(TcHmi) User Control ⇒ FW Control conversion

Introduction

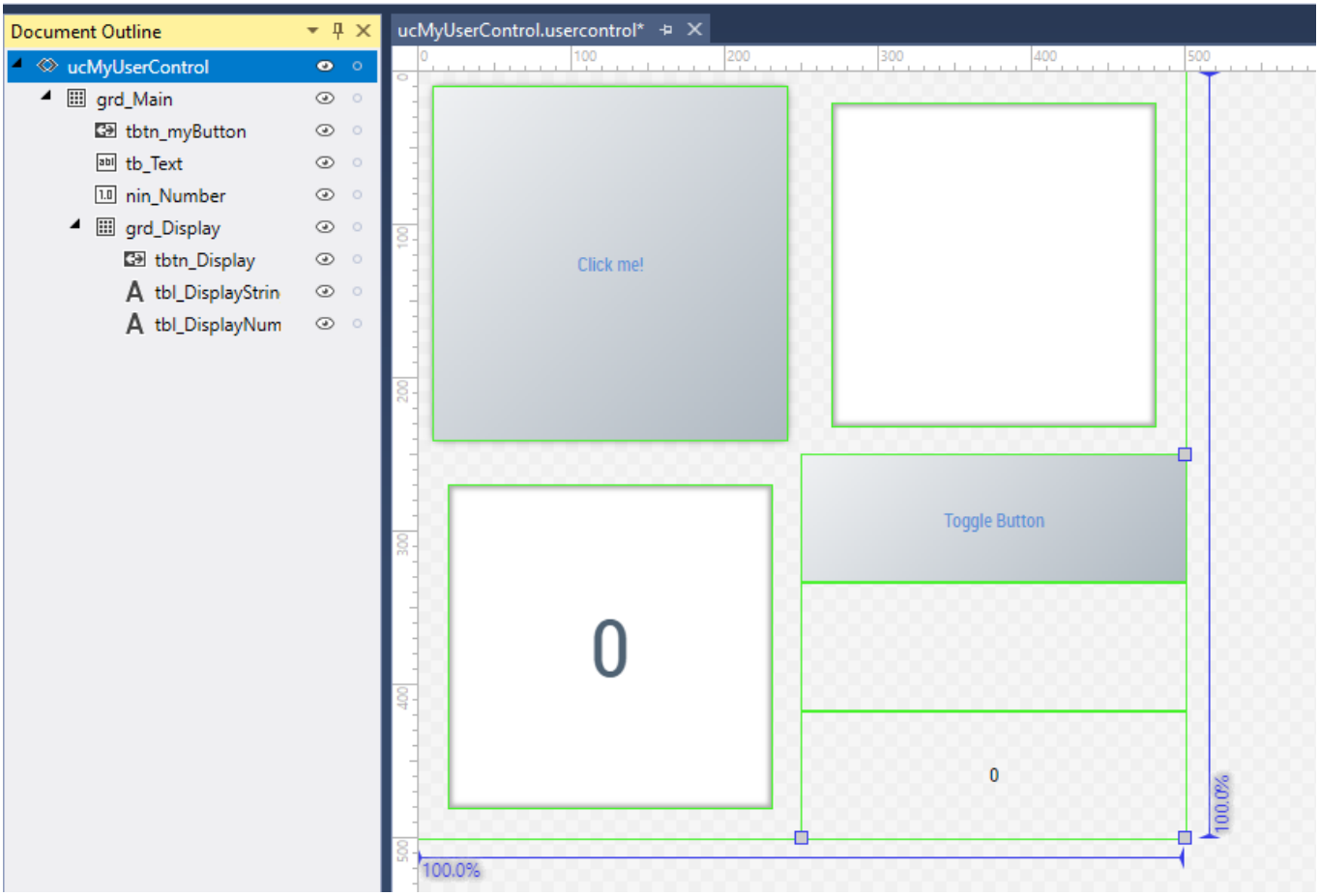
The advantage of Framework Controls within the TwinCAT HMI is quite extensive. Some additional features include:

- Complex coding freedom
- Precise lifetime management
- Library creation with help of Nuget packages. This also enables version control and dependency management.

However, the ease of making controls using the WYSIWYG editor is also a very useful design feature of the base HMI software. Unfortunately this visual editor is not available in a Framework Project. But we can use the visual editor to build the basic features of a control and then port the relevant parts to a Framework Control.

Creating a basic User Control

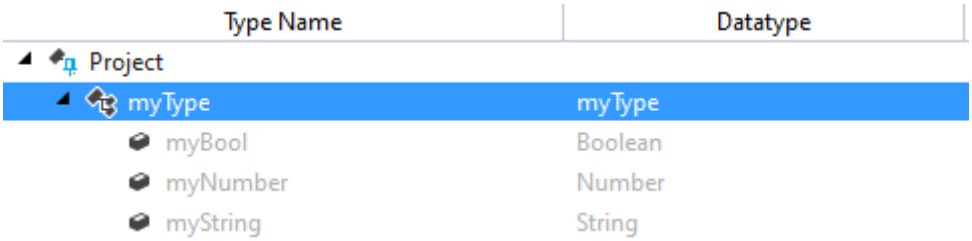
The user control described in this section will be used in a conversion process to a Framework control.
The user control has the following structure:



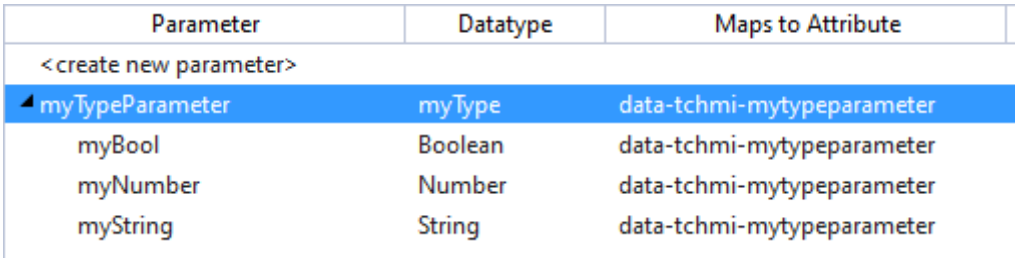
Where the main grid has shape 2x2 and the nested grid has shape 1x3.
A custom datatype was created to be used as an input parameter with the following schema definition:

```
{
  "properties": {
    "myBool": {
      "$ref": "tchmi:general#/definitions/Boolean"
    },
    "myNumber": {
      "$ref": "tchmi:general#/definitions/Number"
    },
    "myString": {
      "$ref": "tchmi:general#/definitions/String"
    }
  },
  "type": "object",
  "id": "tchmi:project#/definitions/myType"
}
```

Which looks like this in the HMI Configurator



This datatype is then used as a parameter for the user control



So we have a finished user control in a base HMI project with the following properties:

- Using standard Beckhoff toolbox controls
- An input parameter
- An associated custom datatype

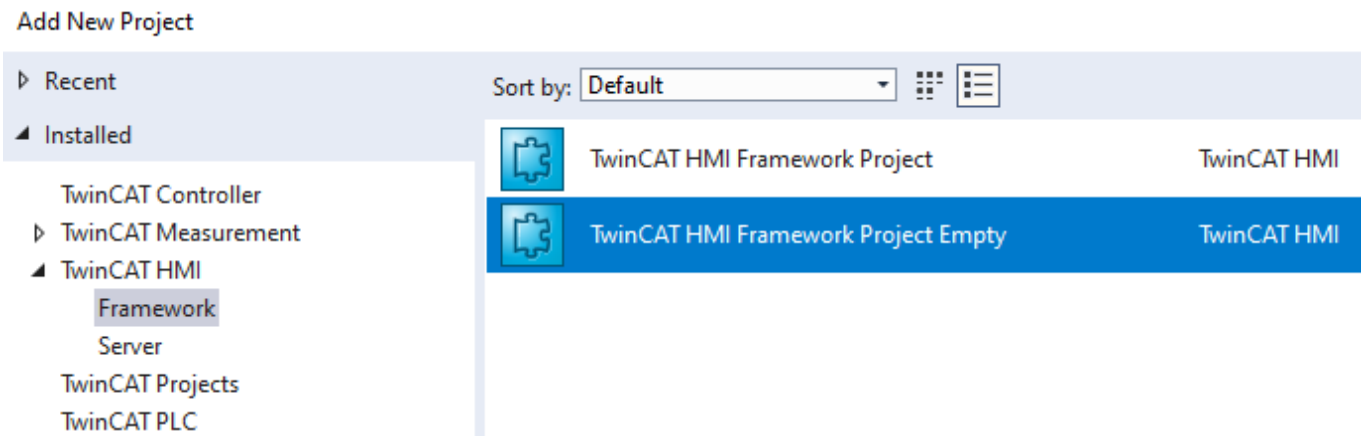
Conversion

To convert this to a Framework control we need to do a few things

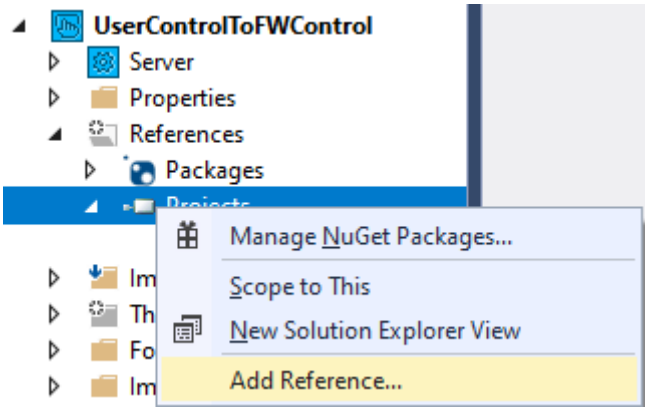
1. Create an FW project
2. Create and FW Control inside the project
3. Copy over the HTML and the parameter information
4. Modify the HTML and parameter information
5. Add the custom datatype
6. Update the `tpl.json` file so the compiler can access all references
7. Add the data communication and bindings to the js/ts

New Framework Project

To create a new FW project, right-click the solution and add a new project. Then select an Empty Framework Project

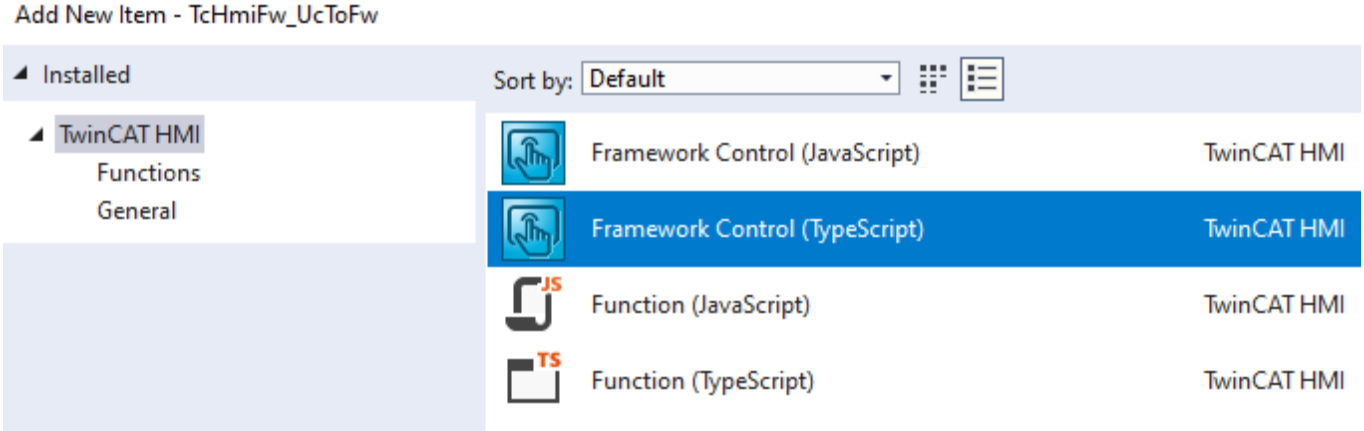


Along with an FW project it is customary to also have a base HMI project which can be used for testing the FW controls. Inside this HMI project you can add the FW project as a reference

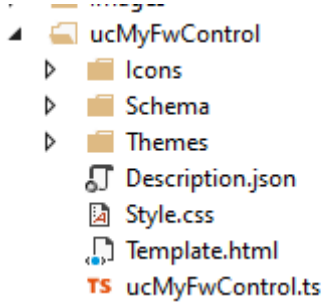


New Framework Control

Then you can add a new FW control to the project. Simply right-click the FW project and choose "Add ⇒ New Item...", then select "Framework Control (TypeScript)"



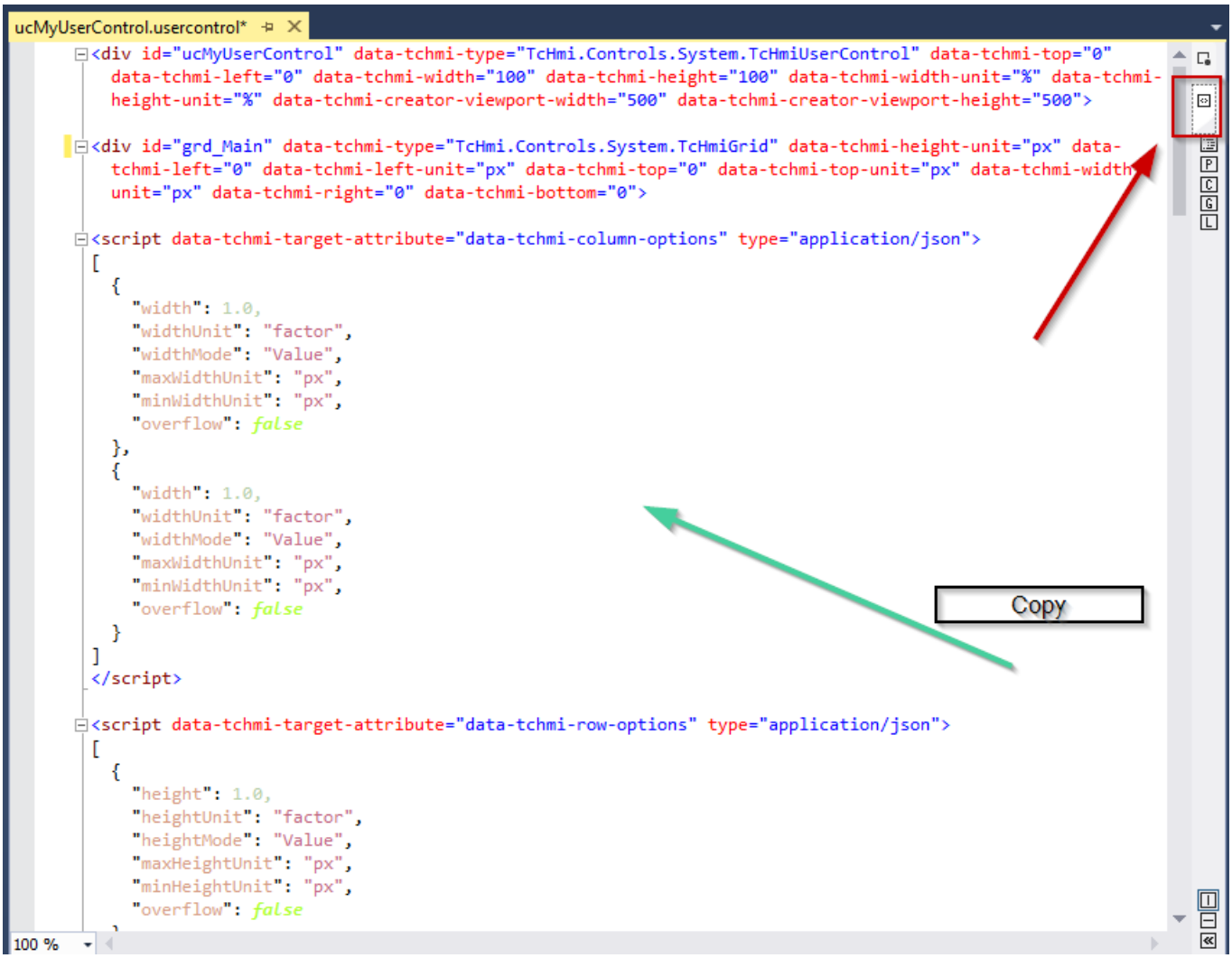
This should add a folder containing the files needed for a basic FW control



Be mindful that the TcXaeShell does not support intellisense for TypeScript. This will require either a full VS version or an external editor (like VS Code).
At this point you should be able to Build the FW Project and the newly created Control should be visible inside the toolbox.

Copy and modify the relevant information

Now go to the User Control inside the base HMI project and copy the HTML code and paste it to the newly created Template.html



And then modify all `id` properties of the HTML to include `{Id}`. This ensures that

the framework will fill in a unique ID for each of these elements. It should look something like this: `id="tbtn_myButton_{Id}"`

Also, remove each of the data bindings, and note the property names. Data bindings can be recognised by the `%` syntax. In this example all bindings are don to parameters, so all bindings will look like this:

```
data-tchmi-text="%pp%myTypeParameter::myString|BindingMode=TwoWay%/pp%">
```

This will not work directly in an FW control, so we need to build these from scratch when attaching the control to the DOM. This will be done in code in a following section.

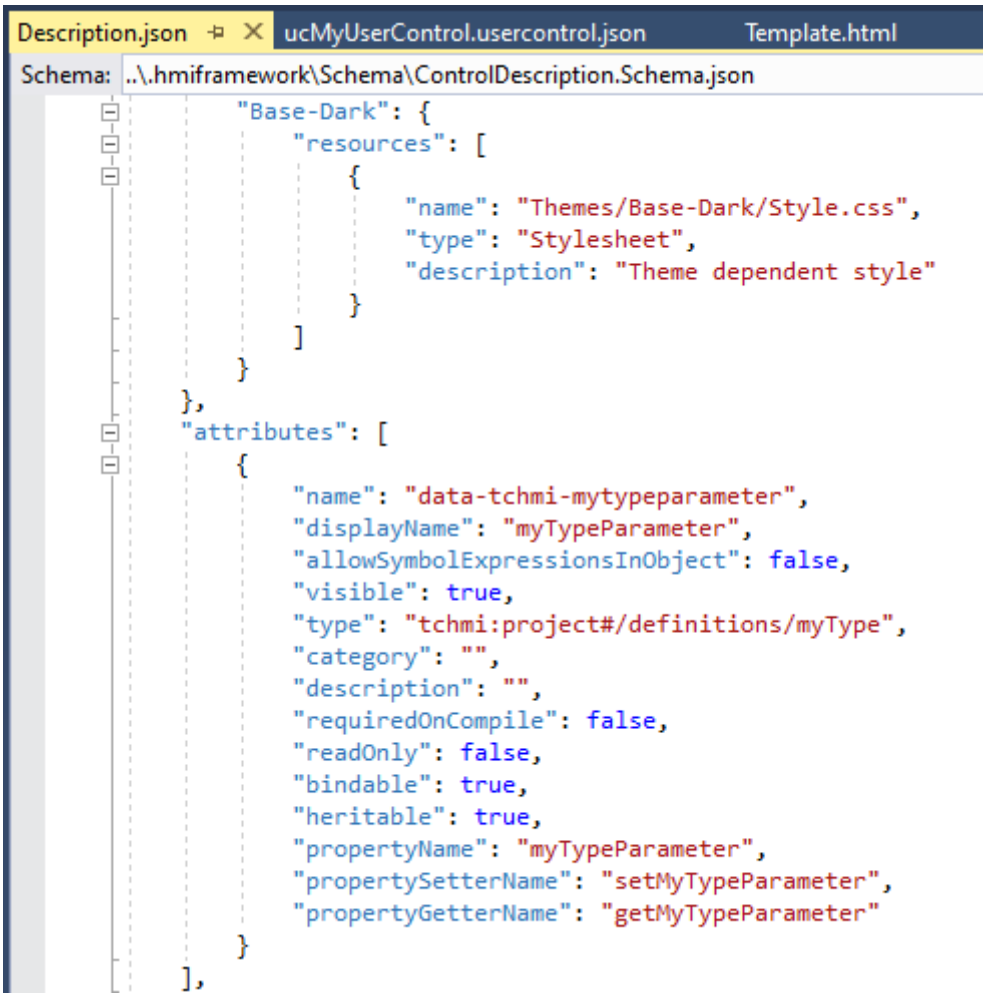
Then we also need to transfer the parameter definition. For this you can open the `<userControlName>.usercontrol.json` file by Right-Clicking and choosing "View Code".

ucMyUserControl.usercontrol.json | Template.html | ucMyUserControl.usercontrol*

Schema: ..\..\Packages\Beckhoff.TwinCAT.HMI.Framework.12.760.59\runtimes\native1.12-tchmi\Schema\UserControlConfig.Schema.json

```
{
  "parameters": [
    {
      "name": "data-tchmi-mytypeparameter",
      "displayName": "myTypeParameter",
      "allowSymbolExpressionsInObject": false,
      "visible": true,
      "type": "tchmi:project#/definitions/myType",
      "category": "",
      "description": "",
      "requiredOnCompile": false,
      "readOnly": false,
      "bindable": true,
      "heritable": true,
      "propertyName": "myTypeParameter",
      "propertySetterName": "setMyTypeParameter",
      "propertyGetterName": "getMyTypeParameter",
      "refTo": ""
    }
  ],
  "virtualRights": [],
  "$schema": "./../../Packages/Beckhoff.TwinCAT.HMI.Framework.12.760.59/runtimes/native1.12-tchmi/Schema/User
  "description": ""
}
```

Then take the parameter definition and copying this to the "description.json" file of the FW control, under the `attributes` item



Feel free to change the names of the properties as required, for more information about the properties look at the documentation on Infosys or in the TE2000 manual.

Add the custom datatype

Custom datatypes can easily be added to the project. When the datatype should also be available inside the TypeScript code this needs to be done in two places. First the FW project itself needs to include the datatype. A default file for adding datatypes has been provided in the skeleton project in the "Schema" folder.
You are free to define the datatype name as you want, though it is useful to namespace this to the name of the FW project as follows (just copy the datatype schema definition from the original datatype from the HMI Configurator):

Types.Schema.json | Description.json | ucMyUserControl.usercontrol.json | Template.html

Schema: http://json-schema.org/draft-04/schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "definitions": {
    "TcHmi.Controls.TcHmiFw_UcToFw.ucMyFwControl": {
      "type": "object",
      "frameworkInstanceOf": "TcHmi.Controls.System.TcHmiControl",
      "frameworkControlType": "ucMyFwControl",
      "frameworkControlNamespace": "TcHmi.Controls.TcHmiFw_UcToFw"
    },
    "TcHmi.Controls.TcHmiFw_UcToFw.myType": {
      "title": "myType",
      "properties": {
        "myBool": {
          "$ref": "tchmi:general#/definitions/Boolean"
        },
        "myNumber": {
          "$ref": "tchmi:general#/definitions/Number"
        },
        "myString": {
          "$ref": "tchmi:general#/definitions/String"
        }
      },
      "type": "object",
      "id": "tchmi:project#/definitions/myType"
    }
  }
}
```

If you want to use the datatype inside TypeScript, then also add an interface as follows (mind the scoping level):

```
5   module TcHmi {
6       export module Controls {
7           export module TcHmiFw_UcToFw {…
144         }
145
146         export namespace TcHmiFw_UcToFw {
147             export interface myType {
148                 myBoolean: Boolean;
149                 myString: String;
150                 myNumber: Number;
151             }
152         }
153     }
154 }
155
156 /**
157  * Register Control
158  */
159 TcHmi.Controls.registerEx('ucMyFwControl', 'TcHmi.Controls.TcHmiFw_UcToFw', TcHmi.Controls.TcHmiFw_UcToFw.ucMyFwControl);
```

Update the typescript compiler so it can access all references.

By default, Beckhoff controls are not included in the TypeScript compiler. This needs to be installed as a Nuget package into the FW project, and communicated to the

compiler. The Nuget package to install is `Beckhoff.TwinCAT.HMI.Controls`. Then the `.tpl.json` file needs to be update with the TypeScript description files (`*.d.ts`) of the desired controls. For this example, the file should be updated like this:

```
"include": [
    "$(Beckhoff.TwinCAT.HMI.Framework).InstallPath/TcHmi.d.ts",
    "$(Beckhoff.TwinCAT.HMI.Controls).InstallPath/TcHmiButton/TcHmiButton.d.ts",
    "$(Beckhoff.TwinCAT.HMI.Controls).InstallPath/TcHmiToggleButton/TcHmiToggleButton.d.ts",
    "$(Beckhoff.TwinCAT.HMI.Controls).InstallPath/TcHmiTextblock/TcHmiTextblock.d.ts",
    "$(Beckhoff.TwinCAT.HMI.Controls).InstallPath/TcHmiTextbox/TcHmiTextbox.d.ts",
    "$(Beckhoff.TwinCAT.HMI.Controls).InstallPath/TcHmiNumericInput/TcHmiNumericInput.d.ts"
],
```

To make sure that external module also know that the Controls package is important, the Manifest.json file should also be updated. Update the file to look like this (the order is important here):

Manifest.json | tsconfig.tpl.json | Desktop.view | Types.Schema.j

Schema: .hmiframework\Schema\Manifest.Schema.json

```
{
  "modules": [
    {
        "type": "Package",
        "nugetId": "Beckhoff.TwinCAT.HMI.Framework"
    },
    {
        "type": "Package",
        "nugetId": "Beckhoff.TwinCAT.HMI.Controls"
    },
    {
        "type": "Control",
        "basePath": "./ucMyFwControl",
        "descriptionFile": "Description.json",
        "toolboxCategory": ""
    }
```

Add data communication and binding

Usually when creating parameters we will need to add setter (and optionally getter) functions inside the code. But since in this example we only use the parameter to set up other data bindings the implementation details are not important. The following code is sufficient in this case:

```
protected __myType: myType | undefined;
public setMyTypeParameter(val: typeof this.__myType) {
    this.__myType = val;
}
public getMyTypeParameter() {
    return this.__myType;
}
```

Of course you are free to use these functions to manually write the data to the

corresponding controls and to add additional functionality to the data communication. For more details, please check out the FW project documentation.

A data binding generally only makes sense while the control is an active part of the DOM. Therefore it is recommended to make bindings only in the `__attach()` function and remove them in the `__detach()` function. This way the control will take up minimal processing power when it is not active.

Data bindings can easily be set up with the `TcHmi.Binding.createEx2()` function. In this case we get the symbol path from the input attribute and then use a helper function to convert this to a symbol

```
// Get paths for bindings
let basePath = this.__attrs["data-tchmi-mytypeparameter"].value as string;
let boolSymbol = this.buildBindingPath(basePath, "myBool");
let numSymbol = this.buildBindingPath(basePath, "myNumber", true);
let stringSymbol = this.buildBindingPath(basePath, "myString", true);
```

Where the helper function is defined as

```
private buildBindingPath(basePath: string, attributeName: string, twoWay: boolean = false): Symbol
{
    let expr = new SymbolExpression(basePath);
    let tag = expr.getTag();
    let rootPath = expr.getFullName();
    return new Symbol(`%${tag}%${rootPath}::${attributeName}${twoWay ? "|BindingMode=TwoWay" : ""}%/${tag}%`);
}
```

Which returns a symbol we can use later to make then binding.
Additionally, to save some typing, we also get the value of the ID of this control

```
const id = this.getId()
```

For the button, which can directly work with a symbol, the coupling is easy to make:

```
// Set up bindings for controls
let btnCtr = TcHmi.Controls.get<TcHmi.Controls.Beckhoff.TcHmiToggleButton>(`tbtn_myButton_${id}`);
btnCtr?.setStateSymbol(boolSymbol);

let btnDisplay = TcHmi.Controls.get<TcHmi.Controls.Beckhoff.TcHmiToggleButton>(`tbtn_Display_${id}`);
btnDisplay?.setStateSymbol(boolSymbol);
```

We use an API function to get a reference to the control. (This is where the previously made change to the HTML comes in. The code tries to find the HTML element with the corresponding name.) Then we can set the state symbol. Here the `?` is shorthand for an if statement that checks whether the control is not null.
Then, for the other bindings we can use the API function in each case. Make sure that the correct property is entered (*Value* vs *Text*, or something else, depending on

the details of the binding):

```
let strCtr = TcHmi.Controls.get<TcHmi.Controls.Beckhoff.TcHmiTextbox>(`tb_myText_${id}`);
if (strCtr !== undefined) {
    TcHmi.Binding.createEx2(stringSymbol.getExpression().toString(), "Text", strCtr);
}

let strDisplay = TcHmi.Controls.get<TcHmi.Controls.Beckhoff.TcHmiTextblock>(`tbl_DisplayString_${id}`);
if (strDisplay !== undefined) {
    TcHmi.Binding.createEx2(stringSymbol.getExpression().toString(), "Text", strDisplay);
}

let numCtr = TcHmi.Controls.get<TcHmi.Controls.Beckhoff.TcHmiNumericInput>(`nin_myNumber_${id}`);
if (numCtr !== undefined) {
    TcHmi.Binding.createEx2(numSymbol.getExpression().toString(), "Value", numCtr);
}

let numDisplay = TcHmi.Controls.get<TcHmi.Controls.Beckhoff.TcHmiTextblock>(`tbl_DisplayNumber_${id}`);
if (numDisplay !== undefined) {
    TcHmi.Binding.createEx2(numSymbol.getExpression().toString(), "Text", numDisplay);
}
```

To remove the bindings in the `__detach()` function, simply repeat the steps from above, but use the `TcHmi.Bindings.removeEx2()` function instead. For more details check the HMI documentation.

Troubleshooting

- The FW project does not compile
  Carefully read the error messages. Usually there is a clear reason why this is not working.
- The Control does not show up in the toolbox
  Make sure the FW project has been added as a reference. If this is the case, make sure that the project has been compiled successfully at least once. If this still does not work, try restarting the TcXaeShell.
- The data binding is not working
  Make sure you are using the correct target for the linked attribute. The symbol expression should also be correct when the binding is made. For debugging it can be useful to add `console.log()` statements, which show up in the browser console (can be opened with F12 on most browsers).