View Data Explanation

`LView` and `TView.data` are how the Ivy renderer keeps track of the internal data needed to render the template. `LView` is designed so that a single array can contain all of the necessary data for the template rendering in a compact form. `TView.data` is a corollary to the `LView` and contains information which can be shared across the template instances.

`LView` / `TView.data` layout.

Both `LView` and `TView.data` are arrays whose indices refer to the same item. For example index `123` may point to a component instance in the `LView` but a component type in `TView.data`.

The layout is as such:

Section	`LView`	`TView.data`
`HEADER`	contextual data	mostly `null`
`DECLS`	DOM, pipe, and local ref instances	
`VARS`	binding values	property names
`EXPANDO`	host bindings; directive instances; providers; dynamic nodes	host prop names; directive tokens; provider tokens; `null`

`HEADER`

`HEADER` is a fixed array size which contains contextual information about the template. Mostly information such as parent `LView`, `Sanitizer`, `TView`, and many more bits of information needed for template rendering.

`DECLS`

`DECLS` contain the DOM elements, pipe instances, and local refs. The size of the `DECLS` section is declared in the property `decls` of the component definition.

```
@Component({
  template: `<div>Hello <b>World</b>!</div>`
})
class MyApp {

  static ɵcmp = ɵɵdefineComponent({
    ...,
    decls: 5,
    template: function(rf: RenderFlags, ctx: MyApp) {
      if (rf & RenderFlags.Create) {
        ɵɵelementStart(0, 'div');
        ɵɵtext(1, 'Hello ');
        ɵɵelementStart(2, 'b');
        ɵɵtext(3, 'World');
        ɵɵelementEnd();
```

```
        ɵɵtext(4, '!');
        ɵɵelementEnd();
      }
      ...
    }
  });
}
```

The above will create following layout:

Index	LView	TView.data
HEADER		
DECLS		
10		{type: Element, index: 10, parent: null}
11	#text(Hello)	{type: Element, index: 11, parent: tView.data[10]}
12	<b>	{type: Element, index: 12, parent: tView.data[10]}
13	#text(World)	{type: Element, index: 13, parent: tView.data[12]}
14	#text(!)	{type: Element, index: 14, parent: tView.data[10]}
...	...	...

NOTE:

- The `10` is not the actual size of `HEADER` but it is left here for simplification.
- `LView` contains DOM instances only
- `TView.data` contains information on relationships such as where the parent is. You need the `TView.data` information to make sense of the `LView` information.

VARS

`VARS` contains information on how to process the bindings. The size of the `VARS` section is declared in the property `vars` of the component definition.

```
@Component({
  template: `<div title="{{name}}">Hello {{name}}!</div>`
})
class MyApp {
  name = 'World';

  static ɵcmp = ɵɵdefineComponent({
    ...,
    decls: 2, // Two DOM Elements.
    vars: 2,   // Two bindings.
    template: function(rf: RenderFlags, ctx: MyApp) {
      if (rf & RenderFlags.Create) {
        ɵɵelementStart(0, 'div');
```

```
        ɵɵtext(1);
        ɵɵelementEnd();
      }
      if (rf & RenderFlags.Update) {
        ɵɵproperty('title', ctx.name);
        ɵɵadvance(1);
        ɵɵtextInterpolate1('Hello ', ctx.name, '!');
      }
      ...
    }
  });
}
```

The above will create following layout:

Index	LView	TView.data
HEADER		
DECLS		
10		{type: Element, index: 10, parent: null}
11	#text()	{type: Element, index: 11, parent: tView.data[10]}
VARS		
12	'World'	'title'
13	'World'	null
...	...	...

NOTE:

- `LView` contain DOM instances and previous binding values only
- `TView.data` contains information on relationships and property labels.

EXPANDO

`EXPANDO` contains information on data which size is not known at compile time. Examples include:

- `Component` / `Directives` since we don't know at compile time which directives will match.
- Host bindings, since until we match the directives it is unclear how many host bindings need to be allocated.

```
@Component({
  template: `<child tooltip></child>`
})
class MyApp {

  static ɵcmp = ɵɵdefineComponent({
    ...,
    decls: 1,
```

```
    template: function(rf: RenderFlags, ctx: MyApp) {
      if (rf & RenderFlags.Create) {
        ɵɵelement(0, 'child', ['tooltip', null]);
      }
      ...
    },
    directives: [Child, Tooltip]
  });
}


@Component({
  selector: 'child',
  ...
})
class Child {
  @HostBinding('tooltip') hostTitle = 'Hello World!';
  static ɵcmp = ɵɵdefineComponent({
    ...
    hostVars: 1
  });
  ...
}

@Directive({
  selector: '[tooltip]'
})
class Tooltip {
  @HostBinding('title') hostTitle = 'greeting';
  static ɵdir = ɵɵdefineDirective({
    ...
    hostVars: 1
  });
  ...
}
```

The above will create the following layout:

Index	LView	TView.data
HEADER		
DECLS		
10	[<child>, ...]	{type: Element, index: 10, parent: null}
VARS		
EXPANDO		
11..18	cumulativeBloom	templateBloom
19	new Child()	Child

20	`new Tooltip()`	`Tooltip`
21	`'Hello World!'`	`'tooltip'`
22	`'greeting'`	`'title'`
...	...	...

`EXPANDO` and Injection

`EXPANDO` will also store the injection information for the element. This is because at the time of compilation we don't know about all of the injection tokens which will need to be created. (The injection tokens are part of the Component hence hide behind a selector and are not available to the parent component.)

Injection needs to store three things:

- The injection token stored in `TView.data`
- The token factory stored in `LProtoViewData` and subsequently in `LView`
- The value for the injection token stored in `LView`. (Replacing token factory upon creation).

To save time when creating `LView` we use an array clone operation to copy data from `LProtoViewdata` to `LView`. The `LProtoViewData` is initialized by the `ProvidesFeature`.

Injection tokens are sorted into three sections:

1. `directives`: Used to denote eagerly created items representing directives and component.
2. `providers`: Used to denote items visible to component, component's view and component's content.
3. `viewProviders`: Used to denote items only visible to the component's view.

```
@Component({
  template: `<child></child>`
})
class MyApp {

  static ɵcmp = ɵɵdefineComponent({
    ...,
    decls: 1,
    template: function(rf: RenderFlags, ctx: MyApp) {
      if (rf & RenderFlags.Create) {
        ɵɵelement(0, 'child');
      }
      ...
    },
    directives: [Child]
  });
}


@Component({
  selector: 'child',
  providers: [
    ServiceA,
```

```
    {provide: ServiceB, useValue: 'someServiceBValue'},
  ],
  viewProviders: [
    {provide: ServiceC, useFactory: () => new ServiceC()}
    {provide: ServiceD, useClass: ServiceE},
  ]
  ...
})
class Child {
  construction(injector: Injector) {}
  static ecmp = eedefineComponent({
    ...
    features: [
      ProvidesFeature(
        [
          ServiceA,
          {provide: ServiceB, useValue: 'someServiceBValue'},
        ],[
          {provide: ServiceC, useFactory: () => new ServiceC())}
          {provide: ServiceD, useClass: ServiceE},
        ]
      )
    ]
  });
  ...
}
```

The above will create the following layout:

Index	LView	TView.data
HEADER		
DECLS		
10	[<child>, ...]	{type: Element, index: 10, parent: null, expandoIndex: 11, directivesIndex: 19, providersIndex: 20, viewProvidersIndex: 22, expandoEnd: 23}
VARS		
EXPANDO		
11..18	cumulativeBloom	templateBloom
	sub-section: `component` *and* `directives`	
19	factory(Child.ecmp.factory)*	Child
	sub-section: `providers`	
20	factory(ServiceA.eprov.factory)*	ServiceA

22	`'someServiceBValue'`*	`ServiceB`
	sub-section: `viewProviders`	
22	`factory(()=> new Service())`*	`ServiceC`
22	`factory(()=> directiveInject(ServiceE))`*	`ServiceD`
...	...	...

NOTICE:

- `*` denotes initial value copied from the `LProtoViewData`, as the tokens get instantiated the factories are replaced with actual value.
- That `TView.data` has `expando` and `expandoInjectorCount` properties which point to where the element injection data is stored.
- That all injectable tokens are stored in linear sequence making it easy to search for instances to match.
- That `directive` sub-section gets eagerly instantiated.

Where `factory` is a function which wraps the factory into object which can be monomorphically detected at runtime in an efficient way.

```
class Factory {
  /// Marker set to true during factory invocation to see if we get into recursive
loop.
  /// Recursive loop causes an error to be displayed.
  resolving = false;
  constructor(public factory: Function) { }
}
function factory(fn) {
  return new Factory(fn);
}
const FactoryPrototype = Factory.prototype;
function isFactory(obj: any): obj is Factory {
  // See: https://jsperf.com/instanceof-vs-getprototypeof
  return typeof obj === 'object' && Object.getPrototypeOf(obj) === FactoryPrototype;
}
```

Pseudo code:

1. Check if bloom filter has the value of the token. (If not exit)
2. Locate the token in the expando honoring `directives`, `providers` and `viewProvider` rules by limiting the search scope.
3. Read the value of `lView[index]` at that location.
    - if `isFactory(lView[index])` then mark it as resolving and invoke it. Replace `lView[index]` with the value returned from factory (caching mechanism).
    - if `!isFactory(lView[index])` then return the cached value as is.

`EXPANDO` and Injecting Special Objects.

There are several special objects such as `ElementRef`, `ViewContainerRef`, etc... These objects behave as if they are always included in the `providers` array of every component and directive. Adding them always there would prevent tree shaking so they need to be lazily included.

NOTE: An interesting thing about these objects is that they are not memoized `injector.get(ElementRef) !== injector.get(ElementRef)`. This could be considered a bug, it means that we don't have to allocate storage space for them.

We should treat these special objects like any other token. `directiveInject()` already reads a special `NG_ELEMENT_ID` property set on directives to locate their bit in the bloom filter. We can set this same property on special objects, but point to a factory function rather than an element ID number. When we check that property in `directiveInject()` and see that it's a function, we know to invoke the factory function directly instead of searching the node tree.

```
class ElementRef {
  ...
  static __NG_ELEMENT_ID__ = () => injectElementRef();
}
```

Consequence of the above is that `directiveInject(ElementRef)` returns an instance of `ElementRef` without `Injector` having to know about `ElementRef` at compile time.

`EXPANDO` and Injecting the `Injector`.

`Injector` can be injected using `inject(Injector)` method. To achieve this in tree shakable way we can declare the `Injector` in this way:

```
@Injectable({
  provideIn: '__node_injector__' as any // Special token not available to the
developer
})
class Injector {
  ...
}
```

NOTE: We can't declare `useFactory` in this case because it would make the generic DI system depend on the Ivy renderer. Instead we have unique token `'__node_injector__'` which we use for recognizing the `Injector` in tree shakable way.

`inject` Implementation

A pseudo-implementation of `inject` function.

```
function inject(token: any): any {
  let injectableDef;
  if (typeof token === 'function' && injectableDef = token.ɵprov) {
    const provideIn = injectableDef.provideIn;
   if (provideIn === '__node_injector__') {
```

```
      // if we are injecting `Injector` than create a wrapper object around the
inject but which
      // is bound to the current node.
      return createInjector();
    }
  }
  return lookupTokenInExpando(token);
}
```

`LContainer`

TODO

Combining `LContainer` with `LView`

TODO