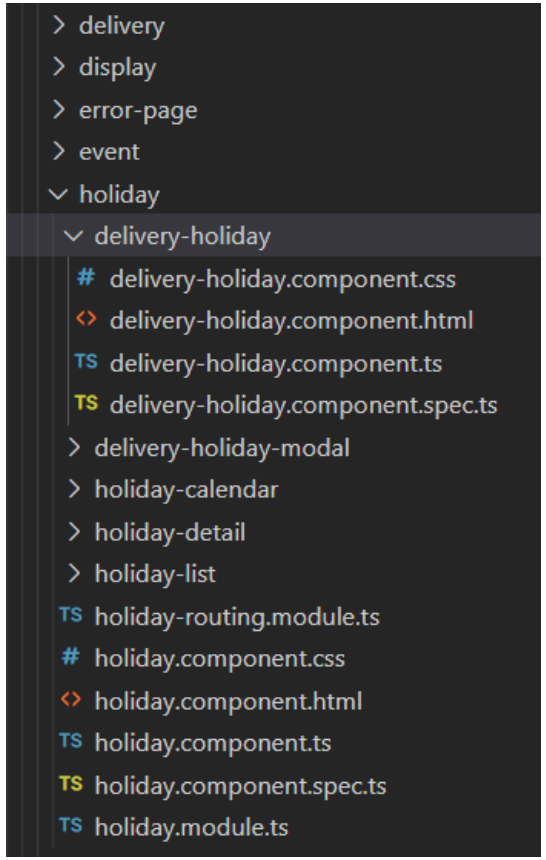
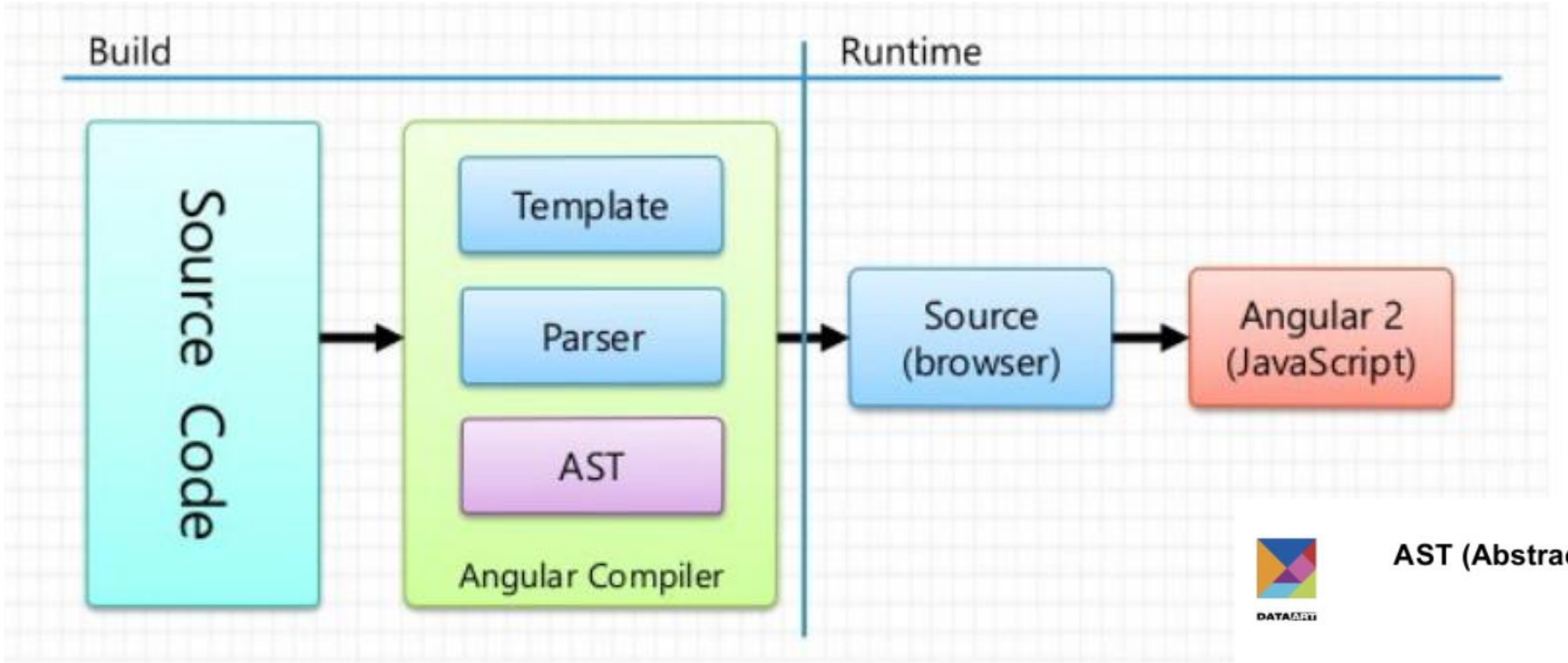
AOT : Angular 컴파일러

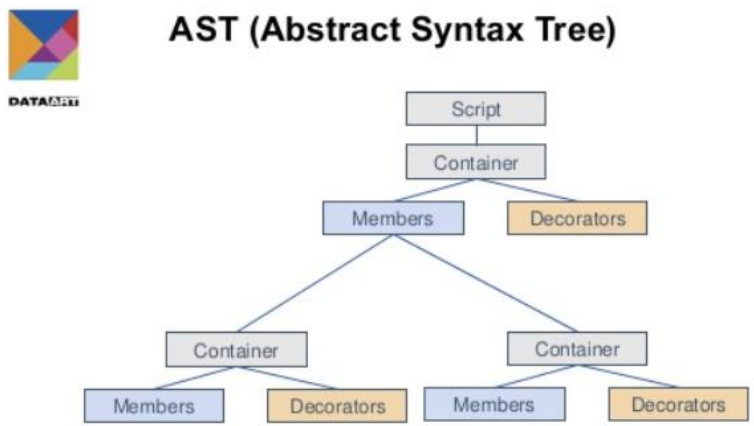
앵귤러 구성




```
import { environment } from 'src/environments/environment';

@Component({
  selector: 'app-bcnc-detail',
  templateUrl: './bcnc-detail.component.html',
  styleUrls: ['./bcnc-detail.component.css']
})
export class BcncDetailComponent implements OnInit {

  token = this.cookie.get('accessTokenCms')
  acceptVersion = '1.0'
  imgUrl = environment.imgUrl
  ApiUrl = environment.ApiUrl

  bcncId

  constructor(
    private router: Router,
    private bcncService: BcncService,
```

크게, 컴포턴트 + 컴포넌트 HTML 템플릿





출처 : https://www.slideshare.net/fwdays/how-to-improve-angular-2-performance

* AST(abstract syntax tree) : 추상 구문 트리. 소스 코드의 추상 구문 구조의 트리이다. 이 트리의 각 노드는 소스 코드에서 발생되는 구조를 나타낸다.

장점

- 렌더링이 빠르다
- 비동기 요청횟수가 줄어든다
- 내려받아야 할 Angular 프레임워크 용량이 줄어든다
- 템플릿 에러를 더 빠르게 발견할 수 있다
- 안정성이 있다

Angular 컴파일 방식

- JIT(Just-in-Time)
  브라우저에서 애플리케이션을 실행하면서 코드를 직접 컴파일(Angular8 까지 기본 컴파일러)

- AOT(Ahead-of-Time)
  브라우저에 애플리케이션 코드를 보내기 전에 미리 컴파일하는 방식(Angular9 부터 기본 컴파일러)

AOT 컴파일러가 동작하는 방식

- 컴파일러는 애플리케이션을 구성하는 개별 요소를 관리하기 위해 **코드에서 metadata를 추출**

  (metadata는 @Component() 나 @Input() 과 같이 데코레이터를 사용해서 명시적으로 지정 가능)

- Angular는 이 metadata에 지정된 내용을 바탕으로 애플리케이션 클래스의 인스터스를 구성

* 데코레이터 : JavaScript 클래스를 변형하는 함수. Angular는 클래스가 어떤 특징을 가지며 어떻게 동작해야 하는지 메타데이터를 사용하는 데코레이터를 여러 개 구현해두고 있음.

AOT 컴파일러가 동작하는 방식

```
@Component({
  selector: 'app-caution-desc-modal',
  templateUrl: './caution-desc-modal.component.html',
  styleUrls: ['./caution-desc-modal.component.css']
})
export class CautionDescModalComponent implements OnInit {

  @Input() orderId

  constructor(
    private router: Router,
    private orderService: OrderService,
    private route: ActivatedRoute,
    private spinner: NgxSpinnerService,
    private activeModal: NgbActiveModal

  ) { }
```

- 이 코드를 Angular 컴파일러가 처리하면 메타데이터를 추출해서 위 component에 대한 팩토리를 만든다
- 위 component의 인스턴스가 필요한 시점에 Angular 가 팩토리를 실행해서 인스터스를 생성하며, 이렇게 생성된 인스턴스를 의존성으로 주입

메타데이터의 제약사항

- JavaScript 문법 중 표현식(expression syntax)은 일부만 사용 가능
- "코드를 폴딩"한 이후에 존재하는 심볼만 참조 가능
- 컴파일러가 지원하는 일부 함수만 사용 가능
- 데코레이터가 사용되거나 데이터 바인딩되는 클래스 멤버는 public으로 지정되어야 함

컴파일 단계

1. 코드 분석

   TypeScript 컴파일러와 AOT 콜렉터가 소스코드에서 필요한 정보를 수집

2. 코드 생성

   1단계에서 수집한 메타데이터를 컴파일러의 StaticReflector가 처리하면서 메타데이터의 유효성을 추가로 검사

3. 템플릿 문법 체크

   Angular *템플릿 컴파일러*가 TypeScript 컴파일러를 사용해서 템플릿에 사용된 바인딩 표현식을 검증

1단계: 분석

- TypeScript 컴파일러가 코드를 컴파일하고 나면 '*타입 정의 파일*' 인 .d.ts 파일이 생성

  => 이 정보는 나중에 AOT 컴파일러가 애플리케이션 코드를 생성할 때 사용

- *AOT collector* 가 각 .d.ts 파일에 있는 Angular 데코레이터의 메타데이터를 분석하고 분석한 내용을 .metadata.json 파일로 생성

1단계: 분석 – 표현식의 한계

- Angular *collector* 는 JavaScript 의 하위집합이며 JavaScript 문법 중 일부만 가능
  => 메타데이터에는 다음과 같은 문법만 허용

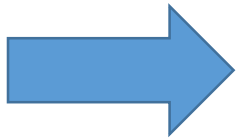
문법	예
객체 리터럴	{cherry: true, apple: true, mincemeat: false}
배열 리터럴	['cherries', 'flour', 'sugar']
배열 안에 사용된 전개 연산자	['apples', 'flour', ...the_rest]
함수 실행	bake(ingredients)
New	new Oven()
프로퍼티 참조	pie.slice
배열 인덱스 참조	ingredients[0]
타입 참조	Component
템플릿 문자열	`pie is ${multiplier} times better than cake`

문법	예
문자열 리터럴	pi
숫자 리터럴	3.141592
불리언 리터럴	true
Null 리터럴	null
접두사 연산자	!cake
바이너리 연산자	a+b
조건 연산자	a ? b : c
괄호	(a+b)

1단계: 분석 – 화살표 함수 사용 불가

- AOT 컴파일러는 *함수 표현식* 과 *화살표 함수(람다 함수)* 를 지원하지 않음

```
@Component({
        ...
        providers: [{provide: server, useFactory: () => new Server() }]
})
```



```
export function serverFactory() {
        return new Server();
}

@Component({
        ...
        providers: [{provide: server, useFactory: serverFactory}]
})
```

* Angular 5 버전 부터는 .js 파일을 생성할 때 이 문제를 자동을 처리

1단계: 분석 – 코드 폴딩(Code Folding)

- AOT 컴파일러는 **export** 키워드가 사용된 심볼만 참조 가능

  But, 콜렉터는 *폴딩* 이라는 것을 통해 export 키워드가 사용되지 않은 심볼도 제한적으로 참조 가능

- 콜렉터는 콜렉션 단계에서 표현식을 평가하고 그 결과를 .metadata.json 파일에 기록, 이 때 원래 코드를 약간 변형해서 기록

  ex. 1+2+3+4 라는 표현식을 평가하고 나면 .metadata.json 파일에는 이 내용을 10으로 기록

  -> **이 과정이 폴딩(folding), 이 과정이 적용될 수 있는 코드를 폴딩할 수 있는(foldable) 코드**

1단계: 분석 – 코드 폴딩(Code Folding)

```
const template = '<div>{{hero.name}}</div>';

@Component({
        selector: 'app-hero',
        template: template
})
export class HeroComponent {
        @Input() hero: Hero;
}
```

콜렉터는 template 변수를 폴딩해서 컴포넌트 메타데이터 안으로

```
@Component({
        selector: 'app-hero',
        template: '<div>{{hero.name}}</div>' })
export class HeroComponent {
        @Input() hero: Hero;
}
```

template 이라는 변수가 없고, 콜렉터가 생성한
.metadata.json 파일을 사용하는 컴파일러도 정상적으로 실행

2단계: 생성

- *콜렉터*는
  메타데이터를 이해하는 것 **X**
  메타데이터를 찾아서 .metadata.json에 모으는 역할 **O**

- .metadata.json 파일을 해석해서 코드를 생성하는 것은 컴파일러의 역할
- 콜렉터가 처리할 수 있는 문법은 컴파일러도 모두 처리 가능

2단계: 생성 – public 심볼

컴파일러는 파일 외부로 오픈된(exported)된 심볼만 참조 가능

- 컴포넌트 클래스 멤버 중 데코레이터가 사용된 멤버는 반드시 public 이어야 함

  => @Input() 프로퍼티도 private이나 protected로 지정되면 안됨

- 데이터 바인딩으로 연결된 프로퍼티도 반드시 public이어야

```
@Component({
        selector: 'app-root',
        template: '<h1>{{title}}</h1>'
})
export class AppComponent {
        private title = 'My App'; // Bad - title이 private으로 지정
}
```

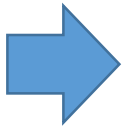
2단계: 생성 – 클래스, 함수 지원

- 콜렉터는 함수 실행이나 new 키워드를 사용한 객체 생성 문법을 지원
  But, 일부 함수나 일부 객체 생성 코드는 컴파일러가 처리하지 않는 경우도 존재
- 컴파일러는 특정 클래스의 인스턴스를 생성하거나 코어 데코레이터만 지원

  - 인스턴스 생성
    - @angular/core 가 제공하는 InjectionToken 클래스의 인스턴스 생성만 가능
  - 사용할 수 있는 데코레이터
    - @angular/core 모듈에 있는 Angular 데코레이터만 지원
  - 함수 실행
    - 팩토리 함수는 반드시 export 로 지정되어야 하며, 함수의 이름이 있어야 함. (람다 표현식은 사용 불가)

2단계: 생성 – 메타데이터 재구축

```
class TypicalServer {

}

@NgModule({
  providers: [{provide: SERVER, useFactory: () => TypicalServer}]
})
export class TypicalModule {}
```



```
class TypicalServer {

}

export const ɵ0 = () => new TypicalServer();

@NgModule({
  providers: [{provide: SERVER, useFactory: ɵ0}]
})
export class TypicalModule {}
```

- 람다 함수 지원 **X**
- TypicalServer 클래스도 export로 지정되지 **X**

3단계: 템플릿 타입 체크

- 템플릿 표현식에 사용된 코드의 타입을 체크
  => 실행 시점에 발생하는 문제로 앱이 종료되는 것 미리 방지 가능
- Angular 템플릿 컴파일러가 TypeScript 컴파일러 활용

```
@Component({
  selector: 'my-component',
  template: '{{person.addresss.street}}'
})
class MyComponent {
  person?: Person;
}
```

[에러 표시]

```
my.component.ts.MyComponent.html(1,1): : Property 'addresss' does not exist on type 'Person'. Did you mean 'address'?
```

3단계: 템플릿 타입 체크 – 타입 구체화

- **ngIf** : TypeScript 코드에 사용하는 if 처럼 타입을 구체화하는 역할

```
@Component({
  selector: 'my-component',
  template: '<span *ngIf="person"> {{person.addresss.street}} </span>'
})
class MyComponent {
  person?: Person;
}
```

* ngIf 를 사용하면 TypeScript 컴파일러가 person 객체의 타입을 추론 가능
=> 이 객체가 undefined라면 바인딩 표현식도 실행되지 않음

3단계: 템플릿 타입 체크 – null 방지 연산자

- person과 address의 값은 동시에 할당 => person만 검사하면 address가 null 이 아닌 것 보장
- But, TypeScript나 템플릿 컴파일러는 이를 알 수 없음 => address 프로퍼티에 'undefined' 에러 발생 가능

```
@Component({
  selector: 'my-component',
  template: '<span *ngIf="person"> {{person.name}} lives on {{address!.street}} </span>'
})
class MyComponent {
  person?: Person;
  address?: Address;

  setData(person: Person, address: Address) {
    this.person = person;
    this.address = address;
  }
}
```

* address!.street
 => address가 null이 아닐 때만 street 프로퍼티를 참조하라

```
'<span *ngIf="person && address"> {{person.name}} lives on {{address.street}} </span>'
```

vs JIT(Just-In-Time)

- 브라우저에서 컴파일
- 소스코드 변경 후 컴파일 할 필요가 없음

*** 도대체 JIT는 왜 존재?**

✓ 개발시 사용(ng serve)
✓ 동적으로 함수나 변수, 필요한 정보를 주입 가능

참고 사이트

- https://medium.com/angular-in-depth/having-fun-with-angular-and-typescript-transformers-2c2296845c56
- https://www.slideshare.net/fwdays/how-to-improve-angular-2-performance
- https://angular.kr/guide/aot-compiler
- https://medium.com/@Sujithnath/angular-aot-vs-jit-comparison-ce1d96ede491
- https://ko.mort-sure.com/blog/angular-aot-vs-jit-comparison-19b266/

감사합니다~