Performance Optimization

Лектор: Петър Маламов

Change Detection

Какво е Change Detection?

Последователност от стъпки, която гарантира, че потребителският интерфейс отразява актуалното състояние на приложението.

Change Detection в Angular

Процес, при който Angular проверява стойностите на променливите, използвани в шаблоните на компонентите, и сравнява текущите стойности с предишните.

Ако открие промяна в стойността на променлива, Angular актуализира съответния елемент в DOM-а.

TS app.component.ts

```ts
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  template: `
    <div>
      {{ title }}
    </div>
  `,
})
export class AppComponent {
  title = 'Hello';

  updateTitle() {
    this.title = 'World';
  }
}
```

Важно!

Angular не „пререндерират“ компонентите по време на **change detection**, това означава че не се заменя целия DOM елемент отговарящ за компонента.

Компонентите в Angular имат две състояния:

- **При създаване** – изпълнява се при начално зареждане на приложението (bootstrap) или при създаване на компонента, като включва създаване на DOM елементите.
- **При промяна** – изпълнява се при change detection и актуализира само тези елементи, които се нуждаят от обновяване.

Change Detection в Angular

Процеса на Change Detection може да се раздели на 2 фази:

- Уведомяване за промяна
- Проверка и актуализация на компонентите.

Уведомяване за промяна

Тази фаза има за цел да информира Angular, че е настъпило събитие или промяна, която може да изисква актуализация на компонентите.

Има два начина, по които ръчно можем да уведомим за промяна в някой от компонентите:

- ChangeDetectorRef
- ApplicationRef

ChangeDetectorRef

Базов клас, използван от всички компоненти, който предоставя възможност за извършване на change detection на компонента, към който се отнася референцията, както и на всички негови вложени компоненти.

```
class ViewRef implements ChangeDetectionRef
```

TS app.component.ts

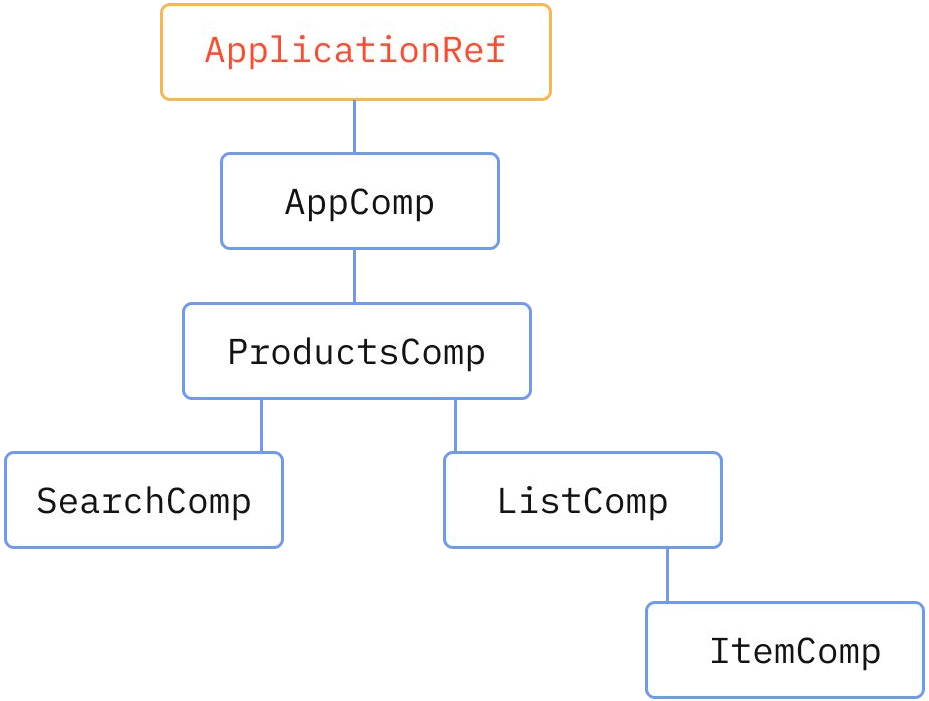
```ts
export class AppComponent {
  title = 'Hello';

  constructor(private cdr: ChangeDetectorRef) {}

  updateTitle() {
    this.title = 'World';
    this.cdr.detectChanges();
  }
}
```

ApplicationRef

Клас, който предоставя референция към Angular приложението, което се изпълнява на страницата. Той предоставя различни методи и свойства за управление на жизнения цикъл на приложението.

Позволява ръчно да се задейства Change Detection в цялото приложение с метода **tick**.

```
class ApplicationRef {
  readonly destroyed: boolean;
  readonly componentTypes: Type<any>[];
  readonly components: ComponentRef<any>[];
  readonly isStable: Observable<boolean>;
  whenStable(): Promise<void>;
  readonly injector: EnvironmentInjector;

  bootstrap<C>(component: Type<C>, rootSelectorOrNode?: any): ComponentRef<C>;
  bootstrap<C>(componentFactory: ComponentFactory<C>, rootSelectorOrNode?: any): ComponentRef<C>;

  tick(): void;
  attachView(viewRef: ViewRef): void;
  detachView(viewRef: ViewRef): void;
  onDestroy(callback: () => void): VoidFunction;
  destroy(): void;
  readonly viewCount: number;
}
```



TS app.component.ts

```ts
export class AppComponent {
  title = 'Hello';

  constructor(private appRef: ApplicationRef) {}

  updateTitle() {
    this.title = 'World';
    this.appRef.tick();
  }
}
```

Проверка и актуализация на компонентите

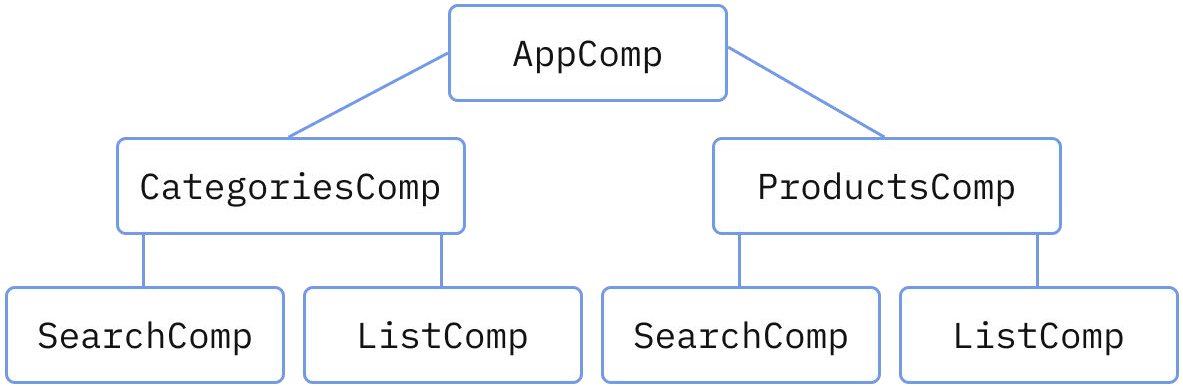
Тази фаза включва реалното изпълнение на Change Detection върху компонентната йерархия, за да се провери за промени и да се актуализира DOM.

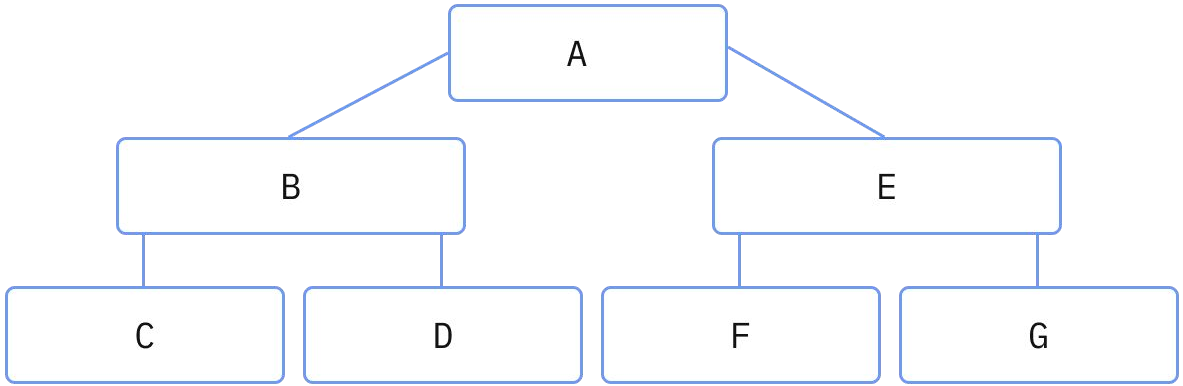
Angular преминава през всички компоненти и проверява стойностите на променливите, използвани в темплейтите. Ако има разлики в стойностите, Angular актуализира DOM елемента.

Когато се стартира процесът на **Change Detection**, той започва от root компонента и обхваща всички компоненти в дървото, като проверява за промени по всеки **node**.

Процесът следва **depth-first search** алгоритъм за да посети всички компоненти.

При открита промяна в компонент, Angular изпълнява някои lifecycle hooks и актуализира темплейта му.





Автоматично уведомяване за промяна

Освен възможността за ръчно задействане на **change detection**, основната роля в управлението на този процес има **scheduler-ът**.

Scheduler-ът за change detection в Angular е базиран на библиотеката **Zone.js**, която следи различни задачи чрез модифициране на браузърните API и прихващане на изпълнението на задачи.

Zone.js

Библиотека за сигнализация, който Angular използва, за да открие кога състоянието на приложението може да е променено.

Той улавя асинхронни операции като:

- setTimeout, setInterval и други.
- click, focus, blur и други.
- HTTP заявки.

Как работи?

Zone.js използва **monkey patching**, за да модифицира основни браузърни API. Това позволява на библиотеката да следи асинхронни операции и да изпълнява определени действия, като задейства Change Detection в Angular, когато се случи промяна в приложението.

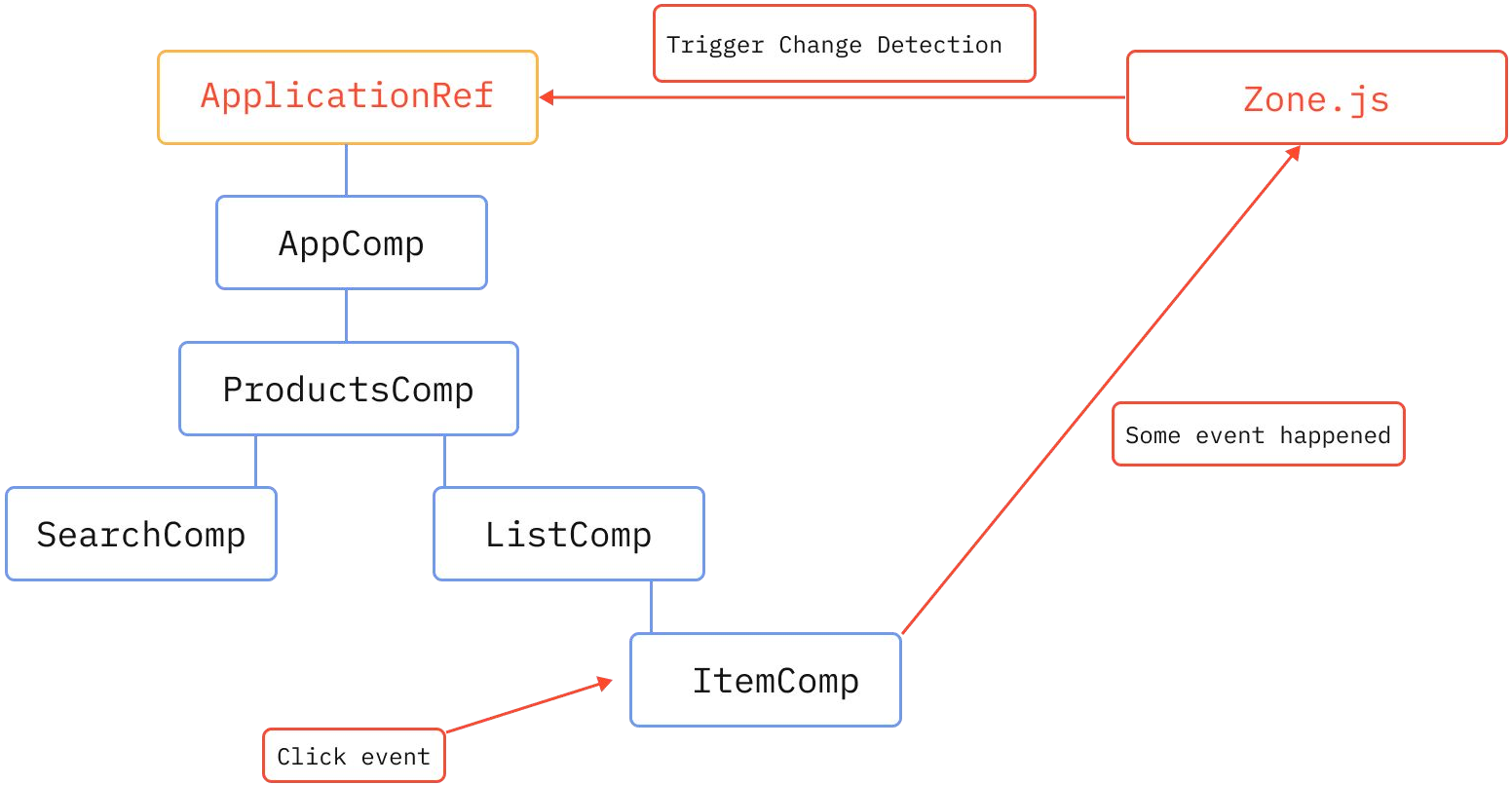
Monkey patching

```
const originalConsoleLog = console.log;

console.log = function(...params) {
    originalConsoleLog(...params);
    // do some action
    const appRootComp = ng.getComponent(/get component by id/);
    appRootComp.changeDetectionRef.detectChanges()
}
```

Как работи?

При bootstrap на приложението се създава инстанция на класа NgZone. При създаване на компонент Angular генерира нова зона за него.

Тази зона проследява асинхронните операции и, когато те завършат, изпраща сигнал към основната инстанция, която задейства Change Detection чрез **ApplicationRef**.



Важно!

Не всяко асинхронно събитие задейства Change Detection.

За да се задейства Change Detection:

- Събитието трябва да бъде изпълнено в рамките на зоната.
- Събитието трябва да има event handler.

TS app.component.ts

```ts
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  template: `
    <div>
      {{ title }}
      <button (click)="updateTitle()">Update</button>
    </div>
  `,
})
export class AppComponent {
  title = 'Hello';

  constructor() {}

  updateTitle() {
    this.title = 'World';
  }
}
```

Важно!

Zone.js не може да определи в кой точно компонент е настъпила промяната, той само сигнализира, че може да е настъпила промяна някъде в приложението, затова се извършва проверка на всички компоненти.

Change Detection strategies

Change Detection strategies в Angular определят как и кога Angular ще проверява дали съществуват промени в състоянието на компонентите и ще актуализира изгледа.

Angular предлага два основни подхода за управление на Change Detection:

- **Default Strategy** - проверява всички компоненти при всяка промяна, започвайки от root компонента
- **OnPush Strategy** - проверява компонента само ако е маркиран за промяна

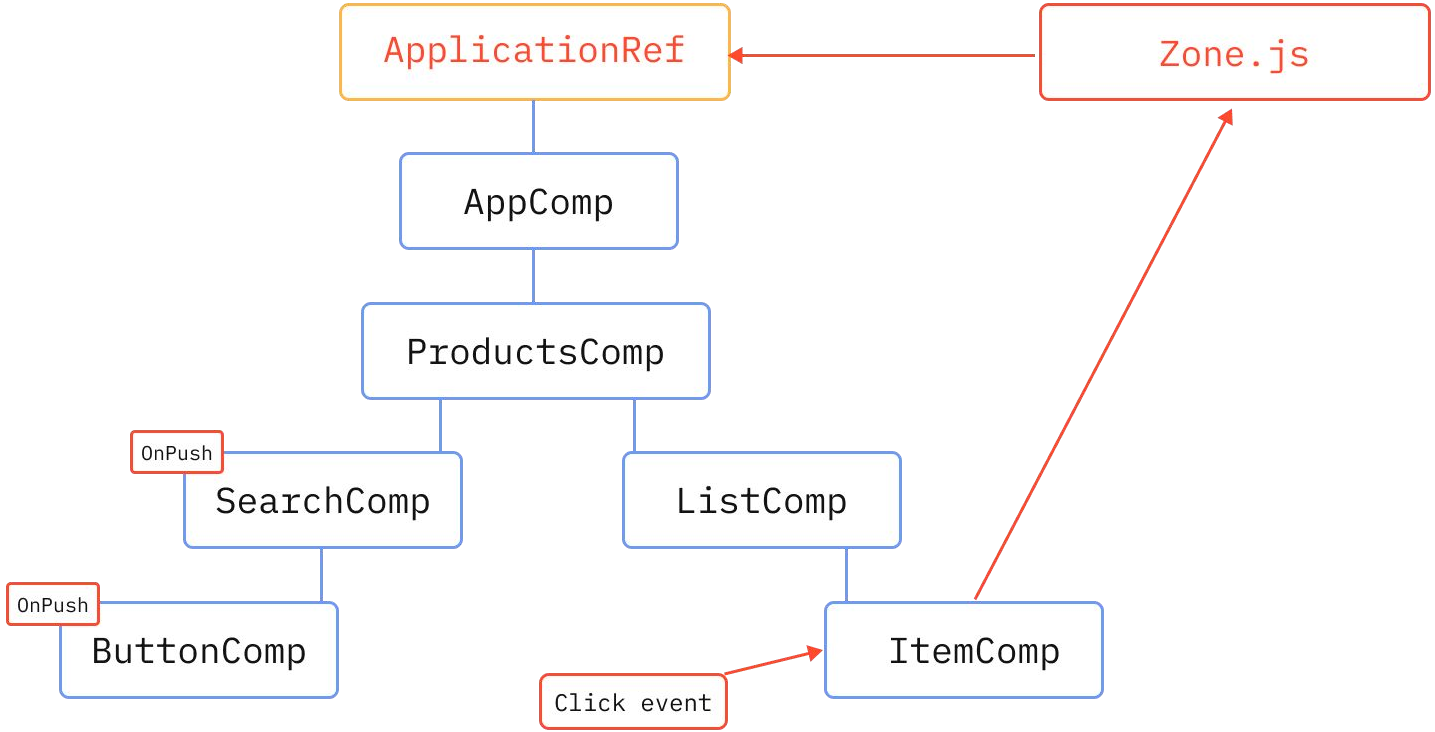
OnPush Strategy

Стратегия, при която компонентът се проверява за промени само когато е маркиран за проверка.

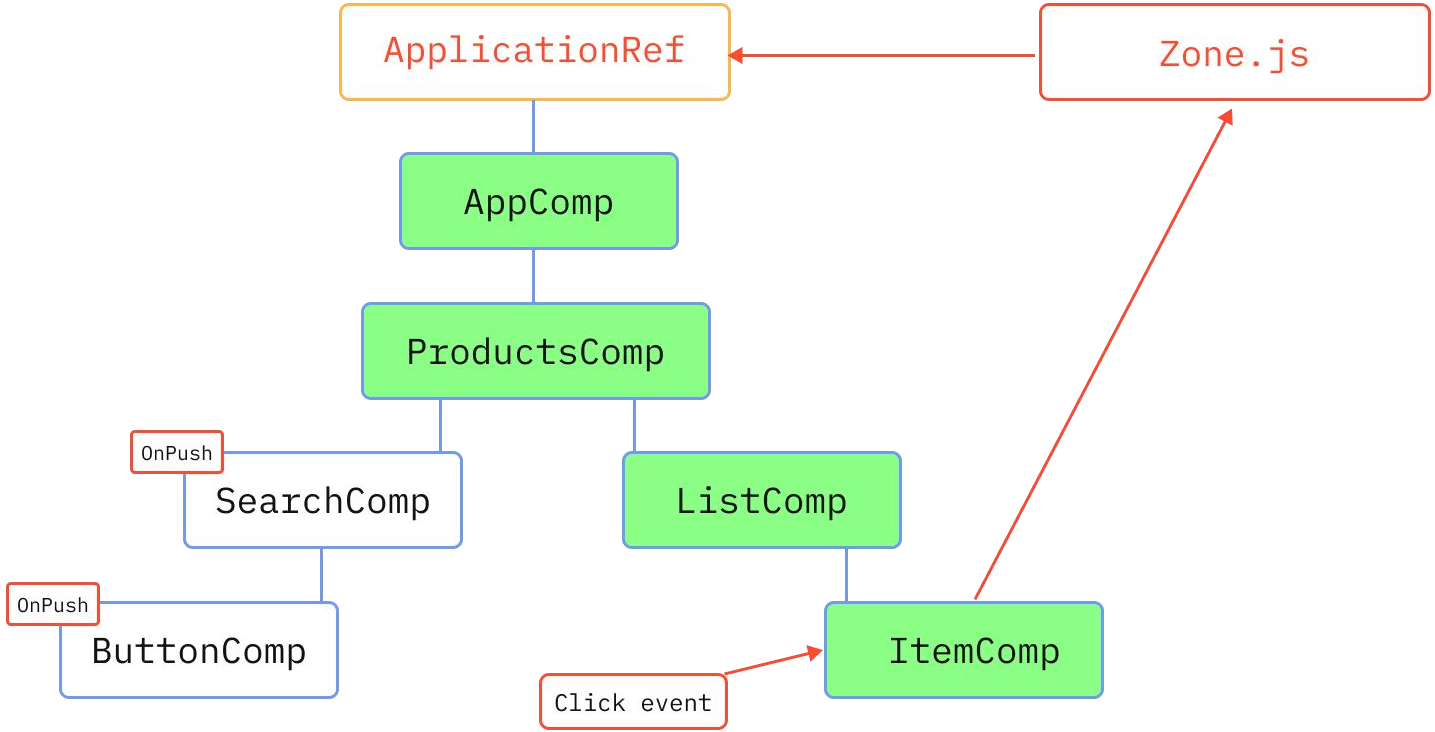
Може да сменим стратегията за Change Detection за даден компонент в Angular, като използваме свойството changeDetection в декоратора на компонента.

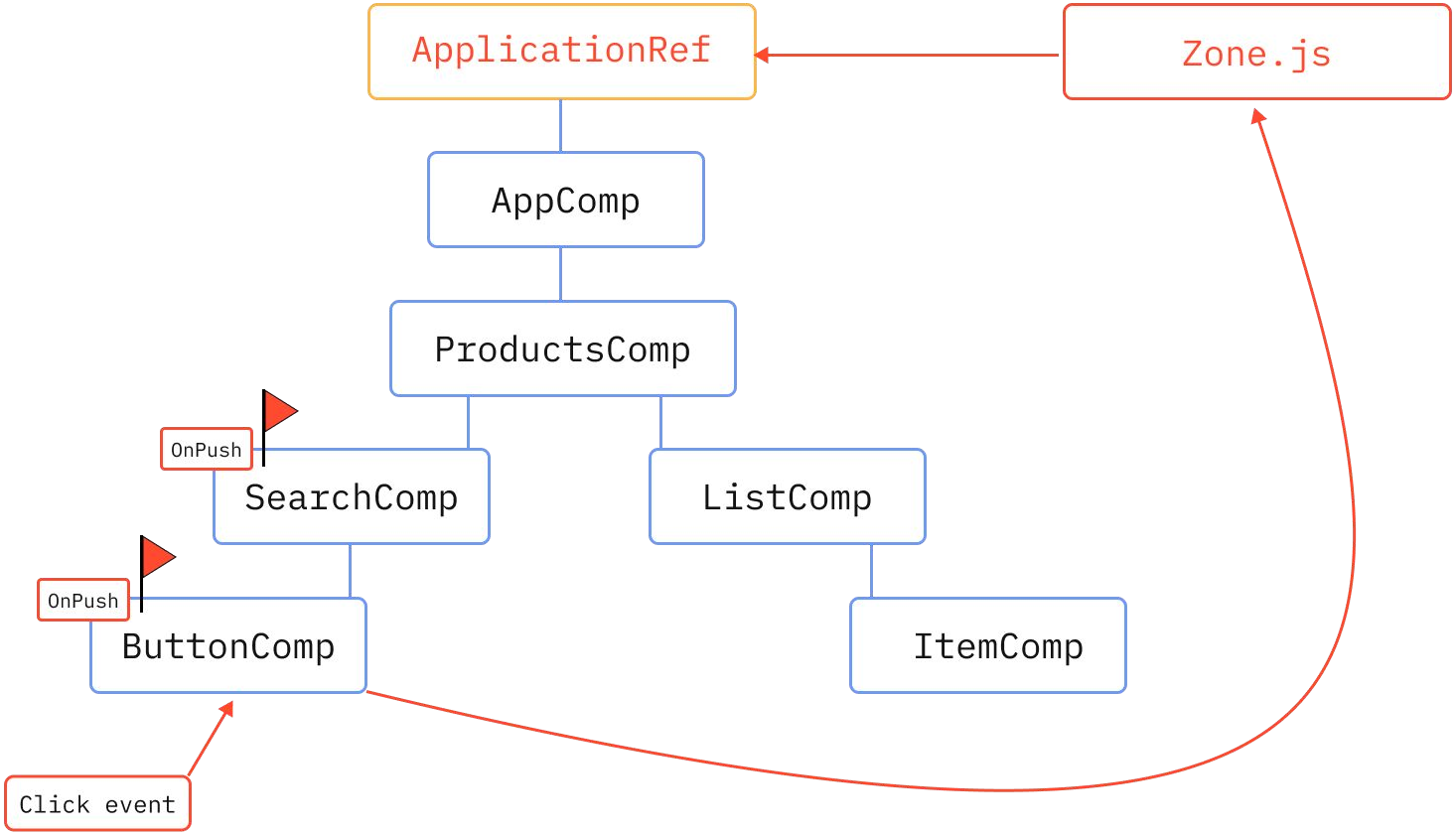
TS app.component.ts

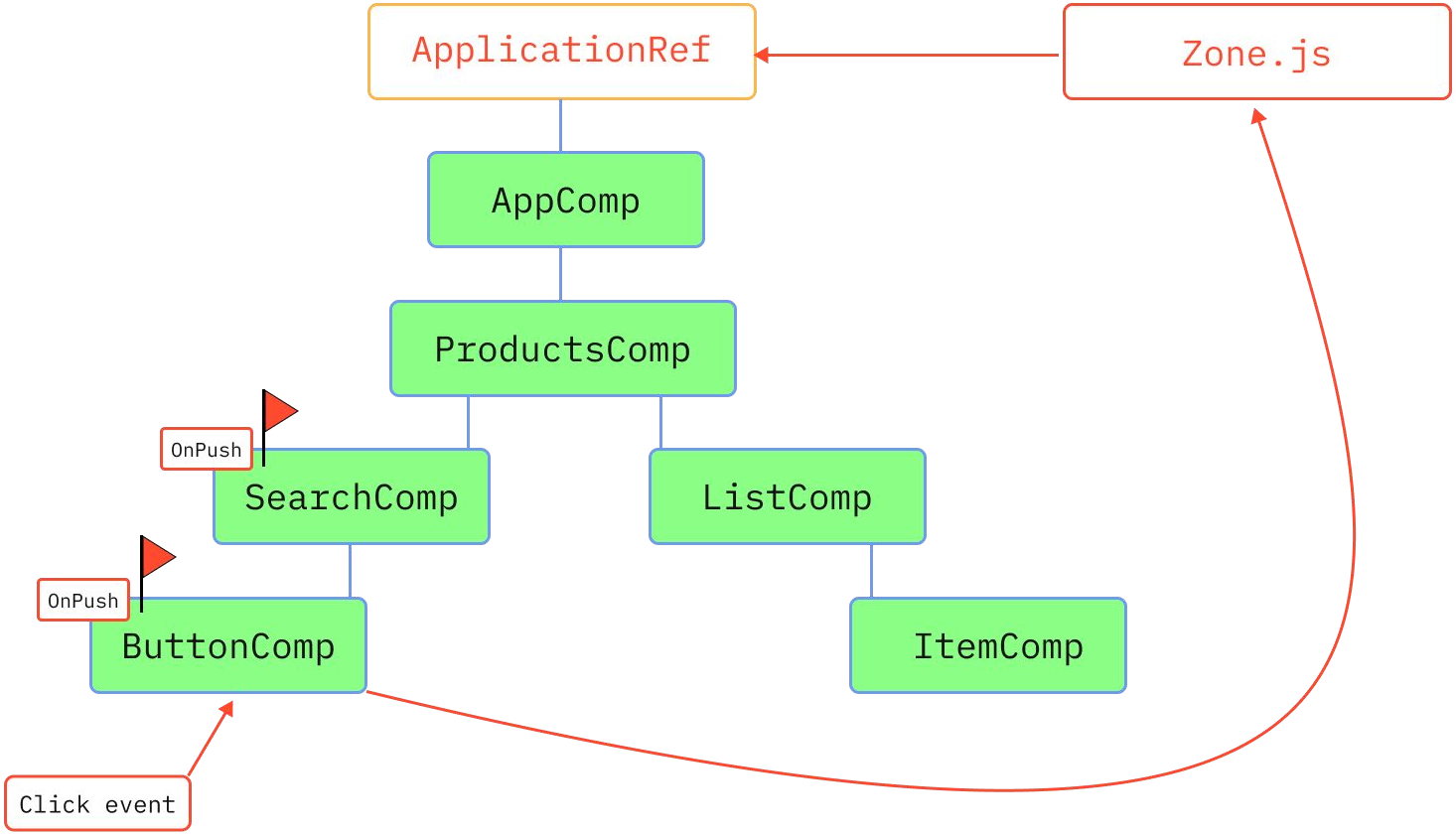
```ts
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div>
      {{ title }}
      <button (click)="updateTitle()">Update</button>
    </div>
  `,
})
export class AppComponent {
  title = 'Hello';

  constructor() {}

  updateTitle() {
    this.title = 'World';
  }
}
```



Проверен за промени



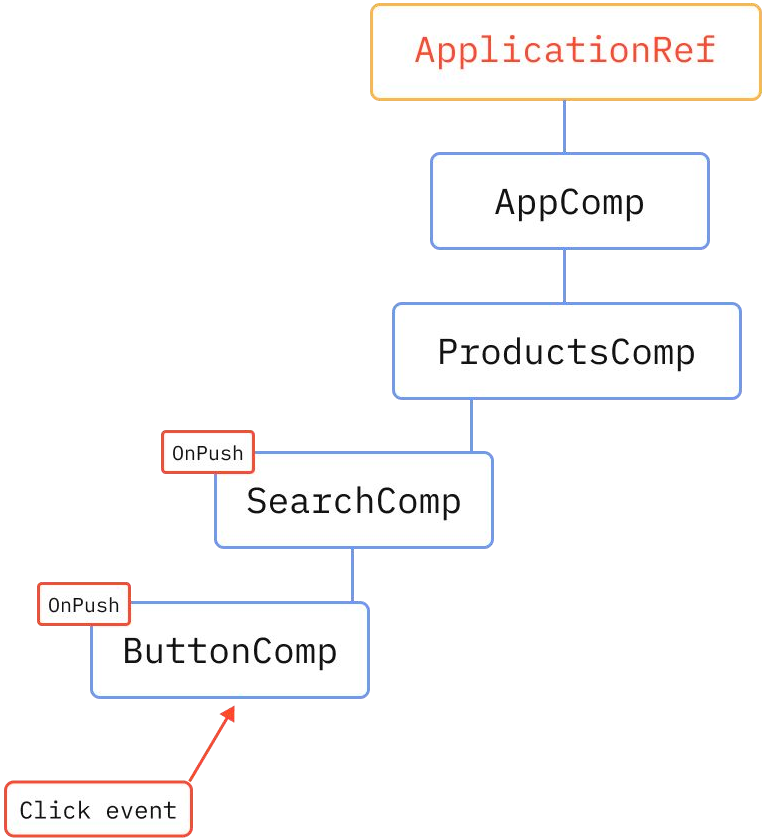


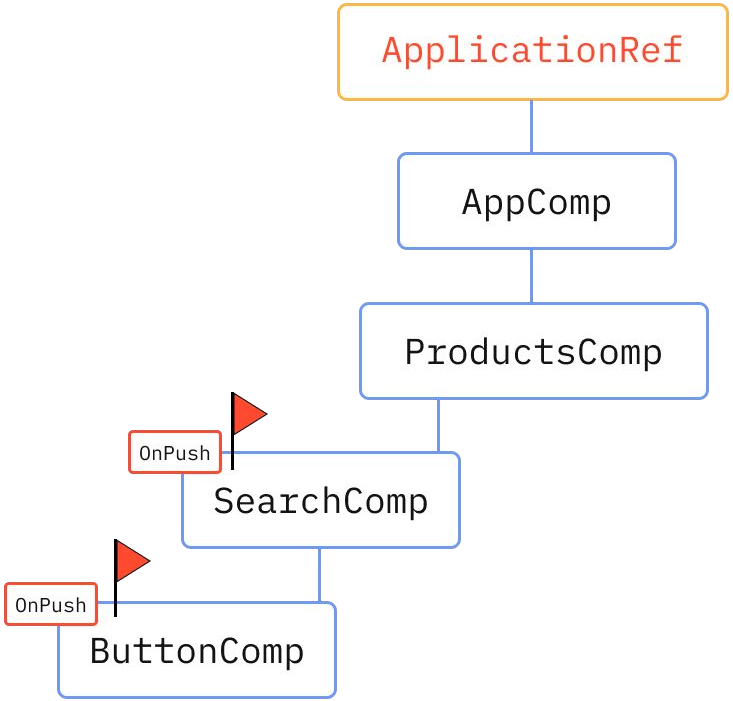


Важно!

Когато маркираме компонент като променен, това също маркира всички негови родителски компоненти, че може да имат промяна и те трябва да бъдат проверени.

- Маркирани за промяна





Как да маркираме компонент за промяна?

Можем да маркираме компонент за change detection с метода **markForCheck** от **ChangeDetectorRef**. Това ще постави компонента в състояние на проверка, така че той да бъде проверен при следващото преминаване през процеса на откриване на промени.

TS app.component.ts

```ts
export class AppComponent {
  title = 'Hello';

  constructor(private cdr: ChangeDetectorRef) {
    setTimeout(() => {
      this.title = 'World';
      this.cdr.markForCheck();
    }, 1000);
  }
}
```

Задейства change detection

Маркира компонента като променен, така че да бъде проверен при следващото изпълнение на change detection.

Автоматично маркиране за промяна

Има няколко случая, в които компонентите се маркират автоматично за промяна и не е необходимо да го правим ръчно:

- При използване на **event handlers**.
- При промяна на входните данни на компонента.
- При използване на **async pipe**

При използване на event handlers

TS app.component.ts

```ts
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div>
      {{ title }}
      <button (click)="updateTitle()">Update</button>
    </div>
  `,
})
export class AppComponent {
  title = 'Hello';

  constructor() {}

  updateTitle() {
    this.title = 'World';
  }
}
```

Всеки event handler е обвит в механизъм, който при извикване на функцията автоматично маркира компонента като 'dirty' (нуждаещ се от промяна)

При промяна на входните данни на компонента

TS app.component.ts

```ts
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ChildComponent],
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div>
      {{ title }}
      <app-child [title]="title"></app-child>
    </div>
  `,
})
export class AppComponent {
  title = 'Hello';

  constructor() {}

  updateTitle() {
    this.title = 'World';
  }
}
```

TS app.component.ts

```ts
export class ChildComponent {
  title = input('');
}
```

По време на проверка на родителския компонент, ако се установи, че някой child компонент е получил входен параметър с нова стойност, този компонент се маркира за промяна.

Важно!

При проверка за промени в входните данни на компонент, Angular сравнява новите и старите стойности по референция.

При използване на Async pipe

TS app.component.ts

```ts
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [AsyncPipe],
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div>
      {{ count$ | async }}
    </div>
  `,
})
export class AppComponent {
  count$ = interval(1000);

  constructor() {}
}
```

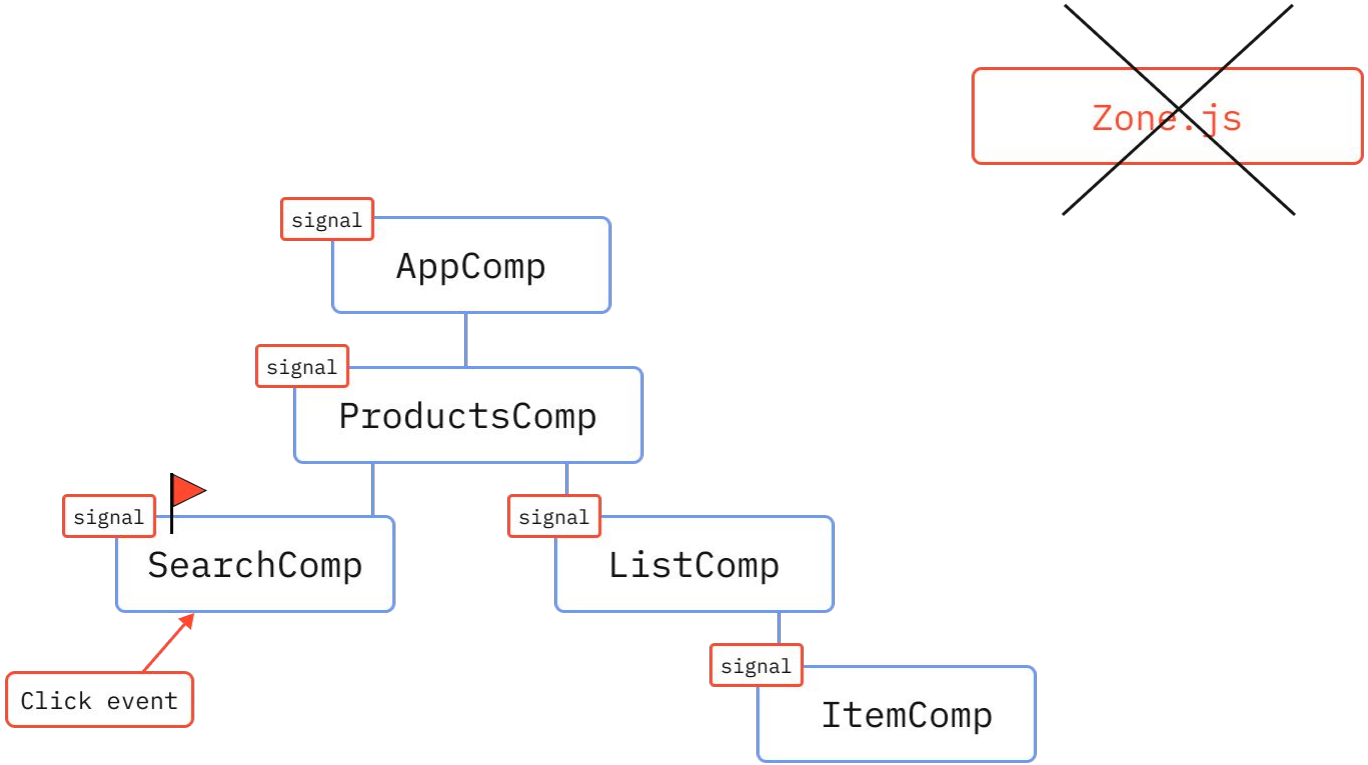
Когато използваме async pipe, всеки път когато observable подаде нова стойност, компонентът автоматично се маркира за промяна.

Angular без Zone.js - Zoneless

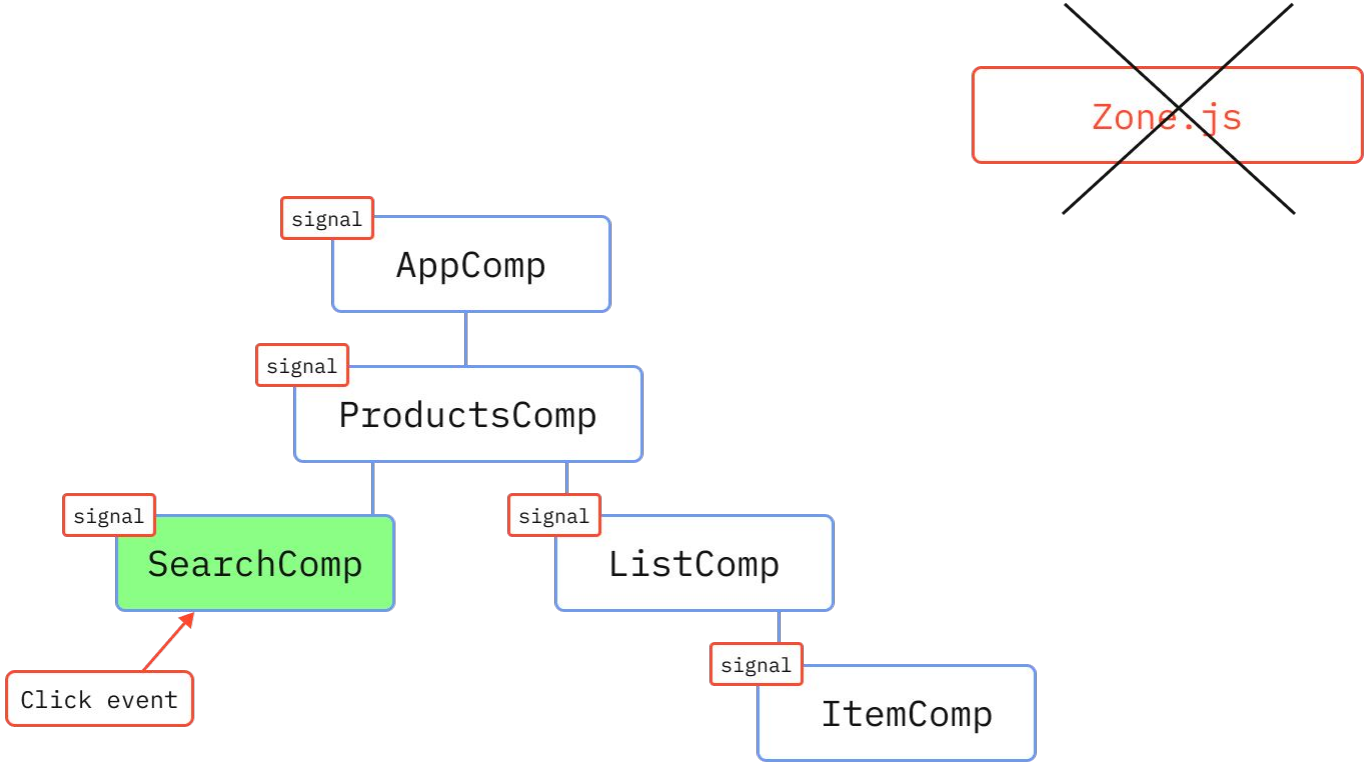
От версия 18 Angular позволява използването на scheduler, който не разчита на zone.js. Този нов scheduler използва signals в темплейта на компонента, за да известява за промени.

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    provideExperimentalZonelessChangeDetection(),
  ],
};
```

Проверен за промени



Проверен за промени



Важно!

Когато даден темплейт прочете сигнал, Angular регистрира този сигнал като зависимост за съответния компонент.

Ако сигналът се промени, Angular получава известие, че трябва да провери и потенциално обнови само компонентите, които използват този сигнал.

По този начин Angular може да определи точно кои компоненти трябва да се проверят и обновят.

Защо да използваме Zoneless ?

- **Подобрена производителност**: - Zone.js често предизвиква ненужни проверки за промени, тъй като няма информация дали състоянието на приложението реално се е променило.
- **По-добри Core Web Vitals** - Zone.js добавя значителен overhead както към размера на payload-а.
- **По-лесно отстраняване на грешки** - Zone.js усложнява разбираемостта на stack trace-овете.
- **По-добра съвместимост с екосистемата** - Zone.js използва monkey patching на браузърни API, което води до несъвместимост с нови и сложни API като async/await.

Web Workers

Какво са Web Workers?

Web Workers са скриптове, които изпълняват JavaScript код на заден фон, отделен от главния thread на браузъра.

Това позволява извършването на тежки изчислителни задачи, без да блокира потребителския интерфейс, като по този начин подобрява производителността на уеб приложенията.

Характеристики на Web Workers

- Работят независимо от главния thread, без достъп до DOM.
- Обмен на данни между Web Worker и главния thread се осъществява чрез обмен на съобщения.
- Web Workers могат да обработват сложни изчисления, без да блокират потребителския интерфейс.

Как се създава Web Worker

Главният thread създава Web Worker, като използва нов обект от тип **Worker**.

Използваме на **postMessage**() за изпращане на данни към Web Worker и **onmessage** за получаване на отговори.

Обработване на съобщение получено от worker-а

Създаваме Worker, като му подаваме пътя към файла.

TS component.ts

```ts
export class BigTaskComponent implements OnInit {
  counter = 0;
  worker!: Worker;

  constructor() {
    setTimeout(() => {
      this.counter++;
    }, 1000);
  }

  ngOnInit(): void {
    this.worker = new Worker(new URL('./big-task.worker', import.meta.url));
    this.worker.onmessage = ({ data }) => {
      console.log('Data', data);
    };
  }

  handleHeavyComputation() {
    this.worker.postMessage('start computation');
  }
}
```

TS worker.ts

```ts
/// <reference lib="webworker" />

addEventListener('message', ({ data }) => {
  const response = `worker response to ${data}`;
  postMessage(response);
});
```

Изпращане на съобщение към worker-а

Изпращане на съобщение от worker-а обратно към главния thread

Ограничения на Web Workers

- Няма достъп до DOM - Web Workers не могат директно да манипулират DOM.
- Ограничена поддръжка на браузъри - Някои браузъри имат ограничения за брой или размер на Web Workers.
- В Angular, при използване на SSR (Server-Side Rendering), Web Workers не могат да бъдат използвани.

Важно!

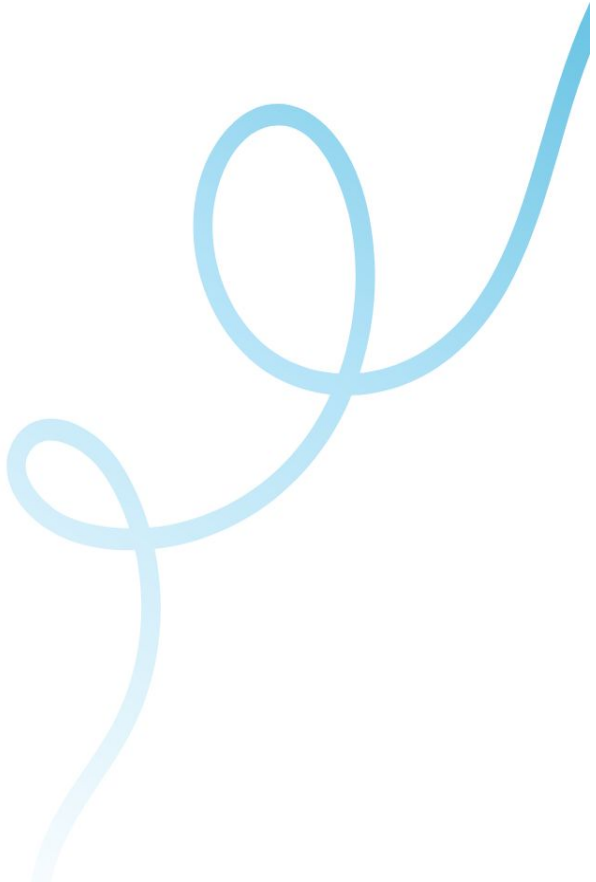
Когато прехвърляме данни между главния процес и фоновия, те се копират чрез structuredClone, което създава deep копие на данните.

Ако не искаме да създаваме копие на данните, можем да използваме transferable objects. При предаване между процесите те не се копират, а просто променят контекста си.

Transferable objects

- `ArrayBuffer`
- `MessagePort`
- `ReadableStream`
- `WritableStream`
- `TransformStream`
- `WebTransportReceiveStream`
- `WebTransportSendStream`
- `AudioData`
- `ImageBitmap`
- `VideoFrame`
- `OffscreenCanvas`
- `RTCDataChannel`
- `MediaSourceHandle`
- `MIDIAccess`

Упражнение



Server-Side Rendering

Какво е Server-Side Rendering ?

Процес, при който уеб страниците се рендерират на сървъра, преди да бъдат изпратени към браузъра. Сървърът генерира HTML, който включва началното състояние на страницата, така че потребителят веднага вижда съдържанието.

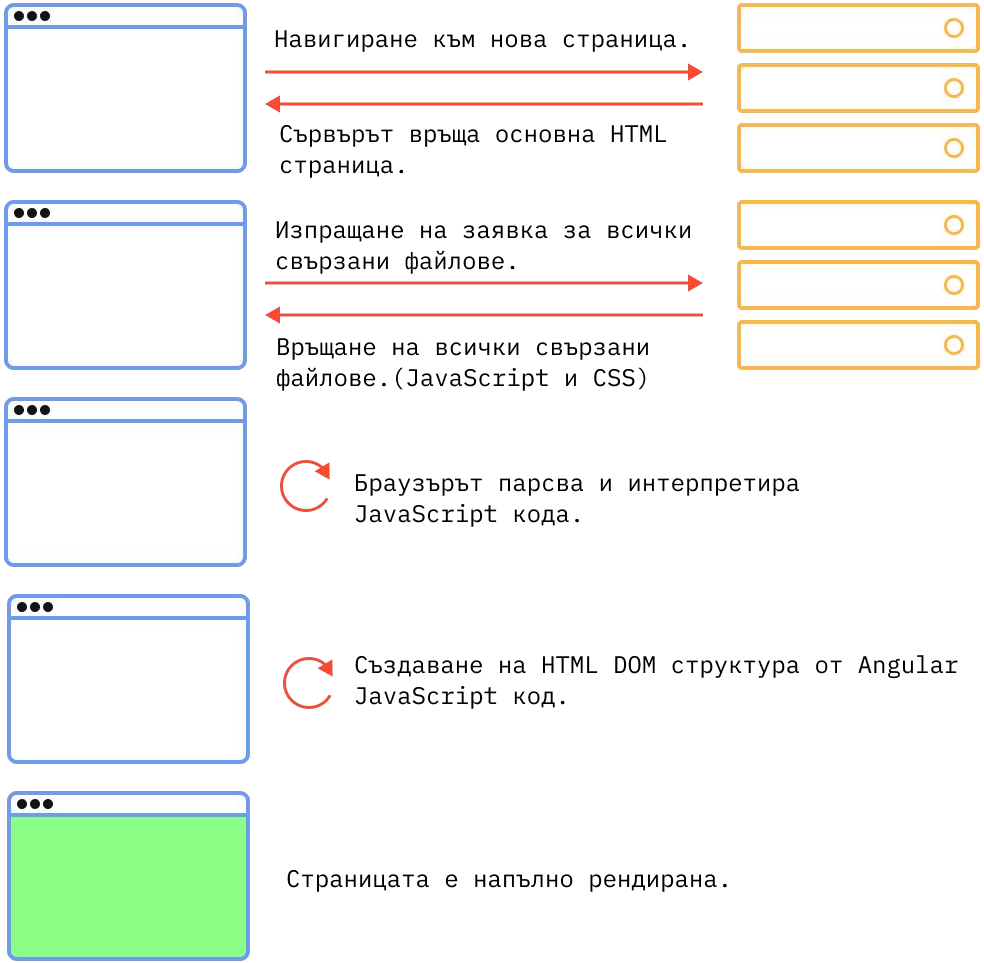
След като HTML е зареден в браузъра, Angular "инициализира" приложението и използва данните от предоставения HTML, за да направи страницата интерактивна. Това осигурява по-бързо първоначално зареждане и подобрява SEO.

CSR vs SSR

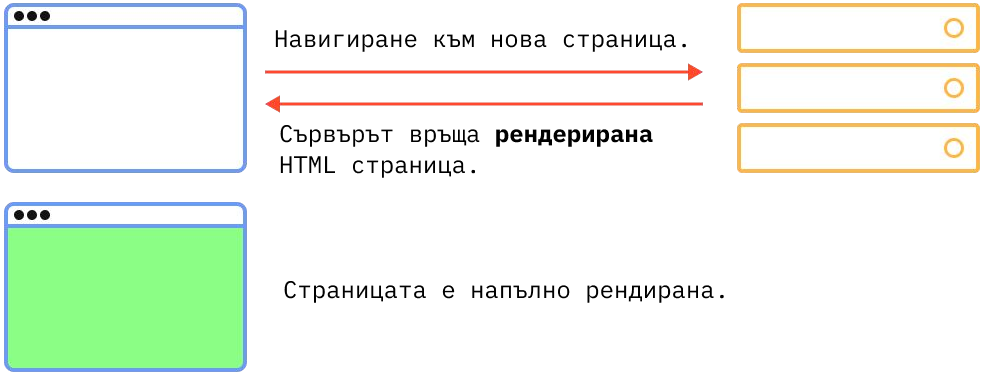
Client-Side Rendering (CSR) е техника за рендиране на уеб страници, при която HTML съдържанието се генерира динамично **в браузъра** с помощта на JavaScript.

Server-Side Rendering (SSR) е техника за рендиране на уеб страници, при която HTML съдържанието се генерира **на сървъра** и се изпраща готово до браузъра.

CSR



SSR



CSR vs SSR

При **CSR** първоначалното зареждане на приложението може да отнеме повече време, ако приложението е по-голямо, тъй като файловете, които трябва да се изтеглят, ще бъдат по-обемни.

Защо да използваме SSR?

- **Performance**- Доставя напълно рендиран HTML на клиента, който браузърът може да парсва и показва още преди да изтегли JavaScript на приложението.

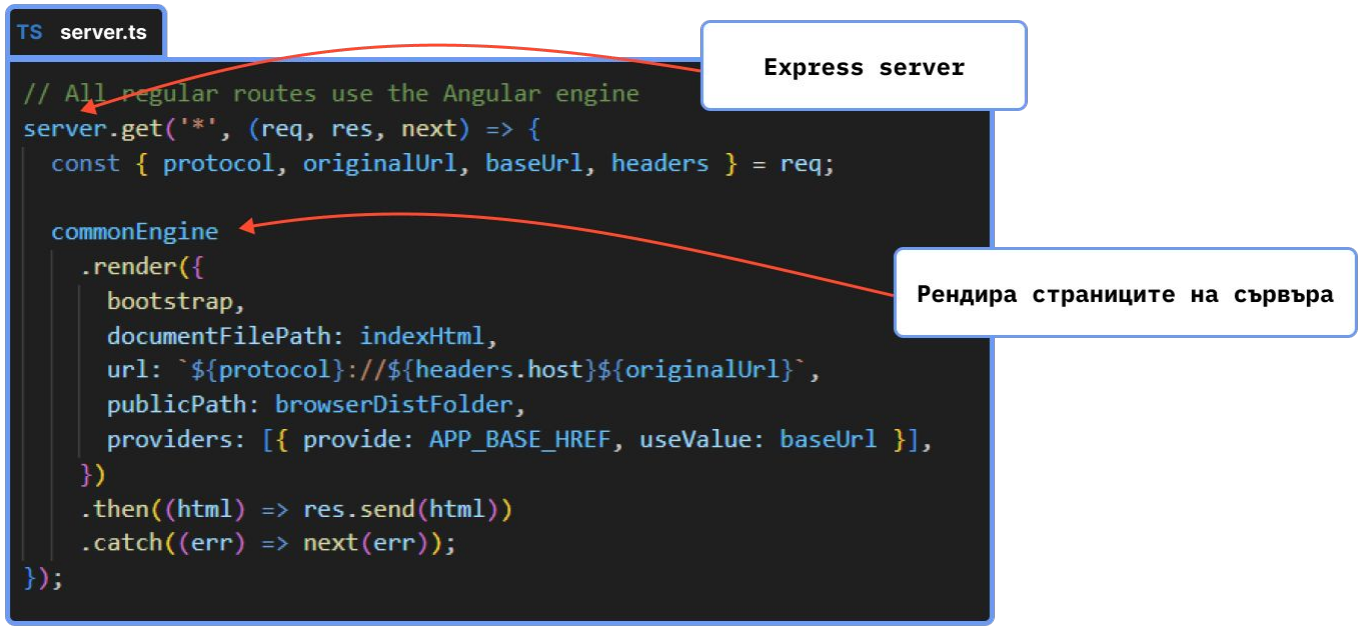
- **Подобрени Core Web Vitals** - SSR води до подобрения в производителността, които могат да бъдат измерени чрез статистики за Core Web Vitals (CWV), като намален First Contentful Paint (FCP), Largest Contentful Paint (LCP) и Cumulative Layout Shift (CLS).

- **По-добро SEO** - улеснява търсачките да индексират и обхождат съдържанието на приложението.

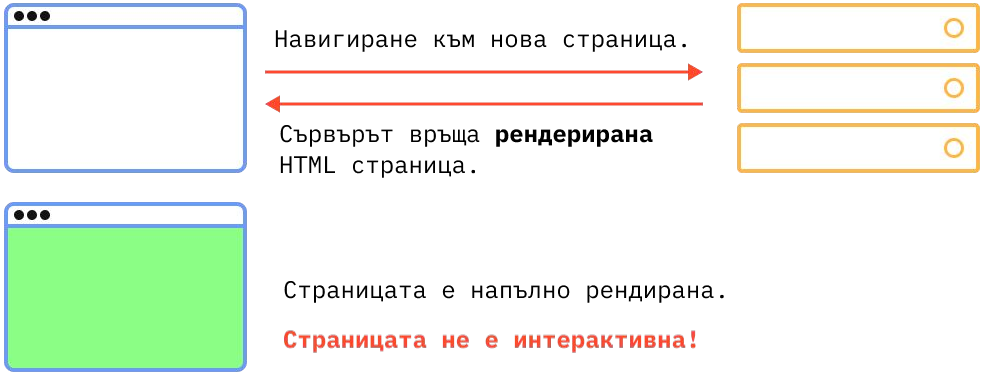
SSR в Angular

Angular използва **Express** сървър заедно с **CommonEngine** за рендиране на приложението на сървъра.

SSR може да бъде добавен към съществуващо Angular приложение чрез командата **ng add @angular/ssr**, или да се създаде ново приложение с SSR чрез **ng new --ssr**.



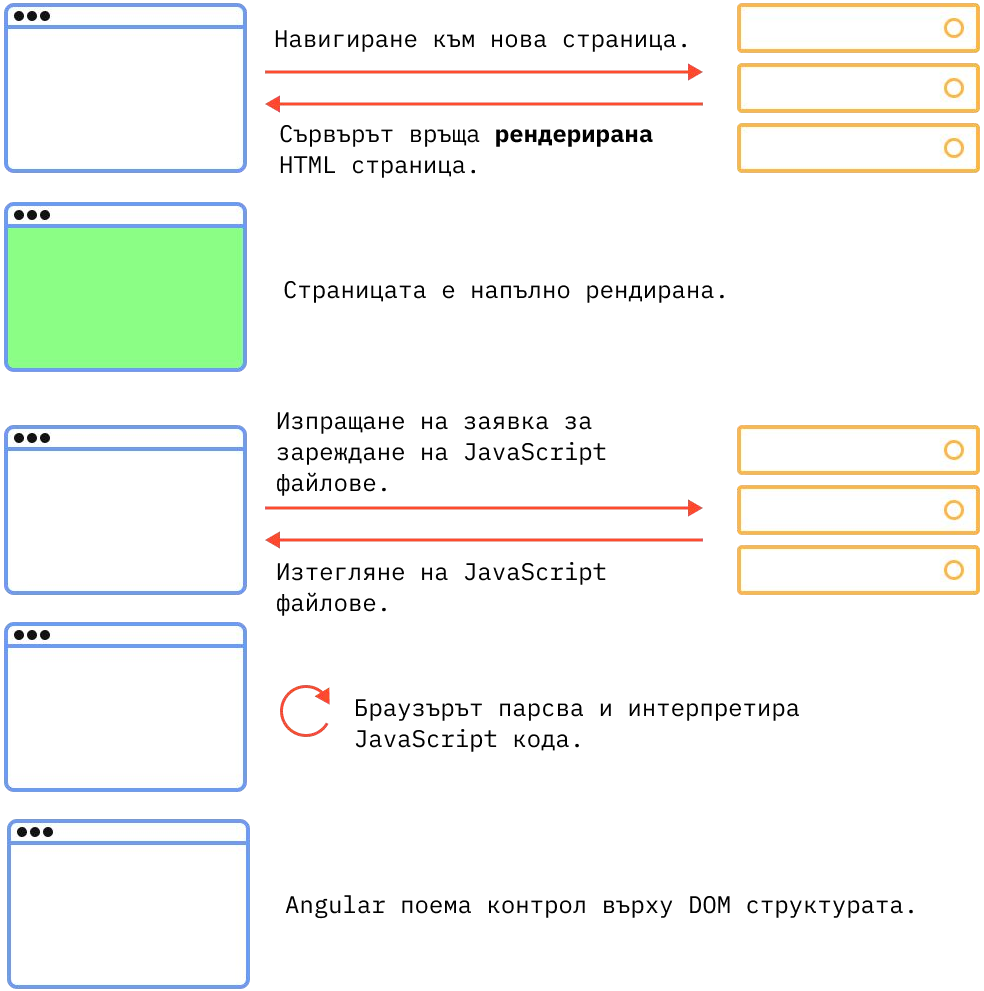
Проблем при SSR



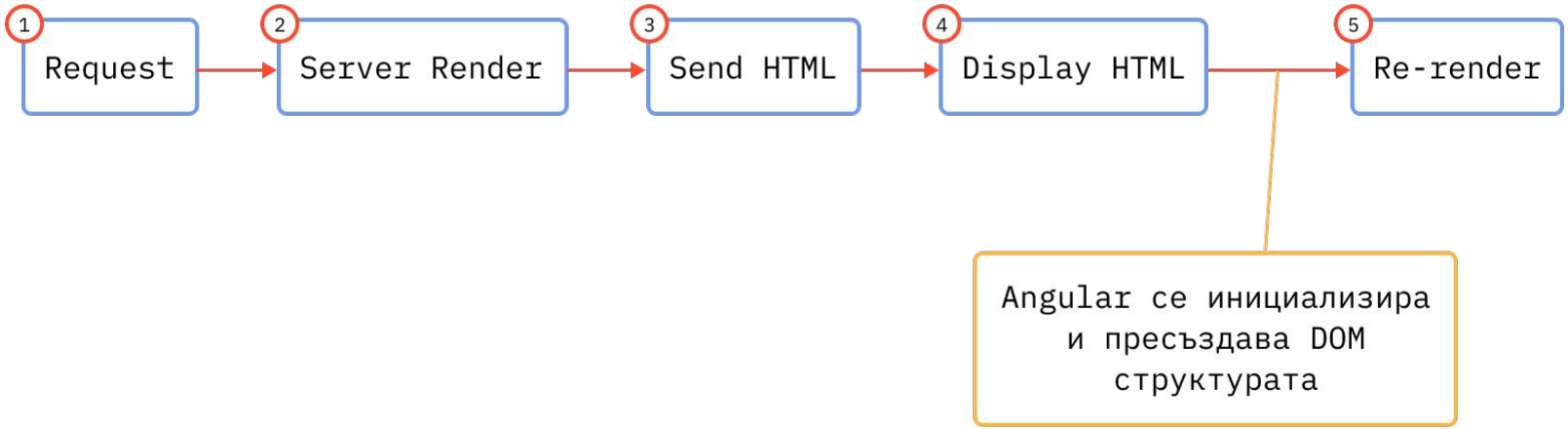
Hydration

Процесът, при който JavaScript кодът на клиента поема контрол върху предварително рендираната HTML страница, генерирана на сървъра.

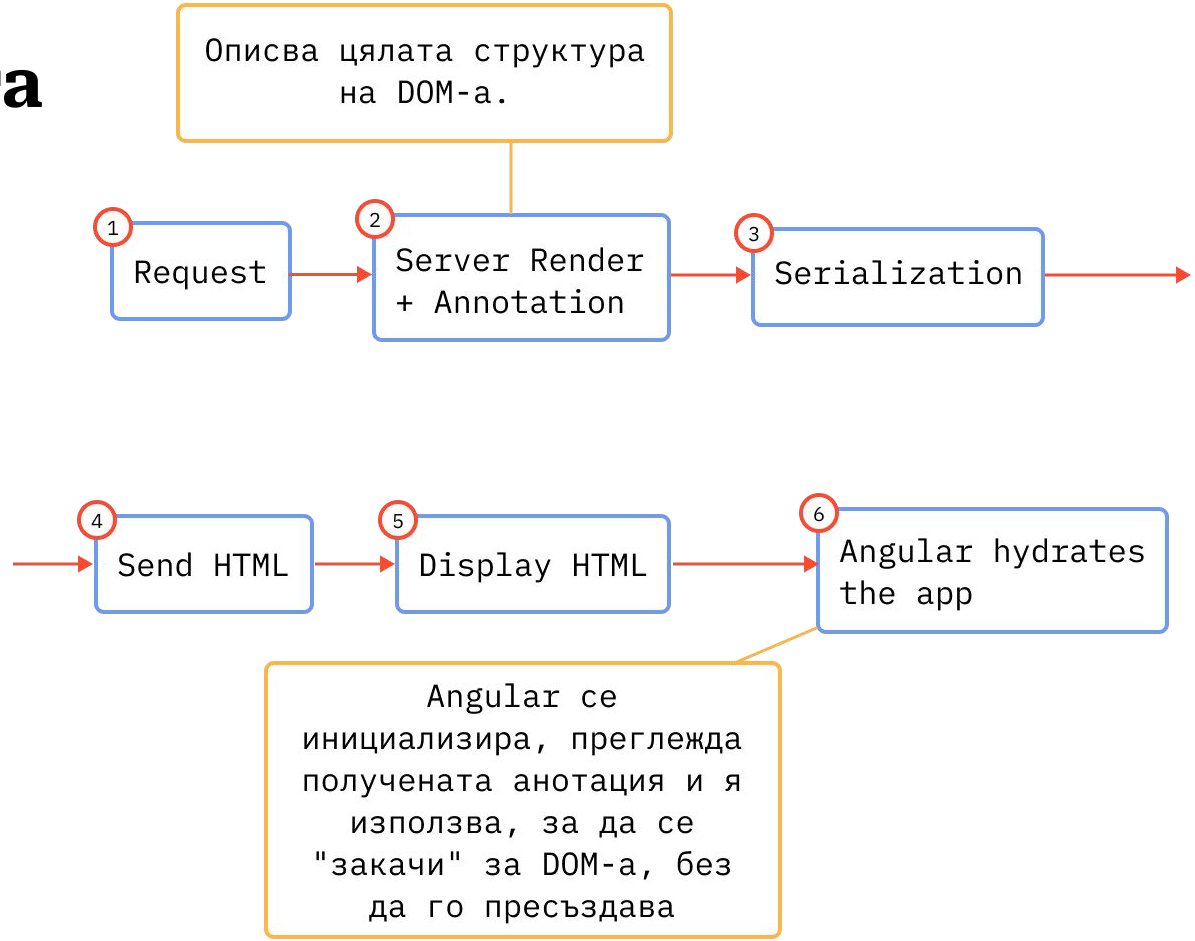
По време на хидратирането браузърът изпълнява JavaScript код, който възстановява функционалността, необходима за динамично поведение в приложението.



SSR преди



SSR сега



Как да използваме Hydration ?

TS app.config.ts

```ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    provideClientHydration()
  ]
};
```

Важно!

HttpClient кешира изходящите мрежови заявки при изпълнение на сървъра. Тази информация се сериализира и прехвърля в браузъра като част от началния HTML.

В браузъра **HttpClient** проверява дали има данни в кеша и ако има, ги използва вместо да прави нова HTTP заявка при първоначалното рендиране на приложението.

Ограничения

Генерираната DOM структура трябва да бъде една и съща както на сървъра, така и в клиента.

Използването на DOM API може да доведе до грешки при хидратиране, тъй като Angular не е наясно с промените в DOM. За да избегнете проблеми, използвайте Angular API вместо директна манипулация с DOM.

Използване на DOM API при SSR

Код, който разчита на DOM API, трябва да се изпълнява само в браузъра. Това може да се гарантира чрез използването на жизнените цикли **afterRender** и **afterNextRender**, които се изпълняват само в браузъра и се пропускат на сървъра.

afterNextRender

TS app.component.ts

```ts
export class AppComponent {
  @ViewChild('content') private _contentRef!: ElementRef;
  private _resizeObserver: ResizeObserver | null = null;

  constructor() {
    afterNextRender(() => {
      this._resizeObserver = new ResizeObserver(() => {
        console.log('Component was resized');
      });
      this._resizeObserver.observe(this._contentRef.nativeElement);
    });
  }
}
```

Използва се за еднократна инициализация на външни библиотеки или API, които работят само в браузъра.

afterRender

TS app.component.ts

```ts
export class AppComponent {
  @ViewChild('content') private _contentRef!: ElementRef;

  constructor() {
    afterRender(() => {
      const element = this._contentRef.nativeElement;
      console.log(
        `After render, content position: (${element.offsetLeft}, ${element.offsetTop})`
      );
    });
  }
}
```

Използва се за синхронизиране с DOM-а и се извиква след всеки change detection

Пропускане на hydration

Angular ни дава възможност да премахнем стъпката с hydration за специфични компоненти като им добавим **ngSkipHydratio** като агрумент.

Атрибутът **ngSkipHydration** кара Angular да пропусне хидратирането на компонента и неговите деца, като го прави да се държи, сякаш хидратирането не е активно, т.е. компонентът ще бъде унищожен и повторно създаден.

```
<app-nav-bar ngSkipHydration></app-nav-bar>
```

Важно!

Ако добавим атрибута ngSkipHydration към root компонента, това е еквивалентно на деактивиране на hydration за цялото приложение.

Prerendering/Static Site Generation (SSG)

Техника за рендиране на уеб страници, при която HTML съдържанието се генерира на сървъра по време на **build process**.

Static Site Generation (SSG) се добавя автоматично към Angular приложение, когато се активира SSR.

```
ng add @angular/ssr
```

{} angular.json

```
"server": "src/main.server.ts",
"prerender": true,
"ssr": {
  "entry": "server.ts"
}
```

Активиране на Static Site Generation

Важно!

По подразбиране, Angular автоматично открива маршрутите за pre-rendering, но можем да променим това поведение и да зададем персонализирани маршрути за pre-rendering.

Options	Details	Default Value
`discoverRoutes`	Whether the builder should process the Angular Router configuration to find all unparameterized routes and prerender them.	`true`
`routesFile`	The path to a file that contains a list of all routes to prerender, separated by newlines. This option is useful if you want to prerender routes with parameterized URLs.	

{} angular.json

```
"server": "src/main.server.ts",
"prerender": {
  "discoverRoutes": false,
  "routesFile": "routes.txt"
},
"ssr": {
  "entry": "server.ts"
}
```

routes.txt

```
product/1
product/2
product/3
```

TS app.routes.ts

```
export const routes: Routes = [
  {
    path: 'product/:productId',
    component: ProductComponent,
  },
];
```

SSR vs SSG

- При SSG не се нуждаем от Node сървър
- При SSG страниците са генерирани веднъж при build, а не при всяка заявка към сървъра.

Кога да използваме SSG?

- Когато съдържанието на сайта не се променя често или е предвидимо.
- Статичните сайтове могат да се хостват на CDN, без да се изисква динамично рендиране от сървър.
- Подобрена SEO оптимизация.

Благодаря за вниманието!