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

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.gle/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

Producers

Consumers

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)

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

Excessive class properties

Tightly coupled code

Random side effects
from void methods

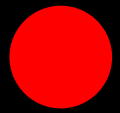
In my experience with Angular applications...

Imperative coding
+ time =

Declarative Programming

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

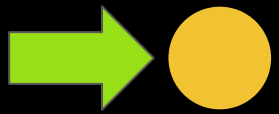
Declarative Programming



Typical of functional languages

Results oriented approach

This is what async apps need



Stateless & execution order agnostic

Declarative libraries & languages

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

Declarative

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

The cats$ property IS the getCats 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

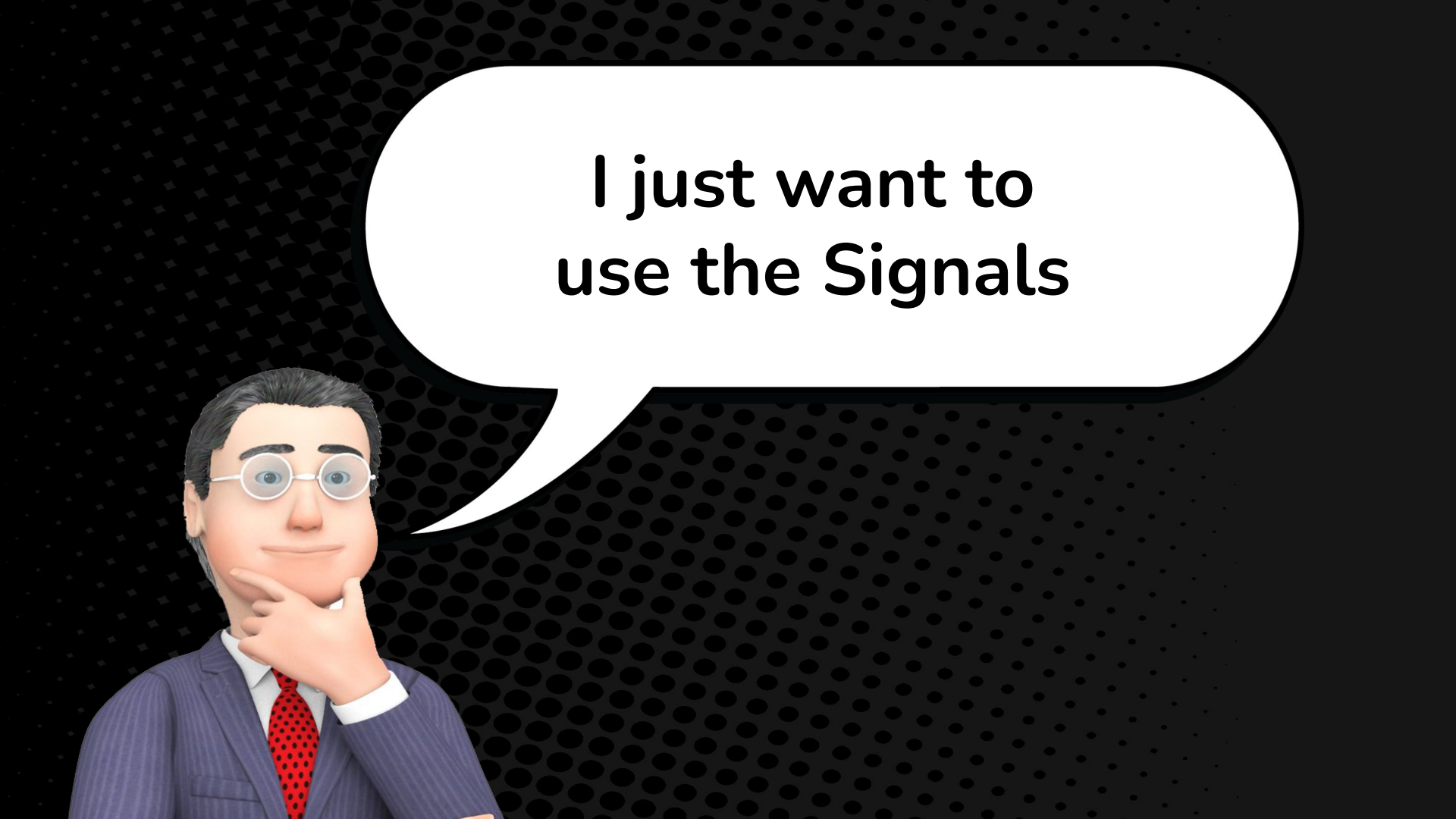
Higher learning curve

Not always as easy to read

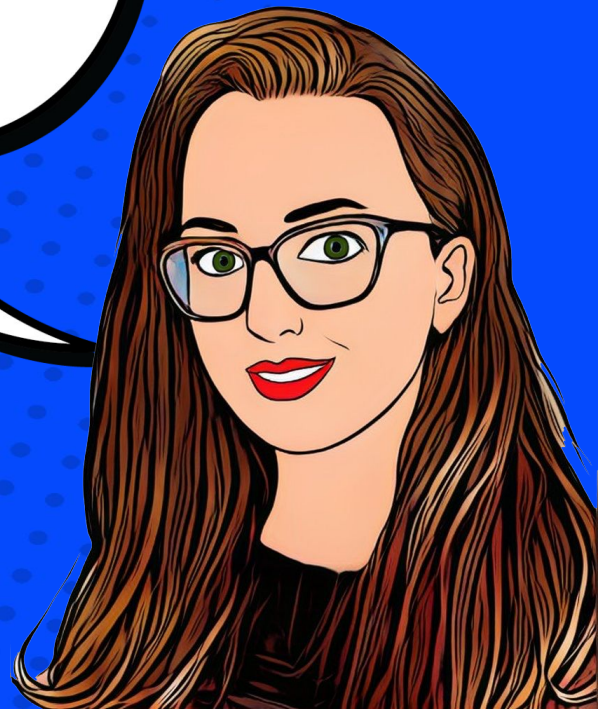
Not how most of us are taught to think about code

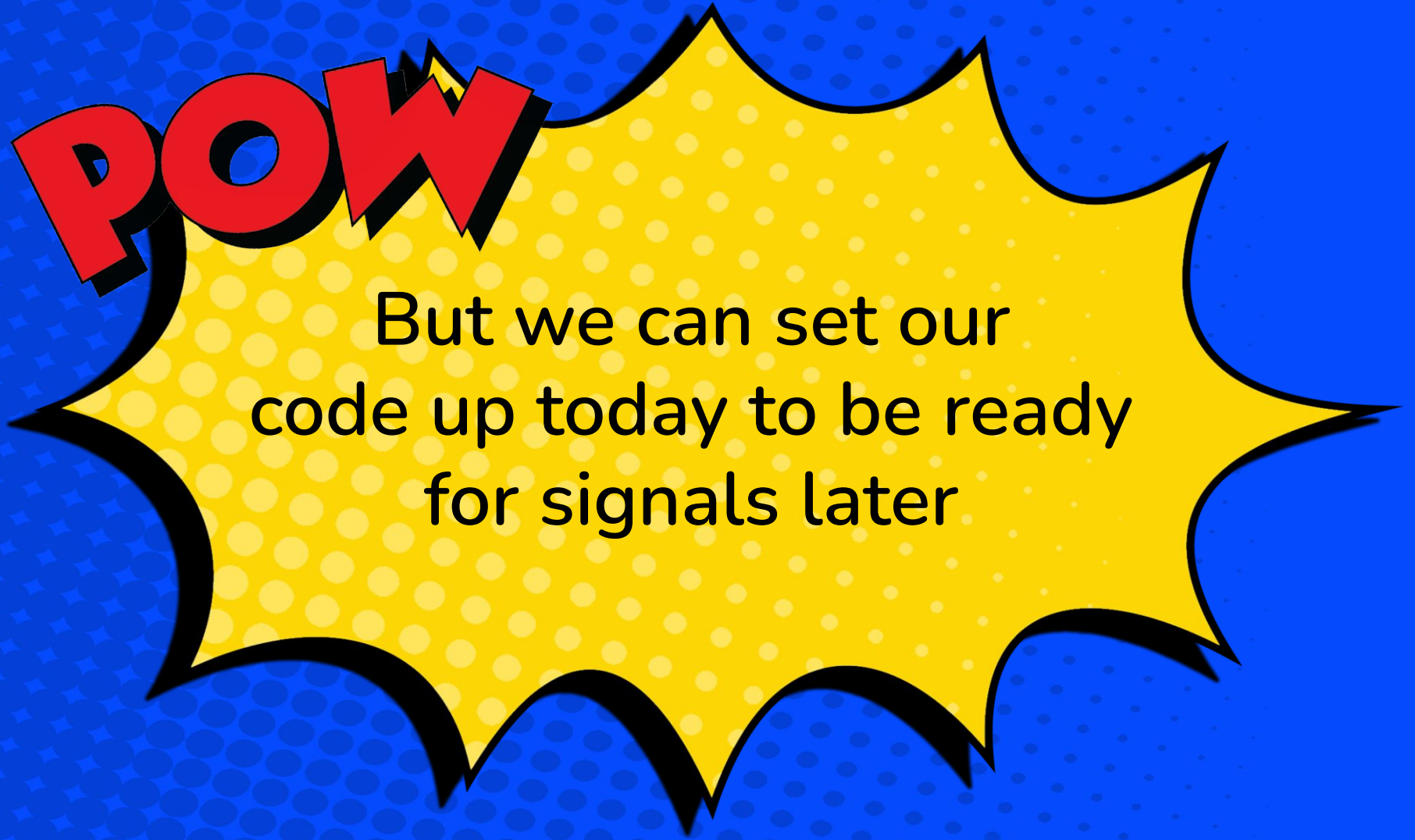
Writing declarative code is just

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!

```
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.

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

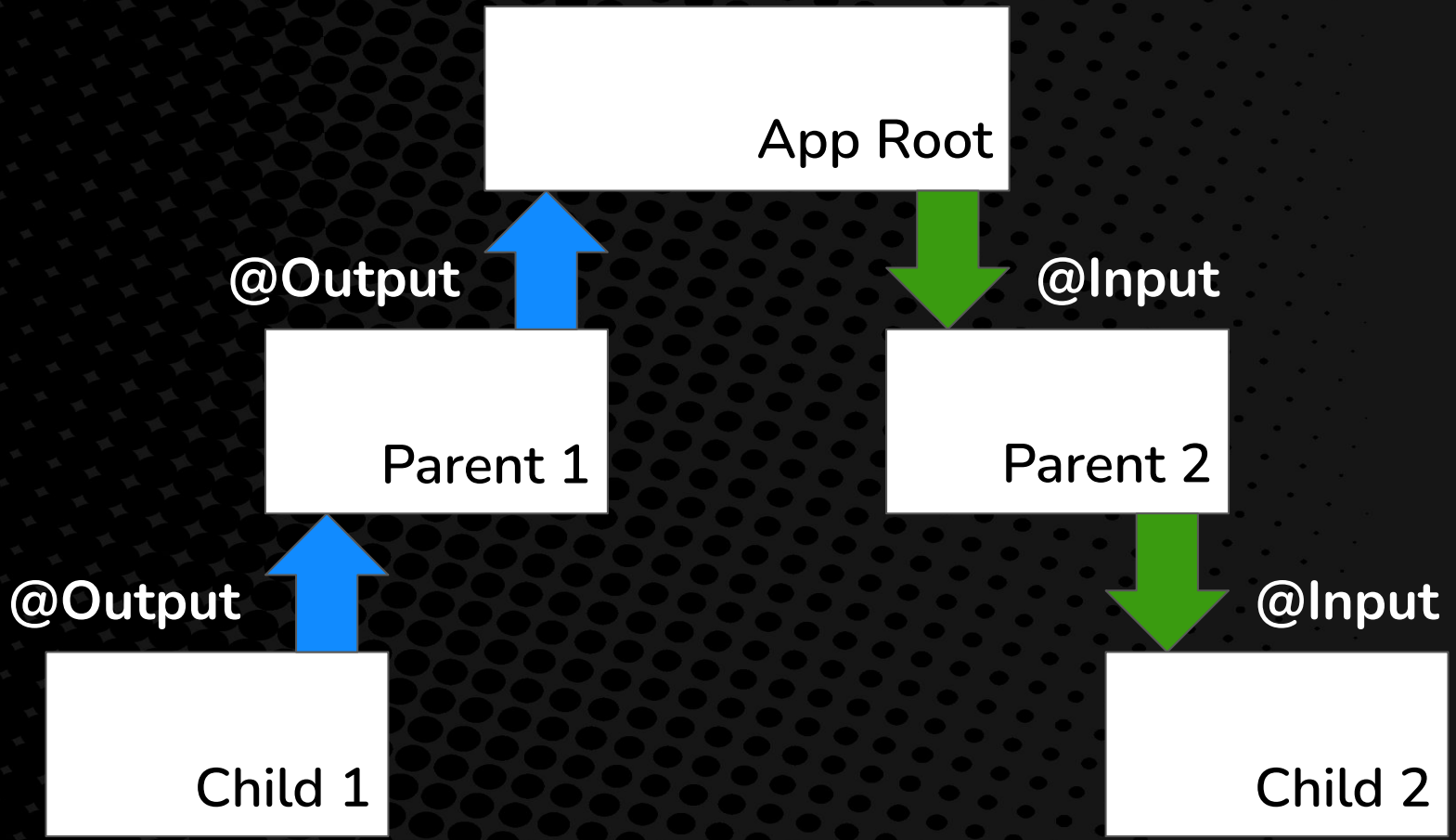
Not great tool for passing data to other parts of the app

Tightly couples the emitting component to the tree

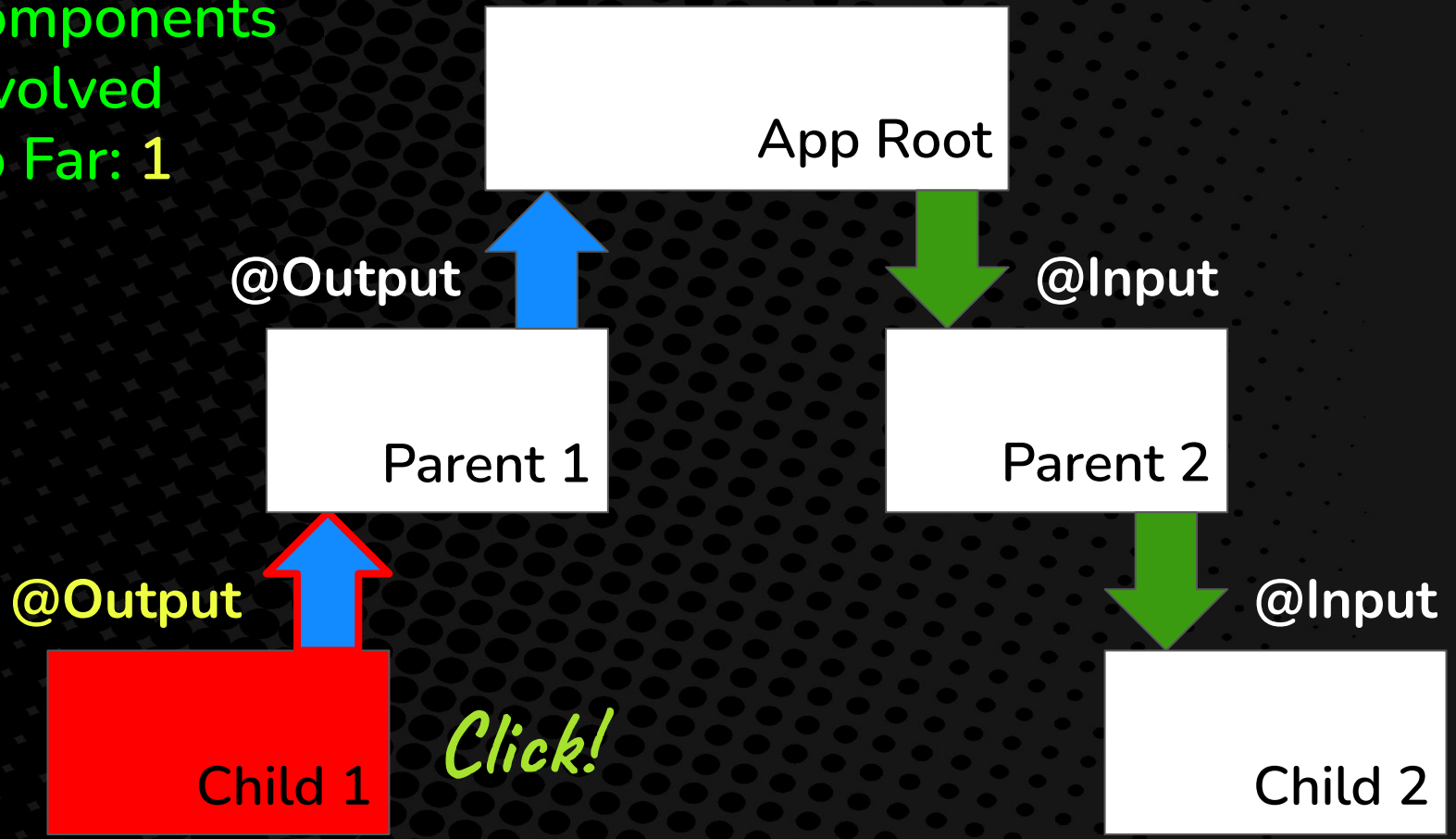
```
export class SideMenuComponent {
  @Output() selectedProduct = new EventEmitter<string>;

  selectProduct(id: string){
    this.selectedProduct.emit(id);
  }
```

Emits event here

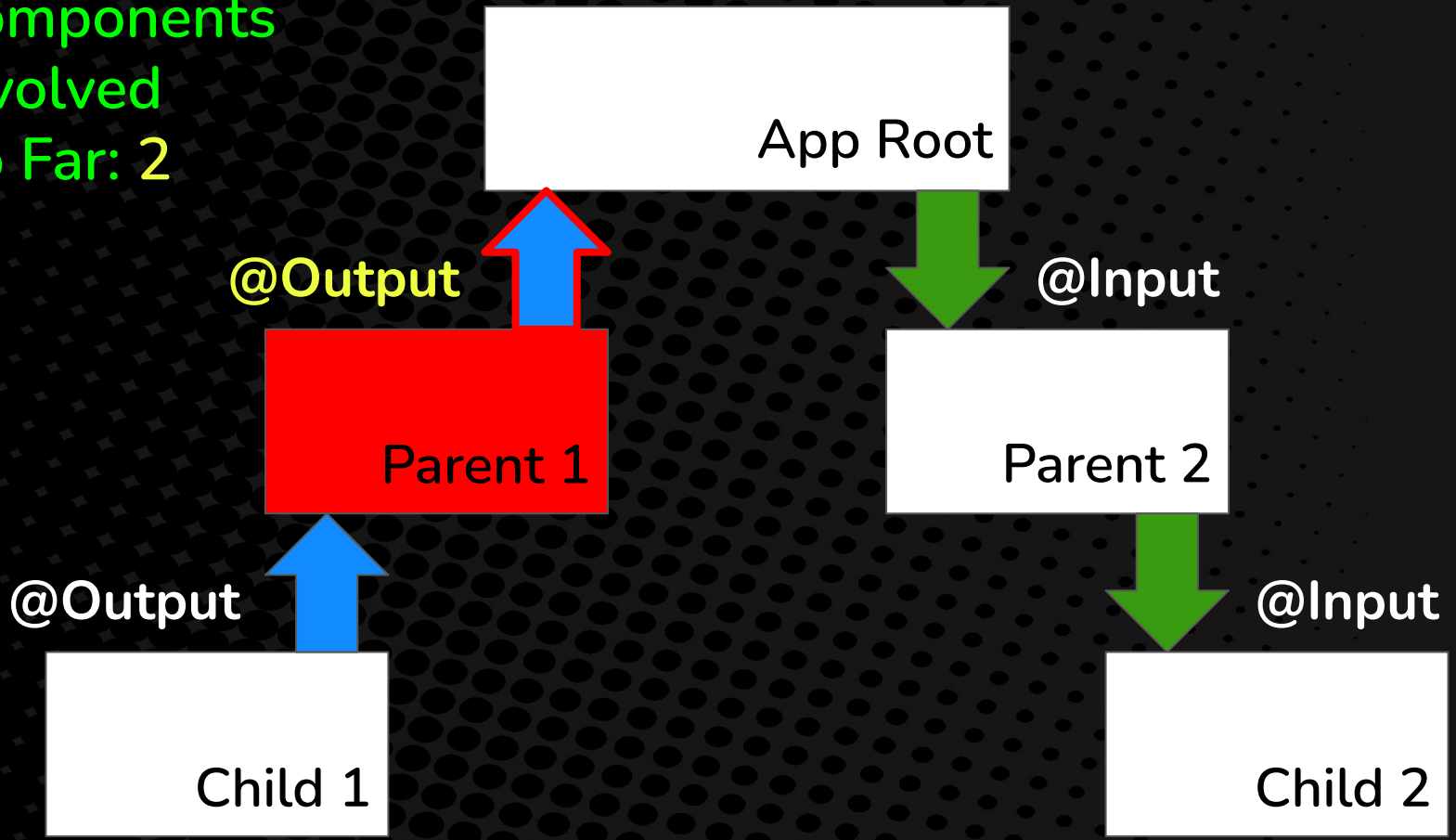
Let’s look at the data flow
selecting a product using Outputs


App Root
@Output
@Input
Parent 1
Parent 2
@Output
@Input
Child 1
Child 2

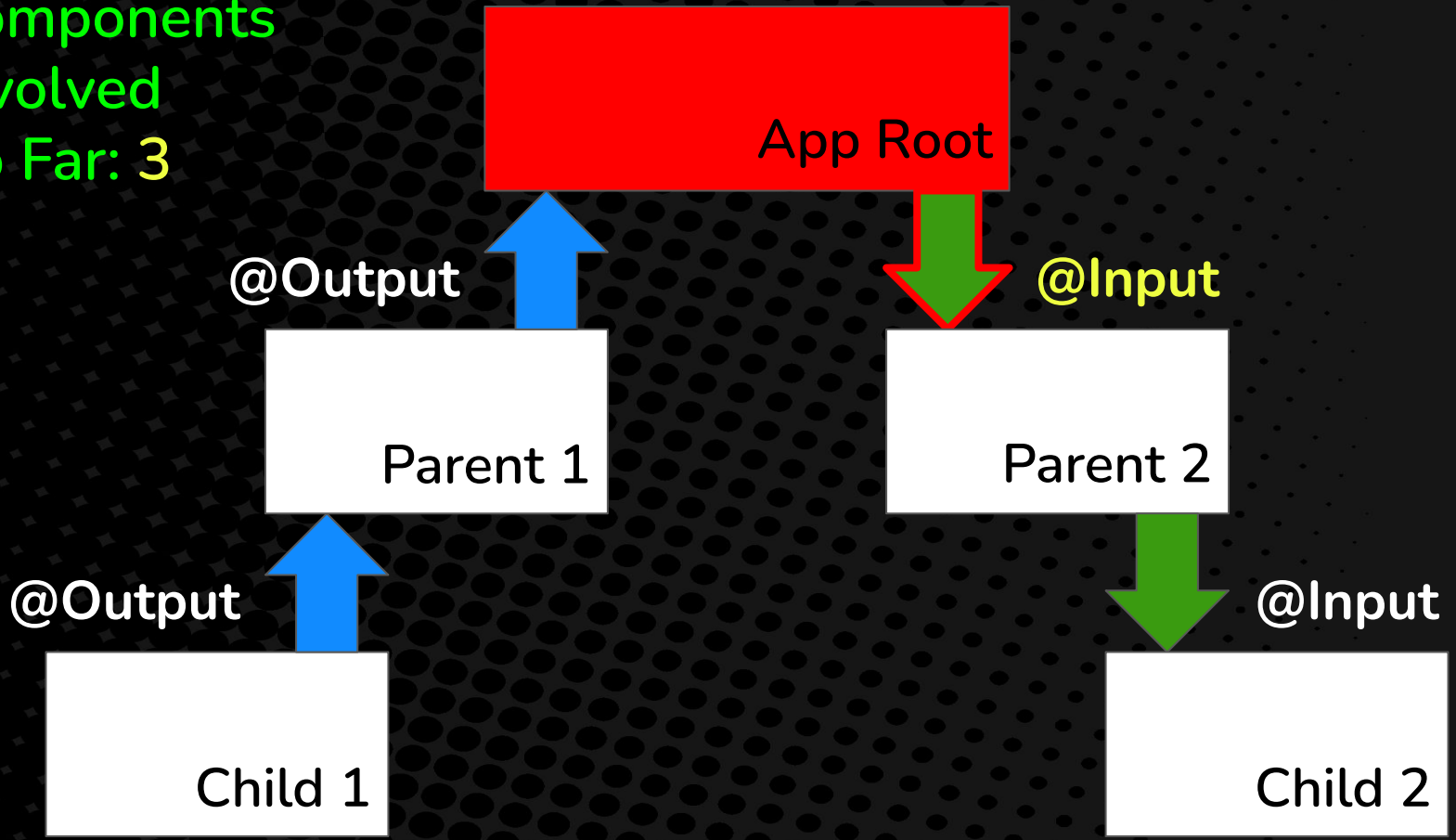
Components Involved So Far: 1



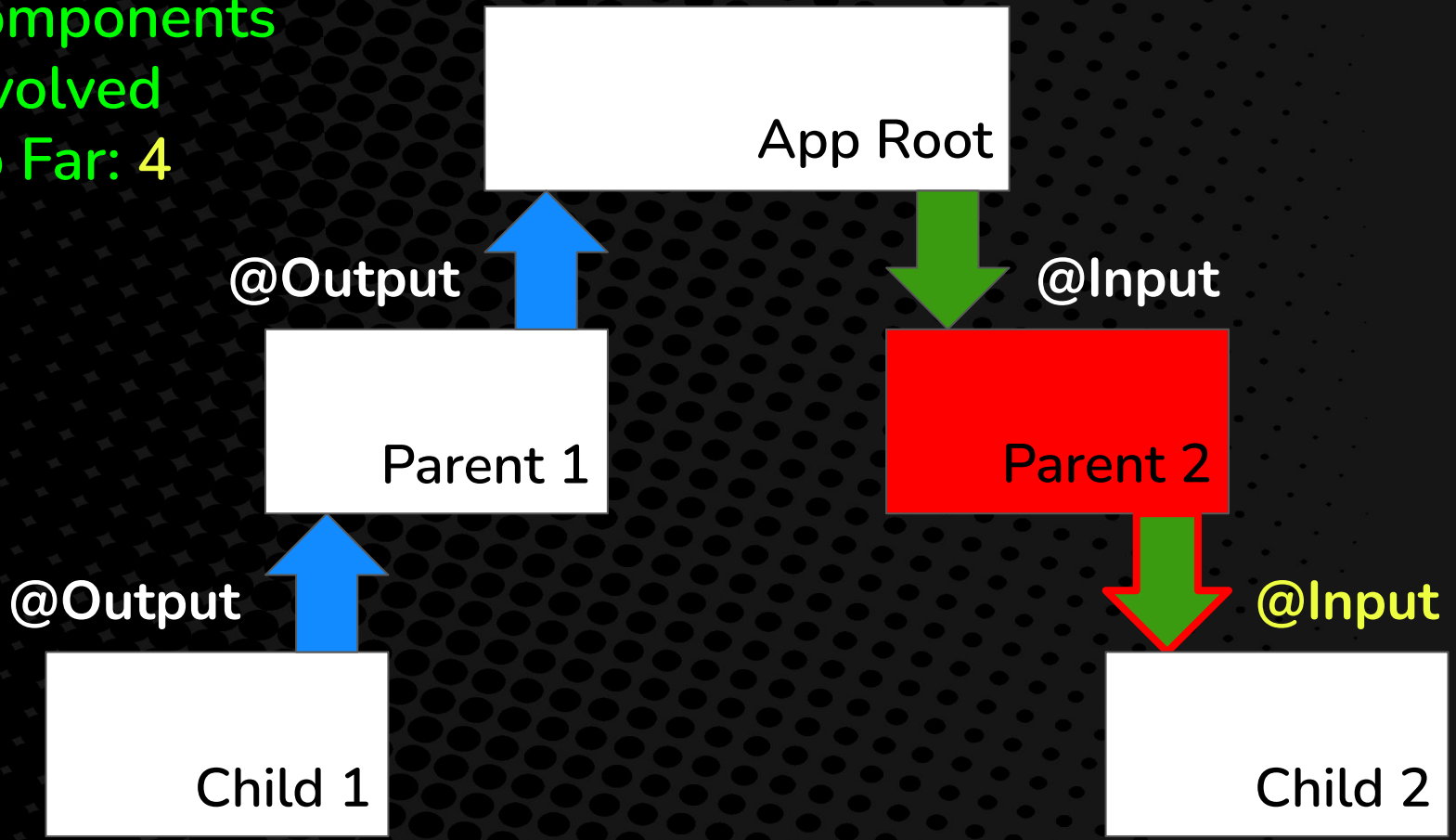
Components Involved So Far: 2



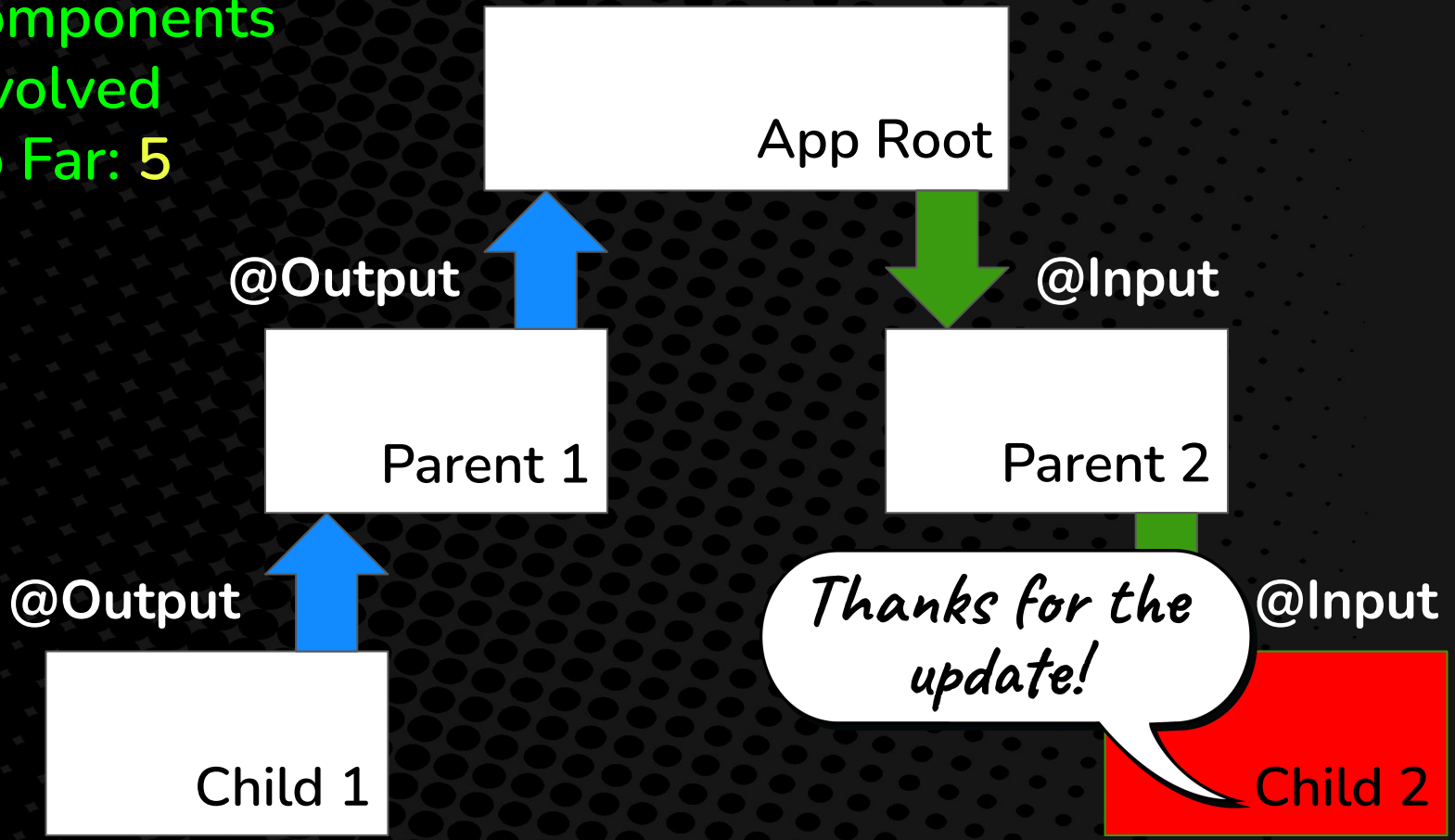
Components Involved So Far: 3



Components Involved So Far: 4

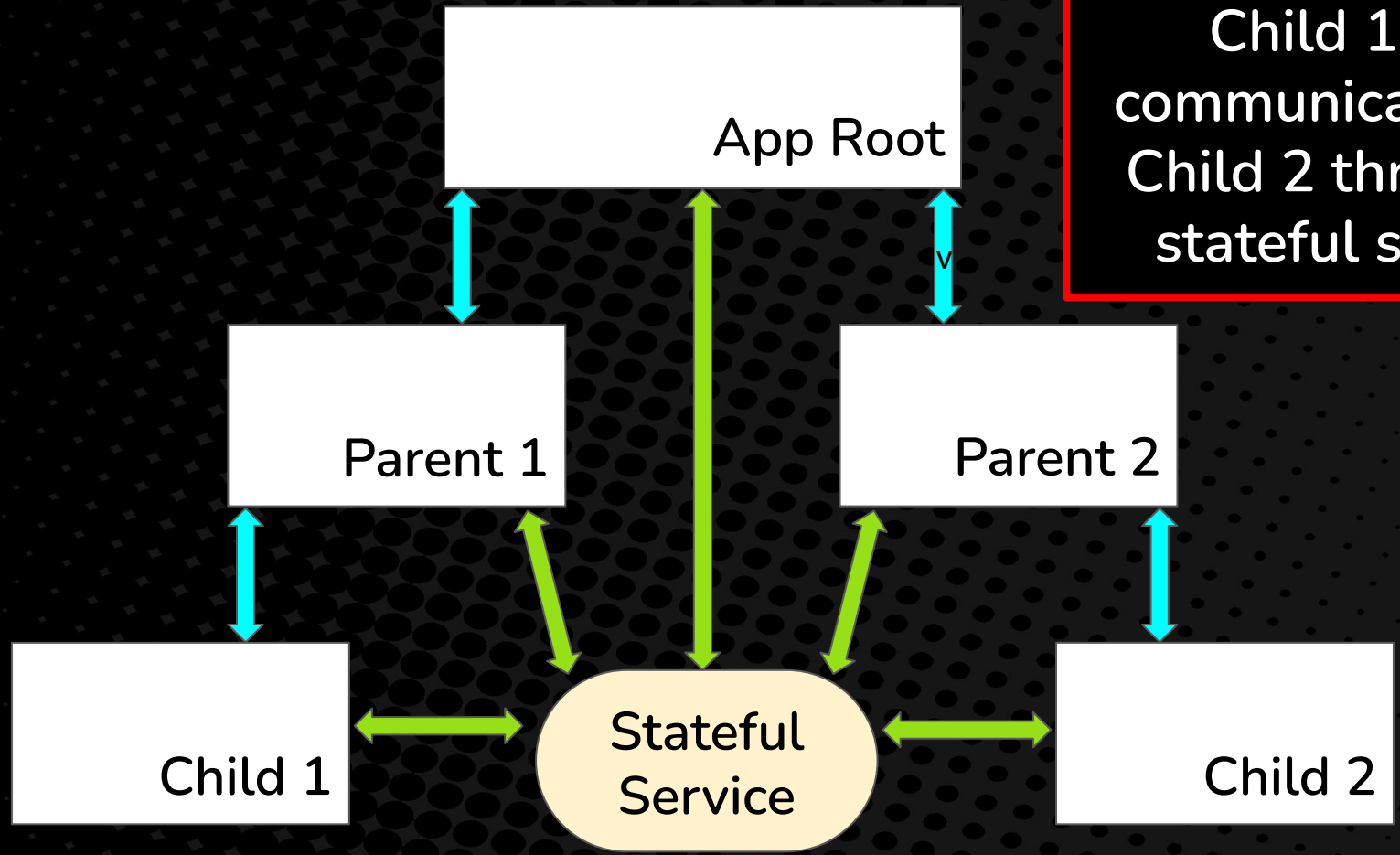


Components Involved So Far: 5

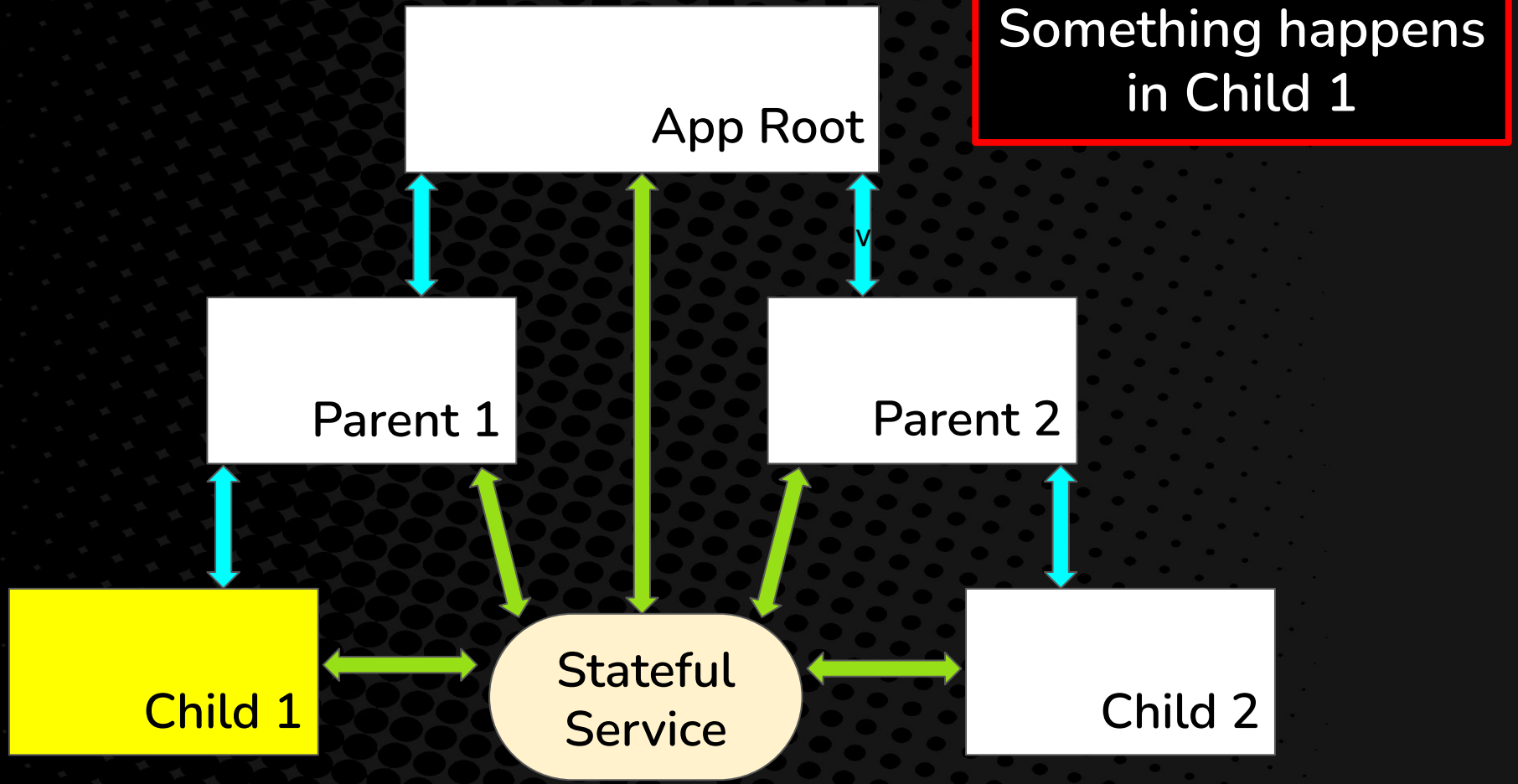


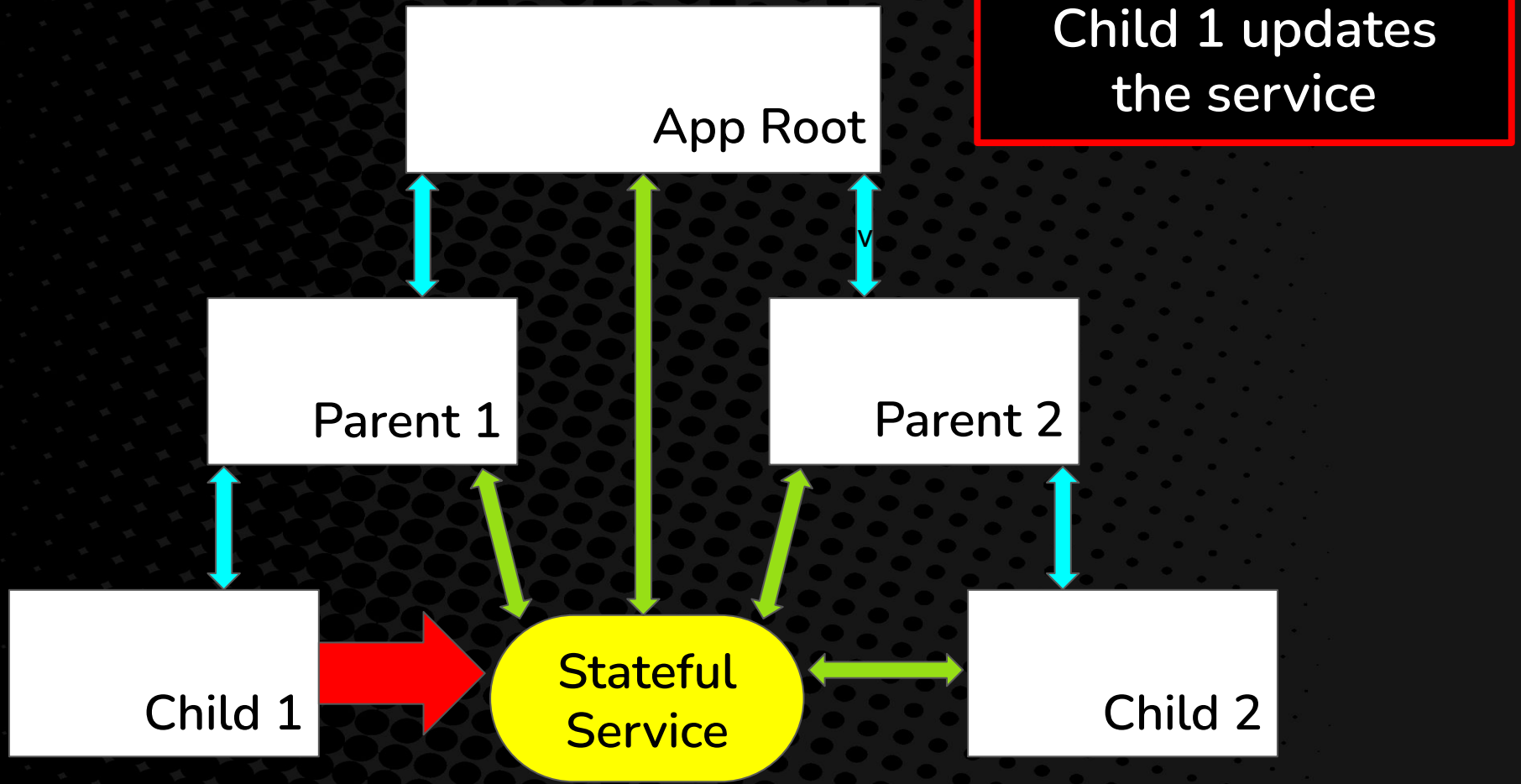
That flow required FIVE components to get the data from the producer to the consumer!

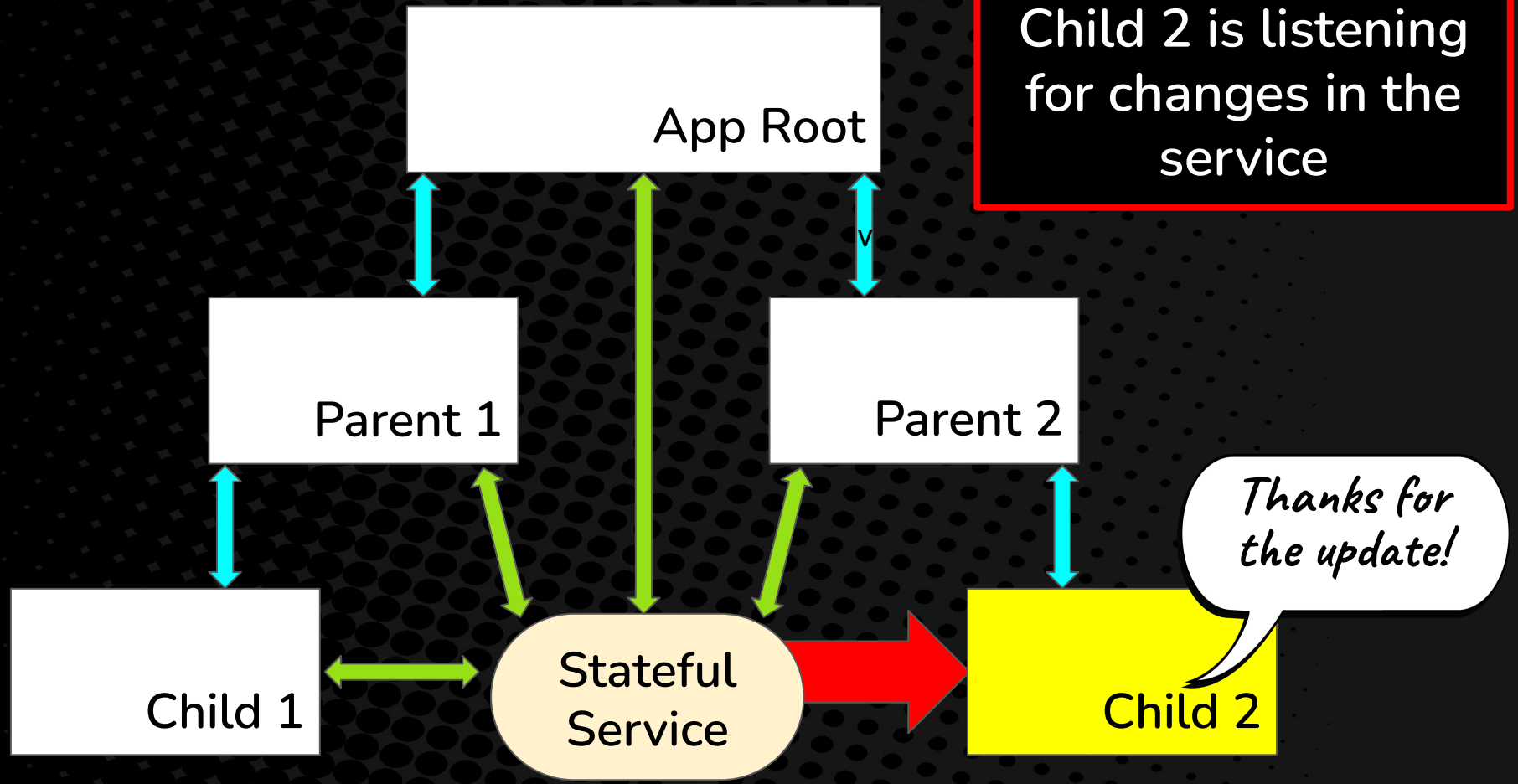
We need a stateful service that can be accessed from anywhere in the app



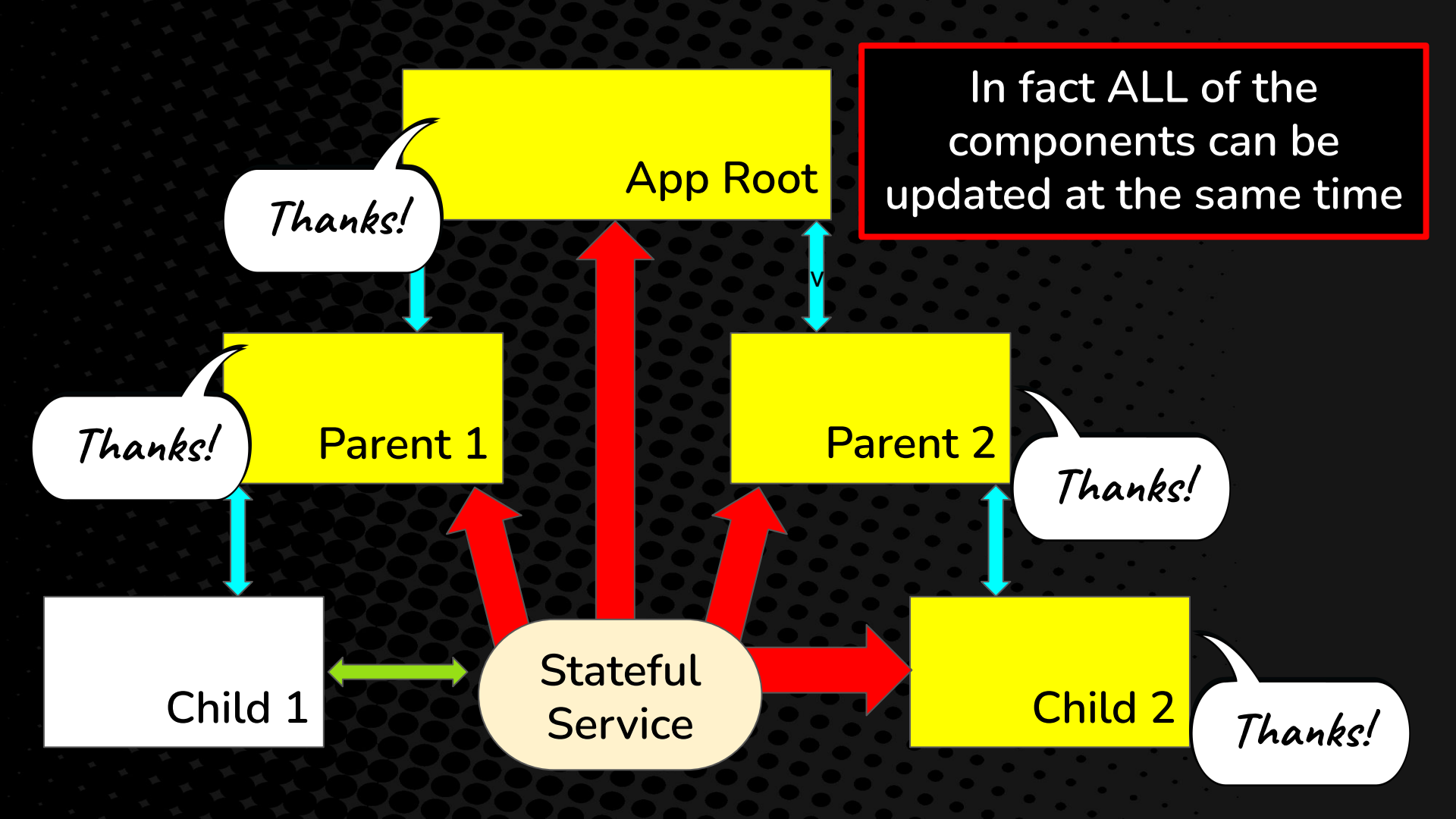
Child 1 can communicate with Child 2 through a stateful service


Something happens in Child 1
App Root
Parent 1
Parent 2
Child 1
Stateful Service
Child 2


Child 1 updates the service
App Root
Parent 1
Parent 2
Child 1
Stateful Service
Child 2

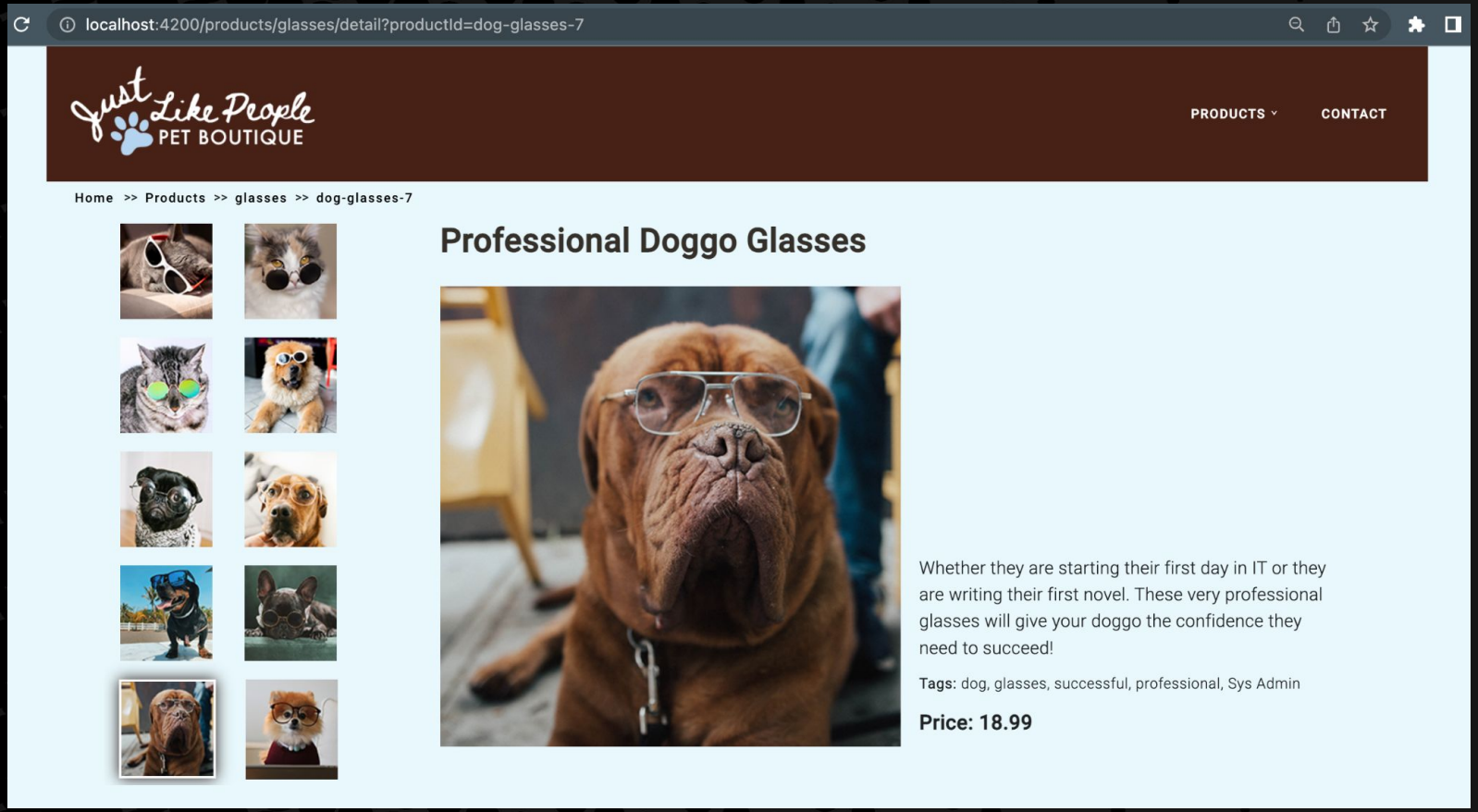


Child 2 is listening for changes in the service


In fact ALL of the components can be updated at the same time
App Root
Thanks!
Parent 1
Thanks!
Parent 2
Thanks!
Child 1
Stateful Service
Child 2
Thanks!

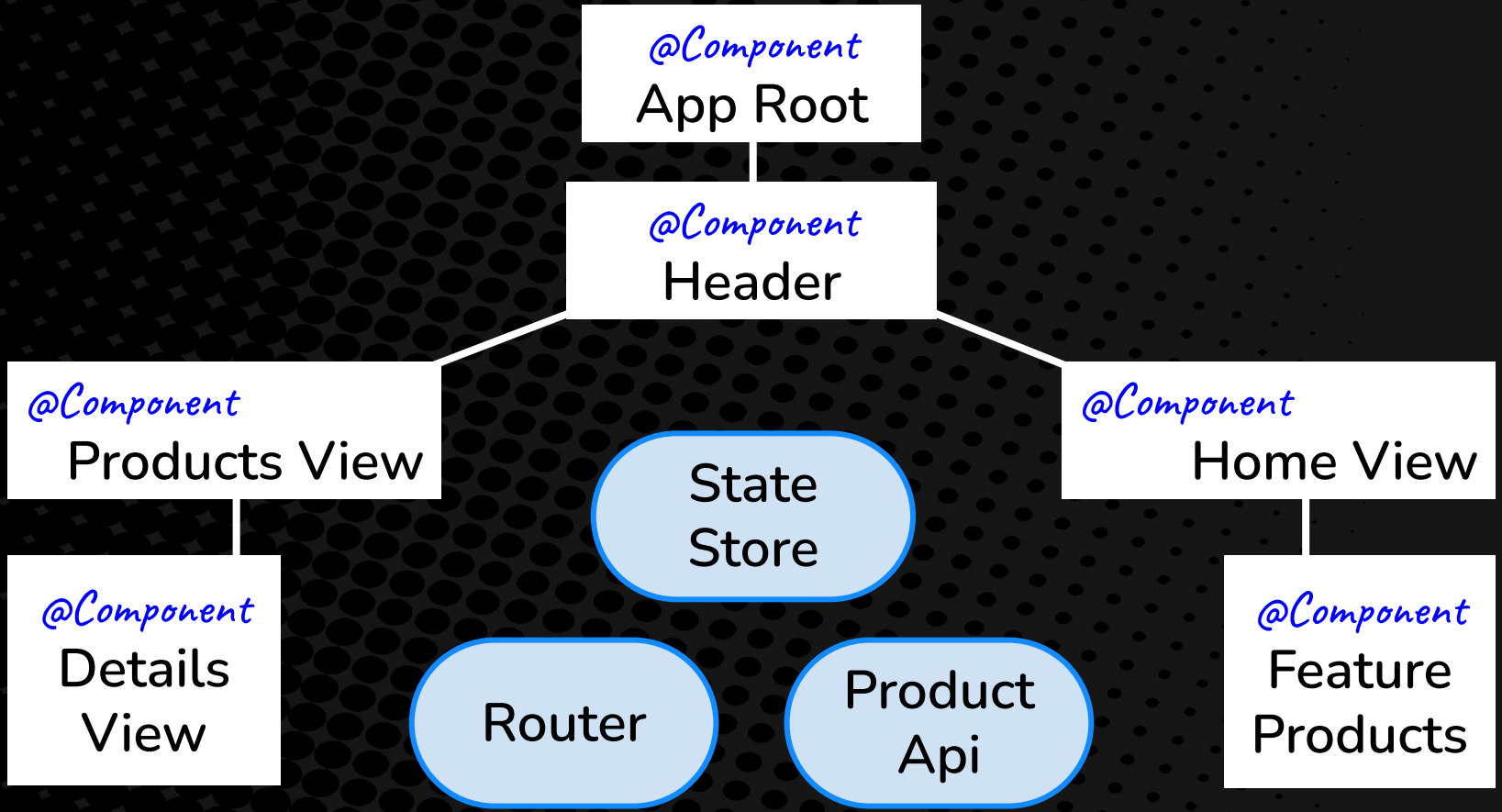
Built-in Stateful Options

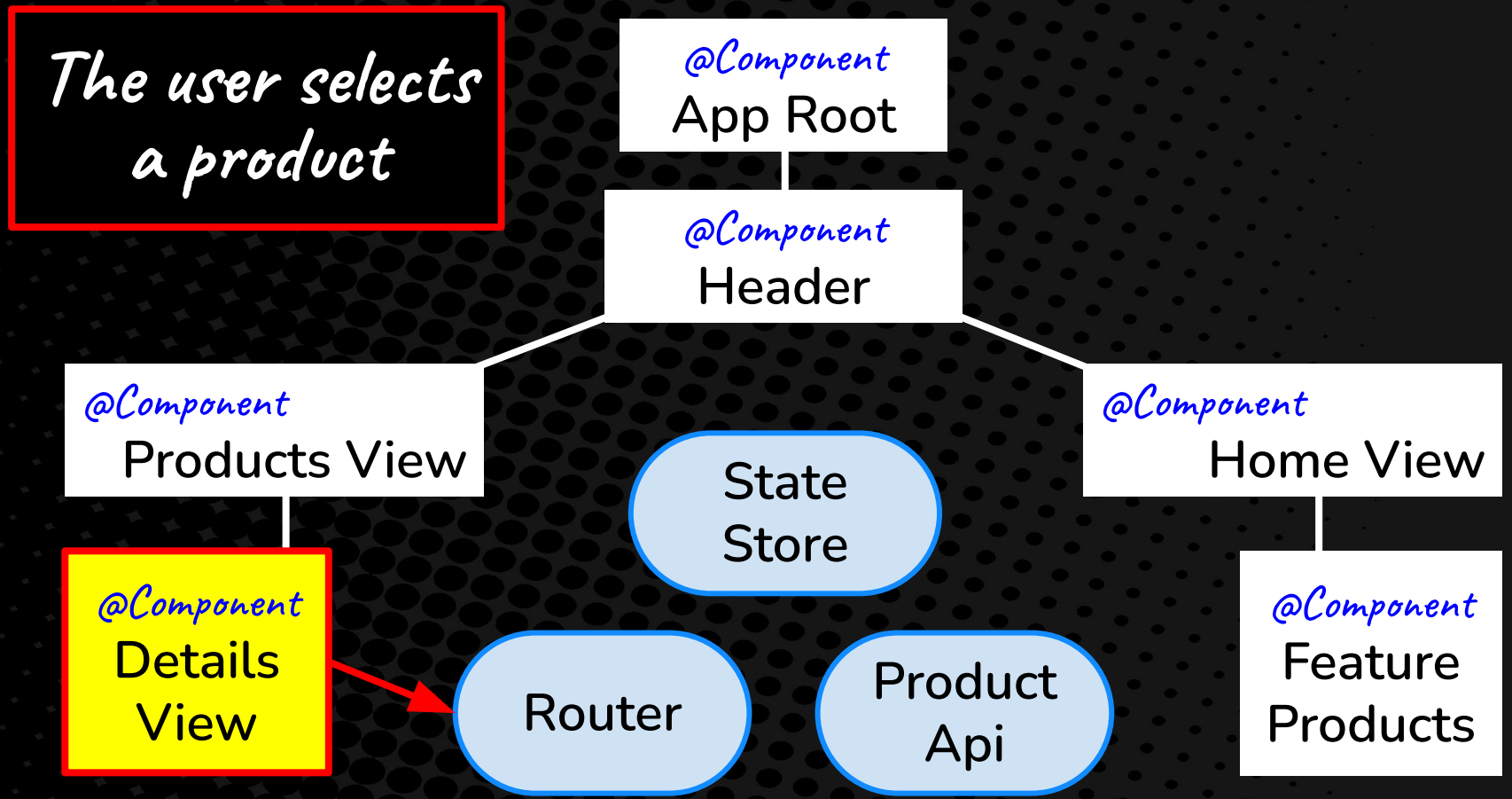
- Service with a subject
- Service with a Signal
- Router Module



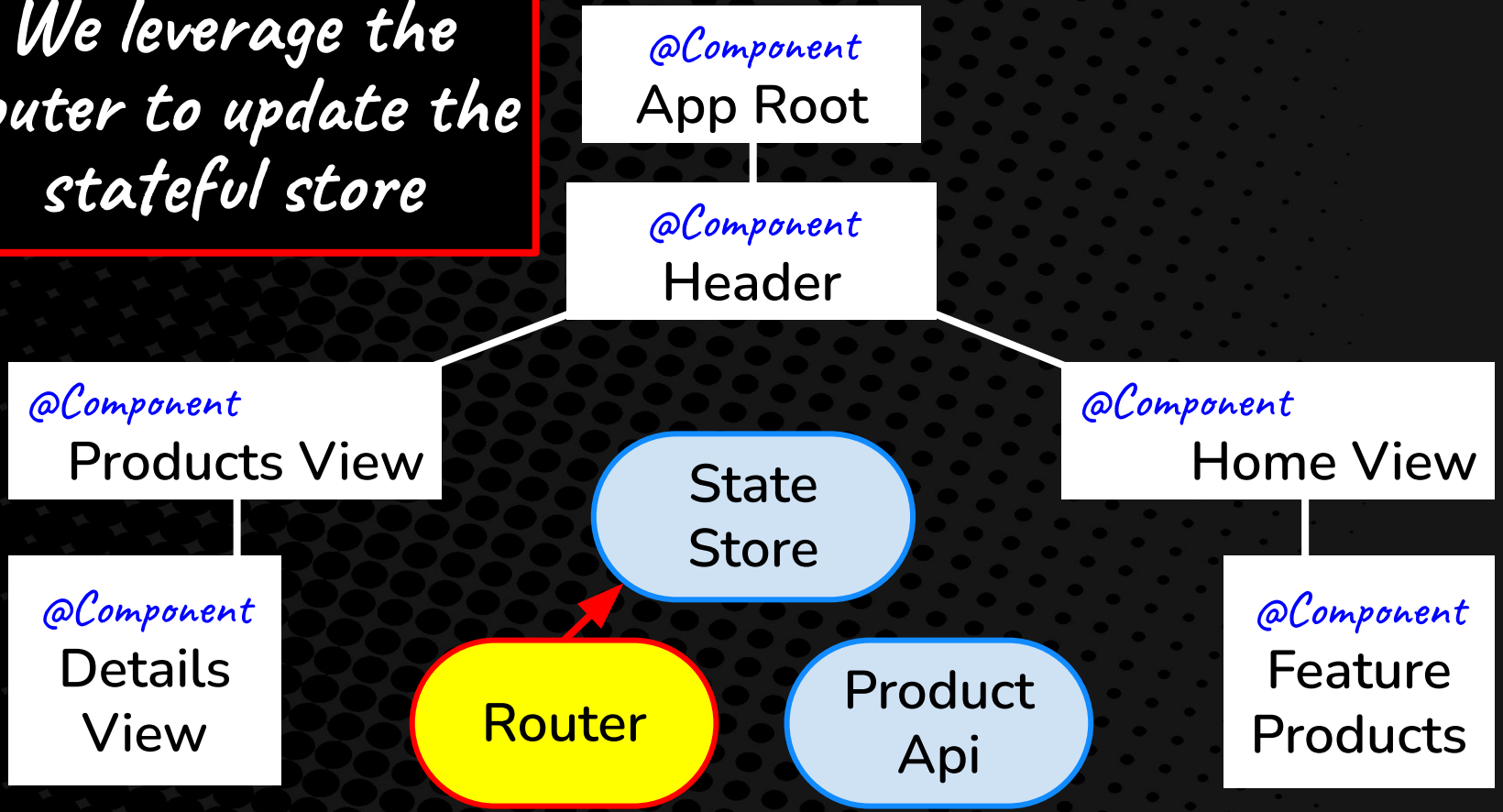
Users want to be able to share links to products

For shareable user selections, we can leverage the Angular Router

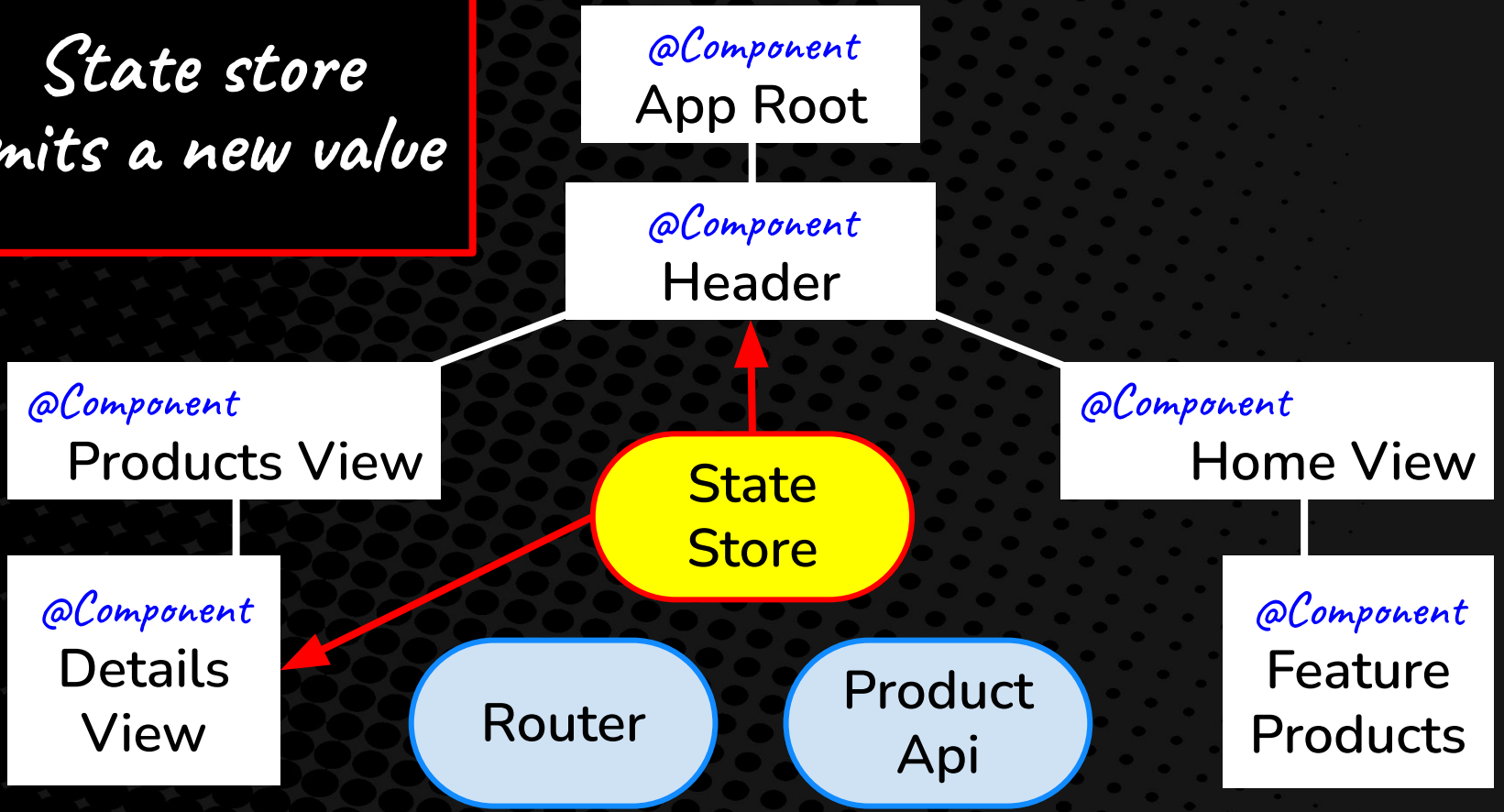

@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State
Store
@Component
Details
View
Router
Product
Api
@Component
Feature
Products

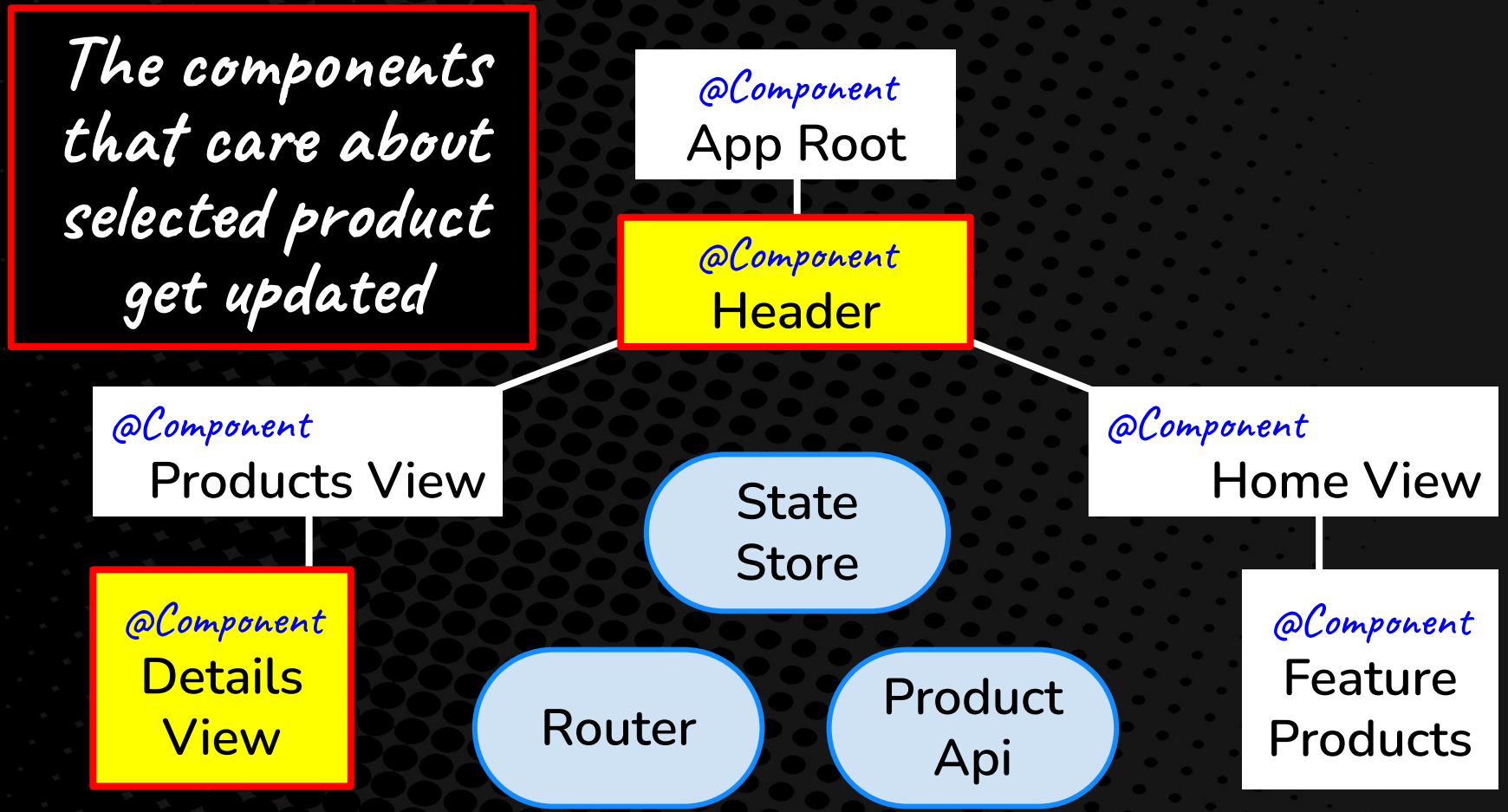

The user selects a product
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

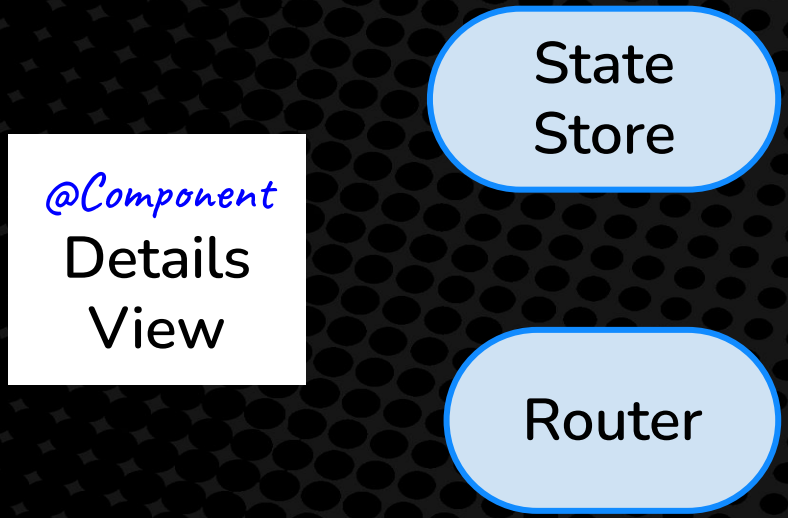
We leverage the router to update the stateful store



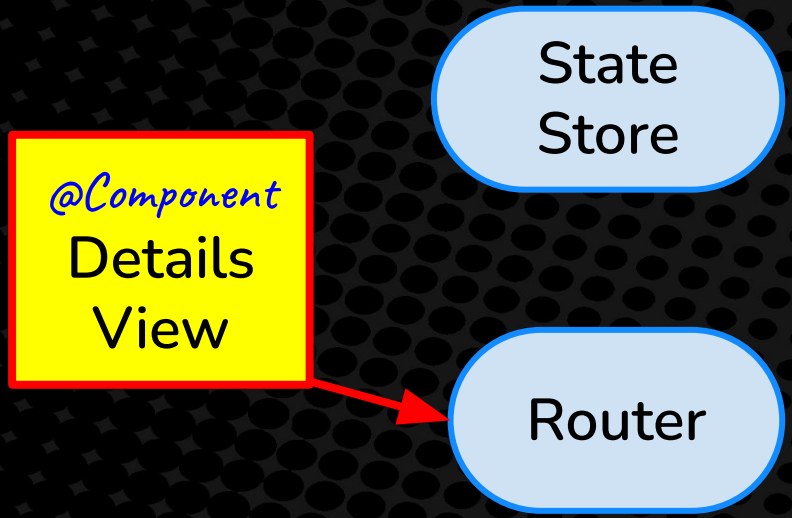
State store emits a new value



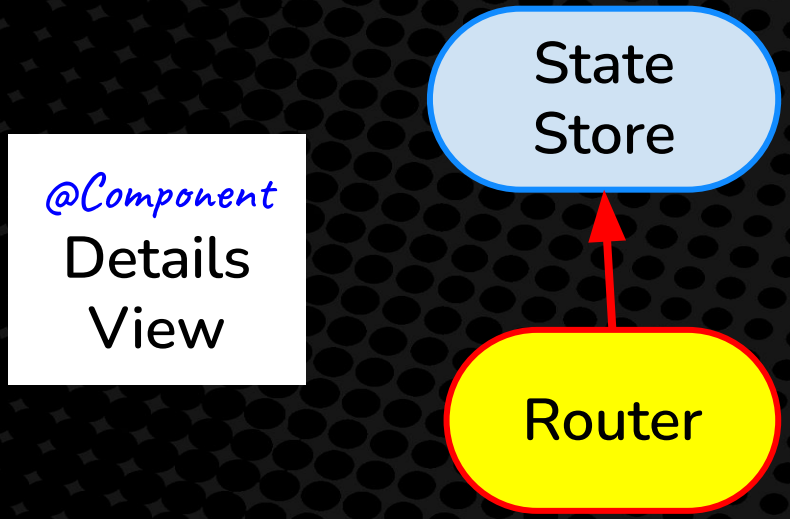

The components that care about selected product get updated
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products



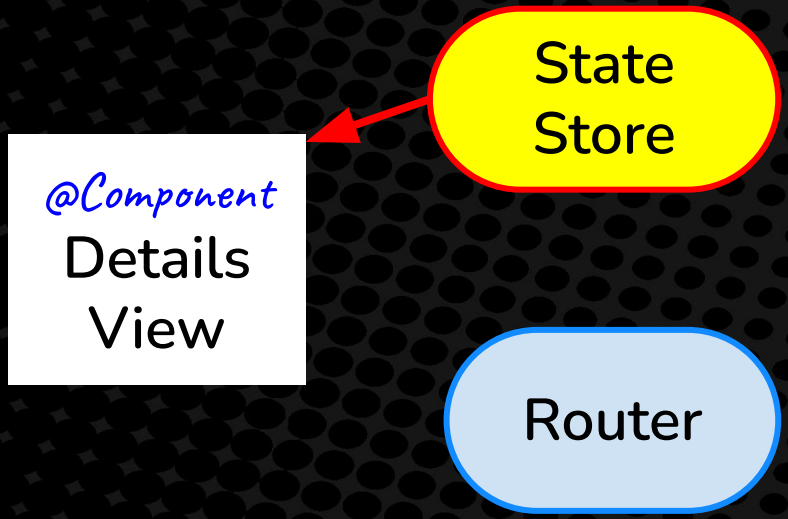
Notice that even within a single component we are still driving state the same way



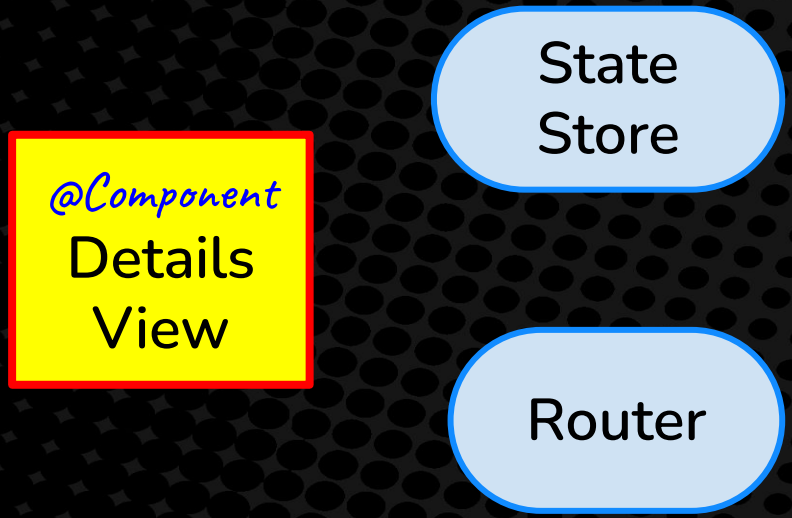
Notice that even within a single component we are still driving state the same way



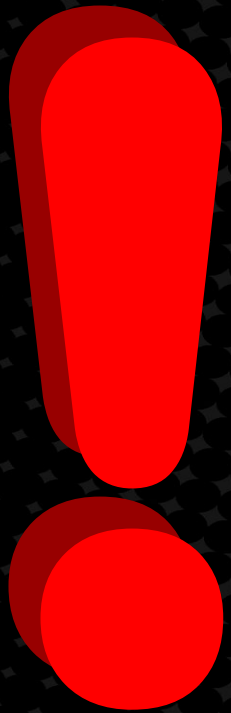
Notice that even within a single component we are still driving state the same way



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



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



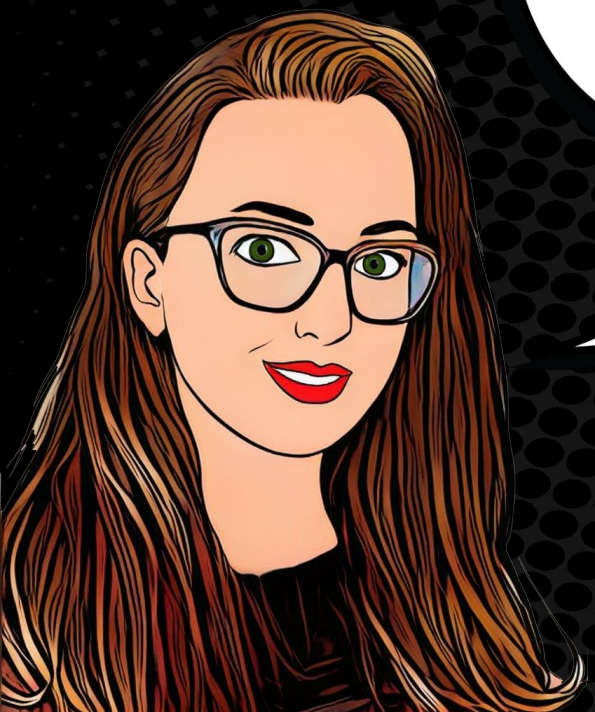
Be careful to maintain a single flow to set state

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

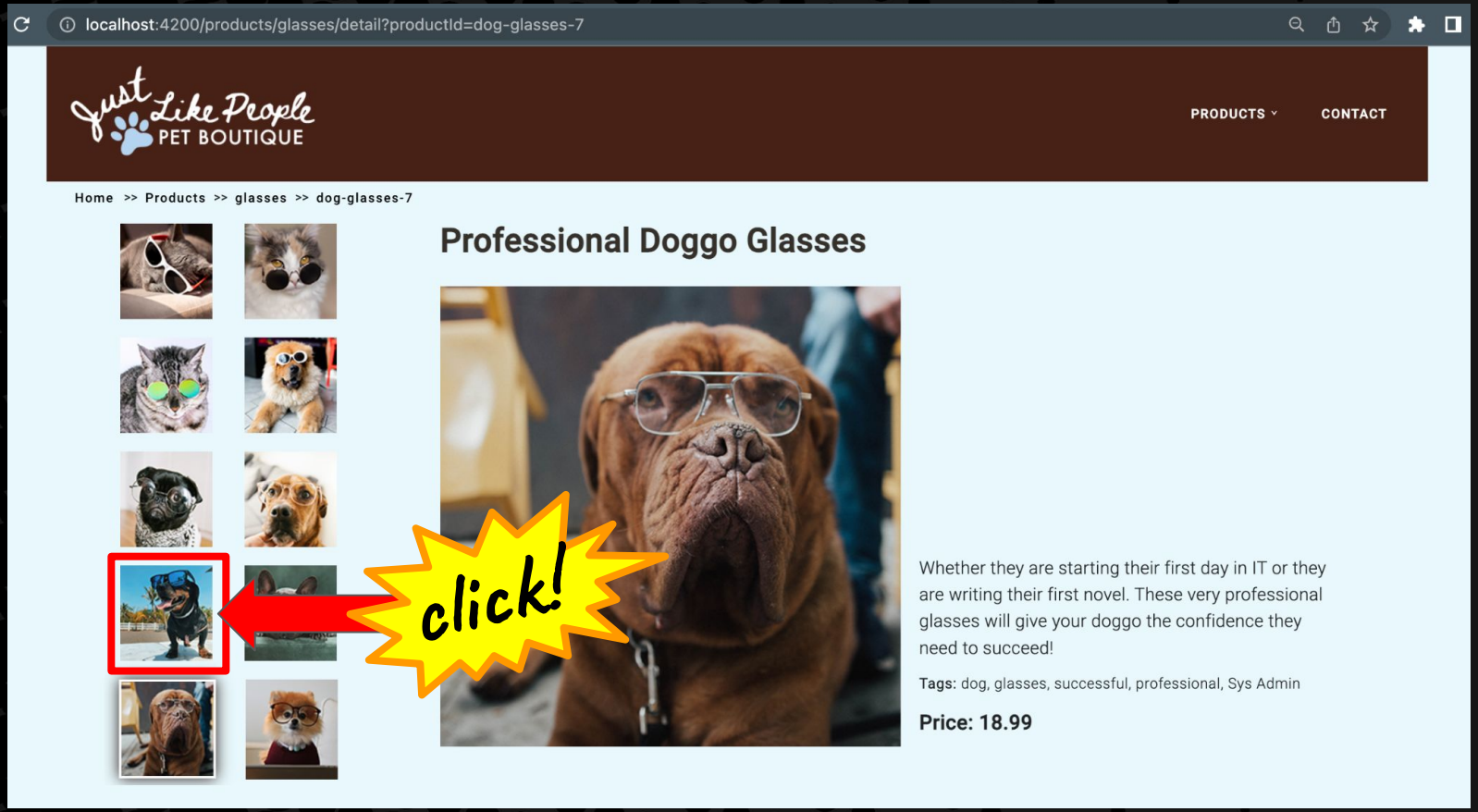
We want to avoid mixing responsibilities

```
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.



Let’s combine the first pattern with the second pattern



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

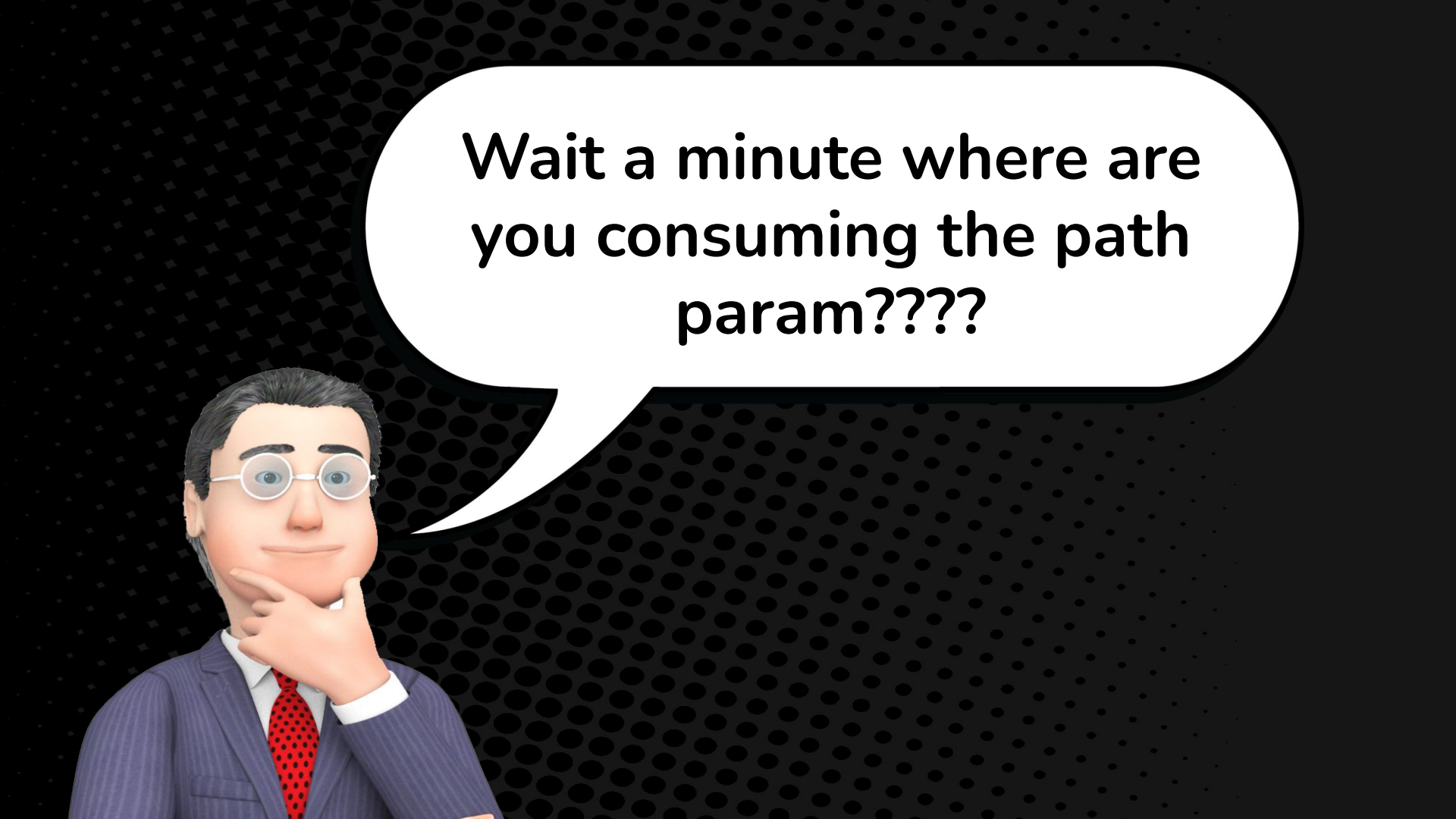
```
<button
  *ngFor="let product of products$ | async"
  class="image-button"
  [routerLink]="['/products', product.category, 'detail']"
  [queryParams]="{productId: product.id}"
  routerLinkActive="router-link-active"
>
```

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.


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

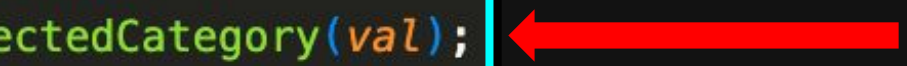
Can be accessed from anywhere so be cautious about duplicating query param state

Let’s step through how
this works in our component

Inside the Component

The categoryId path param changes

```
export class ProductViewComponent {
  @Input() set categoryId(val: string) {
    this.productsService.setSelectedCategory(val);
  }

  private readonly productsService = inject(ProductService);

  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) {
      }),
    ))
  );
```

Inside the ProductsService

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
          }

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

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

```
export class ProductService {
  priv
  read

  setS
    th
  }
```

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

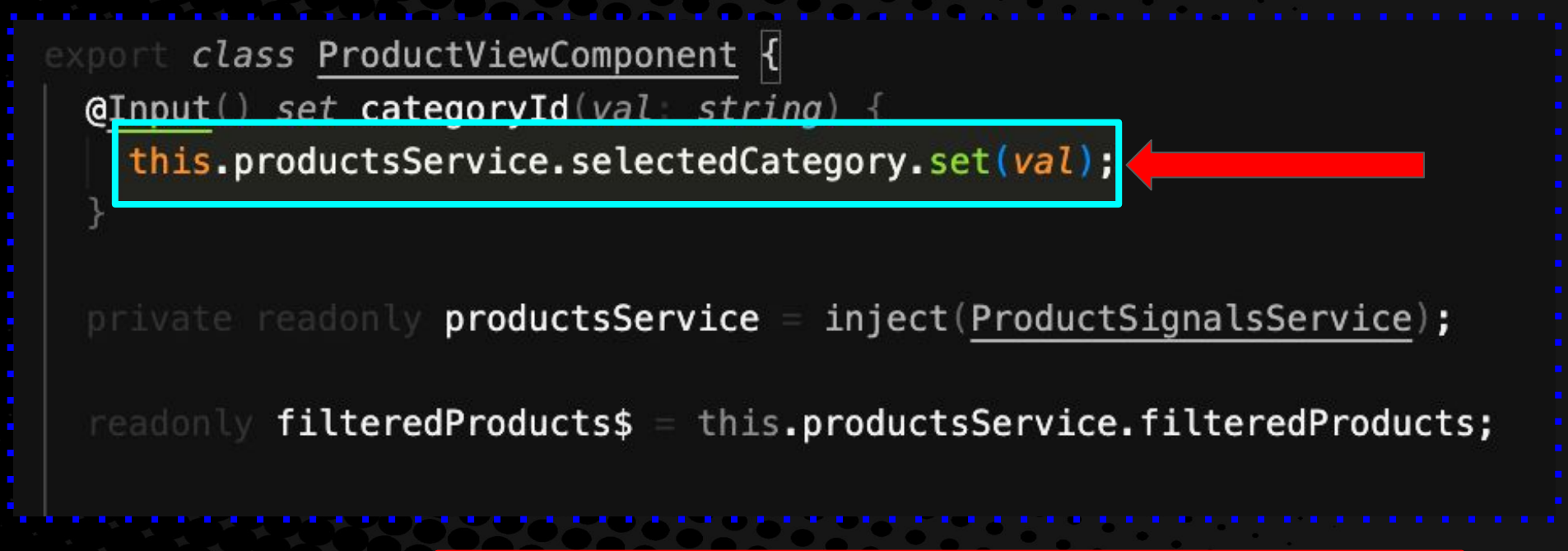
```
readonly filteredProducts$ = this.selectedCategory.pipe(
  switchMap((category) => this.products$.pipe(
    map((products) => {
      if(category === Category.ALL) {
        return products;
      }

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

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



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

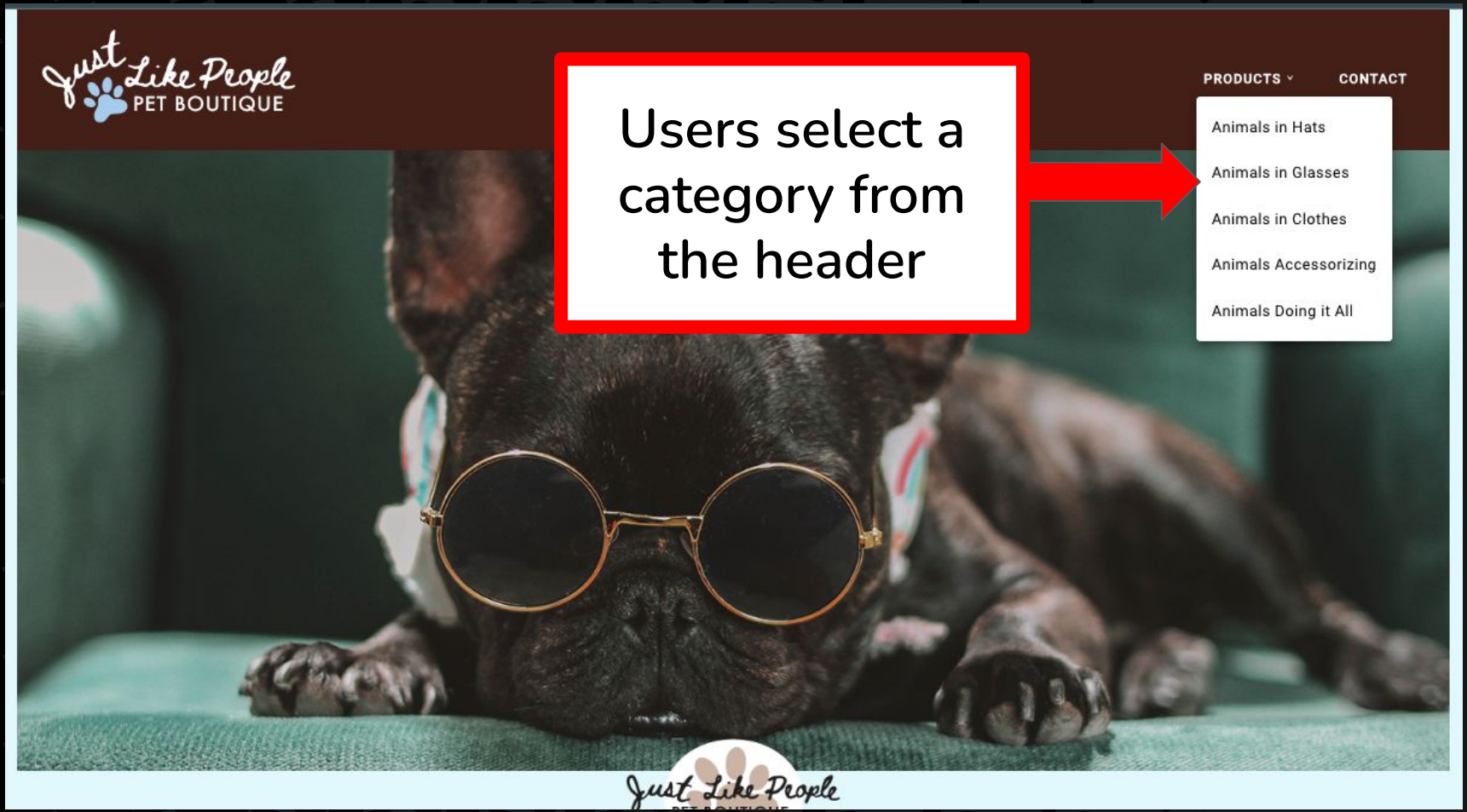
Remember all the consumer wants to do is consume.

Let’s combine our patterns!

Input setters

Emit events
where they happen

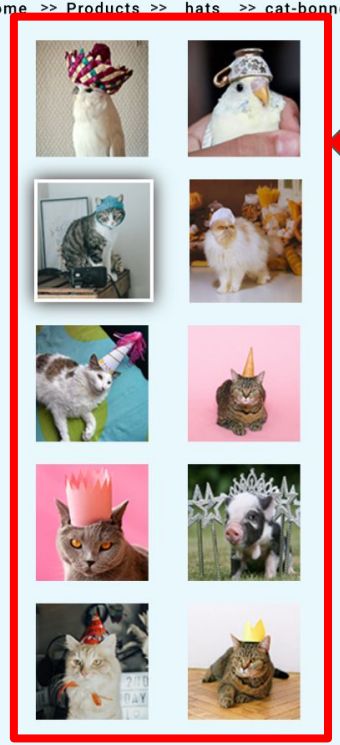
Compose data before it gets
to the consumers


Just Like People
PET BOUTIQUE
Users select a category from the header
PRODUCTS
CONTACT
Animals in Hats
Animals in Glasses
Animals in Clothes
Animals Accessorizing
Animals Doing it All
Just Like People

Just Like People PET BOUTIQUE

PRODUCTS CONTACT

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



Baby Blue Baby Bonnet fo

Products filtered by category populate the side menu

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

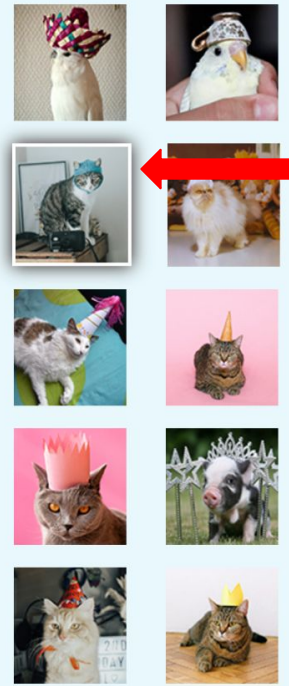
Tags: cat, bonnet, hat, baby, knit

Price: 32.99

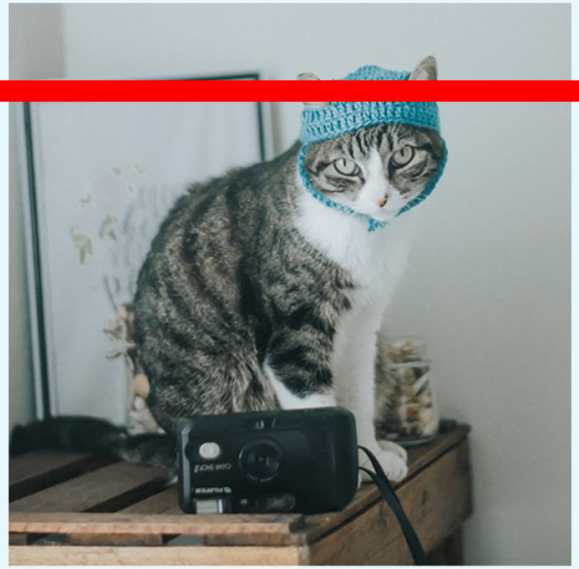
Just Like People PET BOUTIQUE

PRODUCTS CONTACT

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



Baby Blue Baby Bonnet for Big Baby Kitties

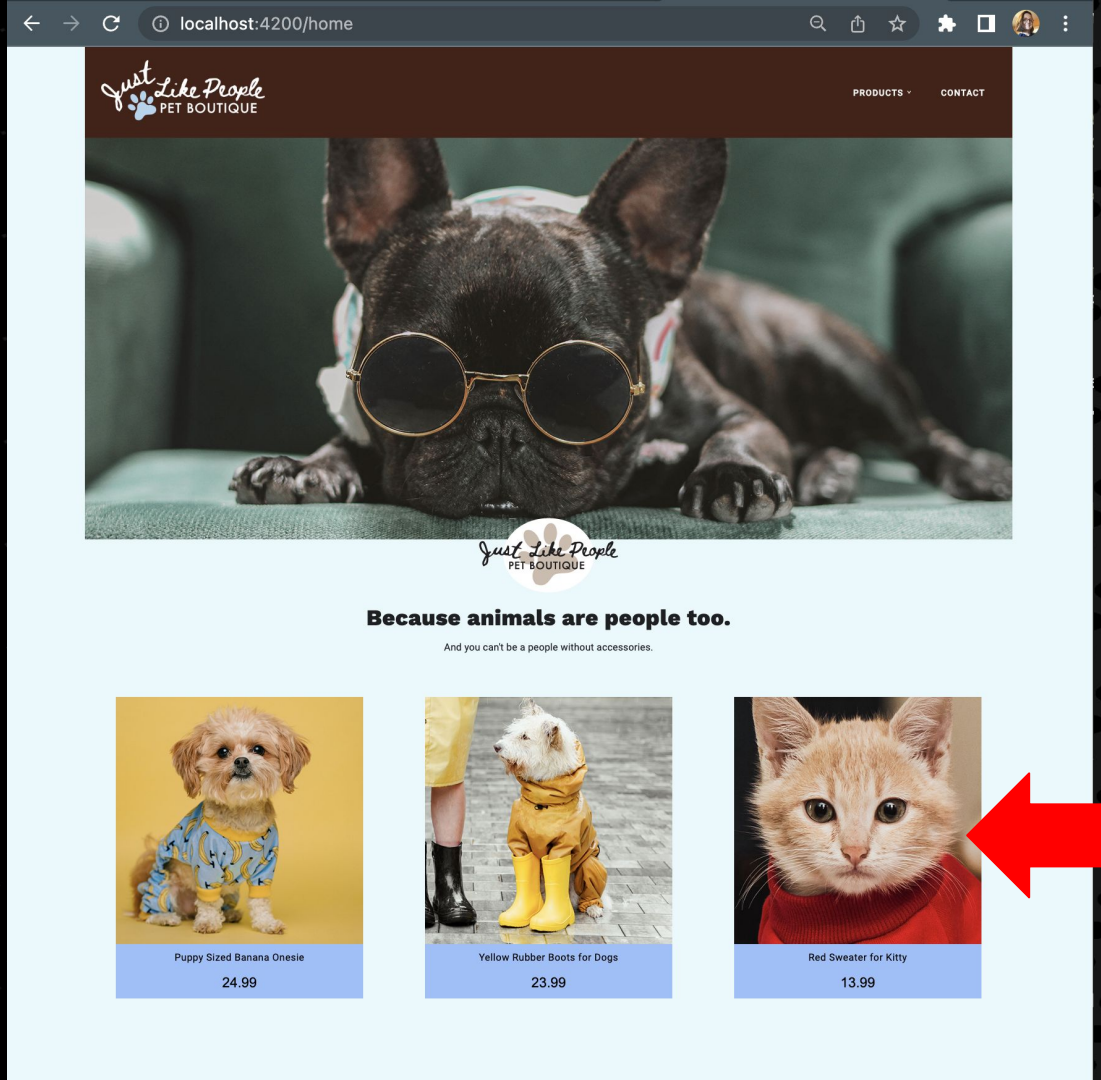


Users can select products from the side menu

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

Tags: cat, bonnet, hat, baby, knit

Price: 32.99



We can also select products from the home page

Just Like People PET BOUTIQUE

PRODUCTS CONTACT

Denim Pooch Pants ON SALE NOW!!!

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

Denim Pooch Pants

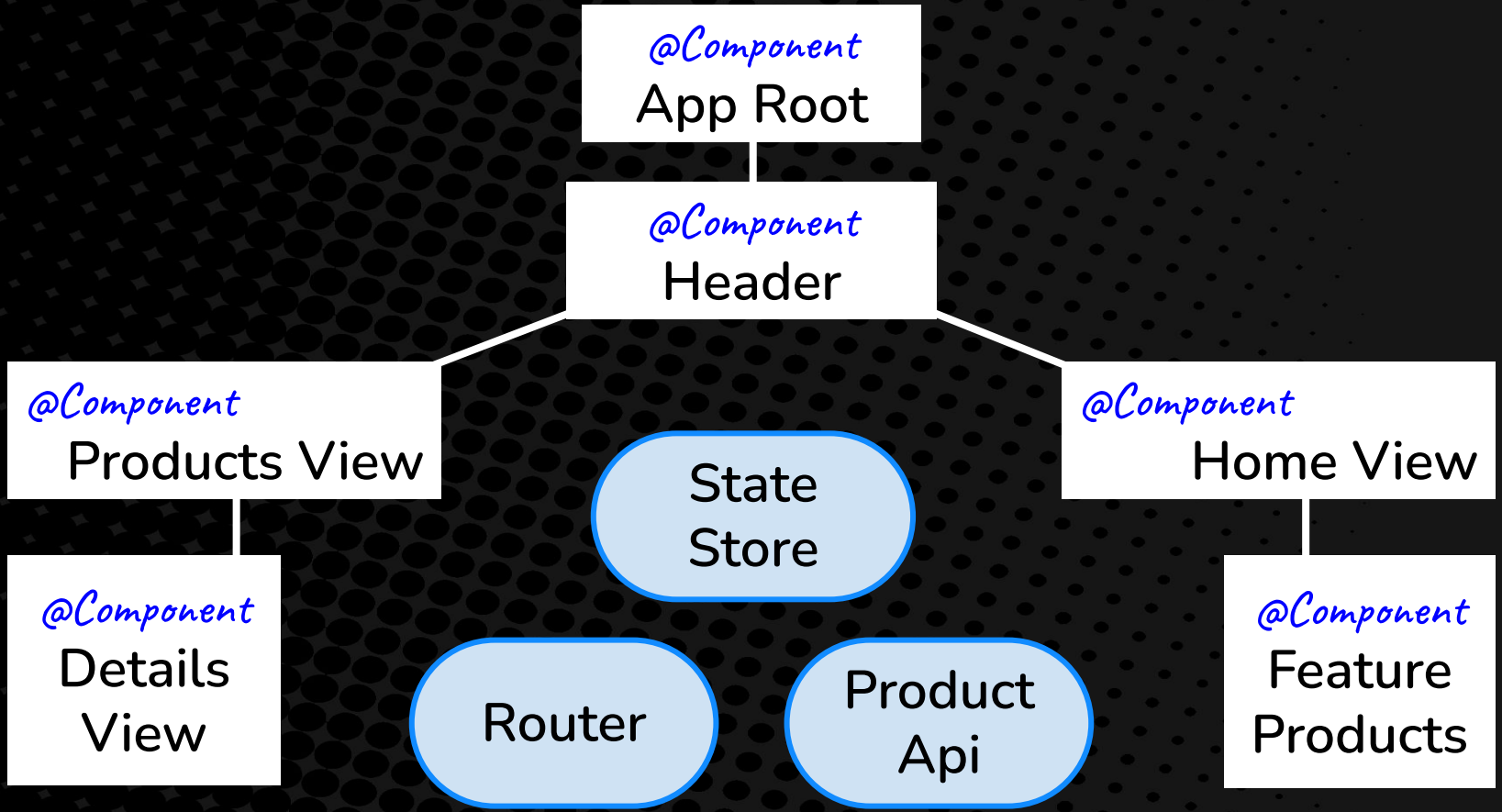
The header and the detail view need the selected product

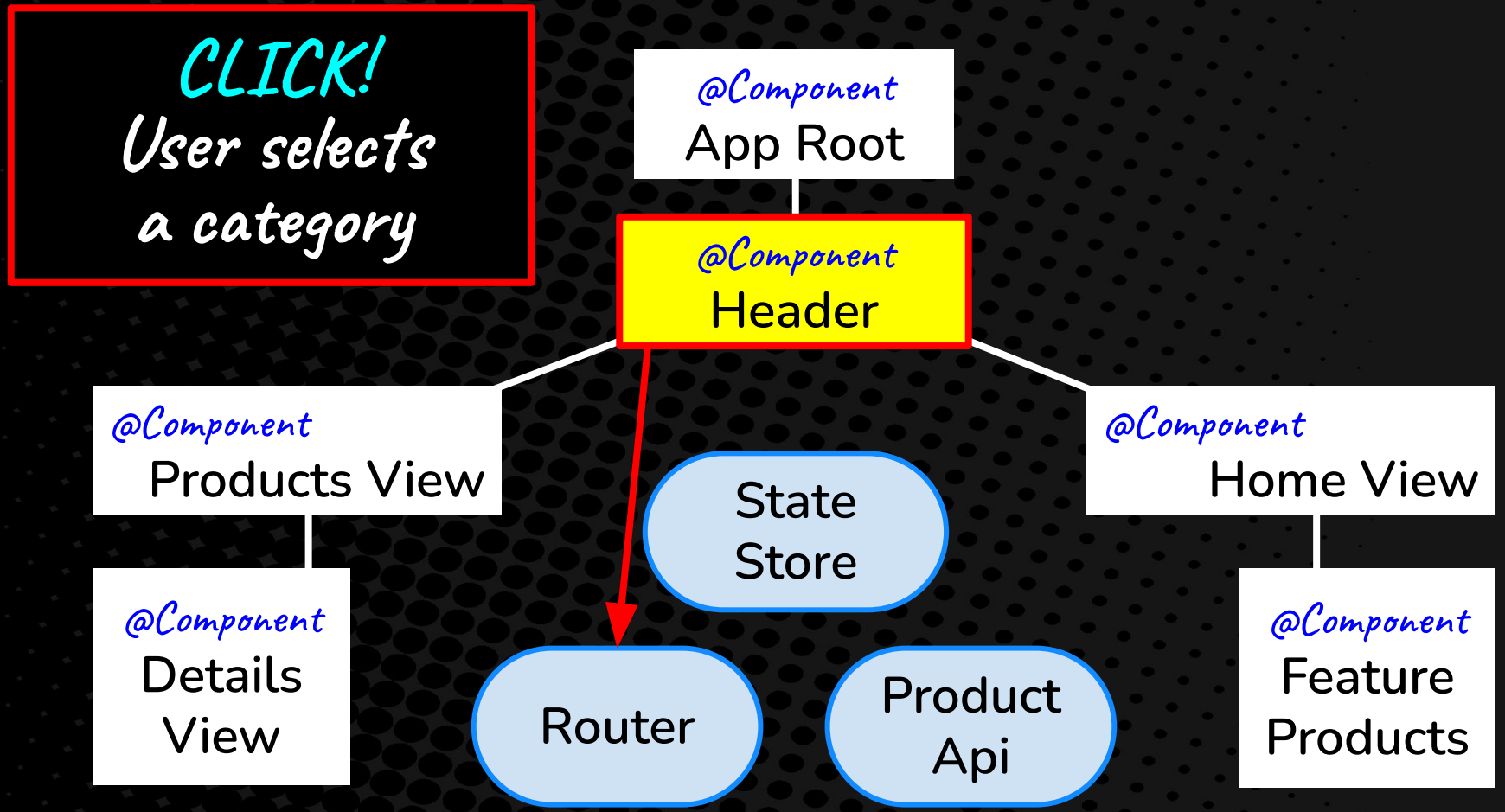
These rugged and durable overalls are great for little poochies who work hard and play harder! Now with 100% more pockets!

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

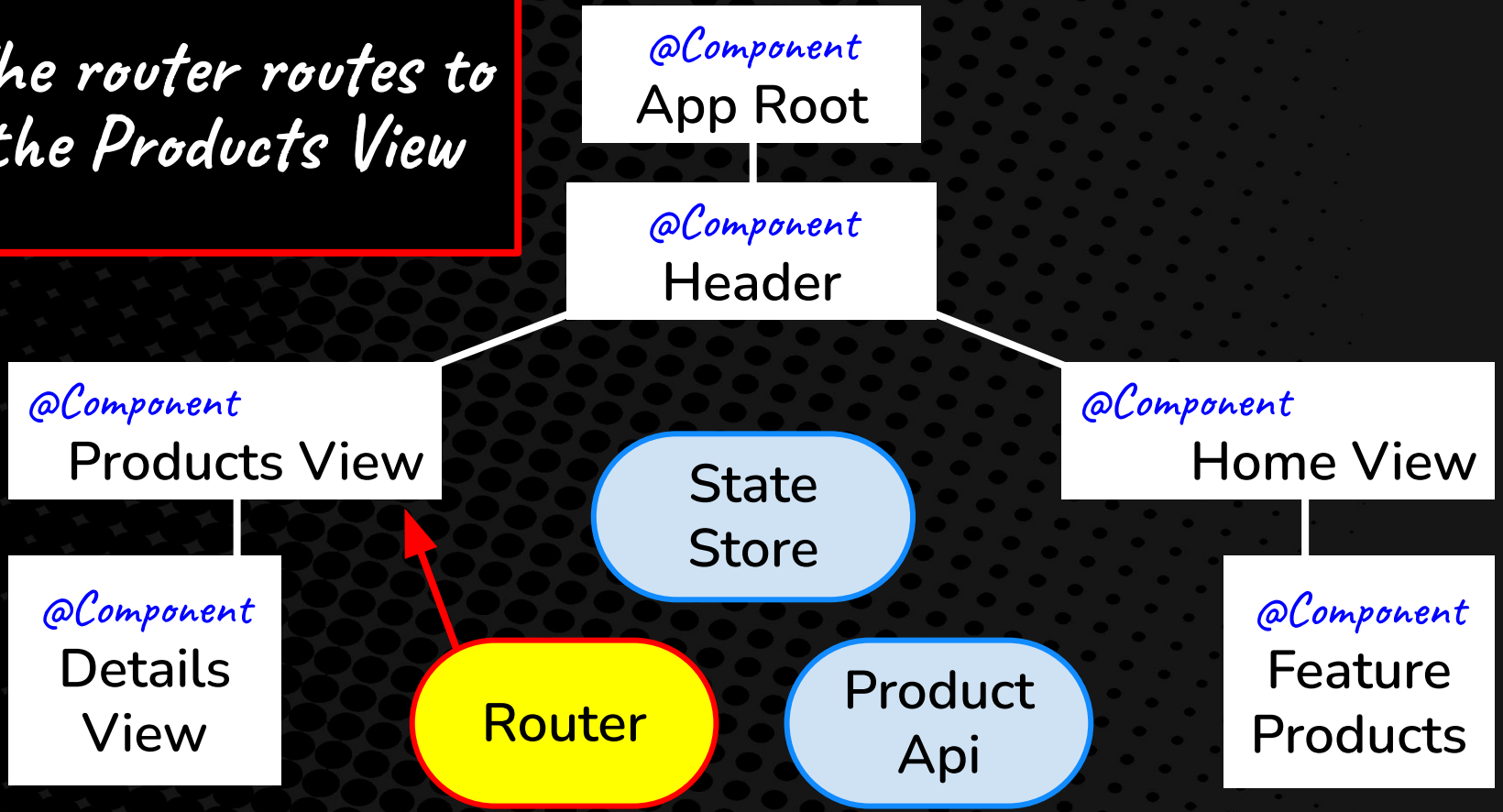
Price: 32.99 ***ON SALE!!***

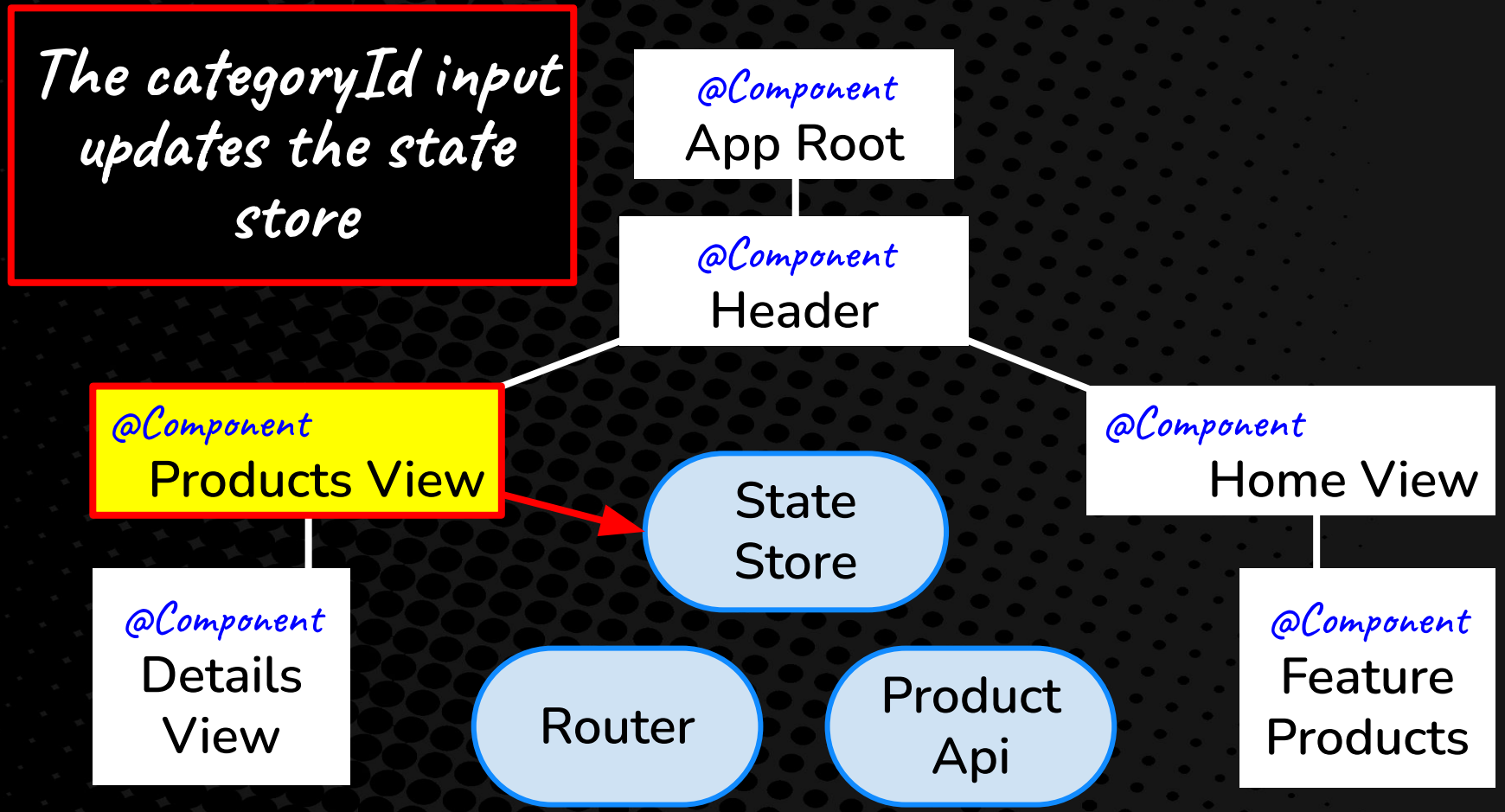
Let's see the state flow for selecting a category


@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State
Store
@Component
Details
View
Router
Product
Api
@Component
Feature
Products

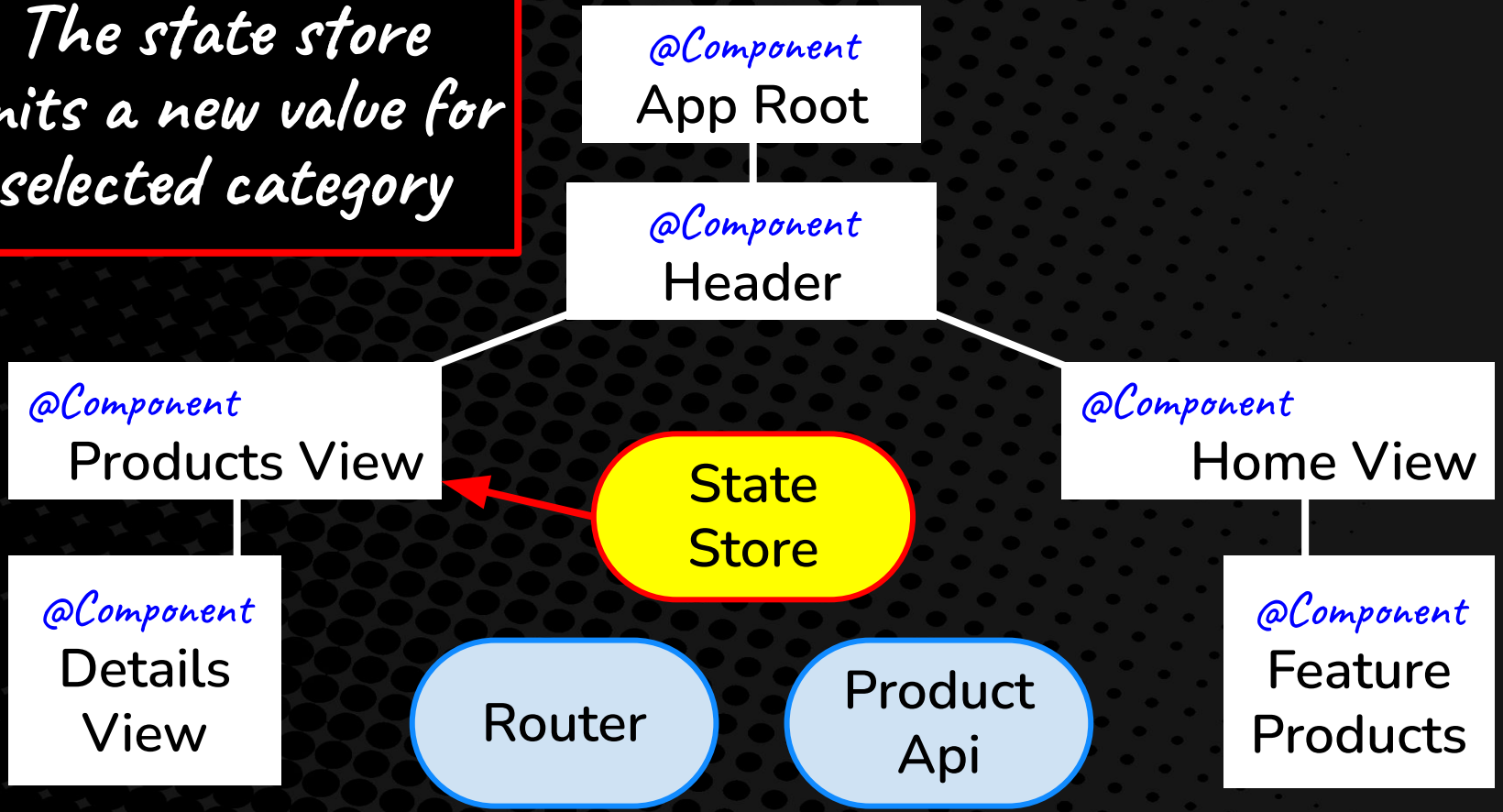

CLICK!
User selects a category
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

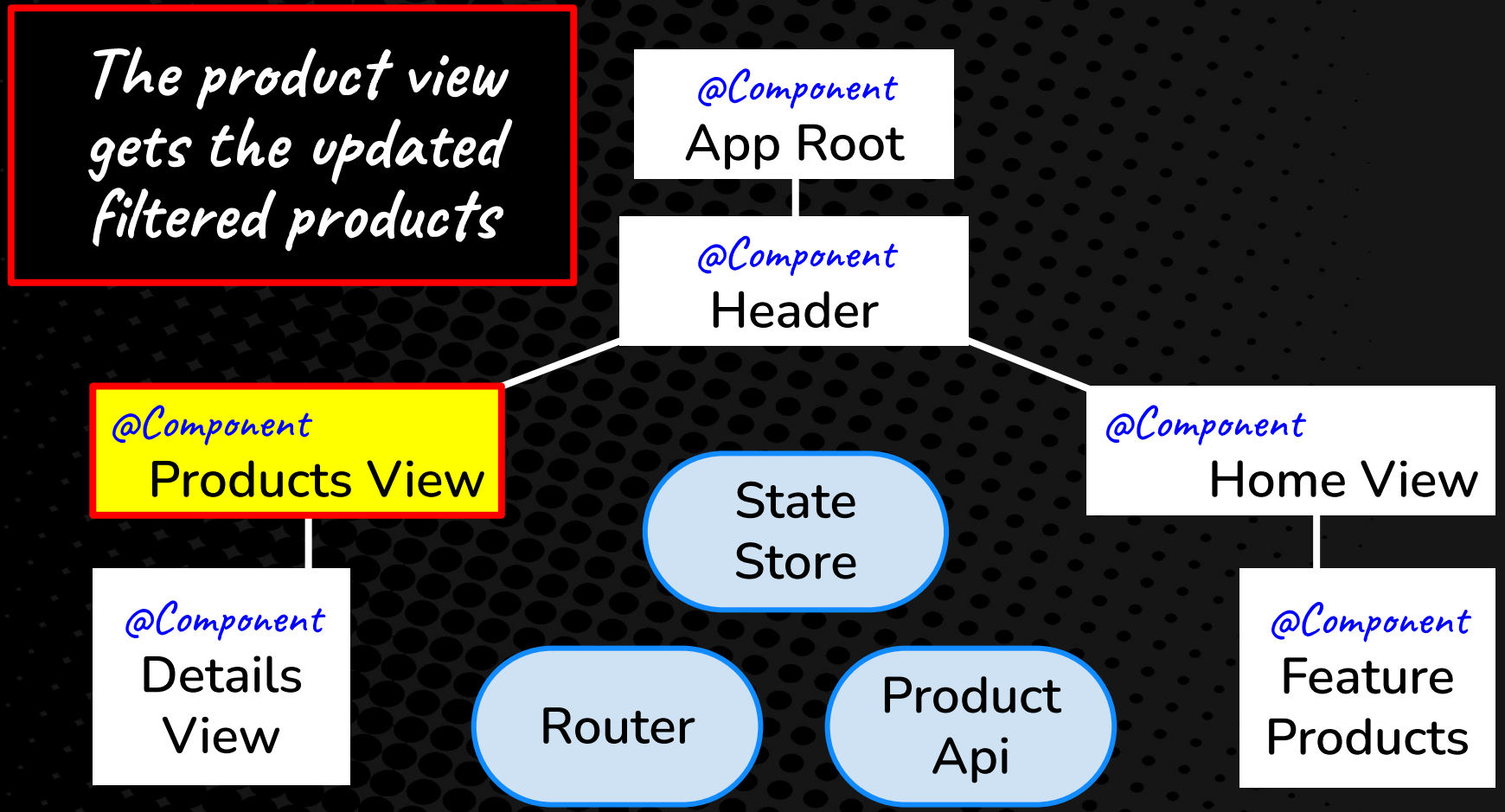
The router routes to the Products View

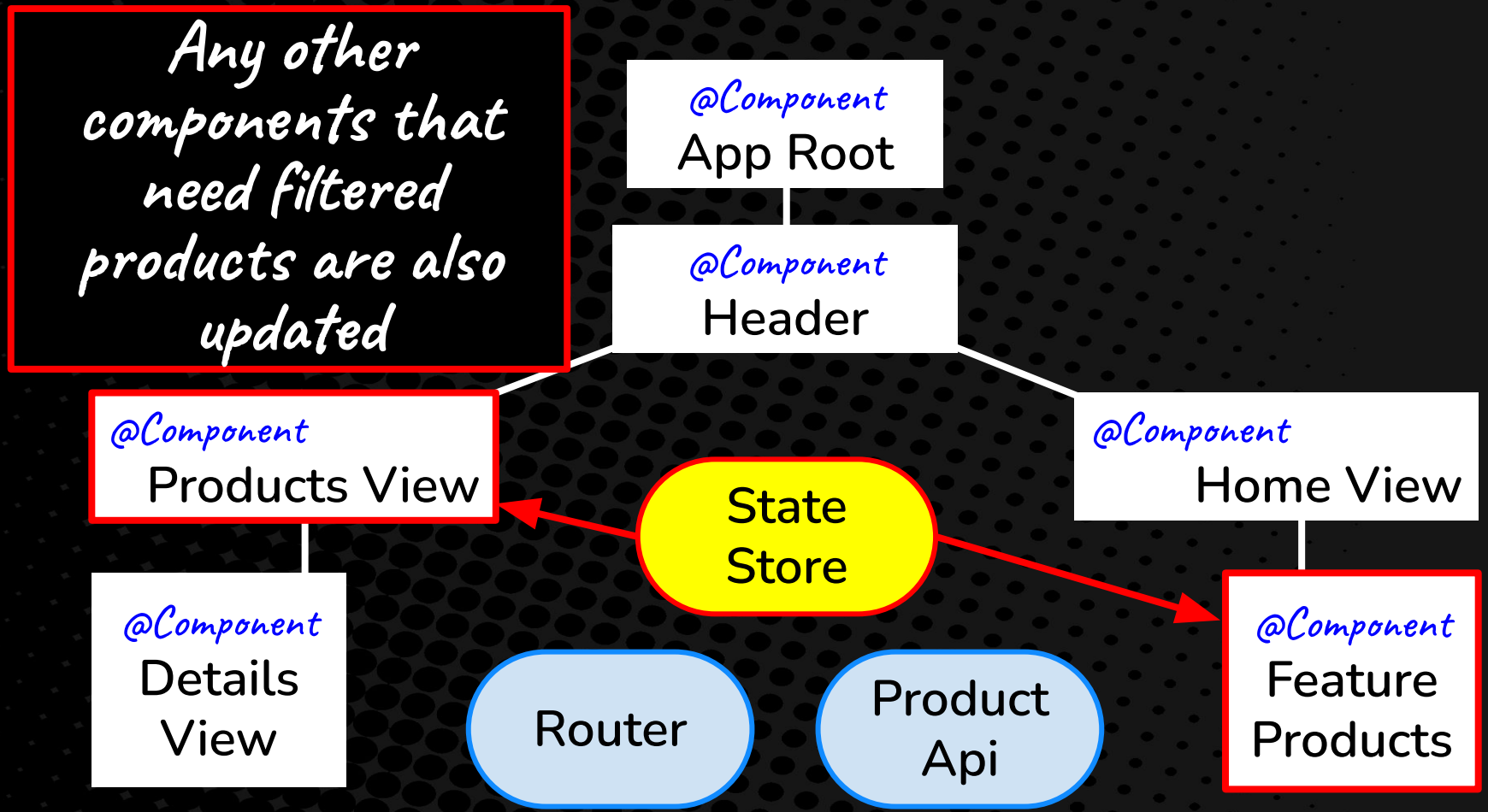



The categoryId input updates the state store
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

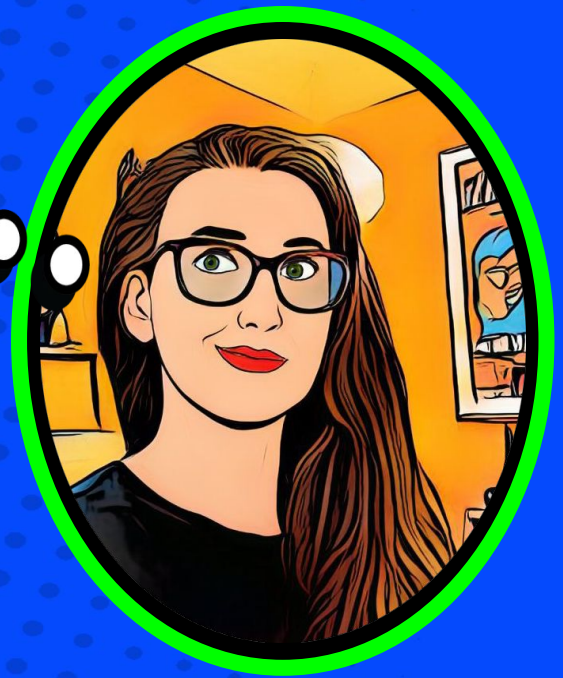
The state store emits a new value for selected category




The product view gets the updated filtered products
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products


Any other components that need filtered products are also updated
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

Let's see how we compose and consume the data



```
export class ProductService {
  priv
  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);
    }),
  ))
);
```

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

```
export class ProductService {
  priv
  read

  setS
    th
  }

  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);
      }),
    ))
  );
```

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

```
export class ProductService {
  priv
  read

  setS
    th
  }

  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);
    }),
  ))
);
```

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

```
export class ProductService {
  private re
  readonly $

  setSelecte
    this.se
  }

  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 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

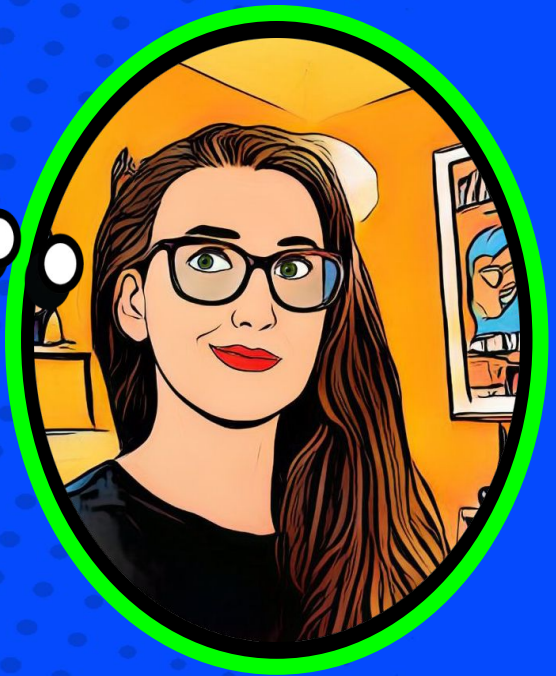
```
export class ProductViewComponent {
  @Input() set categoryId(val: string) {
    this.productsService.selectedCategory.set(val);
  }

  private readonly productsService = inject(ProductSignalsService);

  readonly filteredProducts$ = this.productsService.filteredProducts;
```

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

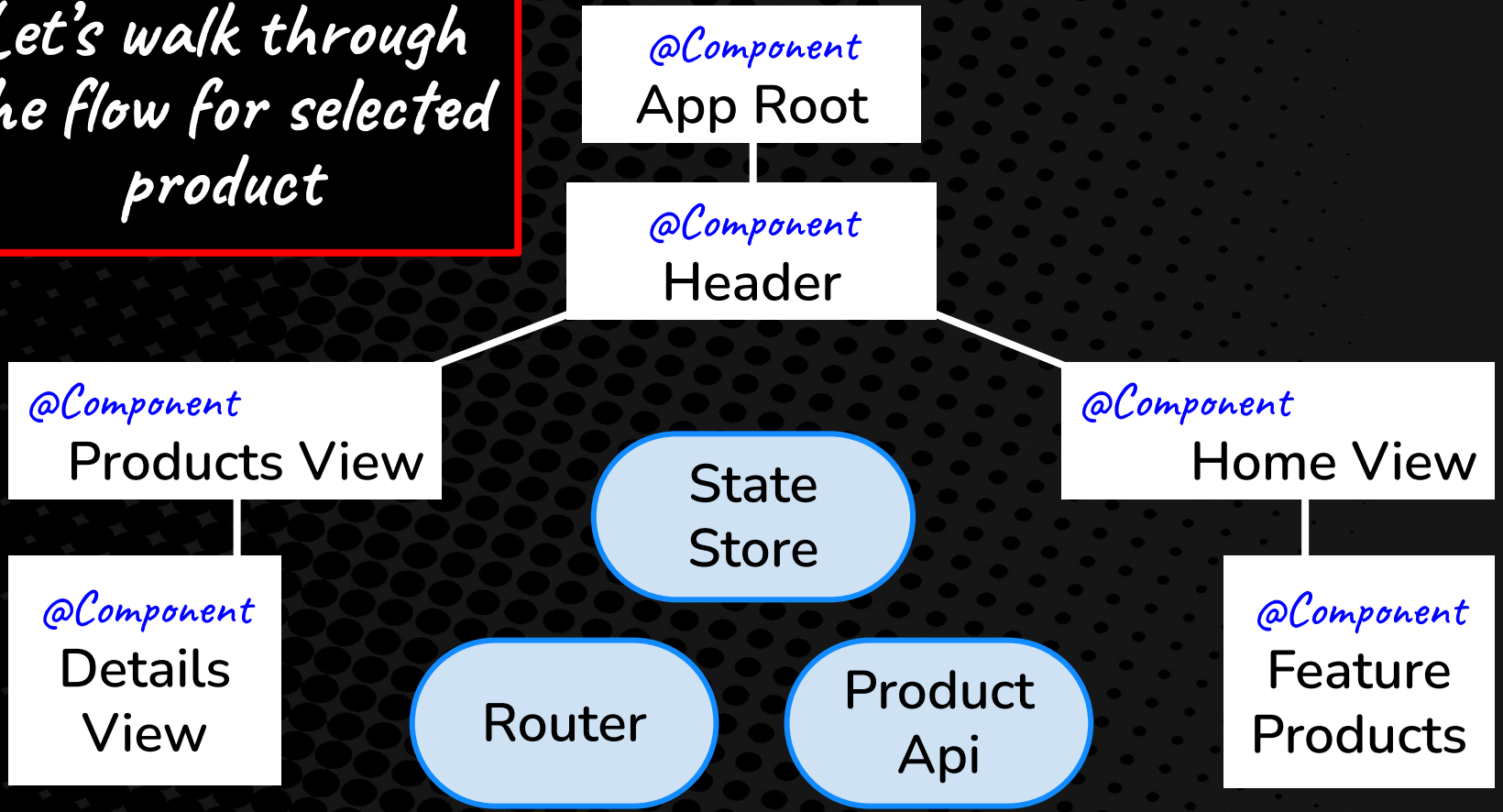
```
<button
  *ngFor="let product of filteredProducts()"
  [routerLink]="['detail']"
  [queryParams]="{
    productId: product.id
  }"
  routerLinkActive="router-link-active"
>
</button>
```

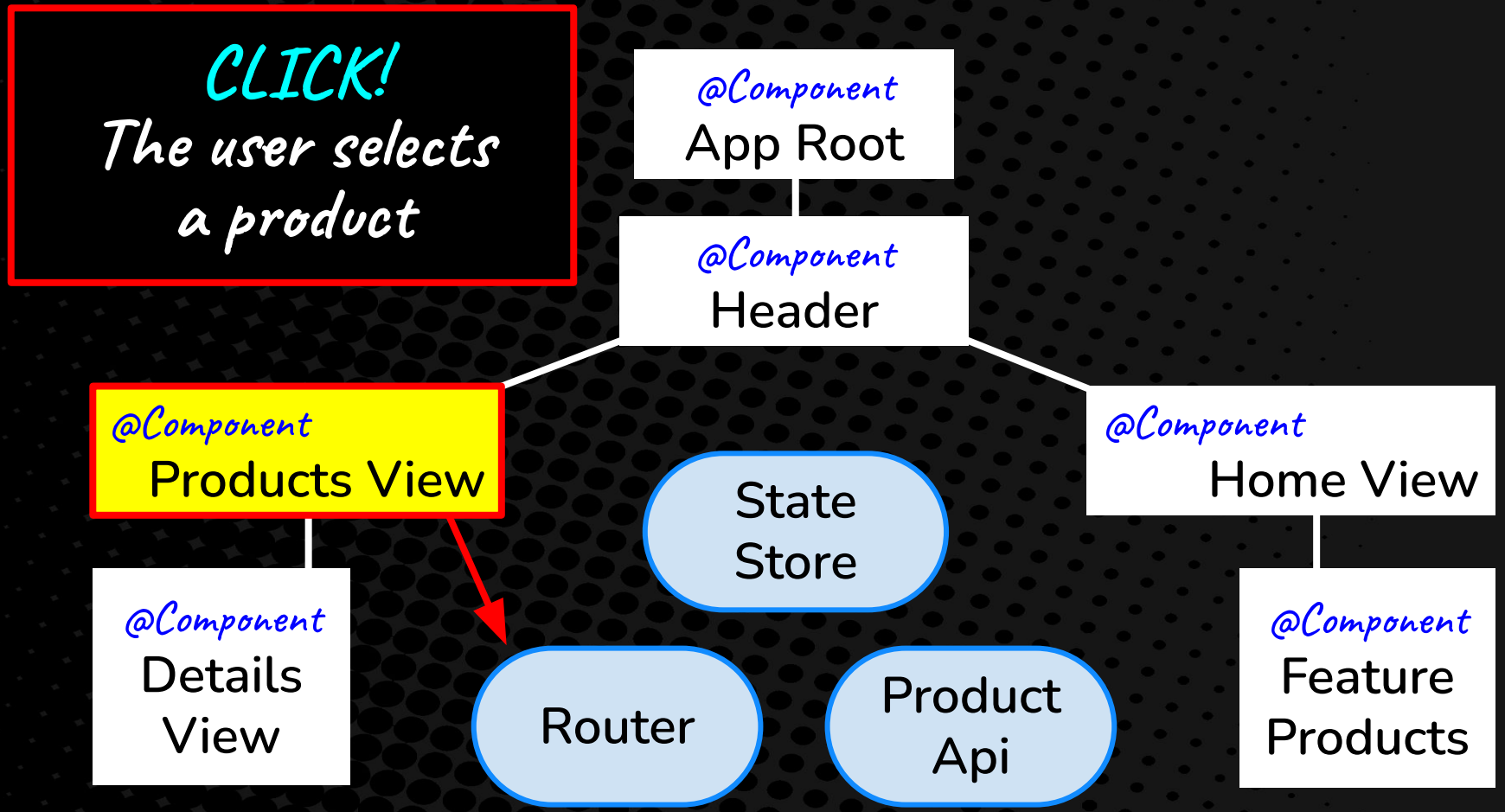
In the template we can use the value of the filteredProducts signal directly

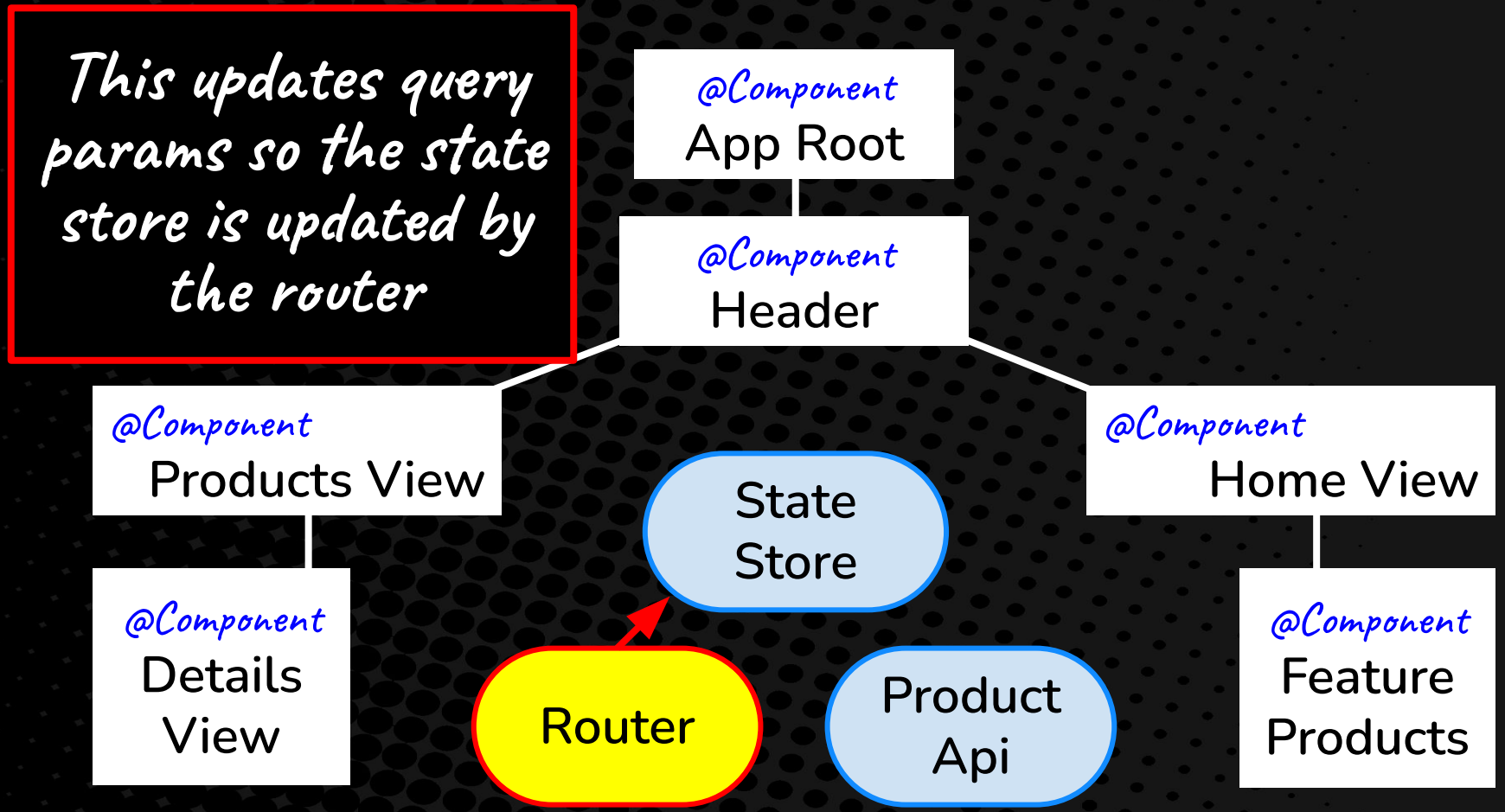
Notice how similar the pattern is for observables and for signals

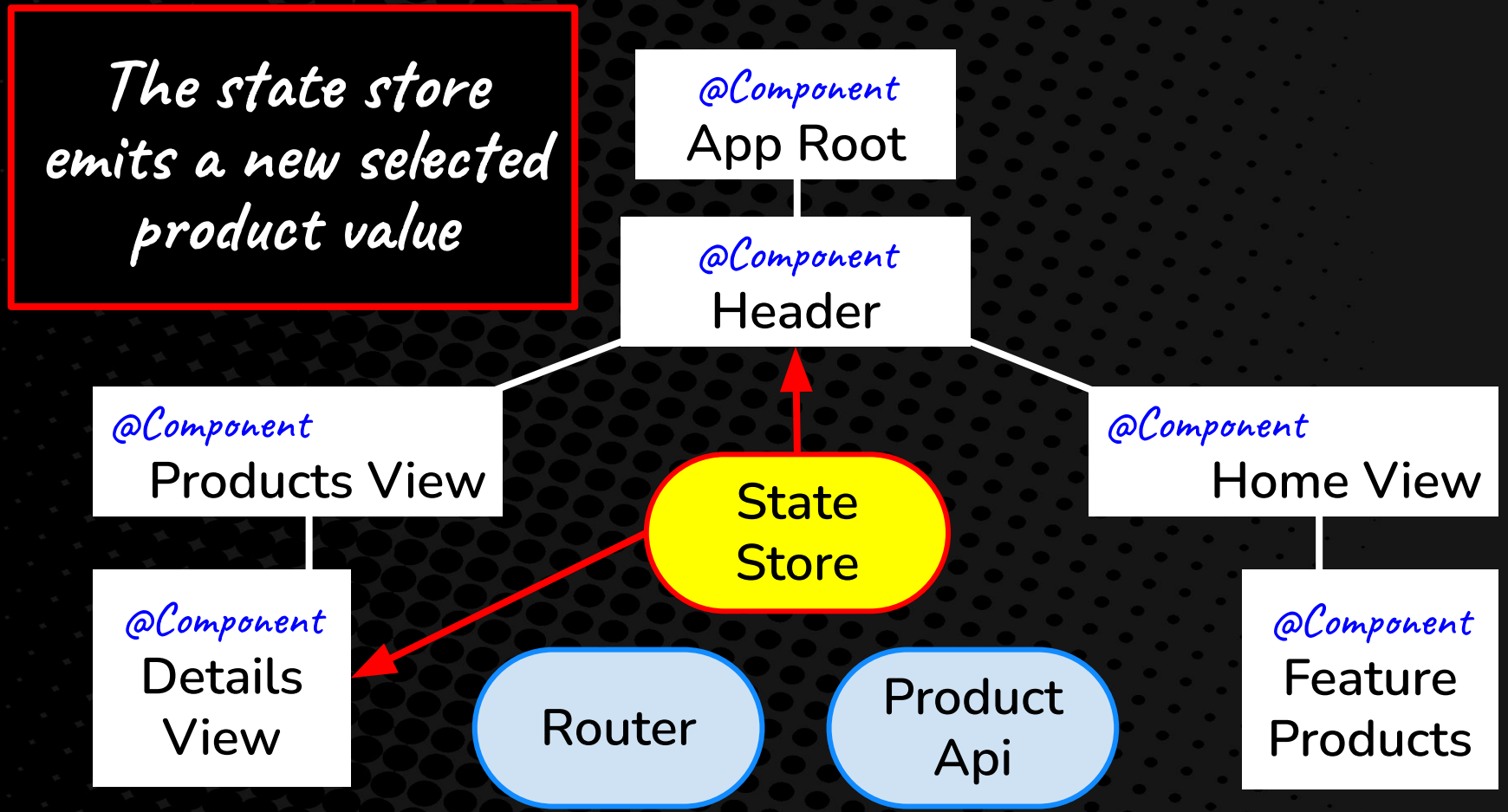


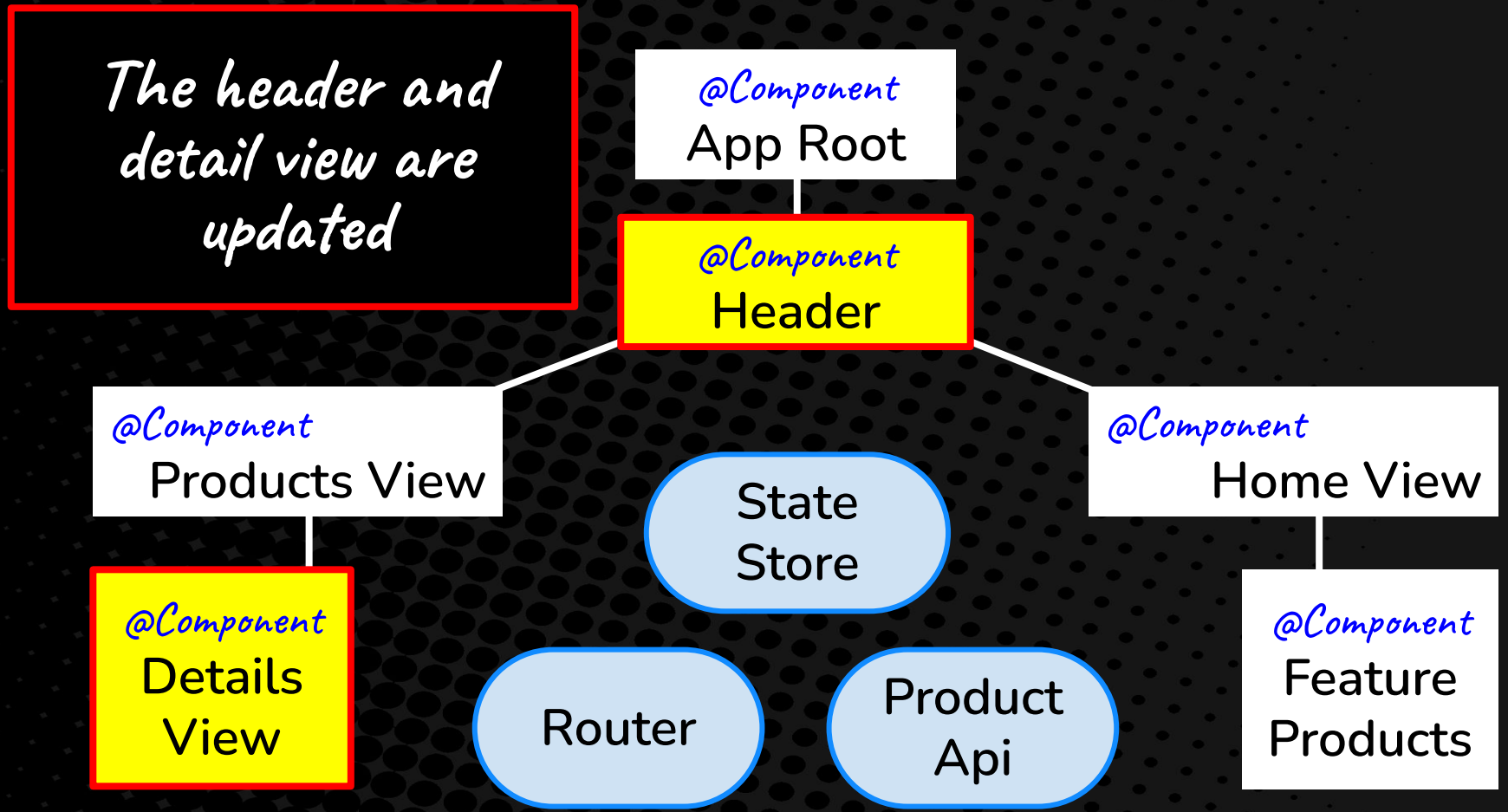
Let's walk through the flow for selected product



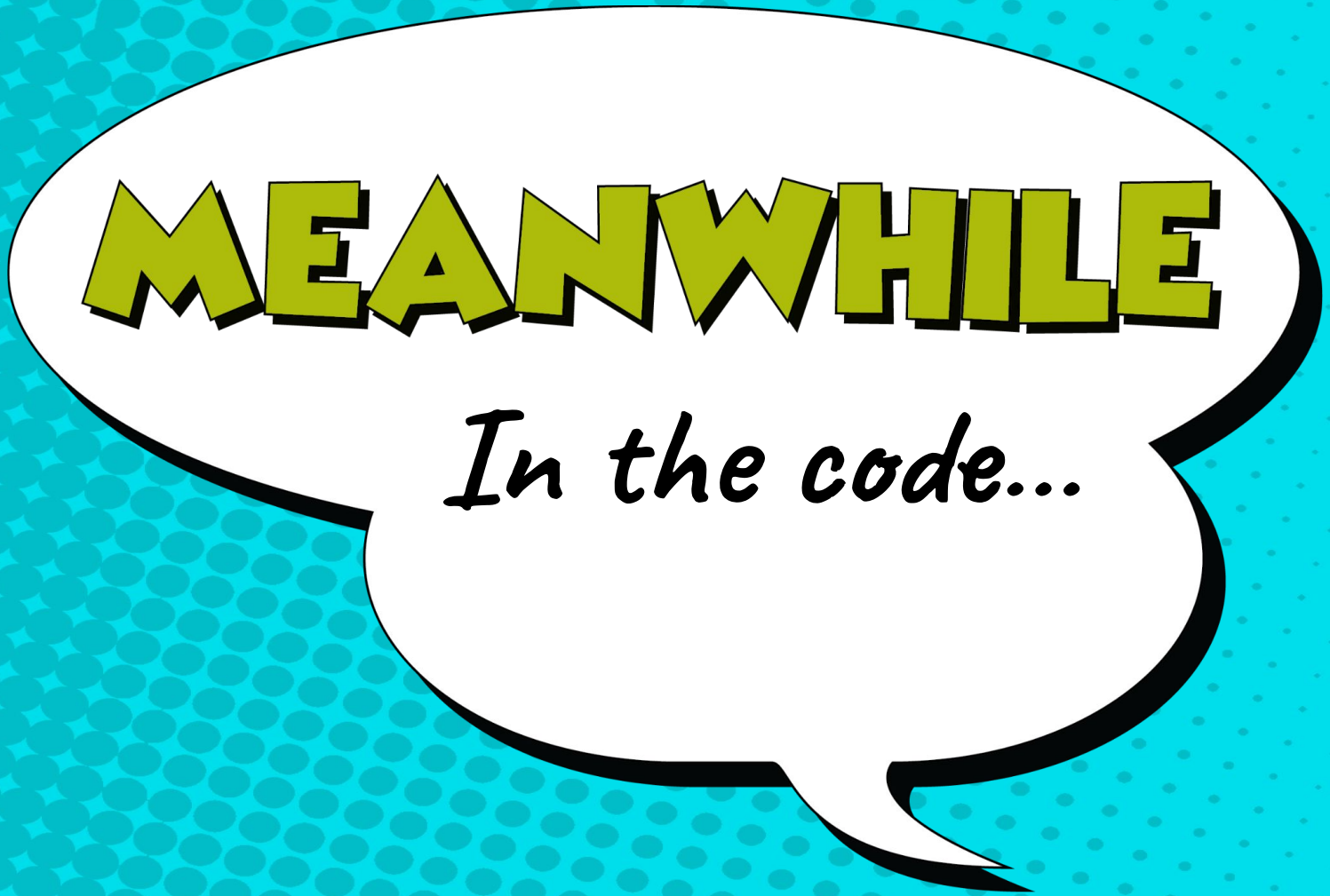

CLICK!
The user selects a product
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

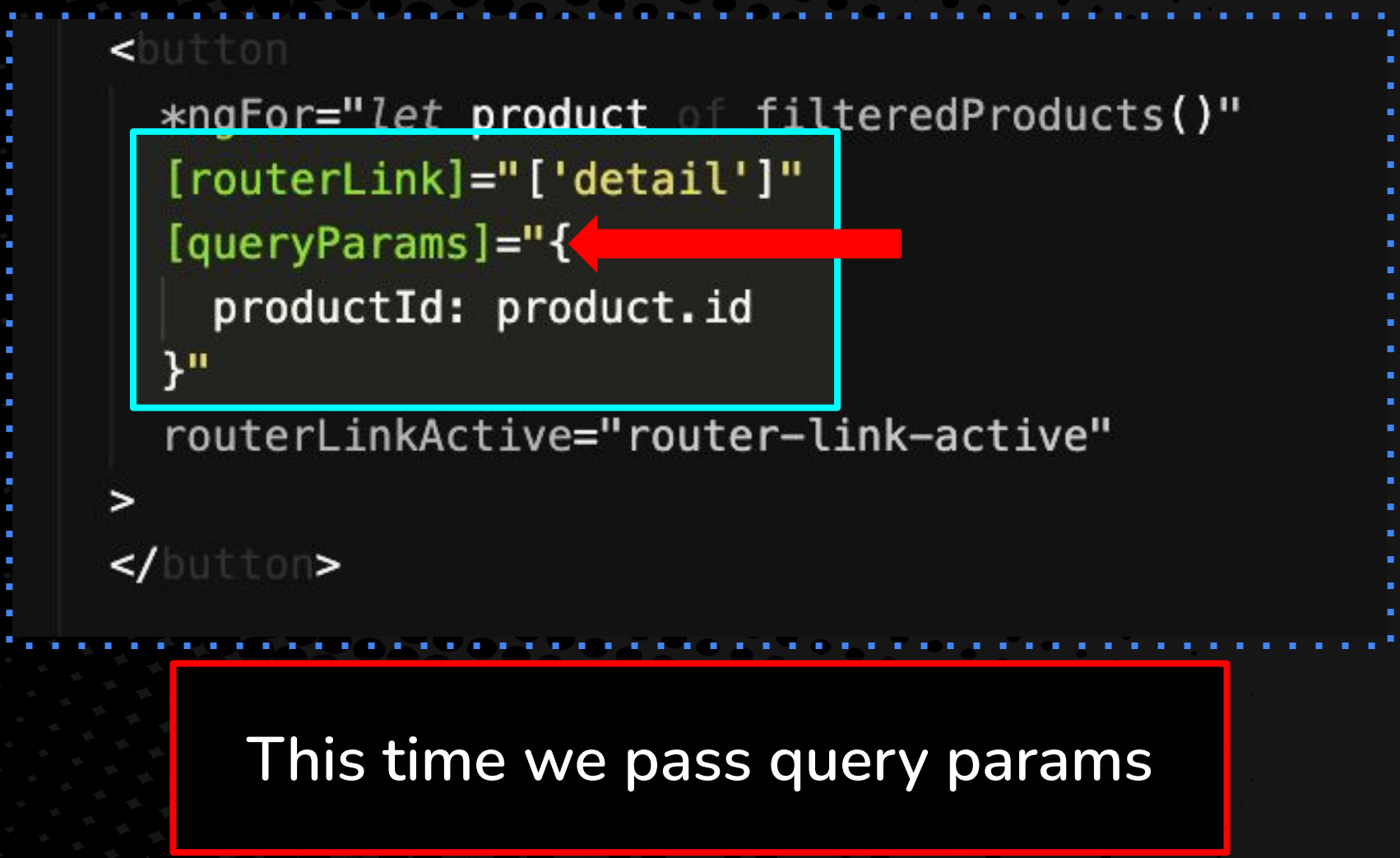

This updates query params so the state store is updated by the router
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
@Component
Details View
State Store
Router
Product Api
@Component
Feature Products


The state store emits a new selected product value
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products


The header and detail view are updated
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

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 *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 *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:
```

We bind the consuming component templates to the selectedProduct property

Just Like People PET BOUTIQUE

PRODUCTS CONTACT

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!!***

Just Like People PET BOUTIQUE

PRODUCTS CONTACT

Denim Pooch Pants ON SALE NOW!!!

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








Denim Pooch Pants

OMG Pooch Pants are on sale!!

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!!*

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?

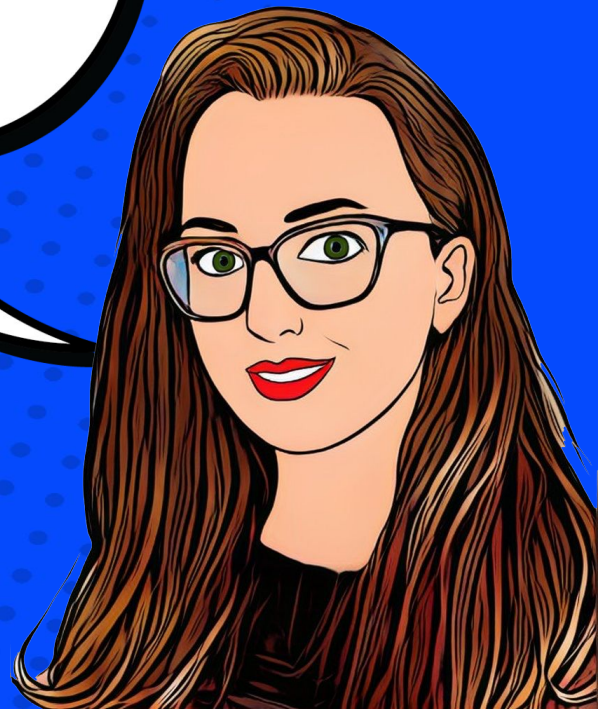
YES!
We are getting ready by...

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



Slide Deck and Demo App

github.com/lara-newsom/reactive-patterns