



Enhance Web Development

Angular 2



Les services

- Services
- @Injectable
- Injecting Services

Services

- Un service est une classe ES6 (comme pour les composants)
- On écrit l'API de notre service en définissant des méthodes et propriétés public sur cette classe

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

export class StateService {
  private _message = 'Hello Message';

  getMessage(): string {
    return this._message;
  }

  setMessage(newMessage: string): void {
    this._message = newMessage;
  }
}
```



Service

@Injectable

- On décore notre classe avec **@Injectable()** pour la rendre éligible à un injection
- L'injecteur va lever une erreur **NoAnnotationError** quand il va essayer d'instancier une classe sans **@Injectable()**

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

@Injectable()
export class StateService {
    private _message = 'Hello Message';

    getMessage(): string {
        return this._message;
    }

    setMessage(newMessage: string): void {
        this._message = newMessage;
    }
}
```



@Injectable()

Injecter un Service

- Importer notre service et l'injecter au travers des paramètre du constructeur
- On peut injecter un Service dans un Service
- Il ne peut y avoir qu'une seule instance d'un Service dans un **Injector** mais il peut y avoir plusieurs **Injector** dans une application. Un Service injecté dans un Injector est disponible pour tous les enfants.

```
import {Component, OnInit} from '@angular/core';
import {StateService} from '../common/state.service';

@Component({
  selector: 'home',
  template: require('./home.component.html')
})
export class HomeComponent implements OnInit {
  title: string = 'Home Page';
  body: string = 'This is the about home body';
  message: string;

  constructor(private _stateService: StateService) { }

  ngOnInit() {
    this.message = this._stateService.getMessage();
  }

  updateMessage(m: string): void {
    this._stateService.setMessage(m);
  }
}
```

Injecter un Service

Services communs

- Les Services qui ne sont pas spécifique à un composant, peuvent être injecter directement dans le composant racine de l'application
- Par exemple les services de routing sont injecter pour toute l'application

```
import 'core-js';
import 'zone.js/dist/zone';

import {bootstrap} from '@angular/platform-browser-dynamic';
import {ROUTER_PROVIDERS} from '@angular/router';
import {AppComponent} from './app.component';

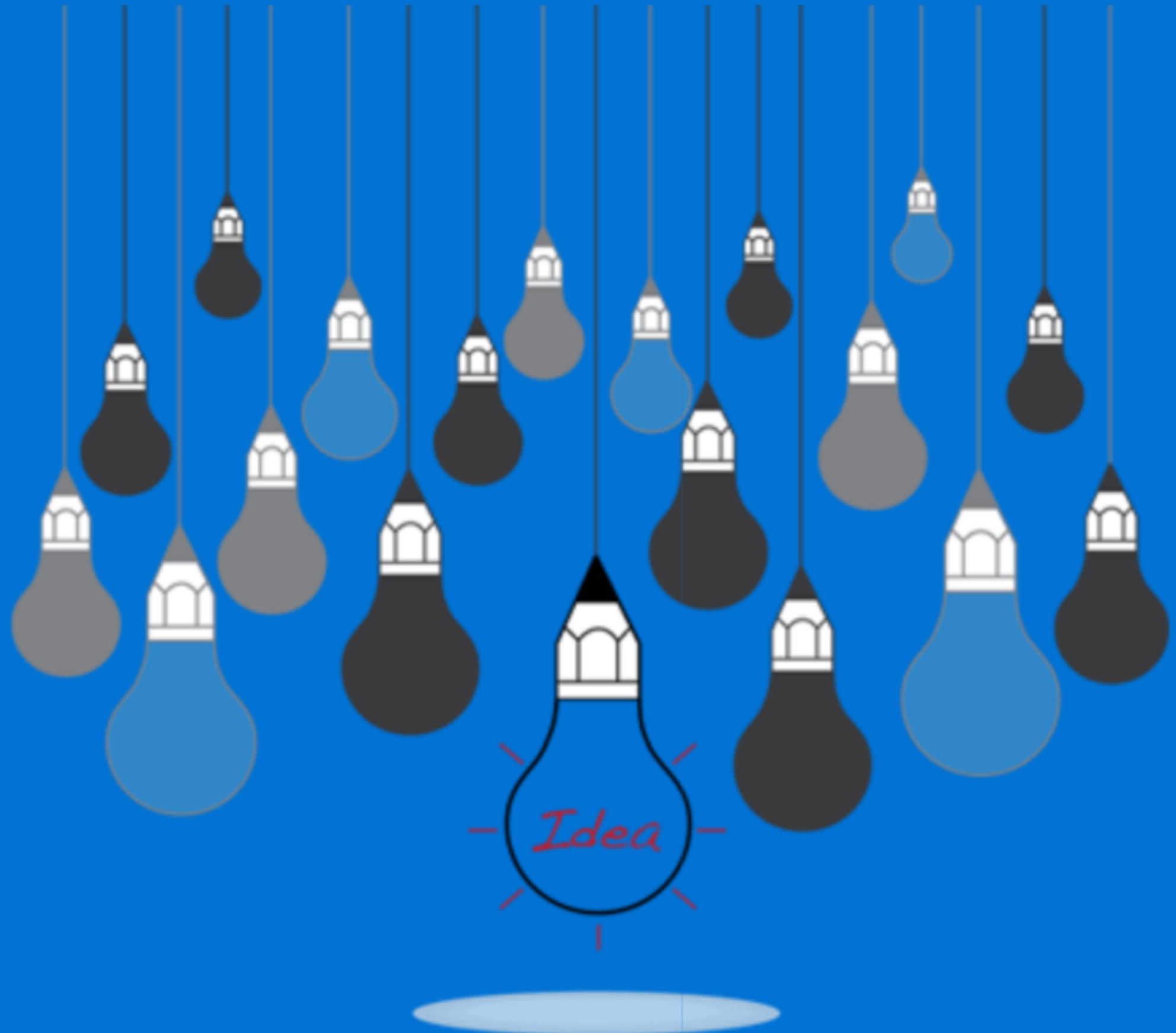
bootstrap(AppComponent, [
  ROUTER_PROVIDERS
]);
```



Services communs

Demo time !





Défi

- Créer un service **widgets** qui s'occupe de gérer une collection de **widgets**
- Décorer avec **Injectable()**
- Injecter ce service dans le composant **widgets** et récupérer la collection dans le composant
- BONUS : Créer un second service **pricing** qui sera injecté dans le service **widgets** pour calculer un pourcentage du prix

Architecture composant

- Système d'architecture composant
- Contrat @Input et @Output
- @Input
- @Output
- Composant intelligent ou stupide
- Encapsulation de vue

Bref historique Angular

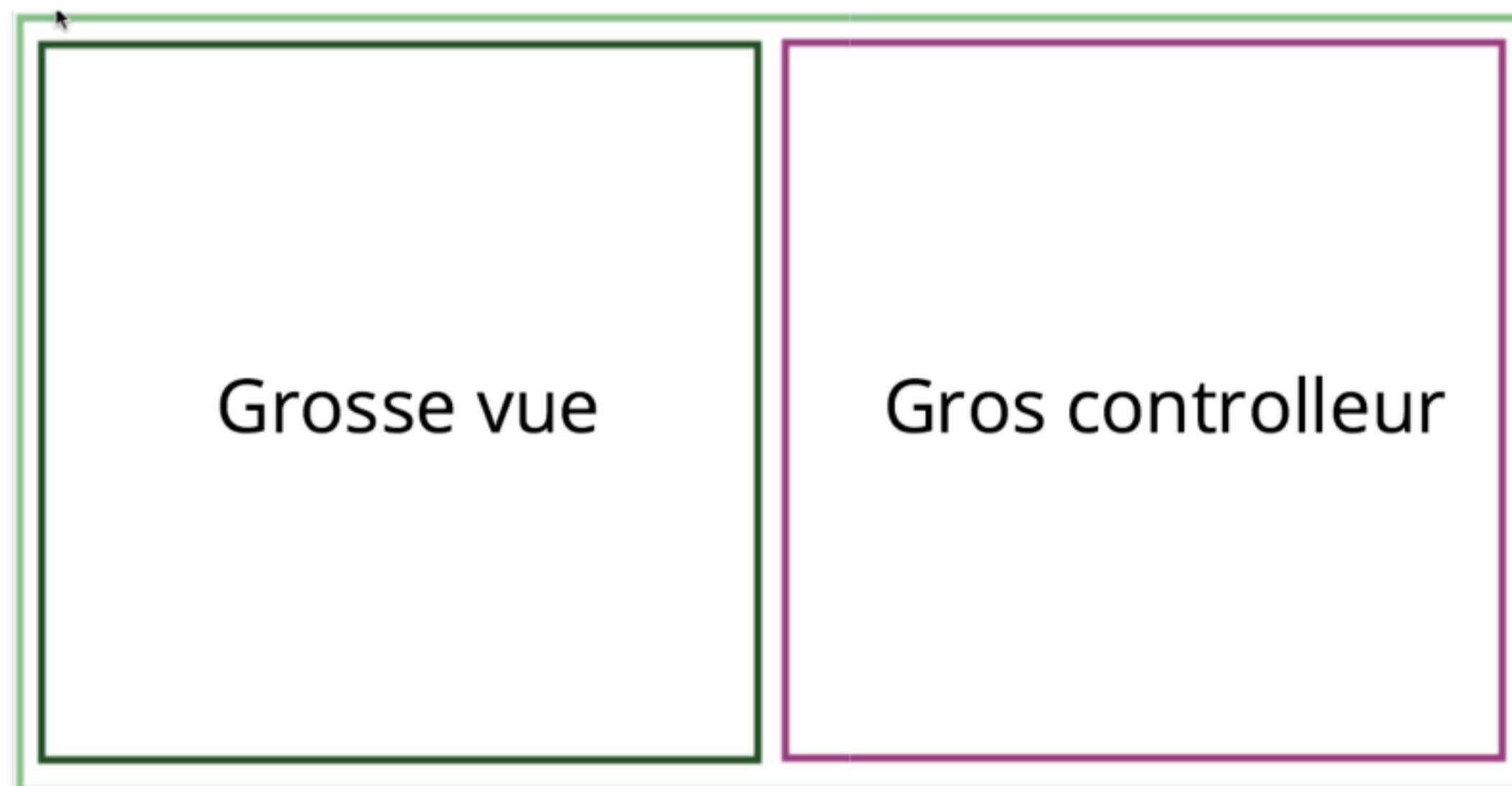




Petite app = petite vue + petit controller

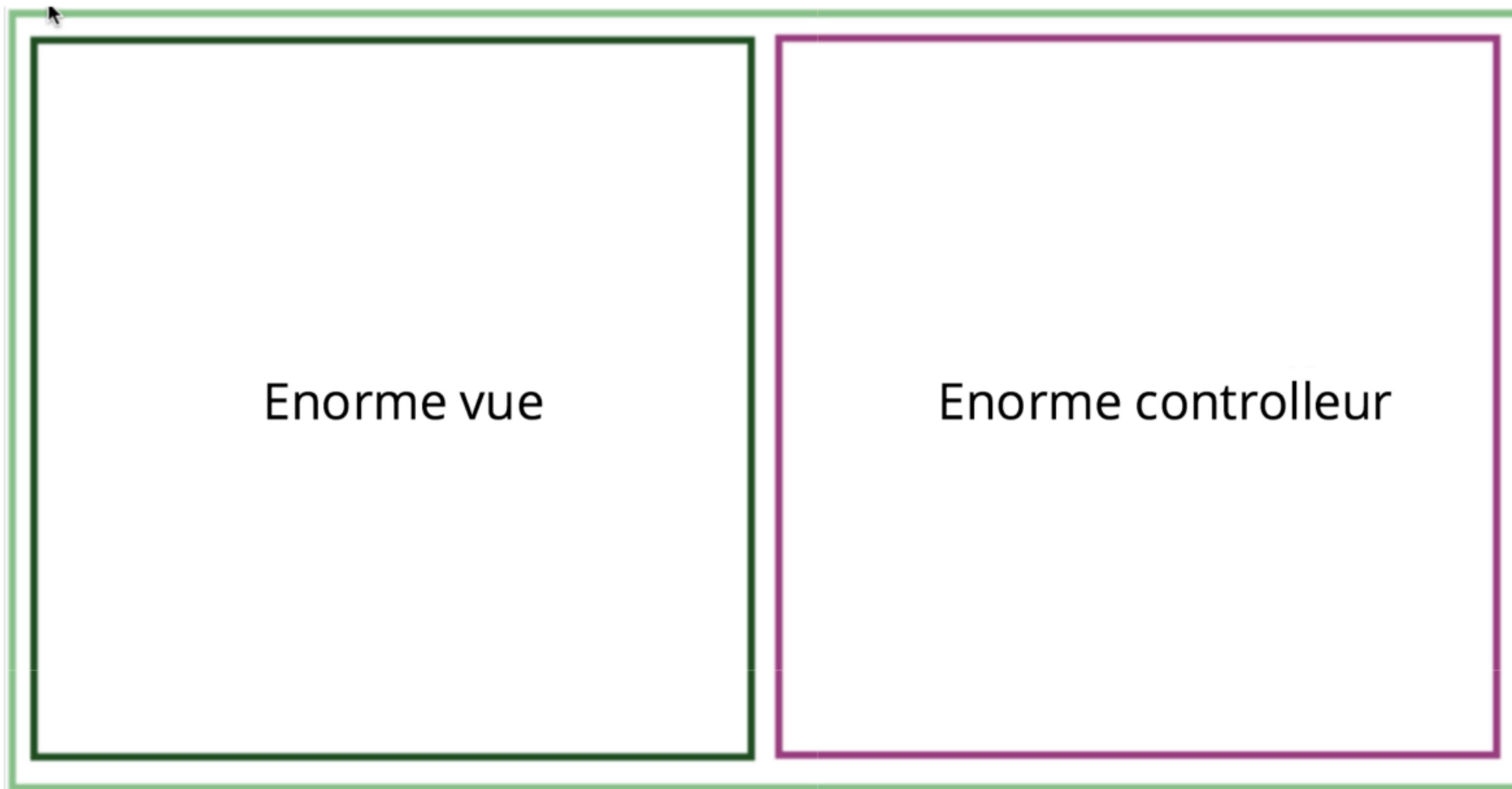
Salut Angular 1.x

Grosse application



Mmmh ...

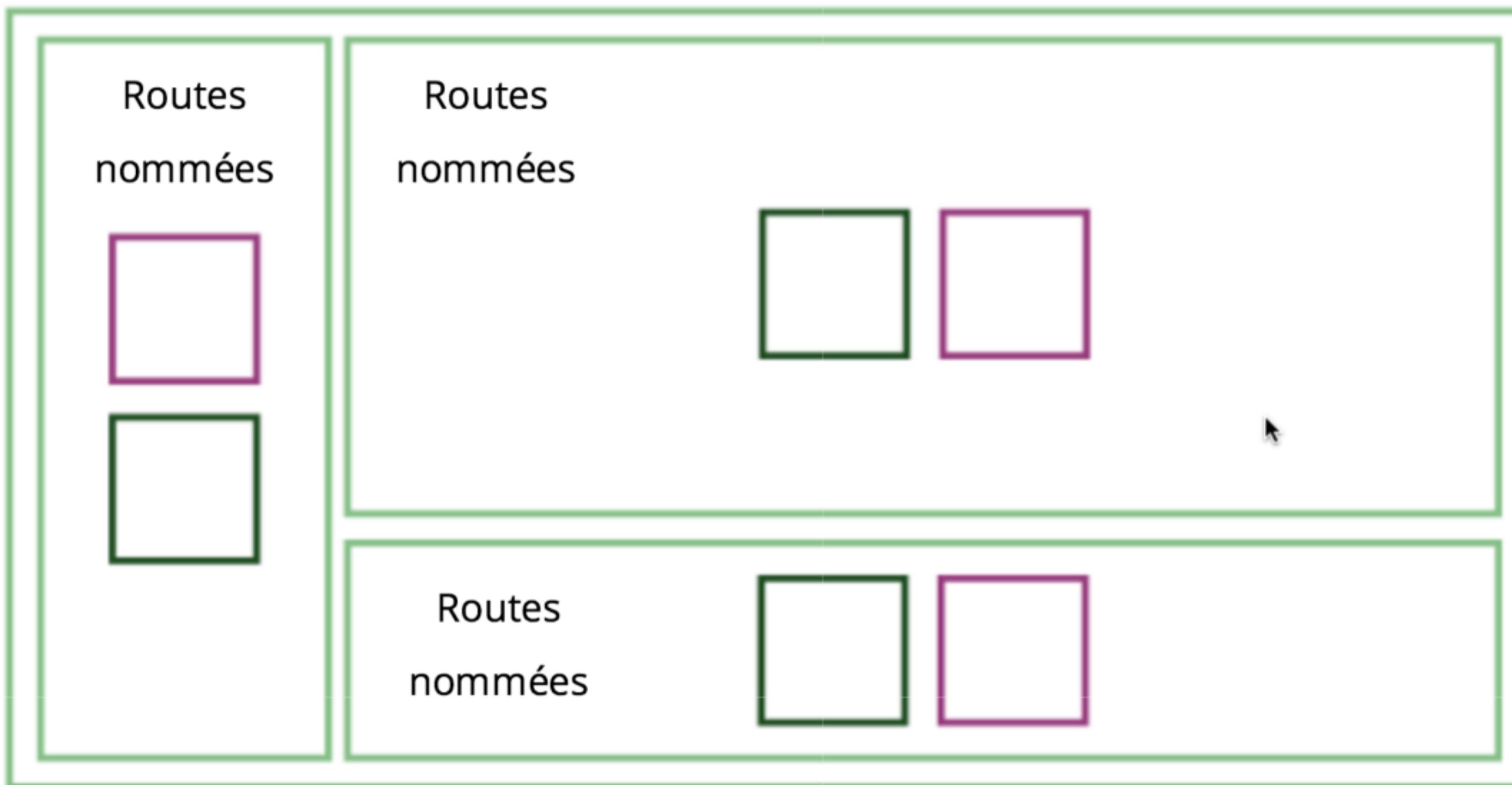
Enorme application



What?

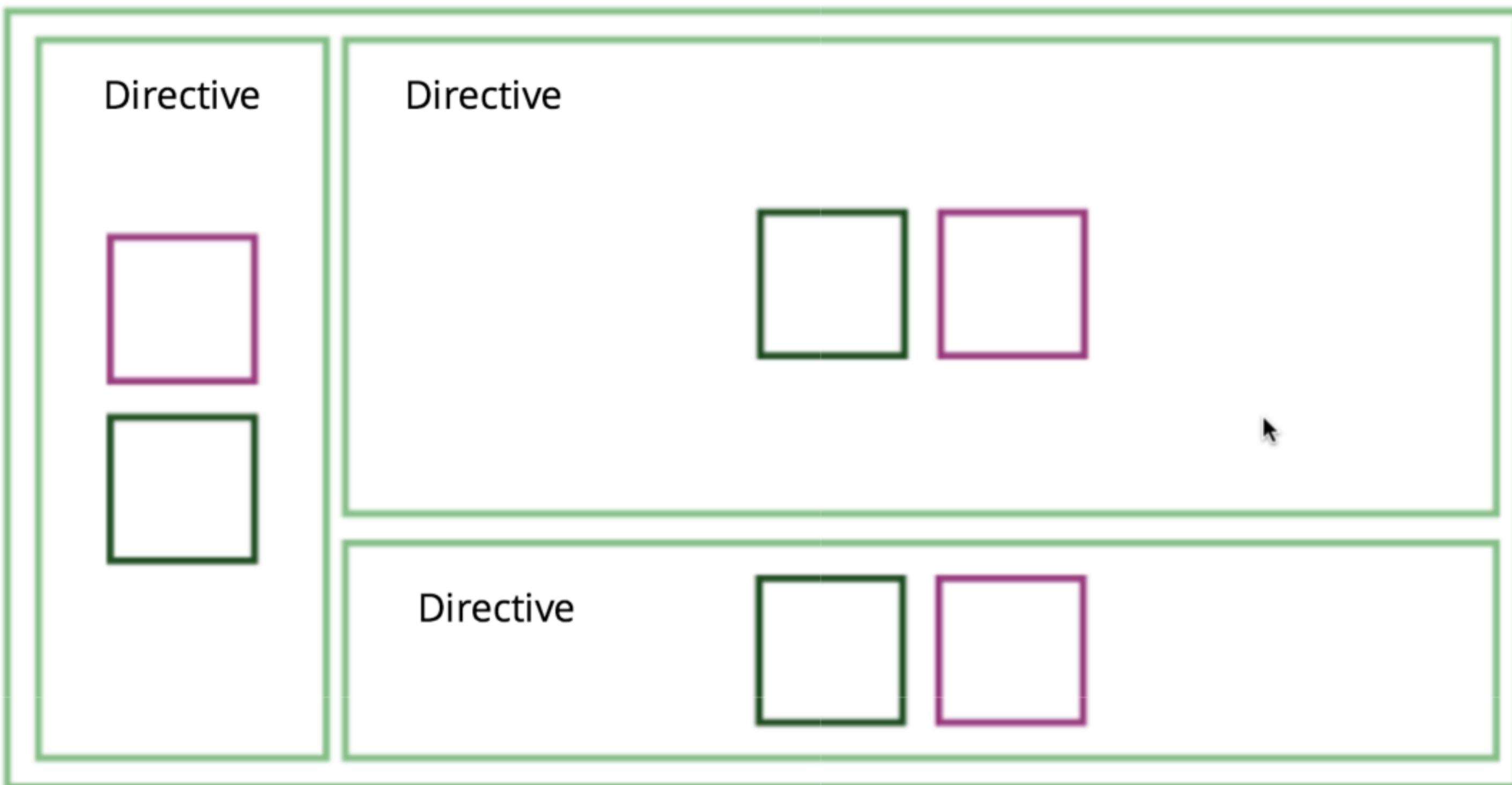
Deux solides approches

Large app 1.x



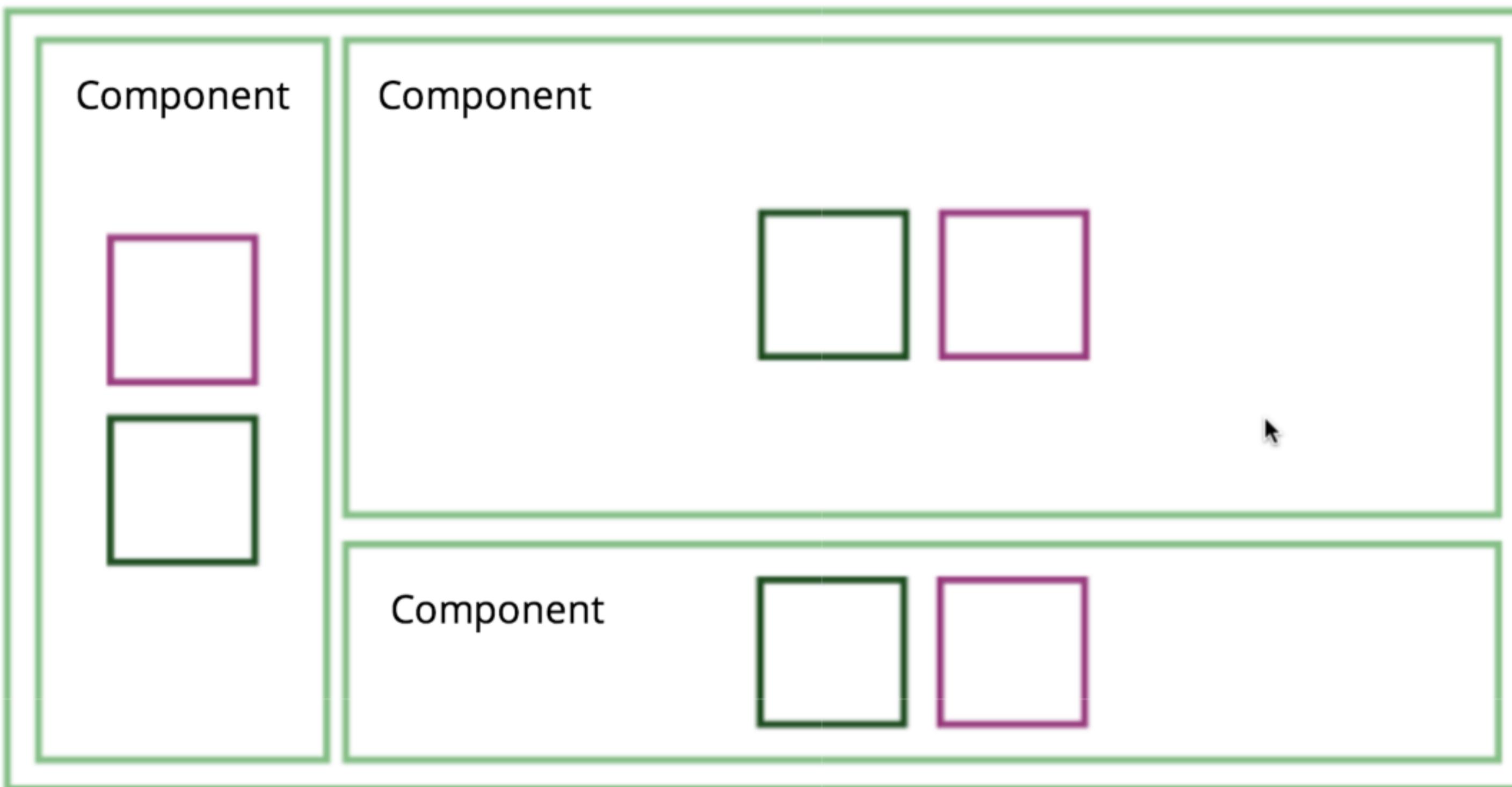
Les routes nommées

Large app 1.x



Les directives

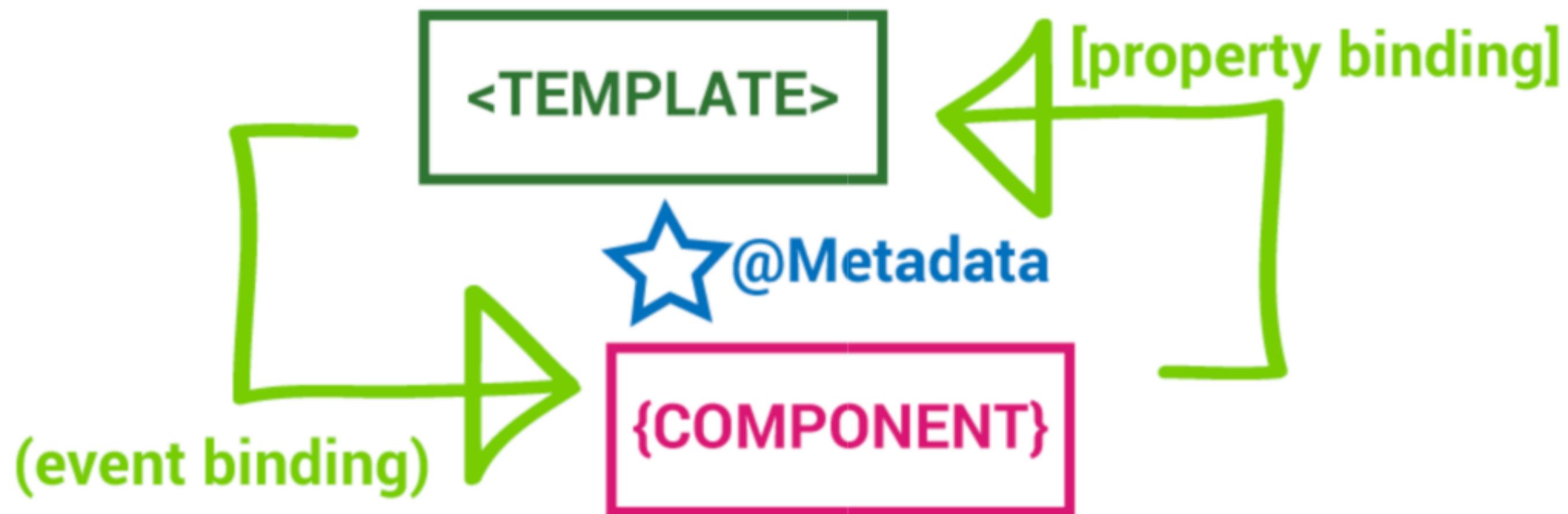
Toutes les app ng2



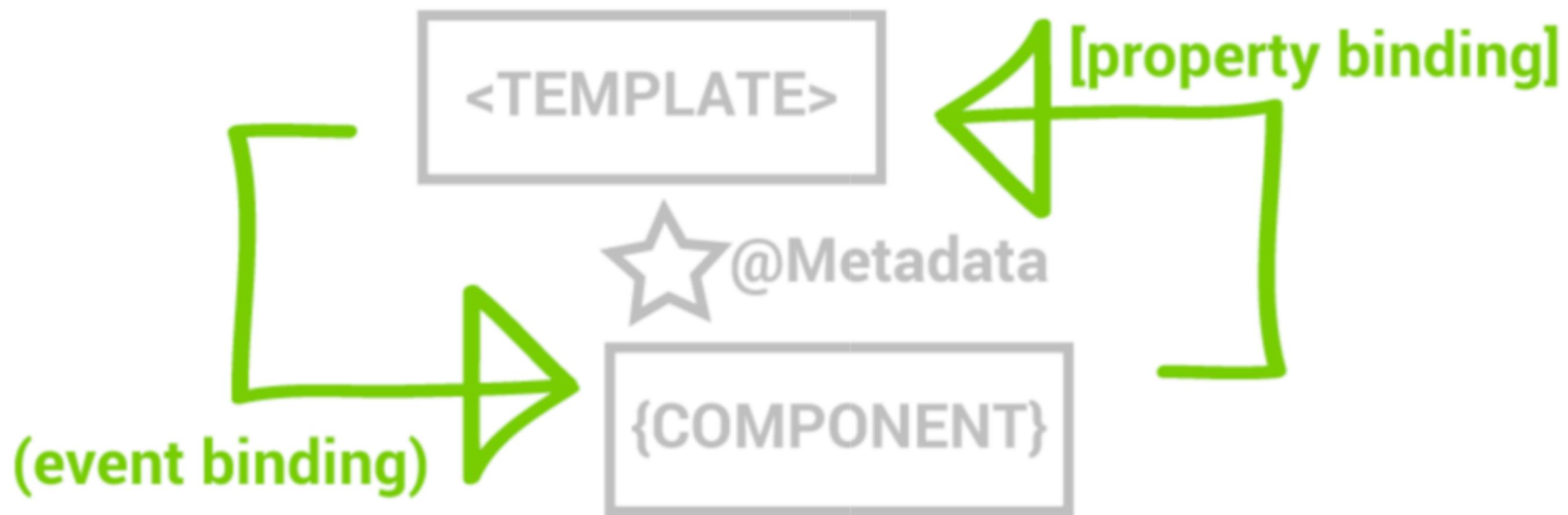
Les composants

Système d'architecture composant

- Les composants sont de petit bout du logiciel qui peuvent être réutilisable dans différents contextes
- Angular 2 et la communauté encouragent vivement ce type d'architecture pour rapidement développer, tester et déployer des fonctionnalités.
- Un composant Angular est autonome au niveau du template, du style et de sa logique



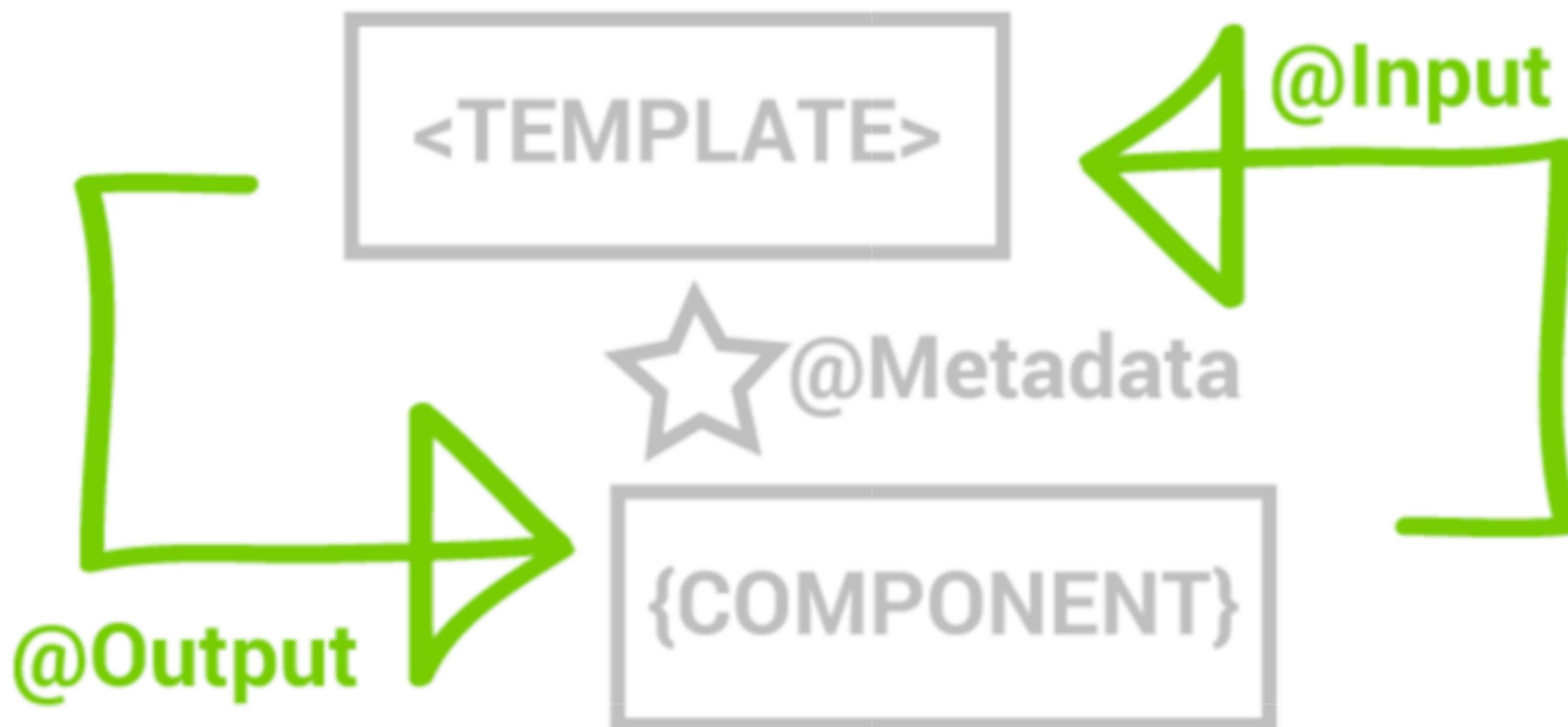
Data binding



Data binding personnalisé

Contrat @Input & @Output

- Un contrat est un engagement entre deux parties: Le développeur et l'utilisateur - ou - le vendeur avec l'acheteur
- **@Input** et **@Output** définissent l'interface d'un composant
- Pour utiliser un composant, on se renseigne sur ces entrées et sorties



Contrat composant

```
<div>
  <item-detail
    (saved)="saveItem($event)"
    (cancelled)="resetItem($event)"
    [item]="selectedItem">Select an Item</item-detail>
</div>
```

Contrat composant

@Input

- Créer un canal de communication depuis un composant parent vers un composant enfant (top-down)
- Décorer une propriété directement dans le composant :
@Input() someValue: string
- On crée le lien dans le parent via un **property binding** :
<component [someValue]="value"></component>
- On peut également différencier le nom de la propriété et le nom du lien dans le template :
@Input('aliasName') someValue: string

```
import { Component, Input } from 'angular2/core';
@Component({
  selector: 'my-component',
  template: `
    <div>Greeting from parent:</div>
    <div>{{greeting}}</div>
  `
})
export class MyComponent {
  @Input() greeting: String = 'Default Greeting';
}
```

@Input

```
import { Component } from 'angular2/core'
import { MyComponent } from './components/my.component'

@Component({
  selector: 'app',
  template: `
    <my-component [greeting]="greeting"></my-component>
    <my-component></my-component>
  `,
  directives: [ MyComponent ]
})
export class App {
  greeting: String = 'Hello child!'
}
```



@Input

@Output

- Expose une propriété de type Event Emitter qui permet de partager les évènements avec le composant parent
- Définition identique à **@Input** avec un décorateur directement dans le composant :
@Output() showValue: new EventEmitter<boolean>
- Le lien avec le parent se fait via un **event binding**
<component (someValue)="handleValue()">
</component>

```
import { Component, Output, EventEmitter } from 'angular2/core'
@Component({
  selector: 'my-component',
  template: `<button (click)="greet()">Greet Me</button>`
})

export class MyComponent {
  @Output() greeter: EventEmitter = new EventEmitter()

  greet() {
    this.greeter.emit('Child greeting emitted!')
  }
}
```



@Output

```
import {MyComponent} from './my-component'

@Component({
  selector: 'app',
  template: `
    <div>
      <h1>{{greeting}}</h1>
      <my-component (greeter)="greet($event)"></my-component>
    </div>
  `,
  directives: [MyComponent]
})

export class App {
  greet(event) {
    this.greeting = event;
  }
}
```

@Output

Composant intelligent ou stupide

- Les composants intelligents sont :
 - connecté à des services
 - Ils savent où chercher leur propres données et comment persister ces données
- Les composants stupide sont :
 - Complètement définie par leurs bindings
 - Toutes leurs données transite via **@Input** et **@Output**
- TIPS : Créer peu de composants intelligents et beaucoup de composants stupide

```
export class ItemsList {
  @Input() items: Item[];
  @Output() selected = new EventEmitter();
  @Output() deleted = new EventEmitter();
}
```

Composant stupide

```
export class App implements OnInit {
  items: Array<Item>;
  selectedItem: Item;

  constructor(private itemsService: ItemsService) {}

  ngOnInit() { }

  resetItem() { }

  selectItem(item: Item) { }

  saveItem(item: Item) { }

  replaceItem(item: Item) { }

  pushItem(item: Item) { }

  deleteItem(item: Item) { }
}
```



Composant Intelligent

Encapsulation de vue

- Cela permet de cloisonner l'inclusion de style
- Il existe (pour l'instant ...) trois types d'encapsulation :
 - **viewEncapsulation.None** : les styles sont globaux à l'application
 - **viewEncapsulation.Emulated** : les styles sont cloisonnées en utilisant un attribut unique sur le composant HTML
 - **viewEncapsulation.Native** : les styles sont cloisonnées en utilisant le shadow DOM natif

```
import { Component, Output, EventEmitter } from 'angular2/core'
@Component({
  selector: 'my-component',
  template: `<button (click)="greet()">Greet Me</button>`,
  encapsulation: ViewEncapsulation.None // or ViewEncapsulation.Emulated or Vie
))

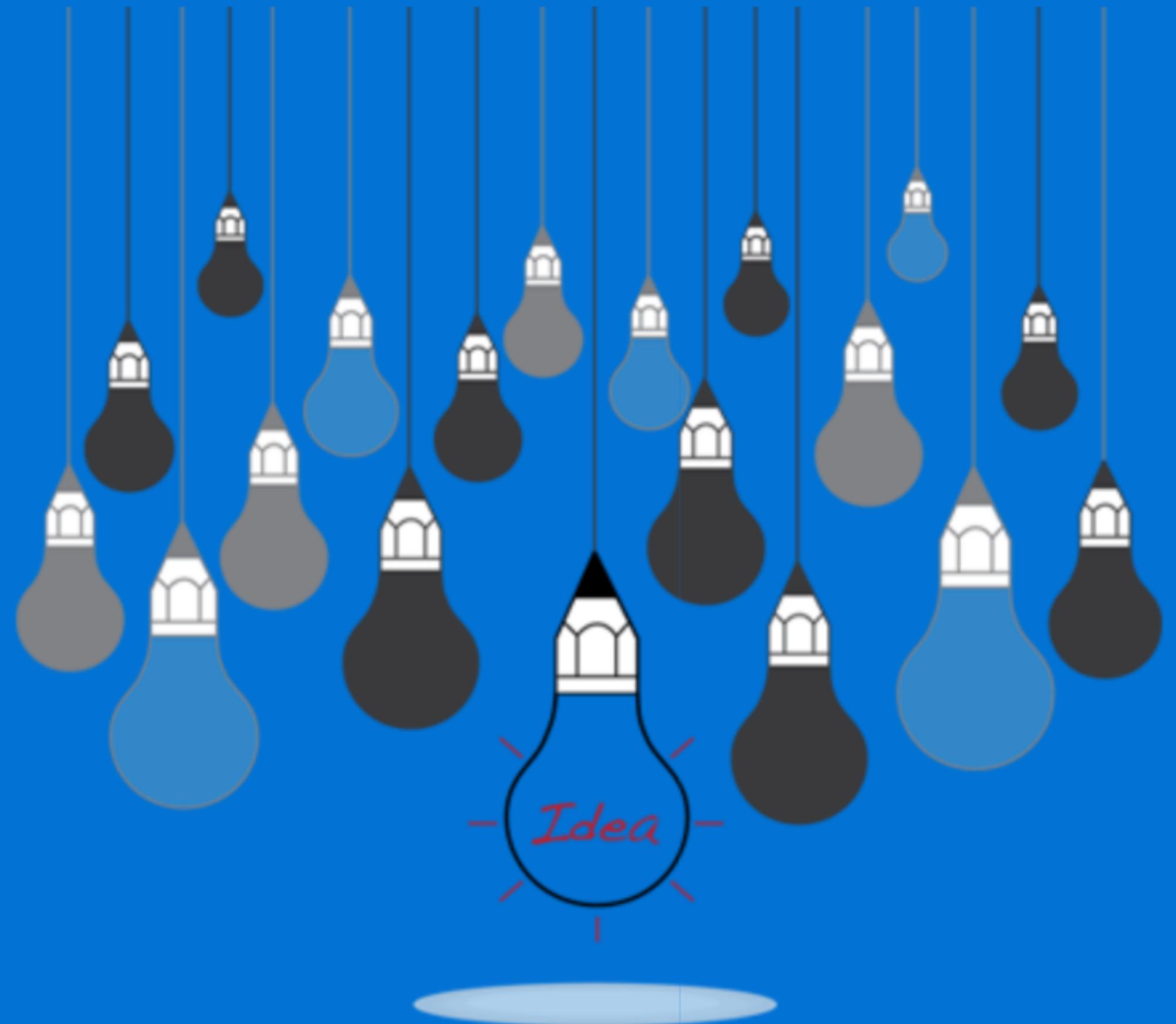
export class MyComponent {
  @Output() greeter: EventEmitter = new EventEmitter()

  greet() {
    this.greeter.emit('Child greeting emitted!')
  }
}
```

Encapsulation de vue

Demo time !





Défi

- Créer un composant stupide **widgets-list** et **item-details** en utilisant **@Input** et **@Output**
- Utiliser le service **widgets**
- Consumer la collection de **widgets** et la faire transiter vers le composant **widgets-list**
- Sélectionner un **widget** depuis la **widgets-list**
- Afficher la sélection dans **item-details**