Workshop

Angular

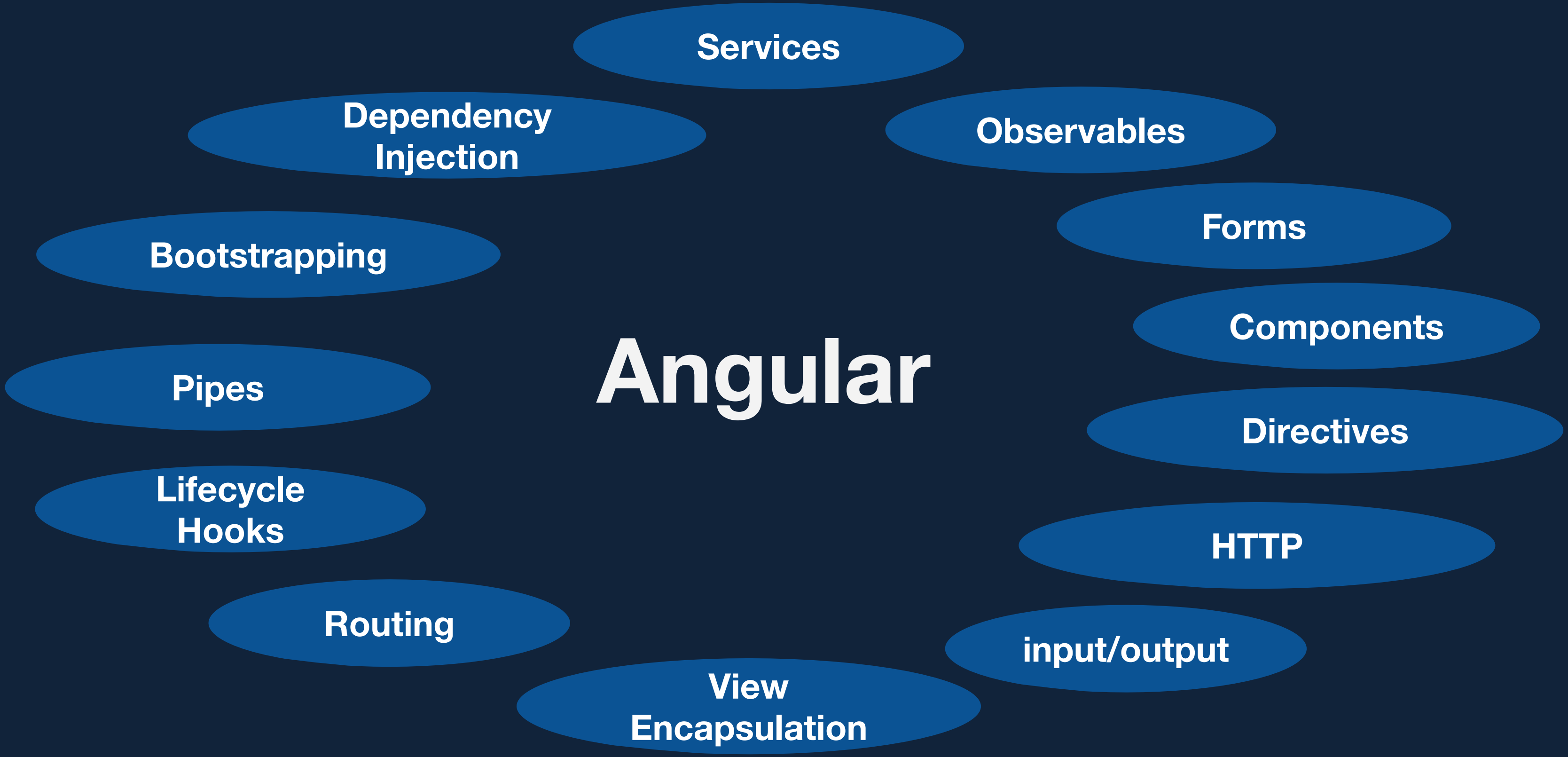
> [!NOTE]
> Dies ist unser erster RFC zum Angular Workshop.
> Alle sind eingeladen RFCs (Request for Comments zu schreiben).
> Geht in die Diskussion, tauscht euch aus.
> Nehmt @GregOnNet gern als Sparring-Partner bei Fragen dazu.

Motivation

- [input](https://www.angular.courses/caniuse?search=input) [output](https://www.angular.courses/caniuse?search=output) sind Nachfolger der Decorator-APIs `@Input` und `@Output`
- Diese APIs sind `stable`.
- Der Einsatz von `input` reduziert Use-Cases für `ngOnChanges`.
- `input`und `output` sind ergonomischer im Umgang, wenn es im TypeSCript-Code genutzt werden muss.
- Die neuen APIs lösen ebenso das Problem mit typescript `strict`-Mode:

Hint for trainers

- Report each change or addition to the **trainers’** Discord-Channel.
- Tell which Slide is affected, why the change is important and what benefit your change provides.
- Use the code-highlighting-app if you work with code-snippets.
- Use the following slide if you want to repeat certain topics of the workshop.



Angular

Services

Observables

Forms

Components

Directives

HTTP

input/output

View Encapsulation

Routing

Lifecycle Hooks

Pipes

Bootstrapping

Dependency Injection

Angular

A platform for building mobile and desktop web applications

Mission

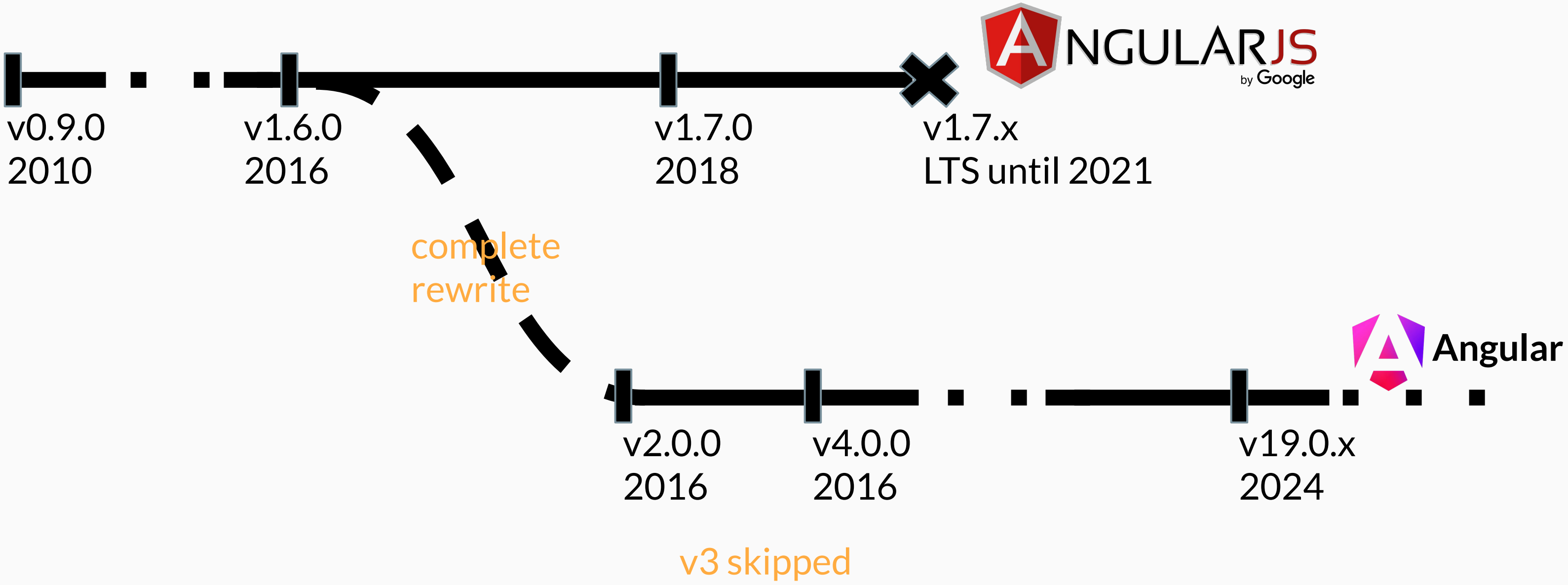
Apps that users ❤️ to use.

Apps that developers ❤️ to build.

Community where everyone feels welcome.

Source: Igor Minar - Ex Tech Lead Angular Team

History



v0.9.0
2010

v1.6.0
2016

v1.7.0
2018

v1.7.x
LTS until 2021

complete
rewrite

v2.0.0
2016

v4.0.0
2016

v19.0.x
2024

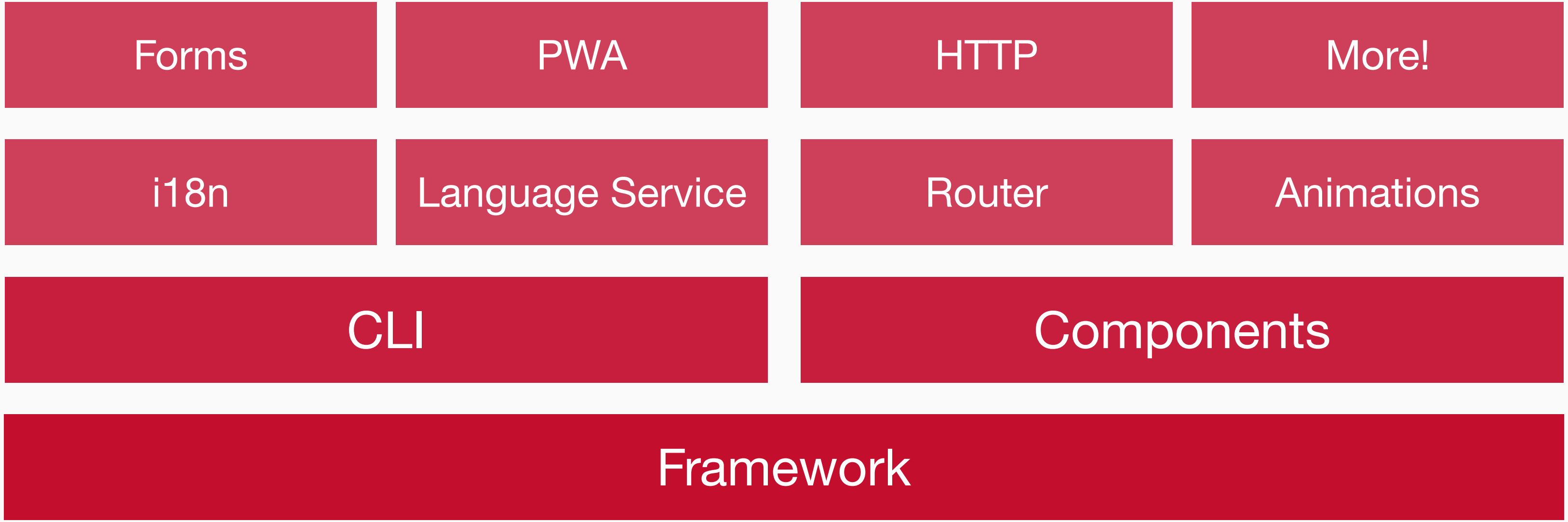
v3 skipped

Release Cycle

> In general, you can expect the following release cycle:
>
> - A major release every 6 months
> - 1-3 minor releases for each major release
> - A patch release almost every week

https://angular.io/guide/releases

Platform overview



Forms

PWA

HTTP

More!

i18n

Language Service

Router

Animations

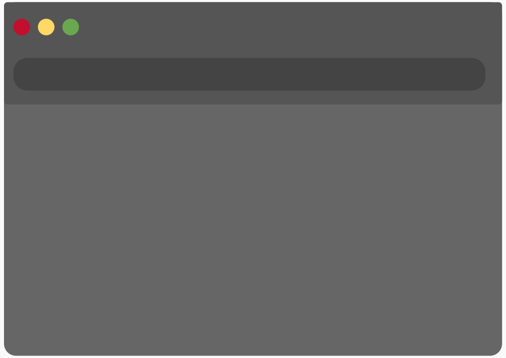
CLI

Components

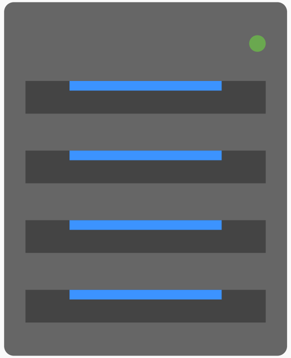
Framework

Image is inspired by @mgechev (speakerdeck)

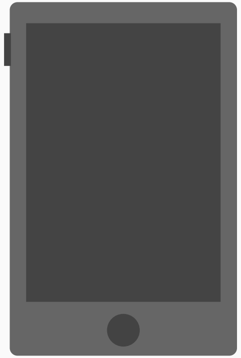
Cross Platform



Browser



Server



Mobile

Cross Platform

ionic

Mobile

Angular@Google

Firebase

Google Cloud Platform

Google Express

Google Analytics

...

+2000 more

Summary

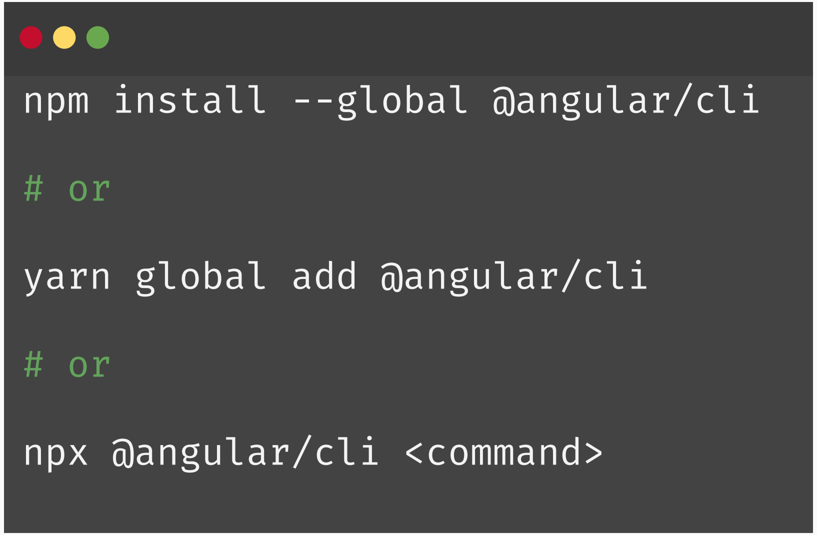
- All-in-one platform
- Has its roots in AngularJS
- Focus on maintainability
  - Automated Updates through CLI
  - Opinionated and lintable style guide
- Oriented to web standards
- Google itself relies heavily on Angular

Angular CLI

Overview

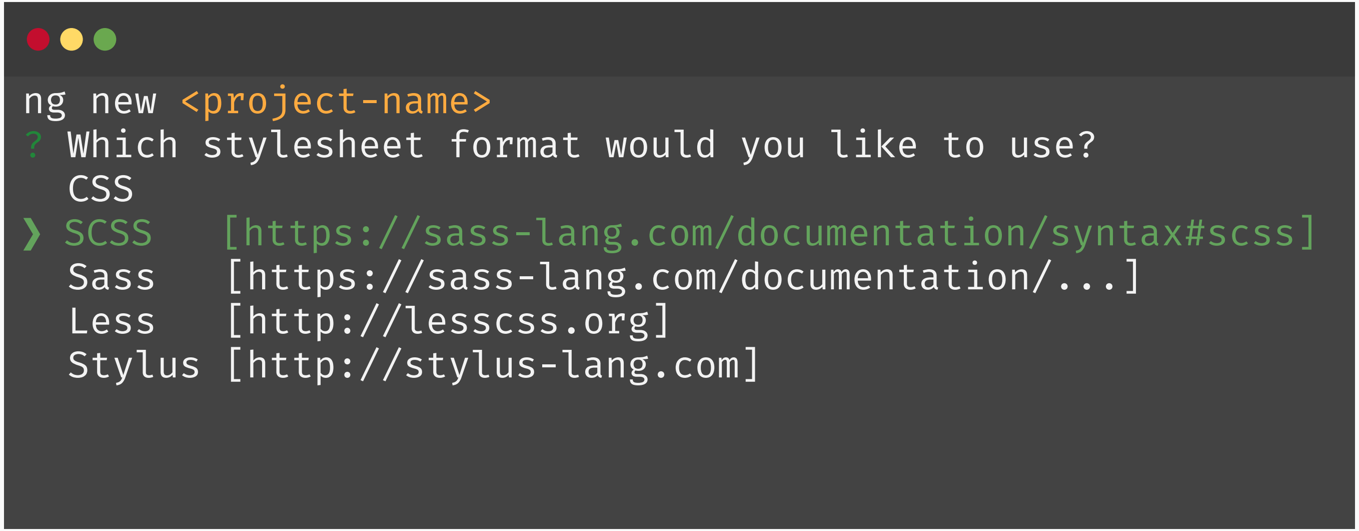
- Project scaffolding
- Code Generation
- Linting
- Test Execution
- Build optimization
- Deployment helpers

Set up the CLI



```
npm install --global @angular/cli

# or

yarn global add @angular/cli

# or

npx @angular/cli <command>
```

Generate a new project

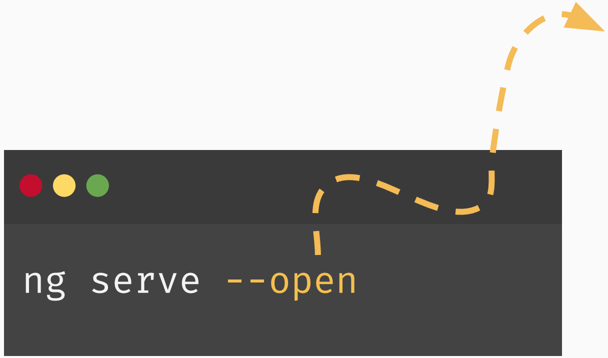


```
ng new <project-name>
? Which stylesheet format would you like to use?
  CSS
❯ SCSS   [https://sass-lang.com/documentation/syntax#scss]
  Sass   [https://sass-lang.com/documentation/...]
  Less   [http://lesscss.org]
  Stylus [http://stylus-lang.com]
```

Generate code

Type	Usage
Component	`ng g component book-list`
Directive	`ng g directive tooltip`
Service	`ng g service book-data`
Pipe	`ng g pipe shout`
Interface	`ng g interface book`
Class	`ng g class book`

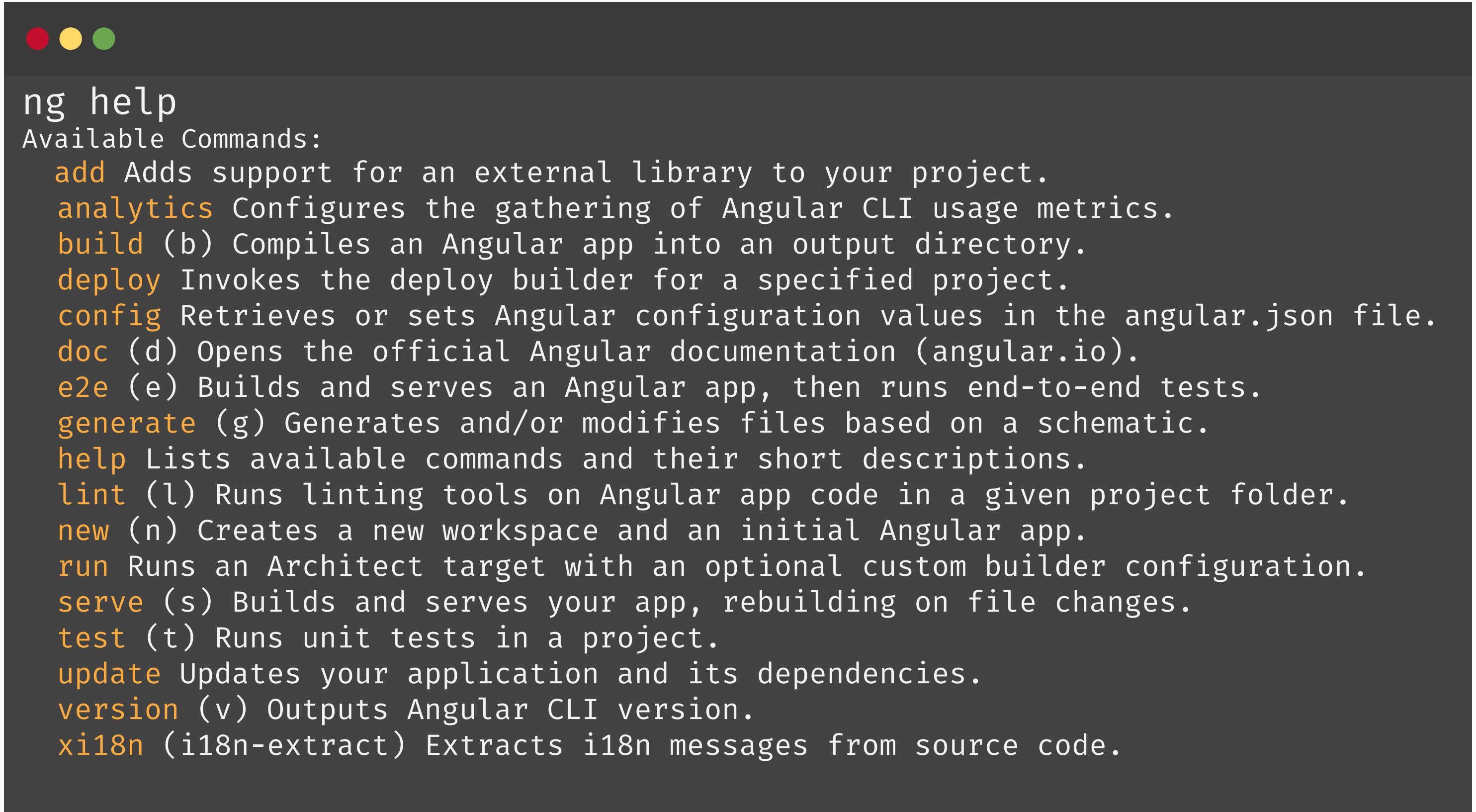
Start your app



```
ng serve --open
```

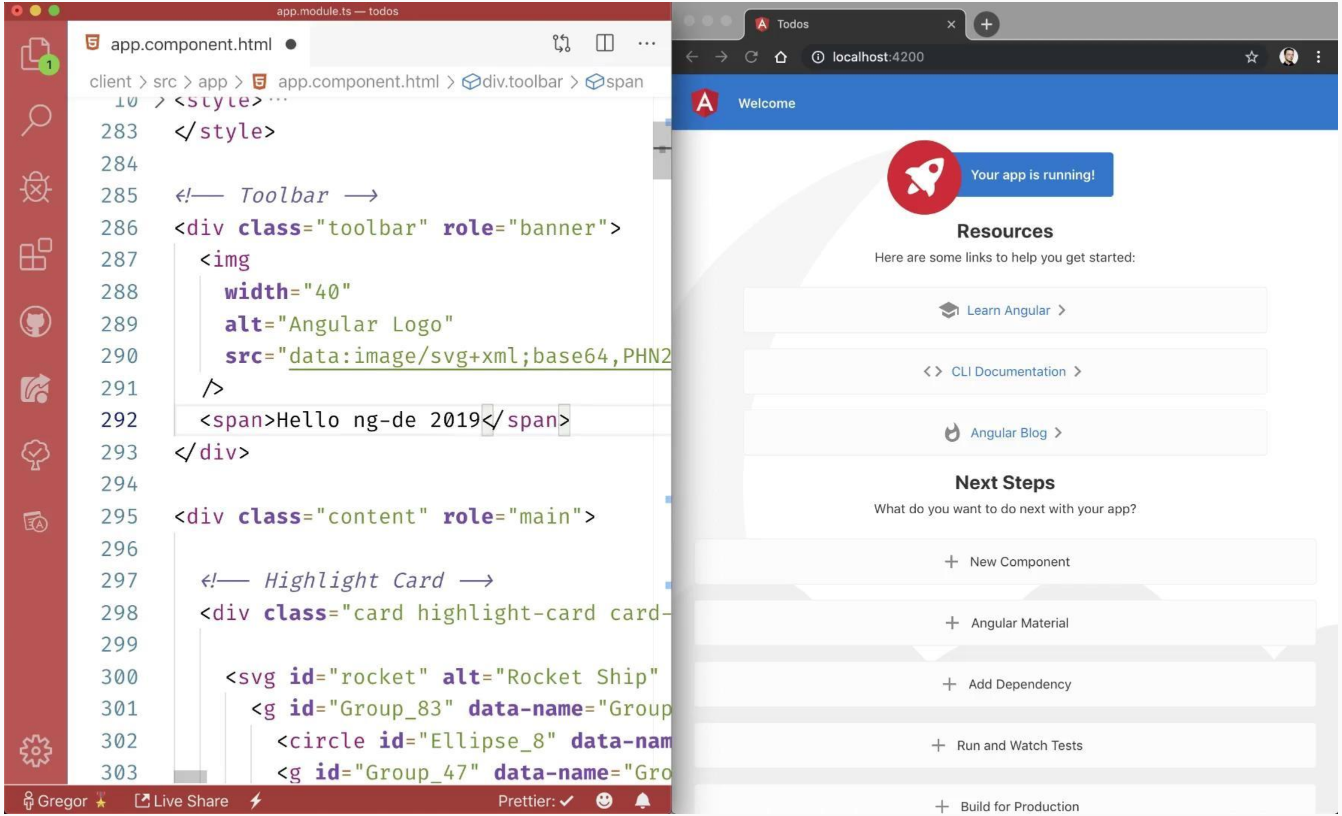
opens your browser

Learn more about the CLI



```
ng help
Available Commands:
  add Adds support for an external library to your project.
  analytics Configures the gathering of Angular CLI usage metrics.
  build (b) Compiles an Angular app into an output directory.
  deploy Invokes the deploy builder for a specified project.
  config Retrieves or sets Angular configuration values in the angular.json file.
  doc (d) Opens the official Angular documentation (angular.io).
  e2e (e) Builds and serves an Angular app, then runs end-to-end tests.
  generate (g) Generates and/or modifies files based on a schematic.
  help Lists available commands and their short descriptions.
  lint (l) Runs linting tools on Angular app code in a given project folder.
  new (n) Creates a new workspace and an initial Angular app.
  run Runs an Architect target with an optional custom builder configuration.
  serve (s) Builds and serves your app, rebuilding on file changes.
  test (t) Runs unit tests in a project.
  update Updates your application and its dependencies.
  version (v) Outputs Angular CLI version.
  xi18n (i18n-extract) Extracts i18n messages from source code.
```

Reload on code changes



```
<style>···
</style>

<!-- Toolbar -->
<div class="toolbar" role="banner">
  <img
    width="40"
    alt="Angular Logo"
    src="data:image/svg+xml;base64,PHN2
  />
  <span>Hello ng-de 2019</span>
</div>

<div class="content" role="main">

  <!-- Highlight Card -->
  <div class="card highlight-card card-

    <svg id="rocket" alt="Rocket Ship"
      <g id="Group_83" data-name="Group
        <circle id="Ellipse_8" data-nam
        <g id="Group_47" data-name="Gro
```

Welcome

Your app is running!

Resources

Here are some links to help you get started:

- Learn Angular >
- CLI Documentation >
- Angular Blog >

Next Steps

What do you want to do next with your app?

- + New Component
- + Angular Material
- + Add Dependency
- + Run and Watch Tests
- + Build for Production

New build system

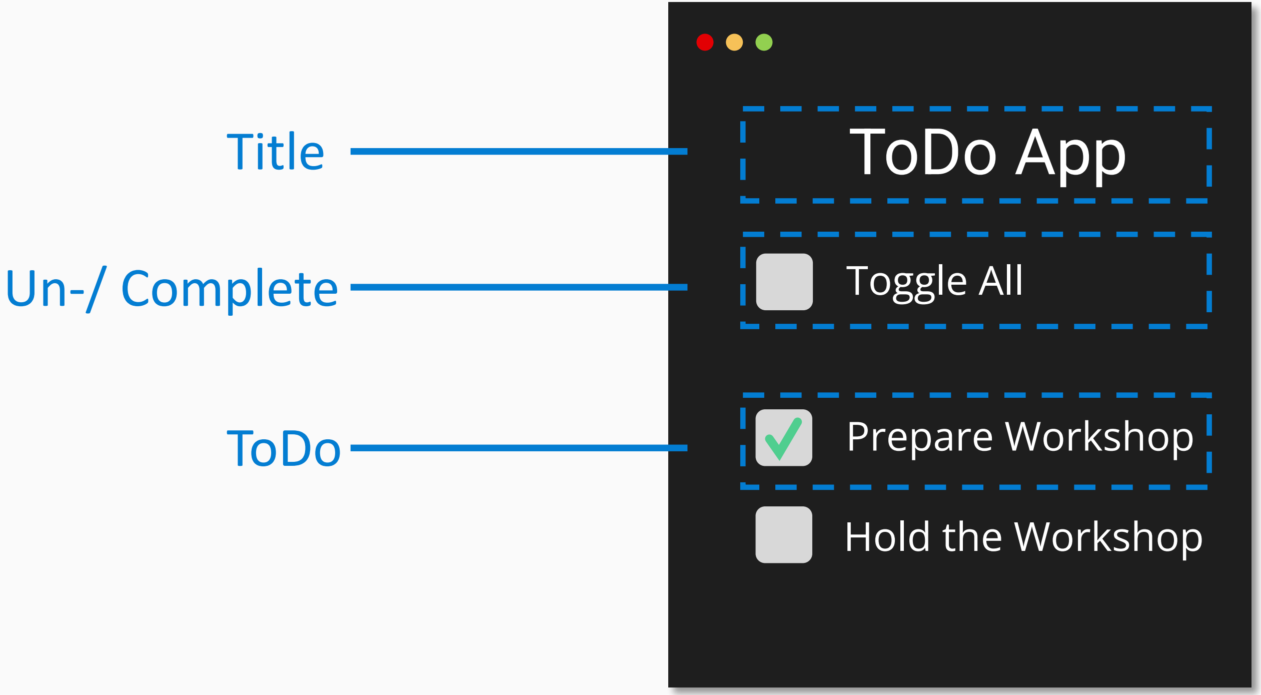
esbuild and vite

- New build system provides an improved way to build Angular applications
- Modern output format using ESM, with dynamic import expressions to support lazy module loading.
- Faster build-time performance for both initial builds and incremental rebuilds
- Integrated SSR and pre-rendering capabilities
- Existing Webpack-based build system is still considered stable and can still be used

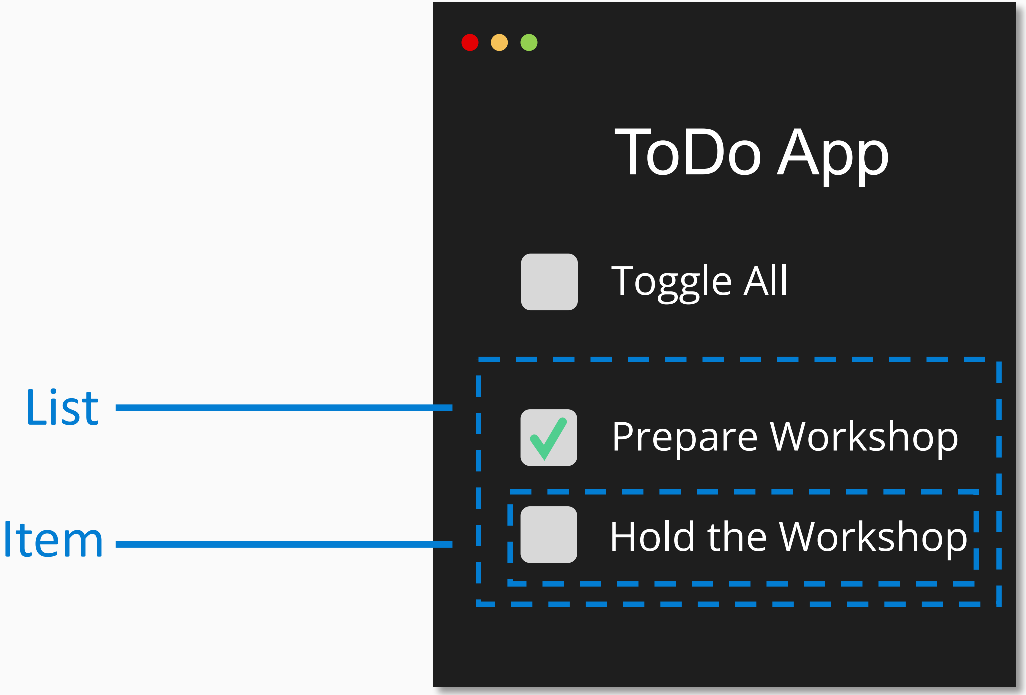
Components

The heart of Angular

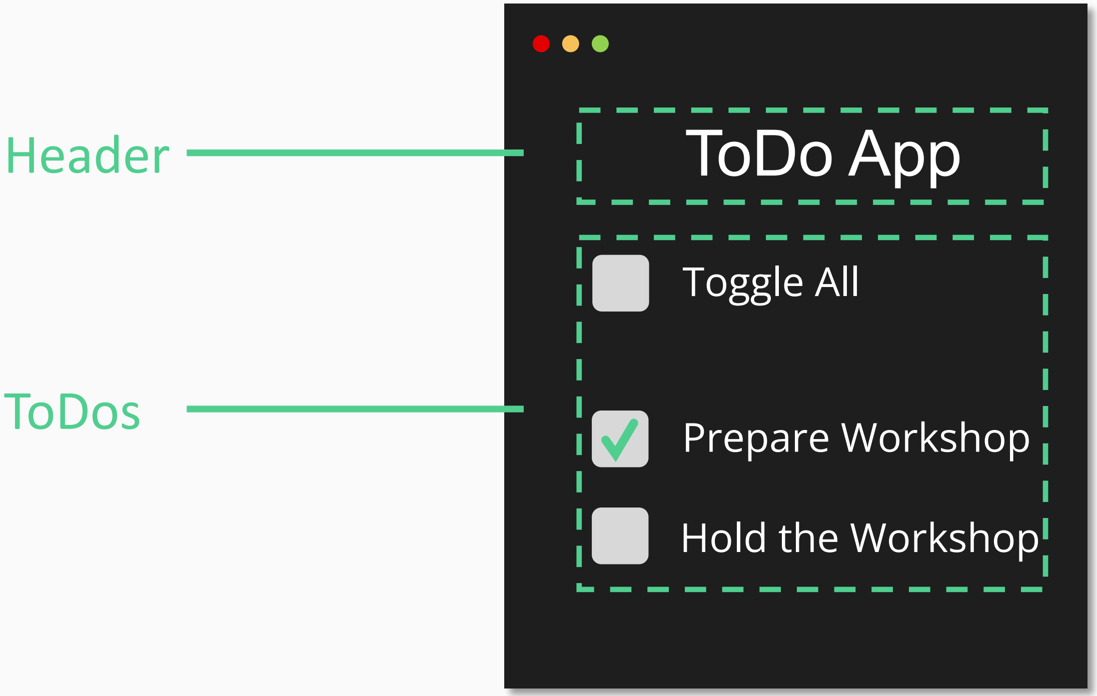
Think in small building blocks



Group wisely



Think in features



Header

ToDos

ToDo App

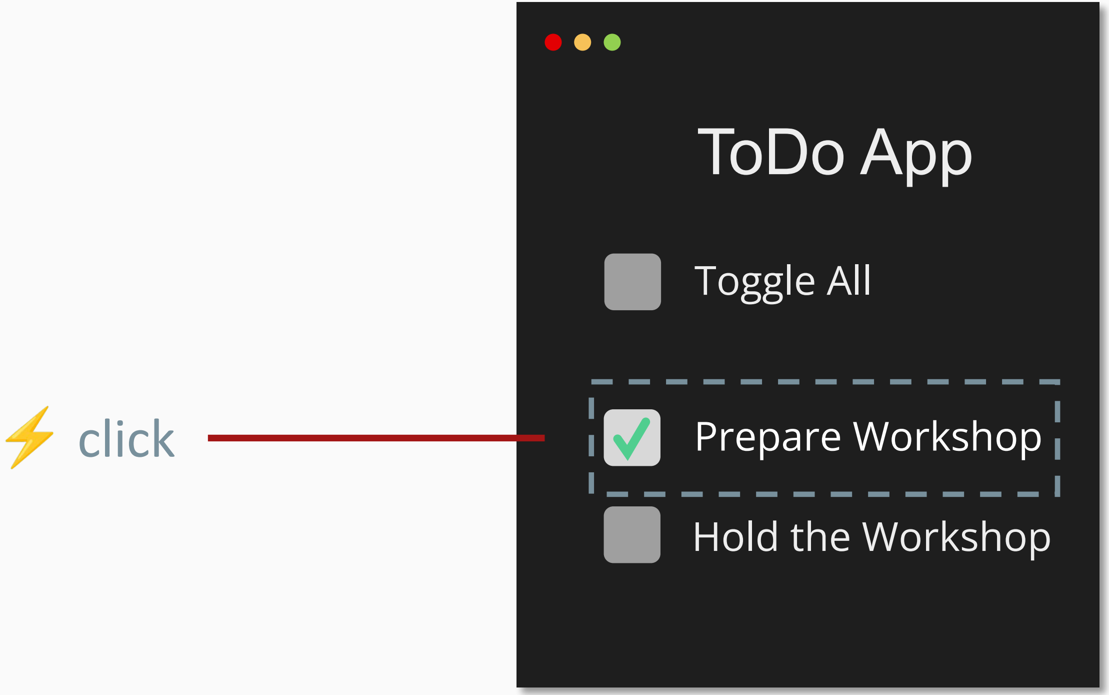
☐ Toggle All

☑ Prepare Workshop

☐ Hold the Workshop

What is a Component?

Representation & Interaction



⚡ click

ToDo App

□ Toggle All

☑ Prepare Workshop

□ Hold the Workshop

What is a Component?

Template & Styling

What is a Component?

Styling: We can use CSS-Extension-Languages, too.

What is a Component?

Interaction

Component Declaration

The `AppComponent`

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class App {
  title = 'My App';
}
```

Component Declaration

<code>

The selector

index.html

```html
<body>
   <app-root></app-root>
</body>
```

app.component.ts

```typescript
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {
  title = 'My App';
}
```

Component Declaration

Available selector types

`element-name`	select by element name. (preferred way)
`.class`	select by class name.
`[attribute]`	select by attribute name.
`[attribute=value]`	select by attribute name and value.
`:not(sub_selector)`	select only if the element does not match the *sub_selector*.
`selector1, selector2`	select if either *selector1* or *selector2* matches.

Component Declaration

Example: `class`-selector

Ausgeblendet / wenig relevant

index.html

```html
<table>
   <tr class=”app-row”></tr>
</table>
```

app.component.ts

```typescript
@Component({
 selector: '.app-row',
 // ...
})
export class RowComponent {}
```

Component Declaration

Example: `attribute-value`-selector

Ausgeblendet / wenig relevant

index.html

```
<div app-summary=”pricing”>
```

app.component.ts

```
@Component({
 selector: '[app-summary=pricing]',
 // ...
})
export class SummaryComponent {}
```

Selectors give you a lot of possibilities to integrate your component in the DOM. Nevertheless, in `95%` of all cases you stick with the element-selector.

Component Declaration

Standalone flag

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {
  title = 'My App';
}
```

Component Declaration

Imports

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {
  title = 'My App';
}
```

Component Declaration

The template

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {
  title = 'My App';
}
```

Component Declaration

<code>

The style

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {
  title = 'My App';
}
```

Component Declaration

The logic

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {
  title = 'My App';
}
```

Component Declaration

Binding Expression

app.component.html

```
<span>
  {{ title }} app is running!
</span>
```

app.component.ts

```
@Component(/* ... */)
export class AppComponent {
  title = 'My App';
}
```

Component Declaration

Binding expression: Dos and Don’ts

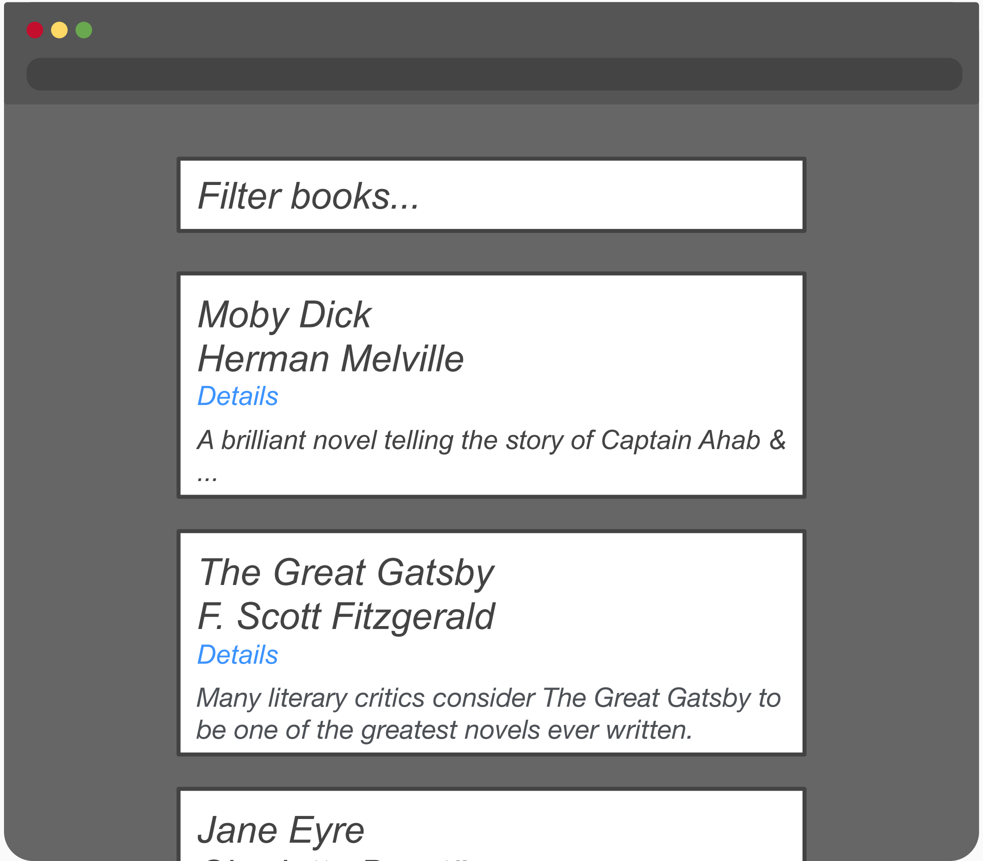
`{{ expression }}`	✅	Display data.
`{{ 1 + 2 + book.price }}`	🤔 **sometimes ok**	Execute simple calculations.
`{{ showPrice(book) }}`	⚠️ ⚠️ **Please, be careful**	Call a function in a Binding.

Task

Meet the AppComponent

AppComponent - The entry point of your App

What are you going to build?



Filter books...

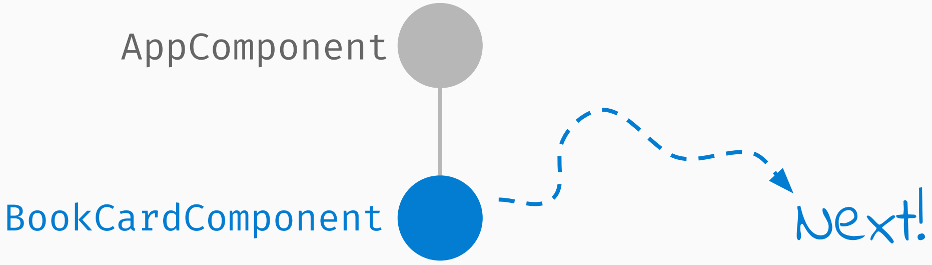
Moby Dick
Herman Melville
Details
A brilliant novel telling the story of Captain Ahab & ...

The Great Gatsby
F. Scott Fitzgerald
Details
Many literary critics consider The Great Gatsby to be one of the greatest novels ever written.

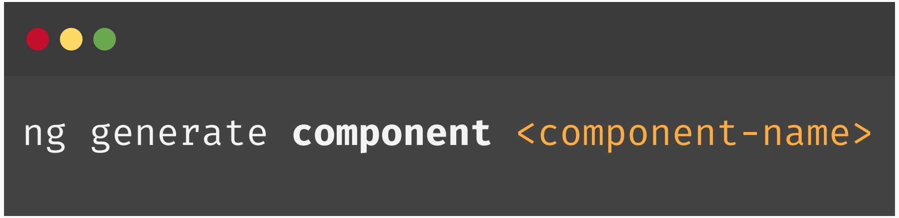
Jane Eyre

AppComponent - The entry point of your App

Add more features by creating more components

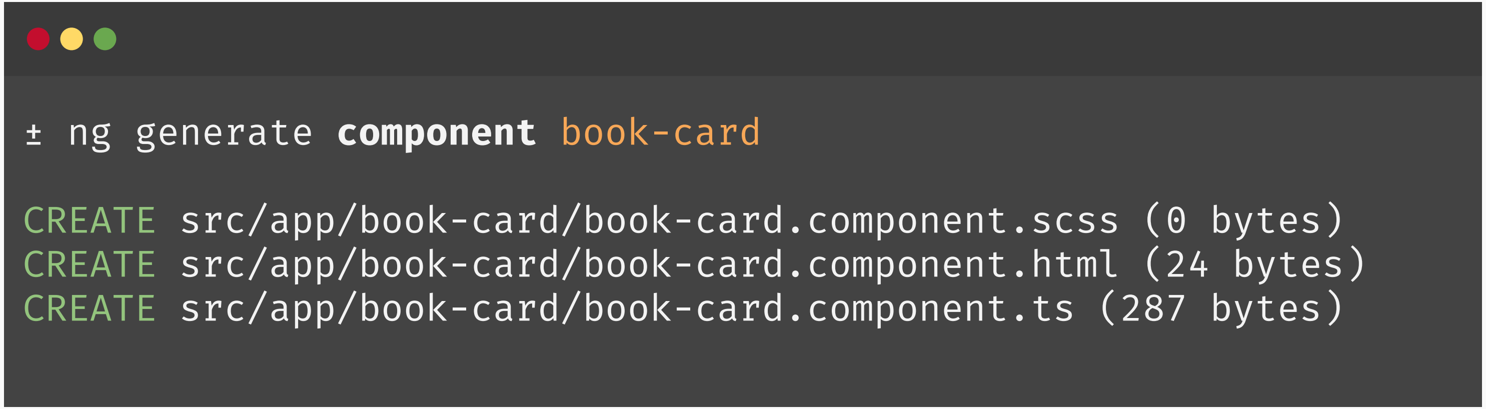


Generate a new component



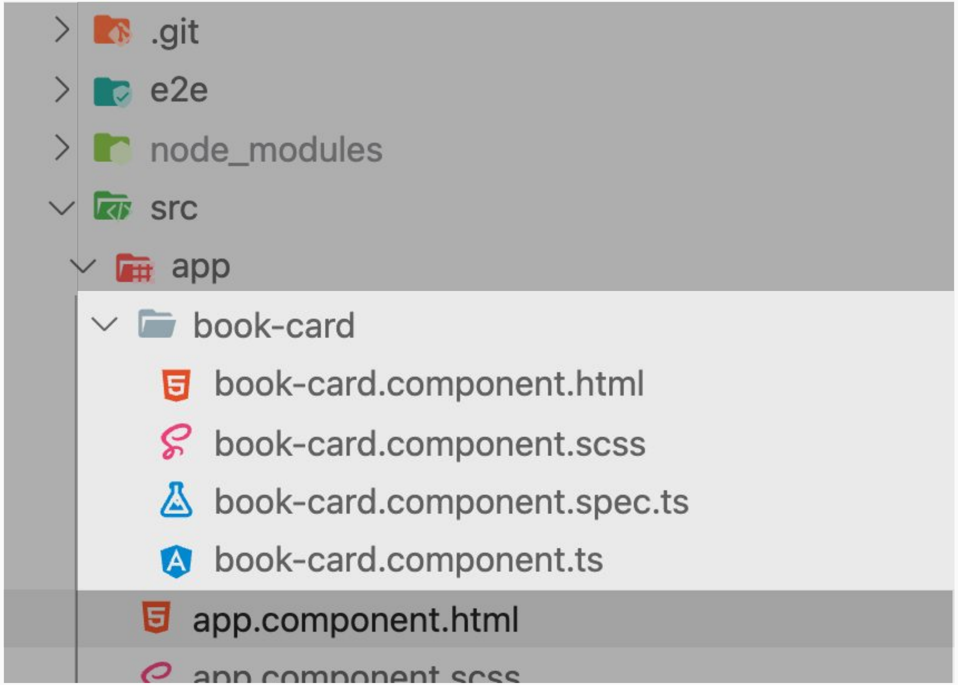
```
ng generate component <component-name>
```

Generate a new component



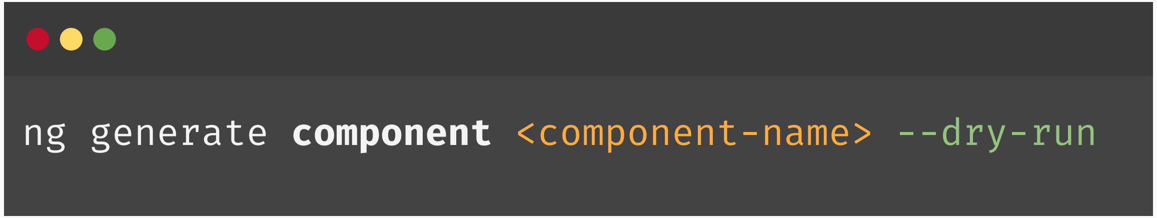
```
± ng generate component book-card

CREATE src/app/book-card/book-card.component.scss (0 bytes)
CREATE src/app/book-card/book-card.component.html (24 bytes)
CREATE src/app/book-card/book-card.component.ts (287 bytes)
```

Generate a new component



```
.git
e2e
node_modules
src
  app
    book-card
      book-card.component.html
      book-card.component.scss
      book-card.component.spec.ts
      book-card.component.ts
    app.component.html
    app.component.scss
```

Generate a new component - Dry Run



```
ng generate component <component-name> --dry-run
```

Link the new component to your app

<code>

app.component.html

```
<app-book-card></app-book-card>
```

Import the new component to your app

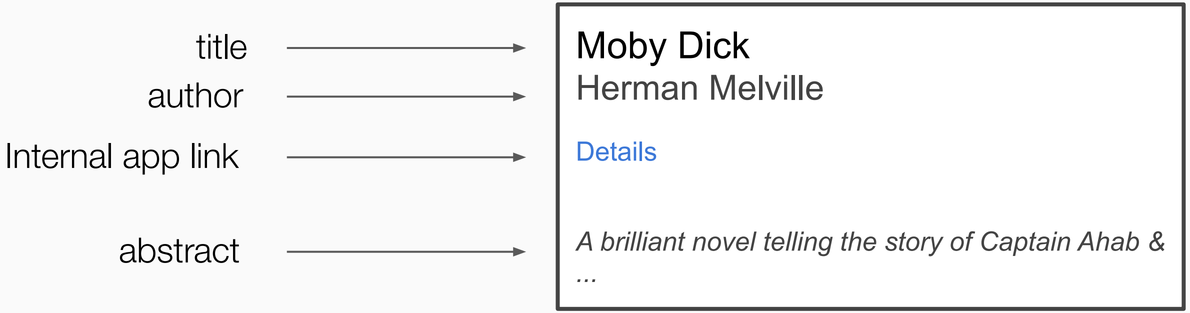
<code>

app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [BookCardComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {}
```

Build your first component

Show some book information to your user.



title

author

Internal app link

abstract

Moby Dick

Herman Melville

Details

A brilliant novel telling the story of Captain Ahab & ...

Task

Your 1st Component

Property-Binding

What is it?

- Write data to an `HTML-Element`.
- Surround an attribute with brackets `[ ]`.
- Finally assign a value.

```
<div [class]="value"></div>
```

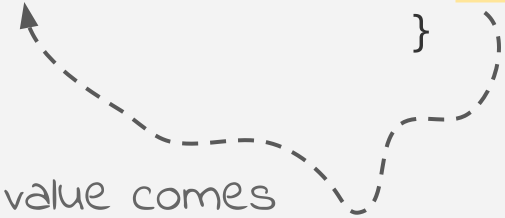
Bind a property of a component

component.html

```
<div

  [class]="value">

</div>
```

component.ts

```
@Component({ /* ... */})
export class Component {
  value = 'red';
}
```



The value comes
from the
corresponding
Component class.

Why [ ]?

- JavaScript’s bracket-syntax serves as model.

```
myObject['property'] = 'red';

divElement['class'] = 'red';
```

Angular adds helpful utilities binding `css classes`, `styles` or any `custom attribute` to an element.

Bind a css class

- Setting a css class conditionally.

component.html

```
<input
   [class.error]="hasError"
   [class.success]="!hasError"
/>
```

component.ts

```
@Component({ /* ... */})
export class Component {
  hasError = true;
}
```

Bind a css style

<code>

component.html

```html
<input

  [style.width.px]="width"

/>
```

component.ts

```ts
@Component({ /* ... */})
export class Component {
  width = 150;
}
```

Bind a custom attribute

<code>

component.html

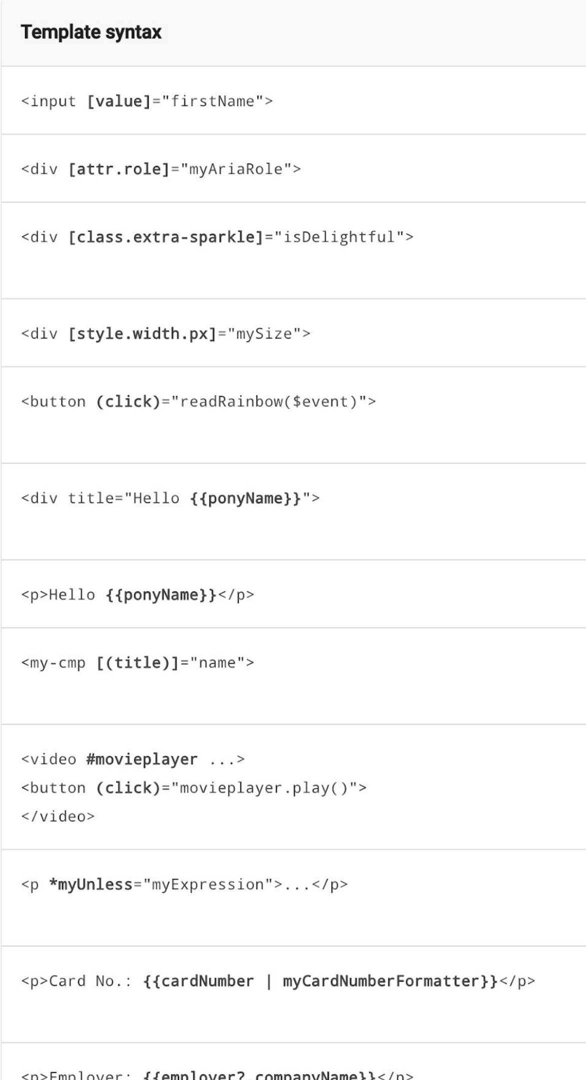
```html
<ul

  [attr.role]="ariaRole">

</ul>
```

component.ts

```ts
@Component({ /* ... */})
export class Component {
  ariaRole = 'navigation';
}
```

Cheat Sheet

- You do not need to remember everything.
- Check out the Angular Cheat Sheet if you want to look up the syntax.
- https://angular.io/guide/cheatsheet



Template syntax

```
<input [value]="firstName">
```

```
<div [attr.role]="myAriaRole">
```

```
<div [class.extra-sparkle]="isDelightful">
```

```
<div [style.width.px]="mySize">
```

```
<button (click)="readRainbow($event)">
```

```
<div title="Hello {{ponyName}}">
```

```
<p>Hello {{ponyName}}</p>
```

```
<my-cmp [(title)]="name">
```

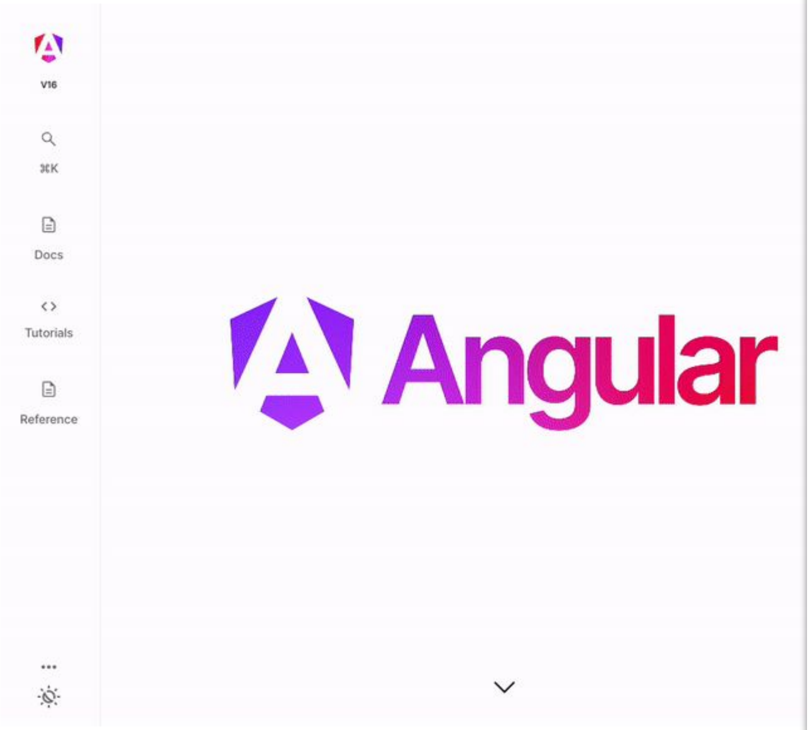
```
<video #movieplayer ...>
<button (click)="movieplayer.play()">
</video>
```

```
<p *myUnless="myExpression">...</p>
```

```
<p>Card No.: {{cardNumber | myCardNumberFormatter}}</p>
```

```
<p>Employer: {{employer?.companyName}}</p>
```

angular.dev



- → "The home for Angular developers"
- → Containing docs, references and interactive tutorials
- → https://angular.dev

Task

Apply a Property-Binding

Lessons learned

Property-Bindings allow to easily pass static or dynamic data to `<HTML-Elements>`.

WHAT IF...?

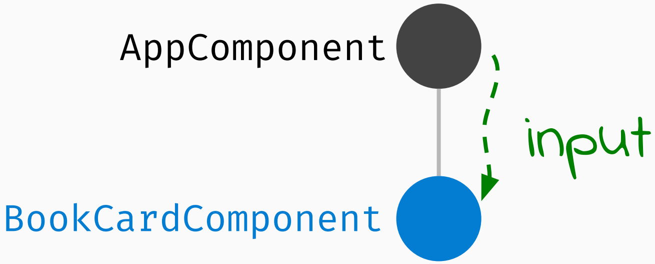
Data needs to be passed to our component?

- We know how to bind data to HTML-Elements!
- What about our component?
- It should be able to receive data, too.

input-Binding

input-Binding

Pass data from a parent component to its children.



Syntax
input-Binding

<code>

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

@Component({ /* ... */ })
export class BookCardComponent {
  content = input();
}
```

Syntax

Bind an input in the template

book-card.component.html

```
<h3>{{ content().title }}</h3>

<h4>{{ content().author }}</h4>

<button>Details</button>

<p>{{ content().abstract }}</p>
```

book-card.component.ts

```
@Component({ /* ... */ })
export class BookCardComponent {
  content = input();
}
```

Syntax

Read input-values in the template

book-card.component.html

```
<h3>{{ content().title }}</h3>
```

The value of the input is read via its "Getter Function".

Syntax

<code>

input can be set via property-binding

Defining component

```
@Component({ /* ... */ })
export class BookCardComponent {
  content = input();
}
```

Calling Component

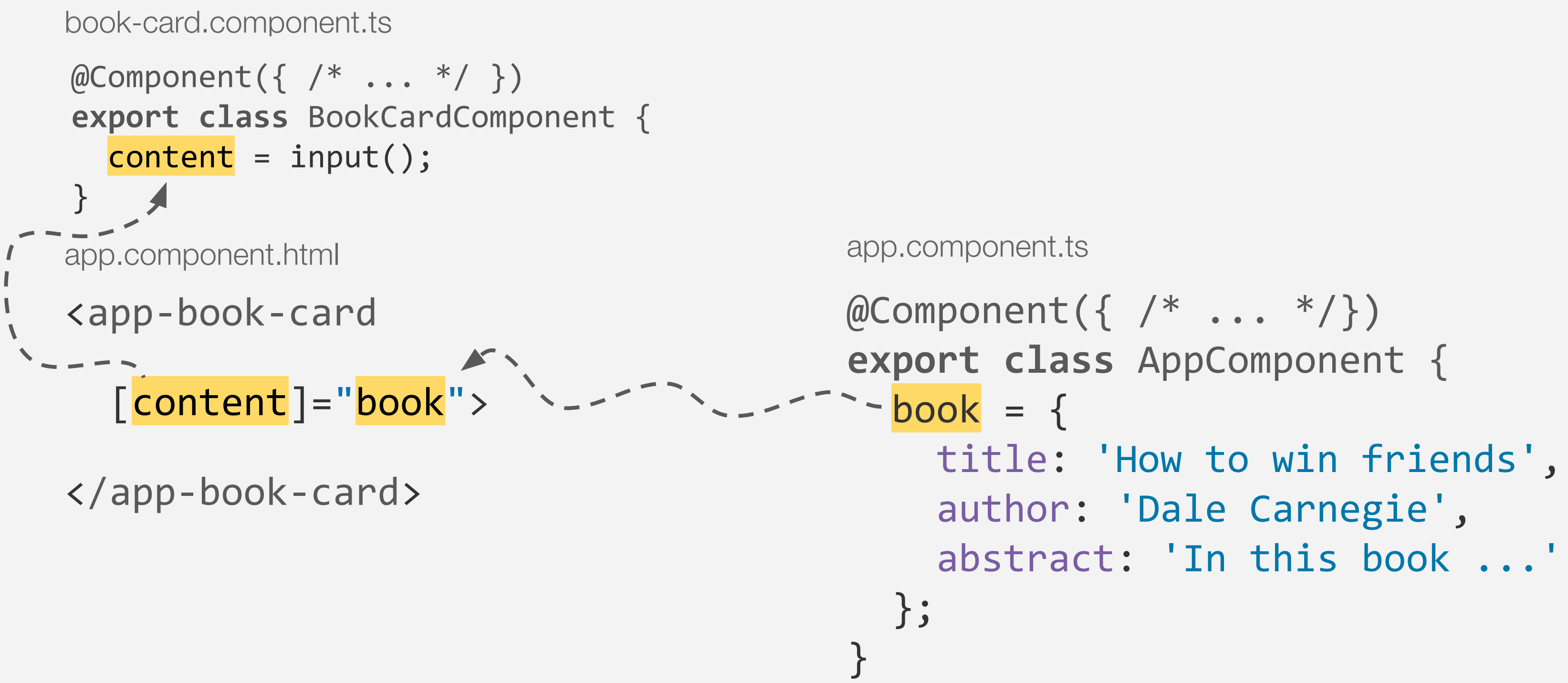
```
<app-book-card

  [content]="book">

</app-book-card>
```

🔧 The Angular Compiler works behind the scenes and creates the attribute for us.

Syntax

Pass data via input

<code>



book-card.component.ts

```
@Component({ /* ... */ })
export class BookCardComponent {
  content = input();
}
```

app.component.html

```
<app-book-card

  [content]="book">

</app-book-card>
```

app.component.ts

```
@Component({ /* ... */})
export class AppComponent {
  book = {
    title: 'How to win friends',
    author: 'Dale Carnegie',
    abstract: 'In this book ...'
  };
}
```

Syntax
The data contract

- The component declaring the `input` defines which data it accepts
- If it does not declare a contract the passed data is treated as `unknown`

Syntax

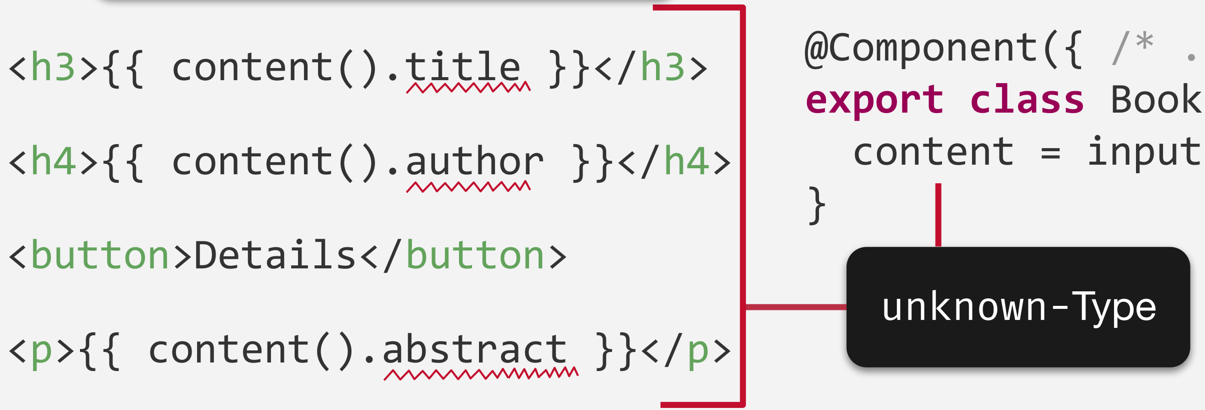
<code>

Missing data contract

book-card.component.html

⚡ Compiler errors

book-card.component.ts



```html
<h3>{{ content().title }}</h3>

<h4>{{ content().author }}</h4>

<button>Details</button>

<p>{{ content().abstract }}</p>
```

```ts
@Component({ /* ... */ })
export class BookCardComponent {
  content = input();
}
```

unknown-Type

Disclaimer any

The next slide demonstrates the use of `any` to define the component’s data contract.

We only do this to enable the next exercise to get familiar with coding and input. Afterwards, you learn about a better approach.

We strictly recommend to avoid the use of `any` in your project.

Syntax

Define a data contract

book-card.component.html

⚠️ You are in “JavaScript-Land”

```
<h3>{{ content().title }}</h3>

<h4>{{ content().author }}</h4>

<button>Details</button>

<p>{{ content().abstract }}</p>
```

book-card.component.ts

```
@Component({ /* ... */ })
export class BookCardComponent {
  content = input<any>();
}
```

any-Type

Task

Apply an `input`-Binding

Lessons learned

An `input-binding`

- is an API to create custom property-bindings
- `needs to declare a data contract`

WHAT IF...?

... we keep using `any` everywhere?

- `any` seems to work…
- What could go wrong?

Question

What happens if someone misspells a property name in `book`?

app.component.html

```
<app-book-card
  [content]="book">
</app-book-card>
```

app.component.ts

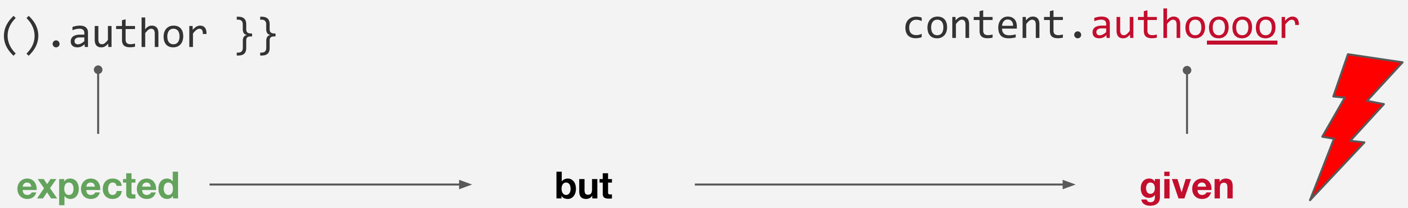
```
@Component({ /* ... */})
export class AppComponent {
  book = {
    title: 'How to win friends',
    authoooor: 'Dale Carnegie',
    abstract: 'In this book ...'
  };
}
```

Answer

The property binding in `app-book-card` stops working.

book-card.component.html

```
<h4>
  {{ content().author }}
</h4>
```



Define a data contract

<code>

Introduce an interface. The blueprint for your model.

```
export interface Book {
  abstract: string;
  author: string;
  title: string;
}
```

Apply the data contract

<code>

Make your code type-safe

```
import { Book } from './book';
```

app.component.ts

```
@Component({ /* ... */})
export class AppComponent {
  book: Book = { /* ... */ };
}
```

book-card.component.ts

```
@Component({ /* ... */})
export class BookCardComponent {
  content = input<Book>();
}
```

book.ts

```
export interface Book {
  abstract: string;
  author: string;
  title: string;
}
```

Declared vs. inferred types

<code>

Even if you say <Book>, you can’t be sure

book-card.component.ts

```
@Component({ /* ... */})
export class BookCardComponent {
  content = input<Book>();
}
```

```
Book | undefined
```

- Defining an input, does not enforce a calling Component to bind data to it.
- We cannot be sure that the data is set.

Syntax

Missing data contract

book-card.component.html

⚡ Compiler errors

```html
<h3>{{ content().title }}</h3>

<h4>{{ content().author }}</h4>

<button>Details</button>

<p>{{ content().abstract }}</p>
```

book-card.component.ts

```typescript
@Component({ /* ... */ })
export class BookCardComponent {
  content = input<Book>();
}
```

possibly undefined

Provide a default value

<code>

An `input` accepts a parameter specifying a default value.

```
@Component({ /* ... */})
export class BookCardComponent {

  content = input<Book>({ title: '', author: '', abstract: '' });

}
```

- We can provide default values to guarantee the presence of data
- But there are various other strategies… 🙋‍♂️

Required inputs

Specify inputs as required

```
@Component({ /* ... */})
export class BookCardComponent {

  content = input.required<Book>();

}
```

- Enforce that a given input must always have a value.
- Angular reports an **error at development-time**, if a component is used without specifying all of its required inputs.

Input transforms

Specify a transform function to change the value of an input

```
@Component({ /* ... */})
export class BookCardComponent {

  title = input('', { transform: trimString });

}

function trimString(value: string | undefined) {

  return value?.trim() ?? '';

}
```

Task

Make an `input`-Binding typesafe

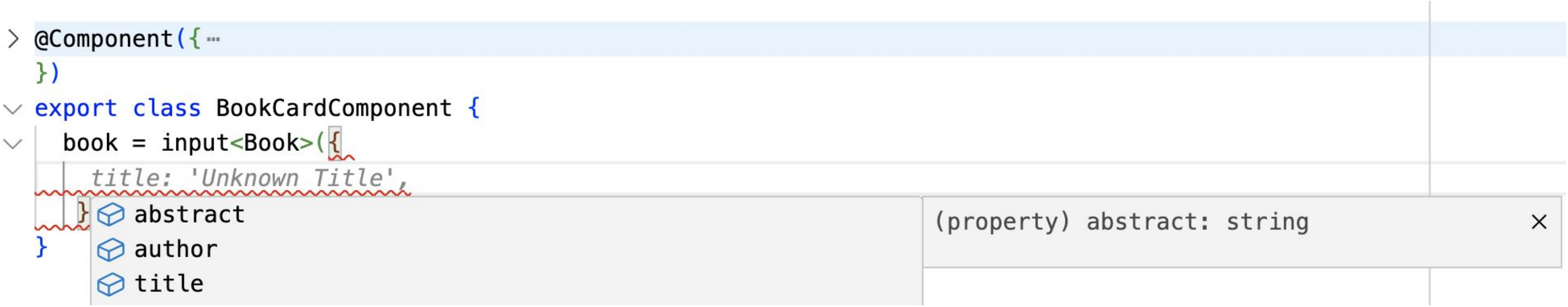
Lessons learned

Angular’s API supports you to build type-safe components.

The more **type-safety** you provide, the more **IDE-support** you will receive.

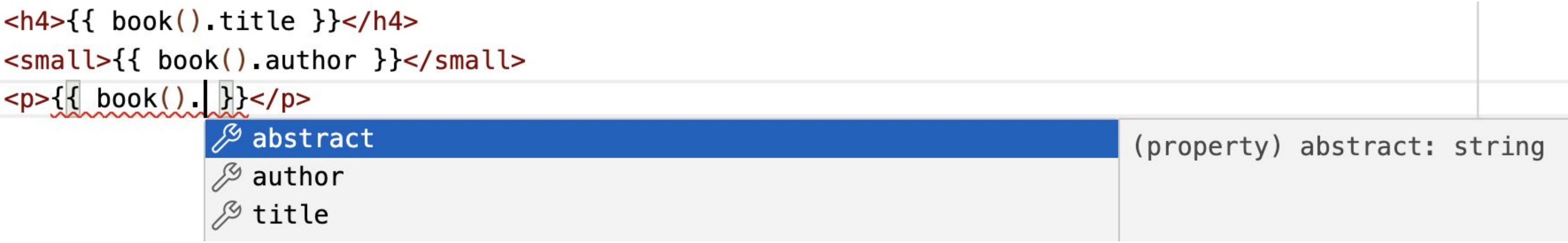
!

TS Better autocompletion in your editor



```
@Component({…
})
export class BookCardComponent {
  book = input<Book>({
    title: 'Unknown Title',
  })
}
```

Autocompletion for templates



```
<h4>{{ book().title }}</h4>
<small>{{ book().author }}</small>
<p>{{ book(). }}</p>
```

You need to have
-Language-Service
installed.

WHAT IF...?

… our components want to communicate a change to its parent?

- Imagine our component made a significant change to the passed data that needs to be reported or persisted by the backend
- How to we trigger those operations?
- We need a way to communicate in the opposite direction!

Events

What is it?

- Listen to an event of an `HTML-Element`.
- Surround an attribute with braces `( )`.
- Finally, assign a component method.

```
<button (click)="componentMethod()"></button>
```

Bind an event to the component

component.html

"Template calls component..."

```
<button
  (click)="componentMethod()">
  Click me!
</button>
```

component.ts

```
@Component({ /* ... */})
export class Component {
  componentMethod() {
    // do something ...
  }
}
```

Access event arguments

component.html

```html
<button

  (click)="componentMethod($event)">

  Click me!

</button>
```

component.ts

```typescript
@Component({ /* ... */})
export class Component {
  componentMethod(eventArgs) {
    // do something with eventArgs
  }
}
```

Why ( )?

- Braces indicate the execution of an operation

```
element.myFunction('parameter')

buttonElement.clickFunction('parameter')
```

Type of the click event

<code>

component.html

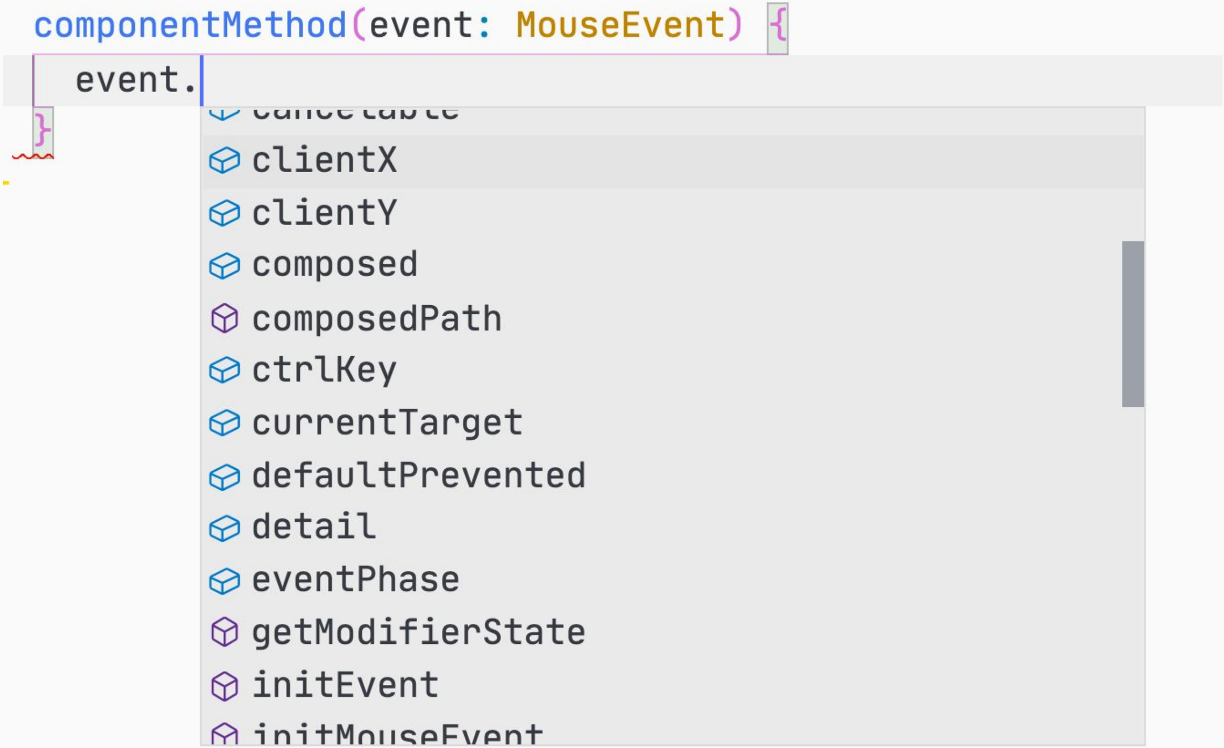
```
<button
  (click)="componentMethod($event)">
  Click me!
</button>
```

component.ts

```
@Component({ /* ... */})
export class Component {
  componentMethod(eventArgs: MouseEvent) { }
}
```

Type is specified by Angular.

Type of the click event



```
componentMethod(event: MouseEvent) {
  event.
}
```

Type of the input event

component.html

```html
<input
  (input)="componentMethod($event)">
  Click me!
</input>
```

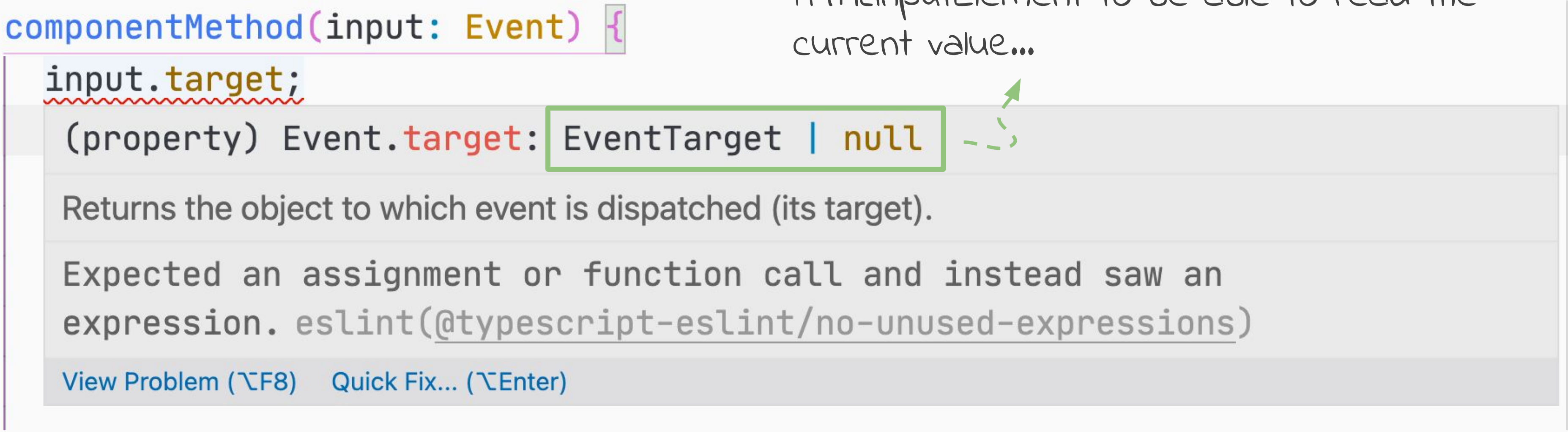
component.ts

```typescript
@Component({ /* ... */})
export class Component {
  componentMethod(eventArgs: Event)
  {
    const inputElement = eventArgs.target;
  }
}
```

General type **Event** is specified by Angular.

Type of the input event

Unfortunately, the target is not typed as HTMLInputElement to be able to read the current value...



```
componentMethod(input: Event) {
  input.target;
```

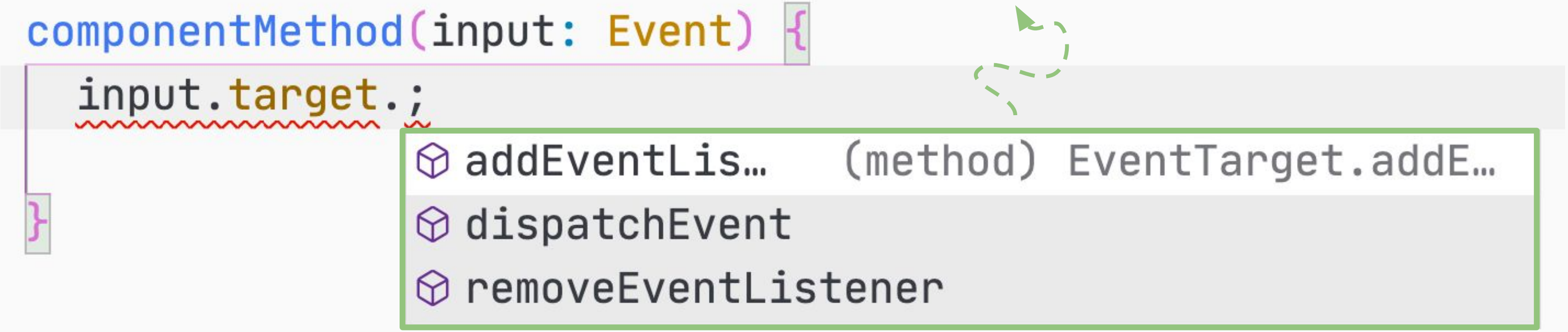
```
(property) Event.target: EventTarget | null
```

Returns the object to which event is dispatched (its target).

```
Expected an assignment or function call and instead saw an
expression. eslint(@typescript-eslint/no-unused-expressions)
```

Type of the input event

we just get very basic DOM-APIs here, but since the event is coming from **HTMLInputElement** we should be able to access more...



```
componentMethod(input: Event) {
  input.target.;
}
```

Type of the input event

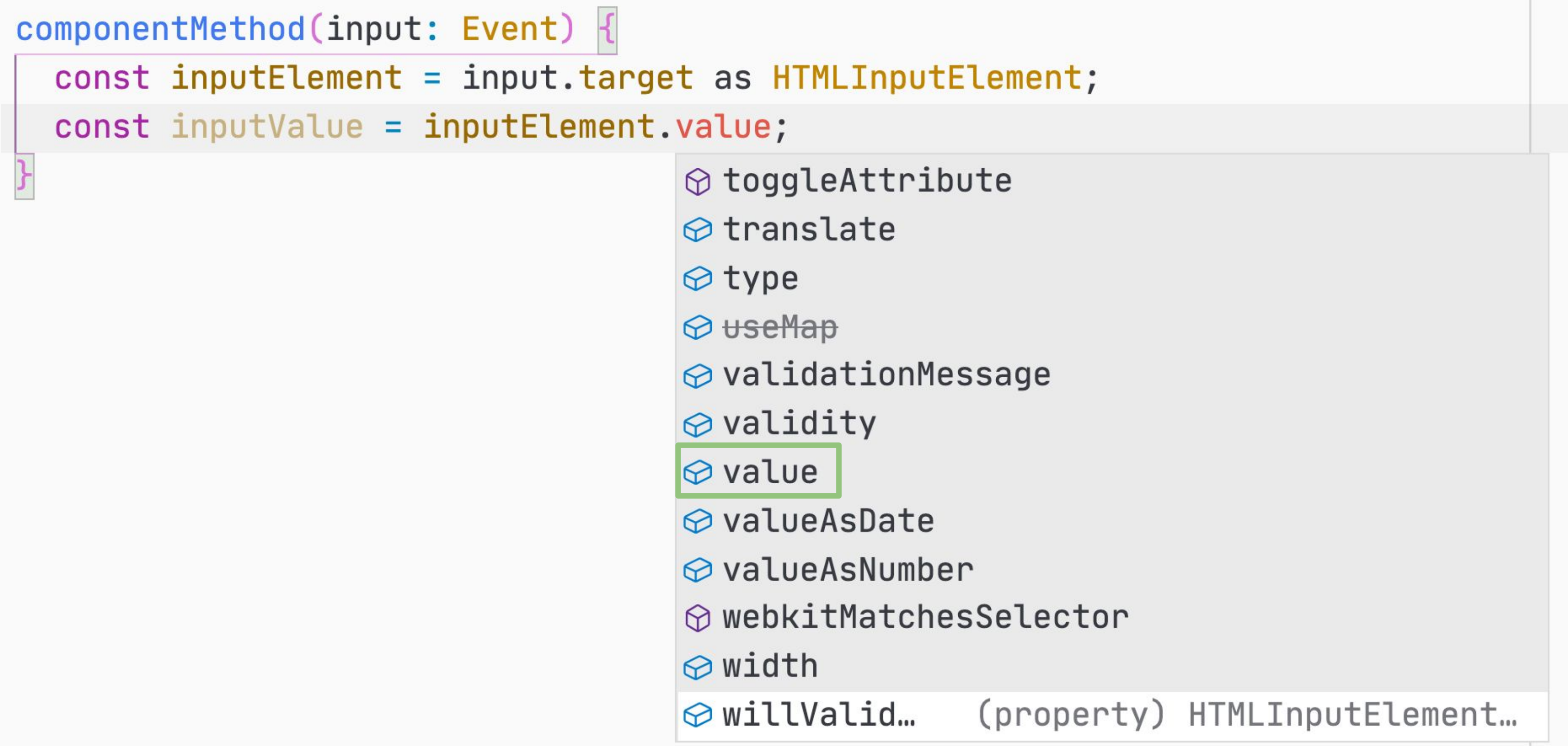
<code>

component.ts

```
@Component({ /* ... */})
export class Component {
  componentMethod(eventArgs: Event)
  {
    const inputElement = eventArgs.target as HTMLInputElement;
    const inputValue = inputElement.value;
  }
}
```

To have more **type support** you can help the TypeScript Compiler to be more concrete.

Type of the input event



```
componentMethod(input: Event) {
  const inputElement = input.target as HTMLInputElement;
  const inputValue = inputElement.value;
}
```

Task

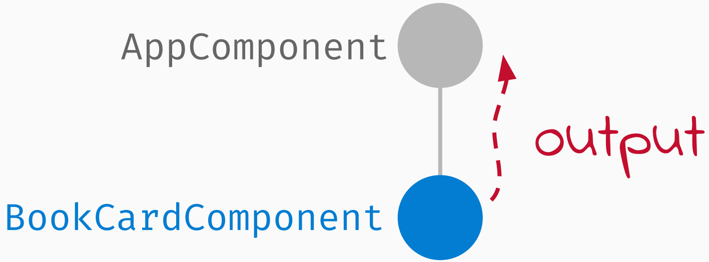
Apply an Event-Binding

output-Binding

Instead of only listening to predefined DOM-Events, we can create them on our own. They are called **`output-Bindings`**.

output-Binding

Pass data from child- to its a parent component.



Declare an output-Binding

<code>

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

@Component({ /* ... */ })
export class BookCardComponent {
  detailClick = output<Book>();
}
```

Apply an `output`-Binding

<code>

book-card.component.html

```
<a

  (click)="handleDetailClick()"

>

  Details

</a>
```

book-card.component.ts

```
@Component({ /* ... */ })
export class BookCardComponent {
  content = input.required<Book>();
  detailClick = output<Book>();

  handleDetailClick(): void {
    this.detailClick.emit(this.content());
  }
}
```

Use an output-Binding

<code>

app.component.html

```html
<app-book-card

  (detailClick)="navigate($event)">

</app-book-card>
```

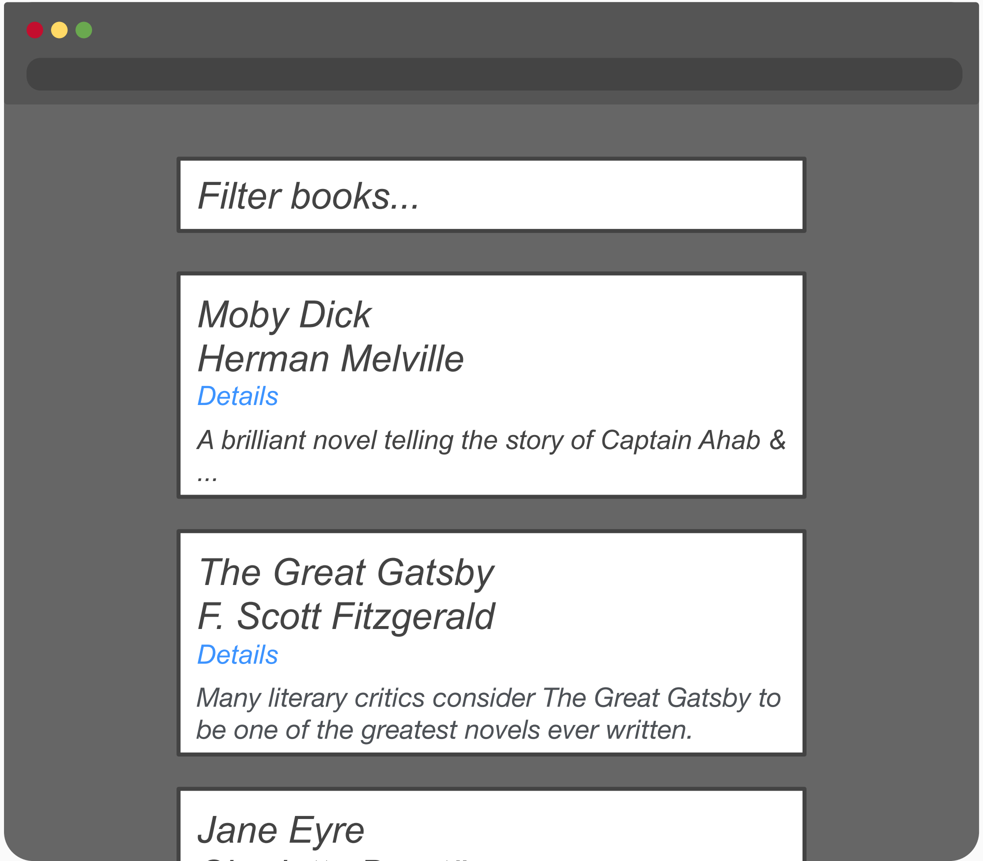
app.component.ts

```typescript
@Component({ /* ... */})
export class AppComponent {
  navigate(book: Book) {
    // navigate somewhere ...
  }
}
```

Task

Apply an output-Binding

What are you going to build?



Filter books...

Moby Dick
Herman Melville
Details
A brilliant novel telling the story of Captain Ahab & ...

The Great Gatsby
F. Scott Fitzgerald
Details
Many literary critics consider The Great Gatsby to be one of the greatest novels ever written.

Jane Eyre

Built-in Control Flow Syntax

New control flow syntax

- New build-in syntax for conditions and loops
- Syntax is based on a user survey
- "Built-in" nature makes optimizations during the “build” phase possible and allows new features (e.g. "defer")
- Stable since Angular 18

@if

<code>

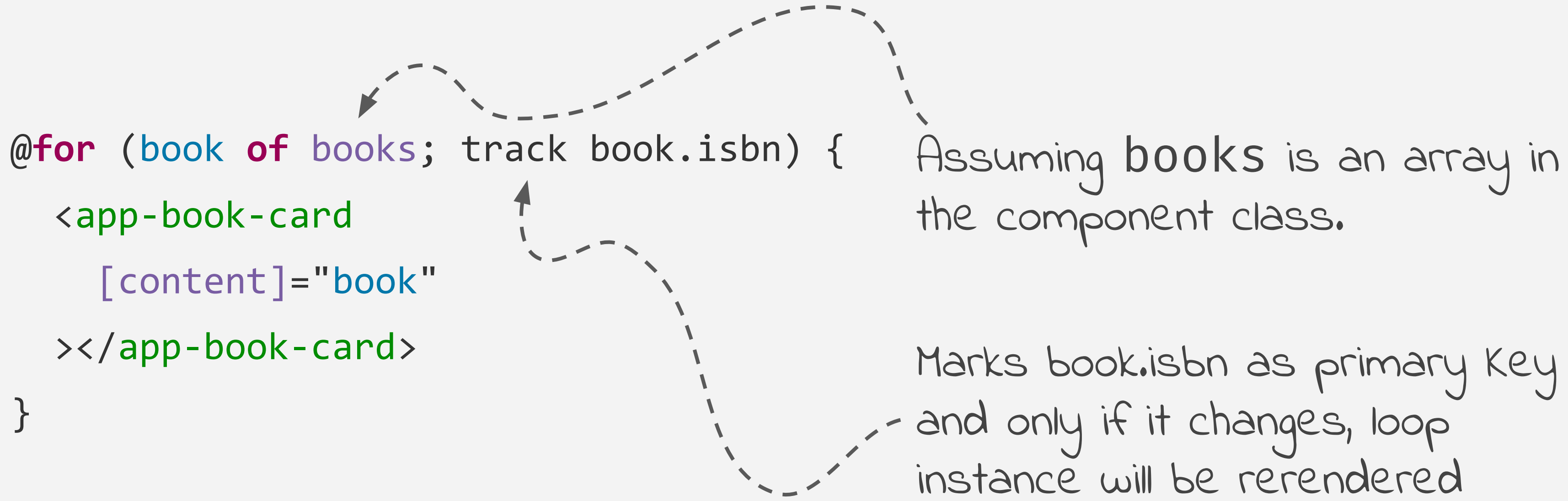
- Renders element if condition is **truthy**.

```
@if (content.abstract) {
  <p>{{ content.abstract }}</p>
}
```

@for

<code>

- Iterates over a list and renders each item.



```
@for (book of books; track book.isbn) {
  <app-book-card
    [content]="book"
  ></app-book-card>
}
```

Assuming **books** is an array in the component class.

Marks book.isbn as primary Key and only if it changes, loop instance will be rerendered

@for

<code>

Attributes

```
@for (book of books; track book.isbn) {
  @if ($first) { Total: {{ $count }} <hr/> }
  {{ $index }} {{ book.title }}
  <!-- {{ $even }}
  {{ $odd }} -->
  @if ($last) { <app-footer /> }
}
```

`@empty` for loops

<code>

Displays content when there are no items

```
@for (book of books; track book.isbn) {
  <app-book-card
    [content]="book"
  ></app-book-card>
} @empty {
  <li> There are no items.</li>
}
```

@switch block - selection

Syntax for switch is inspired by the JavaScript switch statement

```
@switch (condition) {
  @case (caseA) {
    Case A.
  }
  @case (caseB) {
    Case B.
  }
  @default {
    Default case.
  }
}
```

New feature: @defer

Defer the loading of select dependencies within a template

```
@defer (on hover(greeting)) {
  <app-book-card />
} @placeholder {
  <div>Book Card placeholder</div>
}
```

- @defer is really powerful. It is possible to specify loading- and error sections
- There are also a lot of different triggers

Structural Directives (Legacy)

What is it?

- Structural Directives are attributes starting with an asterisk *.
- The concept of structural directives itself will not be deprecated:

  Custom structural directives can still be created

```html
<button *ngIf="condition">Click me!</button>
```

`@if` block conditionals

<code>

Displays its content when its condition expression is truthy

```
// New build-in @if syntax
@if (book) {
  <!-- Content -->
} @else {
  <!-- No Data Content -->
}
```

```
// *ngIf syntax
 <ng-container
   *ngIf="book; else noDat">
   <!-- Content -->
 </ng-container>

 <ng-template #noDat>
   <!-- No Data Content -->
 </ng-template>
```

@for block - loops

<code>

Renders content for each item in a collection

```
// New build-in @for syntax
@for (book of books; track book.isbn) {
  <app-book-card
    [content]="book"
  ></app-book-card>
}
```

```
// *ngFor syntax
<app-book-card
  [content]="book"
  *ngFor="let book of books"
 ></app-book-card>
```

Whenever you read the shorthand `ng`…

It stands for A`ng`ular.

*ngIf

<code>

- Renders element if condition is **truthy**.

```
<p *ngIf="content.abstract">{{ content.abstract }}</p>
```

*ngFor

- Iterates over a list and renders each item.

```
<app-book-card
  [content]="book"
  *ngFor="let book of books">
</app-book-card>
```

Assuming **books** is an array in the component class.

Task

Show a list of Books

Pipes

Next to components there is another concept that makes template syntax and data binding even more powerful.

Let’s take a look at `Pipes`.

What is it?

- Transform data that is bound in the template.
- Filter data if it is passed as collection.
- Is used by writing the |-symbol.

```
<element> {{ birthday | date }} </element>
```

What is it?

- Transform data that is bound in the template.
- Filter data if it is passed as collection.
- Is used by writing the | -symbol.

```
<element> {{ birthday | date }} </element>
```

Pipe-Name

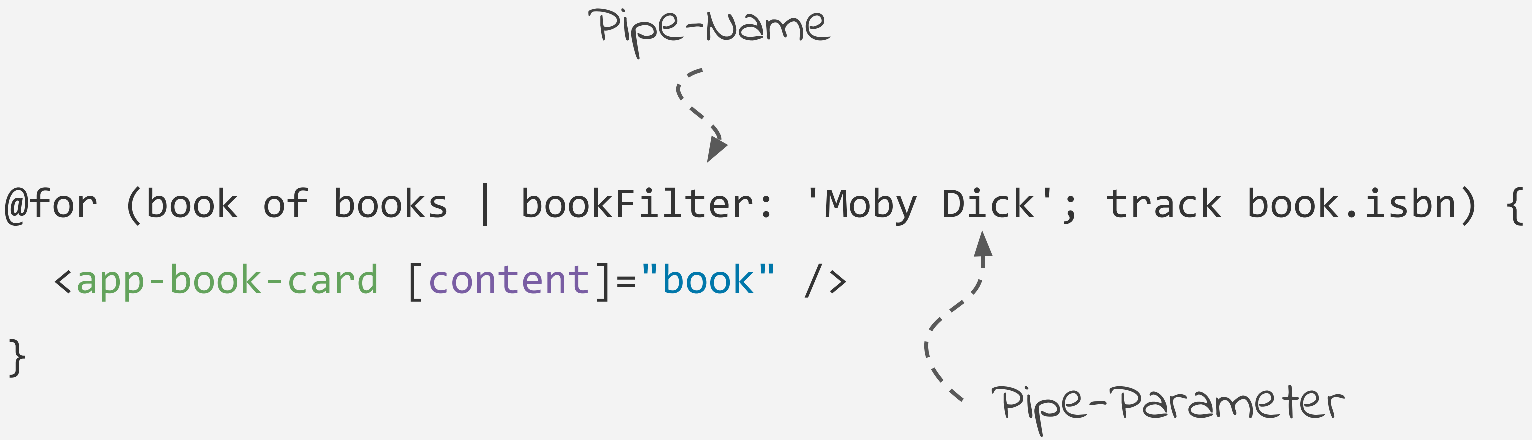
Built-in Pipes

- AsyncPipe
- CurrencyPipe
- DatePipe
- DecimalPipe
- JsonPipe
- KeyValuePipe
- LowerCasePipe
- PercentPipe
- SlicePipe
- UpperCasePipe

Use pipes with structure syntax

<code>

- Iterates over a list and renders each item.



Pipe-Name

```
@for (book of books | bookFilter: 'Moby Dick'; track book.isbn) {
  <app-book-card [content]="book" />
}
```

Pipe-Parameter

Built-In Pipes

<code>

Example: date

app.component.html

```html
<span>
  {{ now | date }}
</span>
```

Formats date based on configured locale.

app.component.ts

```typescript
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.scss']
})
export class AppComponent {
 now = new Date();
}
```

Built-In Pipes

<code>

Example: date with format-string

app.component.html

```html
<span>
  {{ now | date: 'dd.MM.yyyy' }}
</span>
```

Format string can be overridden with a pipe-parameter.

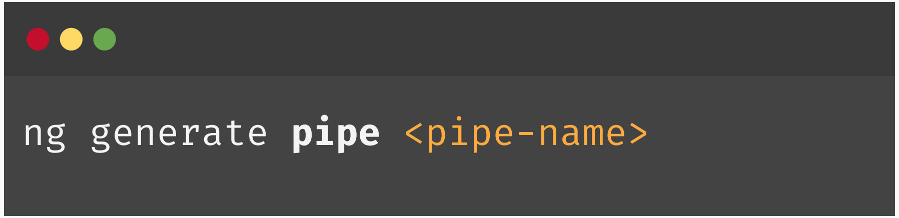
app.component.ts

```typescript
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.scss']
})
export class AppComponent {
 now = new Date();
}
```

Not the Pipes you are looking for …?

Luckily, we can write our own.

Generate a new pipe



```
ng generate pipe <pipe-name>
```

Pipe

<code>

- Generated Boilerplate

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'bookFilter',
  standalone: true
})
export class BookFilterPipe implements PipeTransform {
  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

Pipe

- PipeTransform interface

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'bookFilter',
  standalone: true
})
export class BookFilterPipe implements PipeTransform {
  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

Pipe

- Use the force of type information

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'bookFilter',
  standalone: true
})
export class BookFilterPipe implements PipeTransform {
  transform(books: Book[], searchTerm: string): Book[] {
    // filter your books ...
  }

}
```

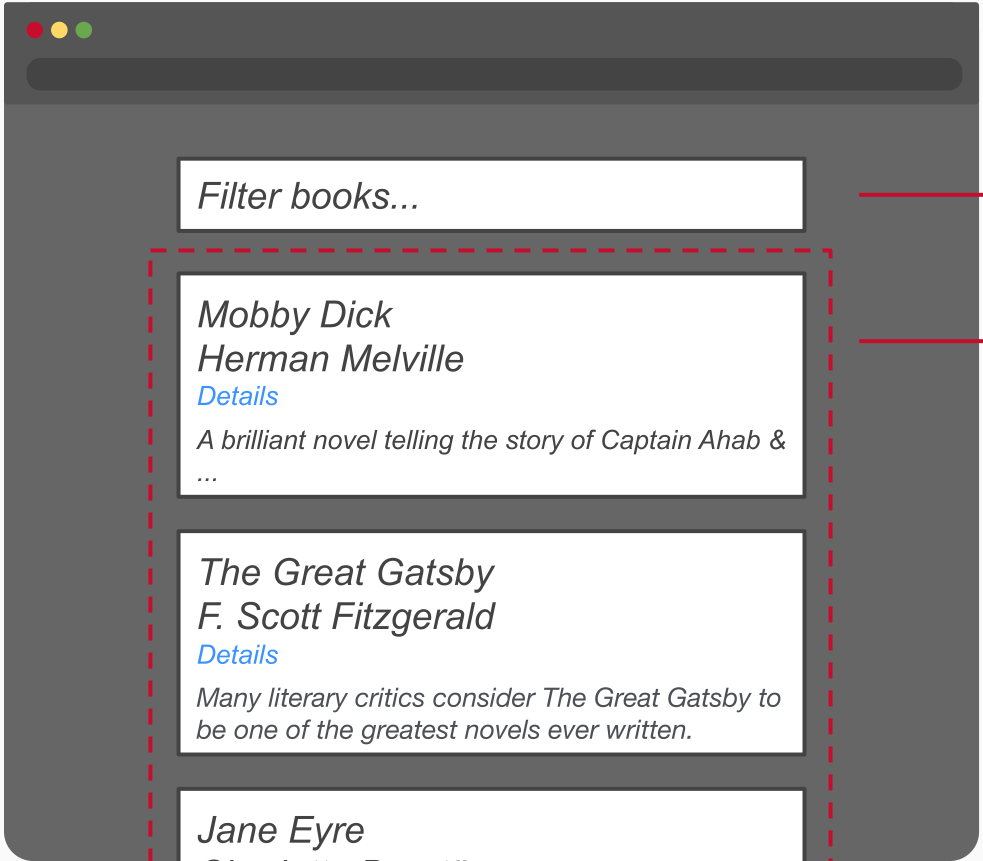
Pipe | Usage

<code>

```
@for (book of books | bookFilter: bookSearchTerm; track book.isbn) {
  <app-book-card [content]="book" />
}
```

... is a property in the component class.

What are you going to build?



Filter books...

Mobby Dick
Herman Melville
Details

A brilliant novel telling the story of Captain Ahab & ...

The Great Gatsby
F. Scott Fitzgerald
Details

Many literary critics consider The Great Gatsby to be one of the greatest novels ever written.

Jane Eyre

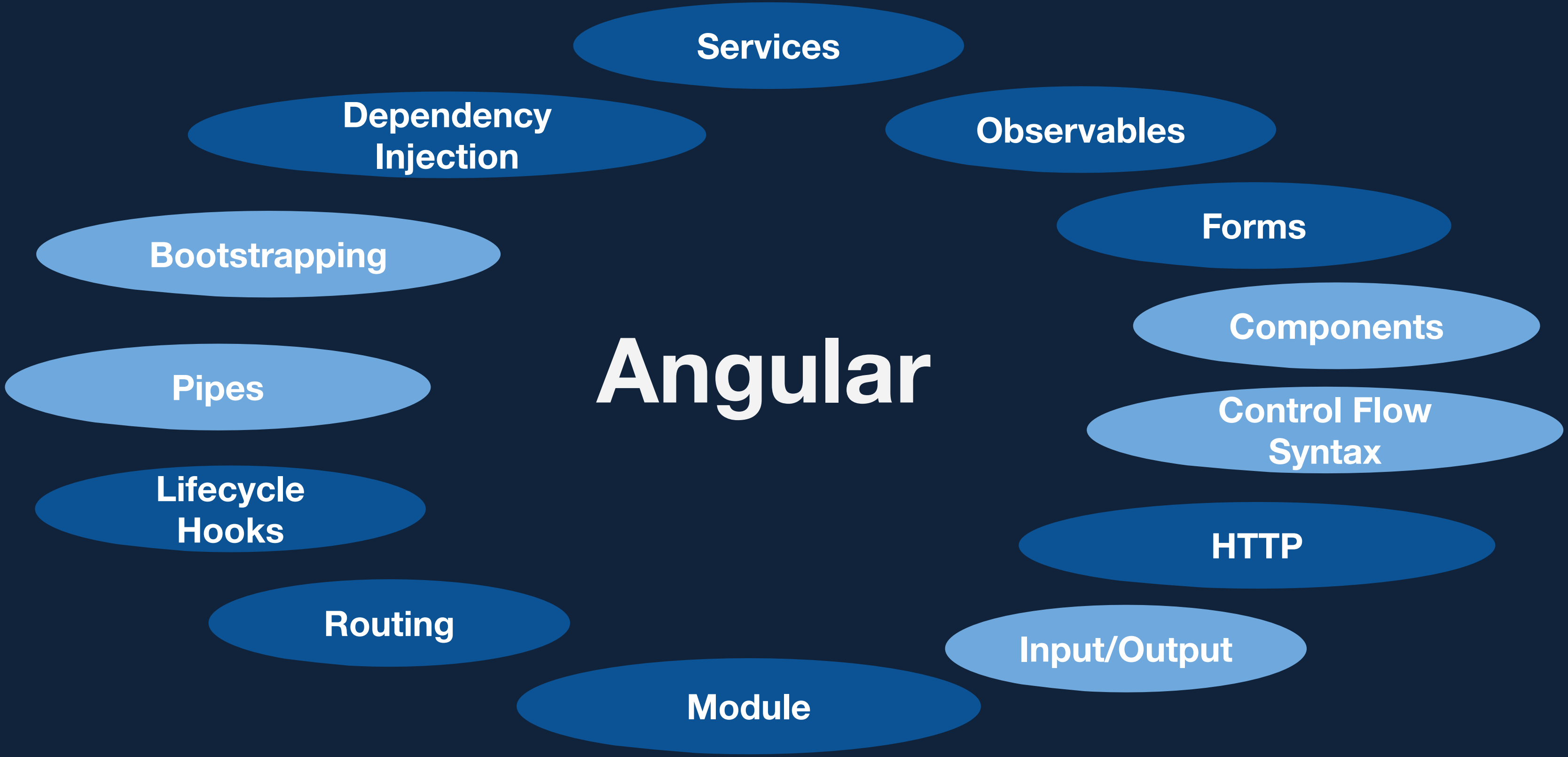
`<input>` updates `bookSearchTerm`

```
@for(”... |
        bookFilter:bookSearchTerm”
```

Task

Filter books

Repeat what you have learned...



Angular

Services

Observables

Forms

Components

Control Flow Syntax

HTTP

Input/Output

Module

Routing

Lifecycle Hooks

Pipes

Bootstrapping

Dependency Injection

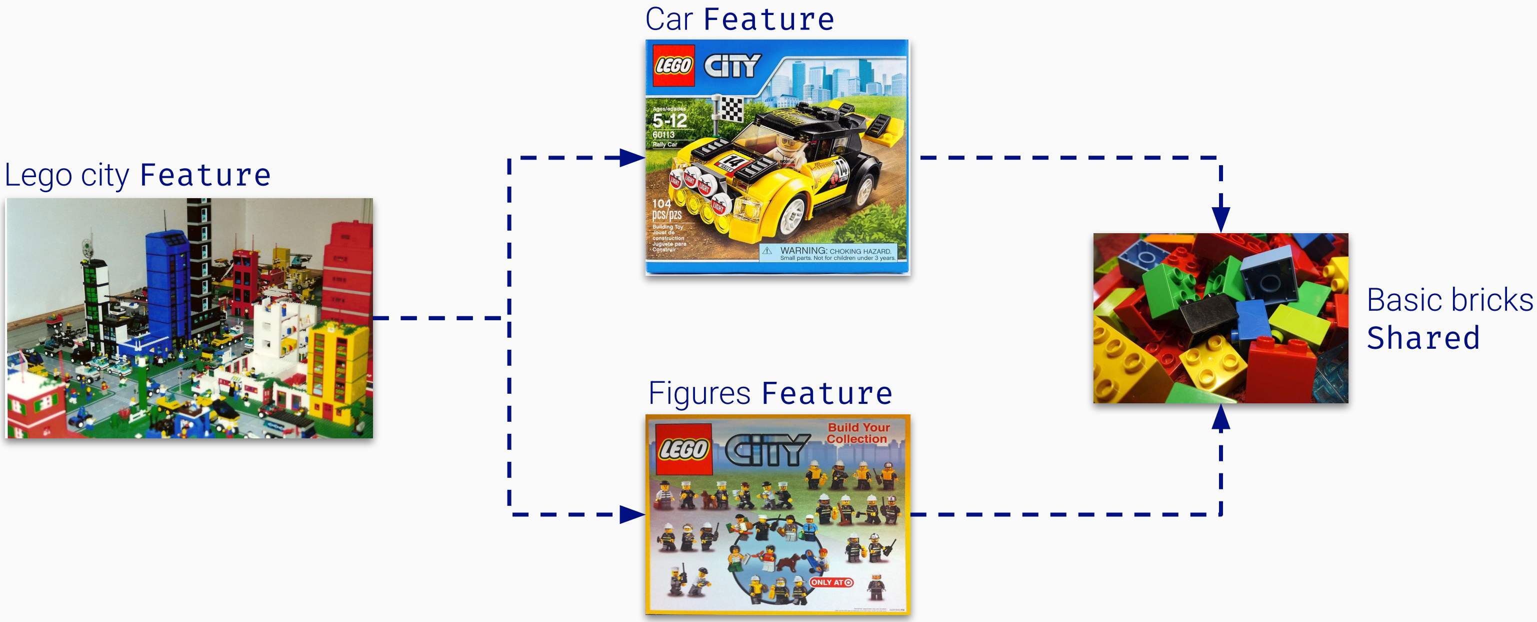
Structure App Code

Structure App Code

- Source Code should be structured
- Split code into reusable, mostly independent parts
- High cohesion, low coupling
- Organize Angular App Code in separate folders

Images
from wikimedia commons
by Semevent from Pixabay
by brickresort from flickr (CC BY-ND 2.0)
by wiredforlego from flickr (CC BY-SA 2.0)

Think in features



Car Feature

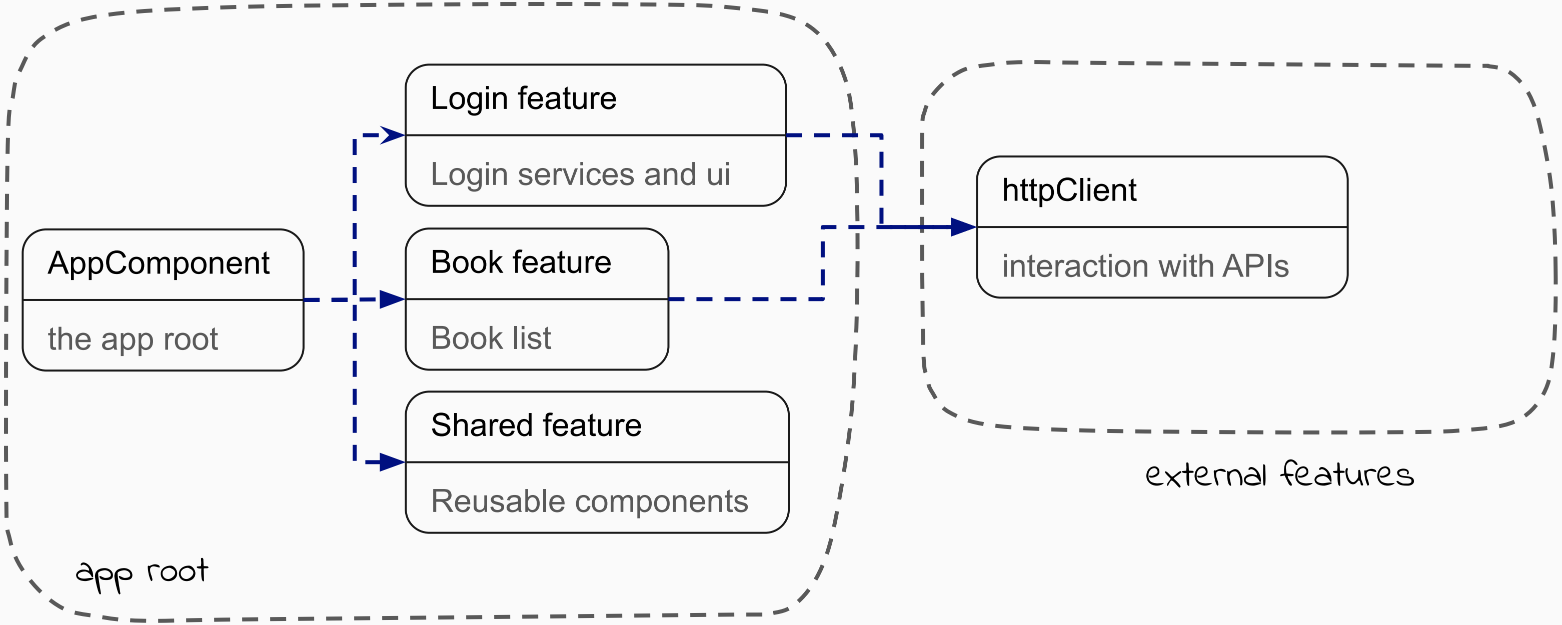
Lego city Feature

Figures Feature

Basic bricks
Shared

Think in features

Organize aspects (or **features**) of an Angular app:



Task

Refactor and structure project

Legacy: @NgModule

Angular Modules / @NgModule

- Old way of organizing and structuring Angular Apps
- Used to declare Components, Service Providers and dependencies to other modules
- Angular Compiler was lagging behind (still from the AngularJS time)
- Not needed anymore in Standalone Apps (Angular 17++)

Angular 2+: @NgModule Decorator

Attention: Legacy Code!

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

Specifies the startup component

Using Legacy Modules

<code>

Import Components from Legacy Modules

```
@Component({
  selector: 'app-book-card',
  standalone: true,
  imports: [MatButtonModule],
  templateUrl: './book-card.component.html',
  styleUrl: './book-card.component.scss'
})
export class BookCardComponent {
  ...
}
```

Modules can simply be imported in Standalone Components

Bootstrapping Standalone Apps

Bootstrapping

Your application needs a starting point!

- main.ts

Bootstrapping an Angular App

<code>

```
// main.ts

import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch(err => console.error(err));
```

Standalone Component
used as Root Component

Referencing App Config

<code>

```
// main.ts

import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch(err => console.error(err));
```

App Config

<code>

Place for registering service providers

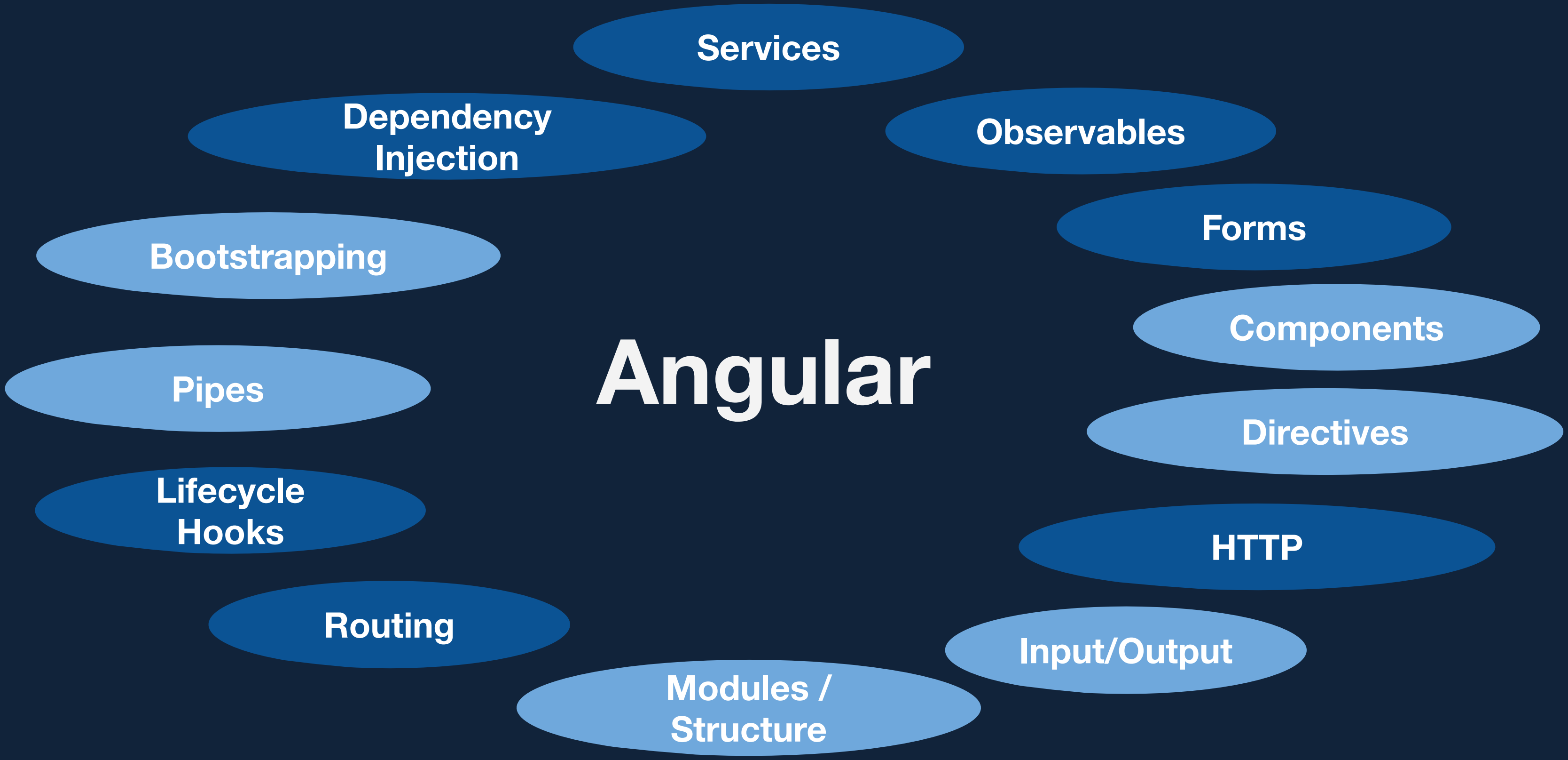
```
// app/app.config.ts
import { ApplicationConfig } from '@angular/core';

export const appConfig: ApplicationConfig = {
  providers: []
};
```

Import Providers from Modules

<code>

```
// app/app.config.ts
import { ApplicationConfig, importProvidersFrom } from '@angular/core';
import { BrowserAnimationsModule } from ...

export const appConfig: ApplicationConfig = {
  providers: [importProvidersFrom(BrowserAnimationsModule)]
};
```

Repeat what you have learned...



Angular

Services

Observables

Forms

Components

Directives

HTTP

Input/Output

Modules / Structure

Routing

Lifecycle Hooks

Pipes

Bootstrapping

Dependency Injection

Services

Services

- (Local) Singletons
- Data- / Business-Layer of our application
- May be injected via Dependency Injection (DI)
- Roles:
  - Provide methods or streams of data to subscribe to
  - Provide operations to modify data
  - Manage State
  - Encapsulate Browser operations or algorithms

Services - Example

Services are the Data-Model-Layer of our application

```
@Injectable({
  providedIn: 'root'
})
export class BookApiService {
  private books: Book[] = [{...}, {...}, {...}];

  getAll() {
    return this.books;
  }
}
```

Services - Example

<code>

They define an API to interact with them

```
@Injectable({
  providedIn: 'root'
})
export class BookApiService {
  private books: Book[] = [{...}, {...}, {...}];

  getAll() {
    return this.books;
  }
}
```

Services - Example

With `providedIn:'root'` the service is registered globally

```
@Injectable({
  providedIn: 'root'
})
export class BookApiService {
  private books: Book[] = [{...}, {...}, {...}];

  getAll() {
    return this.books;
  }
}
```

Services

<code>

Create a service instance for a component and its children

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


@Component({
  // ...
  providers: [BookApiService]
})
export class BookComponent {
  private readonly bookApi = inject(BookApiService)
}
```

Dependency Injection

Dependency Injection - Why

- Keep component classes clean
- Better testable code
- Easy replacement of services

Without Dependency Injection

Dependency Injection

<code>

You have to create instances on your own.

```
@Component({ … })
class BookComponent {

  private bookApiService: BookApiService;

  constructor() {
    const httpClient = new HttpClient(...)
    this.bookApiService = new BookApiService(httpClient);
  }
}
```

With Dependency Injection

Dependency Injection

Giving control to a
framework to get
an instance of the object.

Dependency Injection is a form of **Inversion of control**.

way of injecting properties to an
object is called Dependency Injection
through **Constructor** , **Setter /Getter**,
Interface .

Dependency Injection

<code>

Injection and Singleton Services

```
@Injectable({
  providedIn: 'root'
})
export class BookApiService { }

@Component({...})
class BookComponent {

  private readonly bookApi = inject(BookApiService)

}
```

Dependency Injection - @Injectable()

- Annotation of classes that use DI
- Metadata to compile the type-information to the JS code
- Without an annotation you lose the type information

Dependency Injection

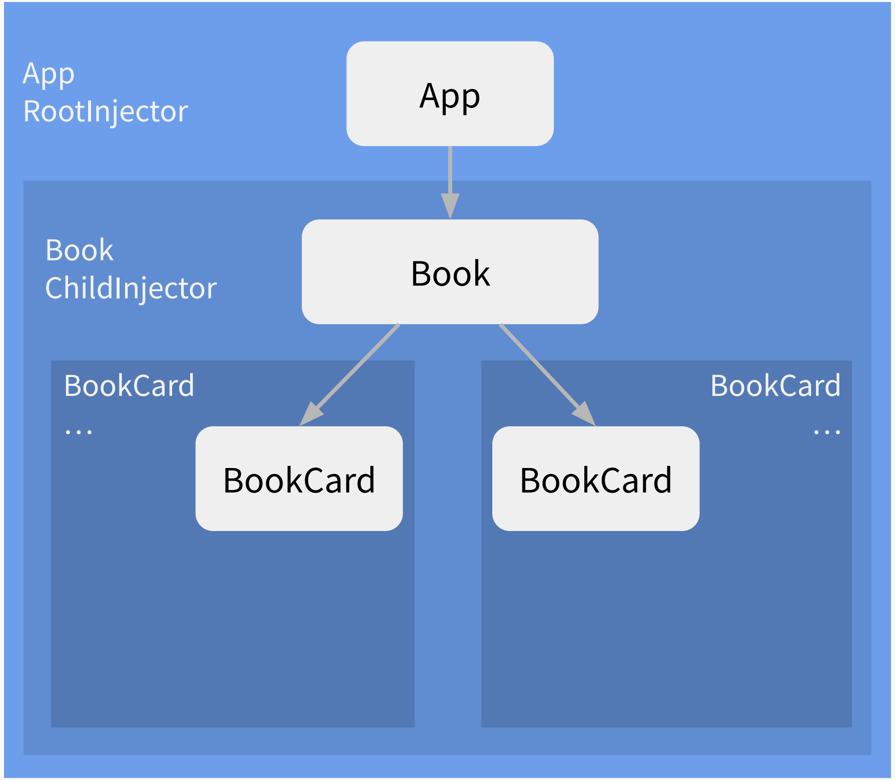
<code>

"Local" Singletons = Multiple Instances per injector

```
@Component({
  providers: [BookApiService]
})
class BookComponent {

  private readonly bookApi = inject(BookApiService)

}
```

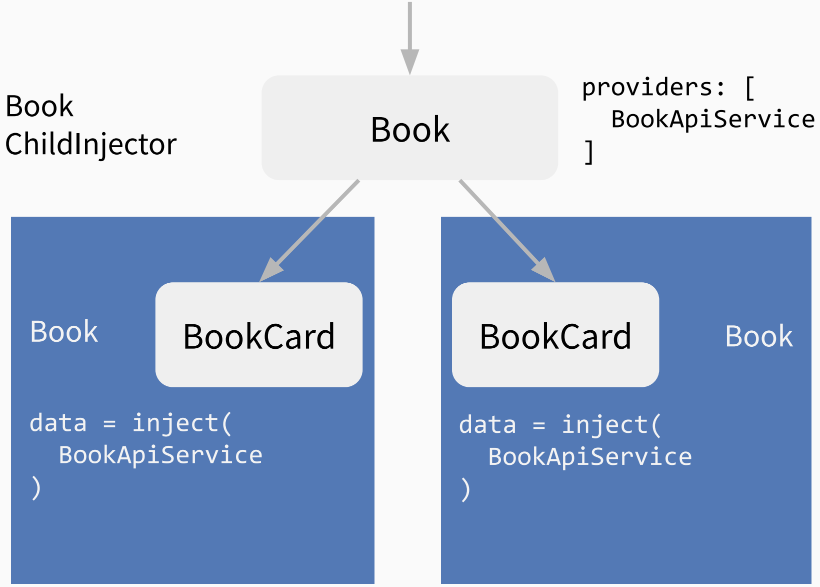
Dependency Injection

- **Injector** per component
- Each component has an own injector
- Base injector = **RootInjector**
- Each nested component has a **ChildInjector**



Dependency Injection

- Share one service instance
- Create instance in parent component Book
- Only inject service in BookComponent → no providers!



Book
ChildInjector

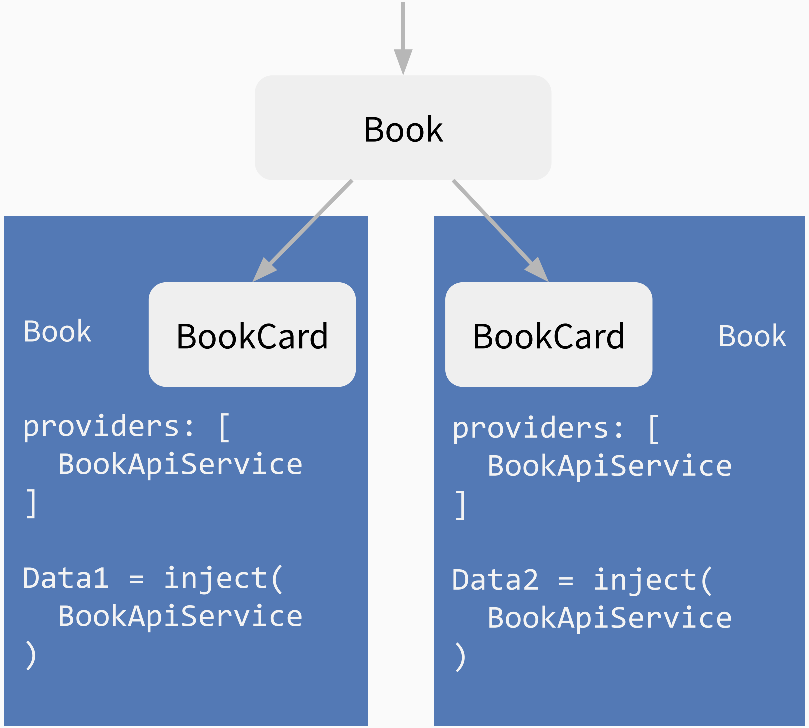
```
providers: [
  BookApiService
]
```

```
data = inject(
  BookApiService
)
```

```
data = inject(
  BookApiService
)
```

Dependency Injection

- New service instance for each BookCardComponent

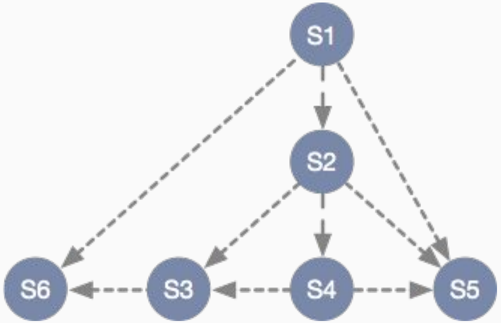


```
providers: [
  BookApiService
]

Data1 = inject(
  BookApiService
)
```

```
providers: [
  BookApiService
]

Data2 = inject(
  BookApiService
)
```

Dependency Injection

- Services can have dependencies, too
- Injects service instances created in a component, where service is used!
- Watch out for dependency cycles!

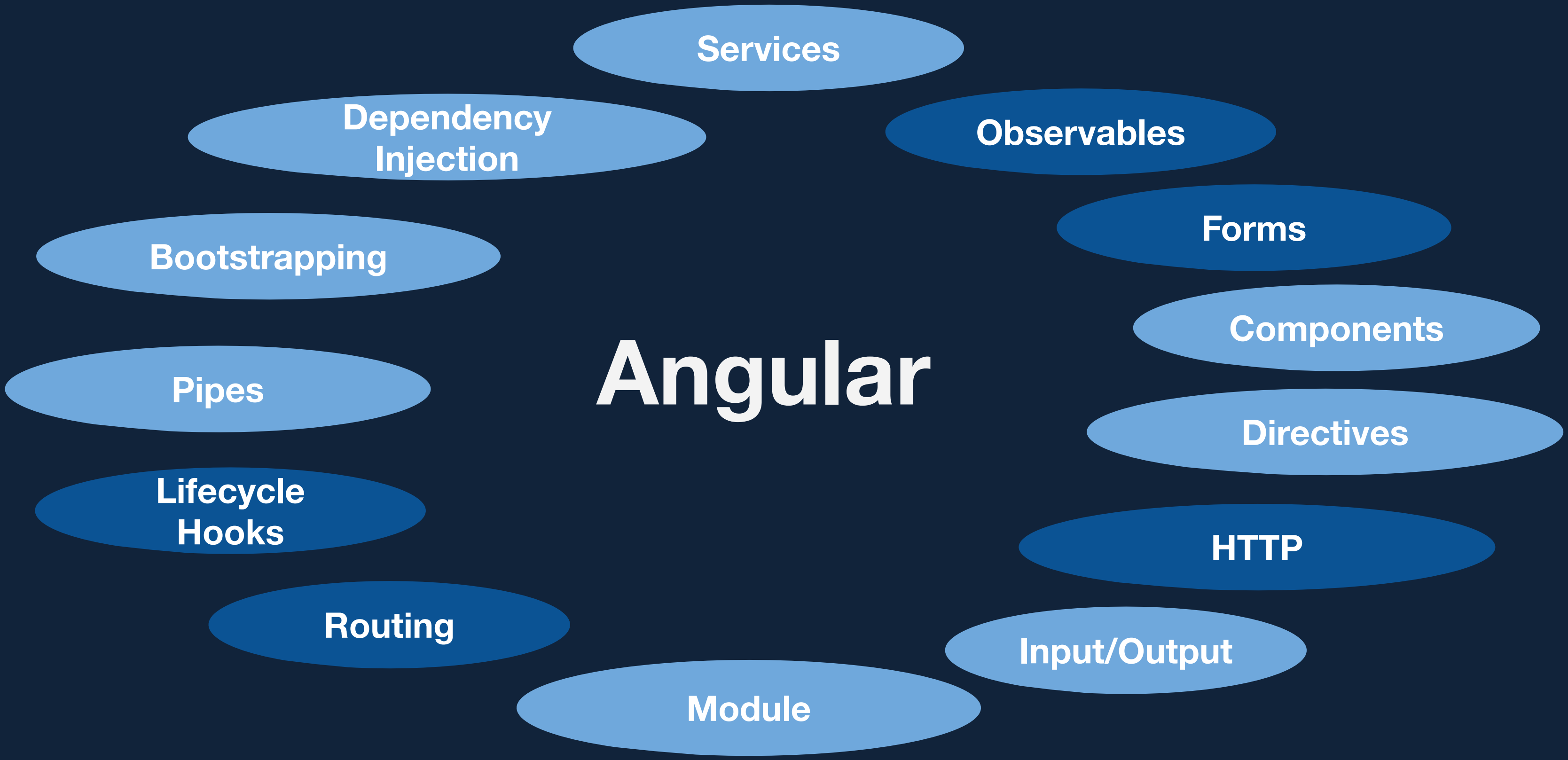


```
service 'S1', (S2, S5, S6)
service 'S2', (S3, S4, S5)
service 'S3', (S6)
service 'S4', (S3, S5)
service 'S5', ()
service 'S6', ()
```

Task

Create a BookApi service

Repeat what you have learned...



Angular

Services

Observables

Forms

Components

Directives

HTTP

Input/Output

Module

Routing

Lifecycle Hooks

Pipes

Bootstrapping

Dependency Injection

Observables

Why we're talking about it?

Angular is using RxJS Observables for **async.**

What is RxJS

- seeing events as collections you can
  - map
  - filter
  - …

Promises vs. Observables

Observables are built

to solve problems around **async**.

(avoid “callback hell”)

Observables

- streams
- any number of things
- Lazy → Only generate values when subscribed to (**cold**)
- can be “unsubscribed” → can be canceled

Observables - subscribing

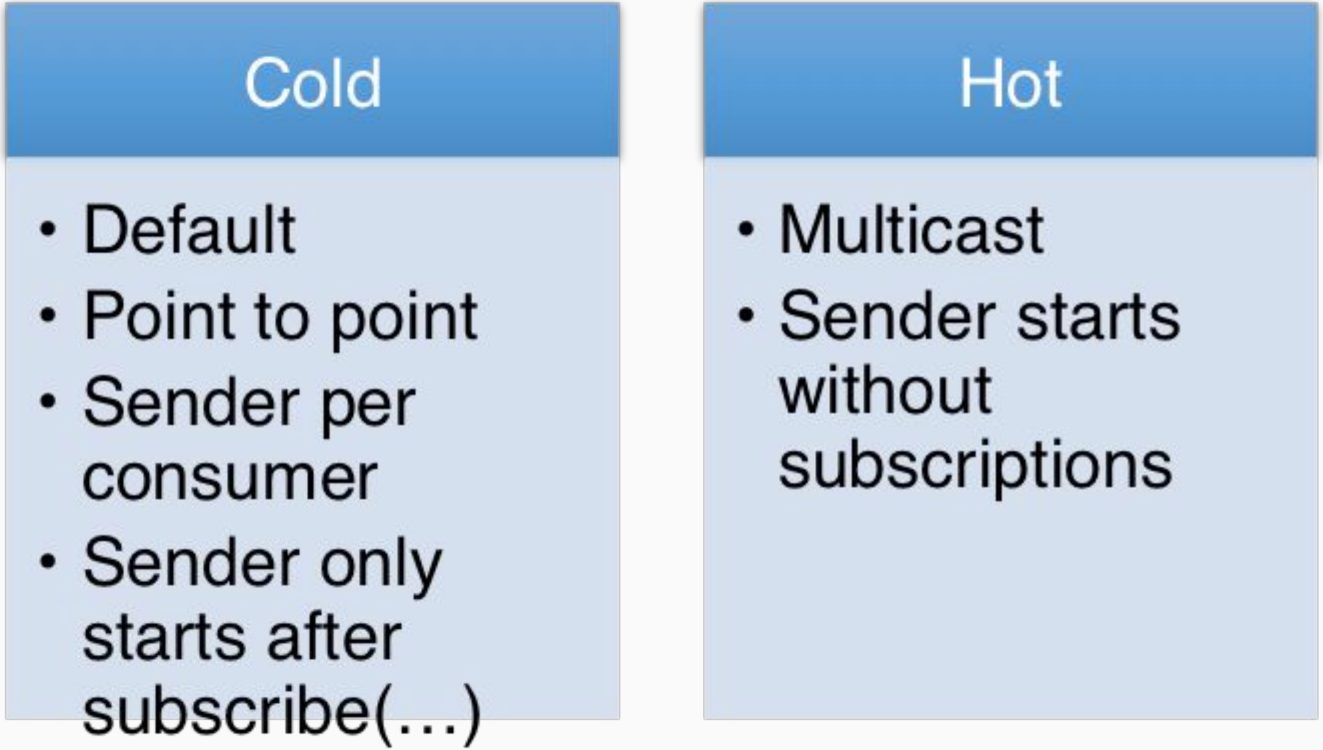
Without subscribing an observable will not be fired

```
.subscribe({

  next: nextFn,

  error: errorFn,

  complete: completeFn

})
```

Observables - Generics

- Functions should return typed data
- Extend type informations of observables with generics
- E.g. `getAll(): Observable<Book[]> {}`

Observables Cold vs. Hot



Cold

- Default
- Point to point
- Sender per consumer
- Sender only starts after subscribe(…)

Hot

- Multicast
- Sender starts without subscriptions

Task

Create an Observable

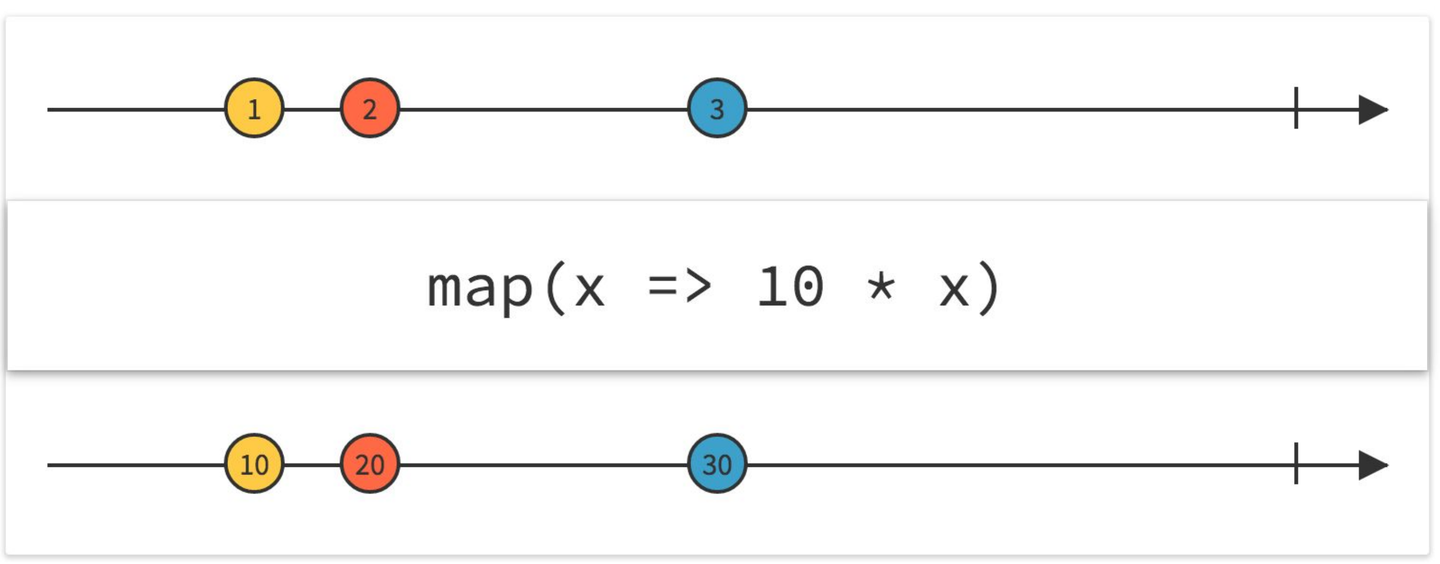
Transformation Operators

Operator

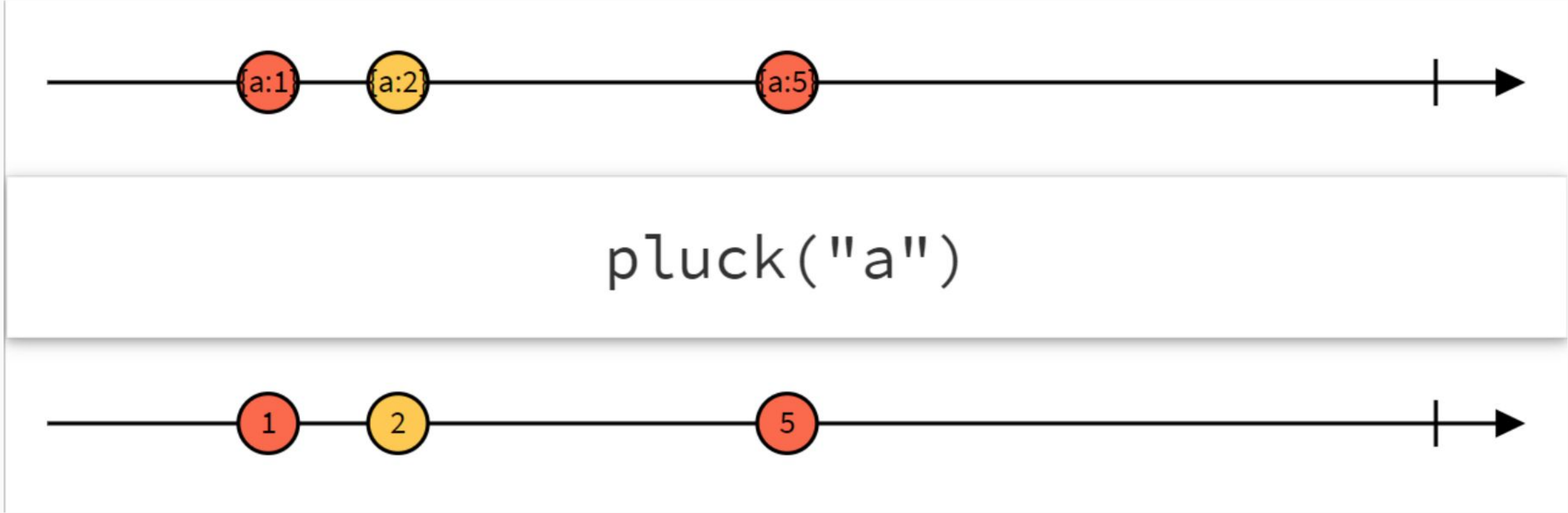
- Operators are functions
- allow complex asynchronous code to be easily composed in a declarative manner.

```
observableInstance.pipe(operator1(), operator2(), ...).
```

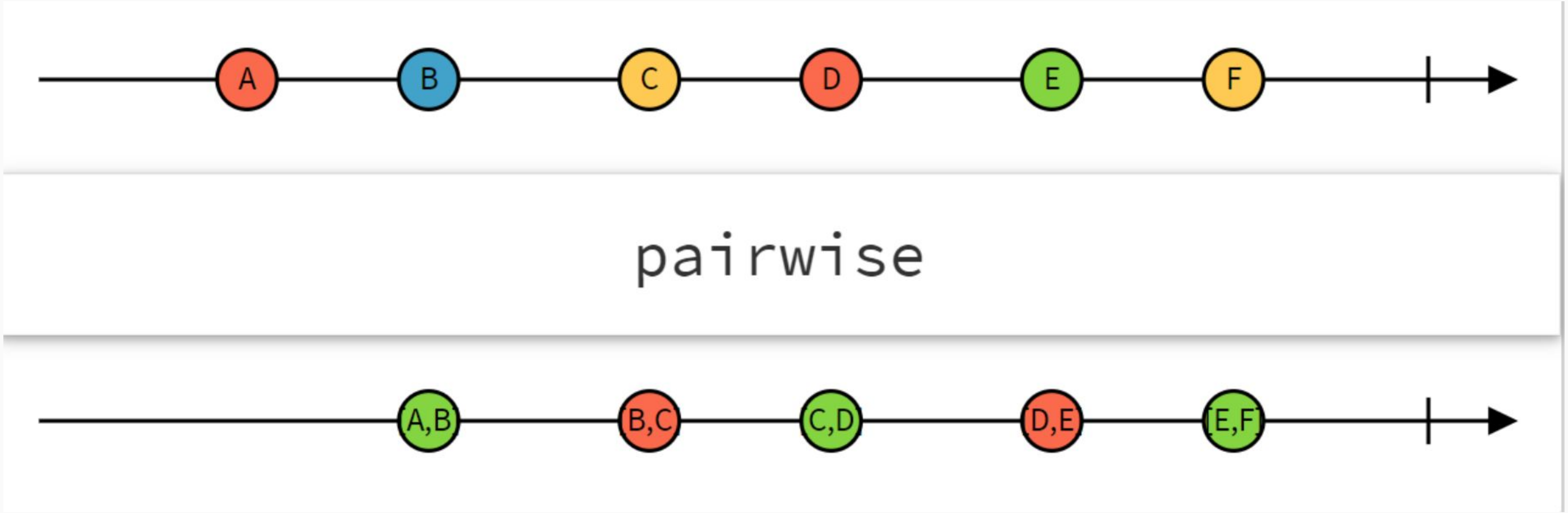
map()



.pluck()

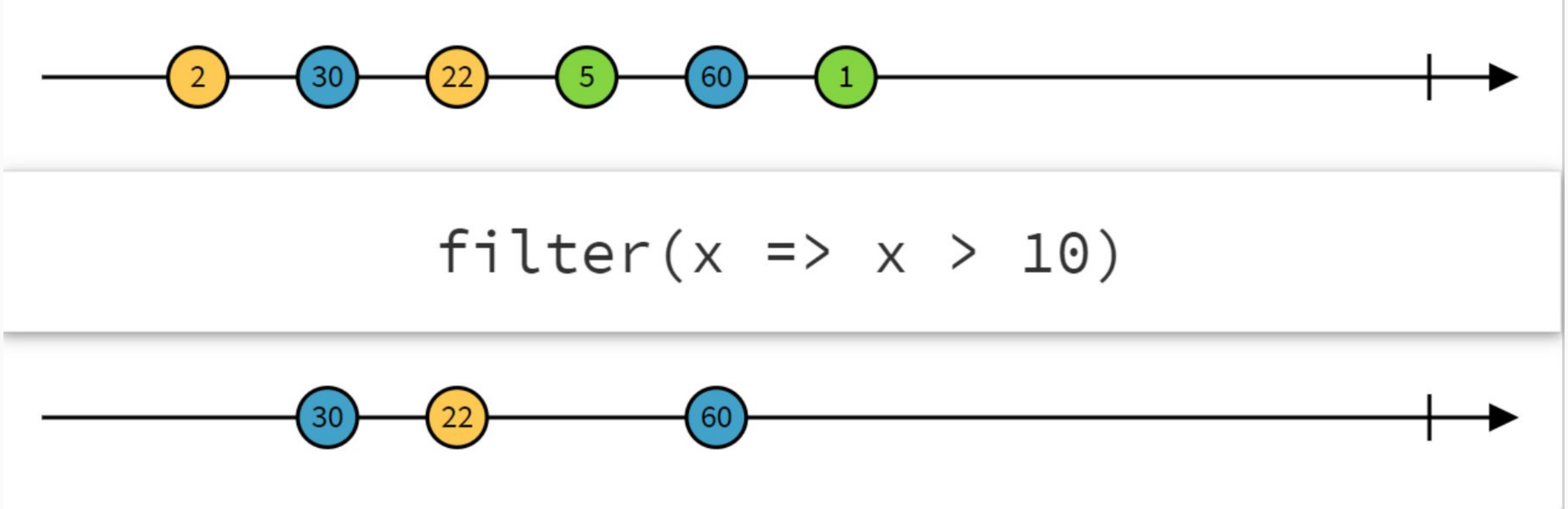


pairwise()

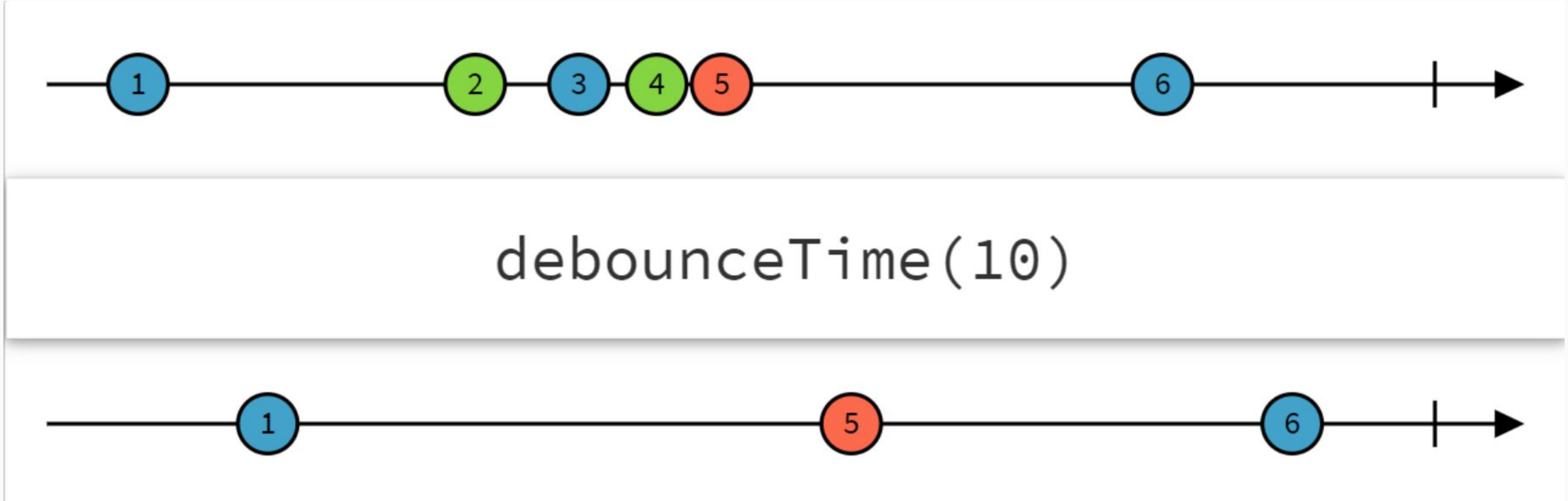


Filtering Operators

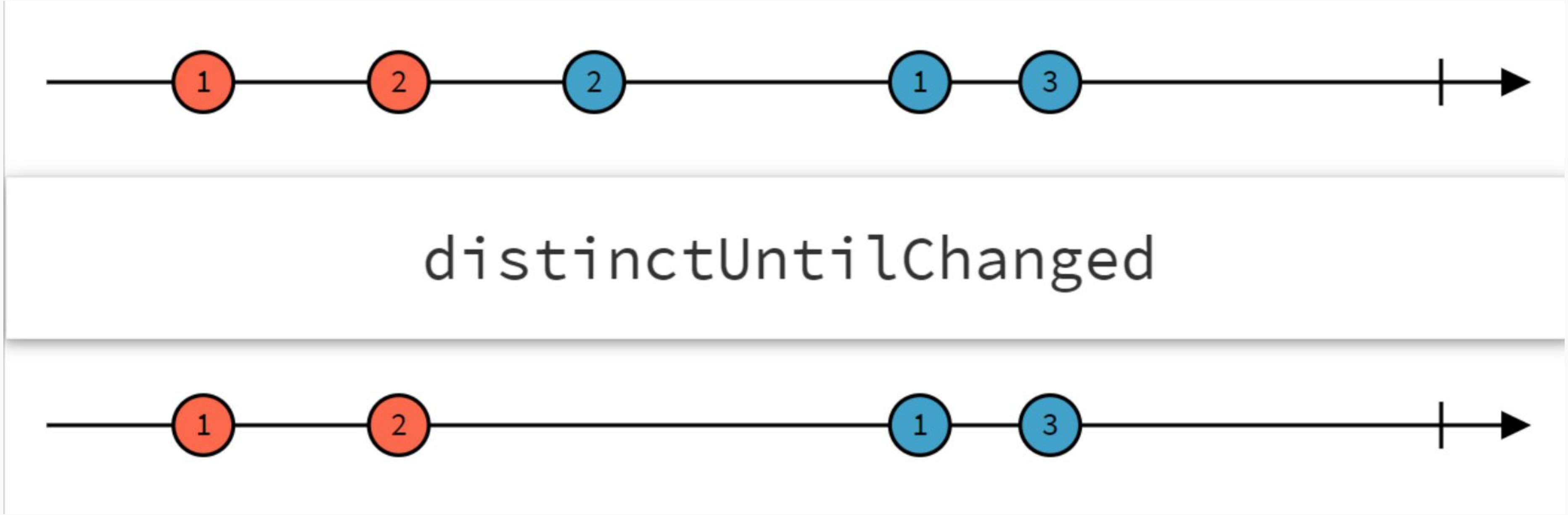
filter()



debounceTime(n: milliseconds)

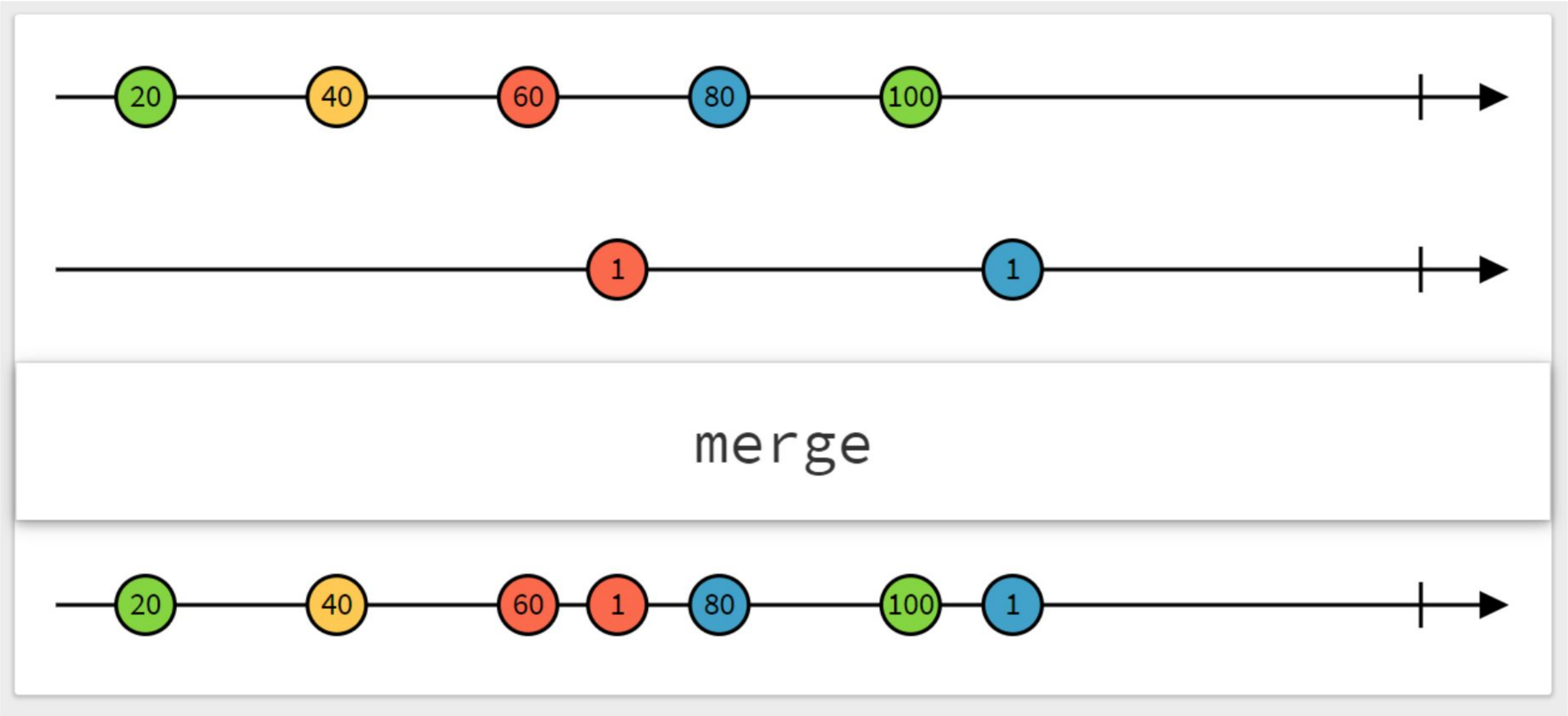


distinctUntilChanged()

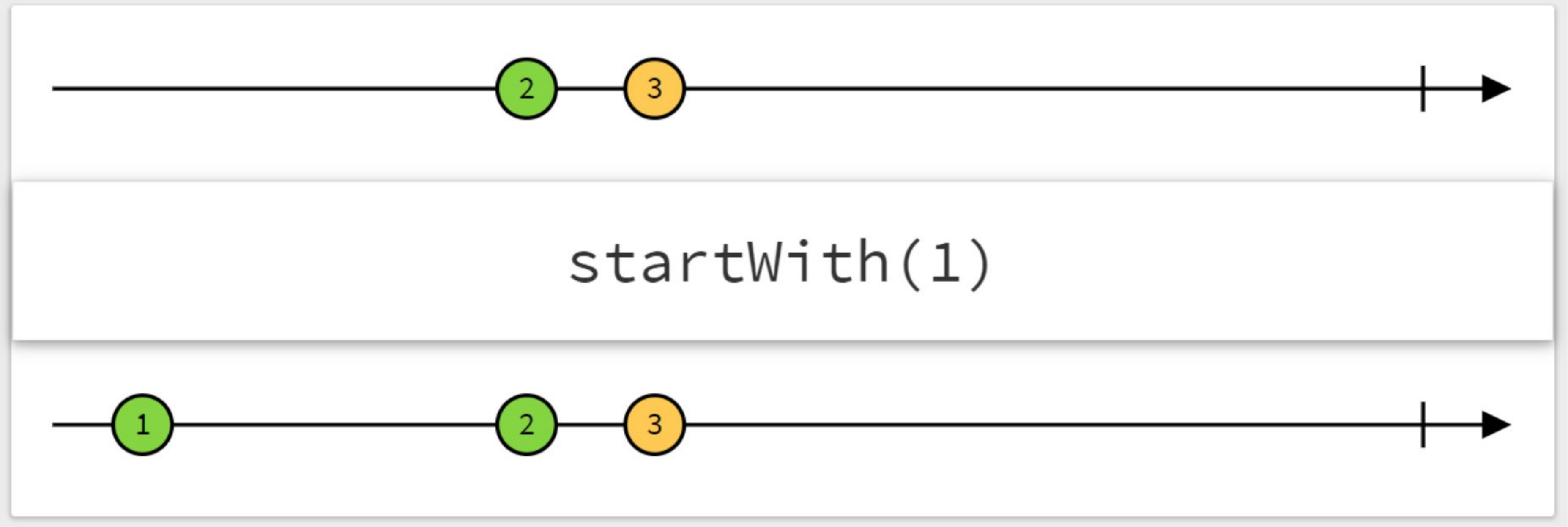


Combination Operators

merge()



startWith()



Error Handling

Operators for Error Handling

- catchError
- retry
- retryWhen

Observable Creation Operators

You are usually not creating

your own observables!

Observables Creation operators

- of(value1, value2, …)
- from(promise/iterable/observable)
- fromEvent(item, eventName)
- Angular HttpClient
- Many more

Observables - cancellation

```
const subscription: Subscription = observable.subscribe(...)

subscription.unsubscribe()
```

Need to unsubscribe!

```
class BookComponent implements OnInit, OnDestroy {
  subscription: Subscription;
  private readonly bookApi = inject(BookApiService);

  ngOnInit() {
    this.subscription = this.bookApi.getAll()
      .subscribe(books => this.books = books);
  }

  ngOnDestroy() {
    this.subscription.unsubscribe()
  }
}
```

Unsubscribe: New Angular Operator

<code>

```
import { takeUntilDestroyed } from '@angular/core/rxjs-interop';

class BookComponent implements OnInit {

  private readonly bookApi = inject(BookApiService);
  private books$ = this.bookApi.getAll().pipe(takeUntilDestroyed())

  ngOnInit() {
    this.bookApi.getAll().subscribe(books => this.books = books);
  }

}
```

Can only be used within an injection context or a destroyRef must be provided

Unsubscribe: New Angular Operator

<code>

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

class BookComponent {

  private readonly destroyRef = inject(DestroyRef);

  anyMethod() {
    this.bookApi.getAll().pipe(takeUntilDestroyed(this.destroyRef))
      .subscribe(books => this.books = books);
  }

}
```

Can only be used within an injection context or a destroyRef must be provided

Need to unsubscribe!

```
class BookComponent implements OnDestroy {
  subscription: Subscription;
  constructor(private readonly bookApi: BookApiService) {
    this.subscription = this.bookApi
      .getAll()
      .subscribe(books => this.books = books);
  }

  ngOnDestroy() {
    this.subscription.unsubscribe()
  }
}
```

Unsubscribe: New Angular Operator

<code>

```
class BookComponent {

  constructor(private readonly bookApi: BookApiService) {
    this.bookApi.getAll()
      .pipe(takeUntilDestroyed())
      .subscribe(books => this.books = books);
  }

}
```

Can only be used within an injection context or a destroyRef must be provided

Hint

Have a look at

RxJS Operators - Explanation, Real World Use Cases, and Anti Patterns

on YouTube to improve your knowledge about handling asynchronous operations with RxJS.

https://www.youtube.com/playlist?list=PLJBWPTwg7fuNKFg5BRXKTvDrEi-45YzSE

HttpClient

Using the HttpClient

- Basic HTTP handling
- Needs to be added to app.config.ts
- `import { provideHttpClient } from '@angular/common/http';`
- Provides methods for
  - GET
  - PUT
  - POST
  - DELETE

HttpClient

<code>

HttpClient needs to be added to **app.config.ts**

```
import { ApplicationConfig } from '@angular/core';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [provideHttpClient()]
};
```

HttpClient Interface

Name	Parameter	Return Value
get	url, options?	Observable<TPayload>
post	url, body, options?	Observable<TPayload>
put	url, body, options?	Observable<TPayload>
delete	url, options?	Observable<TPayload>
patch	url, body, options?	Observable<TPayload>
head	url, options?	Observable<TPayload>
request	Request, options?	Observable<TPayload>

HttpClient usage

<code>

HttpClient methods return observables

```
import { HttpClient } from '@angular/common/http';

class BookApiService {
  private baseUrl = 'http://localhost:4730/';

  constructor(private readonly http: HttpClient) {}

  getAll(): Observable<Book[]> {
    return this.http.get<Book[]>(this.baseUrl)
  }
}
```

HttpClient usage

<code>

HttpClient methods return observables

```
import { HttpClient } from '@angular/common/http';

class BookApiService {
  private baseUrl = 'http://localhost:4730/';

  private readonly http = inject(HttpClient)

  getAll(): Observable<Book[]> {
    return this.http.get<Book[]>(this.baseUrl)
  }
}
```

HttpClient

- Returns an Observable
- Expects data in JSON format

HttpClient - Full response

Use observe: 'response' to get the full response

```
httpClient
  .get<Book[]>('/books', {observe: 'response'})
  .subscribe(resp => {
    console.log(resp.headers.get('X-Custom-Header'));
    console.log(resp.body);
  });
```

HttpClient service

<code>

Subscribe to service observable in a component

```
class BookComponent {

  constructor(private readonly bookApi: BookApiService) {
    this.bookApi
        .getAll()
        .subscribe(books => this.books = books);
  }
```

HttpClient service

<code>

Subscribe to service observable in a component

```
class BookComponent {

  constructor(private readonly bookApi: BookApiService) {
    this.bookApi
        .getAll()
        .subscribe(books => this.books = books);
  }
```

Nothing changed, it is still the same call.

HttpClient service

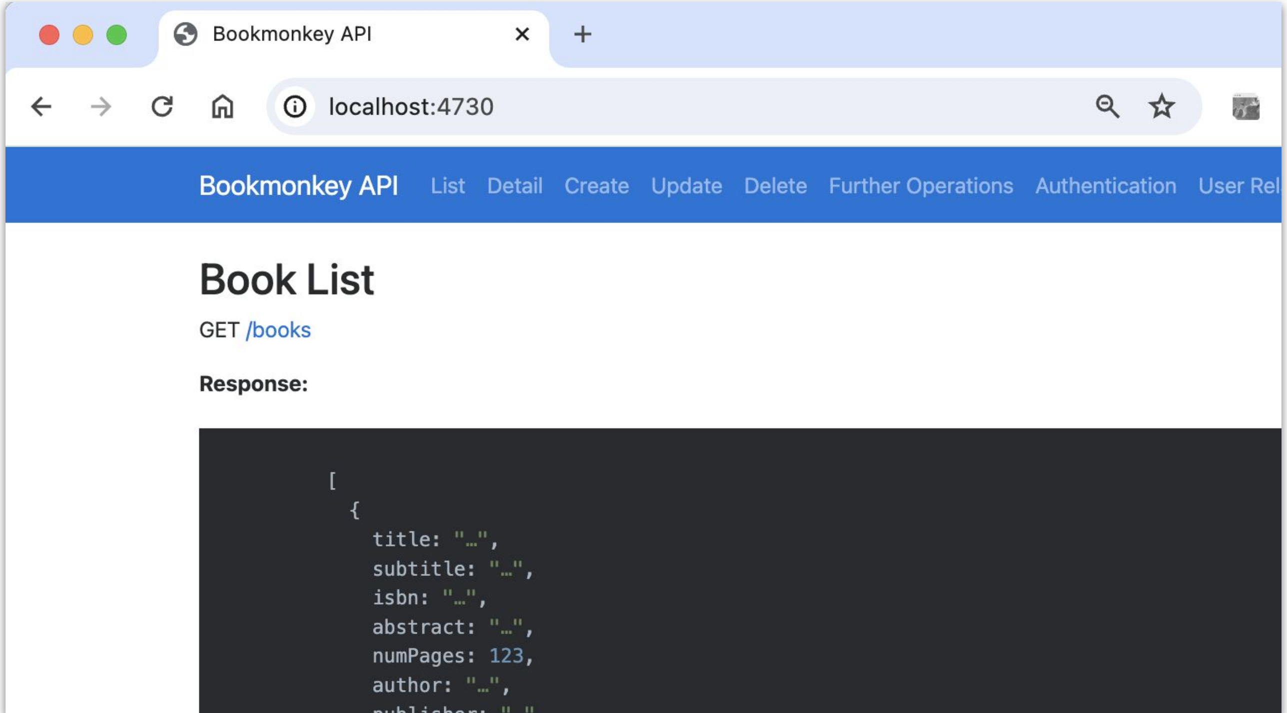
<code>

Subscribe to service observable in a component

```
constructor(private readonly bookApi: BookApiService) {
  this.bookApi
      .getAll()
      .pipe(takeUntilDestroyed())
      .subscribe(books => this.books = books);
 }
```

bookmonkey-api - Demo Backend

```
npm install -g @angular/cli bookmonkey-api
npx bookmonkey-api
```



Book List

GET /books

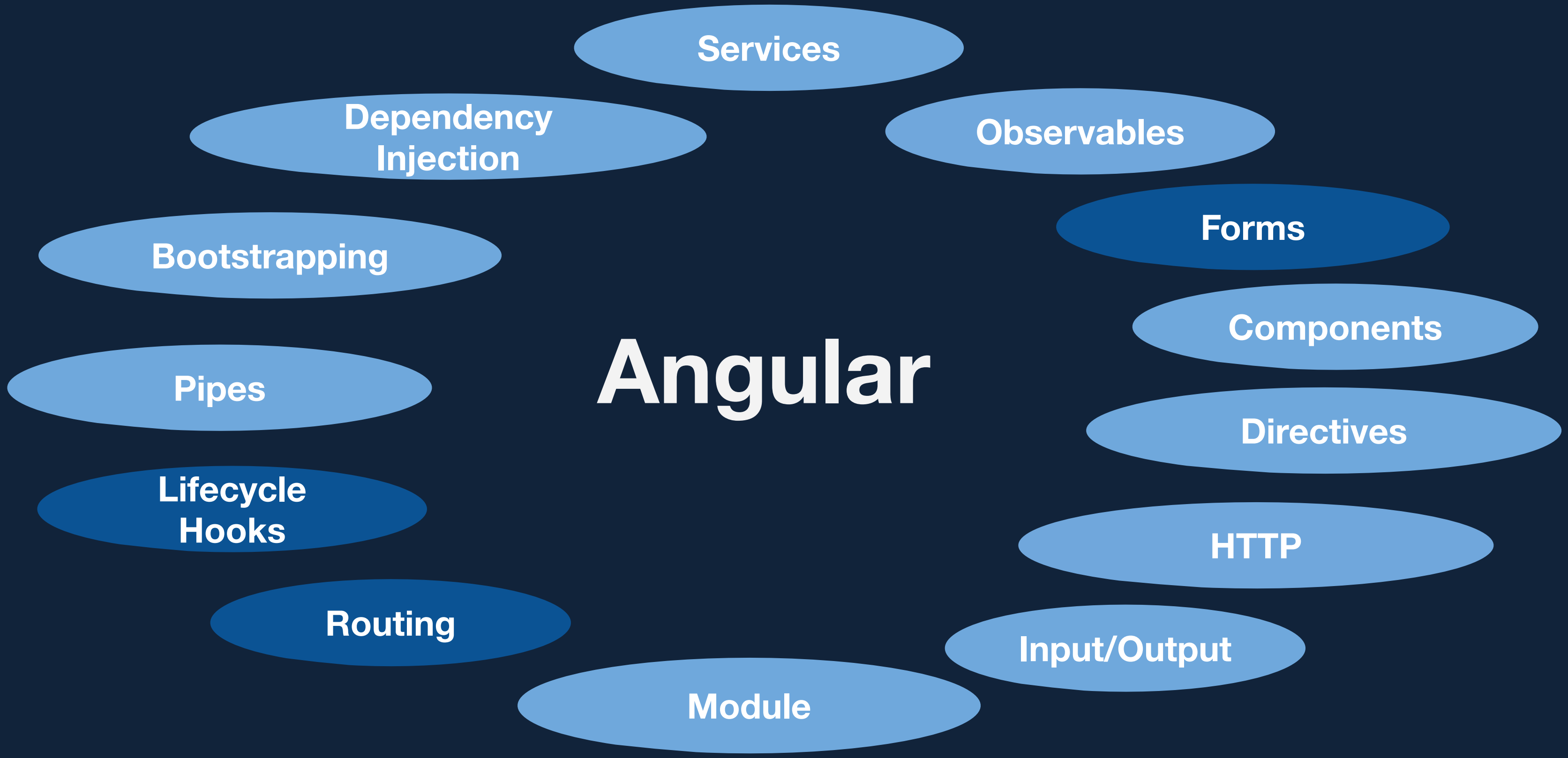
Response:

```
[
  {
    title: "…",
    subtitle: "…",
    isbn: "…",
    abstract: "…",
    numPages: 123,
    author: "…",
    publisher: "…"
```

Task

Load data from local API

Repeat what you have learned...



Angular

Services

Observables

Forms

Components

Directives

HTTP

Input/Output

Module

Routing

Lifecycle Hooks

Pipes

Bootstrapping

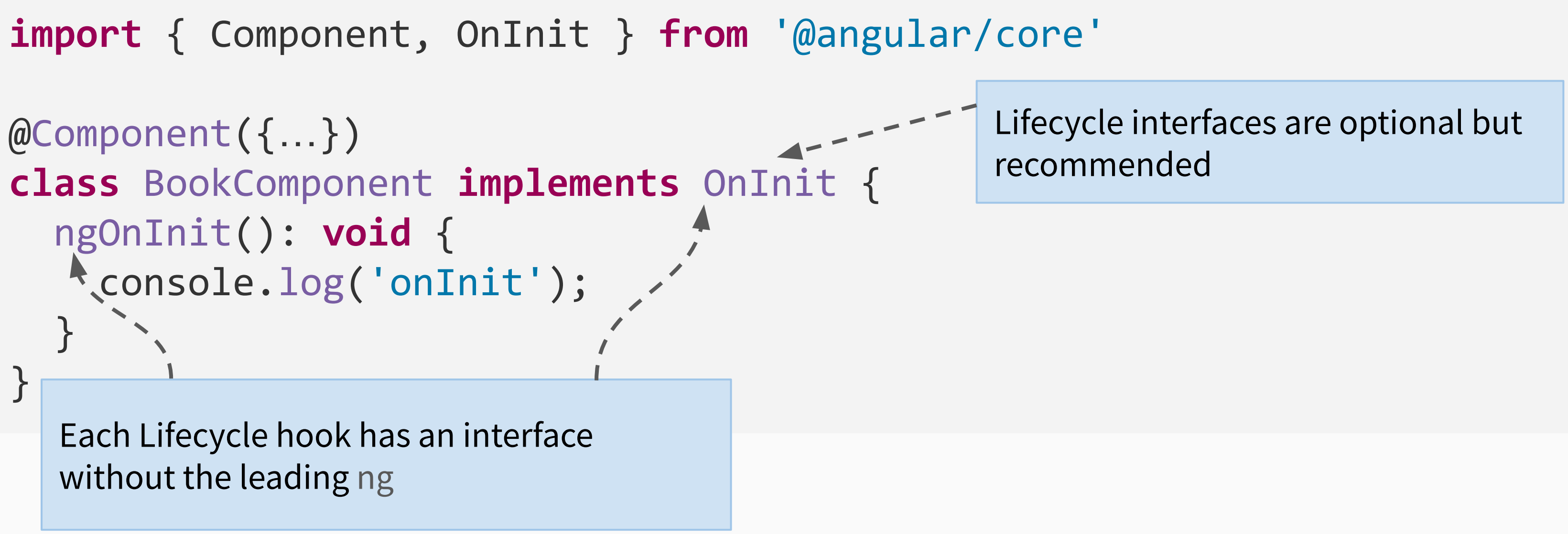
Dependency Injection

Lifecycle Hooks

Lifecycle Hooks

- Components, Directives, Pipes, and Services have a lifecycle managed by Angular
- Visibility of key moments and way to act when they occur
- Classes can implement one or more interfaces with hooks

Lifecycle Hooks - Interfaces



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

@Component({…})
class BookComponent implements OnInit {
  ngOnInit(): void {
    console.log('onInit');
  }
}
```

Lifecycle interfaces are optional but recommended

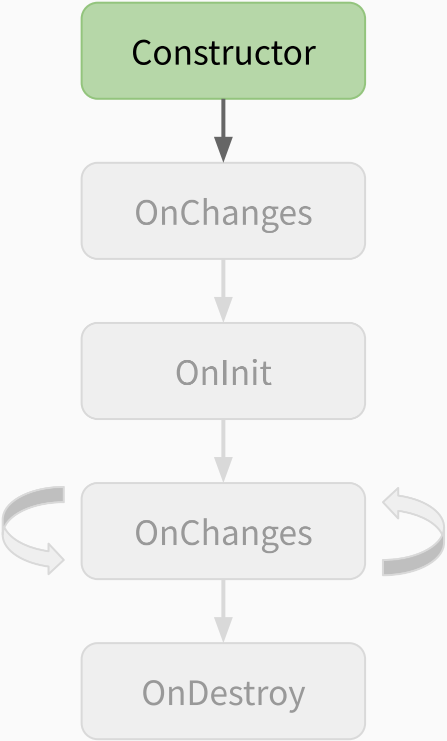
Each Lifecycle hook has an interface without the leading ng

Most important hooks

Component Lifecycle Hooks - Execution

- Injector instantiates component with new

```
@Component({
  selector: 'my-component',
  …
})
class MyComponent {
  constructor () {}
  …
}
```



Constructor

OnChanges

OnInit

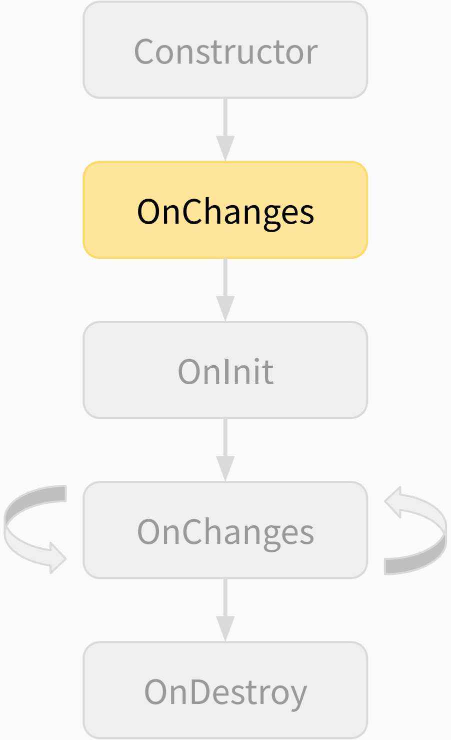
OnChanges

OnDestroy

Component Lifecycle Hooks - Execution

- Check for initial values on `inputs`

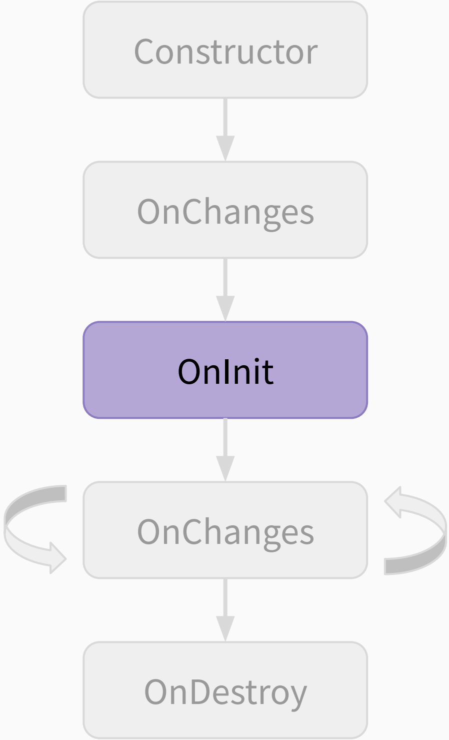
```
<my-component [anInput]="value">
</my-component>
...
class MyComponent {
  anInput = input();
  ...
}
```



Component Lifecycle Hooks - Execution

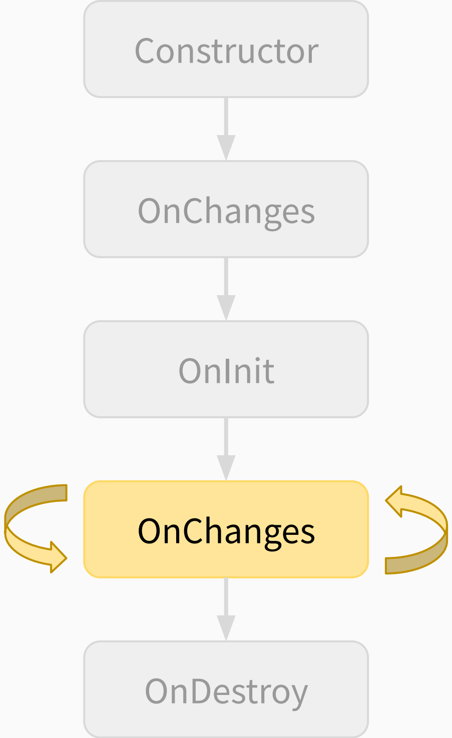
- Initial values are set
- For heavy or async work
- Called only once

```
class MyComponent implements OnInit {
  anInput = input;

  constructor() { // this.anInput is undefined }

  ngOnInit() { // this.anInput is set }
}
```



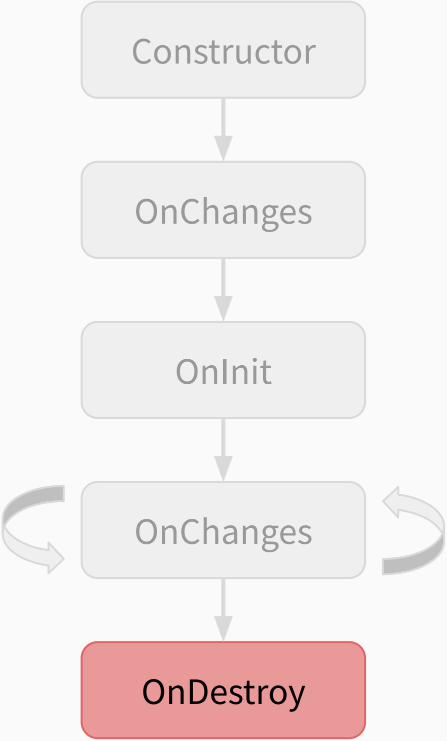
Component Lifecycle Hooks - Execution

→ Every time an `input`-binding changes



Component Lifecycle Hooks - Execution

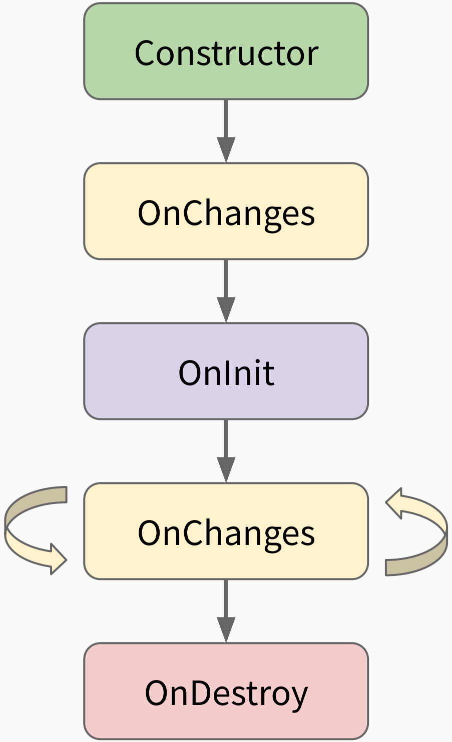
- Cleanup before component is removed
    - Remove event listeners
    - Remove observable subscribers
    - Clean up intervals and timeout
    - Inform other program parts
- Called only once



Component Lifecycle Hooks - Execution

Simplified

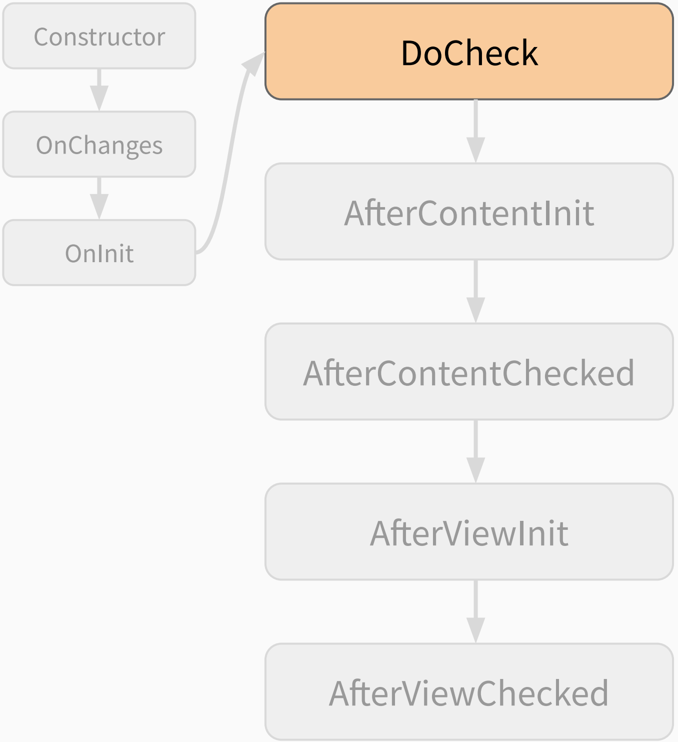
1. Component is instantiated
2. OnChanges: initial `input` values
3. OnInit: once after first OnChanges
4. OnChanges: get changed `input`
5. OnDestroy: component is destroyed



After component creation

Component Lifecycle Hooks - Execution

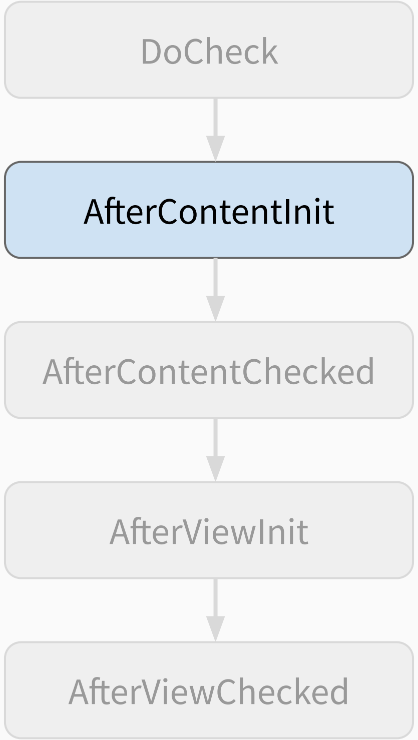
- Every time change detection runs
- Custom change detection function



Component Lifecycle Hooks - Execution

- Content = everything between component tags
- *ngContent* projects content to view after creation
- Hook called after projection finished → only once

```
@Component({
  selector: 'my-component',
  template: '...<ng-content></ng-content>...'
})
...
<my-component><p>Hello</p></my-component>
```



DoCheck

AfterContentInit

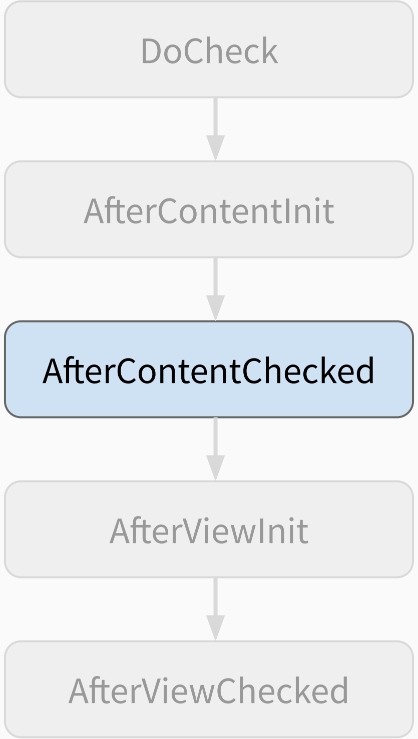
AfterContentChecked

AfterViewInit

AfterViewChecked

Component Lifecycle Hooks - Execution

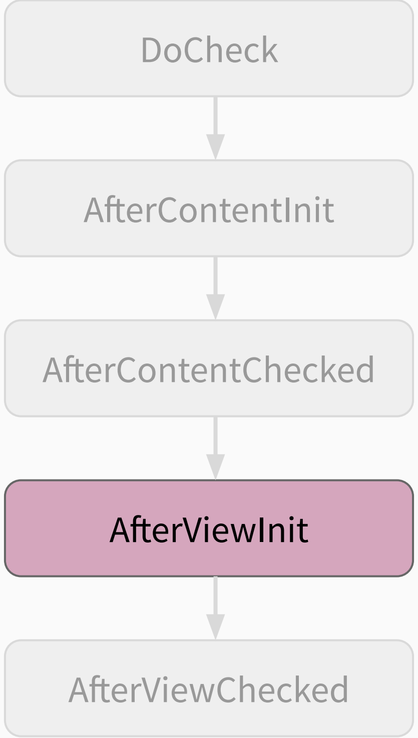
- After change detection → content is checked
- Called every time after DoCheck hook
- Initial check after content is initialised



Component Lifecycle Hooks - Execution

- View = template + bindings
- Called after view and subviews are initialised → only once

```
@Component({
  selector: 'my-component',
  template: `
    <h1>Hello</h1>
    <another-component></another-component>
  `
})
```



DoCheck

AfterContentInit

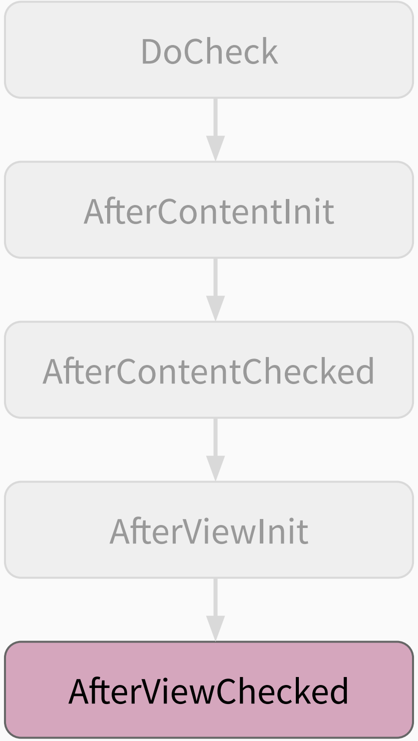
AfterContentChecked

AfterViewInit

AfterViewChecked

Component Lifecycle Hooks - Execution

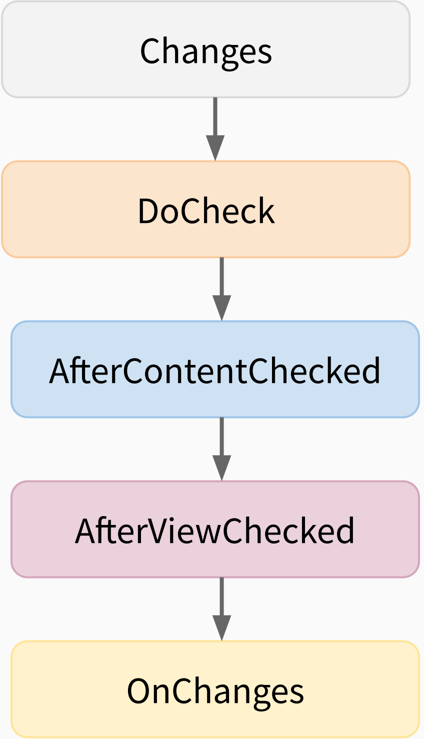
- After change detection → view is checked
- Called every time after DoCheck hook and after AfterContentChecked
- Initial call after the view is initialised



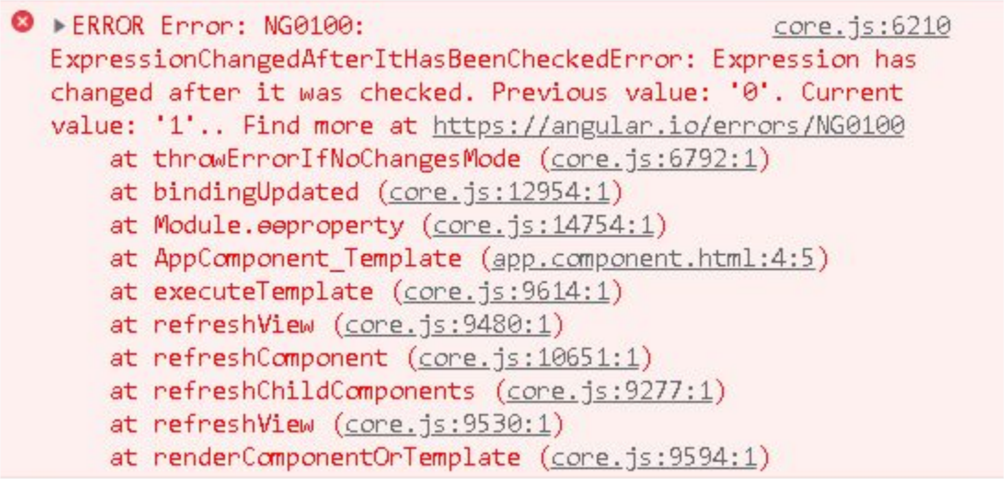
After change detection

Component Lifecycle Hooks - Execution

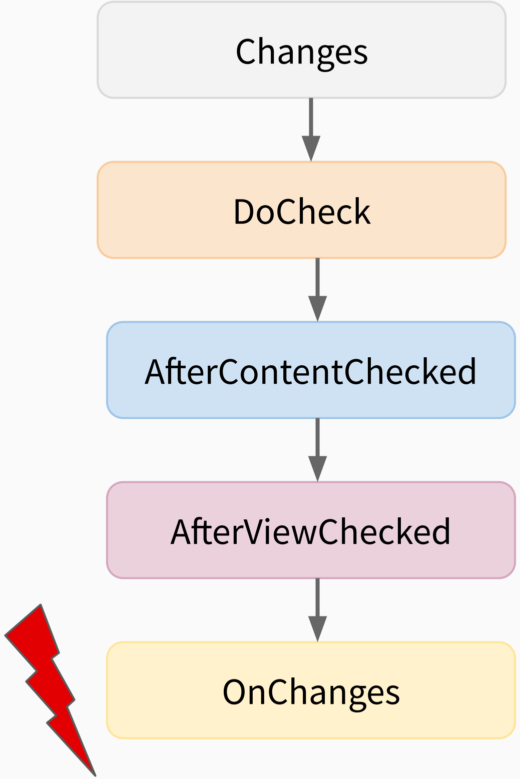
- After a change
- Change detection calls custom DoCheck function
- Content and view are checked
- Inform about possible changes



Component Lifecycle Hooks - Dev Check

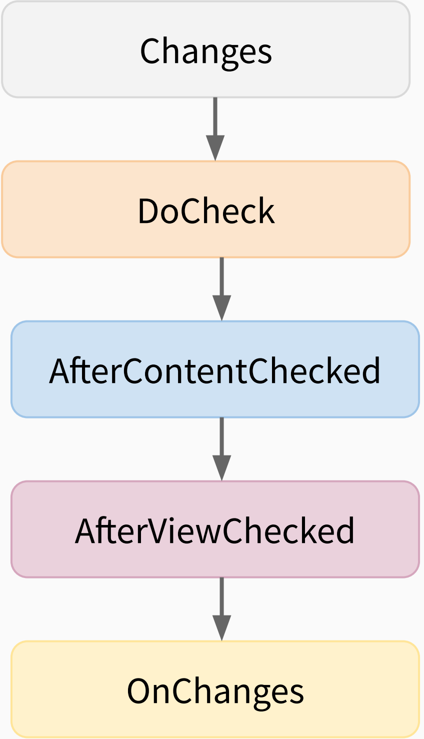


```
▶ERROR Error: NG0100:                                    core.js:6210
ExpressionChangedAfterItHasBeenCheckedError: Expression has
changed after it was checked. Previous value: '0'. Current
value: '1'.. Find more at https://angular.io/errors/NG0100
    at throwErrorIfNoChangesMode (core.js:6792:1)
    at bindingUpdated (core.js:12954:1)
    at Module.ɵɵproperty (core.js:14754:1)
    at AppComponent_Template (app.component.html:4:5)
    at executeTemplate (core.js:9614:1)
    at refreshView (core.js:9480:1)
    at refreshComponent (core.js:10651:1)
    at refreshChildComponents (core.js:9277:1)
    at refreshView (core.js:9530:1)
    at renderComponentOrTemplate (core.js:9594:1)
```



Component Lifecycle Hooks - Dev Check

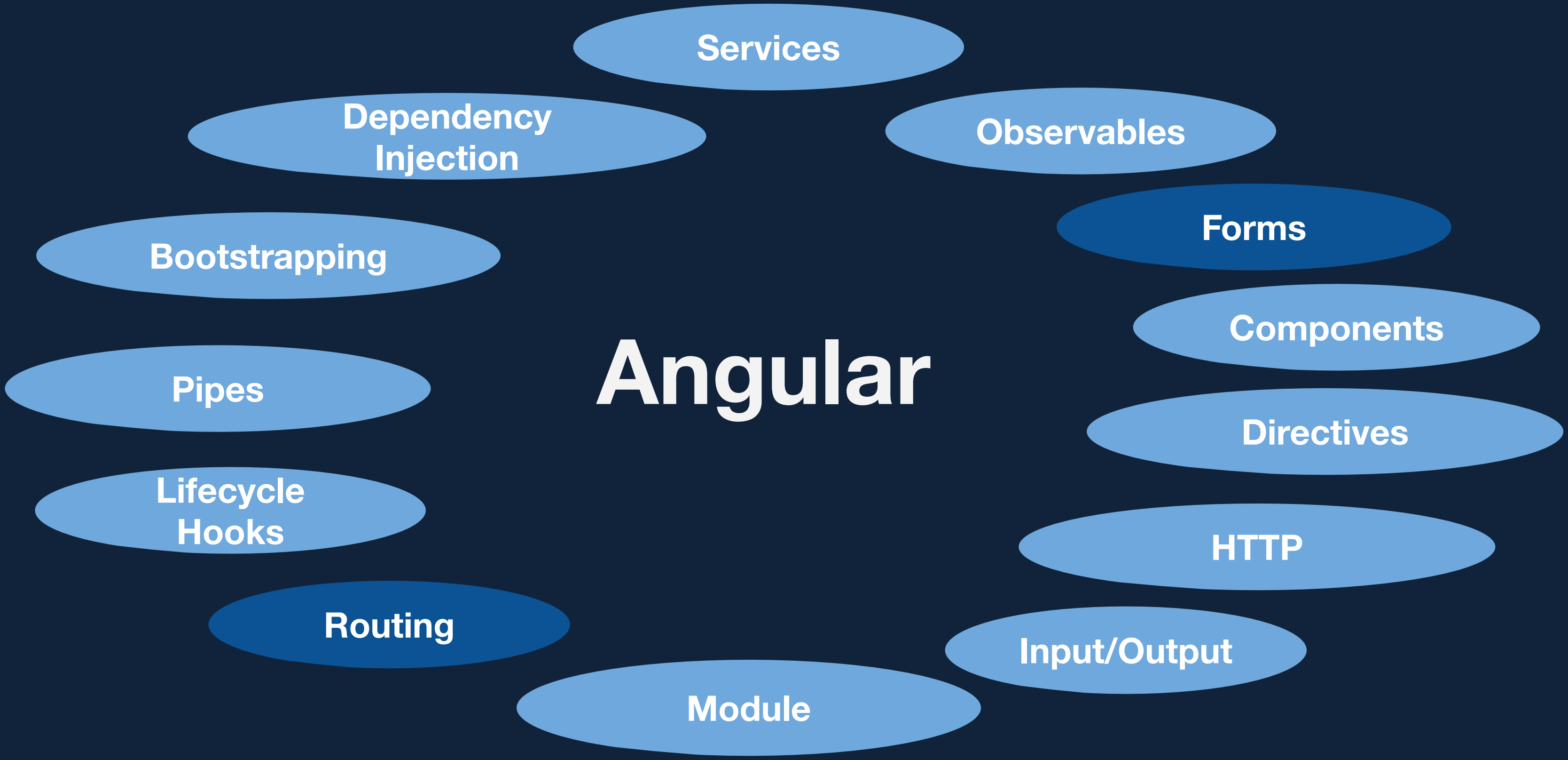
- After the *check* hooks are called once → change detection runs again to check for unexpected changes → triggers the *check* hooks again!
- If Angular notices changes after first change detection run → error in JavaScript console (not in production mode)



Task

Component LifeCycle Basic

Repeat what you have learned...



Angular

Services

Observables

Forms

Components

Directives

HTTP

Input/Output

Module

Routing

Lifecycle Hooks

Pipes

Bootstrapping

Dependency Injection

Async Pipe

The AsyncPipe accepts a Promise or Observable as input and subscribes to the input automatically, eventually returning the emitted values.

Async Pipe

- Subscribe to Observable
- UnSubscribe on component destruction
- Built-In Pipe
- Simple use: {{ books$ | async }}

Async Pipe

<code>

For every async a new subscription is made. Try to minimize use of async

```
@for(book of books$ | async; track book.isbn){
  <app-book-card [content]="book" />
}

<span>{{ (books$ | async).length }}
```

Two subscriptions created causes **two** http-requests sent to the API.

Async Pipe with @if

For every async a new subscription is made. Try to minimize use of async

```
@if (books$ | async; as books) {
  @for (book of books | bookFilter: searchTerm; track book.isbn) {
    <li>{{book.title}}</li>
  }
  <span>{{ books.length }}</span>
}
```

Avoid unwanted multiple HTTP Requests

Share connection among multiple subscribers,

```
import { share } from 'rxjs/operators';

private getAll(): Observable<Book[]> {
  return this.http
    .get<Book[]>(${todosUrl})
    .pipe(share())
}
```

Async Pipe

<code>

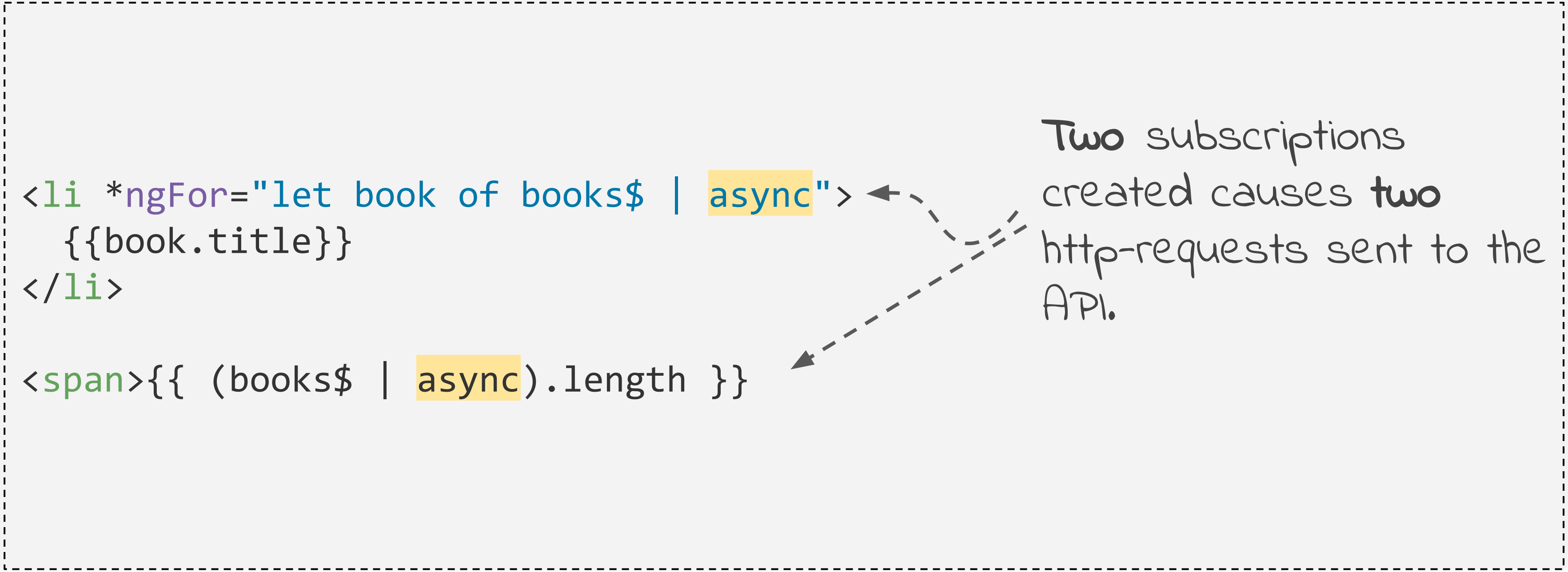
Finnish Notation. Naming observables with an $ suffix

```
@for(book of books$ | async; track book.isbn){
  <app-book-card [content]="book" />
}
```

Async Pipe

<code>

For every async a new subscription is made. Try to minimize use of async



```
<li *ngFor="let book of books$ | async">
  {{book.title}}
</li>

<span>{{ (books$ | async).length }}
```

Two subscriptions created causes **two** http-requests sent to the API.

Async Pipe with *ngIf

For every async a new subscription is made. Try to minimize use of async

```
<div *ngIf="books$ | async as books">
    <li *ngFor="let book of books">
      {{book.title}}
    </li>

    <span>{{ books.length }}</span>
</div>
```

Async Pipe

Finnish Notation. Naming observables with an $ suffix

```
<li *ngFor="let book of books$ | async">
  {{book.title}}
</li>
```

Task

Use the async pipe

Signals

Signals

- New Reactive Primitives
- Alternative for reactive programming / Observables
- Since Angular 17
- More and more signal-based features will be added
- Will not replace Observables!

Signals - Advantages

- Easier to learn / start with
- No subscribe/unsubscribe is necessary
- Signals always have a value (like a BehaviorSubject)
- Signals are always shared (no share-operator needed)
- Some additional features (e.g. better change detection)

Signals vs. Observables

- Signals are not meant to replace observables entirely
- Limited to some main use cases
- Observables are much more powerful
- Angular already provides functions for converting Signals to Observables and vise versa

Usage

3 new Reactive Primitives

1. Signals
2. Computed
3. Effects

1.) (Writable) Signals

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

// Create Signal
book = signal<Book>({...}); // book: WritableSignal<Book>

// Set new value
book.set(otherBook);

// Set new value based on old value
book.update(book => ({ ...book, title: 'NEW: ' + book.title }));
```

Make sure to create a new reference here!

Signals - Template Binding

Component Class:

```
userName = signal('Albert Einstein');
```

Template html:

```
{{ userName() }}    =>
```

Albert Einstein

Signals vs. Observables - Syntax

```
const book = signal(book);
```

```
const book$ = new BehaviorSubject(book)
```

```
{{ book().isbn }}
```

```
{{ (book$ | async)?.isbn }}
```

```
const bookRO = book.asReadonly();
```

```
const bookRO$ = book$.asObservable();
```

2.) Computed Signals

```
userName = signal('Albert Einstein');

// Create Computed Signal based on other Signal
salutation = computed(() => `Hello ${this.userName()}!`);
```

Template:

`{{ salutation() }}`   =>   **Hello Albert Einstein!**

Salutation will change, when userName changes!

Computed Signals vs. Observables - Syntax

```
priceLabel = computed(
  () => (isloggedIn() ? price() : 'Please log in')
);
```

```
priceLabel$ = combineLatest([isloggedIn$, price$]).pipe(
  map(([isloggedIn, price]) => (isloggedIn ? price : 'Please log in'))
);
```

3.) Effects

```
effect(
  () => console.warn(User name changed: ${this.userName()})
);
```

- ➢ Will be executed when referenced Signals change
- ➢ Will be executed at least once

Signals + Observables

Create Signal from Observable

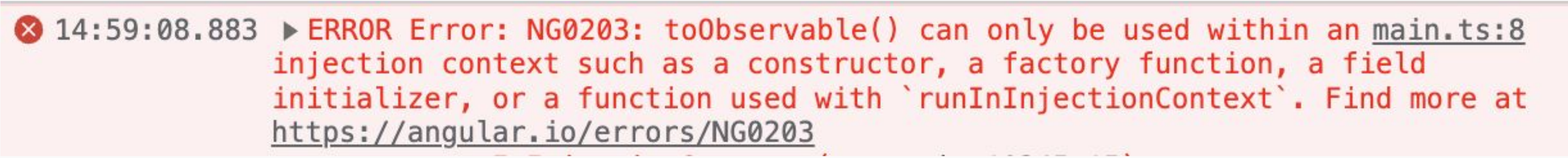
```
import { toObservable, toSignal } from '@angular/core/rxjs-interop';

// Initial value will be 'undefined'
books: Signal<Book[] | undefined> = toSignal(this.bookService.getAll());

// Set initial value
books: Signal<Book[]> = toSignal(this.bookService.getAll(), {
  initialValue: [],
});
```

Create Observable from Signal

```
// Basic usage
const observable$ = toObservable(signal);
```

```
// Example
const name = signal('aeinstein');


const profile$ = toObservable(name).pipe(
  switchMap((userId) => myService.getProfile$(userid))
);
```

Error outside injection context



```
14:59:08.883 ▶ERROR Error: NG0203: toObservable() can only be used within an main.ts:8
             injection context such as a constructor, a factory function, a field
             initializer, or a function used with `runInInjectionContext`. Find more at
             https://angular.io/errors/NG0203
```

- **`toSignal`** and **`toObservable`** require to be called inside of a so called injection context
- Lifecycle hooks like **ngOnInit** don’t provide an injection context!

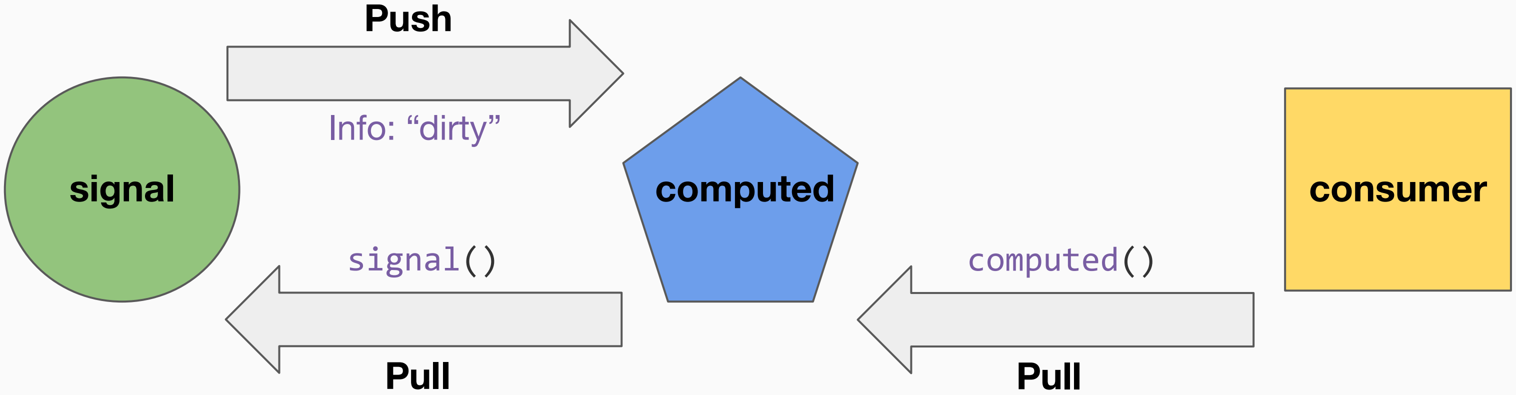
Injection Context Error - Solution

- Either call **`toSignal`** and **`toObservable`** inside of an injection context - e.g. a constructor
- Or provide an injector manually

```
private readonly injector = inject(Injector)

ngOnInit() {
  const userId$ = toObservable(signal('aeinstein'), {
    injector: this.injector,
  });
}
```

Additional Advantages

Push/Pull Pattern



Glitch free computation - “Diamond-Problem”

```
const valueA = signal('A1');
const valueB = signal('B1');

effect(() =>
  console.log(valueA() + valueB())
);


valueA.set('A2');
valueB.set('B2');

// Output: A2B2
```

```
const valA$ = new BehaviorSubject('A1');
const valB$ = new BehaviorSubject('B1');

combineLatest([valA$, valB$])
  .subscribe(
    ([valA, valB]) =>
      console.log(valA + valB));

valA$.next('A2');
valB$.next('B2');

// Output: A1B1
// Output: A2B1
// Output: A2B2
```

Glitch free computation - “Diamond-Problem”

```
label$ = combineLatest([isloggedIn$, price$]).pipe(
  map(([isloggedIn, price]) => (isloggedIn ? price : 'Please log in'))
);
```

isloggedIn$	price$	combineLatest	label$
true	19.99	true + 19.99	19.99
false	18.99	true + 18.99	18.99
		false + 18.99	Please log in

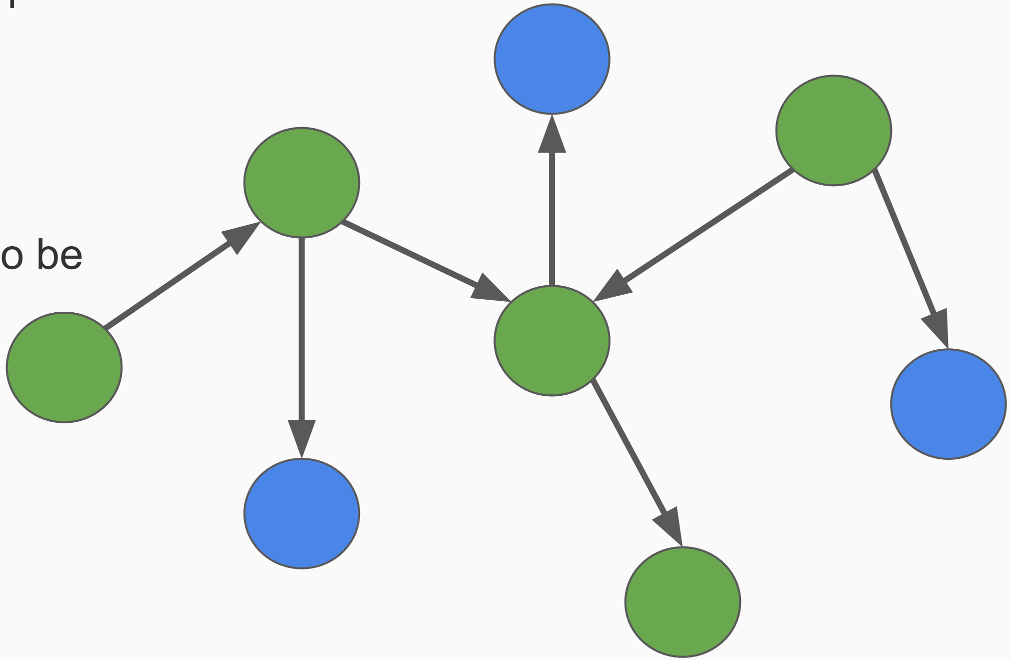
Glitch free computation

```
label = computed(
  () => (isloggedIn() ? price() : 'Please log in')
);
```

isloggedIn()	price()	label()
true	19.99	19.99
false	18.99	Please log in

Better Change Detection

- Signals inform Angular, when their values change
- Results in a more precise Change Detection
- Not the whole graph needs to be updated



Conclusion

Conclusion

- Signals can be used in production with Angular 17+
- Easier for Angular beginners
- Simple Apps can possibly be built only with Signals
- More complexe Apps will still use RxJS

Signals in larger Apps

Not clear yet:

- Will complexe Apps use both, RxJS AND Signals?
- Will we decide by App complexity or by Use Cases what to use?

One approach for combining Signals AND RxJS:

- Signals for template bindings (component state)
  - better Performance
- RxJS for complexe reactivity / external events
  - better Code

Task

Use a Signal

Cheat Sheets

Types

Booleans	`boolean`
Numbers	`number`
Strings	`string`
Lists	`number[] | Array<number>`
Maps	`interface /* separated defined and named */ | {…} /* inline */`
Enumeration	`enum Employees {Miriam, Matthias}`
Any	`any`
Void	`void // only as return type for functions/methods`
Type Casting	`<type> | varName as type`

ES2015/TS Classes

`class`	nicer way to define prototypes
`public`	the **default** for attributes and methods
`private`	only accessible within their declaring class
`protected`	accessible from within their declaring class and classes derived from their declaring class
`static`	methods or attributes can be called or get and set without an instance
`extends`	class gets extended by another class

Interfaces - The new keywords

- `interface`

  create a shape with types

- `implements`

  `class`es can implement an `interface`

Component Decorator - Interface

Name	Description	Default
selector	Define CSS Selector to match the element	undefined
template	View-Template as string	' ' (Empty String)
templateUrl	View-Template via URL	undefined
styleUrls[ ]	Reference to styles via URL	[ ]
directives[ ]	Inject other directives	[ ]
pipes[ ]	Inject other pipes	[ ]
providers[ ]	Define the injectable services	[ ]

Component Decorator - Interface

Name	Description	Default
encapsulation	Define the scoping of styles	Emulated
changeDetection	specify a custom changeDetection	CheckAlways

- there are more, but this are the most used and important ones

Component Lifecycle Hooks

Hook method	Interface	Description
`ngOnChanges`	OnChanges	Called when an input or output binding value changes
`ngOnInit`	OnInit	After the first `ngOnChanges`
`ngDoCheck`	DoCheck	Developer's custom change detection
`ngAfterContentInit`	AfterContentInit	After component content initialized
`ngAfterContentChecked`	AfterContentChecked	After every check of component content
`ngAfterViewInit`	AfterViewInit	After component's view(s) are initialized
`ngAfterViewChecked`	AfterViewChecked	After every check of a component's view(s)
`ngOnDestroy`	OnDestroy	Just before the directive is destroyed

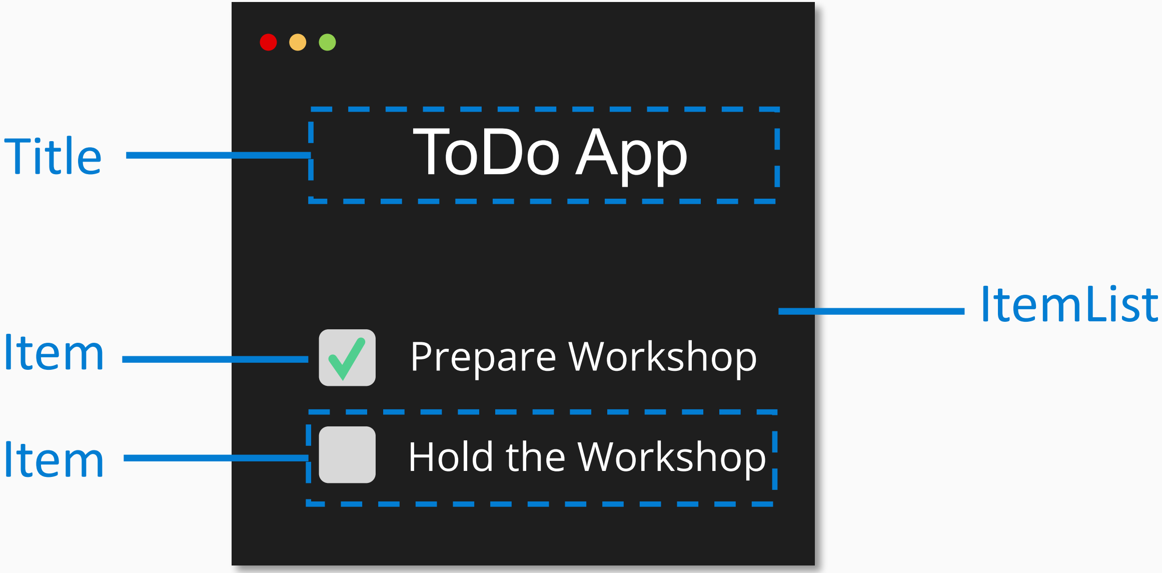
View Encapsulation

Mode	Description
`ViewEncapsulation.`**`None`**	No encapsulation, styles in head
`ViewEncapsulation.`**`Emulated`**	Styles in head with attribute suffix (scoped styles)
`ViewEncapsulation.`**`ShadowDom`**	Use the Shadow DOM

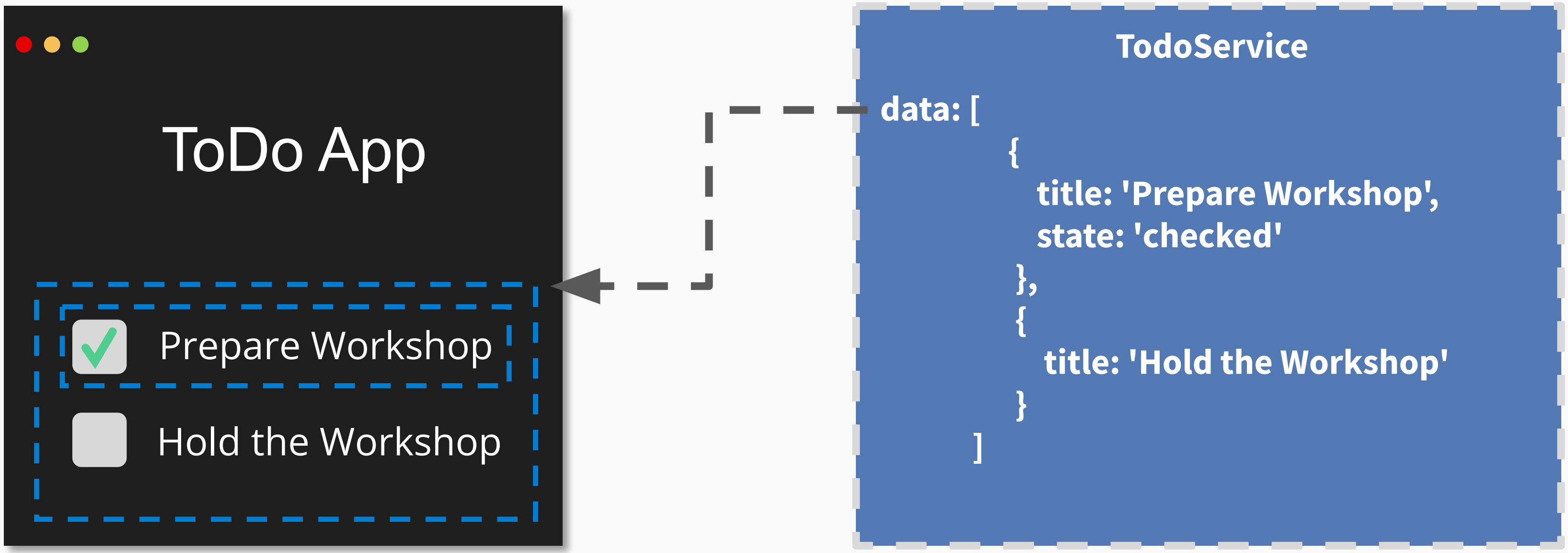
We teach.

workshops.de

Group wisely

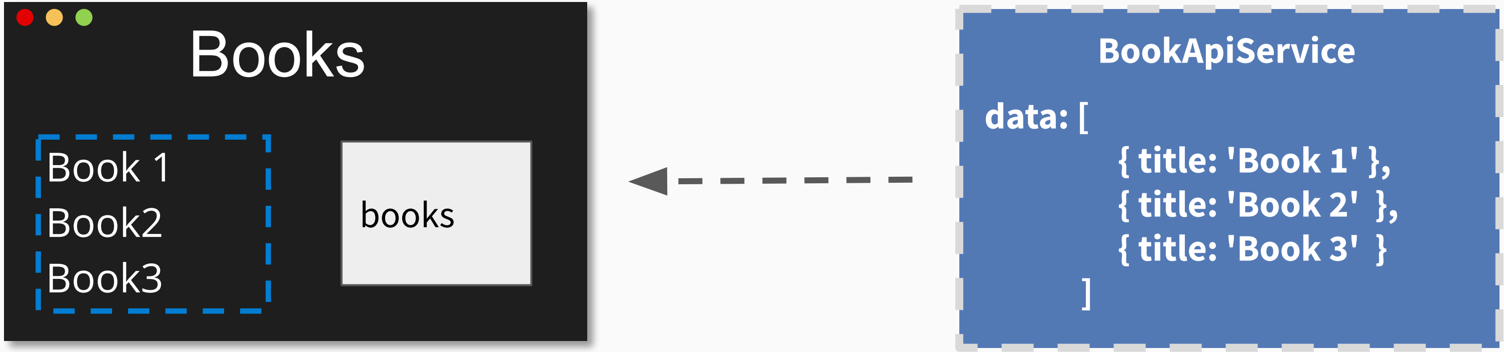


Group wisely

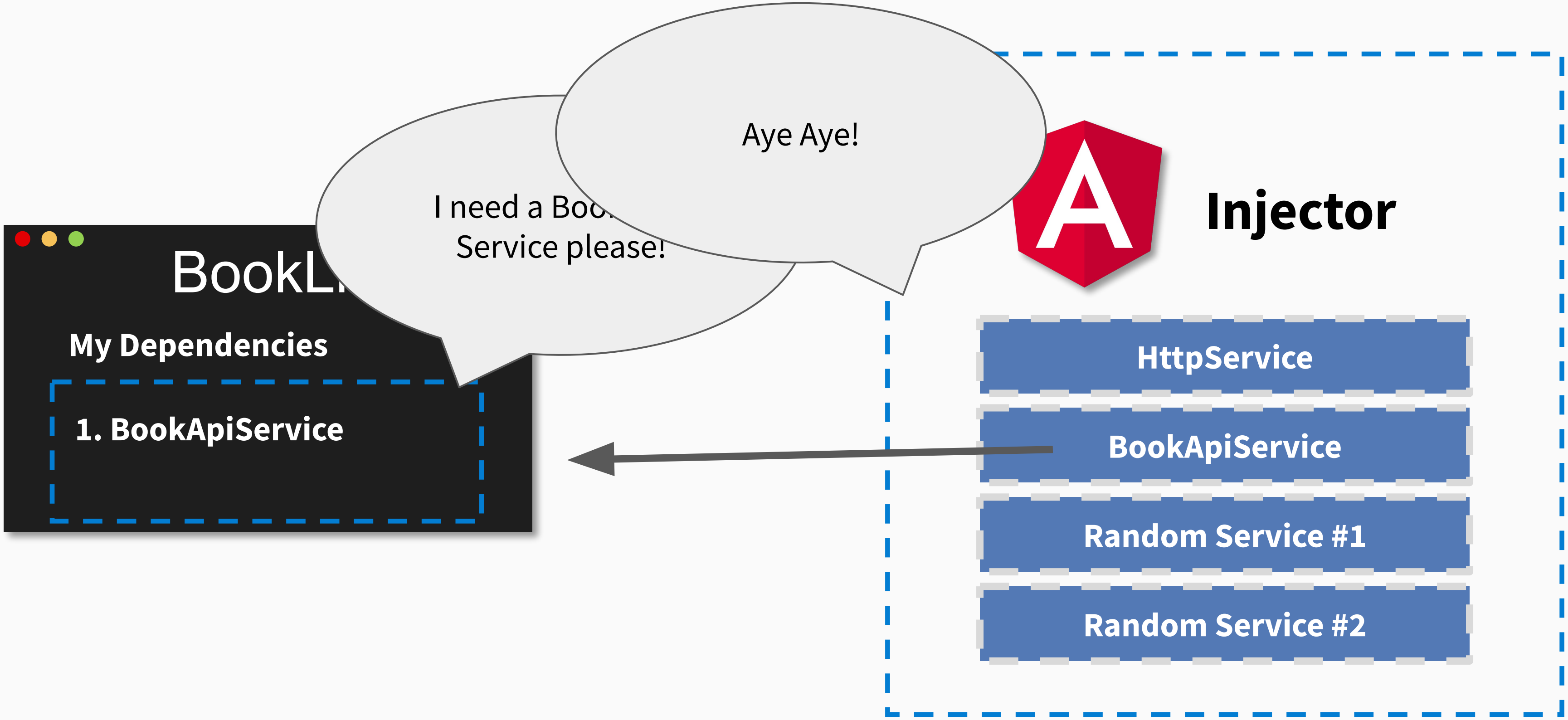


```
data: [
        {
          title: 'Prepare Workshop',
          state: 'checked'
        },
        {
          title: 'Hold the Workshop'
        }
      ]
```

Group wisely



```
data: [
        { title: 'Book 1' },
        { title: 'Book 2'  },
        { title: 'Book 3'  }
    ]
```



BookLi

My Dependencies

1. BookApiService

I need a Boo
Service please!

Aye Aye!

Injector

HttpService

BookApiService

Random Service #1

Random Service #2