CREATING IONIC APPLICATIONS WITH STENCILJS

JOSHUA MORONY

Preview Lesson

This PDF provides a preview of one of the lessons from Creating Ionic Applications with StencilJS. The lesson below is from the **Basics** chapter, which covers all of the important concepts you will need to know to build applications with StencilJS and Ionic.

For more details about the rest of the book, and if you would like to purchase the book, please click here.

Lesson 12: Handling Forms & User Input

Some frameworks will have some built-in mechanism for handling forms and form data/validation and so on. With StencilJS we aren't using a "framework" and like with a lot of aspects of building an Ionic/StencilJS application there isn't a particular "StencilJS way" to handle forms - it is up to us to handle it as we wish with standard JavaScript.

As always, this is good in some ways in that it provides us with more freedom, but also bad in the sense that often you won't know where to start if you've never done it before. With that in mind, we can lean into general JavaScript principles/best-practices for handling forms.

On the surface, it seems simple. We have Ionic components like `<ion-input>` or `<ion-textarea>` that will allow the user to enter values into them, and then we want to use those values. However, there is actually a lot of functionality we may need to set up like:

- Binding the value of an input field to a variable we can access
- Updating a form value when the input changes
- Updating a form value when the variable it is bound to changes
- Check if a value for a particular field is valid
- Check if the entire form is valid

You may not need to tackle all of these problems for every form, but there are many different ways you could approach this, and without careful consideration, you could end up with a large and messy solution.

In this lesson, we are first going to look at handling a very simple form, and then we will look at a slightly more advanced form that also handles validating the input the user supplies.

IMPORTANT: When talking about form validation, I always like to highlight that validation on the client-side (like this) is purely for user experience. Any security related data validation/sanitisation should happen on the backend (e.g. a server or cloud functions) where the user can't just modify the code to bypass your validations.

A Basic Form

We'll take a look at the "logic" side of things first, and then we will look at the template that makes use of it.

```
@State() formControls = {
  username: null,
  password: null
};

changeFormValue(controlName, value) {
  this.formControls = {
    ...this.formControls,
    [controlName]: value
  };
}

handleSubmit(e) {
  e.preventDefault();
  console.log(this.formControls);
}
```

At first glance, this might look quite convoluted/confusing. You could certainly take an easier approach to doing this, but the method I am about to outline will scale well when your form has a lot of inputs. With just one or two inputs in your form, you could easily just manage things manually and set up individual variables/change handlers for each input. However, if you need to write a function for every one of your fields, along with a variable, and possibly even a validation function, things will get messy fast. This method avoids that and will be much easier once you get your head around it. Let's talk through what is happening here.

First, we have a `formControls` object set up with the `@State()` decorator. The purpose of this object is to hold the values for all of the inputs we will have in our form. You could give these some initial values if you like, or just leave them as `null`. The reason we use `@State()` instead of just a normal class member variable is that StencilJS does not have two-way data binding - e.g. if we were to update the `username` value in the `formControls` object, it would not be reflected in the `username` input field. By using `@State()` the `render()` function will be triggered each time the `formControls` object changes. This means that the `username` input field will update with the appropriate value if we modify its value in `formControls`.

Next, we have the `changeFormValue` function. The purpose of this function is to update the value of a particular `formControl` - in just a moment, we will trigger this function in our template whenever the input value for a field changes. We simply supply it with the name of the `formControl` that we want to change (e.g. `username` or `password`) and the value we want to change it to.

Remember that if we were simply to do something like this:

```
this.formControls[controlName].value = value;
```

It would not trigger the `render` function to update our template since `@State()` does not compare differences in objects like this. Instead, we need to create an entirely new object, which we can do like this:

```
this.formControls = {
    ...this.formControls
}
```

However, we don't want to just duplicate the object. We want to provide the new value. By adding in the second line:

```
    this.formControls = {
      ...this.formControls,
      [controlName]: value
    };
```

We are effectively saying "create a new formControls object based on the last one, except replace the old value of the supplied controlName with the new value". If we were to make a call to `this.changeFormValue('username', 'josh')` then it would create a new `formControls` object exactly the same as the last one, except it would update the `username` property to be `josh`.

Finally, we have the `handleSubmit` method. This just basically intercepts the form submission and prevents its default behaviour (which would cause our app to refresh). For now, we are just logging out the values, but you could do whatever you like with the submitted data.

Now let's see how we make use of this in our template:

```
render() {
  return [
    <ion-header>
      <ion-toolbar color="primary">
        <ion-title>Basic Form</ion-title>
      </ion-toolbar>
    </ion-header>,

    <ion-content class="ion-padding">
      <form onSubmit={e => this.handleSubmit(e)} novalidate>
        <ion-list lines="none">
          <ion-item>
            <ion-label position="stacked" color="primary">
```

```
              Username
            </ion-label>
            <ion-input
              name="username"
              type="text"
              value={this.formControls.username}
              onInput={(ev: any) =>
                this.changeFormValue("username",
ev.target.value)
              } />
          </ion-item>

          <ion-item>
            <ion-label position="stacked" color="primary">
              Password
            </ion-label>
            <ion-input
              name="password"
              type="password"
              value={this.formControls.password}
              onInput={(ev: any) =>
                this.changeFormValue("password",
ev.target.value)
              } />
          </ion-item>
        </ion-list>

        <div class="ion-padding">
          <ion-button type="submit" expand="block">
```

```
                Log in
              </ion-button>
            </div>
          </form>
        </ion-content>
      ];
    }
```

The general idea is pretty straight-forward, we just bind the `value` of each of our inputs to the corresponding control in `formControls`, e.g:

```
value={this.formControls.password}
```

We also bind the `onInput` event for each input to the `changeFormValue` method and pass in the control name and the value from the input field. It is necessary to give the event an `: any` type here, as otherwise type checking will fail when attempting to access `ev.target.value`. Passing just the value, rather than the entire event, to the method allows us to update form values both through the `onInput` event or by just manually calling `changeFormValue`.

The only other thing of note here is that we set up the following event:

```
onSubmit={e => this.handleSubmit(e)}
```

on the `<form>`. This will cause our `handleSubmit` method to trigger when the form is submitted (which will also prevent the normal submission behaviour of the form).

For reference, here is the completed example:

```
import { Component, State, h } from "@stencil/core";

@Component({
  tag: "app-home",
  styleUrl: "app-home.css"
})
export class AppHome {
  @State() formControls = {
    username: null,
    password: null
  };

  changeFormValue(controlName, value) {
    this.formControls = {
      ...this.formControls,
      [controlName]: value
    };
  }

  handleSubmit(e) {
    e.preventDefault();
    console.log(this.formControls);
  }

  render() {
    return [
      <ion-header>
        <ion-toolbar color="primary">
          <ion-title>Basic Form</ion-title>
        </ion-toolbar>
      </ion-header>,

      <ion-content class="ion-padding">
        <form onSubmit={e => this.handleSubmit(e)} novalidate>
          <ion-list lines="none">
            <ion-item>
              <ion-label position="stacked" color="primary">
                Username
              </ion-label>
              <ion-input
                name="username"
                type="text"
                value={this.formControls.username}
                onInput={(ev: any) =>
                  this.changeFormValue("username",
ev.target.value)
                } />
            </ion-item>

            <ion-item>
```

```
              <ion-label position="stacked" color="primary">
                Password
              </ion-label>
              <ion-input
                name="password"
                type="password"
                value={this.formControls.password}
                onInput={(ev: any) =>
                  this.changeFormValue("password",
ev.target.value)
                } />
            </ion-item>
          </ion-list>

          <div class="ion-padding">
            <ion-button type="submit" expand="block">
              Log in
            </ion-button>
          </div>
        </form>
      </ion-content>
    ];
  }
}
```

An Advanced Form With Validation

The approach above is fine for many different scenarios, but we may also want to validate the data that is entered into field. We might want to display a message to the user to indicate when a field is not valid, and we might want to prevent form submission if all of the fields are not valid.

We are going to use the same general approach as above, but we are going to extend it a little to include the ability to validate fields. Again, we'll take a look at the logic first, and then the template.

```
@State() formControls = {
  username: {
    value: null,
    validate: value => {
      if (value) {
        return true;
      } else {
        return false;
      }
    },
    isValid: false
  },
  password: {
    value: null,
    validate: value => {
      if (value) {
        return true;
      } else {
        return false;
      }
    },
    isValid: false
  }
};

@State() submitted = false;

componentDidLoad() {
  // Set up username validate function
  this.formControls.username.validate = value => {
    if (value && value.length > 4) {
      return true;
    } else {
      return false;
    }
  };

  // Set up password validate function
  this.formControls.password.validate = value => {
    if (value && value.length > 8) {
      return true;
    } else {
      return false;
    }
  };
}
```

```
changeFormValue(controlName, value) {
  this.formControls = {
    ...this.formControls,
    [controlName]: {
      ...this.formControls[controlName],
      value: value,
      isValid: this.formControls[controlName].validate(value)
    }
  };
}

handleSubmit(e) {
  e.preventDefault();

  this.submitted = true;
  let isFormValid = true;

  //Run all validation functions
  for (let controlName in this.formControls) {
    let control = this.formControls[controlName];
    control.validate(control.value);
    if (!control.isValid) {
      isFormValid = false;
    }
  }

  console.log(this.formControls);
  console.log("Form valid: ", isFormValid);
}
```

Much is the same, but there is a considerable amount more code now. First of all, we have changed the structure of our `formControls` and we have included an additional `@State()` member variable:

```
@State() formControls = {
  username: {
    value: null,
    validate: value => {
      if (value) {
        return true;
      } else {
        return false;
      }
    },
    isValid: false
  },
  password: {
    value: null,
    validate: value => {
      if (value) {
        return true;
      } else {
        return false;
      }
    },
    isValid: false
  }
};

@State() submitted = false;
```

Now, instead of just storing a simple value, we store three things for each control: a `value`, a `validate` function, and an `isValid` flag. The `validate` function will provide the function that we want to use to check if the value supplied to the input is valid or not. By default, we just set these to return `true` as long as a value is supplied, but we can overwrite them with more complicated validation functions later if we like (as we do in the `componentDidLoad` method).

The purpose of the `submitted` variable is just to check if the form has been submitted yet - of course, as soon as the page loads the fields would be invalid because the user hasn't had a chance to enter anything yet. We won't actually display any error messages until they attempt to submit the form. Now let's take a look at the `componentDidLoad` hook:

```
  componentDidLoad() {
    // Set up username validate function
    this.formControls.username.validate = value => {
      if (value && value.length > 4) {
        return true;
      } else {
        return false;
      }
    };

    // Set up password validate function
    this.formControls.password.validate = value => {
      if (value && value.length > 8) {
        return true;
      } else {
        return false;
      }
    };
  }
```

We are just using this to overwrite the initial `validate` functions with some more complex ones. You can perform whatever kind of logic you like - just return `true` for any valid values, and `false` for any invalid values.

```
  changeFormValue(controlName, value) {
    this.formControls = {
      ...this.formControls,
      [controlName]: {
        ...this.formControls[controlName],
        value: value,
        isValid: this.formControls[controlName].validate(value)
      }
    };
  }
```

The `changeFormValue` method remains more or less the same, except that we have updated it appropriately overwrite the new structure of each control (now each control is its own object, as opposed to before where it was just a simple value). We copy in the previous structure of the control object using the spread operator again, and then we overwrite `value` and `isValid` with the new values. The value of `isValid` is determined by running the `validate` function of the particular control against the `value` that was supplied.

```
handleSubmit(e) {
  e.preventDefault();

  this.submitted = true;
  let isFormValid = true;

  //Run all validation functions
  for (let controlName in this.formControls) {
    let control = this.formControls[controlName];
    control.validate(control.value);
    if (!control.isValid) {
      isFormValid = false;
    }
  }

  console.log(this.formControls);
  console.log("Form valid: ", isFormValid);
  }
```

In our `handleSubmit` function, we now mark `submitted` as true to indicate that there has been a form submission attempt. Since our `validate` function only run when the value is changed in a field, we need to make sure we run all of our `validate` functions manually in order to ensure that the form is valid. To do this, we loop through each `property` in our `formControls` object. We then set up a reference to each control, and manually call the `validate` method with the current value of the control. After doing that, if the current `isValid` flag is `true` we do nothing. If the current `isValid` flag is `false` then we switch our `isFormValid` boolean to false.

We can then log out all of the values for the form, and also whether or not the form is valid (e.g. all inputs in the form are marked as being valid).

The template remains pretty much the same, except that we need to update the `value` to reflect the new structure of `formControls`. We also add in some error messages that only display when a particular field is marked as invalid, and when a form submission attempt has already been made. You can see these changes in the completed example below:

```
import { Component, State, h } from "@stencil/core";

@Component({
  tag: "app-profile",
  styleUrl: "app-profile.css"
})
export class AppProfile {
  @State() formControls = {
    username: {
      value: null,
      validate: value => {
        if (value) {
          return true;
        } else {
          return false;
        }
      },
      isValid: false
    },
    password: {
      value: null,
      validate: value => {
        if (value) {
          return true;
        } else {
          return false;
        }
      },
      isValid: false
    }
  };

  @State() submitted = false;
```

```
componentDidLoad() {
  // Set up username validate function
  this.formControls.username.validate = value => {
    if (value && value.length > 4) {
      return true;
    } else {
      return false;
    }
  };

  // Set up password validate function
  this.formControls.password.validate = value => {
    if (value && value.length > 8) {
      return true;
    } else {
      return false;
    }
  };
}

changeFormValue(controlName, value) {
  this.formControls = {
    ...this.formControls,
    [controlName]: {
      ...this.formControls[controlName],
      value: value,
      isValid: this.formControls[controlName].validate(value)
    }
  };
}

handleSubmit(e) {
  e.preventDefault();

  this.submitted = true;
  let isFormValid = true;

  //Run all validation functions
  for (let controlName in this.formControls) {
    let control = this.formControls[controlName];
    control.validate(control.value);
    if (!control.isValid) {
      isFormValid = false;
    }
  }

  console.log(this.formControls);
  console.log("Form valid: ", isFormValid);
}

render() {
  return [
    <ion-header>
      <ion-toolbar color="primary">
        <ion-title>Advanced Form</ion-title>
      </ion-toolbar>
      </ion-header>,

      <ion-content class="ion-padding">
        <form onSubmit={e => this.handleSubmit(e)} novalidate>
          <ion-list lines="none">
            <ion-item>
              <ion-label position="stacked" color="primary">
                Username
              </ion-label>
              <ion-input
                name="username"
                type="text"
                value={this.formControls.username.value}
                onInput={(ev: any) =>
                  this.changeFormValue("username",
ev.target.value)
                } />
            </ion-item>
            <ion-text color="danger">
              <p
                hidden={
                  this.formControls.username.isValid ||
this.submitted === false
                }
                padding-left
              >
                Username invalid
              </p>
            </ion-text>

            <ion-item>
              <ion-label position="stacked" color="primary">
                Password
              </ion-label>
              <ion-input
                name="password"
                type="password"
                value={this.formControls.password.value}
                onInput={(ev: any) =>
                  this.changeFormValue("password",
ev.target.value)
                } />
            </ion-item>
            <ion-text color="danger">
              <p
                hidden={
                  this.formControls.password.isValid ||
this.submitted === false
                }
                padding-left
              >
                Password invalid
              </p>
            </ion-text>
          </ion-list>

          <div class="ion-padding">
```

```
            <ion-button type="submit" expand="block">
              Log in
            </ion-button>
          </div>
        </form>
      </ion-content>
    ];
  }
}
```

As I mentioned in the beginning, there are many ways you could go about handling forms in your application and if you feel the desire to modify this approach to suit your needs then go right ahead. You might require something simpler or you might require something more advanced. I feel like this approach provides a nice balance between complexity and functionality.