TypeScript Forms

Introduction

TypeScript Forms is a framework for the web which makes it easier to reference html elements and encapsulate logic within controls for re-use. The base elements such as buttons, text boxes, checkboxes etc are encapsulated in a UI control object. Instead of using JavaScript to retrieve the JavaScript objects that represent the html elements the controls are declared in the HTML and constructed into JavaScript objects automatically. The framework is written in TypeScript so all of the controls are available within the d.ts file.

Basics

To use the framework you have to add a reference to the tsf.js to your web page. This file can be located in the TSF project under Scripts/TSF/FrameworkOutput. The .d.ts and map files are also in that directory. To create a basic control all that is required is the var attribute of any html element. This creates a JavaScript variable for this element. In example 1.1 you will see this. As you can see in Example 1.2 you can now set the text on this control by address the global variable that was created by the framework.

Example 1.1

```
<div var="divControl"></div>
```

Example 1.2

```
divControl.Text = 'Hello World';
```

Of course you don't always want to declare everything in the global context. Frequently you will want declare the controls and assign them to some kind of controller. The controller handles the logic for things like displaying data and enabling/disabling of buttons. There are two ways of doing this in this framework. First we will start out with the simple method. In Example 1.3 you can see that there is a control called controller. This javascript variable is declared in the global context but the sub element divControl is assigned to the controller. The preceding "." In the name of the element denotes a relative path. One dot means parent, 2 dots means the parent of the parent and so on. As you can see if you are in a debug window you can now address the divControl through the controller object as exampled in 1.4

Example 1.3

```
<div var="controller">
    <div var=".divControl"></div>
</div>
```

Example 1.4

```
controller.divControl.Text = 'Hello World';
```

Sometimes you will have the desire to encapsulate logic within a control. This framework allows you to do that by specifying the TSClass attribute of any html element. Similarly to how you use the var to declare where you want the object that represents the html element to be set it also allows you to change the class you are creating when the javascript object is created. As you can see in example 1.5 you can specify the class that you want to encapsulate the element. This particular element allows you to change the checkbox to an indeterminate checkbox. That means it will show a – through the checkbox if it was filled with a null value and hasn't been checked or unchecked before. It will also change the context of the onchange event to that of the event handeler in the object. This means that since the `OnCheckedChanged event of the object has the event signature of (sender: UI.Checkbox, args: any).` The this object in this context is the controller; however, if there is no parent control then the this is the object itself (the checkbox). All of the base UI elements are automatically constructed. Example 1.5 is valid but not necessary. If no TSClass if provided it will see the element is a type checkbox and since the name was set in the var attribute it will construct it as a checkbox automatically. You can of course change which class you want it declared as by setting the TSClass. This only works for the provided UI elemnts. If you wish to change the defaults you can do so by looking at the class and changing the value of the TSF.Base.TSBase. `defaultTypes element.`

Example 1.5

```
<div var="controller">
    <input type="checkbox"  TSClass="TSF.UI.Checkbox" indeterminate="true"
onchange="alert(sender.element.checked);" var=".cbxTest" value="Test" />
</div>
```

Events

The framework also comes with event objects that allow multiple listeners to listen for an event that has occurred such as a button click. There are various predefined events as well as a dynamic event class where you can declare a typed event on the fly. Since method calls are screwy in javascript when it comes to the this object, when subscribing to an event you must provide the object you want to use as the this for the method call when the event is fired. The following is an example of how to use an event handeler.

Example 2.1

```
var event = new TSF.Events.EmptyEvent();
event.add(this.methodToFire, this);
```

This adds a listener to the event. The listener will have the methodToFire called when the event is triggered and inside the method when they use the this keyword it will refer to the object that the event was declared in. Similarly there is a remove method to remove a listener from the event handler. To trigger the event simply call the fire method of the event handler. This will alert all of the listeners to the event being triggered. There are various predefined events declared in the framework. Most of the events are located in the TSF.Events namespace. This includes events that take generics to specify what types the inputs to method to be called are. See example 2.2

Example 2.2

```
public methodToFire(message: string)
{
        alert(message);
}

var event = new TSF.Events.ValueEvent<string>();
event.add(this.methodToFire, this);
event.fire("hello world");
```

All of the predefined events are based off of one object which is the event handler. The event handler includes all of the logic for adding methods and removing them but works in a slightly more generic way. You pass in the method signature you are expecting as the generic type. This can be used individually without the use of the predefined event handlers as you can see in Example 3.3. The one issue with this is that the fire method will not have a signature. The underlying method takes an arbitrary number of parameters of any types. To get the signature for the fire event you have to create a new class that inherits from the EventHandler as you can see in example 3.5.

Example 3.4

```
var event = new EventHandler<(value:string) => void>();
event.add(this.methodToFire, this);
```

Example 3.5

```
export class ValueEvent<T> extends EventHandler<(value:T) => void> {
        /**
         * Fires the event
         * @param value - the value to be passed into the listening methods.
         */
        public fire(value:T): void {
            super.fire(value);
    }
}
```

Controls

There are a few controls that come standard with the framework. This list will be expanded in later versions. Right now only the most used controls have classes declared for them. In later documentation full page examples will be available. For now a description of all of the controls will be included in the documentation. More detail can be found in the comments. The following are the provided Controls.

- TSControl – Underlying control for all UI elements. Contains a couple of event handlers and methods for making a control easier. Includes a context variable intended so that you can store context information against a UI element such as a person's contact info. This type of thing can be useful if you have a list of buttons in a grid and you use the context to store the row data on the button itself.
- Button - Standard button. Comes with enumerations for some of the specific properties such as button type.

- Checkbox – Has checked changed event handler as well as properties for using turning on the indeterminate feature of checkboxes.
- DropDown – Drop down control with events for on change and the ability to store actual objects as values of the drop down items.
- Image – standard image control where you can set the image you want to display and listen for click events.
- LogicalControl – Base class for all controllers. This control has useful methods for loading attributes more easily and getting absolute paths from relative paths set in attributes of a control.
- MultiSelect – Multi select drop down control in html. Includes event handlers and the ability to store actual objects as values of the drop down items.
- RadioButton – Event handelers and functionality for managing radio buttons in html.
- TextBox – Includes event handlers for on text change.
- WunderGrid – A fully featured grid control for displaying data. Built in functionality for sorting, resizing columns, auto sizing columns on double click, row selection and event handelers for interfacing with the grid.

Remote Calls

The framework also includes methods for making remote calls more easily. You can of course still us JQuery or regular JavaScript to make remote calls; however, a static Remote class is included in the TSF.Remote namespace to make things easier. This includes an inline method call and a fluent style remote call. Both return promises that can be used with async await in typescript or ECMA6 javascript. One thing to keep in mind though is that debugging code using async await when compiling into ECMA5 is not supported by all browsers at the time this document was written. The browsers lose their place in the code once they hit one of these lines and it becomes impossible to debug. Example 4.1 and 4.2 shows some of the ways that the remote class can be used. Advanced options and headers are optional. Details can be found in the comments of the class. Example 4.2 shows an inline call. There are more input arguments available that you can optionally provide. Both methods allow for the same input and both return a promise. The fire startStop events option triggers an event in the `Remote.RemoteCall class. This fires a start event if no other calls are currently being made and a new call is created. Once all active calls have finished the stop event is fired. This allows you to show loading signs on your screen if desired by subscribing to the OnStart and OnStop events.`

Example 4.1

```
new Remote.RemoteCall(this.dataUrl, dataInput).advancedOptions(Remote.RequestType.POST,
this.fireStartEndEvents).headers({ header1: 5, header2: 'value2' }).call().then((data) =>
{
     this.data = data;
     this.updateDS(this.data);
}).catch((ex) => {
     console.error(ex);
     this.onError.fire(ex);
```

```
});
```

Example 4.2

```
Remote.RemoteCall.callInline(this.dataUrl,inputdata).then((data) => {
     this.data = data;
     this.updateDS(this.data);
}).catch((ex) => {
     console.error(ex);
     this.onError.fire(ex);
});
```

Data

The framework also comes with data sources. Data sources encapsulate common functionality when retrieving data from the database. They have events for when the data has been retrieved as well as start and end events to the remote call. They also provide the ability to do server side paging, and have the ability to filter the data source.

Server side paging is often times done when the data source you are querying is to large to return to the server. So instead you return small chunks at a time. The data source has a common object that it passes up to the server that denotes the start and end record indexes that it expects. The data source takes a dataurl and a counturl. If only the dataurl is provided then it is expected no server side paging will take place. If the count url is provided it will behave in a way that expects the data to be paged server side. The count url should point to a method that returns the maximum number of records possible for the data set you are querying. The data url will actually do the query and retrieve the data. The data source will call the count url first to update the page count and then call the data url to retrieve the data for the specific page.

The data object that the data source passes up can also include filters and sorting as well as record start an stop indexes. The filter criteria Can be hierarchical if desired. Each CriteriaGroup encapsulates its filter criteria in parenthesis. These allows for a standard method to specify filter criteria. A similar approach is taken for the sort order. The framework comes with the data structures for the filter criteria but does not at the moment contain code to convert it into SQL. This will be added at a later date.

The table that is provided with the framework can be connected to a data source. By doing this you can display the data that is retrieved by the data source. Once the data source is assigned to the table simply refreshing the data source will cause the grid to refresh. The grid of course can also be bound by calling the bind method and bypassing data sources if desired.