Template-driven forms in Angular are a way of creating forms using Angular directives and binding directly in the template. They are often simpler and more declarative than reactive forms, making them suitable for less complex forms where most of the logic resides in the template.

Here's a detailed breakdown of template-driven forms in Angular:

1. **Import FormsModule:**

In your Angular module, import the `FormsModule` to enable template-driven forms.

```
// app.module.ts
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule],
  // ...
})
export class AppModule { }
```

2. **Creating a Simple Form:**

In your component's template, use the `ngForm` directive to create a form. Bind form controls using `ngModel`.

```
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" ngModel required>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" ngModel required>

  <button type="submit">Submit</button>
</form>
```

3. **Form Submission:**

Handle form submission by binding to the `(ngSubmit)` event and defining a corresponding method in your component.

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```
})
export class AppComponent {
  onSubmit() {
    // Form submission logic
    console.log('Form submitted!');
  }
}
```

4. **Form Controls and Validation:**

Use `ngModel` to bind form controls to properties in your component. You can apply built-in and custom validators.

```
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" ngModel required minlength="3"
maxlength="30">

  <div *ngIf="myForm.controls.name.invalid && myForm.controls.name.touched">
    <p *ngIf="myForm.controls.name.errors.required">Name is required.</p>
    <p *ngIf="myForm.controls.name.errors.minlength">Name must be at least 3
characters.</p>
    <p *ngIf="myForm.controls.name.errors.maxlength">Name cannot exceed 30
characters.</p>
  </div>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" ngModel required email>

  <div *ngIf="myForm.controls.email.invalid && myForm.controls.email.touched">
    <p *ngIf="myForm.controls.email.errors.required">Email is required.</p>
    <p *ngIf="myForm.controls.email.errors.email">Invalid email format.</p>
  </div>

  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

5. **Two-Way Data Binding:**

`ngModel` provides two-way data binding, allowing you to bind a property in the component to the input field.

```
<!-- app.component.html -->
<label for="name">Name:</label>
<input type="text" id="name" name="name" [(ngModel)]="formData.name" required>
```

6. **NgModelGroup:**

Use `ngModelGroup` to group related form controls.

```
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  <div ngModelGroup="user">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" ngModel required>
  </div>

  <div ngModelGroup="emailGroup">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" ngModel required email>
  </div>

  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

Summary:

Template-driven forms provide a quick and easy way to build forms in Angular by using directives directly in the template. They are suitable for simpler forms where most of the logic is in the template itself. However, for more complex scenarios, reactive forms might be a better choice. Understanding the strengths and limitations of both approaches helps in making an informed decision based on the requirements of your application.

Template-driven forms in Angular are a convenient way to handle forms directly in the template using directives like `ngForm`, `ngModel`, and others. Here are some important things to know about template-driven forms:

1. **Two-Way Data Binding:**
    - Template-driven forms use two-way data binding with `ngModel` to bind form controls to properties in the component. This allows changes in the component to automatically update the form and vice versa.
2. **ngForm Directive:**
    - The `ngForm` directive is used to create a form. It is automatically added to a form element when using `ngModel` within it.
3. **ngModel Directive:**
    - The `ngModel` directive binds an input, select, textarea, or custom form control to a property on the component. It provides two-way data binding for form controls.

4. **Form Validation:**
    - Template-driven forms support both built-in and custom validation using attributes like `required`, `minlength`, `maxlength`, etc. Error messages can be displayed based on control state using `ngIf` and control properties like `touched` and `invalid`.
5. **Form Submission:**
    - Form submission is handled using the `(ngSubmit)` event on the form element. You can associate a method in the component to be executed when the form is submitted.
6. **Disabled State:**
    - The `[disabled]` attribute can be used to disable the submit button until the form is valid. This helps prevent users from submitting incomplete or incorrect data.
7. **Template Reference Variables:**
    - Template reference variables (e.g., `#myForm`) can be used to reference the entire form or individual form controls. These variables can be used to access form control properties and methods in the component.
8. **NgModelGroup:**
    - `ngModelGroup` is used to group related form controls together. It allows you to organize controls within the form and provides a way to validate the group as a whole.
9. **NgModel Change Tracking:**
    - The `ngModel` directive keeps track of changes to the form controls and their values, making it easy to react to user input in the component.
10. **NgModel Change Events:**
    - You can use events like `(ngModelChange)` to execute custom logic when the value of an `ngModel` changes. This can be useful for real-time validation or other dynamic behavior.
11. **NgModel Properties:**
    - `ngModel` exposes properties such as `touched`, `dirty`, and `valid`, which can be used to apply dynamic styles or display specific messages based on the state of the form controls.
12. **NgModel Options:**
    - The `ngModel` directive supports options like `name`, `standalone`, and `ngModelOptions` for fine-tuning the behavior of form controls.
13. **NgForm Properties:**
    - The `ngForm` directive provides properties like `submitted`, `form`, and `controls` that can be accessed in the component for advanced form handling.
14. **FormsModule Dependency:**

- To use template-driven forms, you need to import the `FormsModule` in your Angular module.

Understanding these aspects of template-driven forms is crucial for effectively working with forms in Angular applications. It's important to evaluate whether template-driven forms are the right choice for your specific use case or if reactive forms might be more suitable. Each approach has its own strengths and considerations.

Example Login `Form`

Below is an example of a template-driven form for a simple login functionality with a username and password. This example demonstrates how to create a login form, perform basic validation, and handle form submission.

```
<!-- app.component.html -->
<div class="login-container">
  <h2>Login</h2>
  <form #loginForm="ngForm" (ngSubmit)="onSubmit()">
    <div class="form-group">
      <label for="username">Username:</label>
      <input
        type="text"
        id="username"
        name="username"
        ngModel
        required
        minlength="3"
        maxlength="20"
      />
      <div *ngIf="loginForm.controls.username.invalid &&
loginForm.controls.username.touched">
        <p *ngIf="loginForm.controls.username.errors?.required">Username is
required.</p>
        <p *ngIf="loginForm.controls.username.errors?.minlength">
          Username must be at least 3 characters.
        </p>
        <p *ngIf="loginForm.controls.username.errors?.maxlength">
          Username cannot exceed 20 characters.
        </p>
      </div>
    </div>

    <div class="form-group">
      <label for="password">Password:</label>
      <input type="password" id="password" name="password" ngModel required
minlength="6" />
```

```
      <div *ngIf="loginForm.controls.password.invalid &&
loginForm.controls.password.touched">
        <p *ngIf="loginForm.controls.password.errors?.required">Password is
required.</p>
        <p *ngIf="loginForm.controls.password.errors?.minlength">
          Password must be at least 6 characters.
        </p>
      </div>
    </div>

    <button type="submit" [disabled]="loginForm.invalid">Login</button>
  </form>
</div>
```

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  onSubmit() {
    // Perform login logic here
    console.log('Login successful!');
  }
}
```

In this example:

- The form uses `ngForm` to create a template-driven form.
- Two form controls (`username` and `password`) are created using `ngModel` for two-way data binding.
- Basic validation is applied using attributes like `required`, `minlength`, and `maxlength`.
- Error messages are displayed conditionally based on the form control state and user interaction.
- The `(ngSubmit)` event is used to trigger the `onSubmit()` method in the component when the form is submitted.
- The submit button is disabled if the form is invalid.

Remember to add appropriate styling to enhance the visual presentation of your form. This is a basic example, and in a real-world scenario, you would typically communicate with a backend service to authenticate the user.

Template-Driven Forms in Angular

Overview:
Template-Driven Forms in Angular are suitable for developing simple forms with a limited number of fields and straightforward validations. Each field is represented as a property in the component class. Validation rules are defined in the template using HTML5 attributes, and validation messages are displayed using Angular's validation properties.

Key Points:

1. **FormsModule Import:**
   - To use Template-Driven Forms, import `FormsModule` from the `@angular/forms` package.

```
// app.module.ts
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule],
  // ...
})
export class AppModule { }
```

2. **Validation Rules in Template:**
   - Validation rules are defined directly in the HTML template using attributes like `required`, `minlength`, and `pattern`.
   - Example:

```
<!-- app.component.html -->
<form #myForm="ngForm">
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" ngModel required
minlength="3">

  <div *ngIf="myForm.controls.username.invalid &&
myForm.controls.username.touched">
    <p *ngIf="myForm.controls.username.errors?.required">Username is
required.</p>
    <p *ngIf="myForm.controls.username.errors?.minlength">Username must
be at least 3 characters.</p>
  </div>

  <!-- Other form fields and validation here -->

  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

3. **Validation Messages:**
   - Display validation messages based on the state of form controls using Angular directives like `*ngIf`.
4. **FormsModule for Two-Way Data Binding:**
   - Template-Driven Forms leverage two-way data binding using the `ngModel` directive to bind form controls to component properties.

Example:

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  onSubmit() {
    // Form submission logic
    console.log('Form submitted!');
  }
}
```

```
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" ngModel required
minlength="3">

  <div *ngIf="myForm.controls.username.invalid &&
myForm.controls.username.touched">
    <p *ngIf="myForm.controls.username.errors?.required">Username is required.
</p>
    <p *ngIf="myForm.controls.username.errors?.minlength">Username must be at
least 3 characters.</p>
  </div>

  <!-- Other form fields and validation here -->

  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

In this example, the form has a `username` field with the `required` and `minlength` attributes. Validation messages are displayed based on the state of the form control. The `onSubmit` method is called when the form is submitted, and you can add your logic for form submission in that method.

Template-Driven Forms provide a quick and declarative way to handle forms in Angular applications, making them suitable for simpler scenarios with limited form complexity.

Regular Expressions in Angular

Overview:

Regular expressions (regex or regexp) are powerful sequences of characters that define a search pattern. In Angular, regular expressions are commonly used for tasks like form validation, input filtering, and string manipulation.

Key Concepts:

1. **Definition and Usage:**
    - A regular expression is a sequence of characters that defines a search pattern.
    - It is used to denote regular languages and is employed for matching character combinations in strings.
2. **String Searching:**

- Regular expressions are crucial for string searching algorithms, helping find and manipulate substrings based on defined patterns.

3. **Zero or More Occurrences (`y*`):**
   - In a regular expression, `y*` denotes zero or more occurrences of the character `y`.
   - Example: `y*` can generate `{e, y, yy, yyy, ...}`.

```
const pattern = /y*/;
console.log(pattern.test(''));   // true
console.log(pattern.test('y'));  // true
console.log(pattern.test('yy')); // true
```

4. **One or More Occurrences (`y+`):**
   - In a regular expression, `y+` signifies one or more occurrences of the character `y`.
   - Example: `y+` can generate `{y, yy, yyy, ...}`.

```
const pattern = /y+/;
console.log(pattern.test(''));   // false
console.log(pattern.test('y'));  // true
console.log(pattern.test('yy')); // true
```

Example Use in Angular:

In Angular, regular expressions are often used for form validation. Consider a scenario where you want to validate that a user's input in a form field follows a certain pattern, such as a valid email address:

```
// Example Angular Component
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <form>
      <label for="email">Email:</label>
      <input
        type="text"
        id="email"
        name="email"
        [(ngModel)]="userEmail"
        pattern="[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"
        required
      />
      <div *ngIf="emailInput.invalid && emailInput.touched">
        <p *ngIf="emailInput.errors?.required">Email is required.</p>
```

```
        <p *ngIf="emailInput.errors?.pattern">Invalid email format.</p>
      </div>
      <button type="submit" [disabled]="form.invalid">Submit</button>
    </form>
  `,
})
export class AppComponent {
  userEmail: string = '';

  get emailInput() {
    return this.form.get('email');
  }

  get form() {
    // Logic to return the form instance
  }
}
```

In this example, the `pattern` attribute in the input field uses a regular expression to validate that the entered email follows a specific pattern.

Regular expressions are versatile and widely used in various programming scenarios, including Angular applications, for tasks like input validation, string manipulation, and pattern matching.

Below are some common regular expressions with code snippets for usage in terms of template-driven forms in Angular. These examples include form validations commonly used for input fields like email, password, numeric input, etc.

1. Email Validation:

Regular Expression:

```
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
```

Code Snippet (Angular - Template):

```
<input type="text" name="email" [(ngModel)]="userEmail" pattern="^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$" required />
<div *ngIf="myForm.controls.email.invalid && myForm.controls.email.touched">
  <p *ngIf="myForm.controls.email.errors?.required">Email is required.</p>
  <p *ngIf="myForm.controls.email.errors?.pattern">Invalid email format.</p>
</div>
```

2. Password Strength (At least one uppercase, one lowercase, one digit, and one special character):

Regular Expression:

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$
```

Code Snippet (Angular - Template):

```
<input type="password" name="password" [(ngModel)]="userPassword" pattern="^(?
=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$" required />
<div *ngIf="myForm.controls.password.invalid &&
myForm.controls.password.touched">
  <p *ngIf="myForm.controls.password.errors?.required">Password is required.
</p>
  <p *ngIf="myForm.controls.password.errors?.pattern">Invalid password format.
</p>
</div>
```

3. Numeric Input:

Regular Expression:

```
^[0-9]+$
```

Code Snippet (Angular - Template):

```
<input type="text" name="numericInput" [(ngModel)]="numericInput" pattern="^[0-
9]+$" required />
<div *ngIf="myForm.controls.numericInput.invalid &&
myForm.controls.numericInput.touched">
  <p *ngIf="myForm.controls.numericInput.errors?.required">Numeric input is
required.</p>
  <p *ngIf="myForm.controls.numericInput.errors?.pattern">Invalid numeric input
format.</p>
</div>
```

4. Alphanumeric Input:

Regular Expression:

```
^[a-zA-Z0-9]+$
```

Code Snippet (Angular - Template):

```
<input type="text" name="alphanumericInput" [(ngModel)]="alphanumericInput"
pattern="^[a-zA-Z0-9]+$" required />
<div *ngIf="myForm.controls.alphanumericInput.invalid &&
myForm.controls.alphanumericInput.touched">
  <p *ngIf="myForm.controls.alphanumericInput.errors?.required">Alphanumeric
input is required.</p>
  <p *ngIf="myForm.controls.alphanumericInput.errors?.pattern">Invalid
alphanumeric input format.</p>
</div>
```

These examples demonstrate how to integrate regular expressions into template-driven forms for common validation scenarios. Customize the regular expressions based on your specific requirements.

`Validation` properties

In Angular template-driven forms, validation properties are used to provide feedback to users about the validity of form controls. These properties are part of the `AbstractControl` class, which is the base class for form controls. Here is a list of common validation properties used in template-driven forms:

1. `touched`:

- **Description:** Indicates whether the form control has been blurred (lost focus) by the user.
- **Usage:**

```
<div *ngIf="myForm.controls.myField.touched">
  <!-- Display error messages or styling for validation -->
</div>
```

2. `untouched`:

- **Description:** Indicates whether the form control has not been blurred (lost focus) by the user.
- **Usage:**

```
<div *ngIf="myForm.controls.myField.untouched">
  <!-- Display instructions or styling for untouched fields -->
```

```
</div>
```

3. `pristine`:

- **Description:** Indicates whether the form control value has not been changed.
- **Usage:**

```
<div *ngIf="myForm.controls.myField.pristine">
  <!-- Display instructions or styling for pristine fields -->
</div>
```

4. `dirty`:

- **Description:** Indicates whether the form control value has been changed.
- **Usage:**

```
<div *ngIf="myForm.controls.myField.dirty">
  <!-- Display error messages or styling for validation -->
</div>
```

5. `valid`:

- **Description:** Indicates whether the form control value is valid according to its validation rules.
- **Usage:**

```
<div *ngIf="myForm.controls.myField.valid">
  <!-- Display success or styling for valid fields -->
</div>
```

6. `invalid`:

- **Description:** Indicates whether the form control value is invalid according to its validation rules.
- **Usage:**

```
<div *ngIf="myForm.controls.myField.invalid">
  <!-- Display error messages or styling for invalid fields -->
</div>
```

7. `errors`:

- **Description:** Returns any errors present on the form control.

- **Usage:**

```
<div *ngIf="myForm.controls.myField.errors">
  <!-- Display specific error messages or styling for custom validations
-->
</div>
```

8. `pending`:

- **Description:** Indicates whether an asynchronous validation is pending.
- **Usage:**

```
<div *ngIf="myForm.controls.myField.pending">
  <!-- Display loading indicator or styling for pending validation -->
</div>
```

These properties are typically used in combination with the Angular `ngIf` directive to conditionally display elements based on the state of form controls. Adjust the conditions based on your specific validation requirements and user feedback needs.

Here's a consolidated summary of validation properties commonly used in Angular template-driven forms:

Validation Properties:

1. **touched:**
   - **Description:** Indicates whether the field has been touched (blurred or lost focus) by the user.
   - **Value:** `true` if the field is focused, `false` if the field is not focused.
2. **untouched:**
   - **Description:** Indicates whether the field has not been touched (not blurred or lost focus) by the user.
   - **Value:** `true` if the field is not focused, `false` if the field is focused.
3. **dirty:**
   - **Description:** Indicates whether the field has been modified by the user.
   - **Value:** `true` if the field is modified, `false` if the field is not modified.
4. **pristine:**
   - **Description:** Indicates whether the field has not been modified by the user.
   - **Value:** `true` if the field is not modified, `false` if the field is modified.

5. **valid:**
    - **Description:** Indicates whether the field value is valid.
    - **Value:** `true` if the field value is valid, `false` if the field value is not valid.
6. **invalid:**
    - **Description:** Indicates whether the field value is invalid.
    - **Value:** `true` if the field value is invalid, `false` if the field value is valid.
7. **errors:**
    - **Description:** Returns any errors present on the form control.
    - **Possible Errors:**
        - `required: true/false`
        - `minlength: true/false`
        - `pattern: true/false`
        - `number: true/false`
        - `email: true/false`
        - `url: true/false`

These properties are often used in combination with Angular's ngIf directive to conditionally display elements based on the state of form controls. They provide valuable information for handling form validation and providing feedback to users.

These examples demonstrate how you can use these regular expressions in Angular template-driven forms to enforce specific input patterns. Adjust the patterns based on your specific validation requirements.

1. Digits Only:

- **Regular Expression:** `^[0-9]*$`
- **Description:** Validates if the input contains only digits (0-9).

2. Alphabets Only:

- **Regular Expression:** `^[a-zA-Z]*$`
- **Description:** Validates if the input contains only alphabets (both lowercase and uppercase).

3. Mobile Number (assuming Indian mobile numbers starting with 8 or 9 and having 10 digits):

- **Regular Expression:** `^[89]\d{9}$`
- **Description:** Validates if the input is a valid Indian mobile number starting with 8 or 9 and having a total of 10 digits.

These regular expressions can be used in Angular template-driven forms for pattern-based validation. For example:

```
<!-- Digits Only -->
<input type="text" name="digitsInput" [(ngModel)]="digitsInput" pattern="^[0-
9]*$" required />

<!-- Alphabets Only -->
<input type="text" name="alphabetsInput" [(ngModel)]="alphabetsInput"
pattern="^[a-zA-Z]*$" required />

<!-- Mobile Number -->
<input type="text" name="mobileNumberInput" [(ngModel)]="mobileNumberInput"
pattern="^[89]\d{9}$" required />
```

Creating a `template-driven` form in Angular involves several steps. Below is a detailed guide on how to create a simple template-driven form:

Step 1: Set Up Your Angular Project

Make sure you have Node.js and npm installed. If not, download and install them from Node.js website.

Open a terminal and run the following commands:

```
# Install the Angular CLI globally
npm install -g @angular/cli

# Create a new Angular project
ng new your-project-name

# Navigate to your project folder
cd your-project-name
```

Step 2: Create a Component for Your Form

Create a new component that will host your form:

```
ng generate component your-form
```

Step 3: Design Your Form in the Component's HTML File

Edit the `your-form.component.html` file to design your form. For example:

```
<form #myForm="ngForm" (ngSubmit)="submitForm()">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" [(ngModel)]="formData.name"
required>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" [(ngModel)]="formData.email"
required>

  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

Step 4: Set Up Form Validation

In the component's TypeScript file (`your-form.component.ts`), define the form data structure and any necessary validation:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-your-form',
  templateUrl: './your-form.component.html',
  styleUrls: ['./your-form.component.css']
})
export class YourFormComponent {
  formData = {
    name: '',
    email: ''
  };

  submitForm() {
    // Handle form submission logic here
    console.log('Form submitted:', this.formData);
  }
}
```

Step 5: Import FormsModule

In the `app.module.ts` file, import `FormsModule` from `@angular/forms` and add it to the `imports` array:

```
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    // ... other components
    YourFormComponent
  ],
```

```
  imports: [
    BrowserModule,
    FormsModule // Add this line
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 6: Run Your Angular App

In the terminal, run:

```
ng serve
```

Open your browser and navigate to `http://localhost:4200/` to see your form.

Step 7: Test and Debug

Fill out your form, click the submit button, and check the console for form data.

This is a basic example to get you started. You can enhance your form by adding more fields, validation rules, and handling form submission based on your application's requirements.