## NgModule.providers vs Component.providers vs Component.viewProviders

We can configure injectors in Angular by:

1. providers on *NgModule*.
2. providers on *Components* and *Directives*.
3. viewProviders on *Components*.

So the question is where do you configure your provider?

Understanding *where* to configure your provider is a key piece of understanding *how* to architect your application, so we are going to explain this via a real practical example.

**Learning Objectives**

• Know the difference between configuring a provider on an NgModule, a component or directives providers property and a components viewProviders property.

**Setup**

We create a class called SimpleService which has one property called value which holds a string.

class SimpleService {

value: string;

}

We also have a component called ParentComponent which has a child component called ChildComponent.

@Component({

selector: 'child',

template: `

<

div class="child"

>

<p>Child</p>

{{ service.value }}

①

<

/div

>

`

})

class ChildComponent {

constructor(private service: SimpleService) { }

②

}

1. We use *string interpolation* to bind to the value property of SimpleService.
2. We *inject* an instance of SimpleService into the constructor.

@Component({

selector: 'parent',

template: `

<

div class="parent"

>

<p>Parent</p>

<form novalidate>

<div class="form-group">

<input type="text"

class="form-control"

name="value"

[(ngModel)]="service.value">

①

</div>

</form>

<child></child>

②

<

/div

>

`

})

class ParentComponent {

constructor(private service: SimpleService) { }

③

}

1. We use *two way data binding* to bind to the value property of SimpleService.
2. We render the ChildComponent inside this ParentComponent.
3. We *inject* an instance of SimpleService into the constructor.

The ParentComponent has just one input box which reads and writes to the SimpleService value property using two way ngModel binding, the ChildComponent just renders the value to the screen with {{ }}.

We render two *side by side* <parent> tags in our root AppComponent module, like so:

@Component({

selector: 'app',

template: `

<div class="row">

<div class="col-xs-6">

<parent></parent>

</div>

<div class="col-xs-6">

<parent></parent>

</div>

<

/div

>

`

})

class AppComponent {

}

We set up our

NgModule

and bootstrap it, like so:

@NgModule({

imports: [ BrowserModule, FormsModule ],

declarations: [ AppComponent, ParentComponent, ChildComponent ],

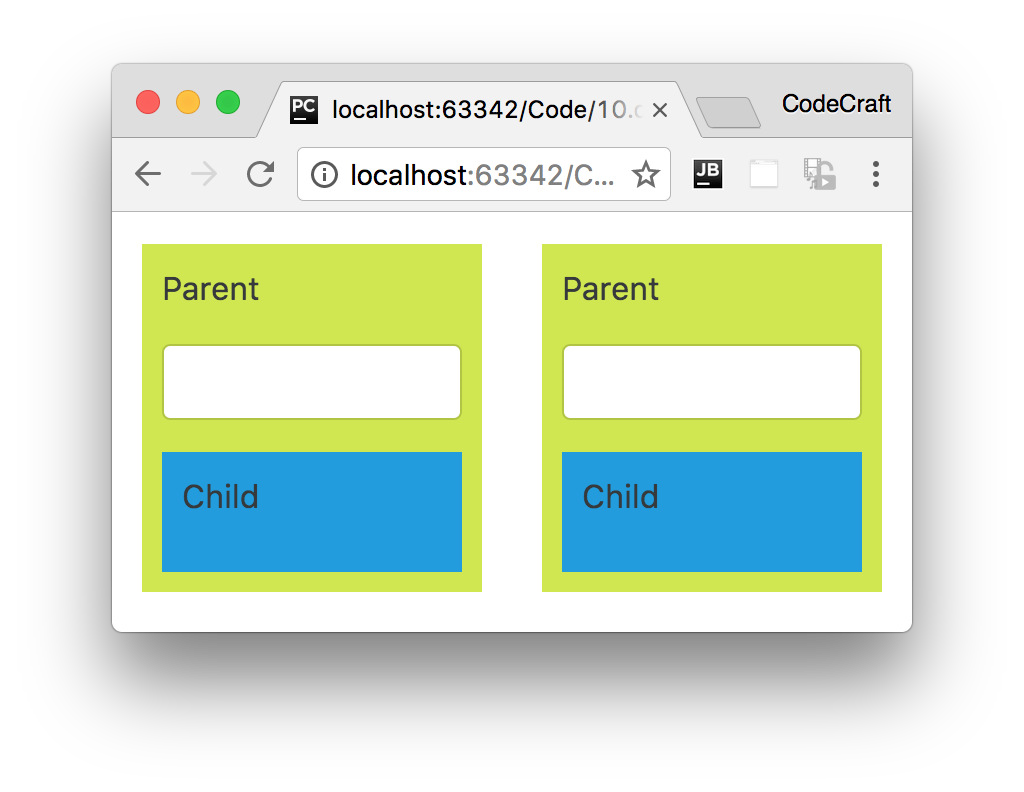
bootstrap: [ AppComponent ]

})

class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);

In the end when we run our application we should end up with something that looks like this:





We have also added some css styles on our component which has been removed

from the above code, the full code can be found in the listing at the end of this

lecture.

**NgModule.providers**

We’ll first configure our SimpleService on the root NgModule, like so:

@NgModule({

imports: [ BrowserModule, FormsModule ],

declarations: [ AppComponent, ParentComponent, ChildComponent ],

bootstrap: [ AppComponent ],

providers: [ SimpleService ]

①

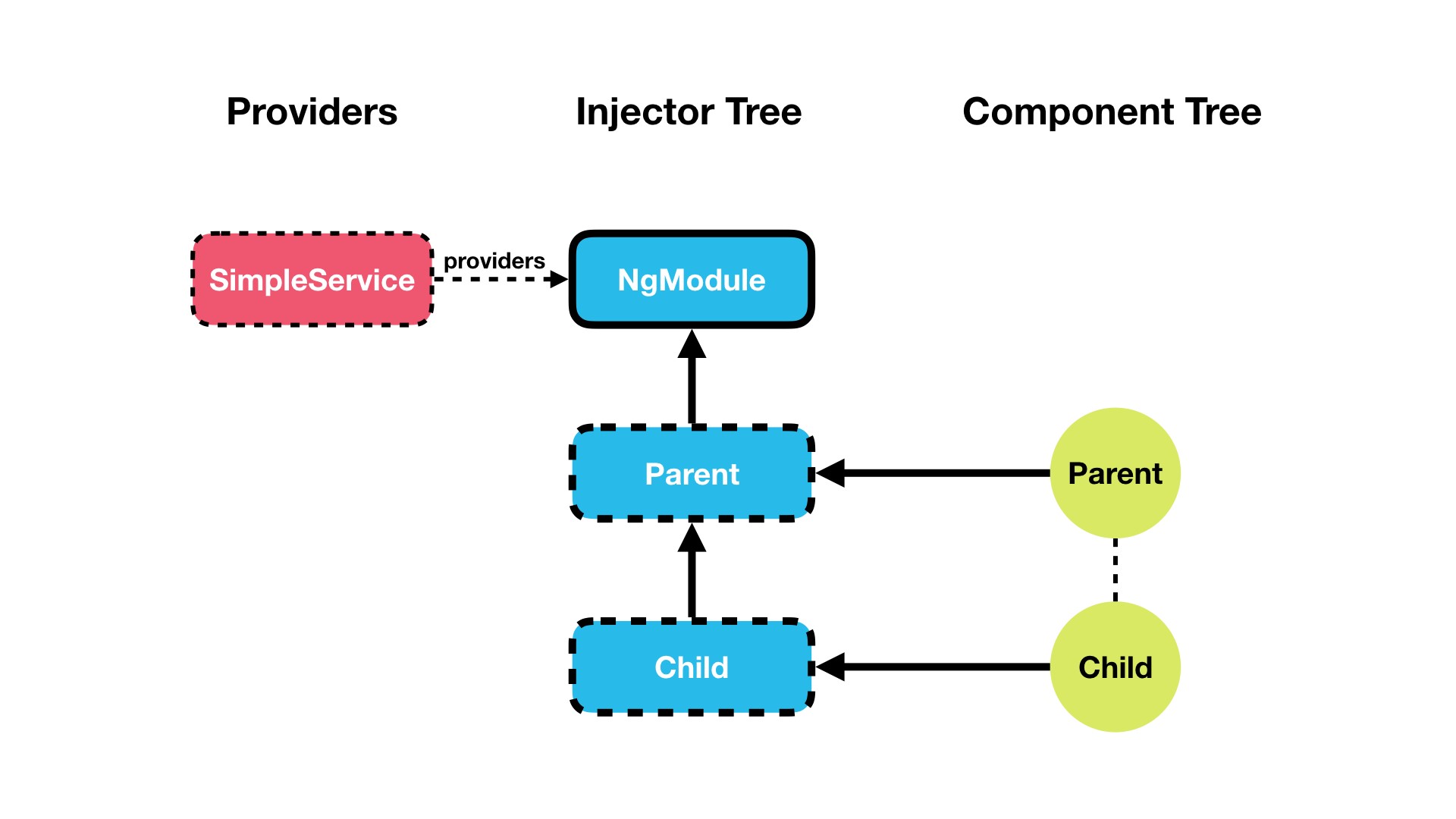
})

class AppModule { }

① We’ve configured our NgModule with a class provider of SimpleService.

In this configuration the service has been injected onto our applications root NgModule and therefore is in our root injector.

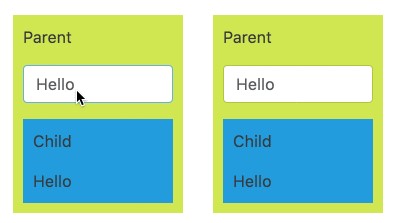
So every request to resolve and inject the token SimpleService is going to be forwarded to our single root injector.



Therefore since we only have one injector which is resolving the dependency, every-time we request an instance of SimpleService to be injected into one of our components it’s *always* going to inject the *same* instance.

Remember if we request the *same* token from the *same* injector we get the *same* instance.

Since we’ve bound the input field directly to the simple service value field **and** it’s the same instance of simple service used everywhere, then when we type into one input control it automatically updates the other input control and also the child components.





If we want to share

*one*

instance of a service across the

*entirety*

of our application

we configure it on our

NgModule

.

**Component.providers**

Let’s now see what happens when we configure our SimpleService *additionally* on the ParentComponent via the providers property.

@Component({

selector: 'parent',

template: `...`,

providers: [ SimpleService ]

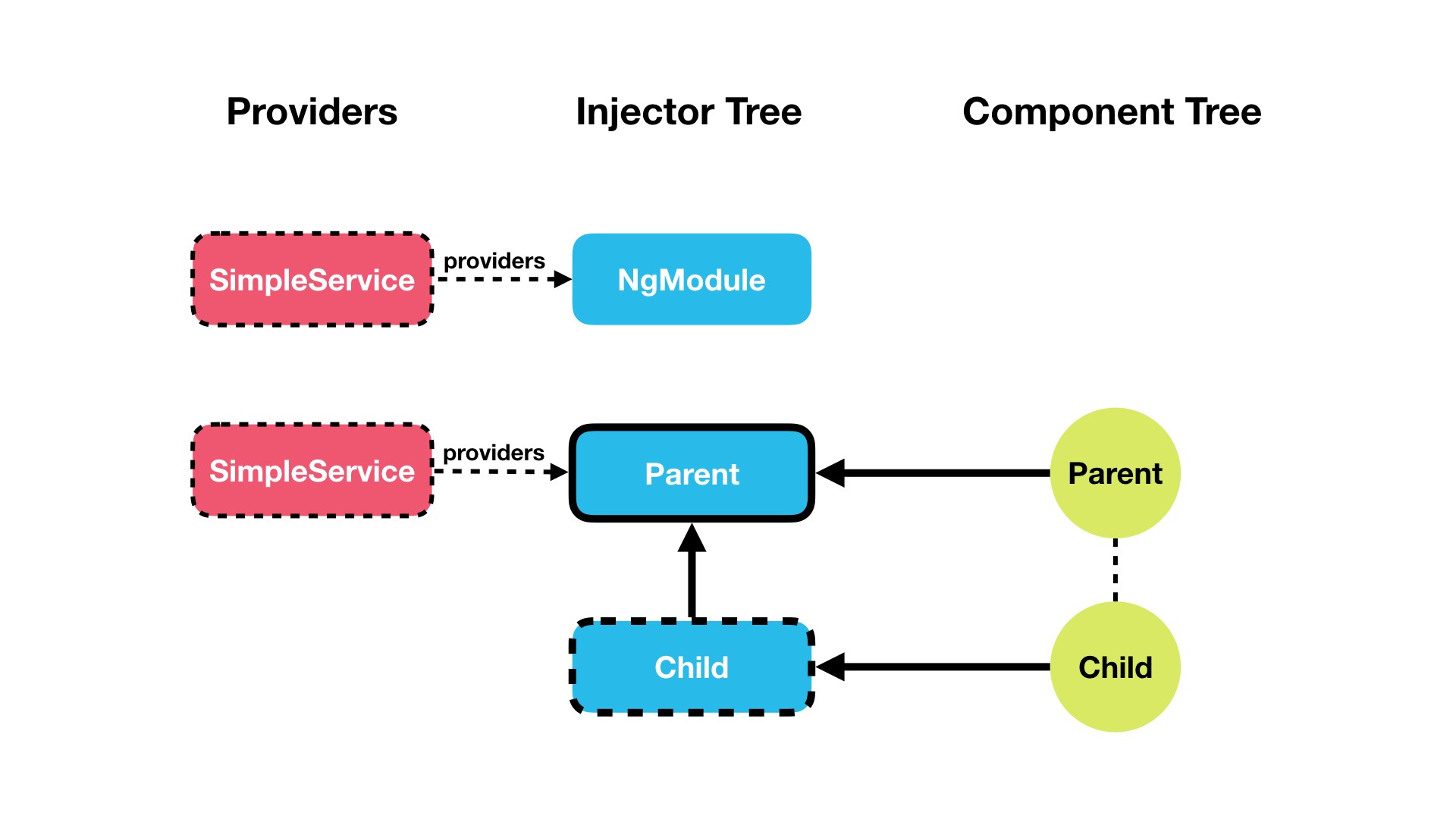
})

class ParentComponent {

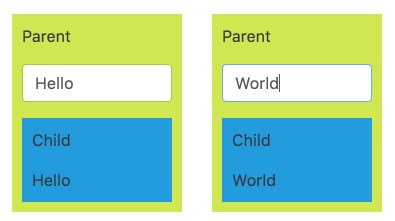
constructor(private service: SimpleService) { }

}

Now *each* ParentComponent has it’s *own* child injector with SimpleService configured, like so:



We can see from the running the code above that if we type into one parent component only *that* parent component and it’s child component automatically updates, like so:



Each instance of ParentComponent now has it’s *own* instance of SimpleService, so state is not shared globally but only between a ParentComponent and it’s child components.

That’s because each instance of ParentComponent has it’s own child injector with SimpleService configured as a provider.

Remember when we request the *same* token from *different* injectors we get the

*different* instances.

When we configured the SimpleService on the parent component it created a child injector, and when we tried to inject SimpleService into the parent component constructor it resolved and created an instance of SimpleService from it’s own injector.

If we want to have *one* instance of a service *per* component, and shared with all

the components children, we configure it on the providers property on our component decorator.

**Component.viewProviders**

If we now configure the SimpleService provider on the viewProviders property on the ParentComponent nothing changes, we still get the functionality we had before.

But lets use content projection and the ng-content component to change the child component from being a *view child* of parent to to being a *content child* of parent. i.e. lets pass in <child></child> to the parent component like so:

<

parent><child></child></parent

>

So we change the AppComponent template to pass in child to the parent component, like so:

<div class="row">

<div class="col-xs-6">

<parent><child></child></parent>

</div>

<div class="col-xs-6">

<parent><child></child></parent>

</div>

<

/div

>

Change the ParentComponent template to *project* the passed in content to the same place the child component used to be, like so:

<

div class="parent"

>

<p>Parent</p>

<form novalidate>

<div class="form-group">

<input type="text"

class="form-control"

name="value"

[(ngModel)]="service.value">

</div>

</form>

<ng-content></ng-content>

①

<

/div

>

① We use content projection to *insert* the ChildComponent where it used to be hard coded.

Now even though child is still rendered under parent, it’s considered a *content child* and not a *view child*.

Lets now change the configuration of ParentComponent to use viewProviders instead.

@Component({

selector: 'parent',

template: `...`,

viewProviders: [SimpleService ]

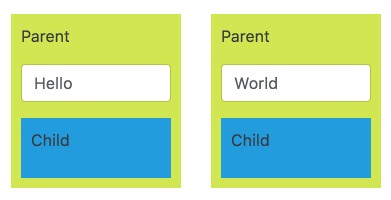
})

class ParentComponent {

constructor(private service: SimpleService) { }

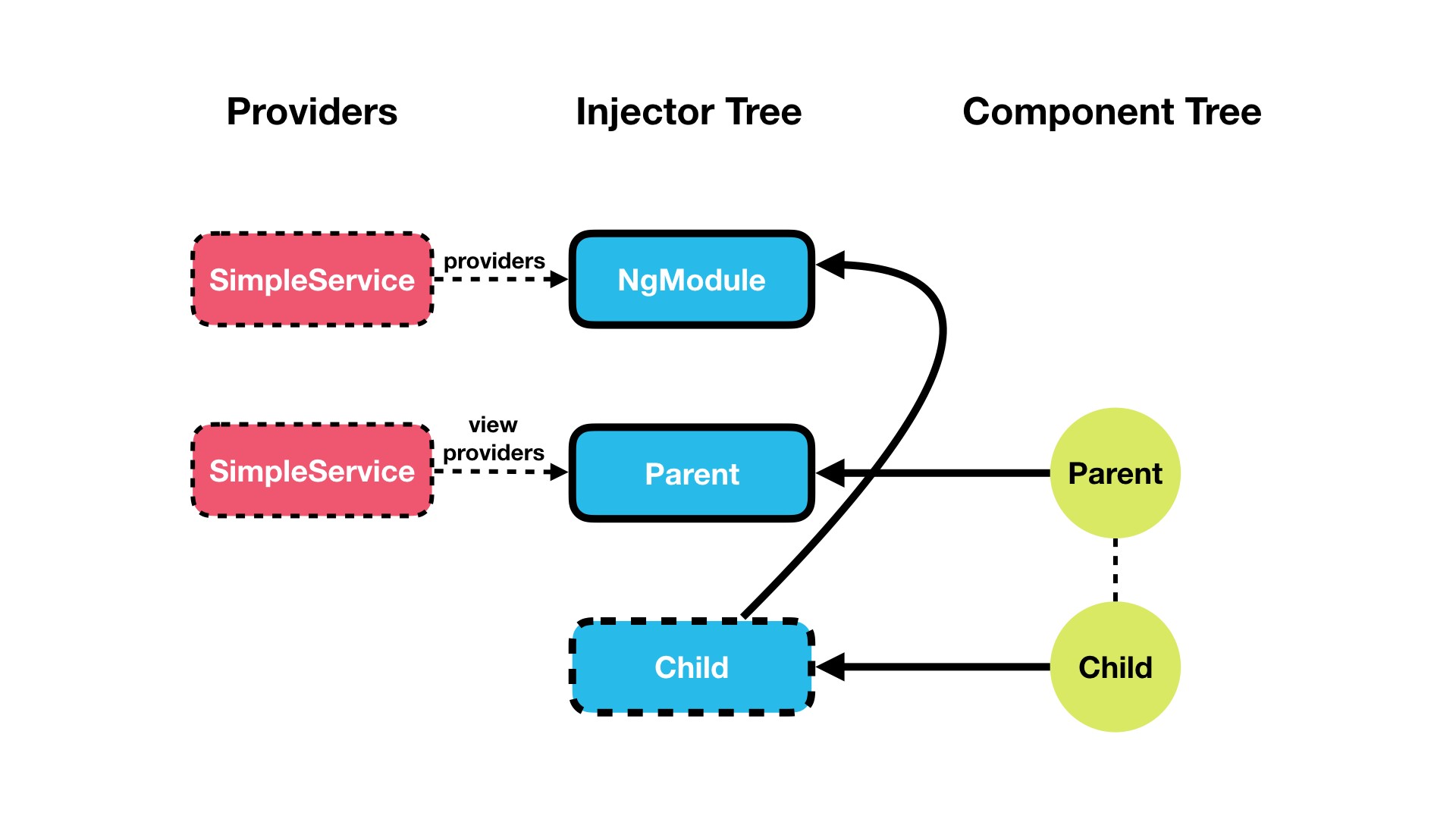
}

Now when we type into the ParentComponent the child component doesn’t update automatically.



That’s because when using viewProviders the component creates an injector which is **only** used by the *current component* and any *view children*.

If you are a *content child*, as our child component now is, then it uses the injector in NgModule to resolve the dependency.





If we want to have

*one*

instance of a service

*per*

component, and shared with

*only*

the components view children and

*not*

the components content children, we

configure it on the

viewProviders

property on our component decorator.

**Summary**

We can configure the DI framework in Angular in three main ways.

We can configure a provider on the NgModule, on a component or directives providers property and on a components viewProviders property.

Deciding where to configure your provider and understanding the different is key in understanding how to architect an Angular application.

If we want an instance of a dependency to be shared globally and share *state* across the application we configure it on the NgModule.

If we want a separate instance of a dependency to be shared across each instance of a component and it’s children we configure it on the components providers property.

If we want a separate instance of a dependency to be shared across each instance of a component and only it’s view children we configure it on the components viewProviders property.

**Listing**

<http://plnkr.co/edit/PTyIJYIrPWqjMSNi9krS?p=preview>

*script.ts*

import { NgModule, Component, Injectable } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { BrowserModule } from '@angular/platform-browser';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

class SimpleService {

value: string;

}

@Component({

selector: 'child',

template: `

<div class="child">

<p>Child</p>

{{ service.value }}

<

/div

>

`,

styles: [`

.child {

background-color: #239CDE;

padding: 10px;

}

`],

// providers: [SimpleService]

})

class ChildComponent {

constructor(private service: SimpleService) { }

}

@Component({

selector: 'parent',

template: `

<div class="parent">

<p>Parent</p>

<form novalidate>

<div class="form-group">

<input type="text"

class="form-control"

name="value"

[(ngModel)]="service.value">

</div>

</form>

<ng-content></ng-content>

<

/div

>

`,

styles: [`

.parent {

background-color: #D1E751;

padding: 10px;

}

`],

viewProviders: [SimpleService ]

// providers: [SimpleService]

})

class ParentComponent {

constructor(private service: SimpleService) { }

}

@Component({

selector: 'app',

template: `

<div class="row">

<div class="col-xs-6">

<parent><child></child></parent>

</div>

<div class="col-xs-6">

<parent><child></child></parent>

</div>

<

/div

>

`

})

class AppComponent {

}

@NgModule({

imports: [ BrowserModule, FormsModule ],

declarations: [ AppComponent, ParentComponent, ChildComponent ],

bootstrap: [ AppComponent ],

providers: [SimpleService]

})

class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);