);
```

## Load

The load option shows the available states. Clicking on "Load Active State" will automatically detect the state based on the width and height of the control/form that is selected in the responsive manager.



The load an existing state, click on "Load From [State Name]". Note that when the "AutoSave" property is true, it will automatically save a state when switching.

Programmatically, loading a state based on the width and height of the active control/form can be done with

```
AdvResponsiveManager1.LoadState;
```

To load an specific state based on the name, use the following code

```
AdvResponsiveManager1.LoadStateByName([State Name]);
```

## Preview

The preview option shows the selected form at designtime as if you would run the application. Ofcourse, this only creates duplicate components, but doesn't add events or code logic that's behind the form. When showing the preview, resizing the form will switch between states defined in designtime. Additionally, a helper state banner is shown to indicate when the state will be active. As you can see from the screenshot below. State 1 will be active from 0 to a certain width, then the second state comes after the first state and so on.



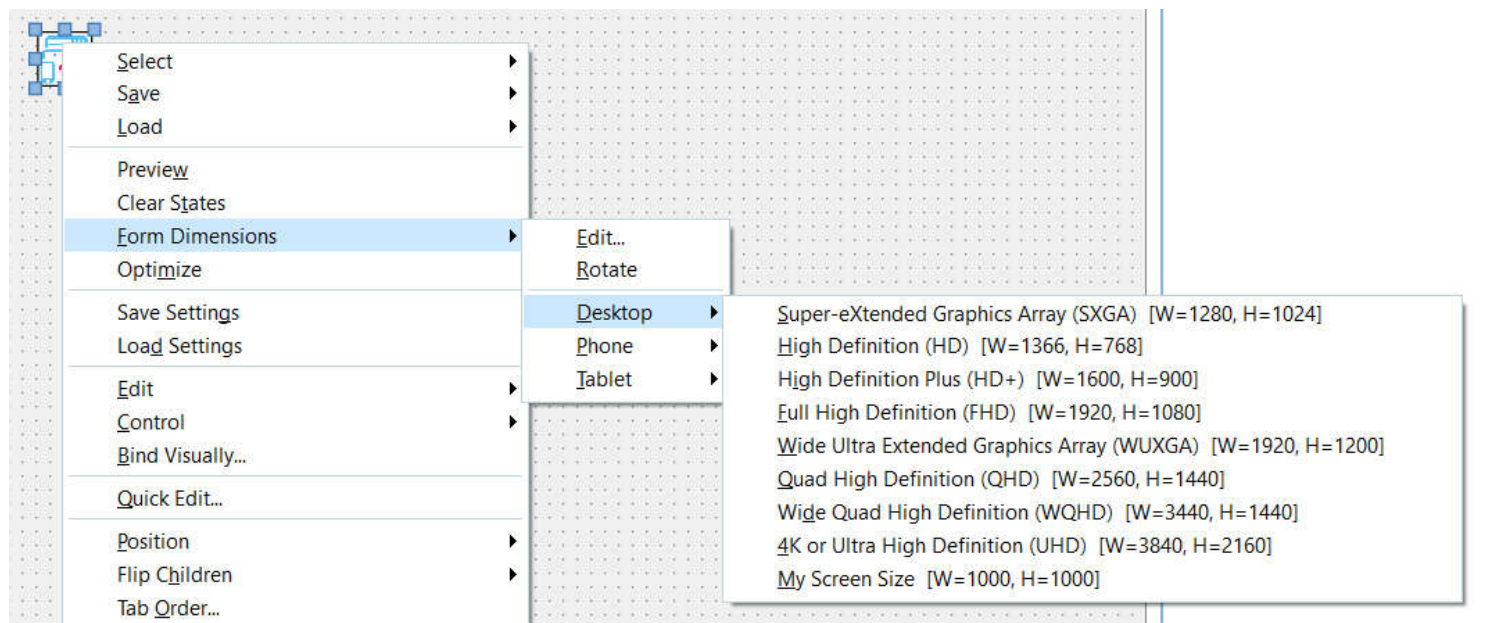
## Clear States

The "Clear States" option will clear all existing states.

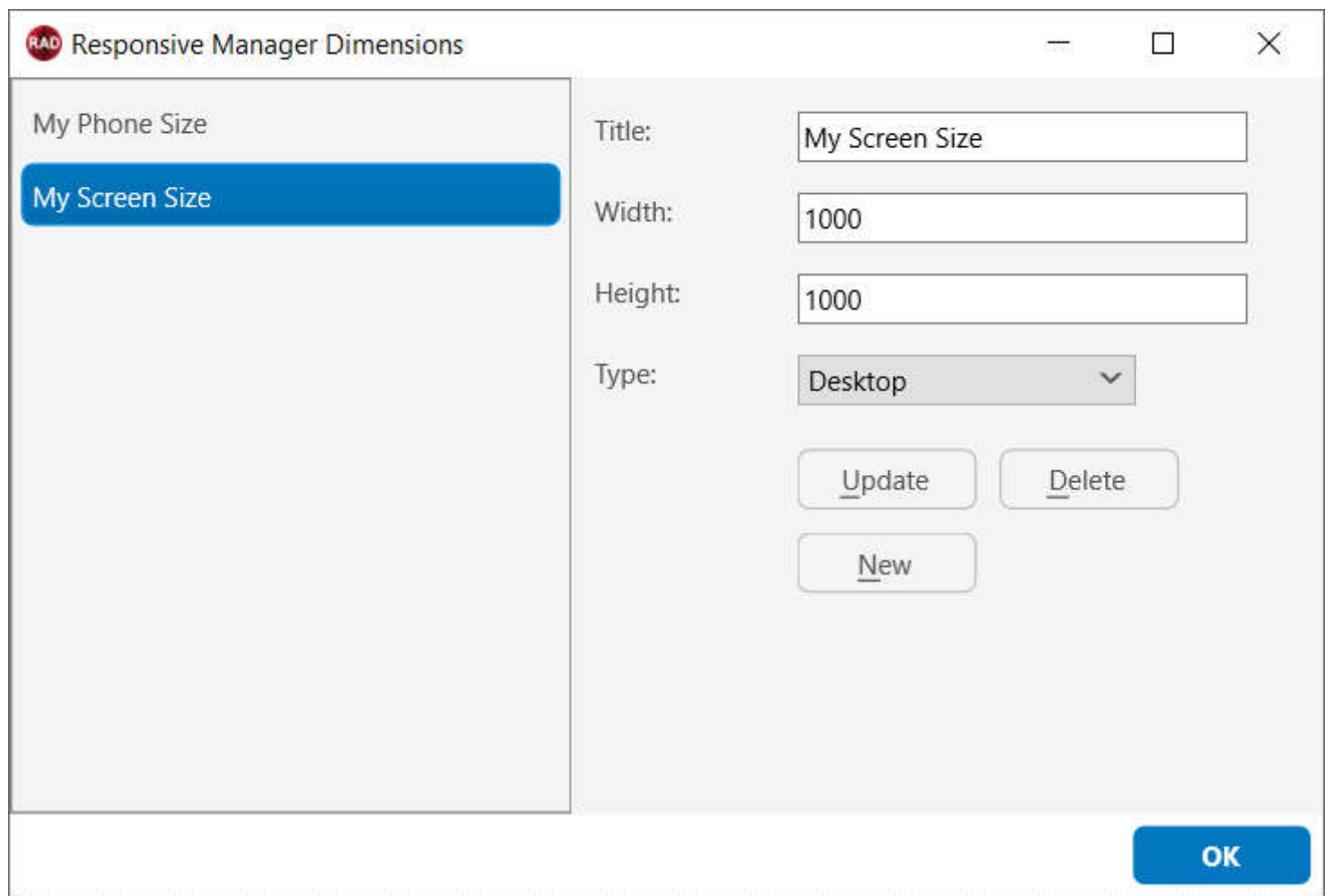
## Form Dimensions

When designing your form, you might want to design it based on a specific size. If it's a mobile application, you want to design it for a phone/tablet size. If it's a desktop application, you want to design

it for specific screen sizes such as HD or 4K. Right-clicking on the responsive manager and selecting "Form Dimensions", will popup options to select from predefined form sizes.



If the predefined form sizes are not sufficient, you can define your own sizes. click on the "Edit..." sub menu item to start the custom dimensions editor.



After defining your own sizes, you can find them in the appropriate size type in the "Form Dimensions" list.

## Optimize

After designing your form, you will notice that the form file will include all related components and settings for each specific state. Optimizing the states will remove all unnecessary settings and keep only the difference between states. Optimizing at designtime is optional, and depends on the number of components and states and can be useful when the form load time is severely affected. Note that at runtime optimization happens automatically to have more performance switching between states. Executing an optimization process at designtime will be irreversibly affect the form. When changes in one or more states are required after optimization, you will need to re-save all existing states.

## Save/Load Settings

This option is available to persist all states to a file in JSON format. Saving the settings can be important before executing an optimize at designtime, or to have a backup of a specific configuration.

## High DPI

---

The responsive manager takes high DPI into account. It detects the DPI of the form at designtime and at runtime and scales the controls when switching states. Additionally, it also handles the form sizes set at designtime when resizing. Note that once your form is designed/created at a specific DPI, it's not possible to open it in another DPI. Although the controls will be properly scaled at designtime, the existing states will still be mapped on the original DPI of the form. If form some reason, the form is opened in another DPI environment, recreating the states will be required.

## Constraints

---

By default, states have a constraint based on width & height. When saving a state, the width and height of the selected control/form is persisted. The mode property at responsive manager level determines how the state will be detected.

- **mrmWidthOnly:** When a state is loaded, checks the width of the selected control/form and finds the closest matching state with the Constraint.Width property.
- **mrmHeightOnly:** When a state is loaded, checks the height of the selected control/form and finds the closest matching state with the Constraint.Height property.
- **mrmWidthFirst:** When a state is loaded, checks the width of the selected control/form and finds the closest matching state with the Constraint.Width property. If the algorithm is finding more than one state, checks the height afterwards.

- **mrmHeightFirst:** When a state is loaded, checks the height of the selected control/form and finds the closest matching state with the Constraint.Height property. If the algorithm is finding more than one state, checks the width afterwards.

When loading a state, the constraint is checked based on the above settings. When the property `AutoLoadOnResize` is true, the responsive manager will automatically call `LoadState`, which will detect which state is matching the constraints and will then load the contents and apply the changes to each control found in the state. When a control is not found or no longer available, the loading of that specific control will be skipped. You can manually call `LoadState` as well from any other event by setting the `AutoLoadOnResize` to false.

## Custom Constraints

If you want to move away from width & height constraints, and you want to have state loading bound to a constraint that you control, it's possible to use one of the following constraints instead:

- **StringValue:** Setting the `StringValue` property will allow you to call `LoadState(AStringValue: string);` This is typically done when string matching is required. Multiple states are possible.
- **BooleanValue:** Setting the `BooleanValue` property will allow you to call `LoadState(ABooleanValue: boolean);` Only 2 states are possible.
- **NumberValue:** Setting the `NumberValue` property will allow you to call `LoadState(ANumberValue: string);`

Additionally, if `StringValue`, `BooleanValue` or `NumberValue` is not sufficient, it's possible to call procedure `LoadStateCustom(ACallback: TAdvStateManagerLoadStateCustomCallback = nil);`, which has a callback parameter. If the callback parameter is nil, the `OnLoadStateCustom` will be called. What this method will do is, loop through every state, and will ask to load it. When setting the `ALoad` var parameter to true in either the callback or the event, the state will be loaded. Each state is also capable of holding custom data in one of the following properties:

- **DataPointer**
- **DataBoolean**
- **DataObject**
- **DataString**
- **DataInteger**

## Properties

---

**ActiveState:** The current active state. This property can be set at designtime to switch between states. When the `AutoSave` property is true, this action will automatically save the state at designtime. When set at runtime, it will load the state based on the index, but will not save or modify states.



**AutoLoadOnResize:** When true, automatically detects the OnResize event of the selected control/form. The responsive manager uses this event to automatically load the state when the form resizes.

**AutoSave:** When true, automatically saves the state when switching between states. Switching states can be done via the ActiveState property or when loading one of the states via the context menu. saving states only happens at design time.

**Mode:** This property is used to determine which constraint will be used when loading the state. By default the mrmWidthOnly mode will only check the width when resizing and loading states, whereas the msmHeightOnly mode will only check the height. With the mrmWidthFirst & mrmHeightFirst modes, the first check is the width or height, and when there are multiple states detected, then it will look at the height or width respectively.

**States:** The collection of states managed by the responsive manager.

**States->Name:** The name of the state. Can be used to identify, save & load a state.

**States->Default:** The default state. Only one state can be default, it automatically loads this state when there is no other state detected during the automatic load sequence.

**States->Constraint:** The constraints of the state. By default it uses the width and height constraint. It's possible to also programmatically load a state based on a different constraint such as the BooleanValue, NumberValue or StringValue. This is explained in the chapter "Constraints".

**States->Content:** The JSON representation of a state. This property is not visible at design time. The Content property will be persisted in the form file and be reloaded when states are changed.

## Events

---

**OnBeforeLoadControlState:** Event called before loading the state of the control. This event can be used to potentially block loading the state content for a specific control, or to prepare the control before loading the state.

**OnBeforeLoadState:** Event called before loading the state.

**OnAfterLoadControlState:** Event called after loading the state of the control. This event can be used to apply changes after loading the state content for a specific control.

**OnAfterLoadState:** Event called after loading the state.

**OnLoadStateCustom:** Event called when loading a state with the LoadStateCustom. In the callback or event it's possible to determine if a state can be loaded or not.

# Methods

---

**procedure SaveToState(AState: TAdvStateManagerItem);** Saves the current content of the selected control/form to an existing state, collection item based.

**procedure SaveToState(AIndex: Integer);** Saves the current content of the selected control/form to an existing state, index based.

**procedure SaveToState(AName: string);** Saves the current content of the selected control/form to an existing state, name based.

**procedure LoadStateByName(AName: string);** Loads the state based on the name.

**procedure LoadStateByIndex(AIndex: Integer);** Loads the state based on the index.

**procedure LoadStateCustom(ACallBack: TAdvStateManagerLoadStateCustomCallback = nil);** Loads the state based on a custom callback or event.

**function FindConflicts(AConflictedControlNames: TStrings): Boolean;** Finds conflicts in all states. For example: if Button1 is found in state 1, but not in state 2, the function will return True, and the AConflictedControlNames will contain a list of control names and their states.

**function GetDefaultState: TAdvStateManagerItem;** Returns the default state.

**procedure LoadState(AStringValue: string);** Loads the state based on a string value. Note that the state collection item needs to have the Constraint.StringValue property set and it needs to match the value passed as a parameter. The default constraint loading is Width/Height based.

**procedure LoadState(ABooleanValue: Boolean);** Loads the state based on a boolean value. Note that the state collection item needs to have the Constraint.BooleanValue property set and it needs to match the value passed as a parameter. The default constraint loading is Width/Height based.

**procedure LoadState(ANumberValue: Extended);** Loads the state based on a number value. Note that the state collection item needs to have the Constraint.NumberValue property set and it needs to match the value passed as a parameter. The default constraint loading is Width/Height based.

**procedure LoadState;** Loads the state based on the width/height constraint matching the selected control/form. Note that the state collection item needs to have the Constraint.Width & Constraint.Height properties set.

**function SaveToNewState: TAdvResponsiveManagerItem;** Saves the content of the selected control/form to a new state, with the constraint set to width & height.



**function SaveToNewState(AStringValue: string): TAdvResponsiveManagerItem;** Saves the content of the selected control/form to a new state, with the constraint set to a string value.

**function SaveToNewState(ABooleanValue: Boolean): TAdvResponsiveManagerItem;** Saves the content of the selected control/form to a new state, with the constraint set to a boolean value.

**function SaveToNewState(ANumberValue: Extended): TAdvResponsiveManagerItem;** Saves the content of the selected control/form to a new state, with the constraint set to a number value.

**function FindStateByName(AName: string): TAdvStateManagerItem;** Returns the state with a specific name.

**function FindState(AStringValue: string): TAdvResponsiveManagerItem;** Returns the state with a specific string value constraint.

**function FindState(ABooleanValue: Boolean): TAdvResponsiveManagerItem;** Returns the state with a specific boolean value constraint.

**function FindState(ANumberValue: Extended): TAdvResponsiveManagerItem;** Returns the state with a specific number value constraint.

**function FindState: TAdvResponsiveManagerItem;** Returns the state based on the selected control/form width & height constraint matching the state constraint.

**procedure Preview;** Launches a preview of the form.

## Samples

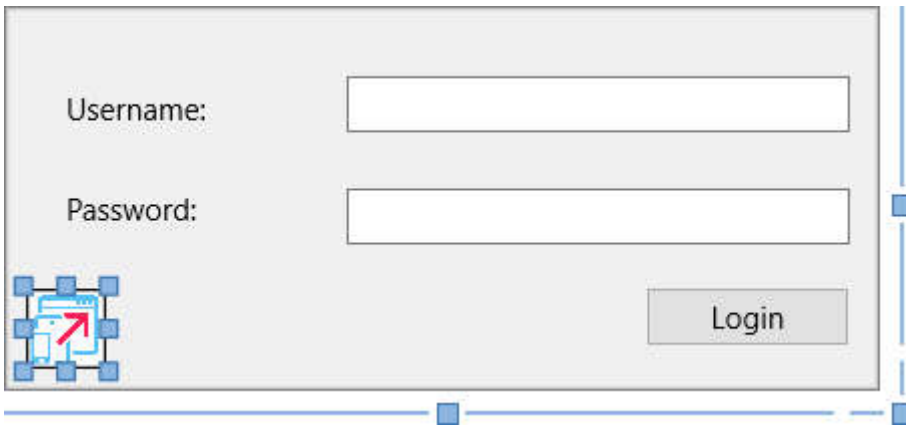
---

### Sample 1: Simple login form using width/height constraints

This sample will show a simple login form, a panel, 2 labels, 2 edits and a button. We are going to create 3 states: small, mid and large, which will be configured based on the width of the form.

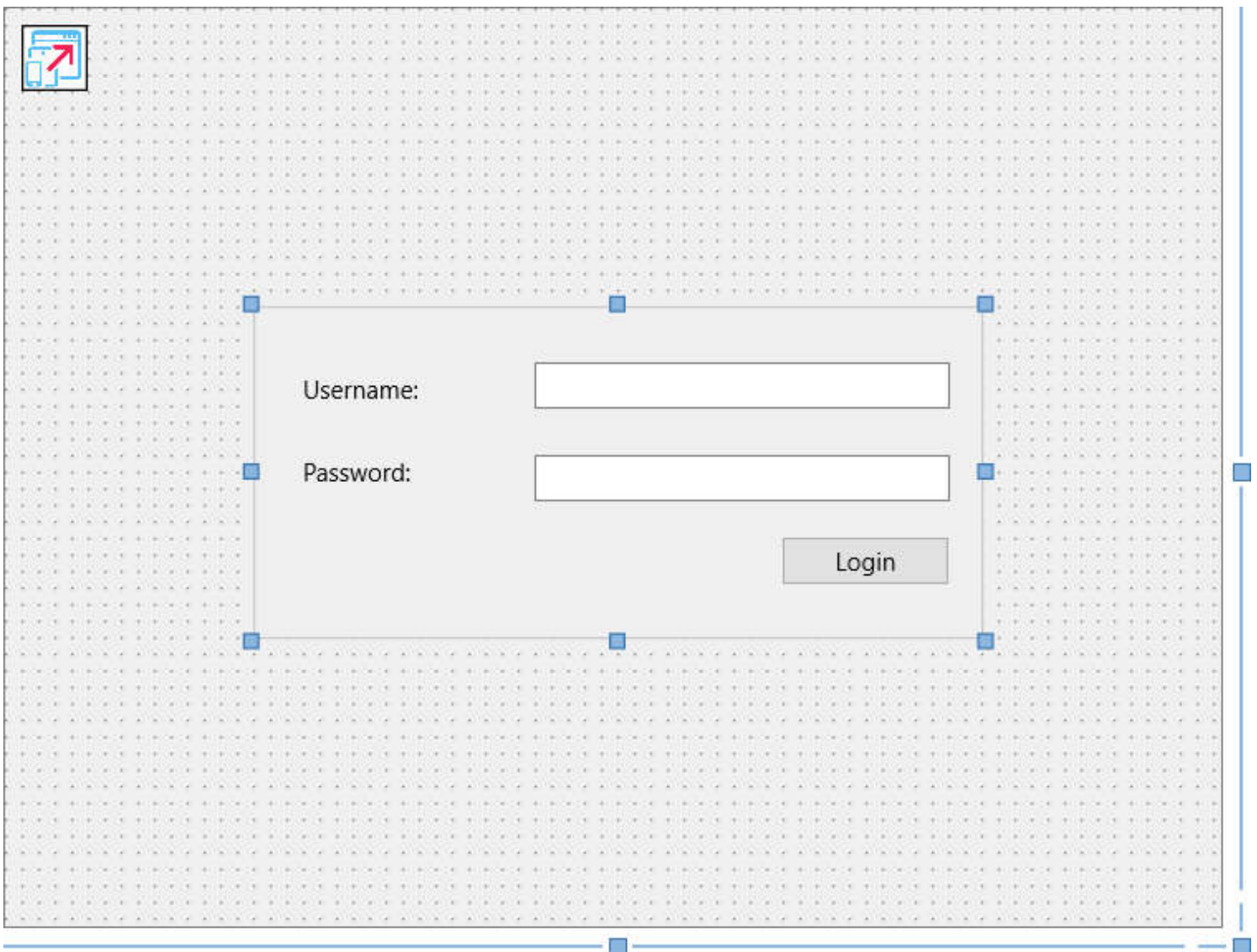
Let's start with state 1, the small state.

1. Drop the required controls and an instance of the TAdvResponsiveManager on the form. Arrange them like the screenshot below, a client-aligned panel, the controls inside the panel are arranged purely with the position, width and height. The form size has been set to a small size.



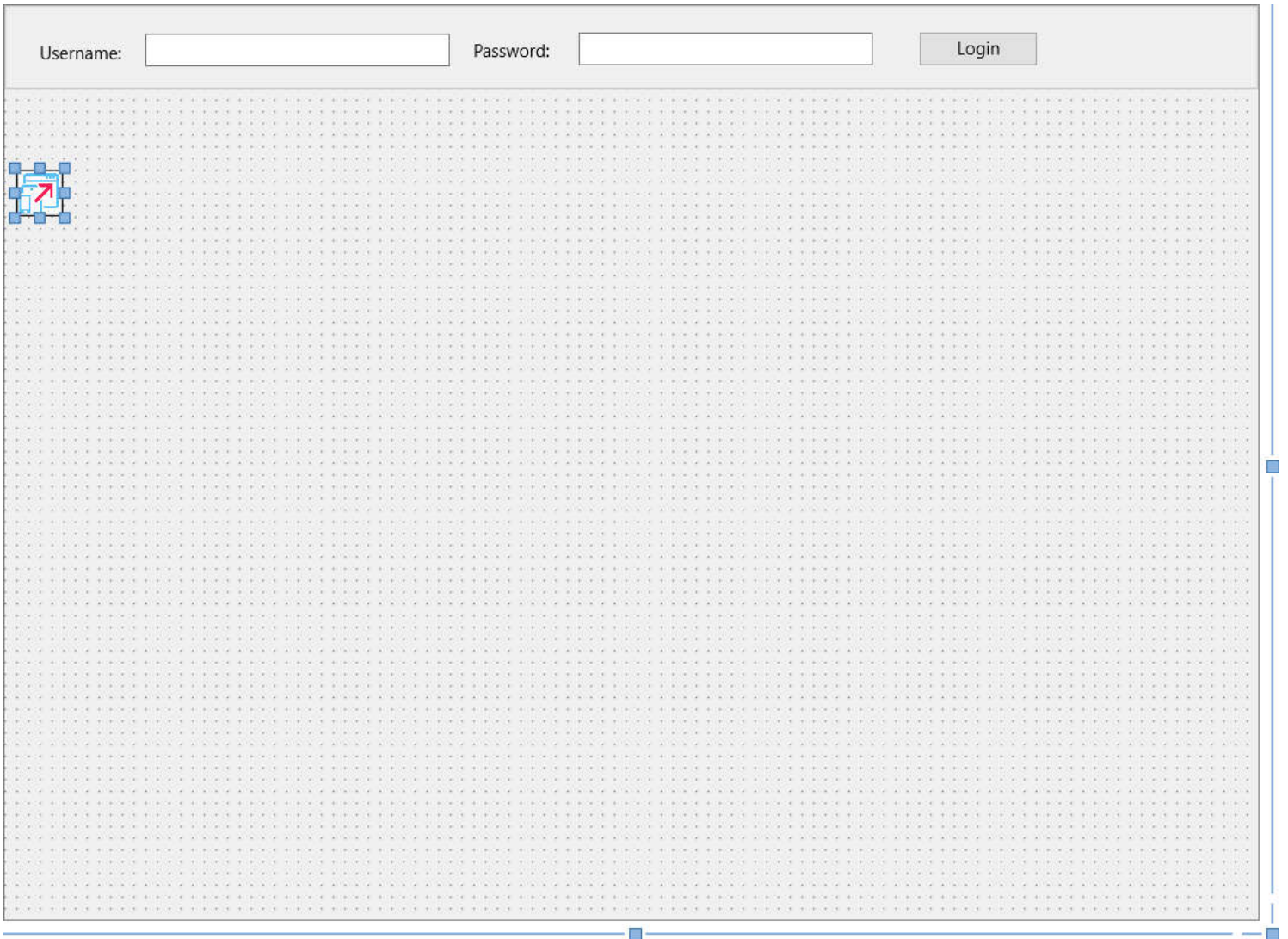
A small form layout on a light gray background. It contains two text input fields, one for "Username:" and one for "Password:". Below the password field is a "Login" button. A responsive manager icon (a blue square with a red arrow) is located in the bottom-left corner of the form. Blue dimension lines and handles are visible around the form, indicating it is being edited.

2. Save the form layout to a new state (right-click on the responsive manager and select Save->Save To New State). Rename the layout to "Small" in the structure pane, in the states collection.
3. Re-size the form to a medium size (double the small size) and re-arrange the controls. For this state, we set the panel to be center-aligned. Again, after arranging the controls, right-click on the responsive manager and select Save->Save To New State. Rename the layout to "Medium".



A medium-sized form layout on a light gray background, centered on a grid. It contains two text input fields, one for "Username:" and one for "Password:". Below the password field is a "Login" button. A responsive manager icon (a blue square with a red arrow) is located in the top-left corner of the form. Blue dimension lines and handles are visible around the form, indicating it is being edited.

4. For our "Large" state, we again double the form size. This time, we top-align the panel and rearrange the controls inside the panel to make room for more content when we want to build out our application in the future. Save this state to a new state and rename it to "Large".



In the structure pane, there should now be three states.

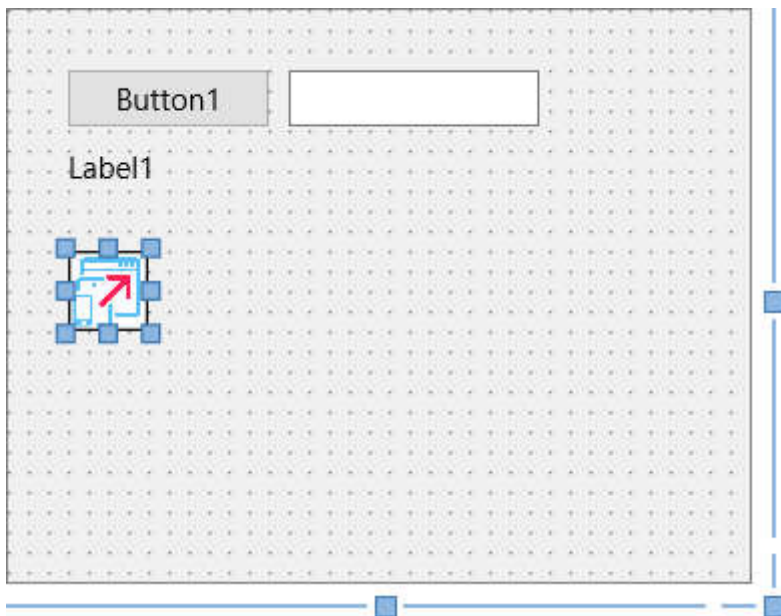


When running the application and resizing the form, you'll notice the three states are detected based on the size of the form and loaded when required. The "Large" state will remain active for all sizes larger than the width constraint.

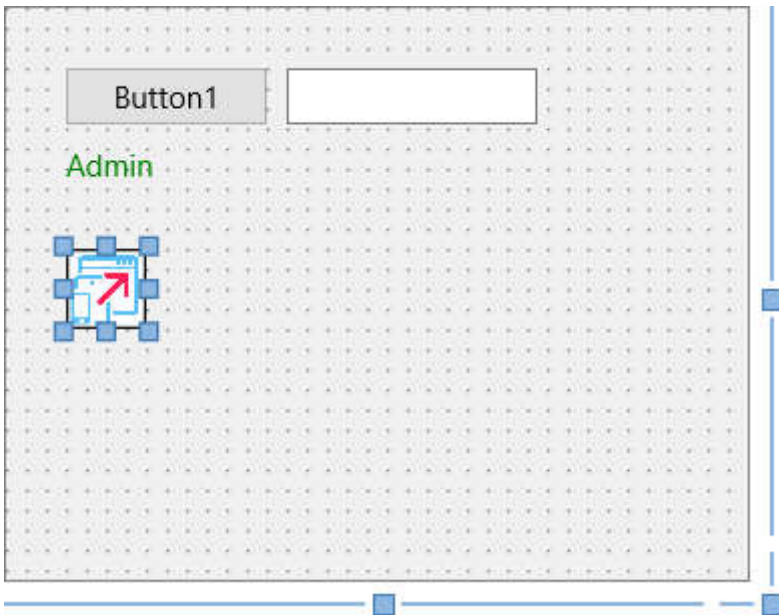
## Sample 2: Admin/User using custom constraints

Although the name is responsive manager as demonstrated in the first example, it's possible to custom load states, based on constraints other than width & height. For this sample, we need a label an edit and a button and we'll use the StringValue property as the constraint.

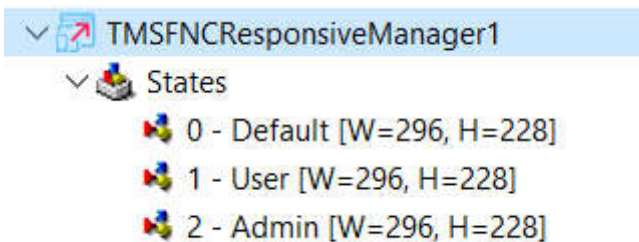
1. We drop our controls on the form, in this case, the layout doesn't really matter.
2. Add an instance of TAdvResponsiveManager and set the AutoLoadOnResize to false, in this case we actually want to programmatically load the state dependent on the constraint, instead of letting the form decide when to load the state.



3. We set the label visible to false, and create our first and default state. This state will be loaded when the form is created and will ensure that the label is hidden until the button is clicked. We set the StringValue of this state to "Default".
4. Now, we need 2 more states which are the "User" and "Admin" states. In each state the label will be set visible again, and the text will reflect what type of login has happened. For the first state, we set the label visible to true, and set the text to "User". We also set the font color to red. Save this layout to a new state and set the StringValue to "User". For the "Admin" state, we set the label font color to green and the text to "Admin". Again, we save this to a new state and set the StringValue to "Admin".



5. For readability we can also set the Name property of each state to the same value. As seen in the screenshot below, we now have our states, ready to be loaded. Note that each state also adds the width and height by default, but this is ignored when loading the state in the next step.



6. When the form is created, we want to load our default state. To do this, we call the following code:

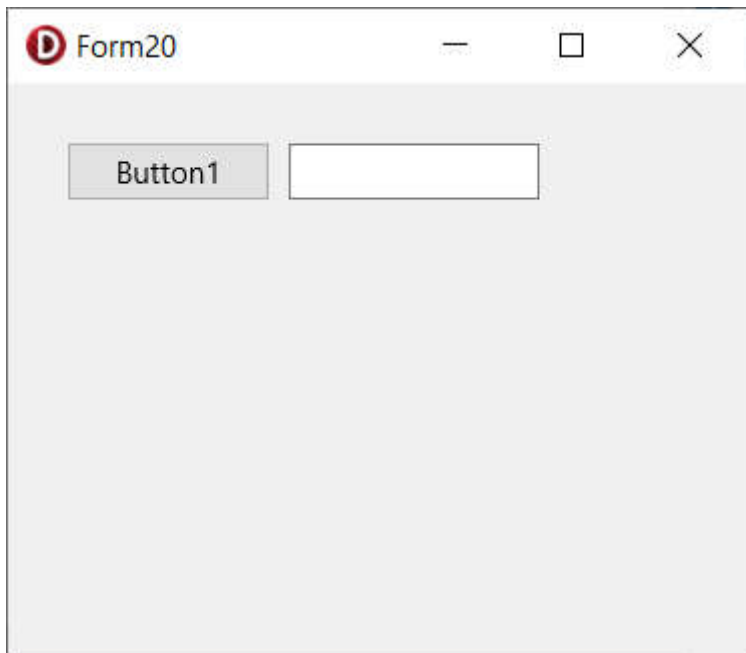
```
procedure TForm1.FormCreate(Sender: TObject);
begin
    AdvResponsiveManager1.LoadState('Default');
end;
```

7. To load our states based on "User" or "Admin", we implement the button click event and the following code:

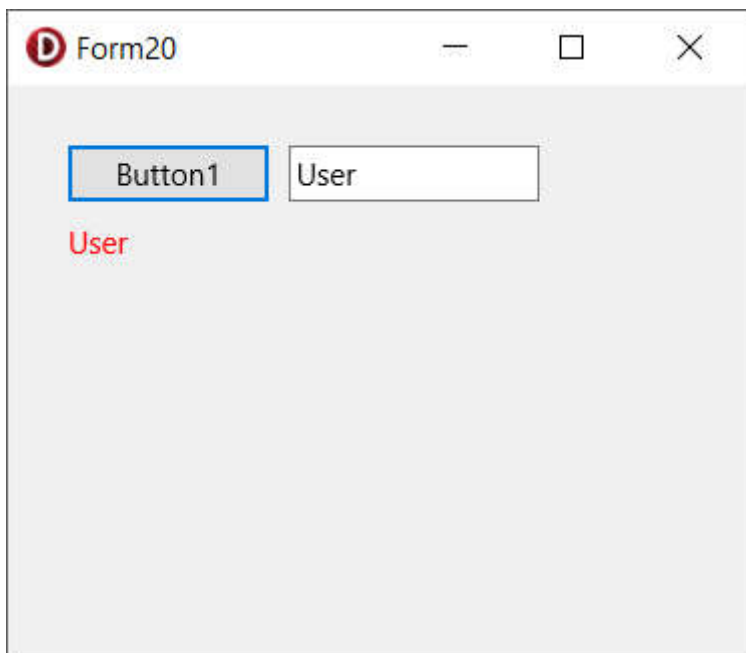
```
procedure TForm1.Button1Click(Sender: TObject);
begin
    AdvResponsiveManager1.LoadState(Edit1.Text);
end;
```

When running this application, and clicking on the button, we can load the different states based on the value of Edit1.Text, which then is checked against the states in the responsive manager. This way, you can

easily configure various control properties and bind them to a state without needing to programmatically write code to set various control properties. Note that when entering an incorrect value, the default state will be loaded.




A screenshot of a window titled "Form20" with a red circular icon containing a white 'D'. The window has standard Windows-style controls (minimize, maximize, close). Inside the window, there is a button labeled "Button1" and an empty text input field to its right.



A screenshot of the same "Form20" window. The button "Button1" is now highlighted with a blue border. The text input field contains the text "User". Below the button and text field, the word "User" is displayed in red text.



 Form20

Button1

Admin

Admin