

# Angular Basics

Erste Einblicke in Angular



# Agenda



Fundamentals



Add IQ



Forms & Validierung

# Agenda

[] Fundamentals

Typescript  
Projektstruktur  
Modules  
Services  
Directives  
Components



Add IQ



Forms & Validierung

# Agenda

[] Add Logic



Fundamentals

Dependency Injection  
Routing  
State-Management  
RxJs  
Observables & Subjects  
Signals & Effects



Forms & Validierung



# Agenda

[📄] UI - Forms & Validierung



Fundamentals



Add IQ

Reactive Forms  
Template Driven Forms  
Pipes  
Validierung

# [] Fundamentals

# Fundamentals

## Javascript -> Typescript

- „typed javascript“
- Wird als javascript kompiliert
- Benefit, man hat Interfaces, Klassen, union types, etc. und Fehler werden beim entwickeln erkannt
- Man kann javascript code in typescript verwenden
- Dateiendung .ts



# Fundamentals

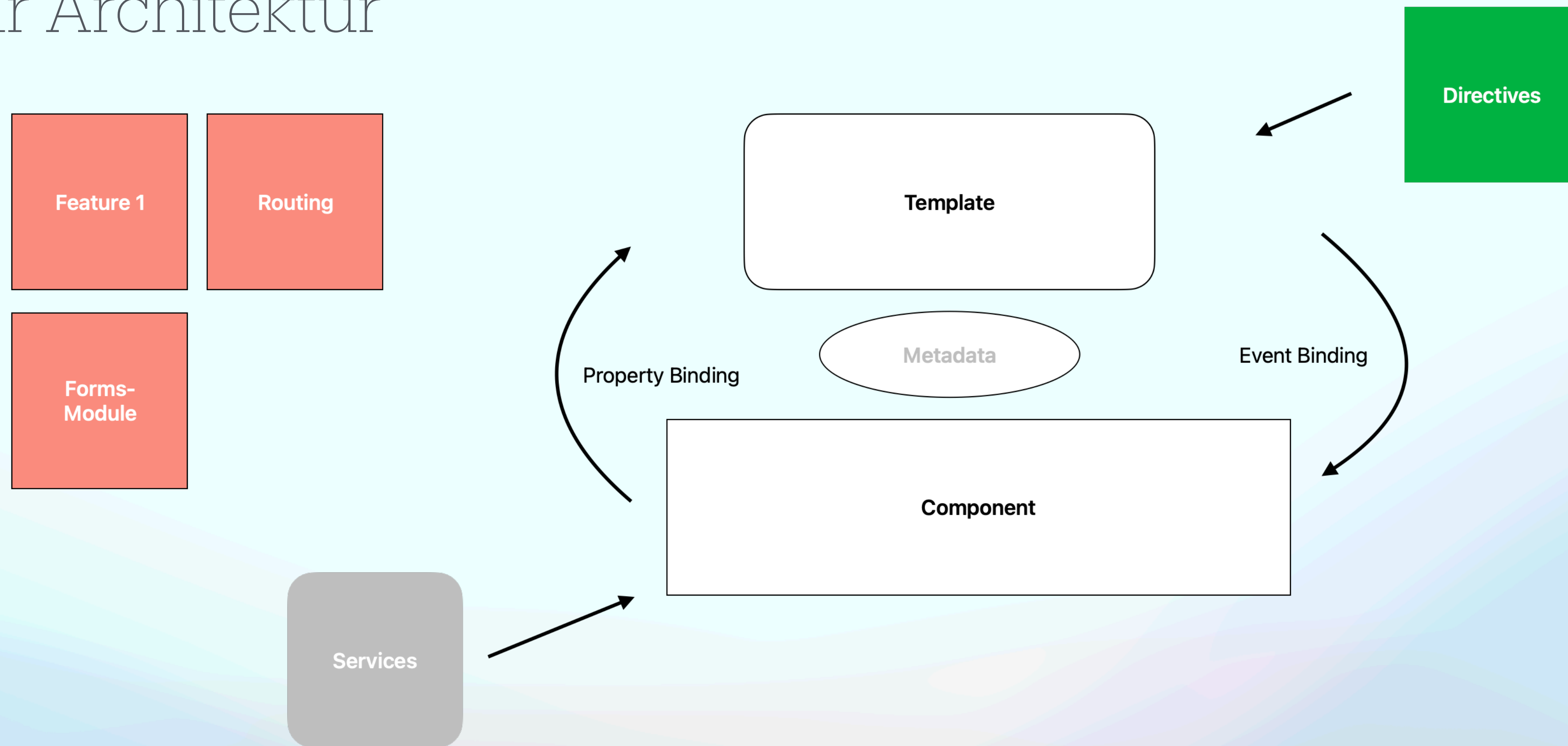
Project structure



Angular

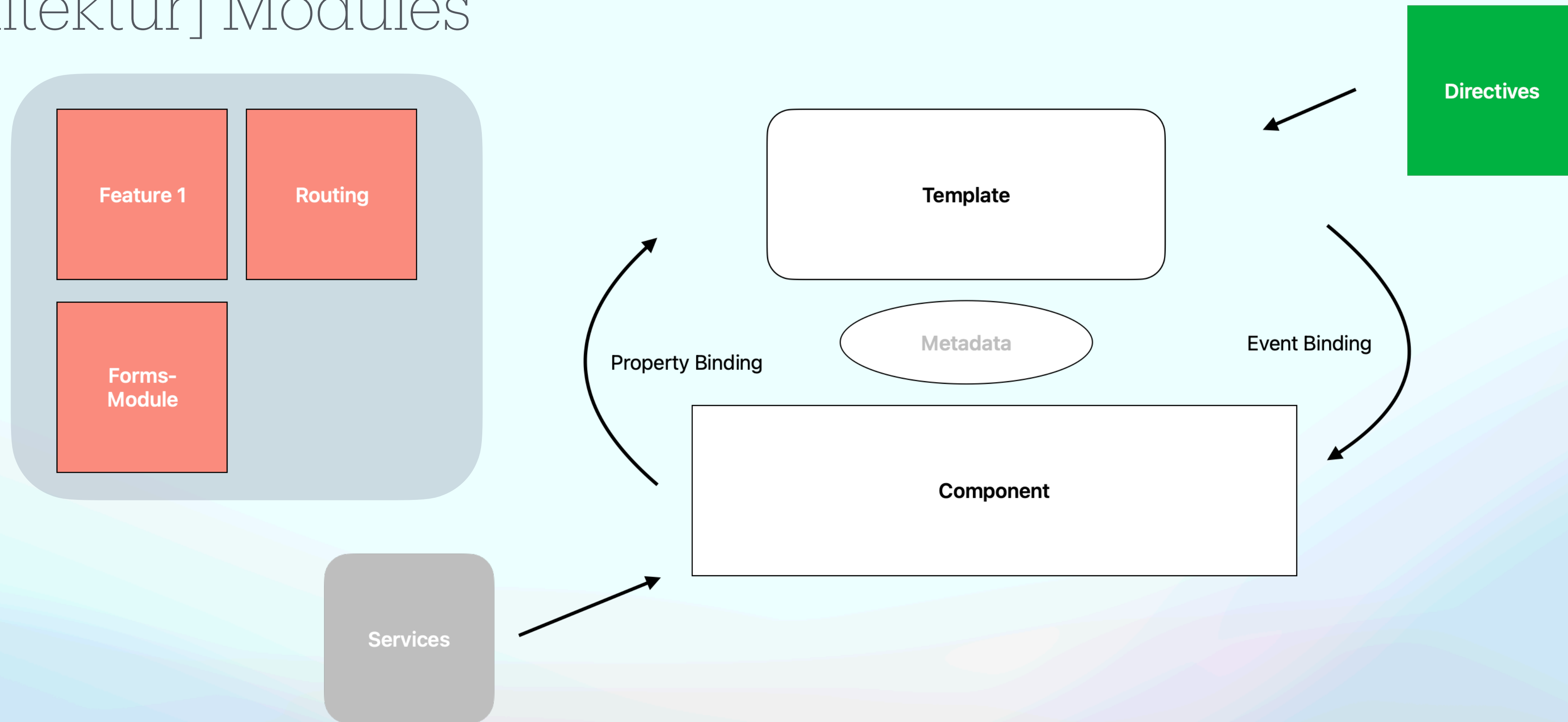
# Fundamentals

## Angular Architektur



# Fundamentals

## [Architektur] Modules



# Fundamentals

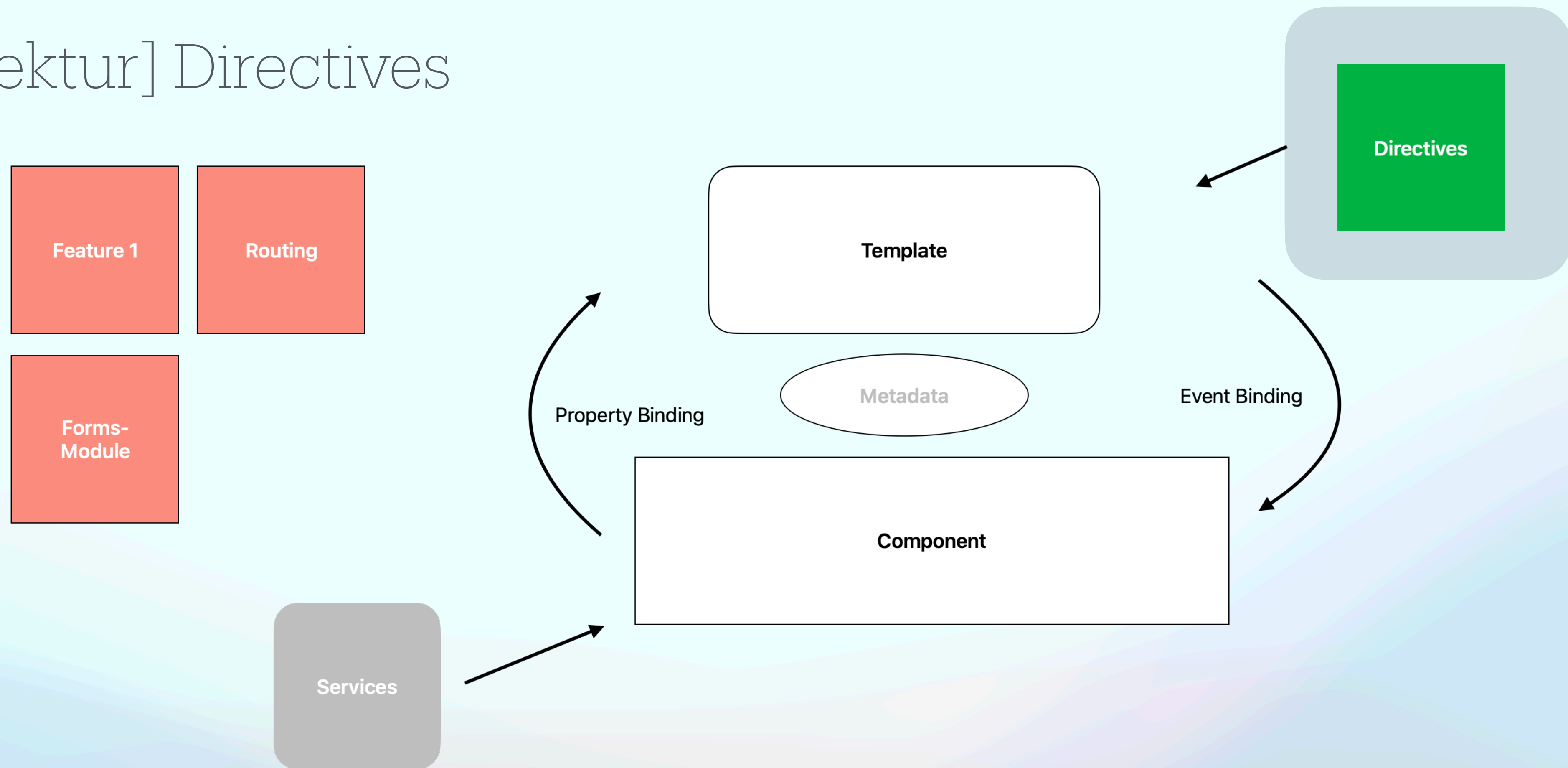
## Modules

- @NgModule - Decorator
- Codeblöcke um Struktur zu erstellen
- RootModule & Feature-Module
- Feature Modules mit Lazy Loading um Geschwindigkeit zu erhöhen
- „deprectaed“

ng generate module <name>

# Fundamentals

## [Architektur] Directives





# Logic

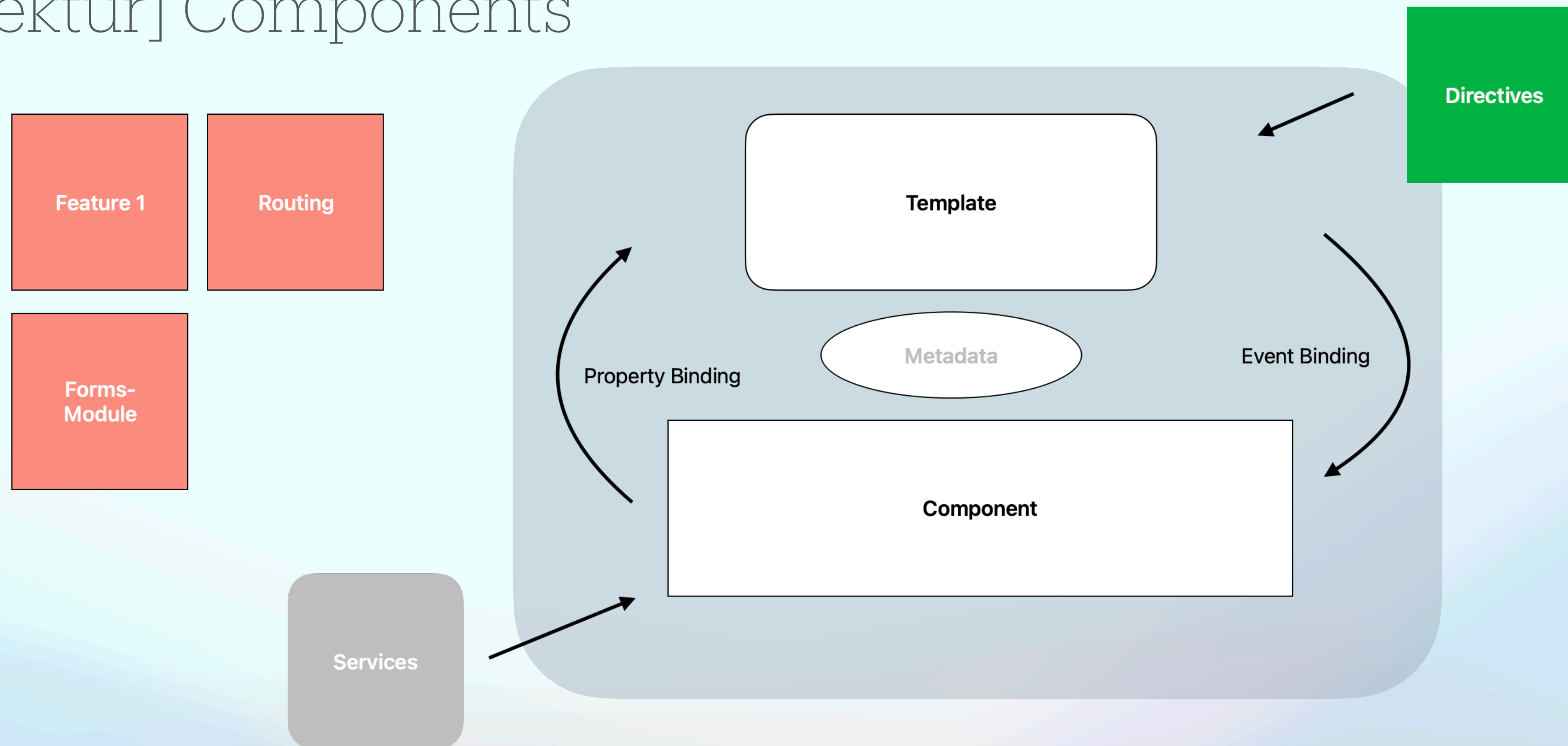
## Directives

- @Directive - Decorator
- Fügen zusätzliche Funktionalität zur Applikation hinzu.
  - Components
  - Attribute Directives: Ändern die Darstellung und auch das Verhalten eines FormElements ( ngClass, ngStyle, ngModel )
  - Structural Directives: Ändern das DOM ( ngIf, ngFor, ngSwitch )
    - ngFor
      - Index: let I=index setzen einer variable mit dem aktuellen index bei jeder iteration
      - trackBy: man kann server calls dadurch unterbinden indem man zB die id an das element bindet
  - Custom directives

ng generate service LoginState

# Fundamentals

## [Architektur] Components



# Fundamentals

## Components

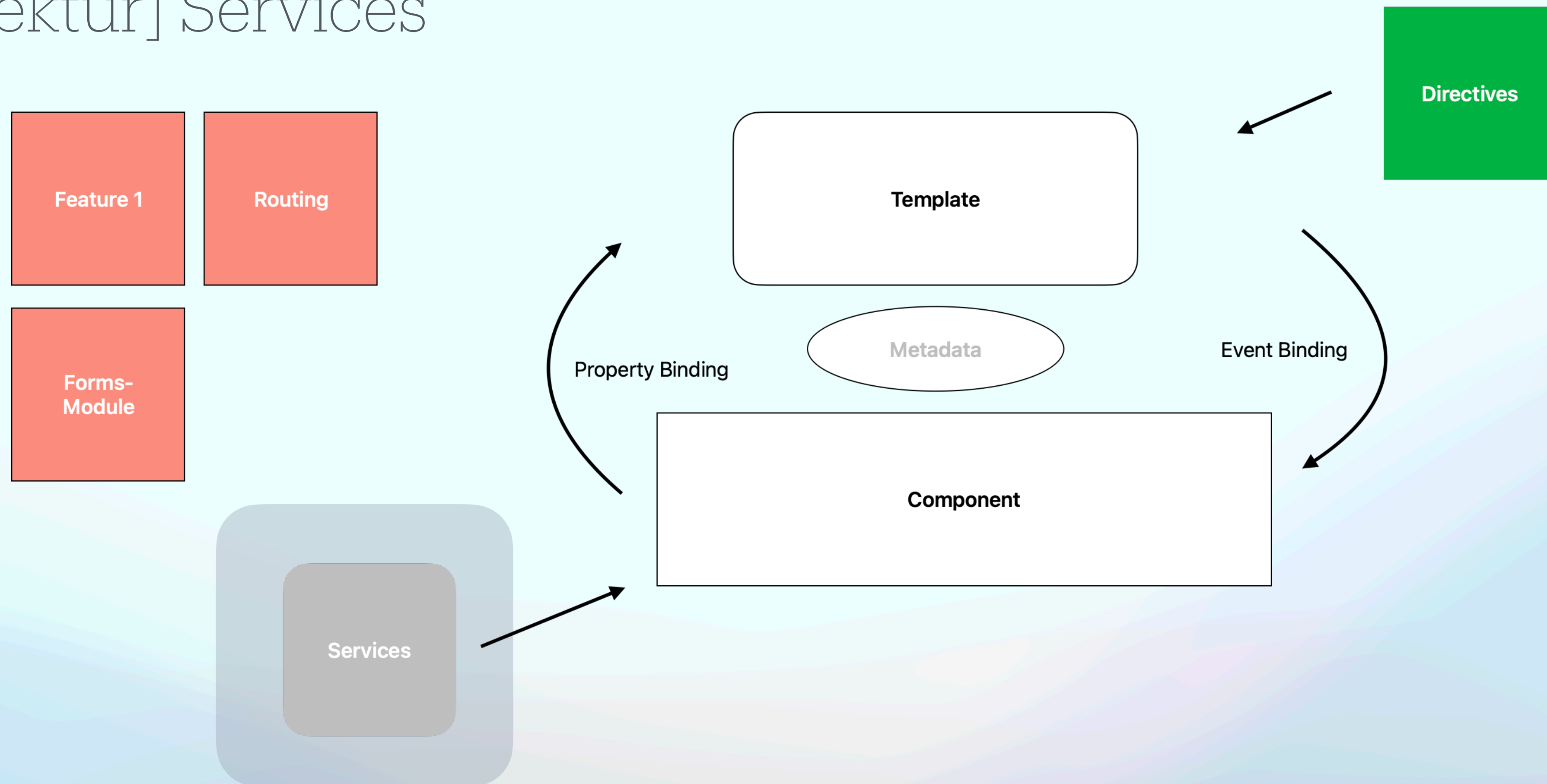
- @Component - Decorator
  - selector, templateUrl, styleUrls, standalone, ...
- Grundstein einer jeder Applikation
- Definiert durch html-File, css/scss-File und ts-File
- Property Binding & Event Binding
- —standalone

ng generate component <name>

ng generate component <name> —standalone

# Fundamentals

[Architektur] Services



# Fundamentals

## Services

- zum speichern von Daten von Rest Calls und oder Logik auszulagern
- durch Dependency Injection verschiedene lifecycles
  - Component, Module oder Applikation-Ebene

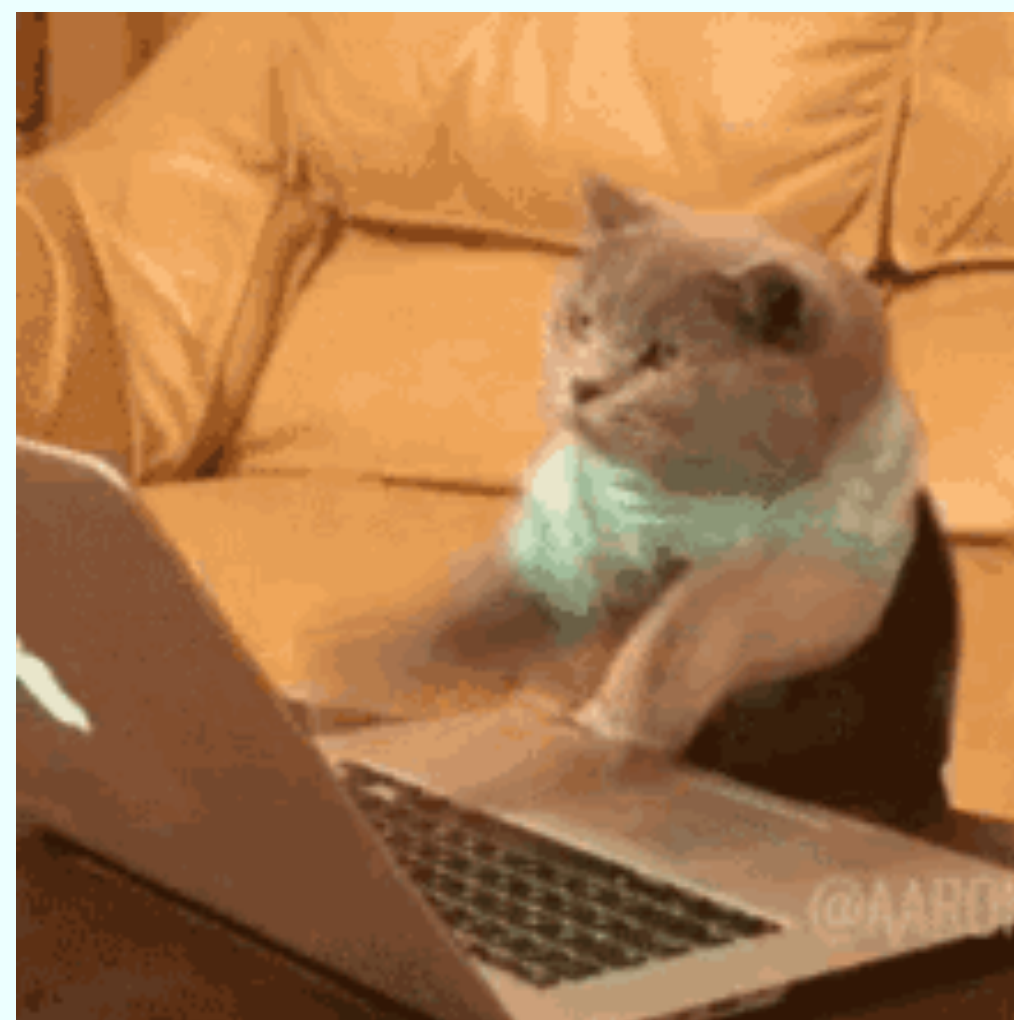
ng generate service LoginState



# Fundamentals

## Pipes

- Funktions um Daten zu manipulieren
- Built-in pipes
  - DatePipe, AsyncPipe, JsonPipe, ...
- Custom-Pipes: für eigene Anforderungen mittels Inpu variable erstellt



[] Add IQ

# Logic

## Dependency Injection

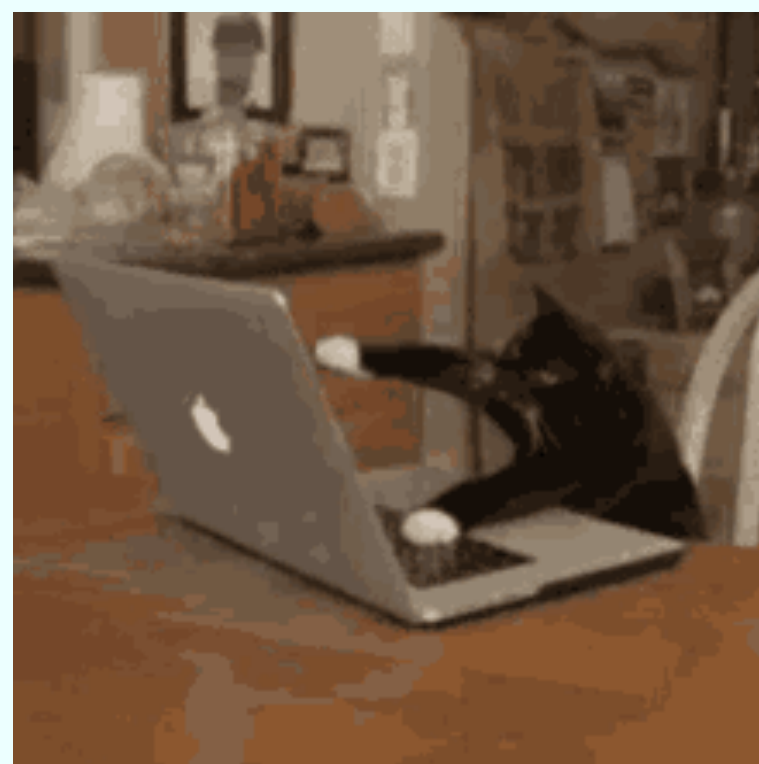
- Integration und bereitstellen von Components, Services, Directives, Pipes, ...
- @Injectable() - Komponenten Level
- @Injectable({providedIn: root}) - Root Level
- Injecten via Component | Module | bootstrap

# Logic

## Routing

- Navigieren zwischen Seiten oder ersetzen von Components in einer Angular Applikation
- „First come first serve“ - Start nach Ende
- Root routing und child-routing
- `provideRoutes( <routes> )`





# Logic

## State Management

- Unterscheidet zwischen
  - Component State
  - Application State
- State in Services wird in Variablen, Observables oder Signals gespeichert.

# Logic

## RxJS

- Reactive Extensions for Javascript
- Library für das asynchrone programmieren und halten von variablen über bestimmte Zeit bzw reaktiv
- RxJS ermöglicht auch das Filtern, Mappen, sortieren, (manipulieren) etc. von Daten

# Logic

## Observables

- „unicast“ - jeder subscriber hat seinen Stream
- "value over time" -> Consumer muss sich auf ein Observable subscriben um das Object zu erhalten
- „push-based“ -> es muss ein Update gemacht und subscribed sein um ein update zu erhalten
- Subscription: hilft beim recyceln
- Einfaches ErrorHandling mittels .pipe()
- Naming convention „schreibt vor“ an die variable als suffix ein \$ anzuhängen zB. username\$, password\$

# Logic

## Observables x Subjects

- Subject ist ein Type von Observables, dass senden ( `.next()` ) und empfangen ( `.subscribe()` ) kann
- BehaviourSubject: ist auch Observable Type der aber genau 1 value halten kann
- „multicast“ - jeder Subscriber/Empfänger bekommt das selbe und neuste Objekt
- Man kann auch observables ( „unicast“ ) zu einem Subject („multicast“) wandeln
  - `shareReplay({ bufferSize: 1, refCount: false })`
    - `bufferSize`: anz. An gesendeten Objekte (beginnend beim neuesten)
    - `refCount`: steuert die wiederverwendbarkeit der Ausführung



# Logic

## Signals

- „pull-based“ - können jederzeit abgerufen werden, also reaktiv
- writable signal ( type: WriteableSignal<T> )




```
var test = signal(„Hallo“);  
this.signal.update(value => value + „Max“)
```

- computed signal
  - readonly und abhängig von einem writable signal

```
var count:writeablesignal<number> ) = signal(0);  
count readable:signal<number> = computed() => count() +2)
```

- „lazy evaluated“ -> beim ausführen gecached und beim 2x nicht erneut ausgeführt, außer writable signal wird upgedated

# Logic

	 Values	 Signals	 Observables
reference	value	container of value	container of values
time	(no concept)	(no concept)	value over time
access	direct	getter -> pull	callback -> push

# Logic

## Effects

- Anwendungsbereiche: Logging, localStorage updaten, render Prozesse um die Logik zu vereinfachen, ... etc.
- Können definiert werden mittels:
  - injection context -> executable code (constructor service / component)
  - Zuweisung einer variable
  - injector der in options übergeben wird (Injector erstellt neue Objekt Instanz)
- Asynchron und entkoppeln nicht vom DOM ( anders wie das computed signal )
- Sind an den Component, Service lifecycle gebunden -> auto recycled oder manuell ( .destroy )
- Don't's: updaten von signals -> können infinite cycles entstehen

```
effect(() => {  
  console.log(`The current count is: ${count()}`);  
});
```







# ] Forms & Validierung

# Forms & Validierung

## Forms

- Was, Wann ?
  - Reactive Forms: große Applikationen mit vielen Inputs
  - Template Driven Forms: simple Anwendung mit wenig inputs
-



# Forms & Validierung

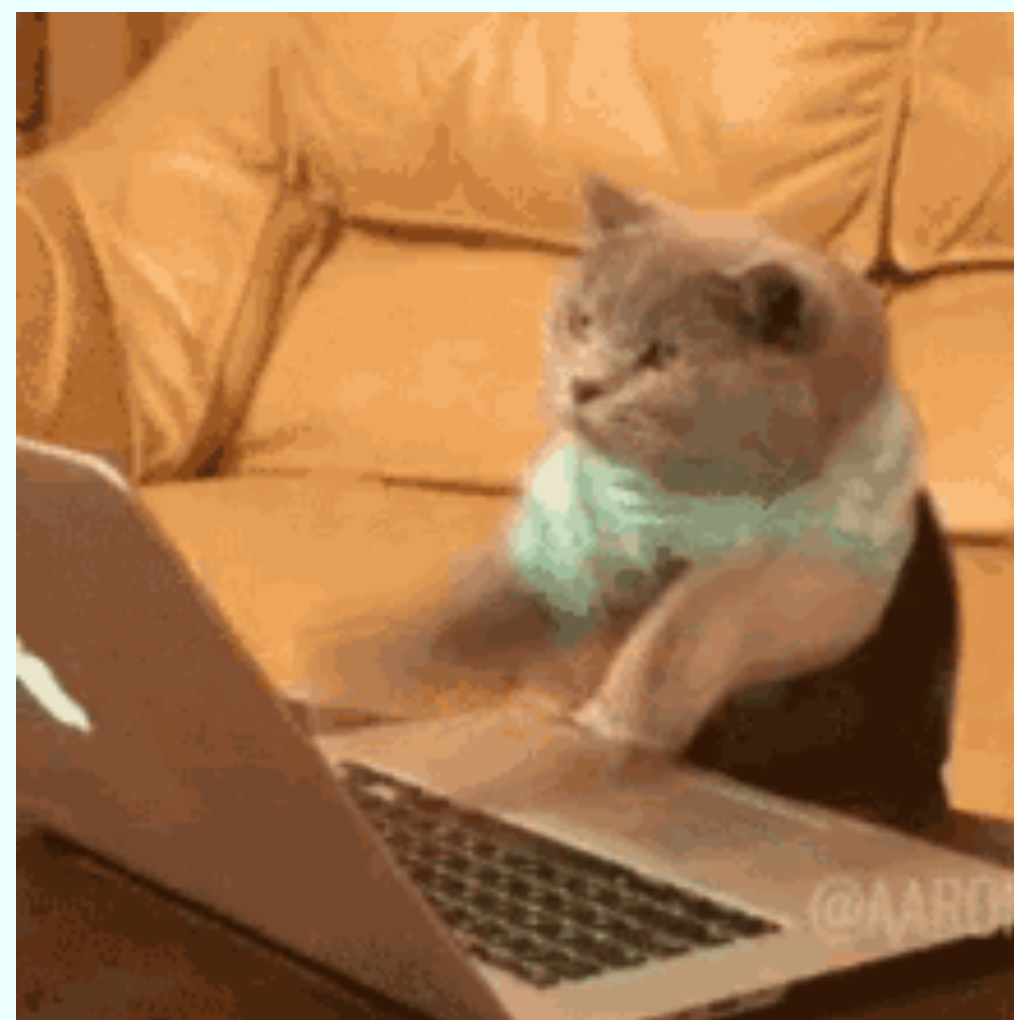
## Reactive Forms

- Direkter Zugriff auf das Steuerelement via FormControl
- Elemente:
  - FormControl: bindet Steuerelement an FormElement und Trakt Validieren und Status
  - FormGroup: Gruppe von FormControls
  - FormArray: trackt FormControls in einem Array
- FormGroup / Typed FormGroup oder FormBuilder Service
- Validieren via Built-in Validators oder Custom Validators (Directives)
- Strukturiertes Datenmodel - mittels get() jederzeit Zugriff auf FormControl
- Gut wiederverwendbar und testbar

# Forms

## Template Driven Forms

- Mit einer Directive an das Formelement gebunden ( [ngModel] ) -> two way databinding
- Unstrukturiertes Datenmodell
- Validieren via Built-in Validators oder Custom Validators (Directive + Wrapper)
- Mit <form #<name>=„ngForm“> kann man auf form Events zugreifen
- Error Handling und Validieren handelt man über FormStatus



# Links

- <https://angular.dev/>
- <https://dev.to/t/angular>
- <https://tailwindcss.com/docs/>
- <https://daisyui.com/docs/themes/>





Vielen Dank  
für die Aufmerksamkeit

