Angular Forms

Handling user input with forms is the aim of many common web applications. Applications use forms to enable users to log in, to update a profile, to enter sensitive information, and to perform many other data-entry tasks.
Angular provides two different approaches to handling user input through forms: reactive and template-driven.
Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.
Reactive and template-driven forms process and manage form data differently. Each offers different advantages.

In general:

Reactive forms are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

Template-driven forms are useful for adding a simple form to an app, such as an email list signup form. They're easy to add to an app, but they don't scale as well as reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, use template-driven forms.

There are some key differences between Reactive and Template Driven forms as

	REACTIVE	TEMPLATE-DRIVEN
Setup (form model)	More explicit, created in component class	Less explicit, created by directives
Data model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Form validation	Functions	Directives
Mutability	Immutable	Mutable
Scalability	Low-level API access	Abstraction on top of APIs

Template-driven forms

- Use template-driven forms when developing static forms.
- Static means the structure and logic of a form is fix.
- E.g. the number of form fields does not vary, form validation rules are the same for different user roles, etc.
- Examples are login forms, reset password forms, forms to enter and edit address data, order data and similar fix data structures.
- In the template-driven approach the form structure and logic is mainly implemented in HTML.
- Based on this a representation of the form in TypeScript is generated automatically.

Template-driven forms support…
- One-way and two-way data-binding.
- Forms to enter new data and to edit existing data (from a backend service) can be developed.
- Creation of nested form fields, e.g. a form containing a user model consisting of user name, email address and postal address — consisting of street, city, zip code.
- Field-spanning validation, e.g. validate entire user model instead of checking each field individually.
- Synchronous and asynchronous validation, e.g. check via remote server whether email address exists.
- Checking form state, e.g. warn when leaving the form with unsaved changes.

Pros
- Form structure and logic is located in one place (HTML)
- New form functionality without / with minimal TypeScript coding
- Makes use of pure web standards, e.g. HTML required attribute for input validation
- Rather easy to understand

Cons
- Beyond static form structure and logic template-driven forms provide only limited capabilities to implement dynamic aspects like variable number of fields, repetitive fields, etc.

Reactive-forms

- Use the reactive forms approach in case the form shall support dynamic data structures and logic.
- Examples are dynamic survey forms, forms to add/delete 0..n tags or phone numbers, forms providing different validation for different user roles, etc.
- The structure and logic of reactive forms is mainly implemented in TypeScript. Corresponding HTML artifacts only refer to the form controls defined in TypeScript.
- At highest expansion stage a reactive form can be entirely generated at runtime based on a data structure.

Pros
- Form definition and logic is mainly implemented in TypeScript.
- Form fields are created programmatically by using FormGroup or FormBuilder class. HTML form tags only reference TypeScript-based form controls.
- Allows programmatic and full control of form value updates and form validation
- Supports creation of forms with dynamic structure at runtime
- Supports implementation of custom form validation

Cons
- Requires more coding, especially in TypeScript
- Is more difficult to understand and to maintain