

# Lara Newsom



Cisco  
Customer Experience



The Angular  
Plus Show



Angular GDE



Nx Champion

# Lara Newsom



lara-newsom



laranewsom  
.bsky.social



@laranerdsom



# Reactive Patterns for Angular

Lara Newsom

Cisco - Technical Leader

A cartoon illustration of a man with dark hair and glasses, wearing a blue pinstripe suit and a red tie with black polka dots. He has his hand to his chin in a thoughtful pose. A large white speech bubble originates from his head, containing the text.

**But aren't Signals here  
to make my app reactive?**

*Signals  
are here...*

But Signals won't  
magically make  
your code reactive



*Why are we  
even talking  
about  
reactivity?*

Javascript has  
asynchronous events

We have to manage state  
as data keeps changing

Users expect the view to  
refresh quickly and accurately



# The other reason we should care about reactivity

Angular 🎉  
@angular

Angular v16!

- + Angular Signals in developer preview ⚡
- + Developer preview of opt-in non-destructive hydration 💧
- + Improved Standalone APIs 🧑
- + Tooling enhancements 📦

You can get all this and more with `ng update`! 🚀

[goo.gl/angular-v16](https://goo.gl/angular-v16)



2:59 PM · May 3, 2023 · 200.4K Views



New signal based  
reactive primitives



Rethinking change  
detection



Moving towards fine  
grained reactivity

**Understanding reactivity and working to improve your code NOW will make it easier to leverage new features as they are released.**



What is reactivity?

What tools do we have?

What changes can you make today?

Handling real time view  
updates to asynchronous  
data changes?

Refreshing user  
content when data  
changes?

## What is reactivity?

Changes to the  
view over time?

A buzz word to  
make me feel bad  
about my code?

# *Three Ingredients for Reactivity*

one

Producers

two

Consumers

three

Side Effects



**Producers:**  
Events or elements  
that are capable of  
being observed

# *Angular producer examples*

- User events
- Event emitters
- Router events
- Observables
- \* Signals
  - \* Still in developer preview



**Consumers:**  
Observe producers  
and use the results

# *Angular consumer examples*

- Components
- View templates
- Component Store
- NgRx reducers



# Side Effects:

**Consumers that  
intercept events to  
trigger a task and then  
emit another event**

*Similar  
programming  
concepts*

Middleware

Monads

Higher Order Functions

# *Angular side effects examples*

- Data services
- NgRx effects
- Http Interceptors
- Pipes
- RxJs Operators
- \* Signals Computed
- \* Signals Effects
  - \* Still in developer preview

*Let's put the  
ingredients together*

**SOMETHING HAPPENED!**



**Producers:** Announce what has happened



# Consumers & Side Effects:

## Listen for specific events

ORDER IN!



Krabby Patty  
OUT!

**Side Effects:** Work is triggered by an event.  
Completion triggers another event.

Your Krabby  
Patties sir...



## Consumers:

Digests (reads) results without side effects (we hope)



**oops**

We get into trouble  
when we mix up  
responsibilities



Data flow through that 3000 line component no one wants to touch

Following good reactive patterns makes it easy to understand where changes should happen

To help with consistent responsibilities we can talk about  
**imperative** vs **declarative**

# Imperative Programming

- Typical of Object Oriented Languages
- Process based approach
- State is managed step by step

# Imperative

```
getCats() {  
    return this.catService.getCats();  
}
```

This method dictates  
what to return.

It is a function that returns an  
observable

# Imperative

```
getCats() {  
    → this.updateParams();  
    → this.selectedFilters = [];  
    return this.catService.getCats();  
}
```

There is nothing to prevent side effects from being added to this method.

# Common Imperative Issues

one

Excessive class properties

two

Tightly coupled code

three

Random side effects  
from void methods

*In my experience with  
Angular applications...*

Imperative coding  
+ time =



# Declarative Programming

- Typical of functional languages
- Results oriented approach
- Stateless & execution order agnostic

# Declarative Programming

*This is  
what async  
apps need*



Typical of functional languages



Results oriented approach



Stateless & execution order agnostic

# *Declarative libraries & languages*

- RxJs
- LINQ
- Elm
- F#
- Prolog
- Lisp
- Haskell
- Miranda
- Erlang

# Declarative

```
cats$ = this.catService.get_cats();
```

The cats\$ property IS the  
get\_cats observable.

There is no room for side effects.

# Declarative

```
vm$ = this.catService.getCats()
    .pipe(
        map((cats) => ({
            breeds: cats.map((c) => c.breed),
            cats
        }))
    );

```

We can manipulate the results, but the only way to change the value of the stream is by pushing a new value into it.

# *Declarative*

```
cats = signal([]);
```

Similarly with Signals,  
the cats property IS the signal.

# Declarative

```
viewModel = computed(() => {
    return {
        cats: cats(),
        breeds: cats().map((c) => c.breed),
    }
});
```

Similarly, signals can be composed together, but their values must be changed through setters

## *Problems with declarative programming*

Higher learning curve

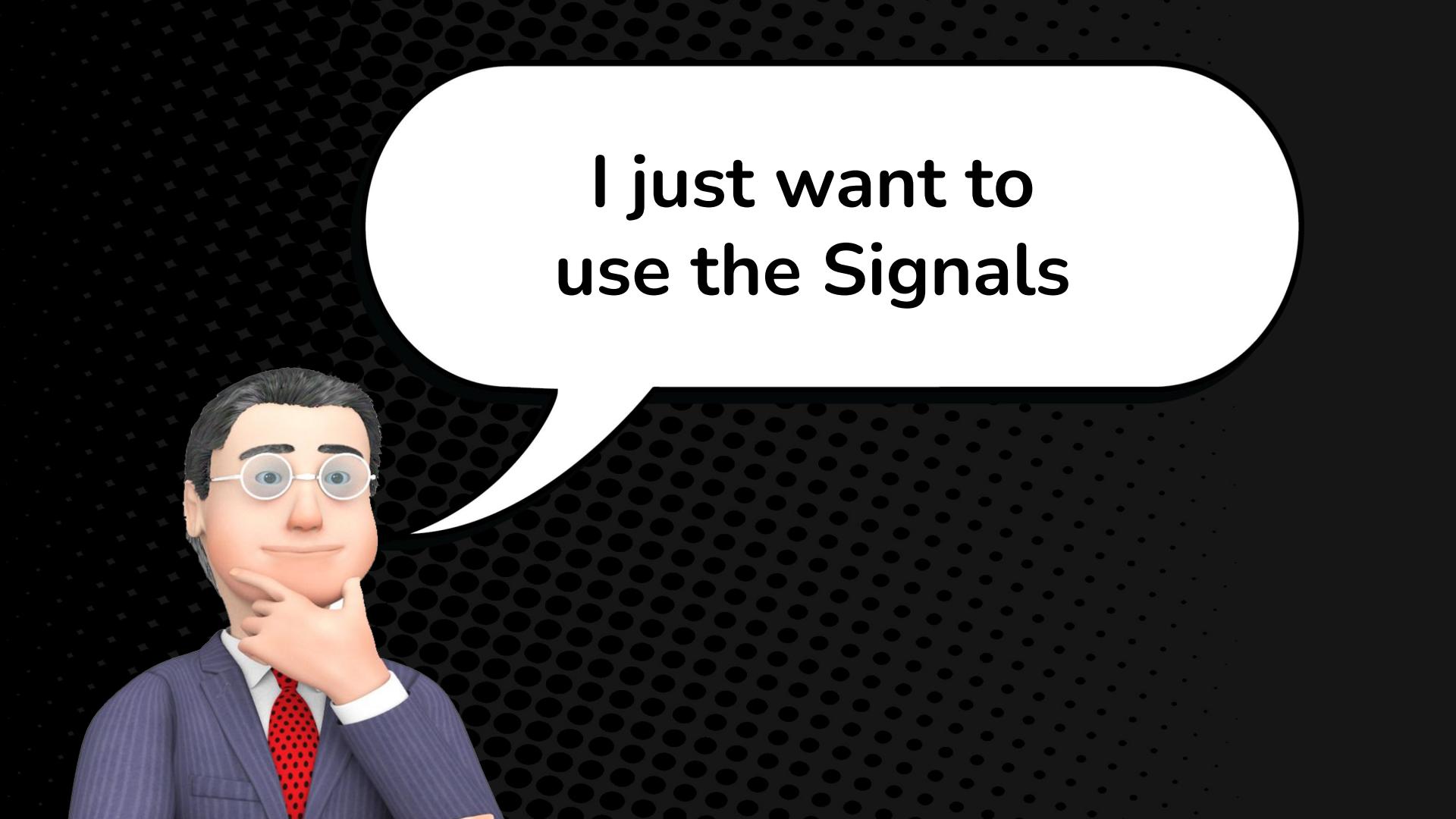
Not always as easy to read

Not how most of us are  
taught to think about code

# Writing declarative code is just

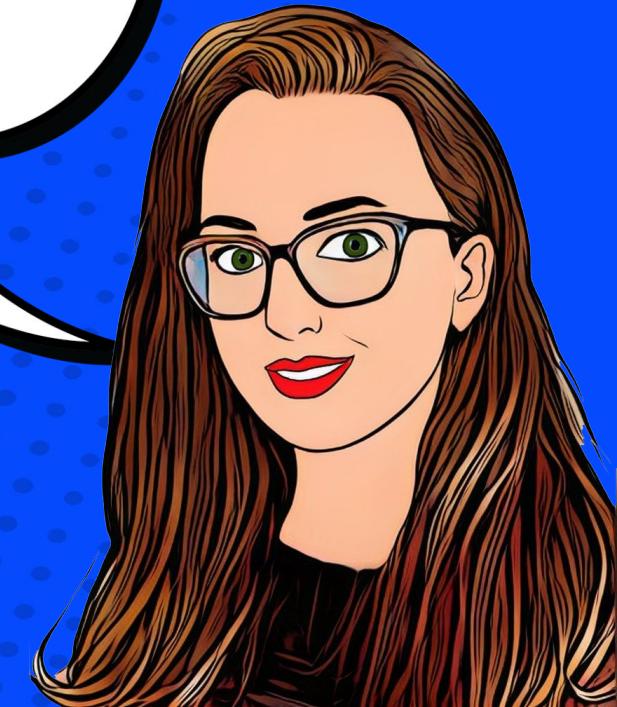


# a different way of seeing things



I just want to  
use the Signals

**Remember Signals are  
still in developer preview**



# POW

But we can set our  
code up today to be ready  
for signals later



**Using setters on Inputs  
instead of ngOnChanges**

```
export class DetailViewComponent implements OnChanges {
  @Input() productId = '';
  @Input() isDetail = false;
  @Input() isFullscreen = false;

  ngOnChanges(changes: SimpleChanges) {
    const firstChange = changes['productId']?.firstChange;
    const hasChanged =
      changes['productId']?.currentValue !== changes['productId']?.previousValue;
    if(firstChange || hasChanged) {
      this.productService.setSelectedProduct(changes['productId'].currentValue);
    }
  }
}
```

This code does work  
It technically reacts to changes

## *Issues with using OnChanges*

It runs when  
any input changes

Requires logic for  
which input changed

Only knows that something  
changed not what changed

```
export class DetailViewComponent {  
  @Input() set productId(val: string) {  
    this.productService.setSelectedProduct(val);  
  }  
}
```

This code is more precise.

```
export class DetailViewComponent {  
  @Input() set productId(val: string) {  
    this.productService.setSelectedProduct(val);  
  }  
}
```

This code is more precise.

## *Using Setters*

A setter only runs when its input changes

No filtering logic required, the setter is tied to one input

No need to implement OnChanges



This pattern can  
be easily adapted to  
Signals!

BAM

```
export class DetailViewComponent {  
  @Input() set productId(val: string) {  
    this.productService.selectedProduct.set(val);  
  }  
}
```

Check me out!  
I'm a signal!

When your app is ready to use signals,  
it is an easy change.

*Pattern*

**two**

**Cut out the middle-man  
when dispatching user  
events and actions**

*Let's have a little chat about  
component Output properties*



# Angular @Output

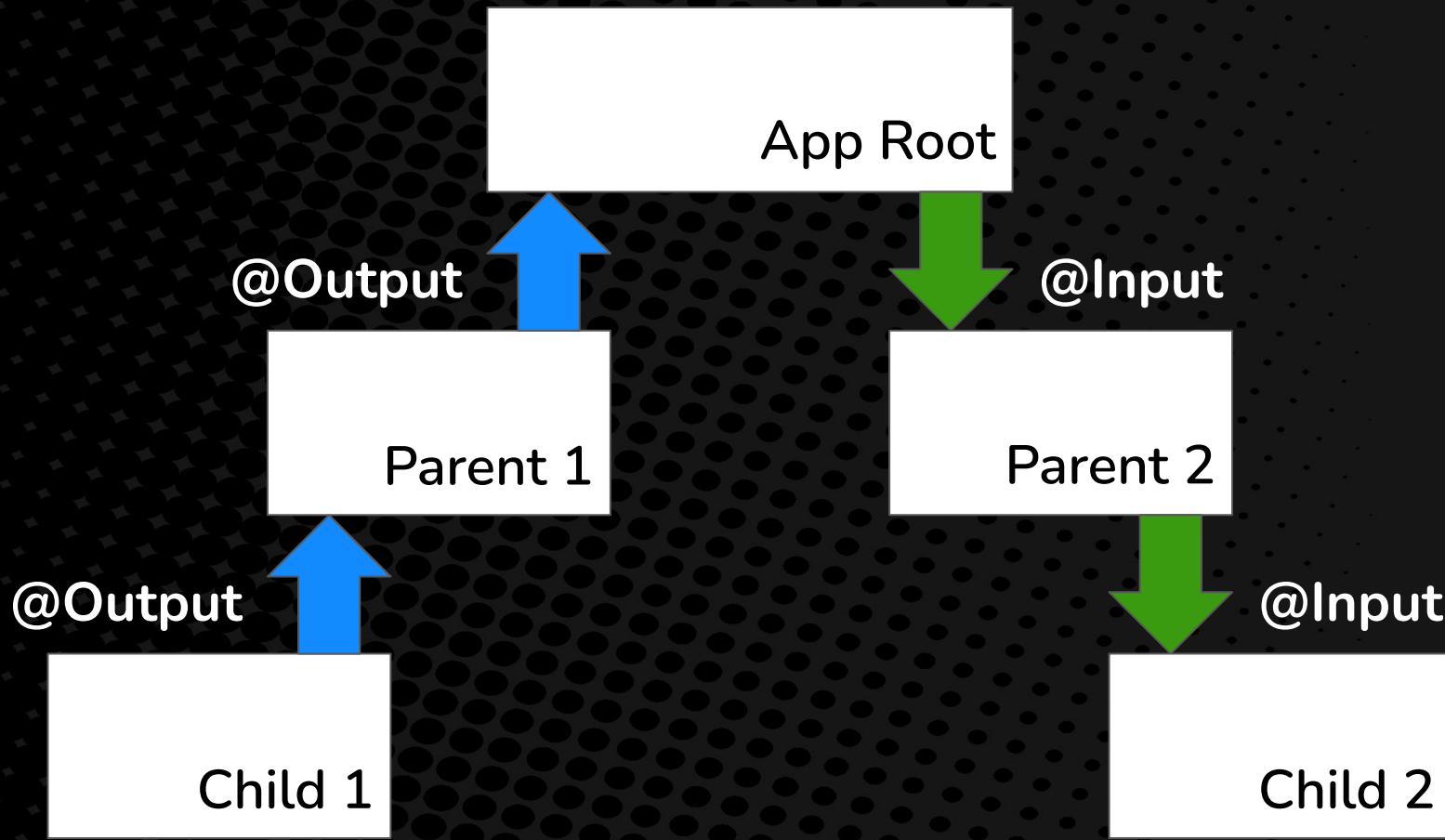
Great tool for passing data from child to parent

Not great tool for passing data to other parts of the app

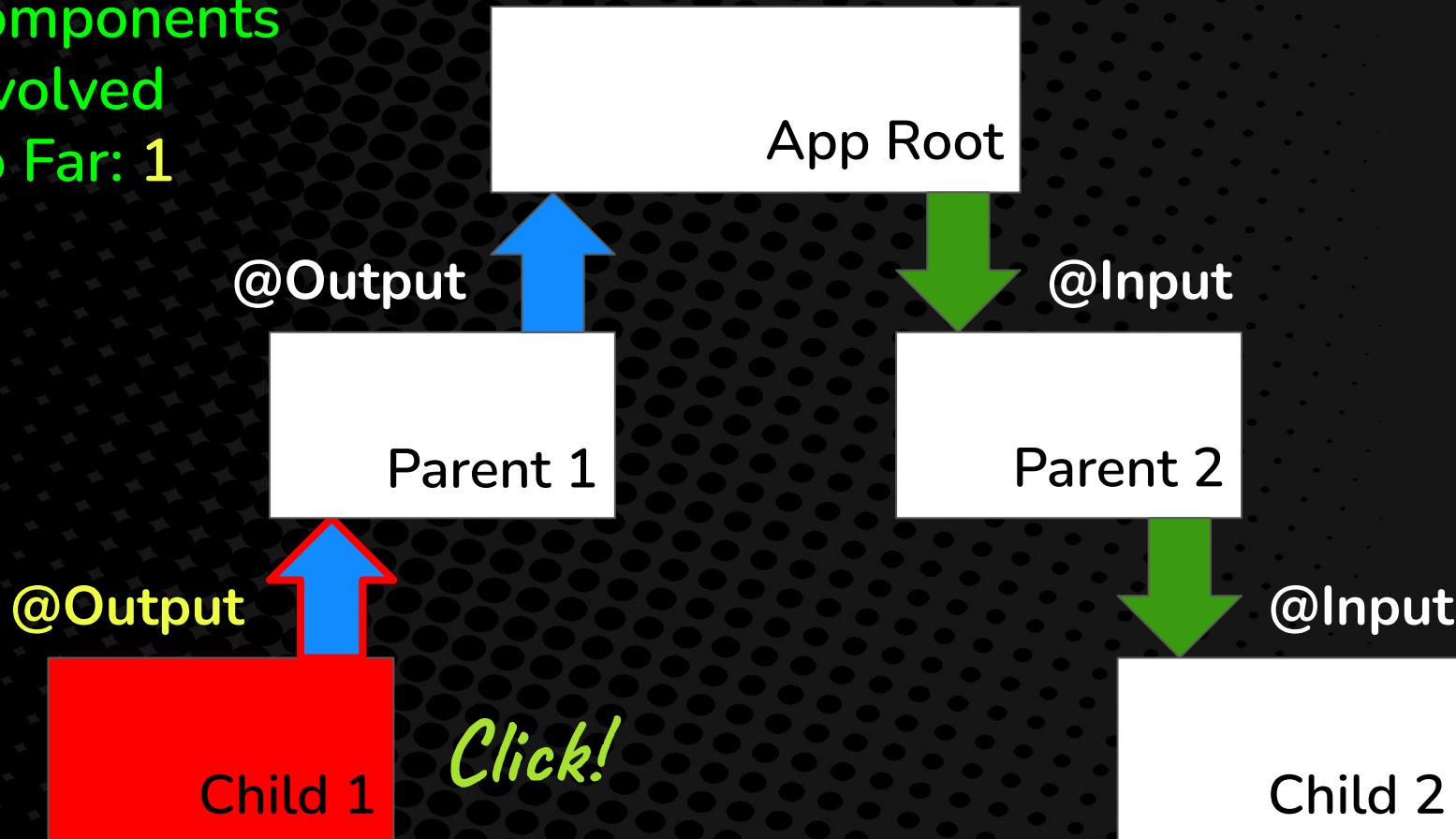
Tightly couples the emitting component to the tree

```
export class SideMenuComponent {  
  @Output() selectedProduct = new EventEmitter<string>;  
  
  selectProduct(id: string){  
    this.selectedProduct.emit(id); ← Emits event here  
  }  
}
```

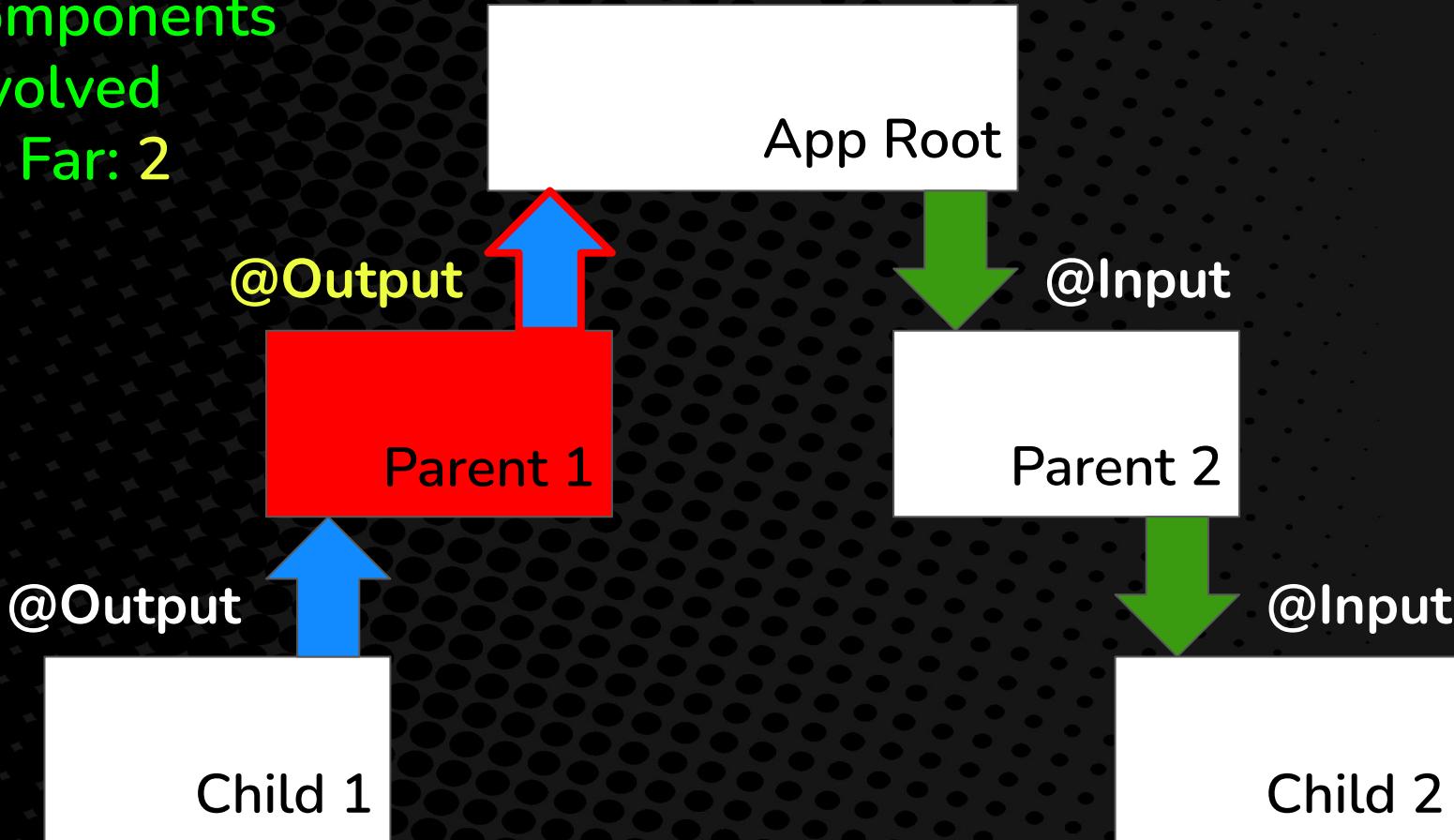
Let's look at the data flow  
selecting a product using Outputs



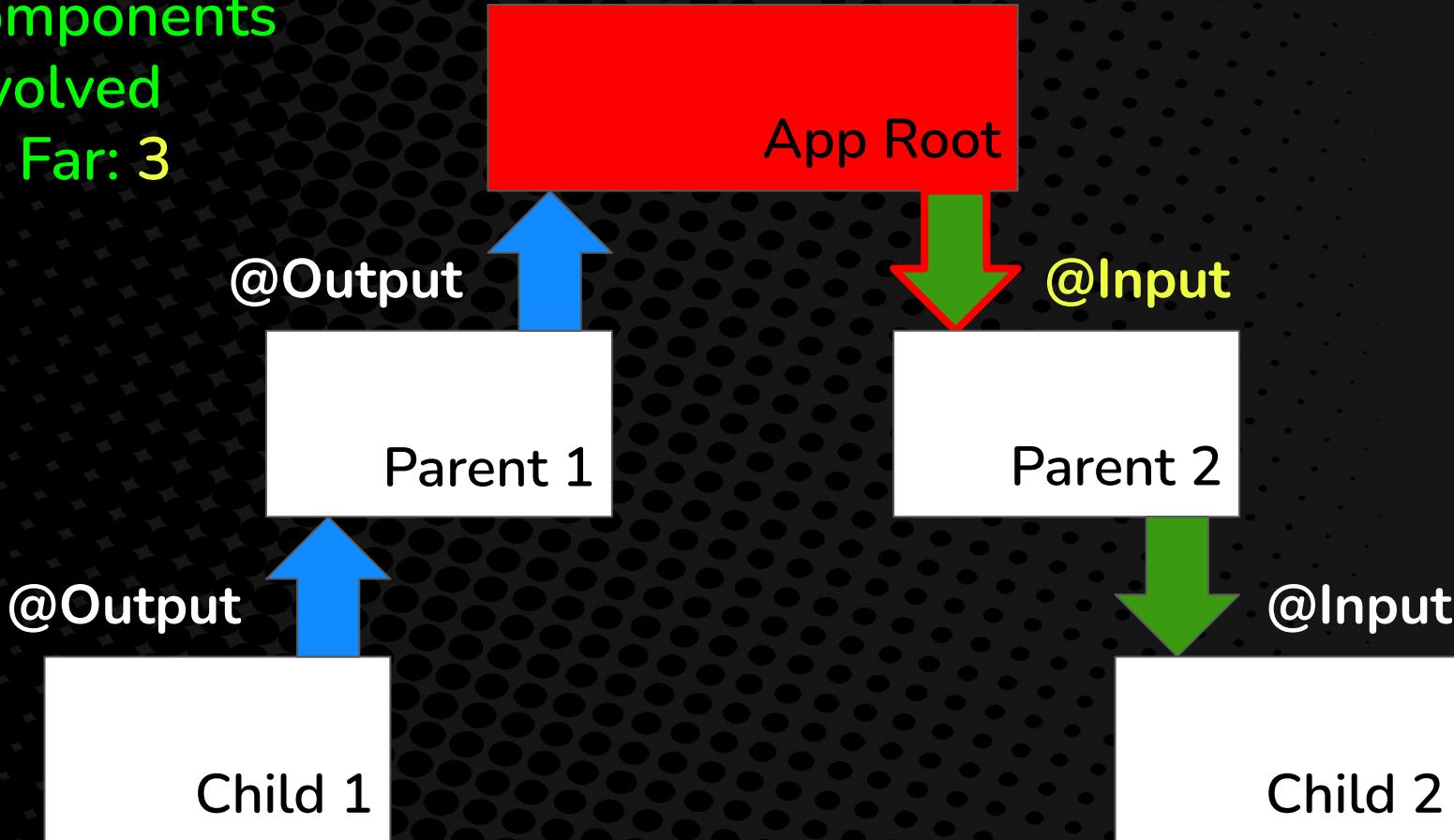
Components  
Involved  
So Far: 1



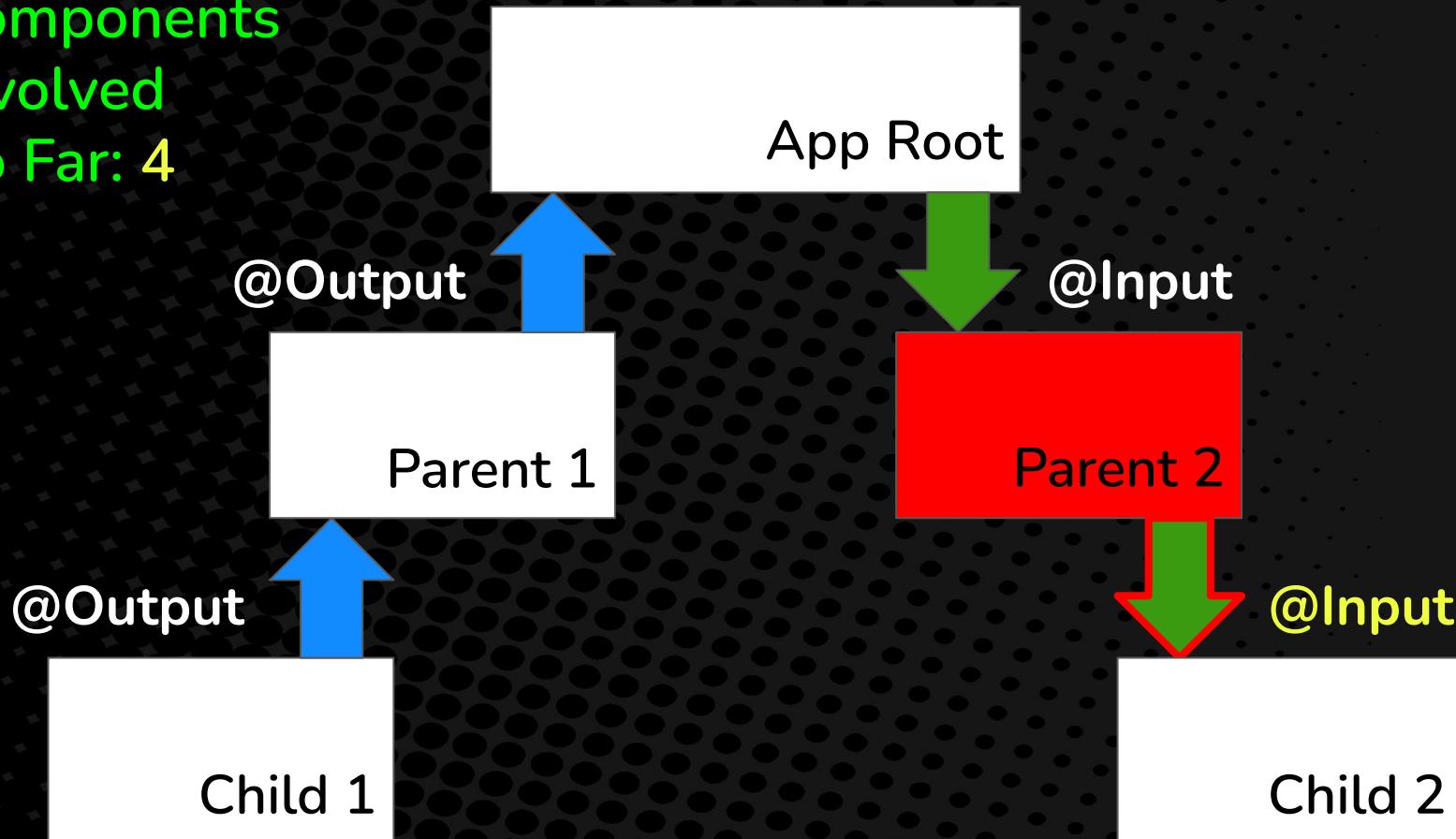
Components  
Involved  
So Far: 2



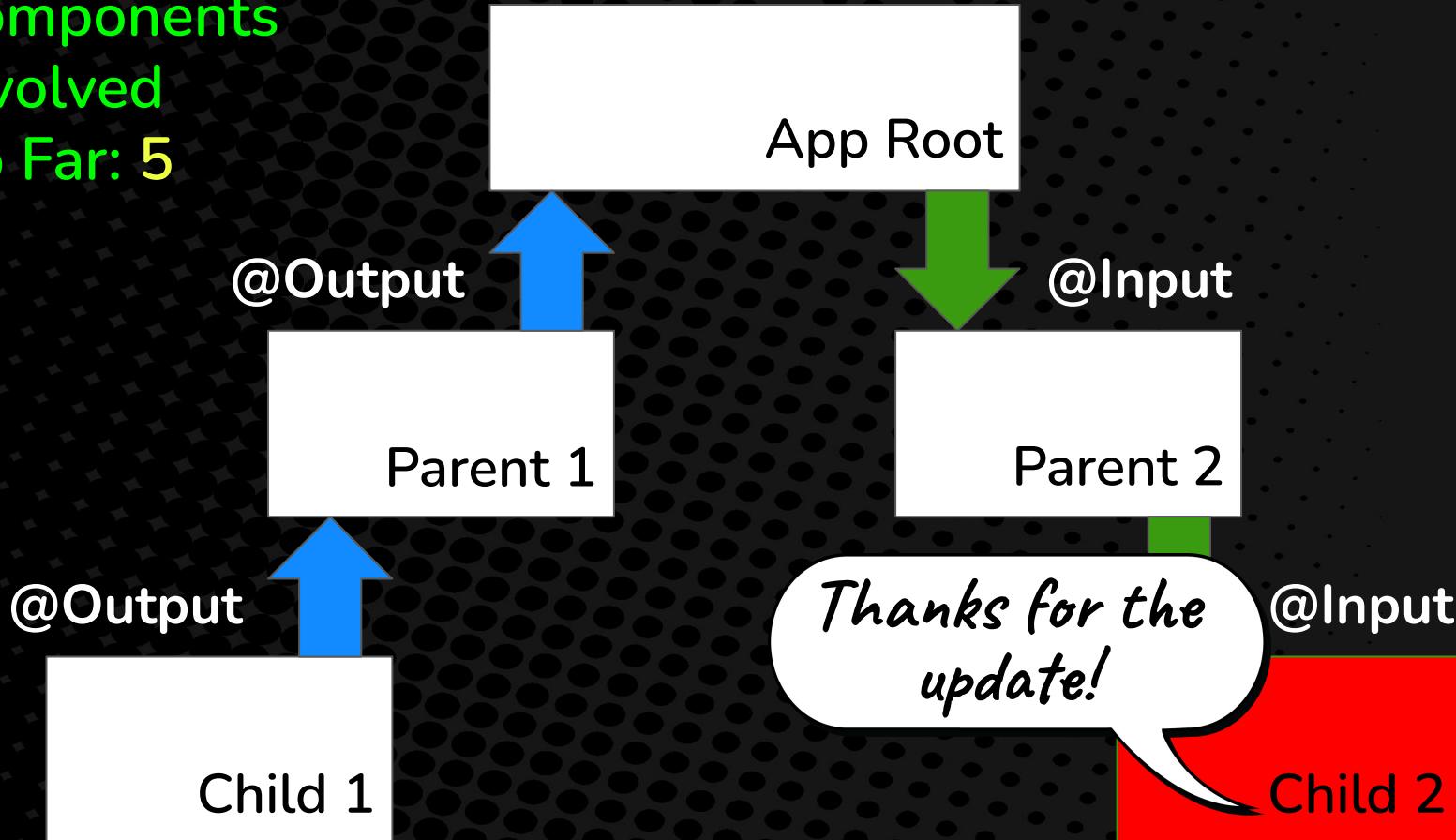
Components  
Involved  
So Far: 3



Components  
Involved  
So Far: 4

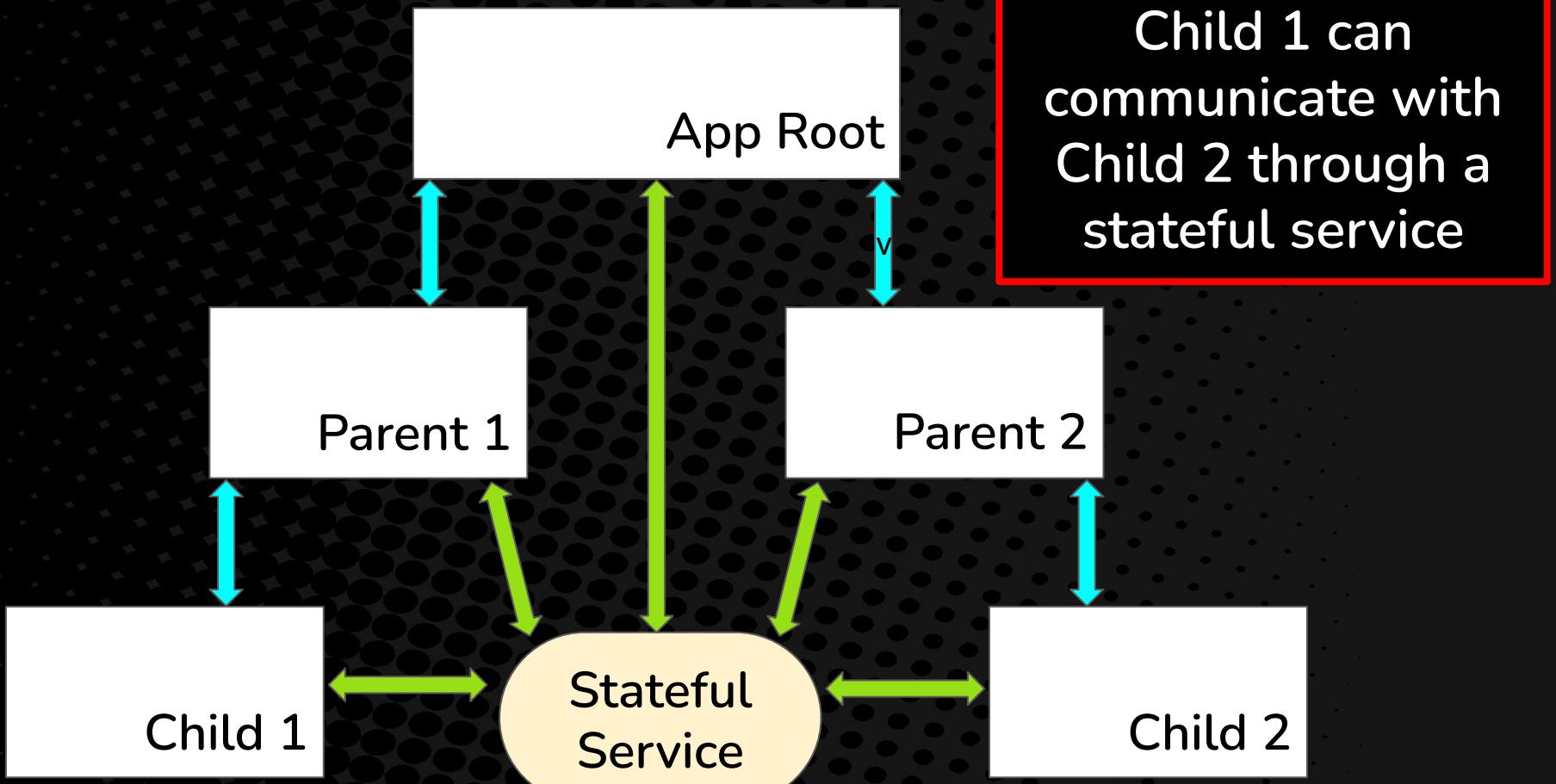


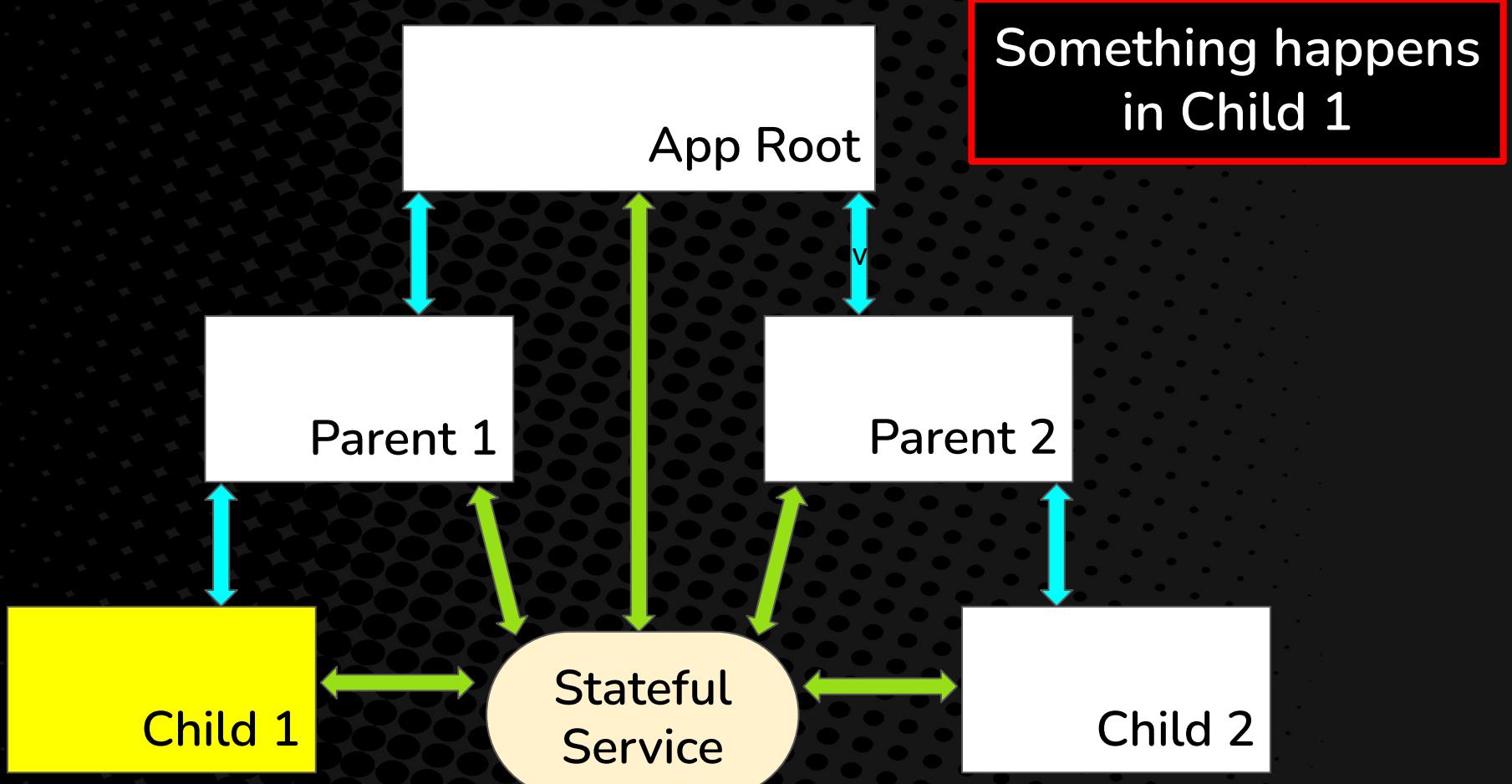
Components  
Involved  
So Far: 5

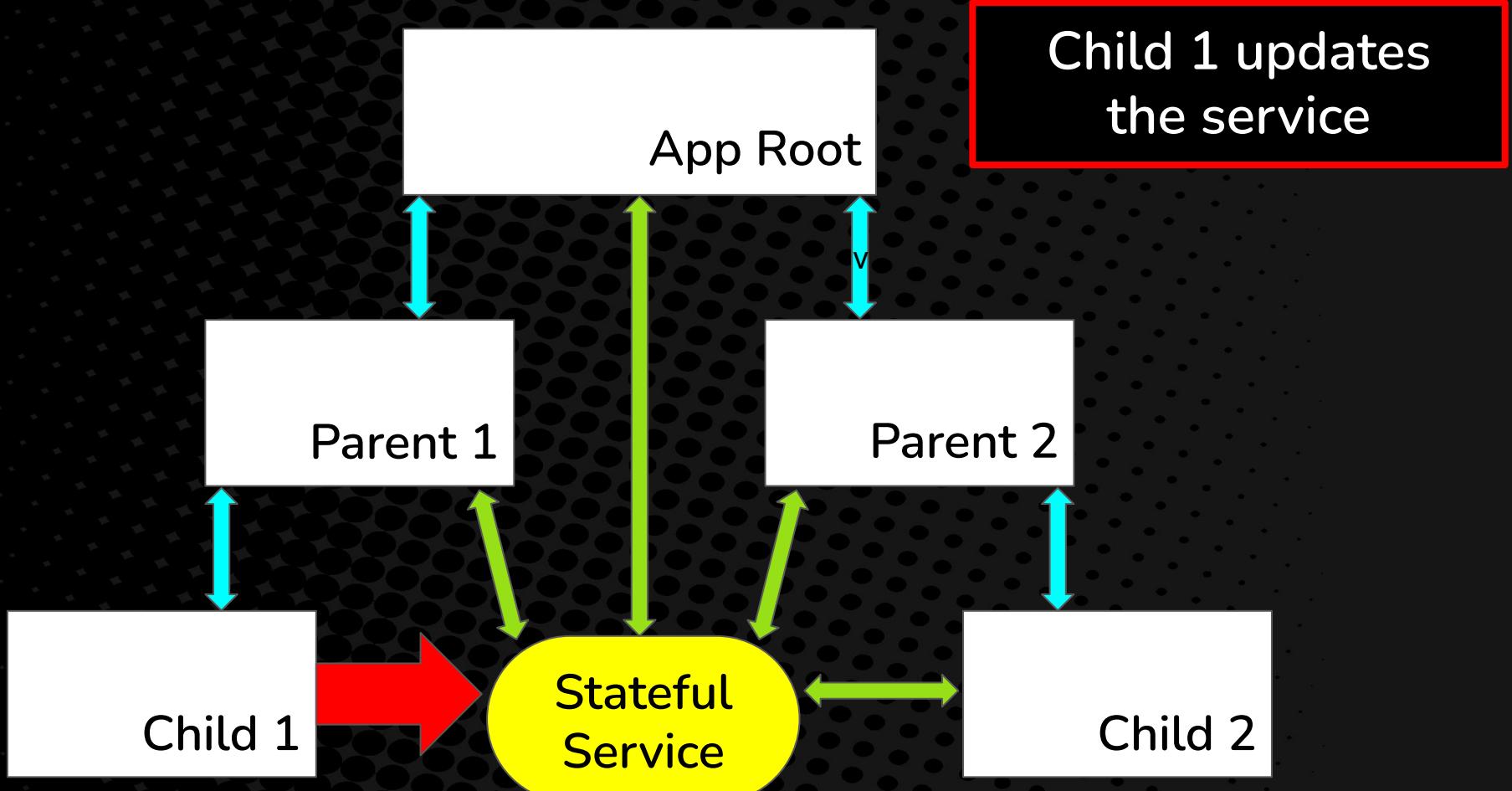


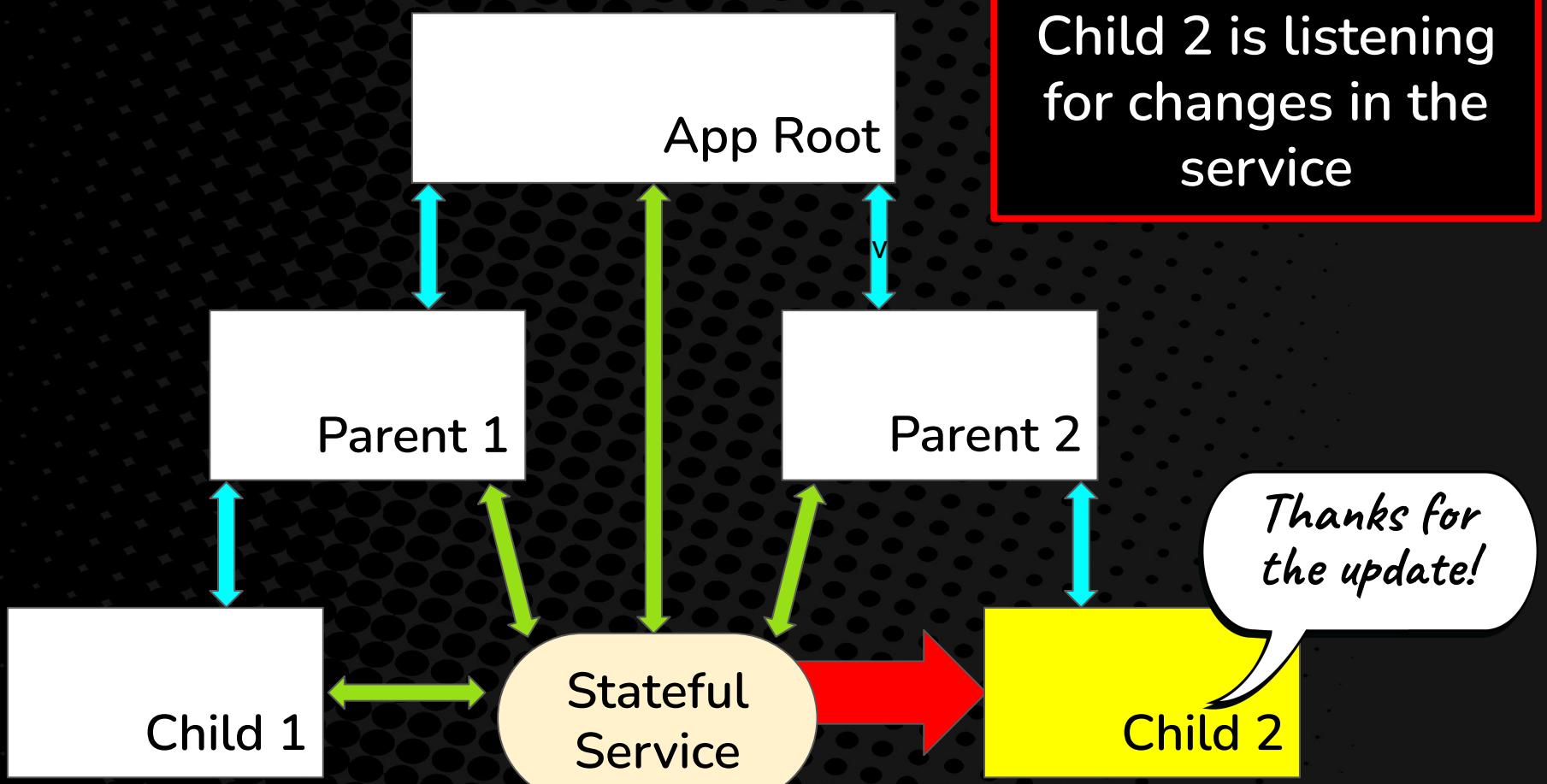
That flow required **FIVE**  
components to get the data from  
the producer to the consumer!

We need a stateful service  
that can be accessed from  
anywhere in the app

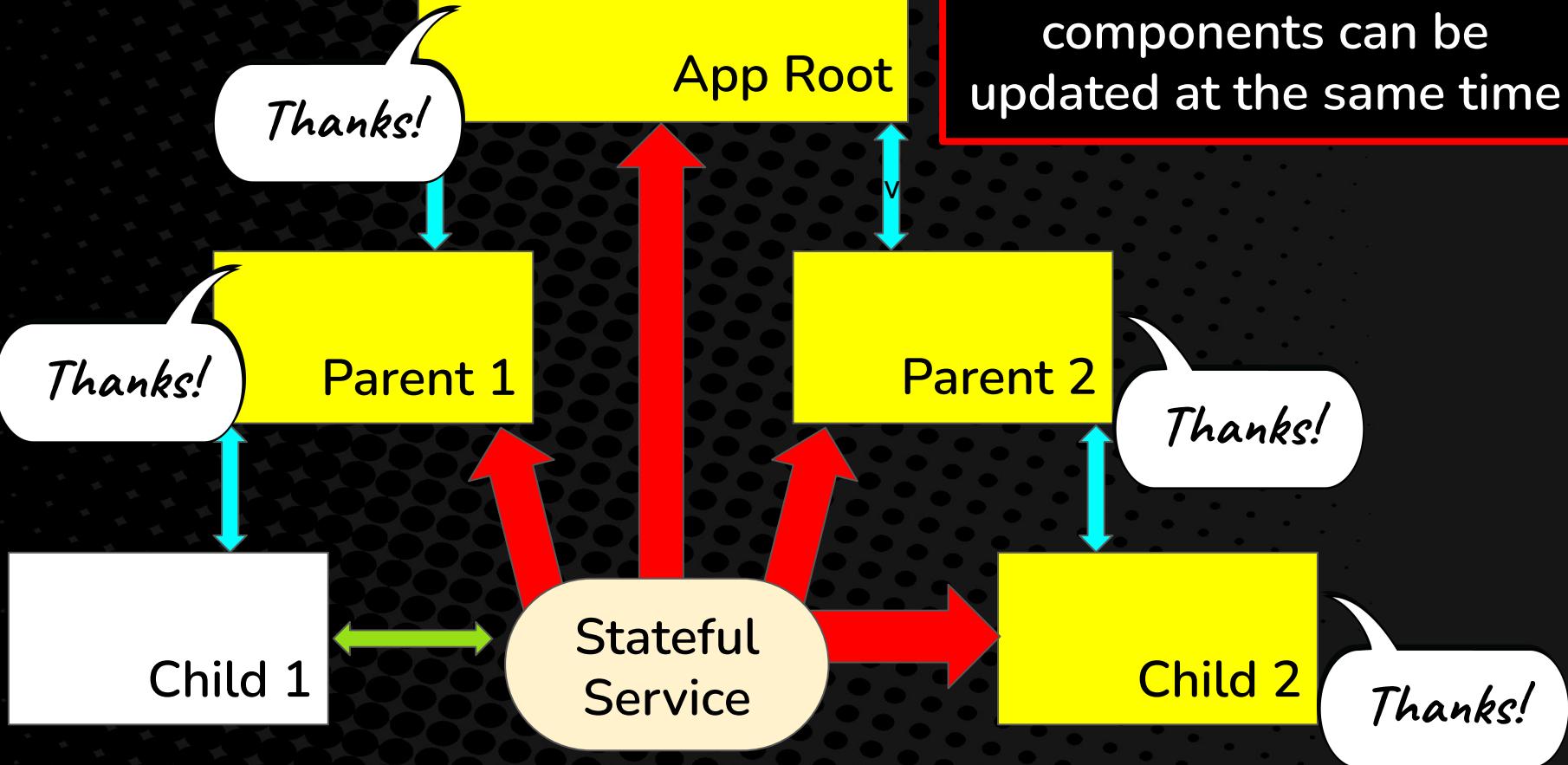








In fact ALL of the components can be updated at the same time



# *Built-in Stateful Options*

Service with a subject

Service with a Signal

Router Module

C ⓘ localhost:4200/products/glasses/detail?productId=dog-glasses-7

The screenshot shows a web browser window with the URL `localhost:4200/products/glasses/detail?productId=dog-glasses-7`. The page has a dark brown header bar with the logo "Just Like People PET BOUTIQUE" on the left and navigation links "PRODUCTS" and "CONTACT" on the right. Below the header, a breadcrumb trail reads "Home >> Products >> glasses >> dog-glasses-7". The main content area features a large image of a brown dog wearing glasses, with several smaller images of various pets (cats and dogs) wearing glasses arranged in a grid to the left. The title "Professional Doggo Glasses" is centered above the main image. To the right of the main image, there is descriptive text and a price. A share icon is visible in the top right corner of the browser window.

Just Like People  
PET BOUTIQUE

PRODUCTS CONTACT

Home >> Products >> glasses >> dog-glasses-7

## Professional Doggo Glasses



Whether they are starting their first day in IT or they are writing their first novel. These very professional glasses will give your doggo the confidence they need to succeed!

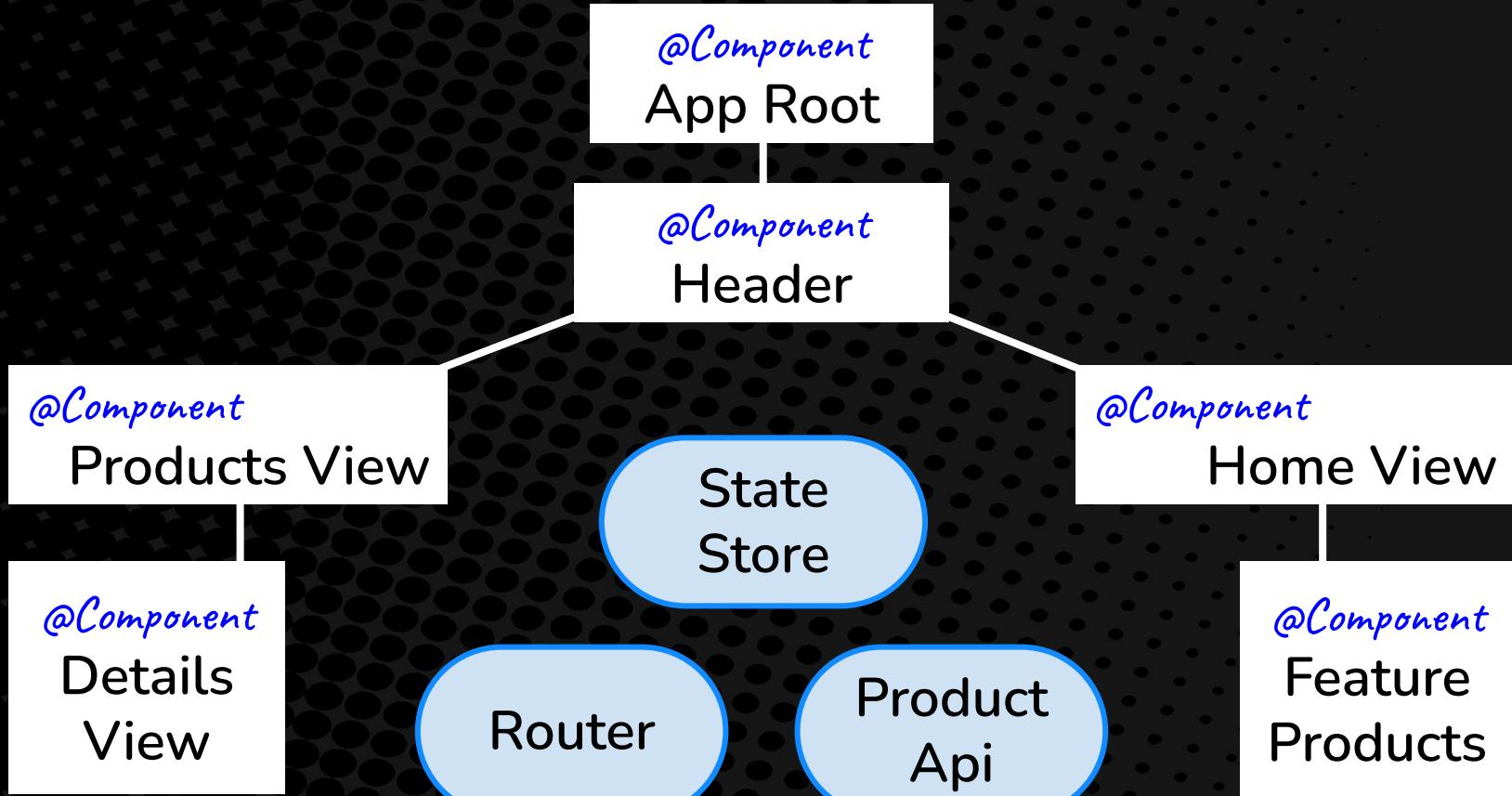
Tags: dog, glasses, successful, professional, Sys Admin

**Price: 18.99**

Users want to be able to share links to products



For shareable user selections, we can leverage the Angular Router



*The user selects  
a product*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

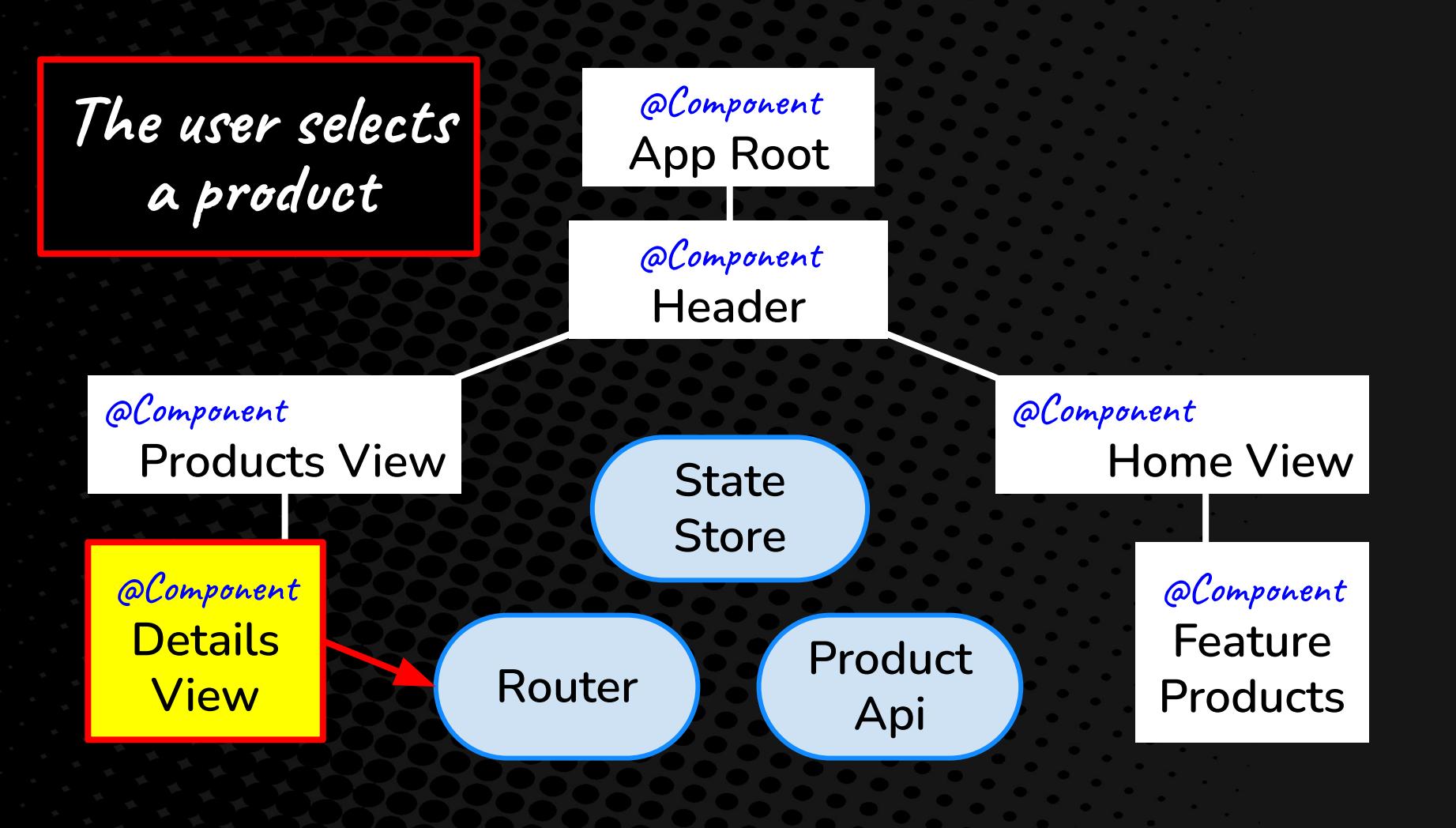
@Component  
Home View

@Component  
Feature Products

State  
Store

Router

Product  
Api



*We leverage the router to update the stateful store*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature Products

State  
Store

Product  
Api

Router

*State store  
emits a new value*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

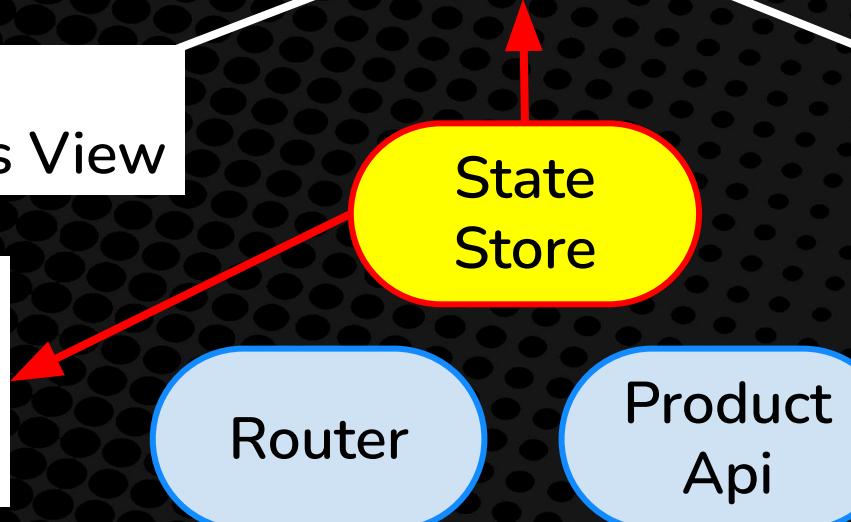
@Component  
Home View

@Component  
Feature Products

State  
Store

Router

Product  
Api



*The components  
that care about  
selected product  
get updated*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature Products

State  
Store

Router

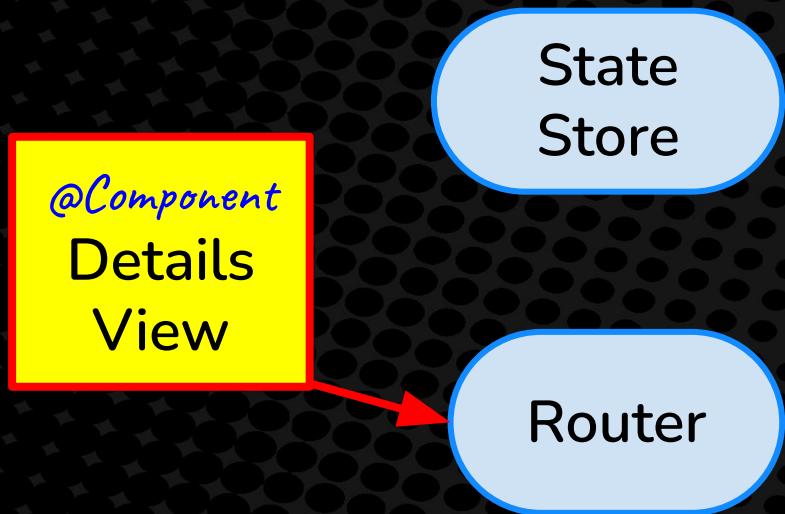
Product  
Api

*@Component*  
Details  
View

State  
Store

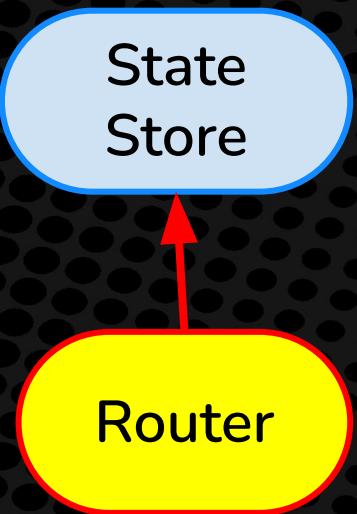
Router

Notice that even  
within a single  
component we are  
still driving state  
the same way

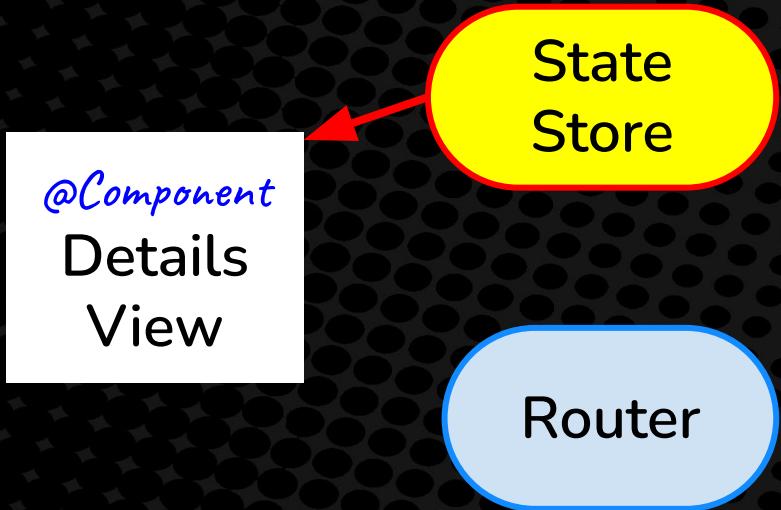


Notice that even within a single component we are still driving state the same way

*@Component*  
Details  
View



Notice that even within a single component we are still driving state the same way



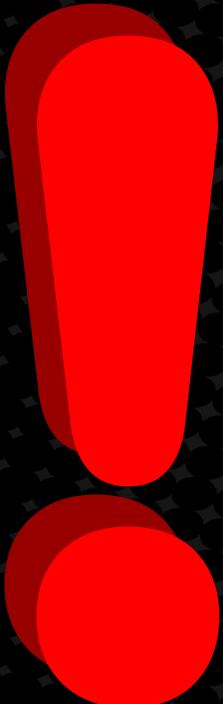
Notice that even within a single component we are still driving state the same way

*@Component*  
Details  
View

State  
Store

Router

Notice that even  
within a single  
component we are  
still driving state  
the same way



**Be careful to maintain  
a single flow to set state**

```
selectProduct(id: string){  
  → this.productService.setSelectedProduct(id);  
  this.router.navigate([''], {  
    queryParams: {  
      productId: id  
    }  
  });  
}
```

We want to avoid mixing responsibilities

```
selectProduct(id: string){  
  this.productService.setSelectedProduct(id);  
  this.router.navigate([''], {  
    queryParams: {  
      productId: id  
    }  
  });  
}
```

If the router is driving state changes for selected product, we don't want this method to also update the selected product.

A stylized illustration of a woman with long brown hair, wearing black-rimmed glasses and red lipstick. She is positioned on the left side of the frame. A large, white, rounded rectangular speech bubble originates from her mouth and extends towards the right. Inside the speech bubble, the text is centered.

**Let's combine the first pattern  
with the second pattern**

C ⓘ localhost:4200/products/glasses/detail?productId=dog-glasses-7

Just Like People  
PET BOUTIQUE

PRODUCTS ▾ CONTACT

Home >> Products >> glasses >> dog-glasses-7

## Professional Doggo Glasses

The initial producer is a click of a side menu item

```
<button  
  *ngFor="let product of products$ | async"  
  class="image-button"  
  [routerLink]=["'/products', product.category, 'detail']"  
  [queryParams]="{{productId: product.id}}"  
  routerLinkActive="router-link-active"  
>
```



*We change the route directly  
from the template using routerLink*

```
<button  
  *ngFor="let product of products$ | async"  
  class="image-button"  
  [routerLink]=["/products", product.category, 'detail']  
  [queryParams]="{{productId: product.id}}"  
  routerLinkActive="router-link-active"  
>
```

Here we are passing a path parameter called categoryId.

*Path params are declared in the route declaration.*

```
export class ProductViewComponent {  
  @Input() set categoryId(val: string) {  
    this.productsService.setSelectedCategory(val);  
  }  
  
  private readonly productsService = inject(ProductService);  
  
  readonly filteredProduct$ = this.productsService.filteredProducts$;
```

In this component we are consuming the categoryId path param from the router data.

A cartoon illustration of a man with dark hair and glasses, wearing a blue pinstripe suit and a red tie with black polka dots. He has his hand to his chin in a thoughtful pose. A large white speech bubble originates from his head, containing the text.

Wait a minute where are  
you consuming the path  
param????

```
export class ProductViewComponent {  
  @Input() set categoryId(val: string) {  
    this.productsService.setSelectedCategory(val);  
  }  
  
  private readonly productsService = inject(ProductService);  
  
  readonly filteredProduct$ = this.productsService.filteredProducts$;
```

Oh hey a  
setter!

As of version 16 we can consume all route  
params, and resolver data in inputs!

```
RouterModule.forRoot(ROUTES, {  
  bindToComponentInputs: true,  
},
```

To use this new feature update  
the Router configuration

# Router Data Input Binding

## Path params

Ideal for input setters because only the component activated by route can access them

VS

## Query params

Can be accessed from anywhere so be cautious about duplicating query param state



**Let's step through how  
this works in our component**

```
export class ProductViewComponent {  
  @Input() set categoryId(val: string) {  
    this.productsService.setSelectedCategory(val);  
  }  
  
  private readonly productsService = inject(ProductService);  
  
  readonly filteredProduct$ = this.productsService.filteredProducts$;
```

*Inside the Component*  
The **categoryId** path param changes

```
export class ProductViewComponent {  
  @Input() set categoryId(val: string) {  
    this.productsService.setSelectedCategory(val);  
  }  
  
  private readonly productsService = inject(ProductsService);  
  
  readonly filteredProduct$ = this.productsService.filteredProducts$;
```

The setter on the input invokes  
**setSelectedCategory** on the **ProductsService**

```
export class ProductService {
  private readonly selectedCategory = new BehaviorSubject<string>(Category.ALL);
  readonly selectedCategory$ = this.selectedCategory.asObservable();

  setSelectedCategory(category: string) {
    this.selectedCategory.next(category)
  }

  readonly filteredProducts$ = this.selectedCategory.pipe(
    switchMap((category) => this.products$.pipe(
      map((products) => {
        if (category === Category.ALL) {
          return products;
        }
        return products.filter((product) => product.category === category);
      })
    ))
  );
}
```

*Inside the ProductService*

**setSelectedCategory** calls next on the  
**selectedCategory** BehaviorSubject

```
export class ProductService {  
    private readonly selectedCategory = new BehaviorSubject<string>(Category.ALL);  
    readonly selectedCategory$ = this.selectedCategory.asObservable();
```

```
    setSelectedCategory(category: string) {  
        this.selectedCategory.next(category);  
    }
```

```
    read  
    sw
```

Since the subject is private we expose the observable on a public property

```
        return products,  
    }  
  
    return products.filter((product: Product) => product.category === category);  
},  
)  
);
```

```
export class ProductService {  
  pri...  
  read...  
  sets...  
  th...  
}
```

Subjects are multicast so anything subscribed to the **selectedCategory** observable will get the new selected category value

```
readonly filteredProducts$ = this.selectedCategory.pipe(←  
  switchMap((category) => this.products$.pipe(  
    map((products) => {  
      if(category === Category.ALL) {  
        return products;  
      }  
  
      return products.filter((product: Product) => product.category === category);  
    }),  
  ))  
;
```

A cartoon illustration of a man with dark hair, wearing round white-framed glasses, a white shirt, and a red tie with black polka dots. He is resting his chin on his right hand, looking thoughtful. A large white speech bubble originates from his head and extends to the right.

**Do it again but this  
time with signals**

```
export class ProductViewComponent {  
  @Input() set categoryId(val: string) {  
    this.productsService.selectedCategory.set(val);  
  }  
  
  private readonly productsService = inject(ProductSignalsService);  
  
  readonly filteredProducts$ = this.productsService.filteredProducts;
```

*Inside the Component*  
The **categoryId** path param changes

```
export class ProductViewComponent {  
  @Input() set categoryId(val: string) {  
    this.productsService.selectedCategory.set(val);  
  }  
  
  private readonly productsService = inject(ProductSignalsService);  
  
  readonly filteredProducts$ = this.productsService.filteredProducts;
```

The setter on the input sets the **selectedCategory** signal value in the **ProductService**

*Wouldn't it be great if the Input property  
could just BE a Signal...*

```
export class ProductSignalsService {  
  readonly selectedCategory = signal<string>(Category.ALL); ←  
  
  readonly filteredProducts = computed(() => {  
    if(this.selectedCategory() === Category.ALL) {  
      return PRODUCTS;  
    }  
  
    return PRODUCTS.filter(  
      (product: Product) => product.category === this.selectedCategory()  
    );  
  });
```

*Inside the ProductService*

Since signals have built in getters and setters,  
we can leave the property public.

```
export class ProductSignalsService {
  readonly selectedCategory = signal<string>(Category.ALL);

  readonly filteredProducts = computed(() => {
    if(this.selectedCategory() === Category.ALL) {
      return PRODUCTS;
    }

    return PRODUCTS.filter(
      (product: Product) => product.category === this.selectedCategory()
    );
  });
}
```

Signals track consumers so when the signal gets a new value this computed will update and return a new value



**Compose data before it  
gets to the consumers**

*Why compose  
data?*

Components have less responsibility

Easier to test in services

All consumers get the exact same values



Remember all the consumer wants to do is consume.

# Let's combine our patterns!

one

Input setters

two

Emit events  
where they happen

three

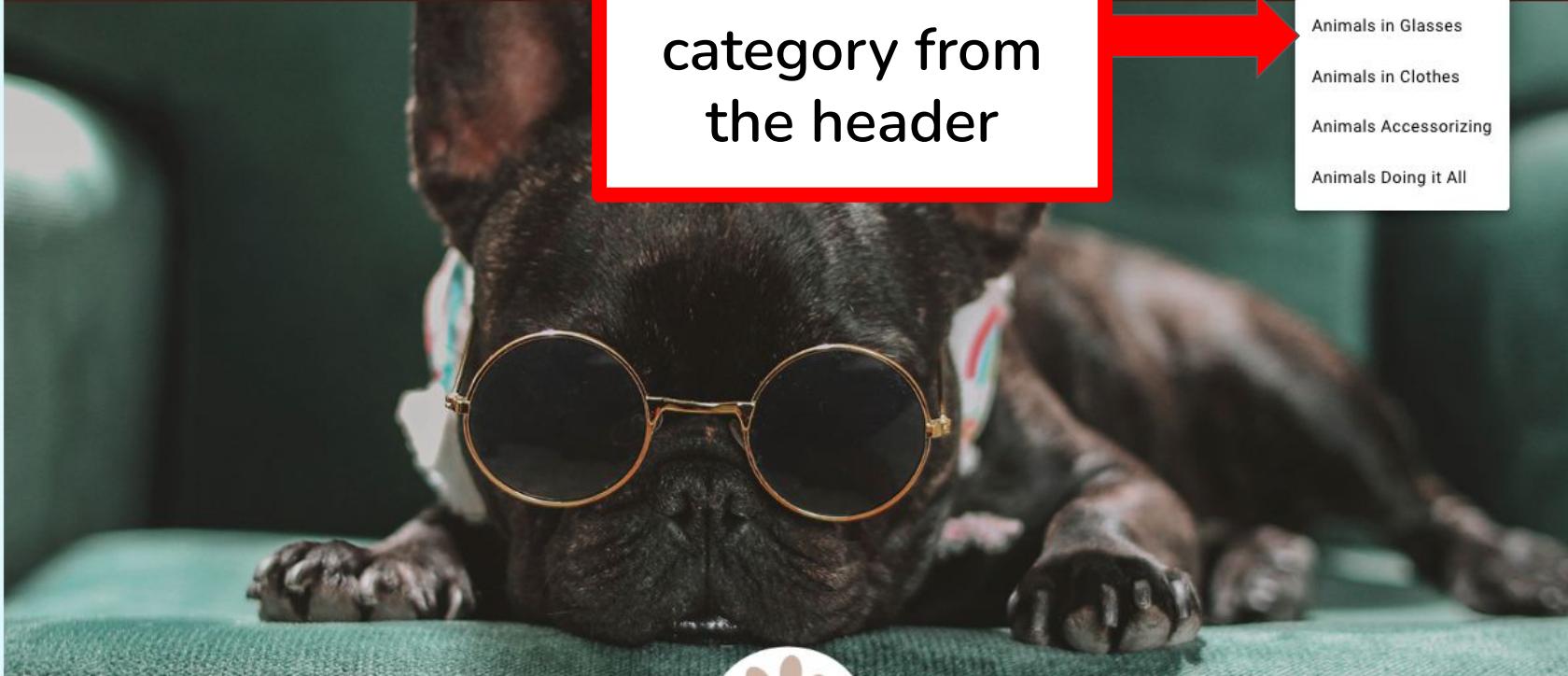
Compose data before it gets  
to the consumers



Users select a category from the header

PRODUCTS ▾ CONTACT

- [Animals in Hats](#)
- [Animals in Glasses](#)
- [Animals in Clothes](#)
- [Animals Accessorizing](#)
- [Animals Doing it All](#)



Home >> Products >> hats >> cat-bonnet



## Baby Blue Baby Bonnet for Cats



Products filtered by category populate the side menu

This beautiful egg shell blue baby bonnet is purrfect for all of your bestest babies. This bonnet is hand knit by midwestern grandmas.

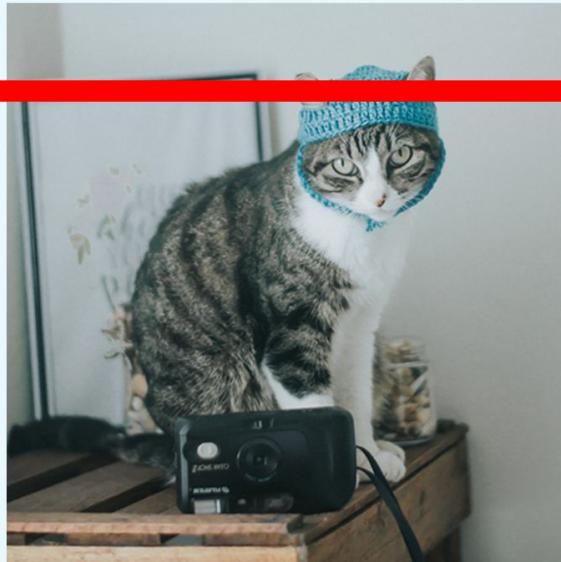
Tags: cat, bonnet, hat, baby, knit

Price: 32.99

Home >> Products >> hats >> cat-bonnet



## Baby Blue Baby Bonnet for Big Baby Kitties



Users can select products from the side menu

This beautiful egg shell blue baby bonnet is purrfect for all of your bestest babies. This bonnet is hand knit by midwestern grandmas.

Tags: cat, bonnet, hat, baby, knit

Price: 32.99



PRODUCTS ▾ CONTACT



## Because animals are people too.

And you can't be a people without accessories.



Puppy Sized Banana Onesie

24.99



Yellow Rubber Boots for Dogs

23.99



Red Sweater for Kitty

13.99



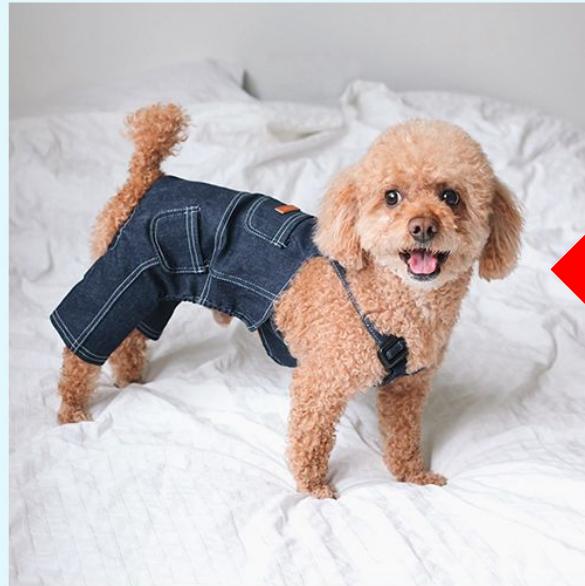
We can also  
select products  
from the home  
page

## Denim Pooch Pants ON SALE NOW!!!

Home >> Products >> all >> overalls-puppy



### Denim Pooch Pants



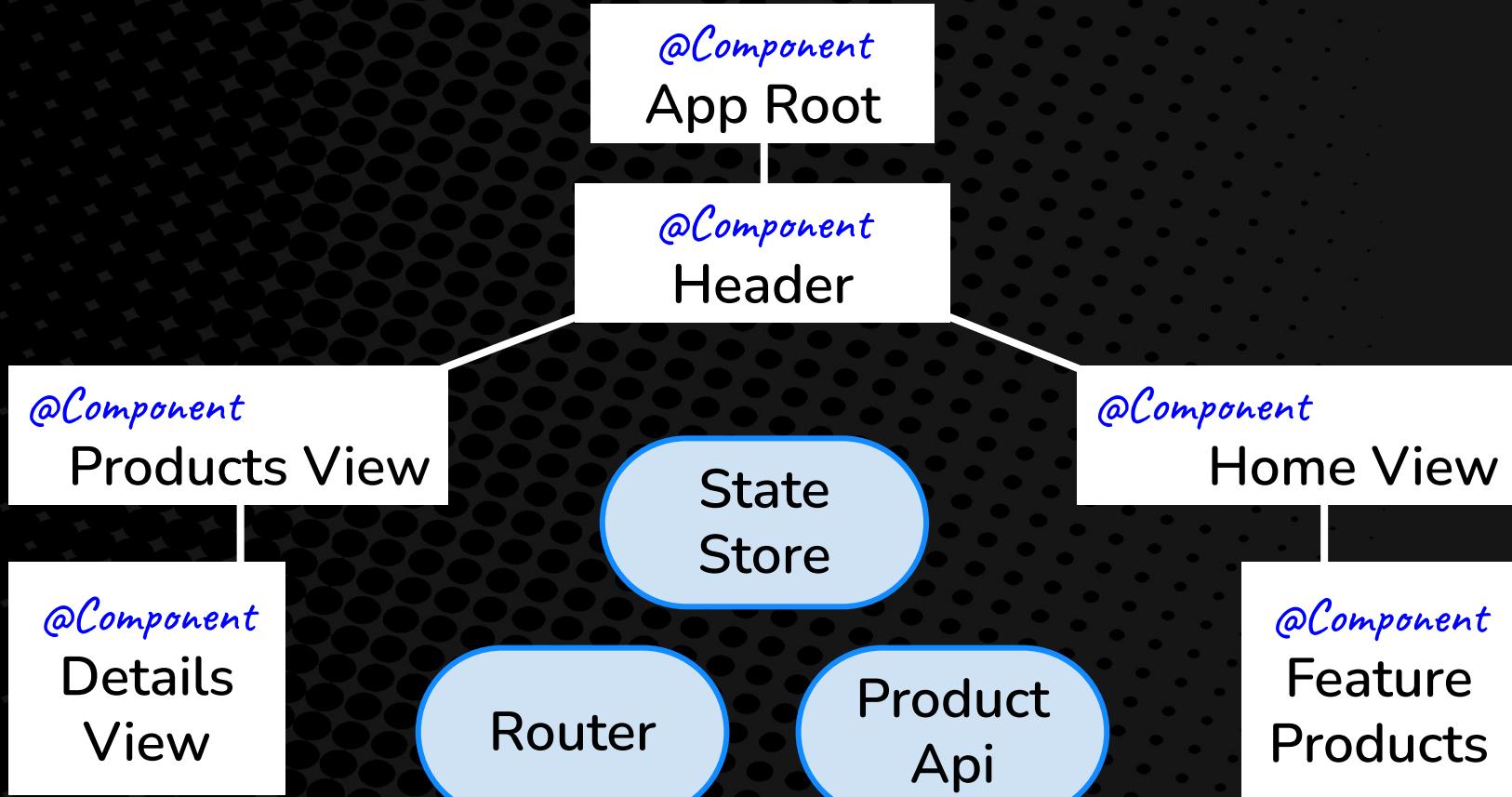
The header and the detail view need the selected product

These rugged and durable overalls are great for little poochies who work hard and play harder! Now with 100% more pockets!

Tags: dog, work pants, overalls, denim, small dogs

Price: 32.99 **ON SALE!!**

*Let's see the state flow  
for selecting a category*



**CLICK!**  
User selects  
a category

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature Products

State  
Store

Router

Product  
Api

Router

Product  
Api

*The router routes to  
the Products View*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature Products

State  
Store

Product  
Api

Router

The categoryId input  
updates the state  
store

@Component  
Products View

@Component  
Details  
View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature  
Products

State  
Store

Router

Product  
Api



*The state store  
emits a new value for  
selected category*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature Products

State  
Store

Router

Product  
Api

*The product view gets the updated filtered products*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature Products

State Store

Router

Product Api

*Any other components that need filtered products is also updated*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

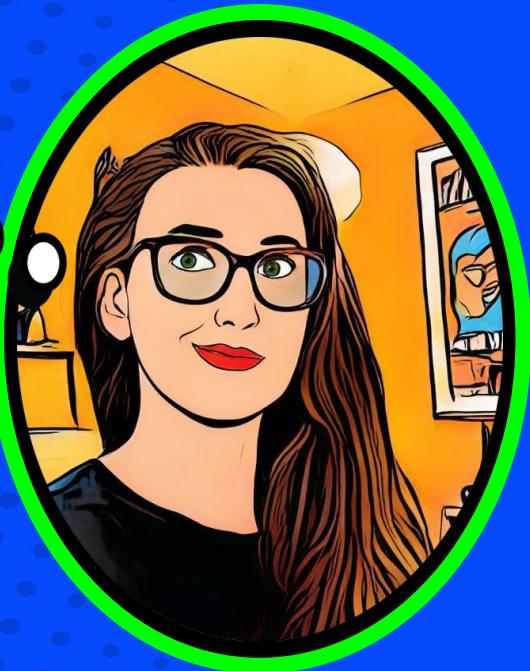
@Component  
Feature Products

State Store

Router

Product Api

*Let's see how we compose and  
consume the data*



```
export class ProductService {  
  pri...  
  read...  
  sets...  
  th...  
}
```

We compose the data inside  
a declared stream

```
readonly filteredProducts$ = this.selectedCategory.pipe(←  
  switchMap((category) => this.products$.pipe(  
    map((products) => {  
      if(category === Category.ALL) {  
        return products;  
      }  
  
      return products.filter((product: Product) => product.category === category);  
    }),  
  ))  
;
```

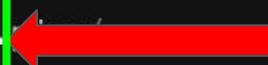
```
export class ProductService {  
  pri...  
  read...  
  sets...  
  th...  
}
```

We use switchMap to get access to the values  
inside the products observable and the  
selectedCategory observable

```
readonly filteredProducts$ = this.selectedCategory.pipe(  
  switchMap((category) => this.products$.pipe(  
    map((products) => {  
      if(category === Category.ALL) {  
        return products;  
      }  
  
      return products.filter((product: Product) => product.category === category);  
    }),  
  )),  
);
```

```
export class ProductService {  
  pri...  
  read...  
  sets...  
  th...  
}
```

SelectedCategory is the outer observable  
because it is the value that changes

```
readonly filteredProducts$ = this.selectedCategory.  
  switchMap((category) => this.products$.pipe(  
    map((products) => {  
      if(category === Category.ALL) {  
        return products;  
      }  
  
      return products.filter((product: Product) => product.category === category);  
    }),  
  ))  
;
```

Products are fetched from an api when the service initializes and stored in a behavior subject  
***products\$** is the inner observable*

```
readonly filteredProducts$ = this.selectedCategory.pipe(  
  switchMap((category) => this.products$.pipe(  
    map((products) => {  
      if(category === Category.ALL) {  
        return products;  
      }  
  
      return products.filter((product: Product) => product.category === category);  
    }),  
  ))  
;
```

```
export class ProductService {  
  private products$: Observable<Product>;  
  readonly selectedCategory$: Observable<Category>;  
  
  setSelectedCategory(category: Category) {  
    this.selectedCategory$.next(category);  
  }  
  
  readonly filteredProducts$ = this.selectedCategory$.pipe(  
    switchMap((category) => this.products$.pipe(  
      map((products) => {  
        if(category === Category.ALL) {  
          return products;  
        }  
  
        return products.filter((product: Product) => product.category === category);  
      }),  
    )),  
  );  
};
```

We pipe off of the products observable and use map to filter the products array by the selected category

```
export class ProductViewComponent {  
  @Input() set categoryId(val: string) {  
    this.productsService.setSelectedCategory(val);  
  }  
  
  private readonly productsService = inject(ProductService);  
  
  readonly filteredProduct$ = this.productsService.filteredProducts$;
```

In the component, we declare a property equal to the **filteredProducts\$** observable

```
<button  
  *ngFor="let product of filteredProducts$ | async"  
    [routerLink]=["'detail'"]  
    [queryParams]="{  
      productId: product.id  
    }"  
    routerLinkActive="router-link-active"  
>
```

The template is subscribed to `filteredProducts$` using the `async` pipe, so the new products are rendered to the page

*Why the  
async pipe?*

Handles subscribing on init

Handles unsubscribing  
on destroy

Marks the component for check  
when new values arrive



That's super cool, but I  
heard we get signals

```
export class ProductSignalsService {
  readonly selectedCategory = signal<string>(Category.ALL);

  readonly filteredProducts = computed(() => {
    if(this.selectedCategory() === Category.ALL) {
      return PRODUCTS;
    }

    return PRODUCTS.filter(
      (product: Product) => product.category === this.selectedCategory()
    );
  });
}
```

When `selectedCategory` updates, the computed reruns the callback function

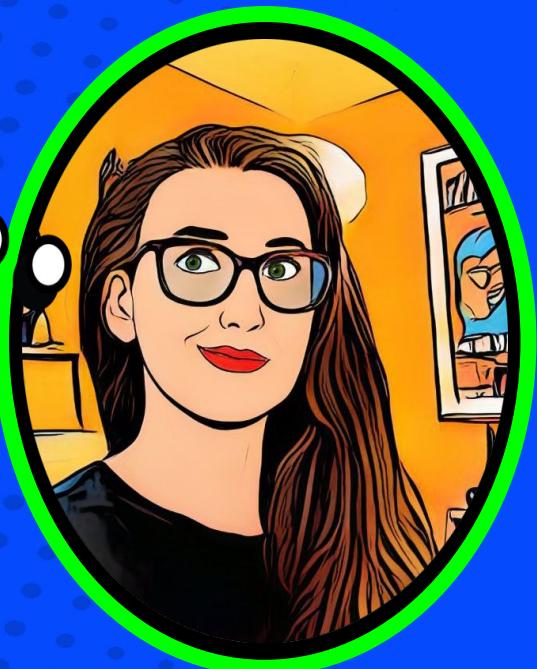
```
export class ProductViewComponent {  
  @Input() set categoryId(val: string) {  
    this.productsService.selectedCategory.set(val);  
  }  
  
  private readonly productsService = inject(ProductSignalsService);  
  
  readonly filteredProducts$ = this.productsService.filteredProducts;
```

In the component, we declare a property equal to the **filteredProducts** computed signal

```
<button  
  *ngFor="let product of filteredProducts()" style="border: none; background-color: transparent; padding: 0; margin-right: 10px;">  
    [routerLink]=["'detail'"]  
    [queryParams]="{  
      productId: product.id  
    }"  
    routerLinkActive="router-link-active"  
  >  
  </button>
```

In the template we can use the value of  
the **filteredProducts** signal directly

Notice how similar the pattern is for observables and for signals



*Let's walk through  
the flow for selected  
product*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature Products

State  
Store

Router

Product  
Api

**CLICK!**  
The user selects  
a product

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

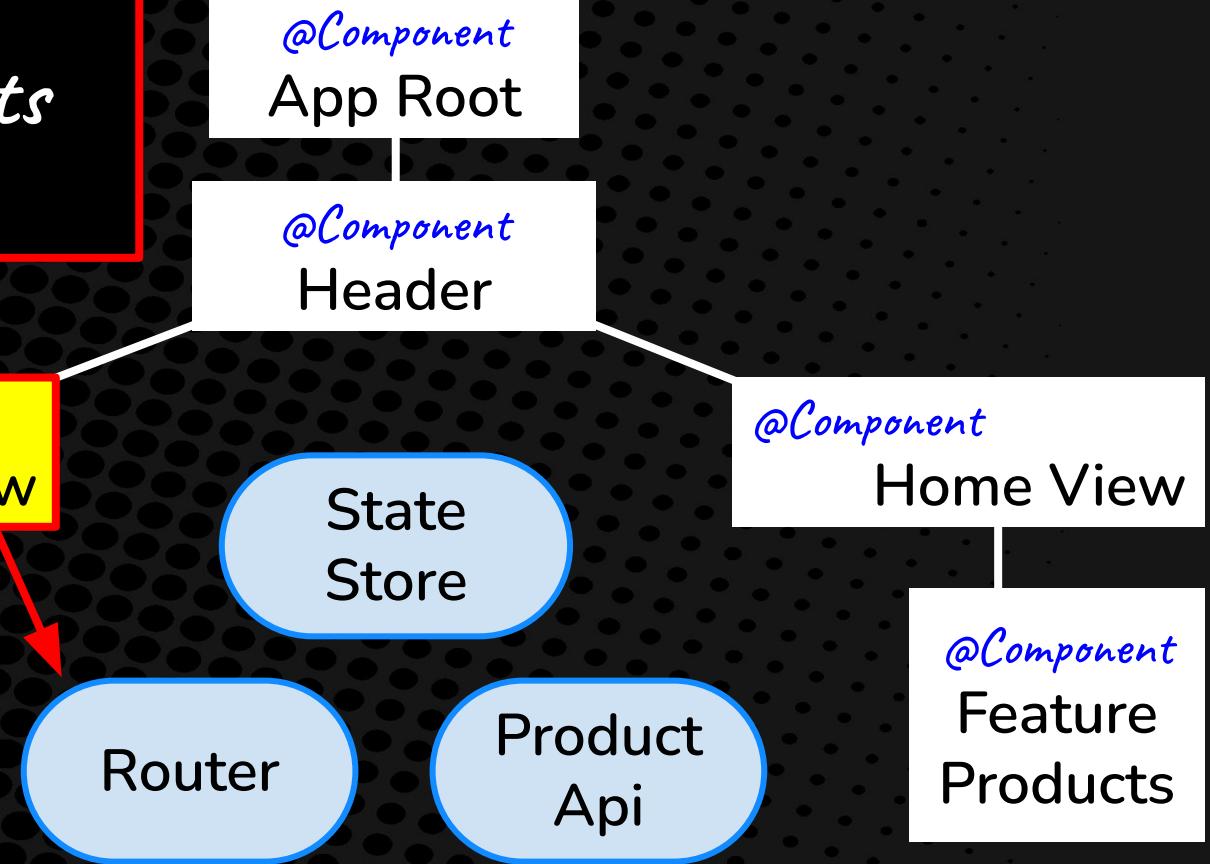
@Component  
Home View

@Component  
Feature Products

State  
Store

Router

Product  
Api



*This updates query  
params so the state  
store is updated by  
the router*

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

@Component  
Home View

@Component  
Feature Products

State  
Store

Product  
Api

Router

The state store  
emits a new selected  
product value

@Component  
Products View

@Component  
Details View

@Component  
App Root

@Component  
Header

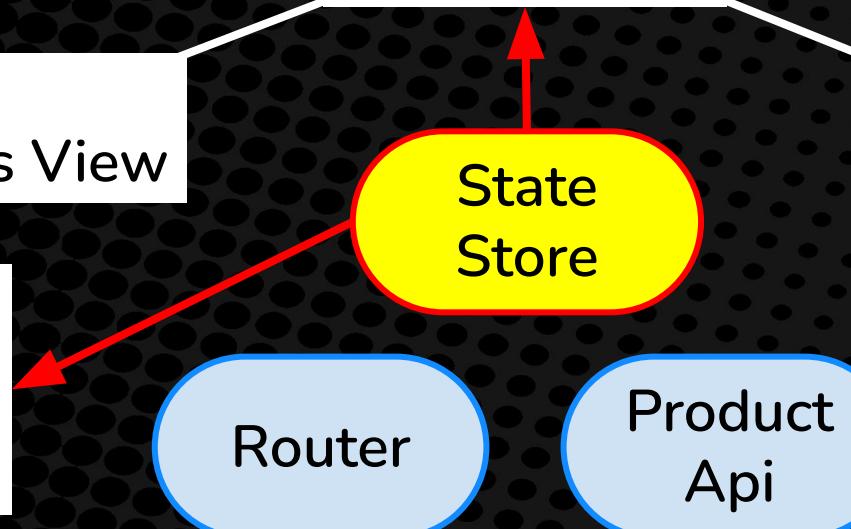
@Component  
Home View

@Component  
Feature Products

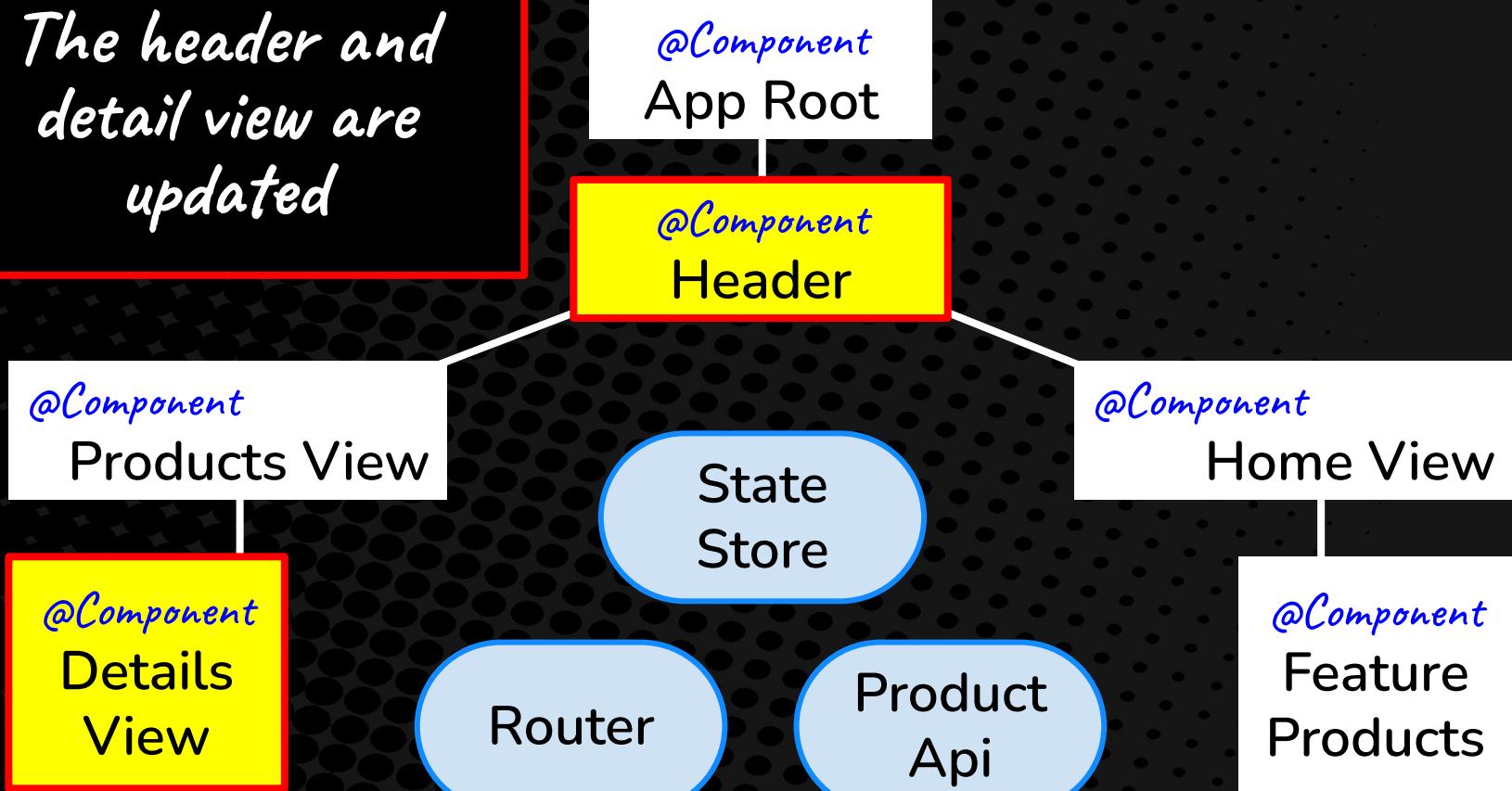
State  
Store

Router

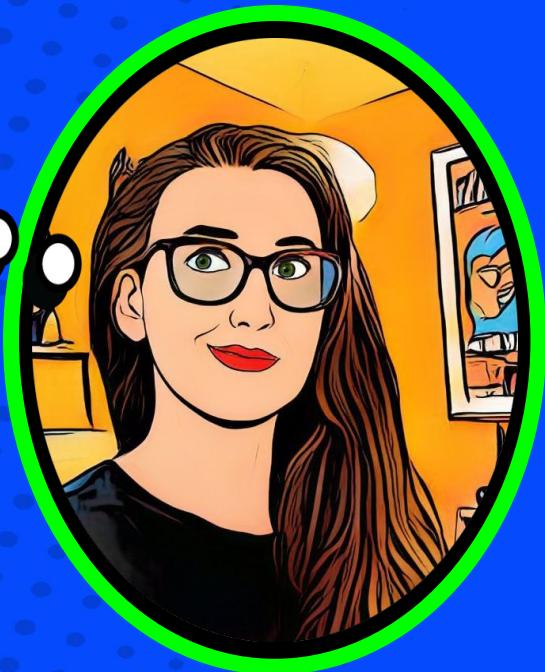
Product  
Api



The header and detail view are updated



*Since the router updates user selections in the state store, users can directly link to products.*



# **MEANWHILE**

*In the code...*

```
<button  
  *ngFor="let product of filteredProducts()"  
    [routerLink]=["'detail'"]  
    [queryParams]="{  
      productId: product.id  
    }"  
    routerLinkActive="router-link-active"  
  >  
  </button>
```

This time we pass query params

```
export class ProductService {
  private readonly activatedRoute = inject(ActivatedRoute);
  readonly selectedProduct$ = this.activatedRoute.queryParams.pipe(
    map((queryParams) => queryParams['productId']),
    switchMap((id) =>
      this.products$.pipe(
        map((products) => {
          return products.find((product) => product.id.toLowerCase() === id?.toLowerCase());
        })
      )
    )
  );
}
```

This service injects the activated route and uses the queryParams observable

```
export class ProductService {
  private readonly activatedRoute = inject(ActivatedRoute);

  readonly selectedProduct$ = this.activatedRoute.queryParams.pipe(
    map((queryParams) => queryParams['productId']), ←
    switchMap((id) =>
      this.products$.pipe(
        map((products) => {
          return products.find((product) => product.id.toLowerCase() === id?.toLowerCase());
        })
      )
    )
  );
}
```

We get the productId query param

```
export class ProductService {
  private readonly activatedRoute = inject(ActivatedRoute);

  readonly selectedProduct$ = this.activatedRoute.queryParams.pipe(
    map((queryParams) => queryParams['productId']),
    switchMap((id) =>
      this.products$.pipe(
        map((products) => {
          return products.find((product) => product.id.toLowerCase() === id?.toLowerCase());
        })
      )
    )
  );
}
```

We follow a similar pattern to filtering products

```
private readonly productsService = inject(ProductService);  
  
readonly selectedProduct$ = this.productsService.selectedProduct$;
```

In the consuming components, we declare a property  
equal to the **selectedProduct** coming  
from the stateful service

```
<ng[red arrow] *ngIf="selectedProduct$ | async as selectedProduct">
  <div *ngIf="selectedProduct.onSale" class="sale-banner">
    <span>{{ selectedProduct.title }}</span>
    <span>{{ ' On Sale Now!!!!' | uppercase }}</span>
  </div>
```

```
<div c[red arrow] *ngIf="selectedProduct$ | async as selectedProduct">
  <h3>{{ selectedProduct.title }}</h3>
  <div class="image-wrapper">
    <img [src]="selectedProduct.image" [alt]="selectedProduct.title" />
    <div class="description">
      <p>{{ selectedProduct.description }}</p>
      <h4>Tags:
    </div>
  </div>
```

We bind the consuming component templates  
to the **selectedProduct** property

Denim Pooch Pants **ON SALE NOW!!!**

Home >> Products >> all >> overalls-puppy



## Denim Pooch Pants



In the view the header and the detail view update at the same time

These rugged and durable overalls are great for little poochies who work hard and play harder! Now with 100% more pockets!

Tags: dog, work pants, overalls, denim, small dogs

Price: 32.99 **ON SALE!!**

## Denim Pooch Pants ON SALE NOW!!!

Home >> Products >> all >> overalls-puppy



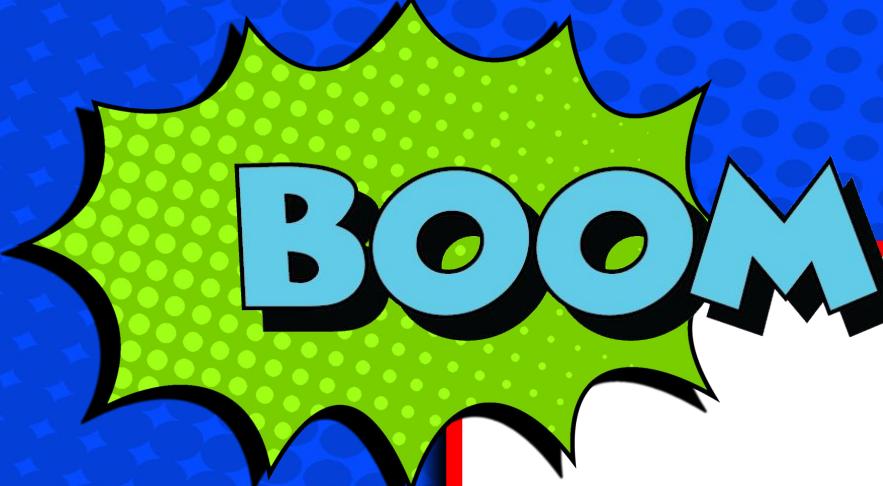
### Denim Pooch Pants



These rugged and durable overalls are great for little poochies who work hard and play harder! Now with 100% more pockets!

Tags: dog, work pants, overalls, denim, small dogs

Price: 32.99 **ON SALE!!**



**BOOM**

When we apply all three patterns together, we have defined a holistic data flow pattern that can be followed throughout the application.



So you are saying signals  
are still in developer  
preview, BUT...

I can make changes today  
to get my code ready to use  
them in production?



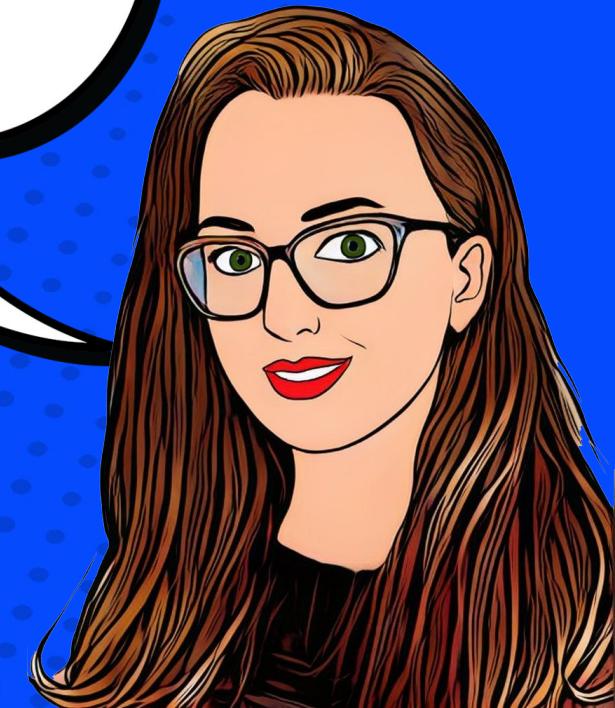
**YES!**  
*We are getting  
ready by...*

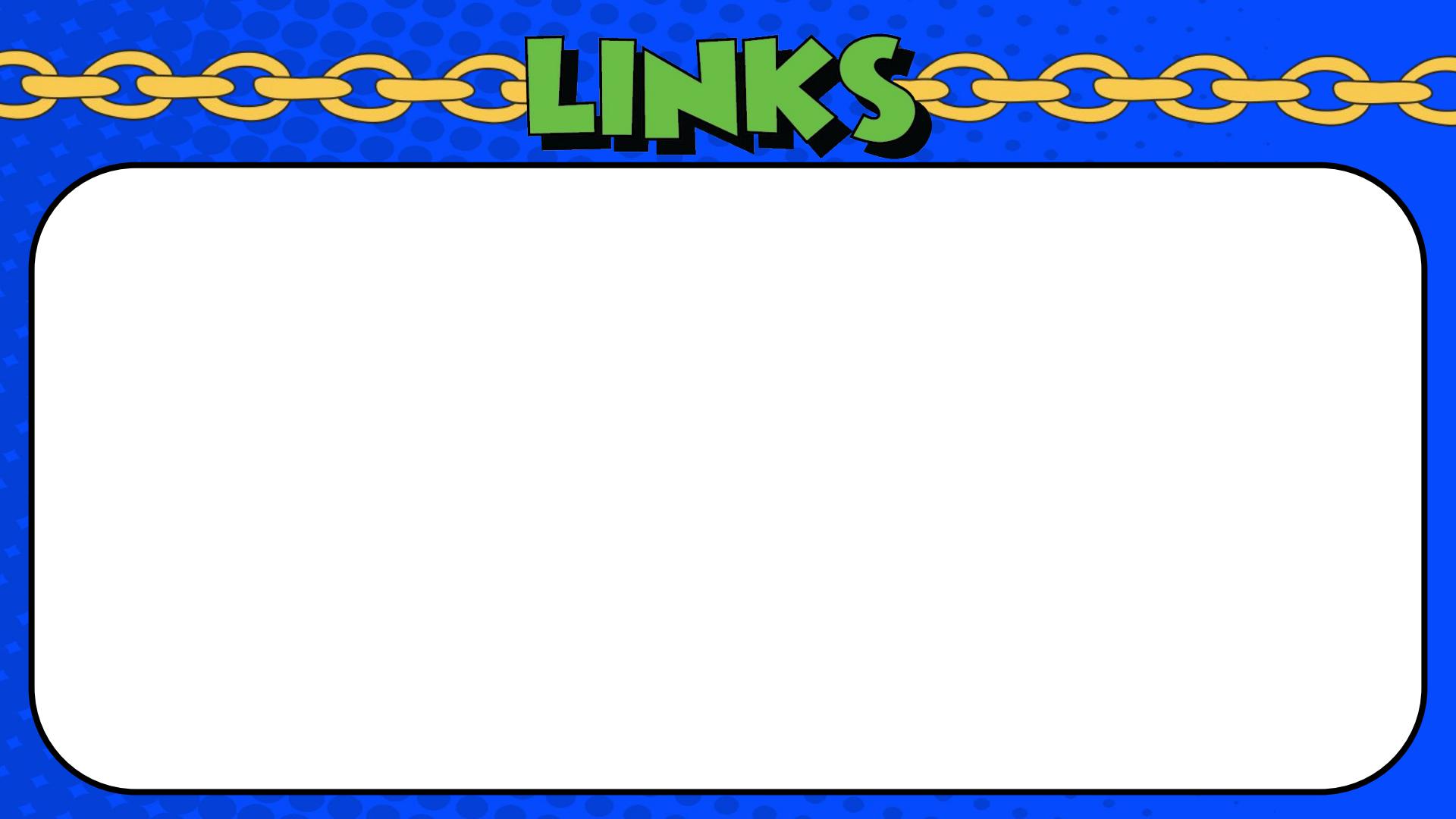
Orchestrating a clear flow of data through the application

Creating single sources of truth

Composing data so all consumers get the same data

And you'll also find that even without Signals defining good reactive patterns can really improve your code





# LINKS

# Slide deck