Forms:

* Angular provides a good amount of ways to handle HTML forms. This includes error detection, data formatting, etc. It offers 2 approaches - *Template-driven* and *Reactive*.
* In the first approach, we only have to create the HTML template of the form and Angular infers the structure of the data (the javascript representation of the form). In the second approach, we manually create the template and the data structure and connect the both for more precise control.
* The forms do not have the action attribute as we want Angular to handle the forms first.

Template-driven approach:

* To get this working, we must import the FormsModule from @angular/forms as it contains a lot of features that we will need to use.
* By default, we cannot see the data structure of the form. Sometimes, we may not need every input value to be part of the final data to be submitted. To control this, we use ngModel (not like how we used it for two-way binding). We also add the name attribute to the control. Default value can be set using one-way binding - [ngModel].
* To see the data, we can submit the data using the ngSubmit event provided by a directive with the same name. The form itself should have a local-reference, which is to be submitted as a parameter to the submit function.
* Now if we run the app, we will get the Angular generated HTML tag as response. To get this in JSON, we have assign ngForm as value to the local-reference of the form. What this does is it informs Angular that this is a form and that we need to get the data in the form in JSON format.
* With this, we get an object in response with a lot of useful information and functions. We can get the data from the form in JSON format from the value property provided by the object.
* We can also get access to the form data via the @Viewchild.

1. Validations:

* Angular has some validation logic baked into itself for forms. These are directives that we add to the inputs. required and email are 2 examples.
* We can confirm the validity of a form by looking into the valid property in ngForm. We can also check it within each control exposed by the form.
* Each input that we track have a set of classes add by Angular based on its state. ng-dirty and ng-invalid are examples of the same. These can be used to add styling to the respective inputs based on its state.
* While adding styles, we have to make sure that the right CSS selectors are in place as these classes are used on the form and all its controls as well. To avoid these classes from styling in the beginning itself, we use another class named ng-touched.
* We can also show a message for each invalid control value. To do that, we first must have a local-reference to the control we are checking. Just like the form, the local-reference is given ngModel as value.
* The message is kept in a span below the control and will use the \*ngIf directive to check if the control is valid and is touched.
* We can enable HTML validation using the ngNativeValidate directive (Angular disables it by default).
* We can also group values of a form using ngModelGroup directive. This will take a string value which will be used as a key in the JSON data. This will now create a separate logical control for the group and will have all the properties of a normal control.
* An error message can be displayed for a group also by assigning ngModelGroup to its local-reference.

1. Setting and Patching Form values:

* We can set data of our form from the TS file itself. This can be achieved using two-way binding. But lets look at a different approach.
* If we have access to the form via ViewChild, we can use the setValue function to set the form data. This function takes an object, that is the exact representation of the form JSON data, as a parameter.
* The disadvantage of using this is that we will be updating all the values in the form. If we want to change just one value, we have to use a different approach.
* The ViewChld is actually just a container to the main form. We can get access to the main form via the form property. This property has a patchValue method, which takes an object as an argument. In this object, we set our required value for the required field. We still have to mention it in the exact format of the JSON output.

Reactive approach:

* This approach creates the form programatically from the TS file. Angular has a bunch of tools to assist in this as well. To make a reactive form, we first need to import the ReactiveFormsModule from @angular/forms into the app module.
* We can create a form of type FormGroup imported from @angular/forms. We initialize the form in the ngOnInit function.
* To the form group instance, we add a JSON object which takes string keys and FormControl values, imported from @angular/forms.
* The control can take a couple of values as arguments. The first one is the default value.
* To connect this TS form to the HTML form, we bind the formGroup directive to the HTML form to the TS form group. Similarly, to bind the controls, we use formControlName and pass the control as a string key.
* While submitting, there is a subtle difference between the 2 approaches. We don’t need the local-reference in this approach as we created the form structure ourselves.
* Validation also works different here. We cannot add directives to the HTML form to validate the form. We use the Validators class imported from @angular/forms. Angular comes with a set of validators which we can use such as required. Note - Do not call it as a method. We only have to refer to the method.
* The validators are added as the second parameter of a FormControl. This can either be a single validator or an array of validators.
* Showing error messages also has a small bit of change. As we are using ngModel anymore, we have to get the value from the bound TS form. We can do this by using the get method provided by the TS form - <TS\_FORM\_NAME>.get(<PROPERTY\_NAME>). This gives us access to all the other checks that we had in the template approach - valid, dirty, touched, etc.
* We can have nested form groups if we keep related data together. We simply add another form group to our TS form group and add the controls that are related to each other.
* In HTML side, we have to just add the formGroupName in the grouping div.

Dynamically adding controls:

* If we need to add an array of controls into a form, we use FormArray imported from @angular/forms. This takes in an array of controls or null as the first argument.
* To bind the array to the HTML we use the formArrayName directive and assign the name to it as value. The hobbies entry in the form group must be type cast to FormArray in order to perform the related actions such push.
* The formControlName while looping through this array is the index itself and hence is bound as a property (not like a directive).

Custom Validators:

* If we want to reserve a set of usernames from being used, we will need our own validators to enforce this. We can do this by writing a validator function, which is a normal function itself.
* This function takes a control as parameter and returns either null or a key value pair if true. We use the value of the control to validate the entry. If the data is invalid, we return a key value pair, where the value is a boolean. Else we return null (not false).
* The custom validator is then added to the respective controls. We have to use the bind(this) function so that when the validator is executed, the correct reference to this is maintained.
* To show error messages based on the type of error, we can access the errors property of the control and check for the error code.
* Validations can also be asynchoronous. We can create asynchronous validators too. This function returns a Promise or Observable. These validators, however, are not added with the regular validators. The FormControl instance takes a 3rd argument which is either a single asynchronous validator or an array of them.
* Forms have 2 observables that we can use - valueChanges and statusChanges. They both do what their suggest, detects changes.

Resetting Forms:

* We can reset forms by calling the reset function. All the entered data will be removed in this case.
* We can remove specific values by passing an object as an argument, which contains the data fields as keys and the reset value.