Angular Master Class

Architecture

Component Communication

Component Communication

There are different possible ways of communicating between components:

Component Communication

There are different possible ways of communicating between components:

- **Inputs/Outputs** - Defining a well-defined API of what goes in and out a component

Component Communication

There are different possible ways of communicating between components:

- **Inputs/Outputs** - Defining a well-defined API of what goes in and out a component
- **Dependency Injection** - Injecting parent components

Component Communication

There are different possible ways of communicating between components:

- **Inputs/Outputs** - Defining a well-defined API of what goes in and out a component
- **Dependency Injection** - Injecting parent components
- **ContentChildren/ViewChildren** - Injecting child components

Component Communication

There are different possible ways of communicating between components:

- **Inputs/Outputs** - Defining a well-defined API of what goes in and out a component
- **Dependency Injection** - Injecting parent components
- **ContentChildren/ViewChildren** - Injecting child components
- **Message Bus** - Implementing a pub/sub service to send and receive messages

Component Types

Component Types

Presentation Components

Component Types

Presentation Components

Business Components

Component Types

Presentation Components

Business Components

View Components

Presentation Components

Presentation components characteristics:

Presentation Components

Presentation components characteristics:

- Display the user interface

Presentation Components

Presentation components characteristics:

- Display the user interface
- Independent of **business** logic (dumb)

Presentation Components

Presentation components characteristics:

- Display the user interface
- Independent of **business** logic (dumb)
- Data usually arrives via bindings (**inputs**)

Presentation Components

Presentation components characteristics:

- Display the user interface
- Independent of **business** logic (dumb)
- Data usually arrives via bindings (**inputs**)
- Data leaves via events (**outputs**)

Business Components

Business components characteristics:

Business Components

Business components characteristics:

- Access to business services & state (smart)

Business Components

Business components characteristics:

- Access to business services & state (smart)
- Bound to specific use case

Business Components

Business components characteristics:

- Access to business services & state (smart)
- Bound to specific use case
- Limited scope of reusability

Business Components

Business components characteristics:

- Access to business services & state (smart)
- Bound to specific use case
- Limited scope of reusability
- May depend on routing

View Components

View components characteristics:

View Components

View components characteristics:

- Build the current view (from a URL)

View Components

View components characteristics:

- Build the current view (from a URL)
- Always bound to a specific URL

View Components

View components characteristics:

- Build the current view (from a URL)
- Always bound to a specific URL
- Composes other components to render UI

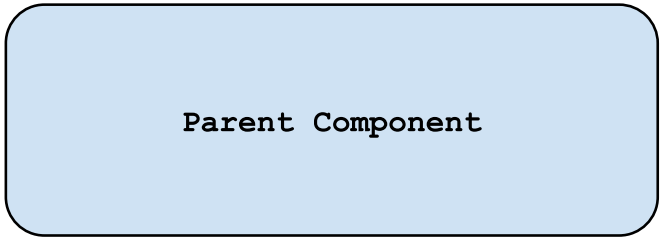
View Components

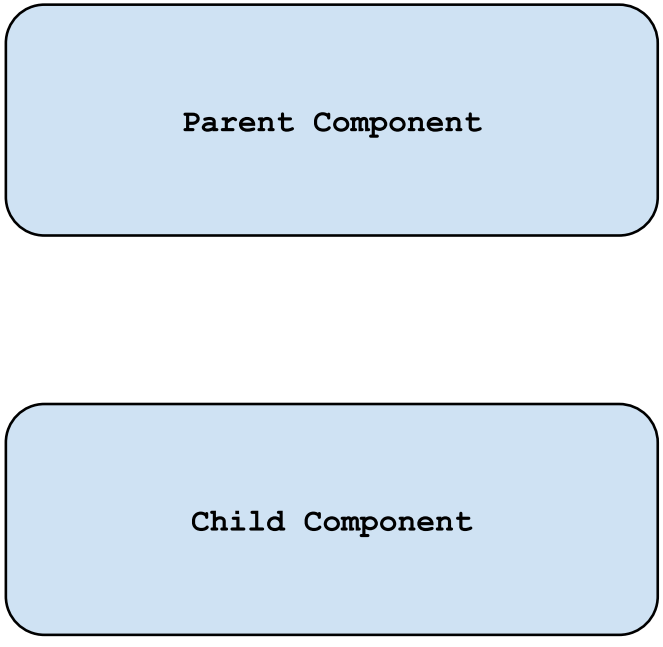
View components characteristics:

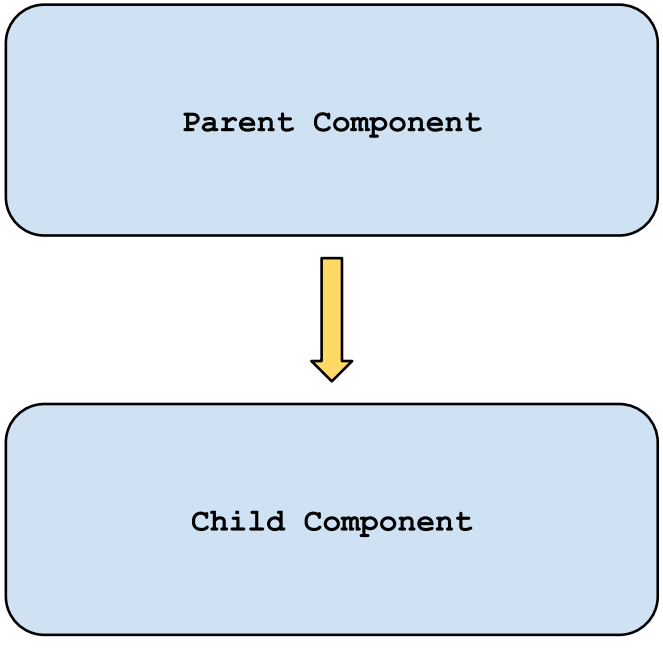
- Build the current view (from a URL)
- Always bound to a specific URL
- Composes other components to render UI
- Can be entry points to applications

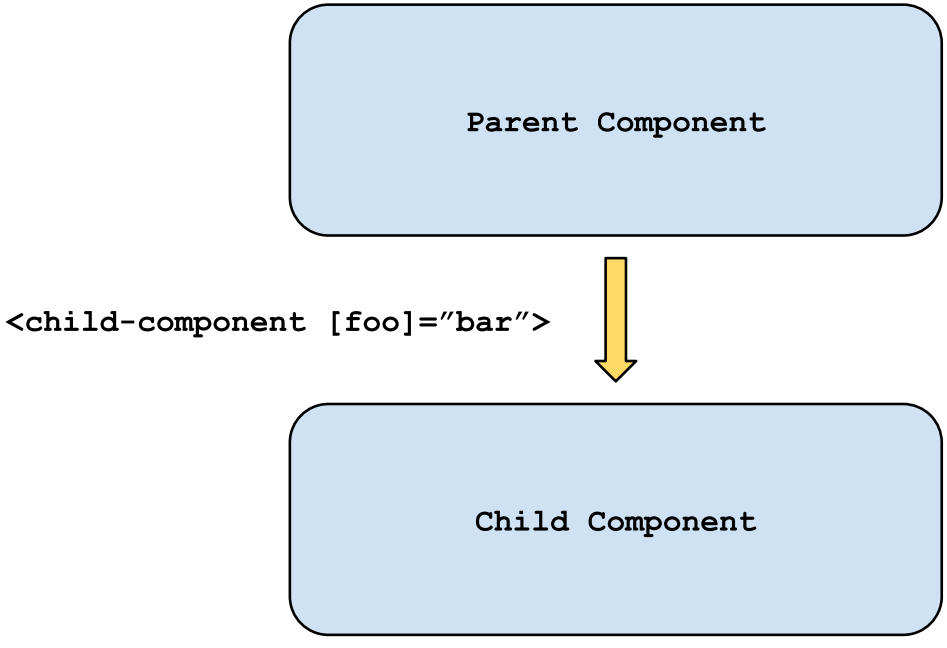
Inputs and Outputs

Inputs describe what **data flows into** a component.


Parent Component


Parent Component
Child Component


Parent Component
Child Component


Parent Component
<child-component [foo]="bar">
Child Component

Inputs using @Input()

We define component inputs using the **@Input** decorator.

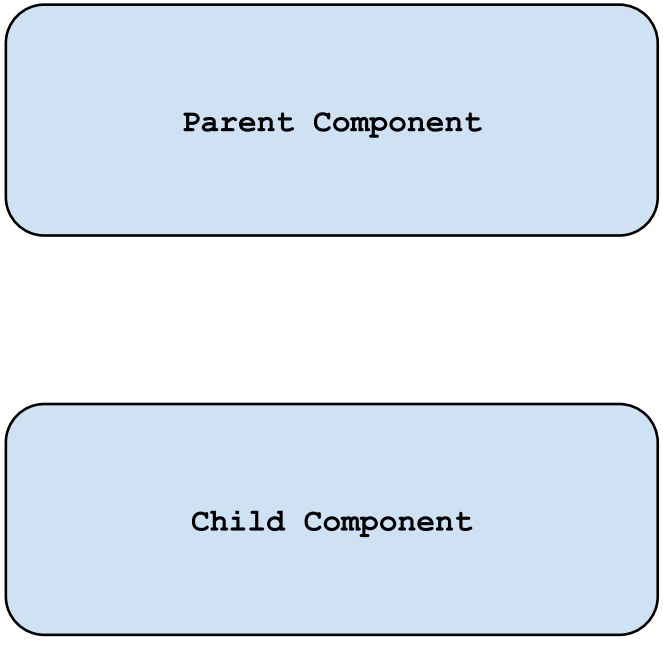
```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'trm-contacts-detail',
  template: 'This is {{contact.name}}'
})
export class ContactsDetailComponent {
  @Input() contact: Contact;
}
```

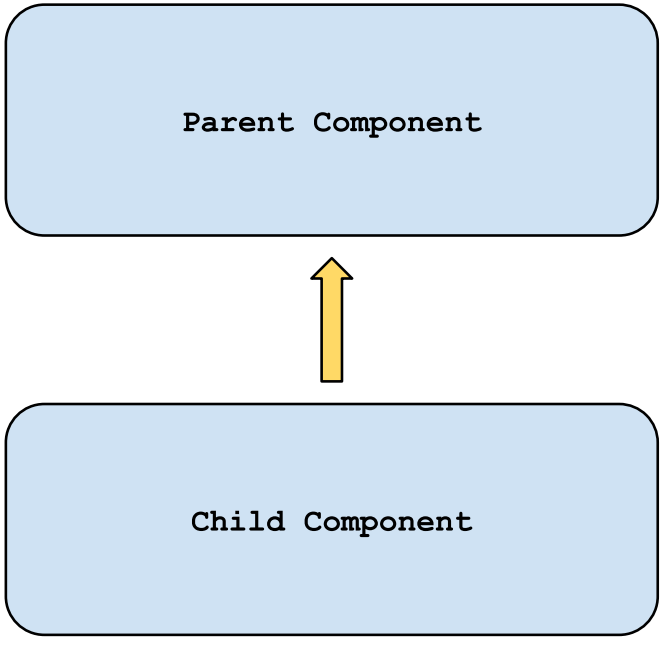
Inputs using @Input()

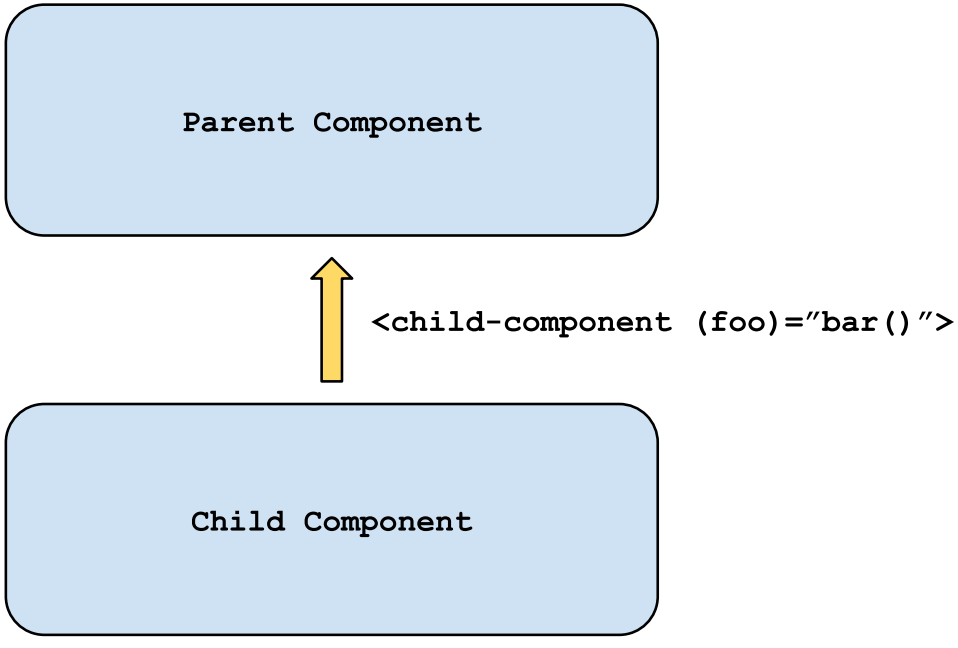
We define component inputs using the **@Input** decorator.

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

@Component({
  selector: 'trm-contacts-detail',
  template: 'This is {{contact.name}}'
})
export class ContactsDetailComponent {
  @Input() contact: Contact;
}
```

Outputs describe what **data flows out** of a component.


Parent Component
Child Component


Parent Component
Child Component


Parent Component
<child-component (foo)="bar()">
Child Component

Outputs

Outputs are decorated with **@Output** and are **EventEmitter** instances.

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

@Component({
  selector: 'trm-contacts-detail',
  template: 'This is {{contact.name}}'
})
export class ContactsDetailComponent {
  @Input() contact: Contact;
  @Output() edit = new EventEmitter<Contact>();
}
```

Outputs

Outputs are decorated with **@Output** and are **EventEmitter** instances.

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

@Component({
  selector: 'trm-contacts-detail',
  template: 'This is {{contact.name}}'
})
export class ContactsDetailComponent {
  @Input() contact: Contact;
  @Output() edit = new EventEmitter<Contact>();
}
```

Emitting custom events

We emit custom events using **EventEmitter.emit()** with an optional payload.

```
@Component({
  selector: 'trm-contacts-detail',
  template: `
    This is {{contact.name}}
    <button (click)="edit.emit(contact)">Edit</button>
  `
})
export class ContactsDetailComponent {
  @Input() contact: Contact;
  @Output() edit = new EventEmitter<Contact>();
}
```

Emitting custom events

We emit custom events using **EventEmitter.emit()** with an optional payload.

```
@Component({
  selector: 'trm-contacts-detail',
  template: `
    This is {{contact.name}}
    <button (click)="edit.emit(contact)">Edit</button>
  `
})
export class ContactsDetailComponent {
  @Input() contact: Contact;
  @Output() edit = new EventEmitter<Contact>();
}
```

Consuming custom events

We listen to custom events the same way we do to native events via event binding

```
<trm-contacts-detail
  [contact]="contact"
  (edit)="doSomething($event)">
</trm-contacts-detail>
```

Consuming custom events

We listen to custom events the same way we do to native events via event binding

```
<trm-contacts-detail
  [contact]="contact"
  (edit)="doSomething($event)">
</trm-contacts-detail>
```

$event holds the optional payload emitted by the EventEmitter.

Exercise: Using Inputs and Outputs

t forget to git commit your solution

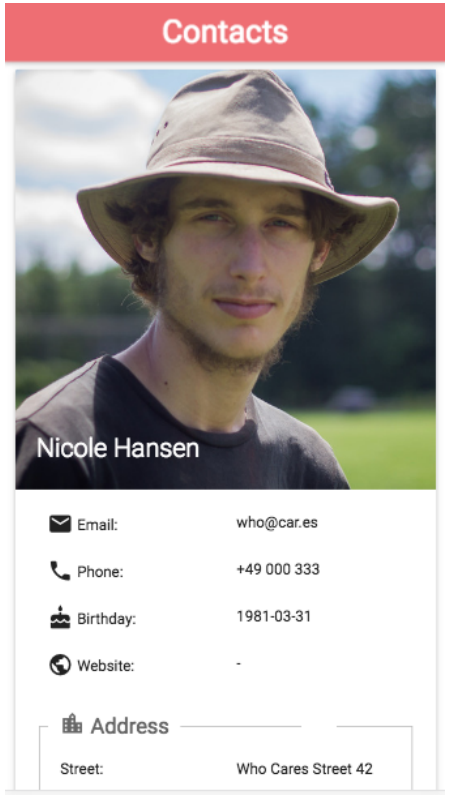
Injecting Parent Components

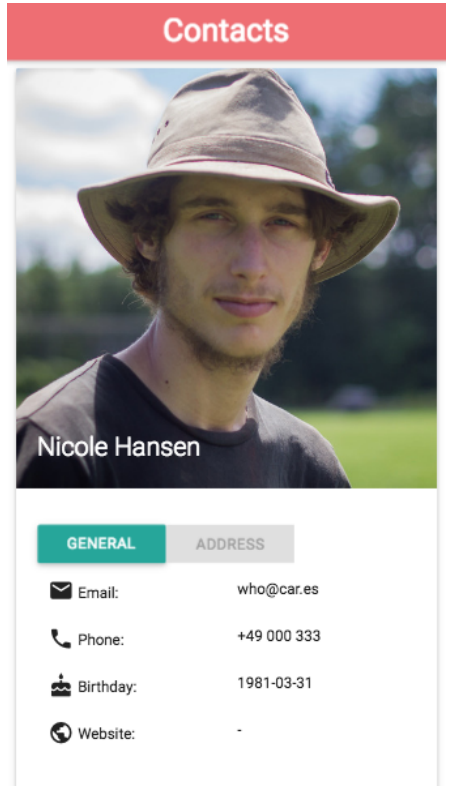
Contacts

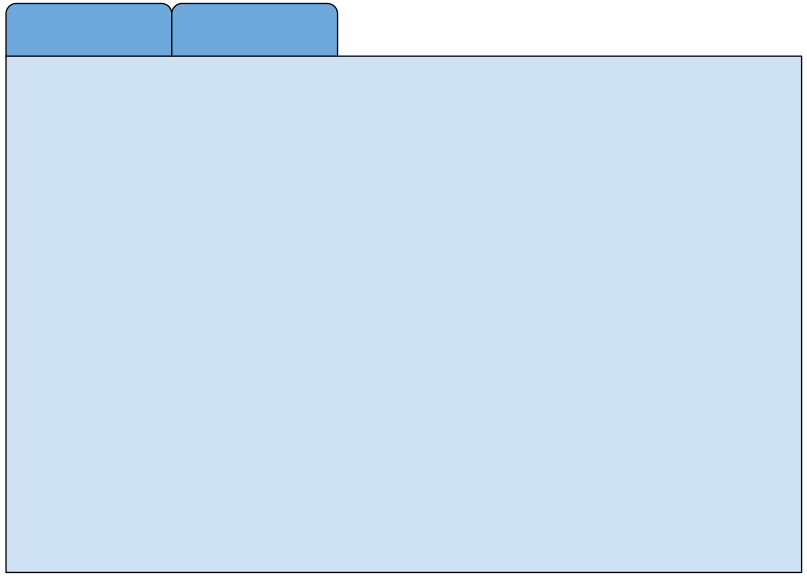
Email:	who@car.es
Phone:	+49 000 333
Birthday:	1981-03-31
Website:	-

Address

Street:	Who Cares Street 42


Contacts
Nicole Hansen
Email:
who@car.es
Phone:
+49 000 333
Birthday:
1981-03-31
Website:
-
Address
Street:
Who Cares Street 42


Contacts
Nicole Hansen
GENERAL
ADDRESS
Email:
who@car.es
Phone:
+49 000 333
Birthday:
1981-03-31
Website:
-



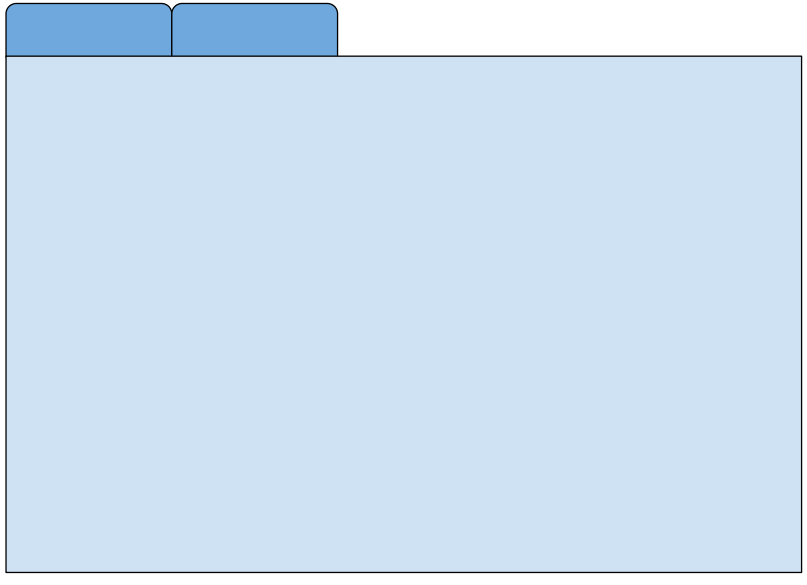
```
<tabs>
  <tab title="General">
    ...
  </tab>

  <tab title="Address">
    ...
  </tab>
</tabs>
```

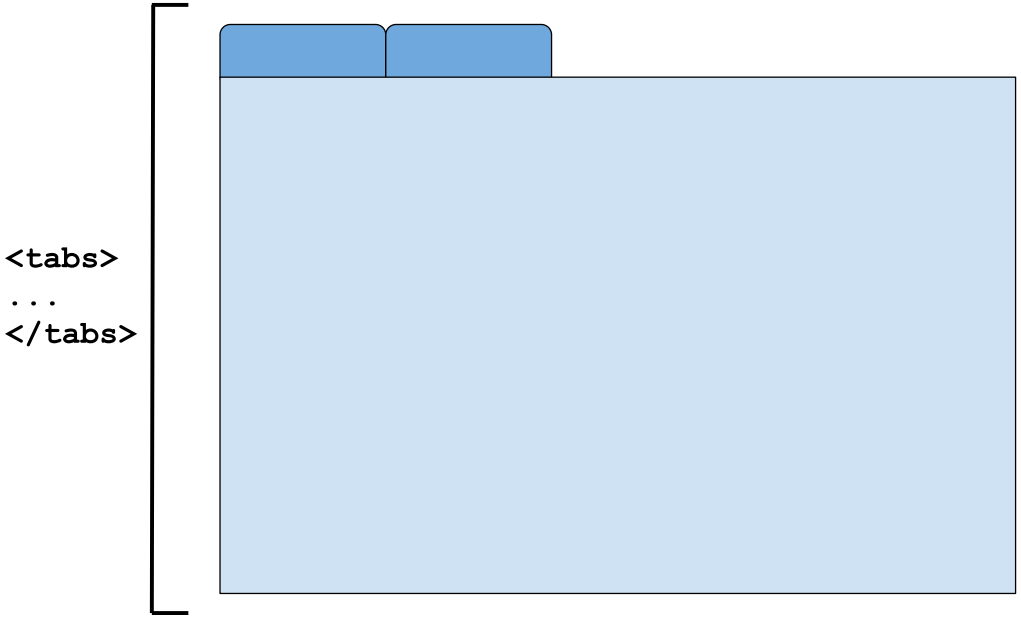
```
@Component({ selector: 'tabs' })
class TabsComponent {}
```

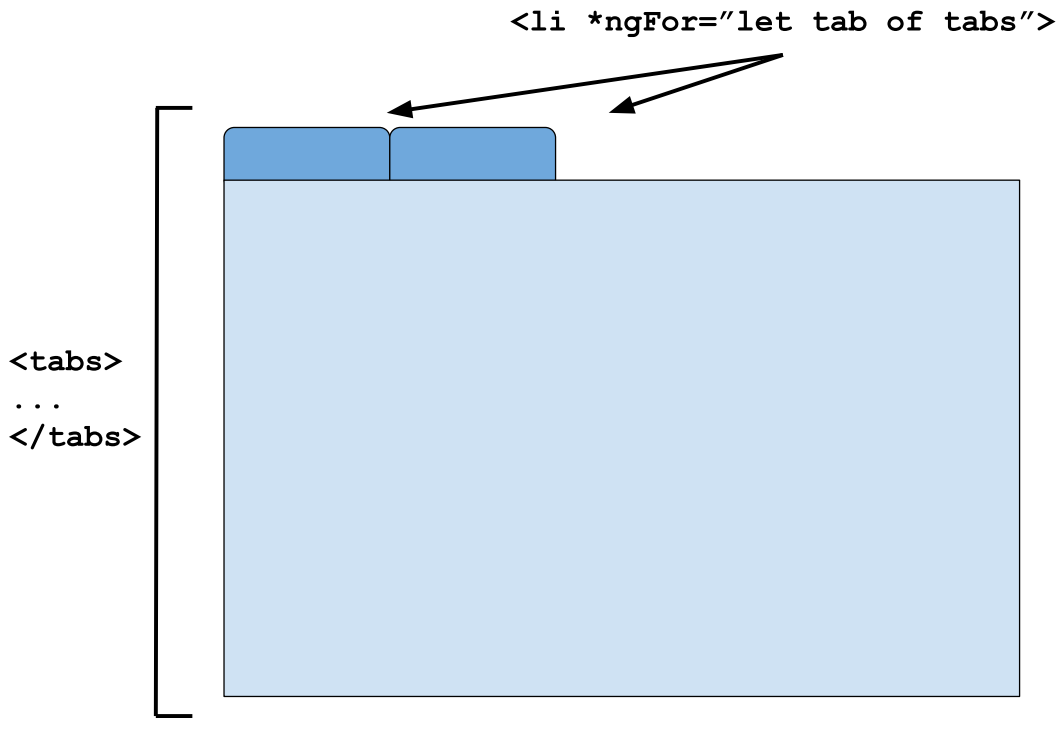
```
<tabs>
  <tab title="General">
    ...
  </tab>

  <tab title="Address">
    ...
  </tab>
</tabs>
```

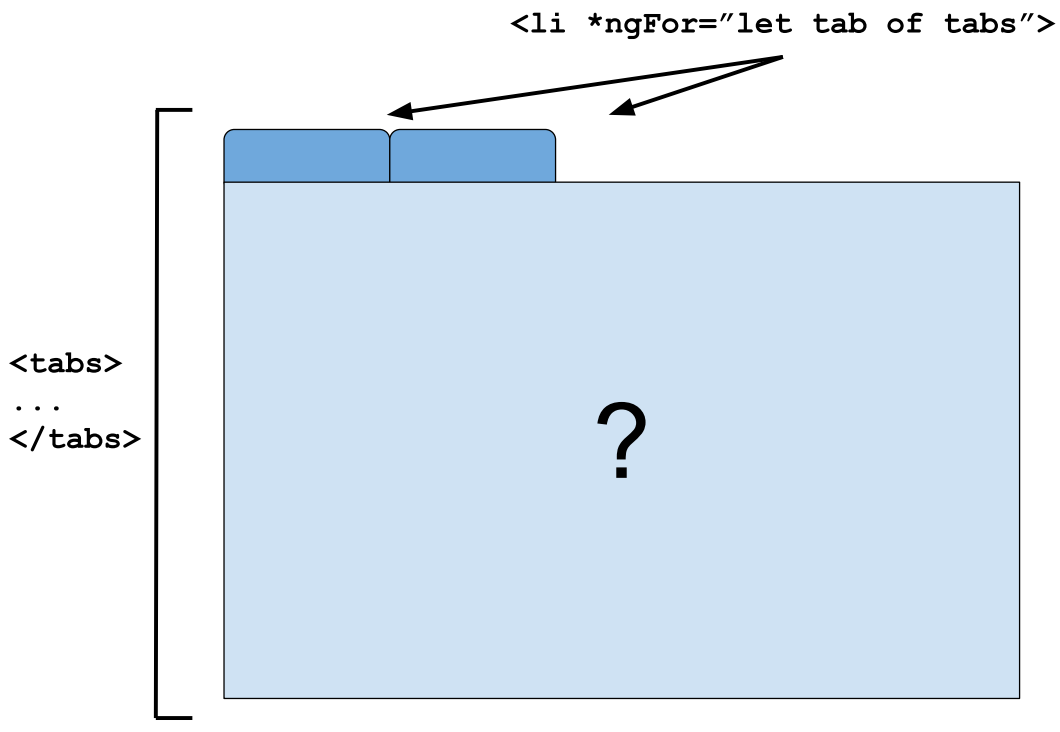
```
<tabs>
  <tab title="General">
    ...
  </tab>

  <tab title="Address">
    ...
  </tab>
</tabs>
```

```
@Component({ selector: 'tab' })
class TabComponent {}
```




<tabs>
...
</tabs>


<li *ngFor="let tab of tabs">
<tabs>
...
</tabs>


<li *ngFor="let tab of tabs">
<tabs>
...
</tabs>
?

```
<tabs>
  <tab title="General">
    ...
  </tab>

  <tab title="Address">
    ...
  </tab>
</tabs>
```

Content Projection

We project a component's content using **<ng-content>**

```
@Component({
  selector: 'trm-tabs',
  template: `
    <div mat-tab-nav-bar>
      <button mat-tab-link
        *ngFor="let tab of tabs">...</button>
    </div>
    <ng-content></ng-content>
  `
})
class TabsComponent {}
```

Content Projection

We project a component's content using **<ng-content>**

```
@Component({
  selector: 'trm-tabs',
  template: `
    <div md-tab-nav-bar>
      <button md-tab-link
        *ngFor="let tab of tabs">...</button>
    </div>
    <ng-content></ng-content>
  `
})
class TabsComponent {}
```

How do we get access to child components?

Injecting Components

Dependency Injection allows us to inject **parent** component instances.

Injecting Components

Dependency Injection allows us to inject **parent** component instances.

```
@Component({
  selector: 'trm-tab'
})
class TabComponent implements OnInit {

  constructor(private tabs: TabsComponent) {}

  ngOnInit() {
    this.tabs.addTab(this);
  }
}
```

Injecting Components

Dependency Injection allows us to inject **parent** component instances.

```
@Component({
  selector: 'trm-tab'
})
class TabComponent implements OnInit {

  constructor(private tabs: TabsComponent) {}

  ngOnInit() {
    this.tabs.addTab(this);
  }
}
```

xercise: Building a Tabs component

t forget to git commit your solution

ContentChildren and ViewChildren

Injecting child components

The following decorators give us access to component children:

Injecting child components

The following decorators give us access to component children:

- **@ContentChild()** - Query single content child

Injecting child components

The following decorators give us access to component children:

- **@ContentChild()** - Query single content child
- **@ContentChildren()** - Query list of content children

Injecting child components

The following decorators give us access to component children:

- **@ContentChild()** - Query single content child
- **@ContentChildren()** - Query list of content children
- **@ViewChild()** - Query single view child

Injecting child components

The following decorators give us access to component children:

- **@ContentChild()** - Query single content child
- **@ContentChildren()** - Query list of content children
- **@ViewChild()** - Query single view child
- **@ViewChildren()** - Query list of view children

What are **content children** and **view children**?

Content Children

Content children are the elements between the opening and closing tag of the host element of a given component (hence, **<ng-content>**).

```
<tabs>
  <tab title="General">
    ...
  </tab>

  <tab title="Address">
    ...
  </tab>
</tabs>
```

View Children

View Children are the elements located **inside** a component's view (template).

```
<ul>
  <li *ngFor="...">
    …
  </li>
<ul>
<ng-content></ng-content>
```

```
@Component({
  selector: 'trm-tabs'
})
class TabsComponent {



  tabs: Array<TabComponent>;
  ...




}
```

```
import { QueryList, AfterContentInit } from '@angular/core';

@Component({
  selector: 'trm-tabs'
})
class TabsComponent {

  @ContentChildren(TabComponent)
  tabs: QueryList<TabComponent>;
  ...




}
```

```
import { QueryList, AfterContentInit } from '@angular/core';

@Component({
  selector: 'trm-tabs'
})
class TabsComponent {

  @ContentChildren(TabComponent)
  tabs: QueryList<TabComponent>;
  ...




}
```

```
import { QueryList, AfterContentInit } from '@angular/core';

@Component({
  selector: 'trm-tabs'
})
class TabsComponent implements AfterContentInit {

  @ContentChildren(TabComponent)
  tabs: QueryList<TabComponent>;
  ...
  ngAfterContentInit() {

  }
}
```

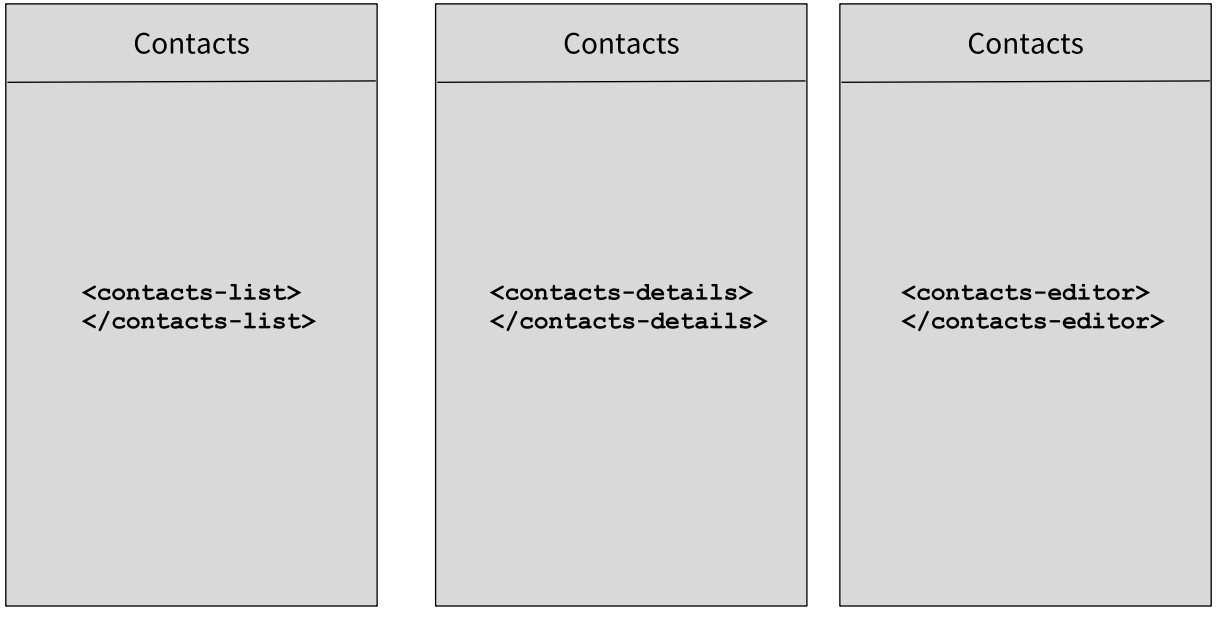
```
import { QueryList, AfterContentInit } from '@angular/core';

@Component({
  selector: 'trm-tabs'
})
class TabsComponent implements AfterContentInit {

  @ContentChildren(TabComponent)
  tabs: QueryList<TabComponent>;
  ...
  ngAfterContentInit() {

  }
}
```

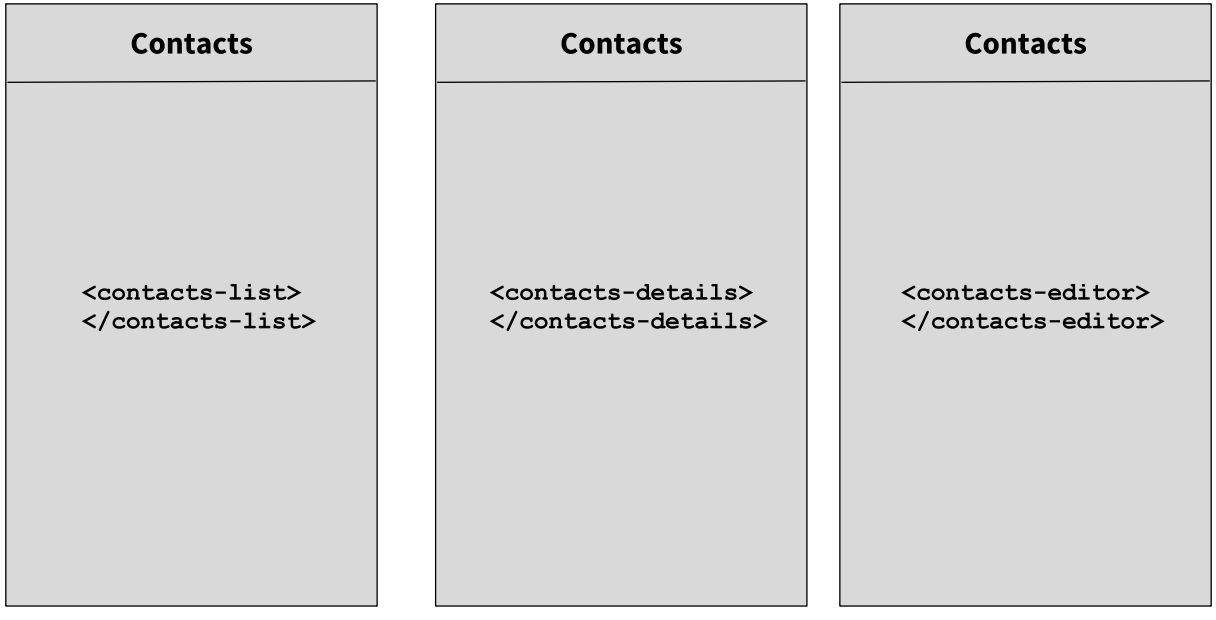
```
import { QueryList, AfterContentInit } from '@angular/core';

@Component({
  selector: 'trm-tabs'
})
class TabsComponent implements AfterContentInit {

  @ContentChildren(TabComponent)
  tabs: QueryList<TabComponent>;
  ...
  ngAfterContentInit() {
    this.selectTab(this.tabs.first);
  }
}
```

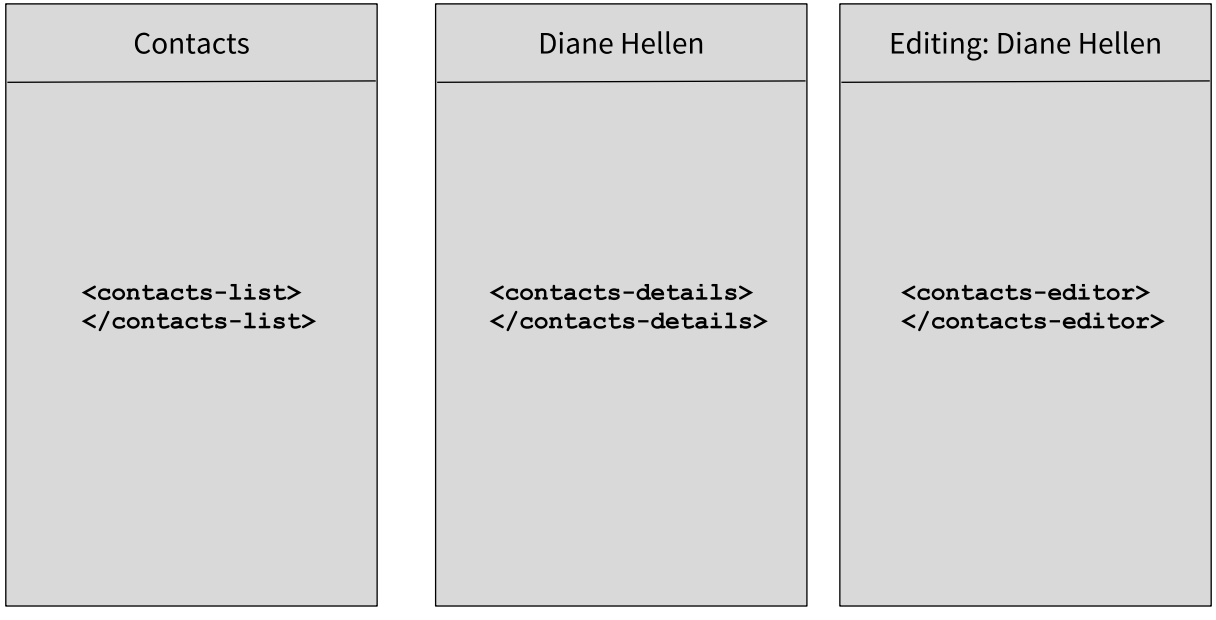
ercise: Refactor TabsComponent with ContentChildren

t forget to git commit your solution

Creating a custom EventBus


Contacts
<contacts-list>
</contacts-list>
Contacts
<contacts-details>
</contacts-details>
Contacts
<contacts-editor>
</contacts-editor>


Contacts
<contacts-list>
</contacts-list>
Contacts
<contacts-details>
</contacts-details>
Contacts
<contacts-editor>
</contacts-editor>


Contacts
<contacts-list>
</contacts-list>
Diane Hellen
<contacts-details>
</contacts-details>
Editing: Diane Hellen
<contacts-editor>
</contacts-editor>

The Problem

Currently, nothing updates the title when a view changes

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-toolbar>Contacts</mat-toolbar>
    <router-outlet></router-outlet>
  `
})
class ContactsAppComponent {...}
```

Solving the problem

It turns out we can solve this problem in a simple way with the tools we already have:

- **Observable Service** - An event bus service that emits when something happened
- **Dependency Injection** - Inject the service where it needs to emit and subscribe

Creating an EventBus

Creating an EventBus

We can use a **Subject** to teach our service how to emit events

```
class EventBusService  {

  private _messages$ = new Subject<EventBusArgs>();









}
```

Creating an EventBus

We can use a **Subject** to teach our service how to emit events

```
class EventBusService  {

  private _messages$ = new Subject<EventBusArgs>();

  emit(eventType: string, data: any) {
    this._messages$.next({ type: eventType, data: data });
  }





}
```

Creating an EventBus

We can use a **Subject** to teach our service how to emit events

```
class EventBusService  {

  private _messages$ = new Subject<EventBusArgs>();

  emit(eventType: string, data: any) {
    this._messages$.next({ type: eventType, data: data });
  }

  observe(eventType: string) {
    return this._messages$.pipe(
      filter(args => args.type === eventType),
      map(args => args.data)
    );
  }
}
```

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-toolbar>{{title}}</mat-toolbar>
    <router-outlet></router-outlet>
  `
})
class ContactsAppComponent {
  constructor(private eventBus: EventBusService) {}

  ngOnInit() {
    this.eventBus.observe('appTitleChange')
                 .subscribe(title => this.title = title);
  }
}
```

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-toolbar>{{title}}</mat-toolbar>
    <router-outlet></router-outlet>
  `
})
class ContactsAppComponent {
  constructor(private eventBus: EventBusService) {}

  ngOnInit() {
    this.eventBus.observe('appTitleChange')
                 .subscribe(title => this.title = title);
  }
}
```

Exercise: Creating an EventBus

t forget to git commit your solution

THOUGHTRAM

EXTEND YOUR MEMORY

- git add .
- git commit -am "(completed) - architecture"
- git tag classroom/architecture