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

Technical Lead Cisco

Slide Deck and Demo App

github.com/lara-newsom/reactive-patterns

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.

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

Using setters on Inputs instead of ngOnChanges

```
export class DetailViewComponent {
  @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.productsSignals.selectedProductId.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

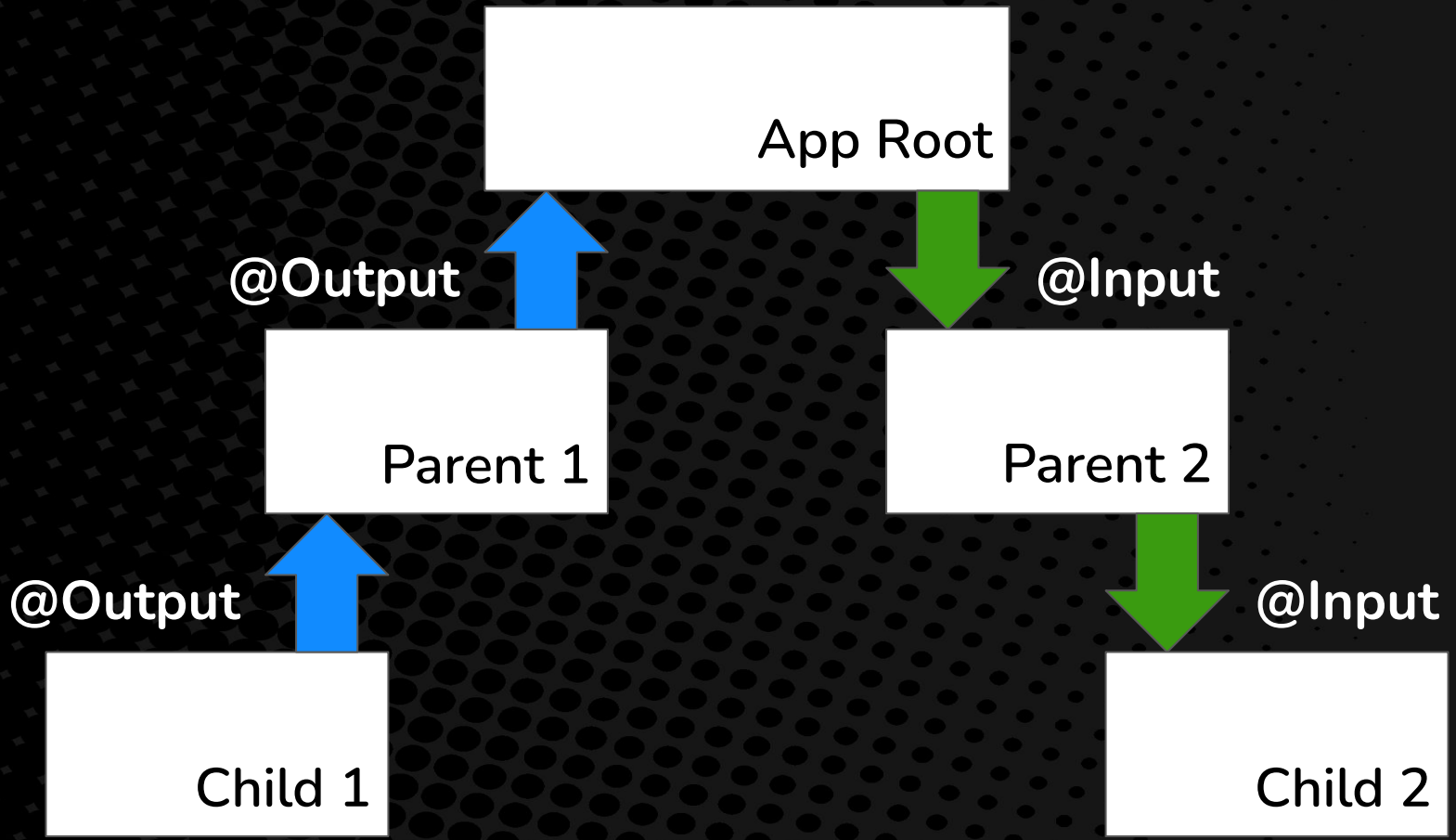
Not great tool for passing data to other parts of the app

Tightly couples the emitting component to the tree

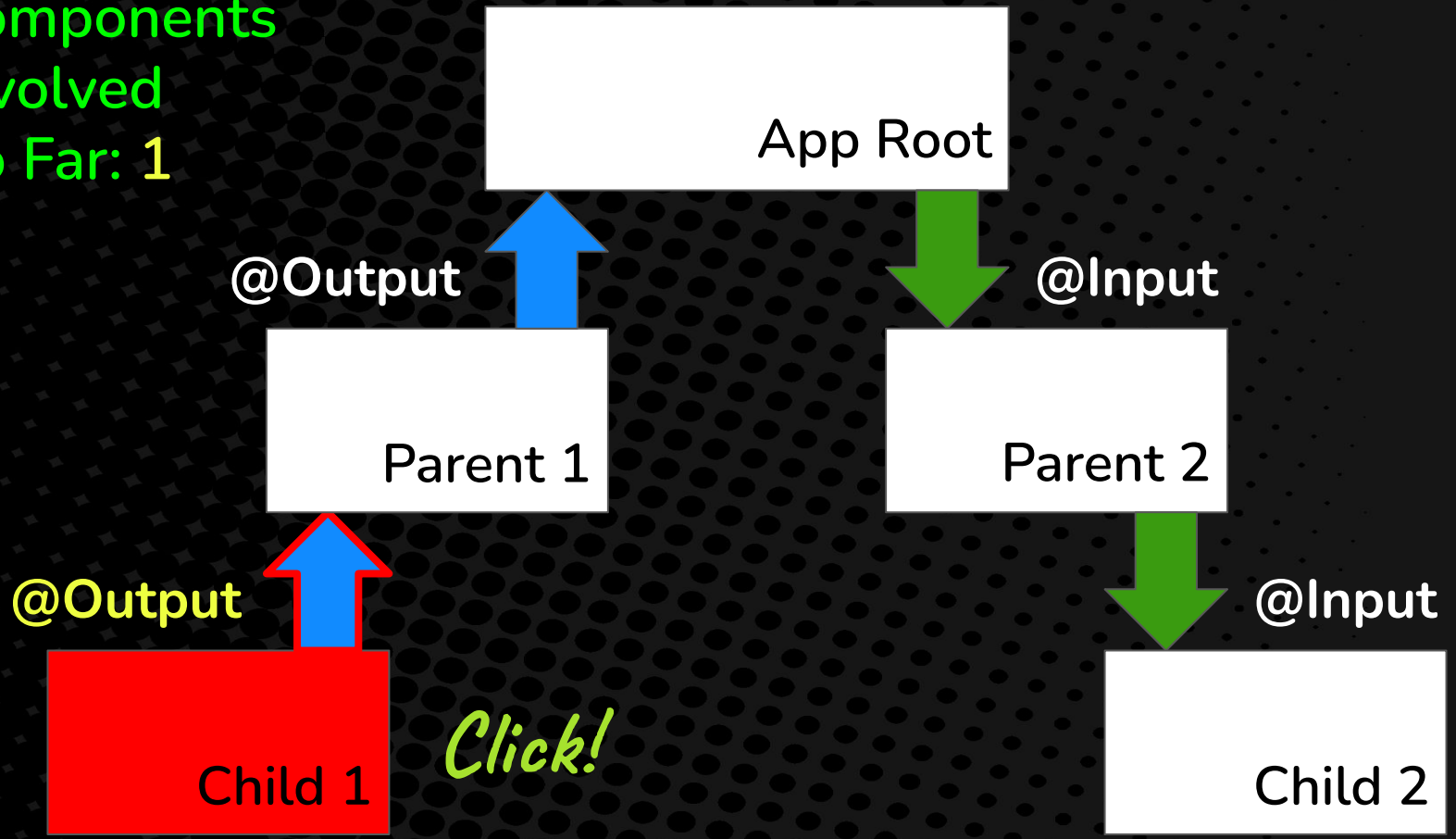
```
export class SideMenuComponent {
  @Output() selectedProduct = new EventEmitter<string>();

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

Emits event here

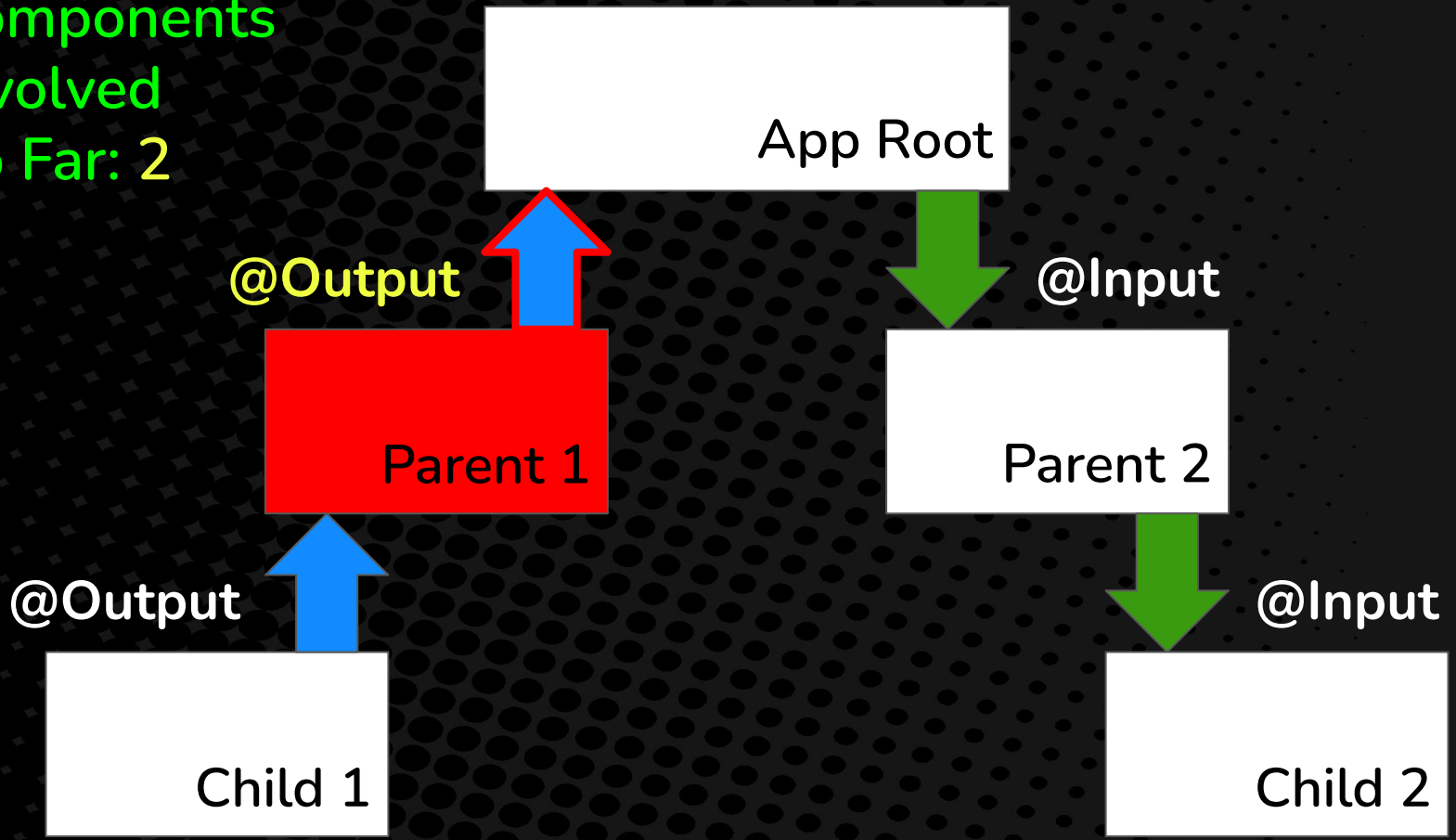
Let’s look at the data flow
selecting a product using Outputs


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

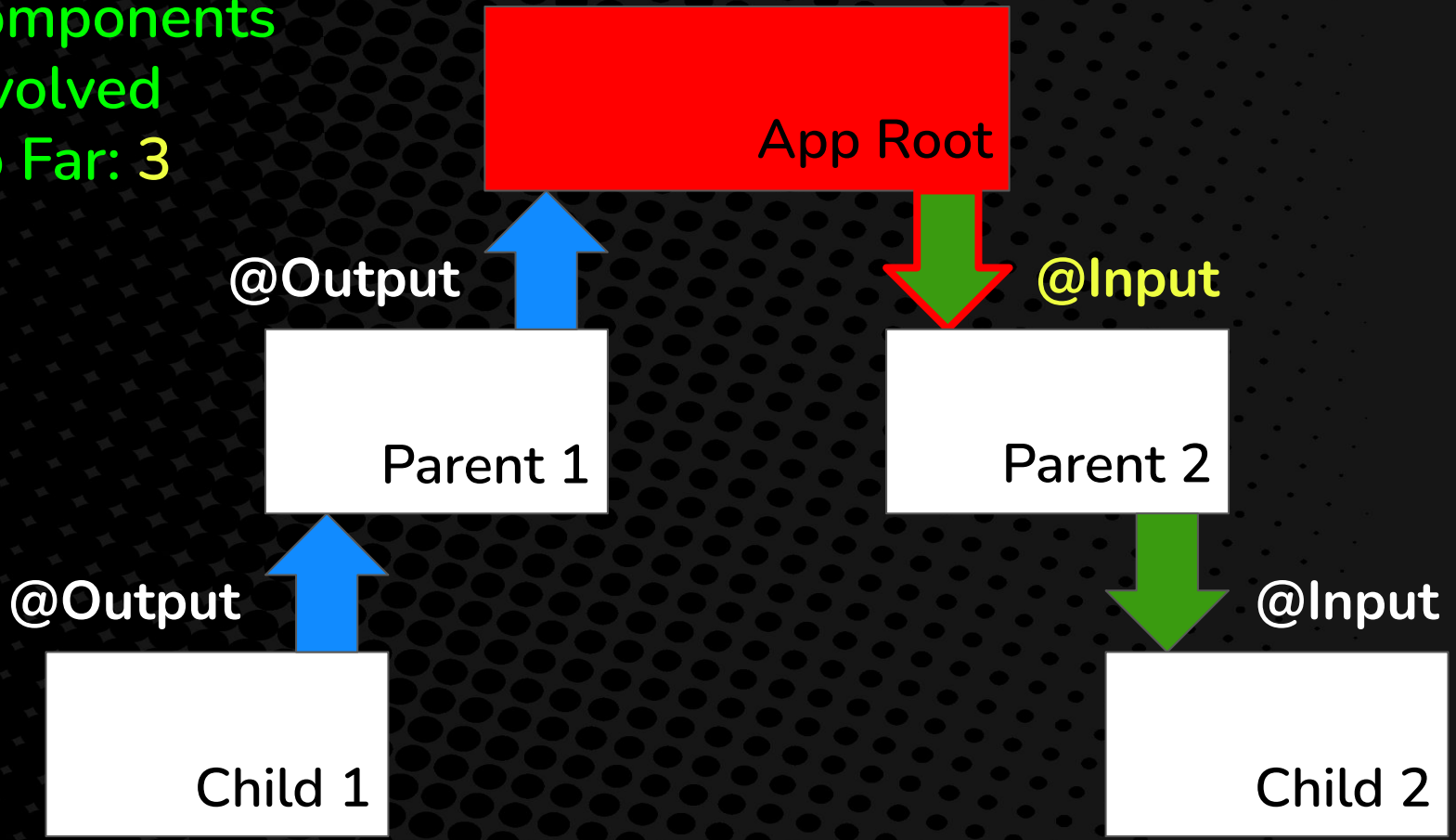
Components Involved So Far: 1



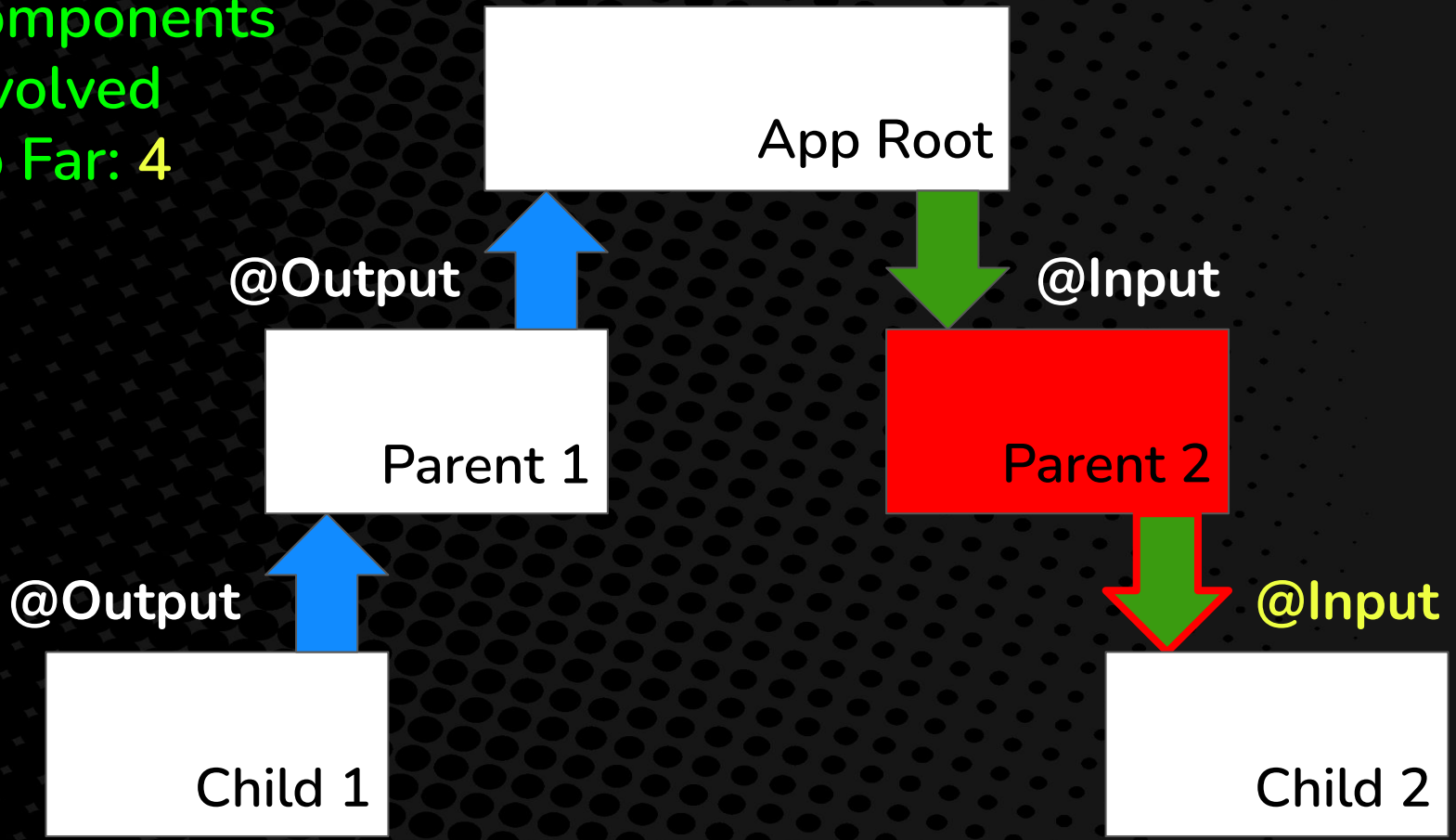
Components Involved So Far: 2



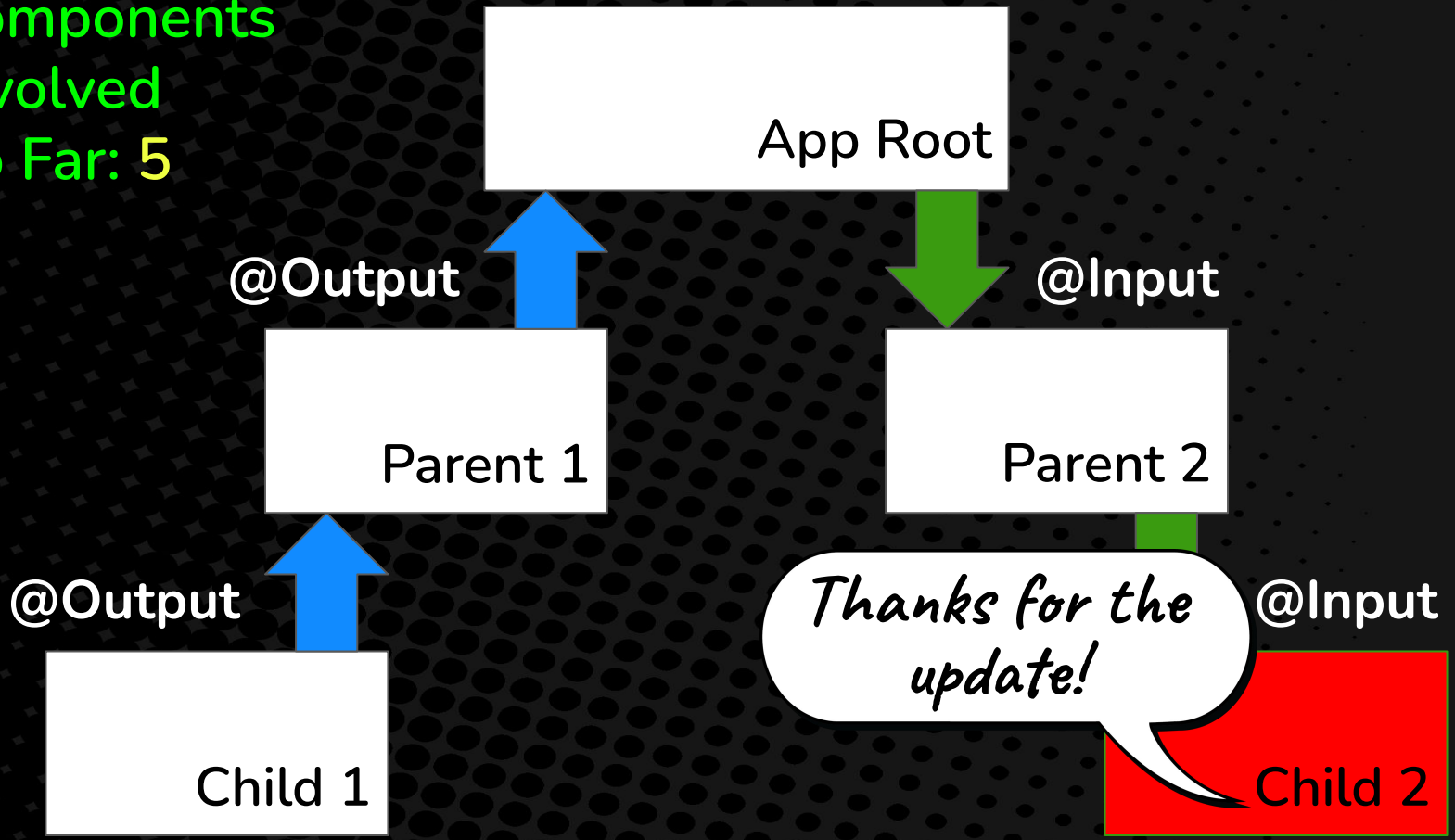
Components Involved So Far: 3



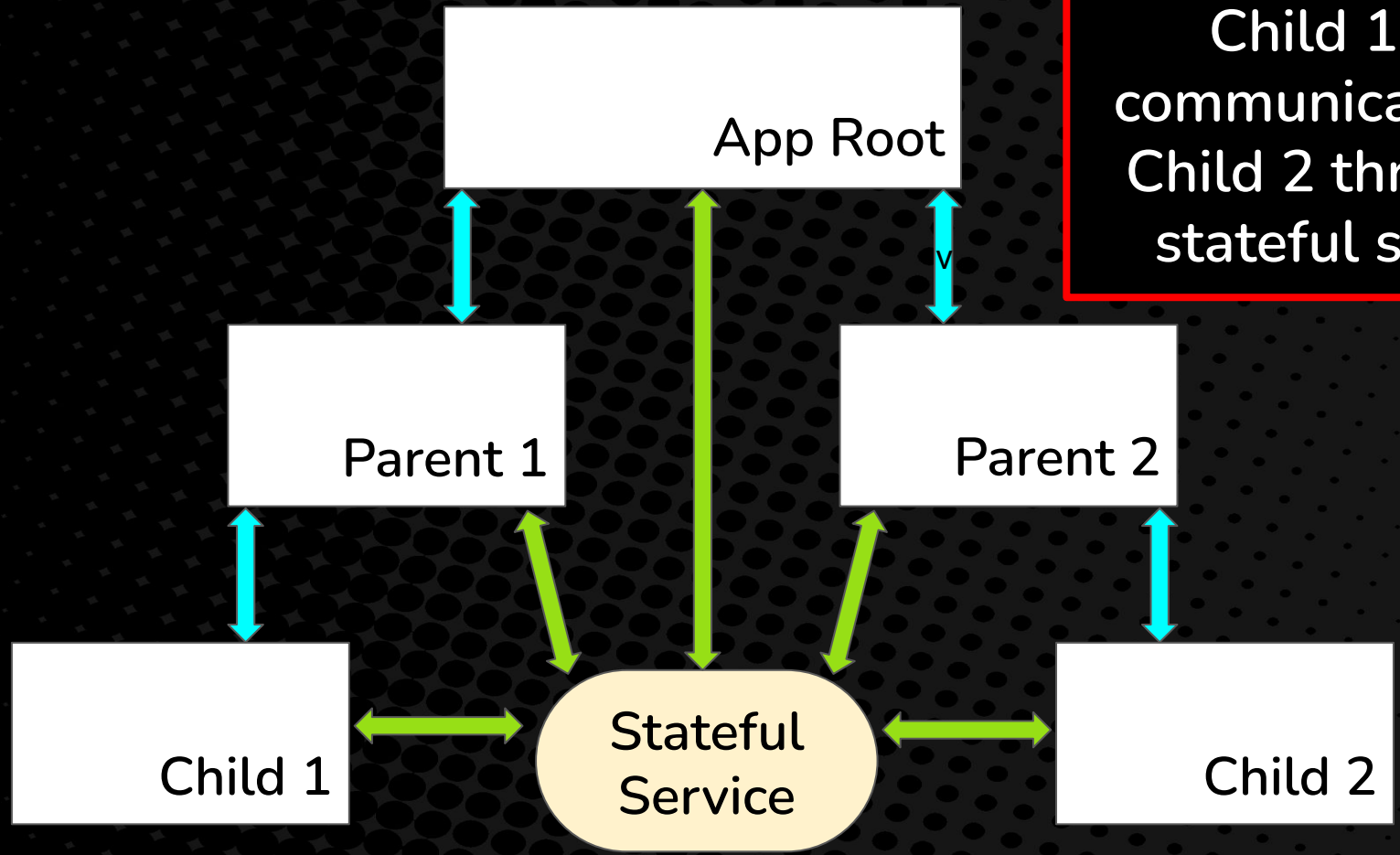
Components Involved So Far: 4



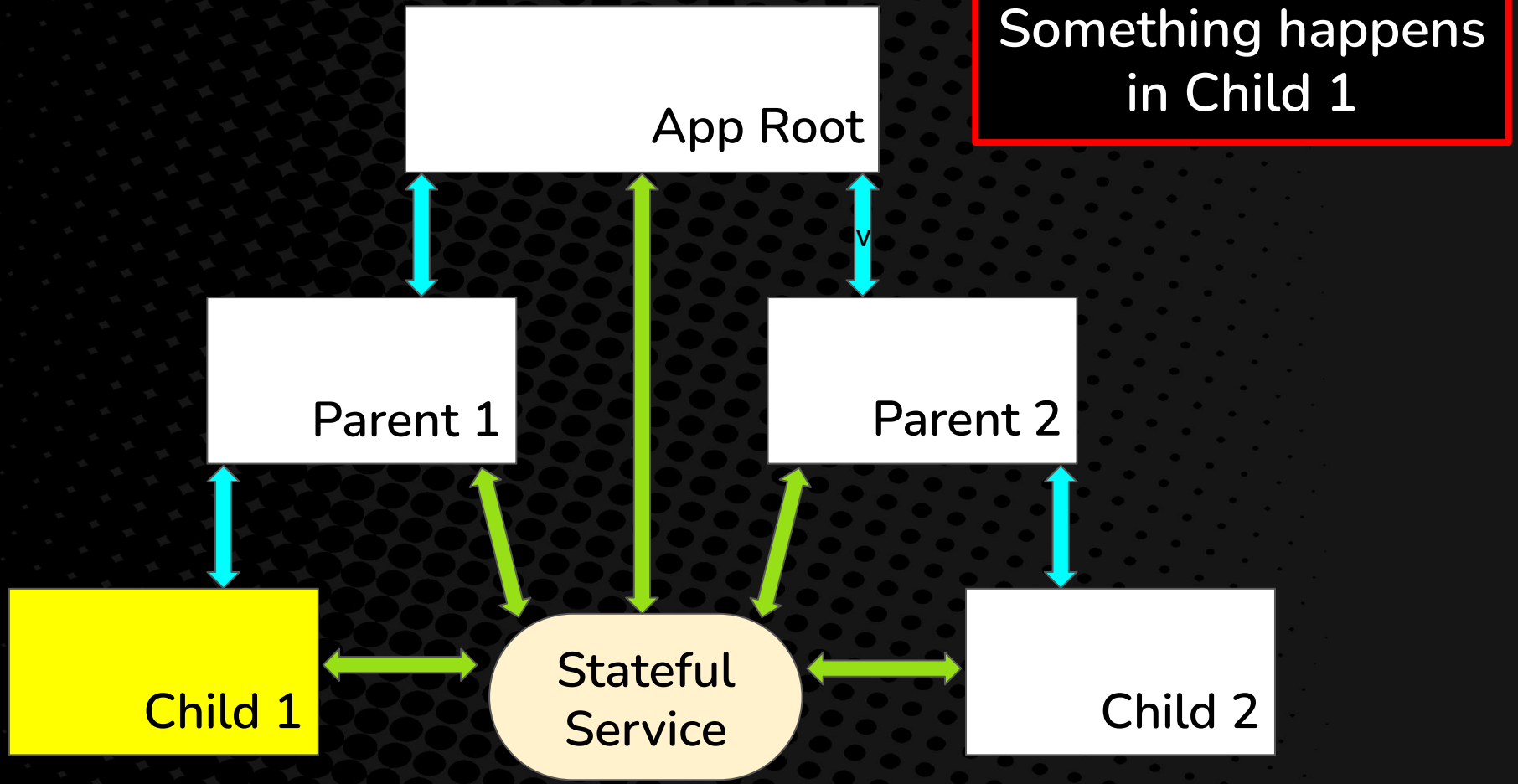
Components Involved So Far: 5

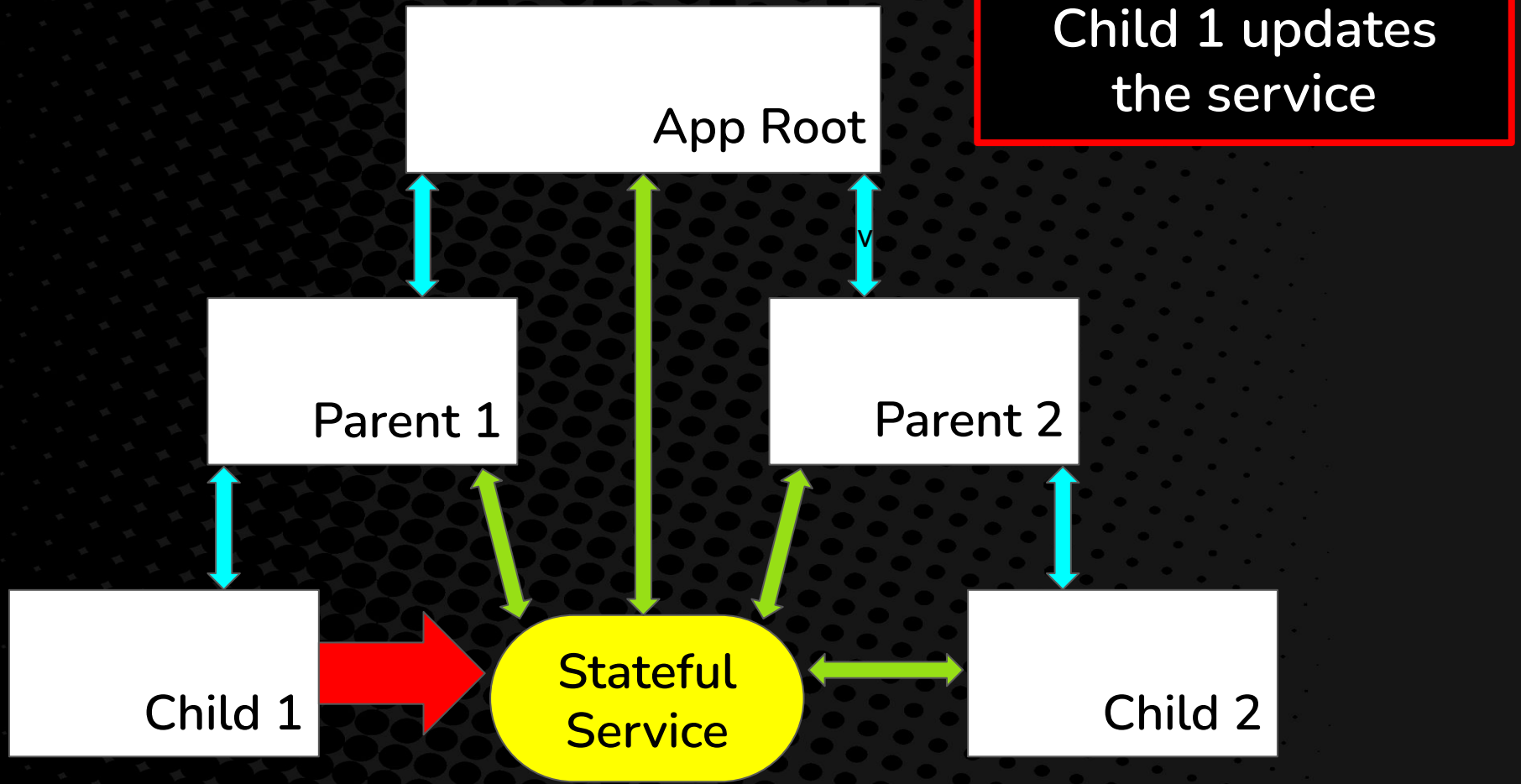


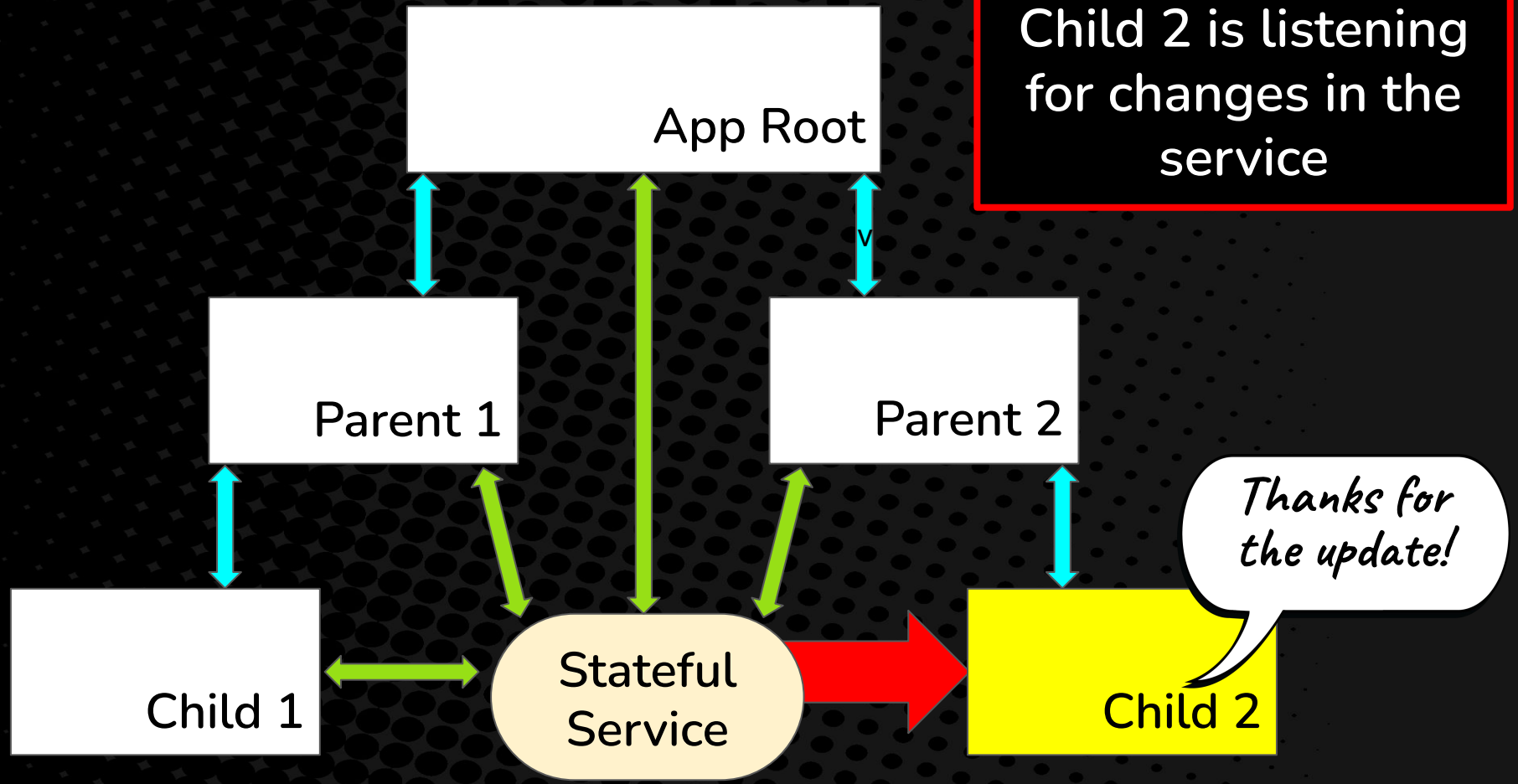
We need a stateful service that can be accessed from anywhere in the app



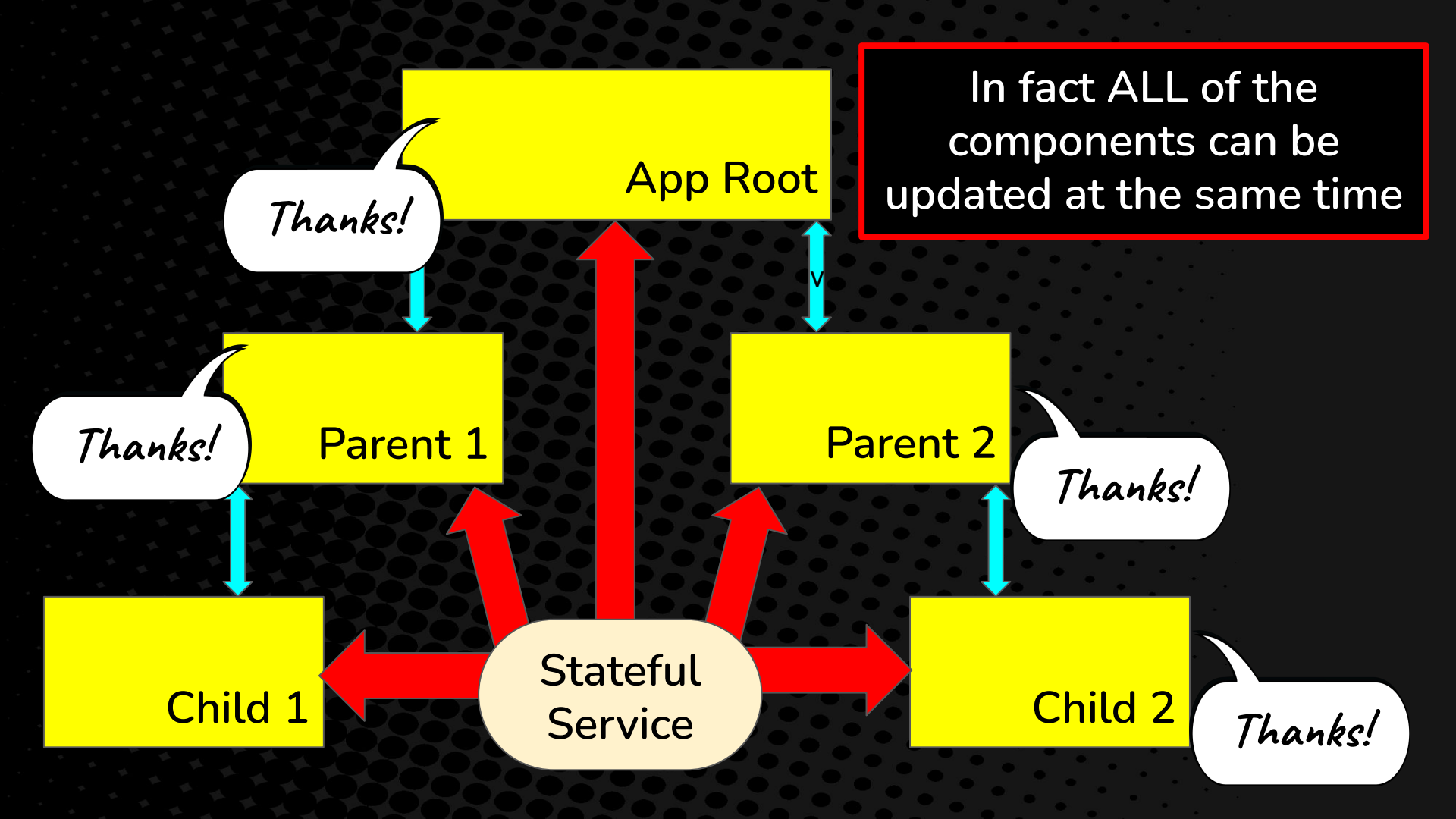
Child 1 can communicate with Child 2 through a stateful service


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



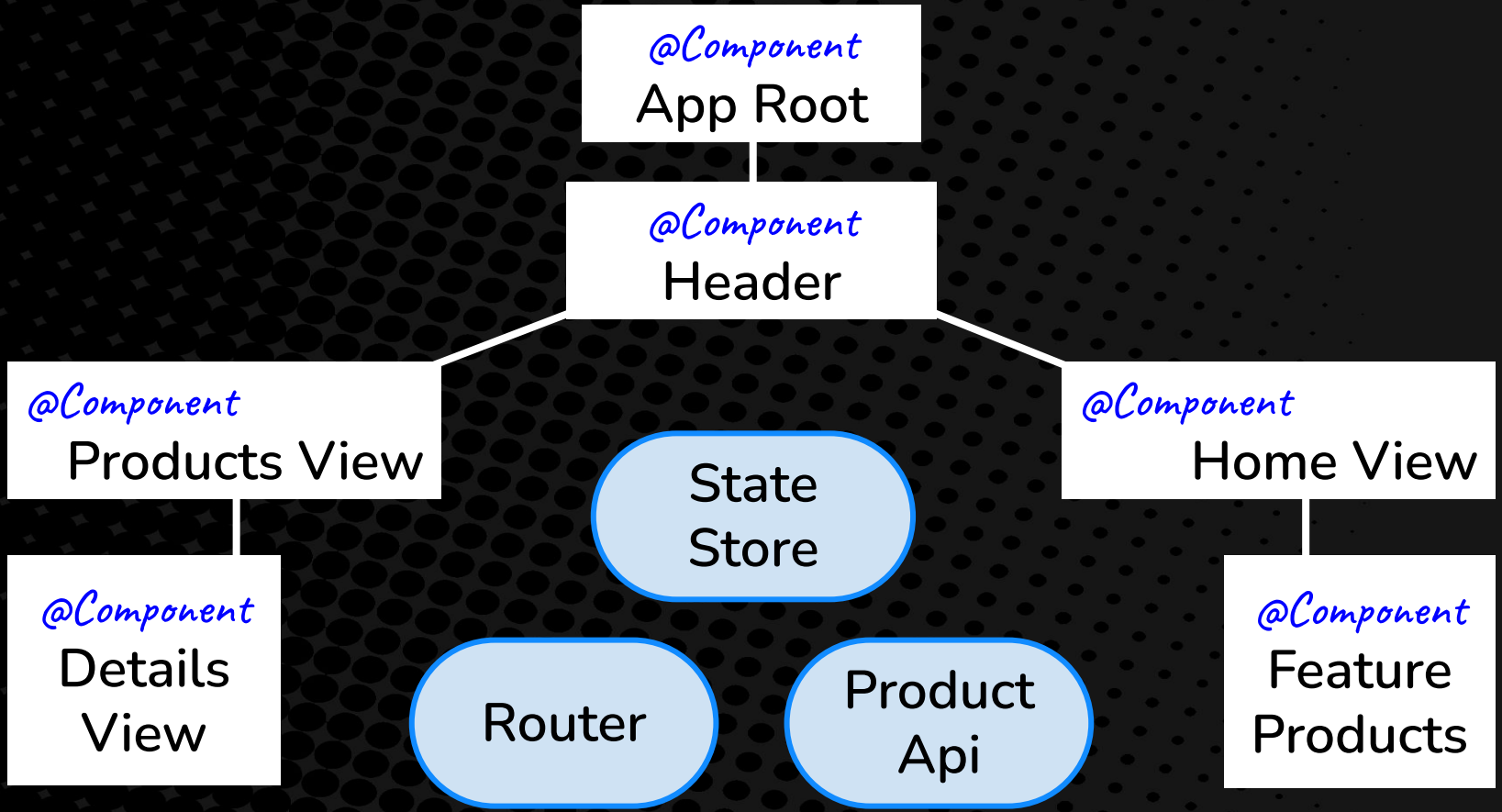
Child 2 is listening for changes in the service

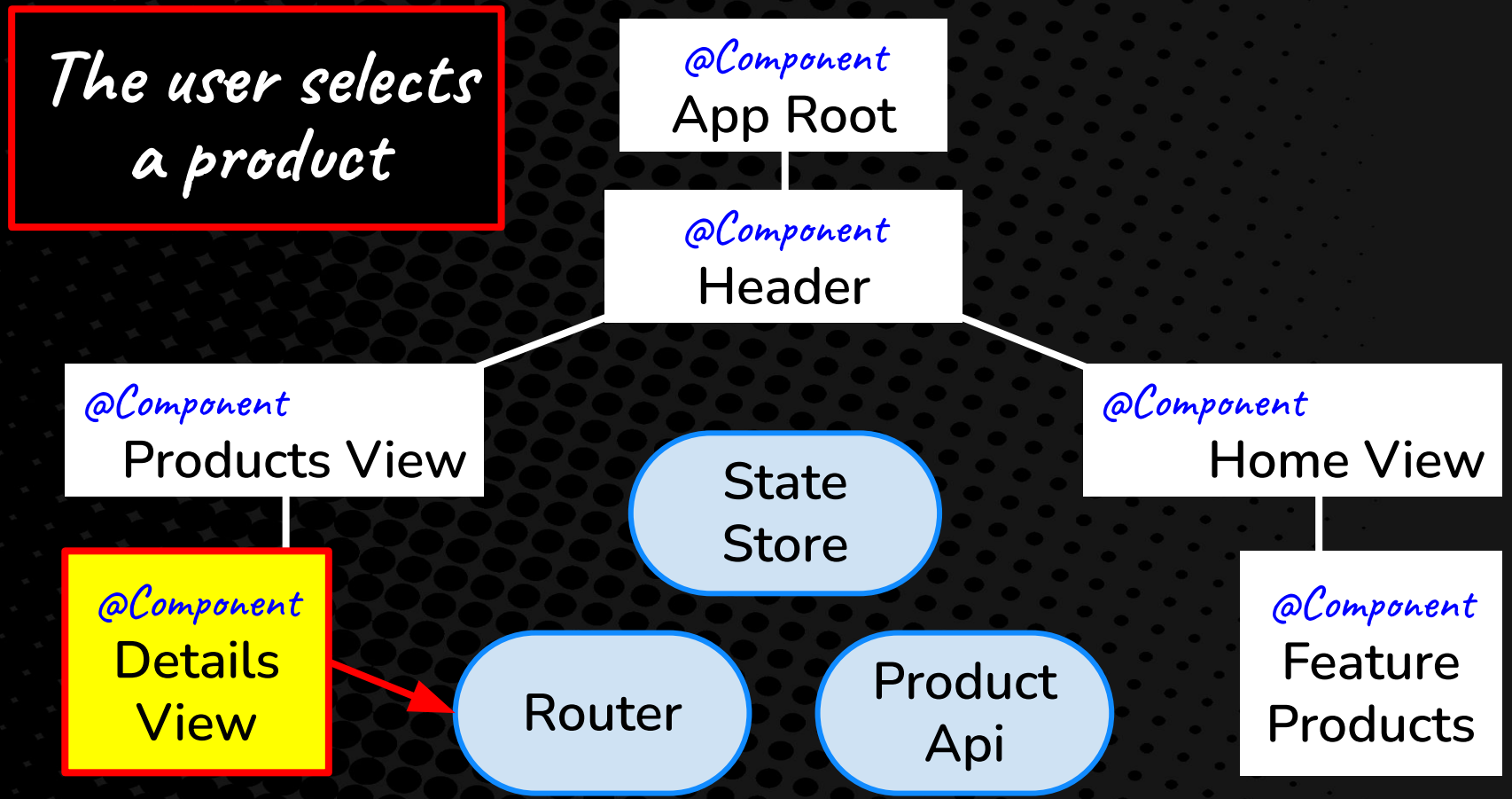

In fact ALL of the components can be updated at the same time
App Root
Thanks!
Parent 1
Thanks!
Parent 2
Thanks!
Child 1
Stateful Service
Child 2
Thanks!

Built-in Stateful Options

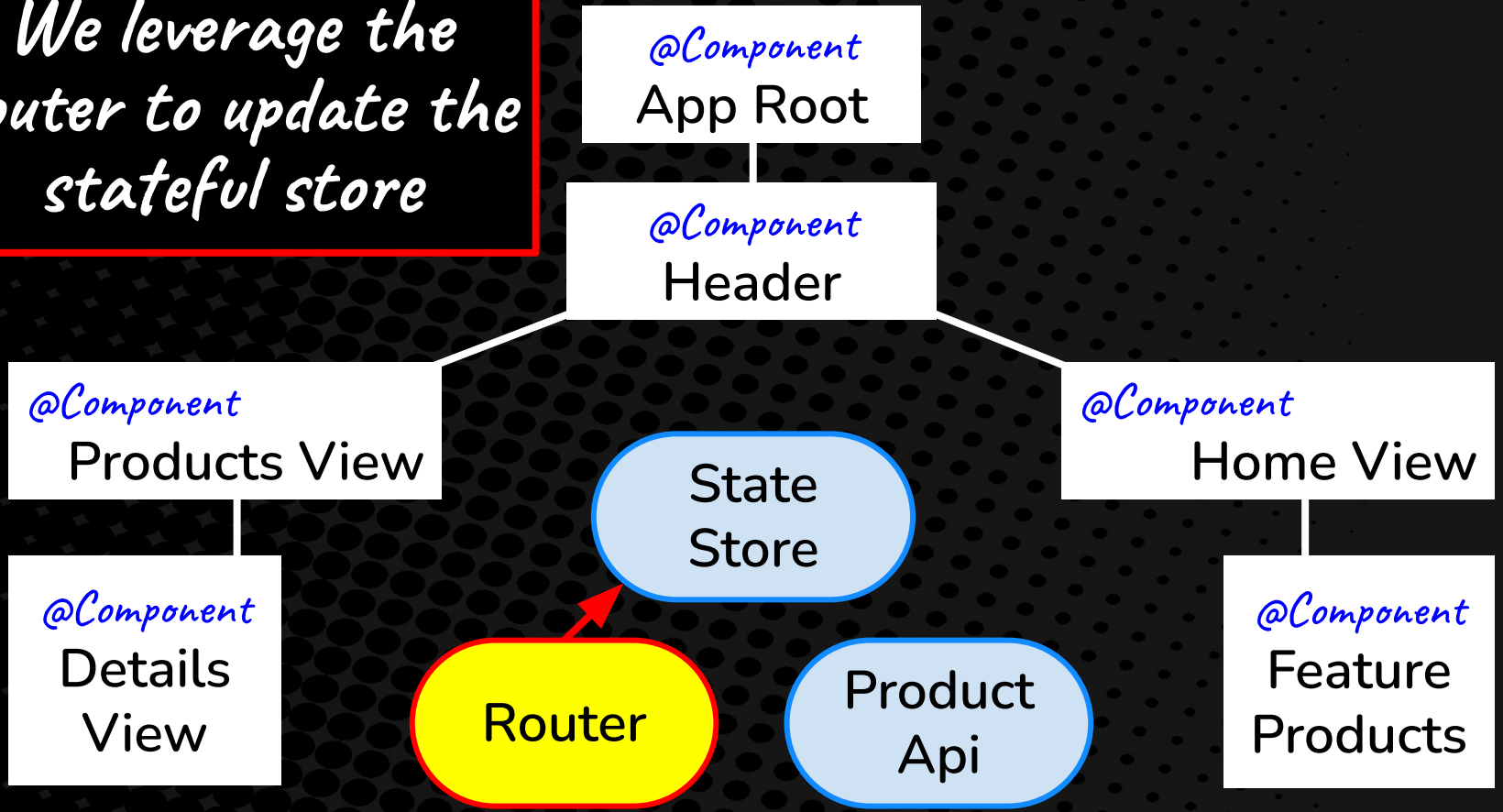
- Service with a subject
- Service with a Signal
- Router Module

For shareable user selections, we can leverage the Angular Router

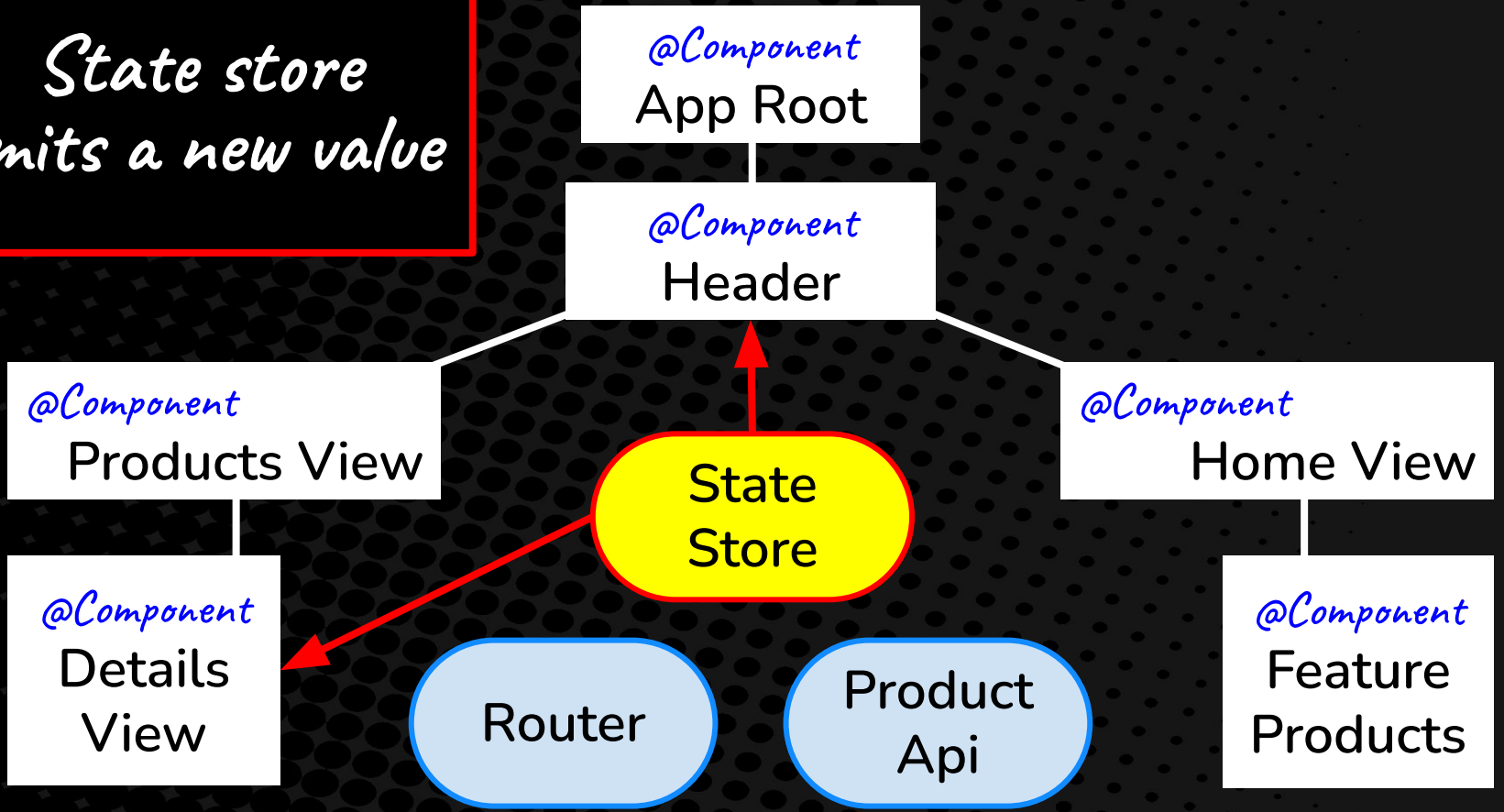

@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State
Store
@Component
Details
View
Router
Product
Api
@Component
Feature
Products

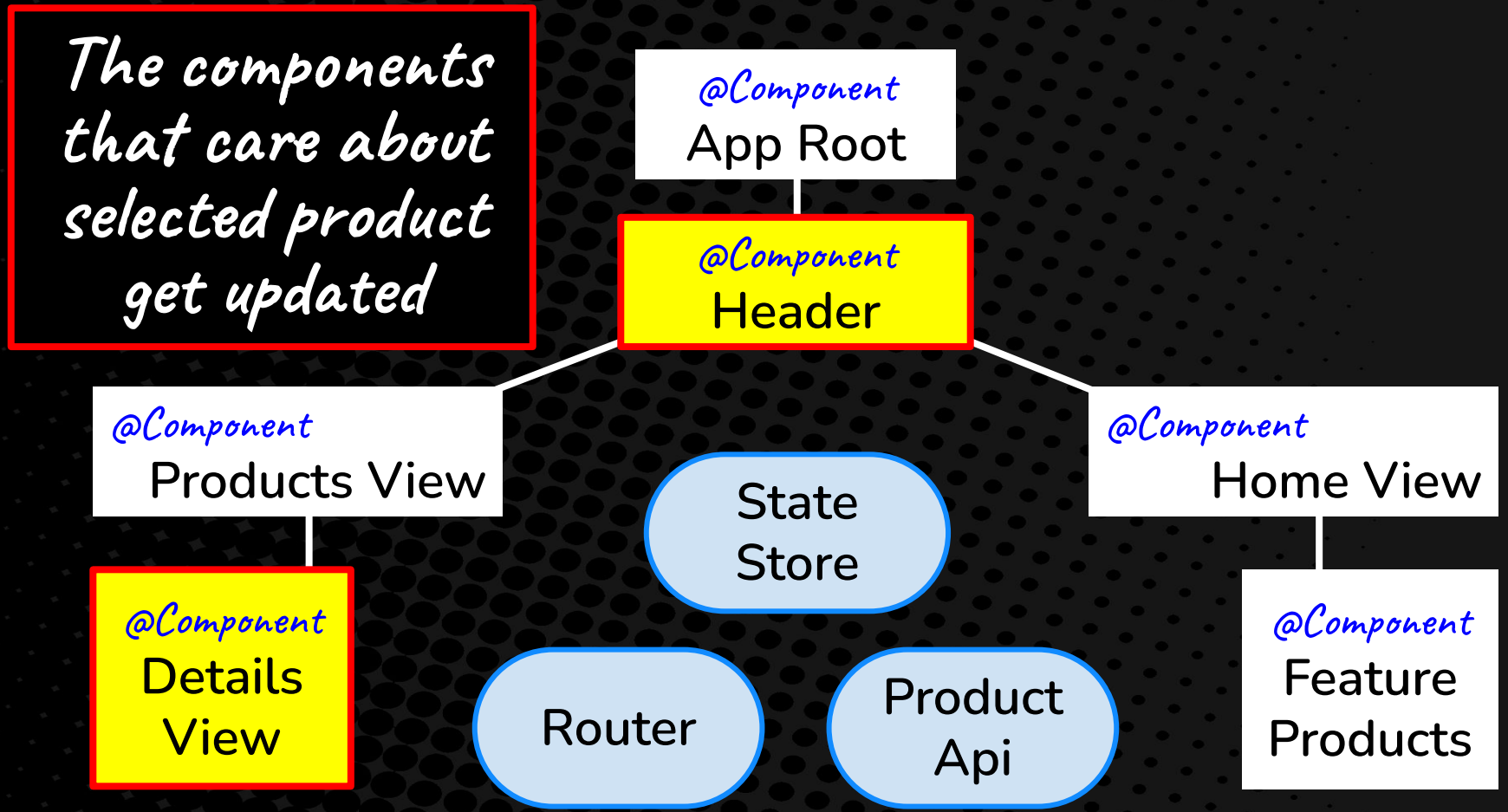

The user selects a product
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

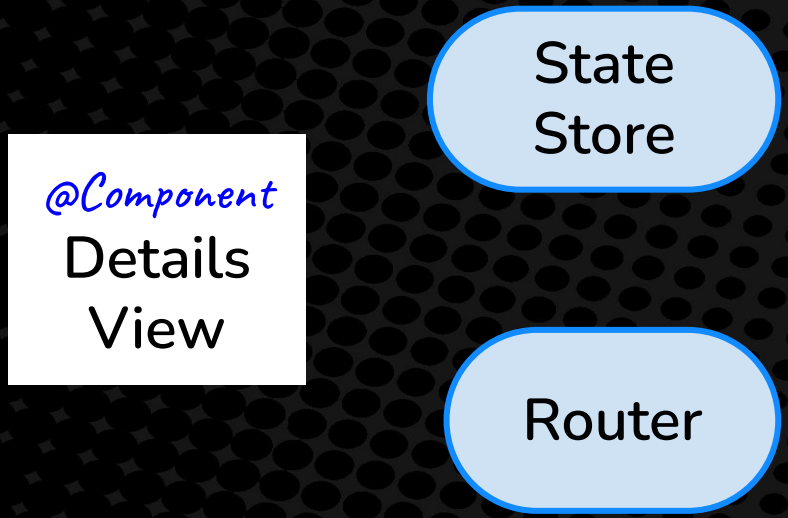
We leverage the router to update the stateful store



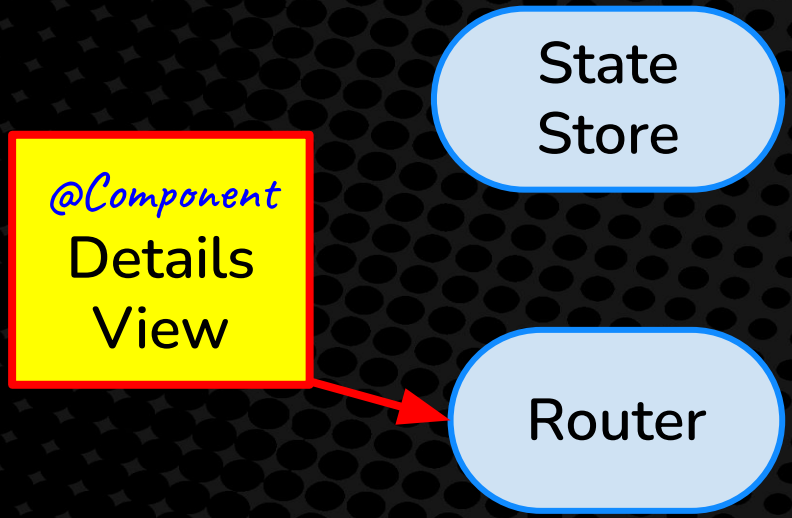
State store emits a new value



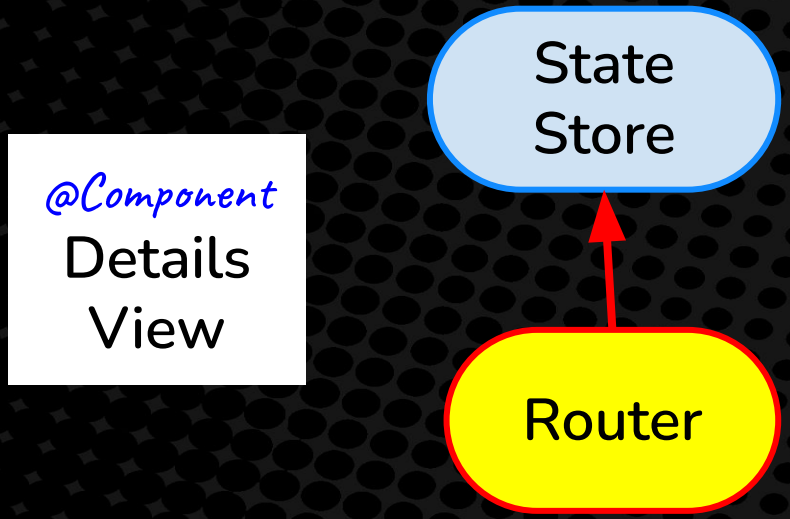

The components that care about selected product get updated
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products



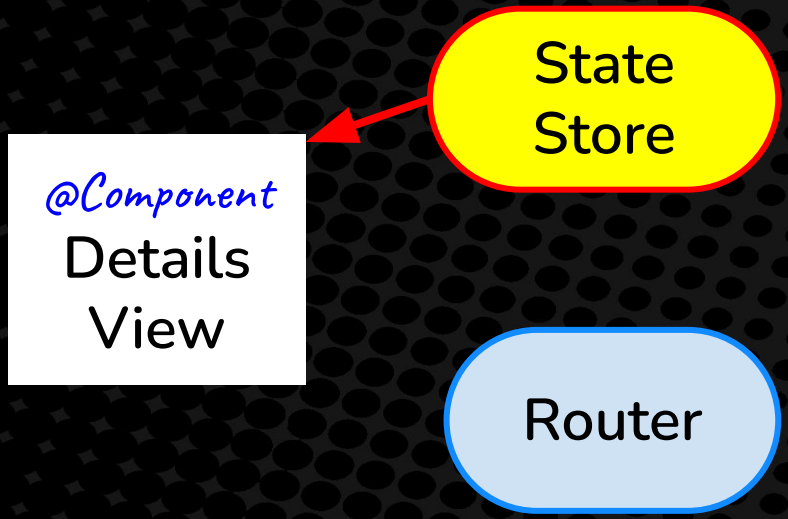
Notice that even within a single component we are still driving state the same way



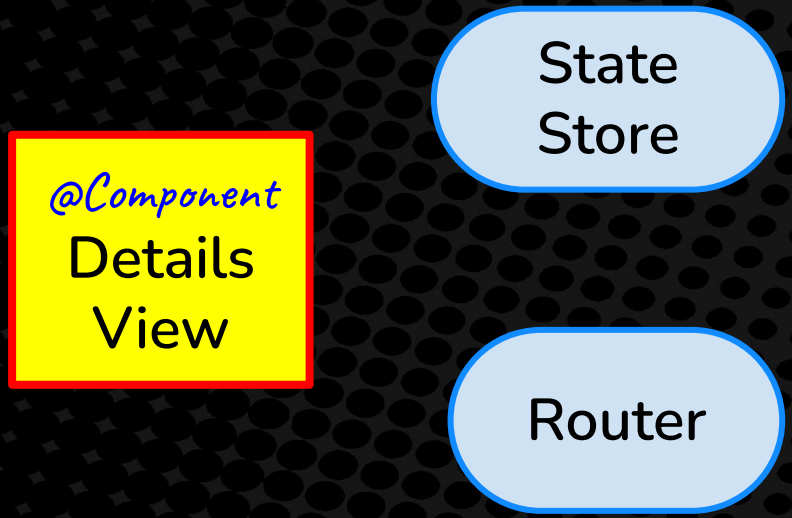
Notice that even within a single component we are still driving state the same way



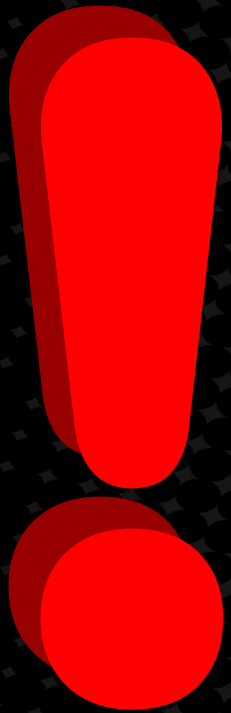
Notice that even within a single component we are still driving state the same way



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



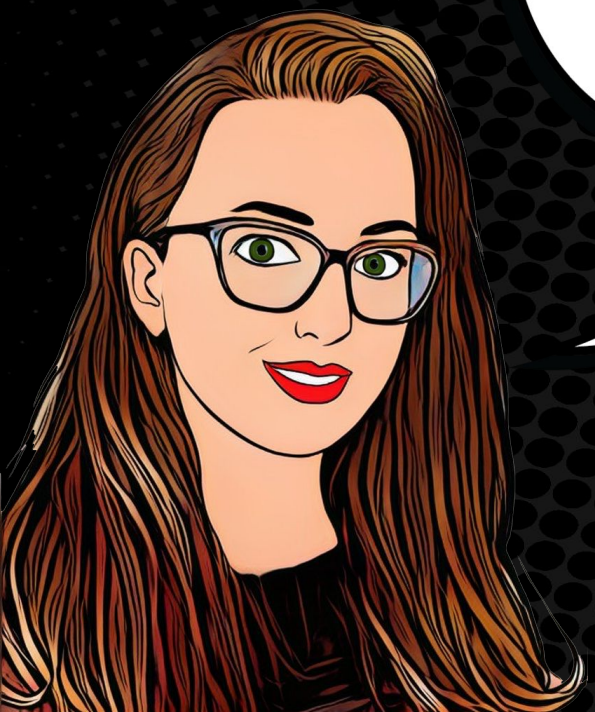
Be careful to maintain a single flow to set state

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

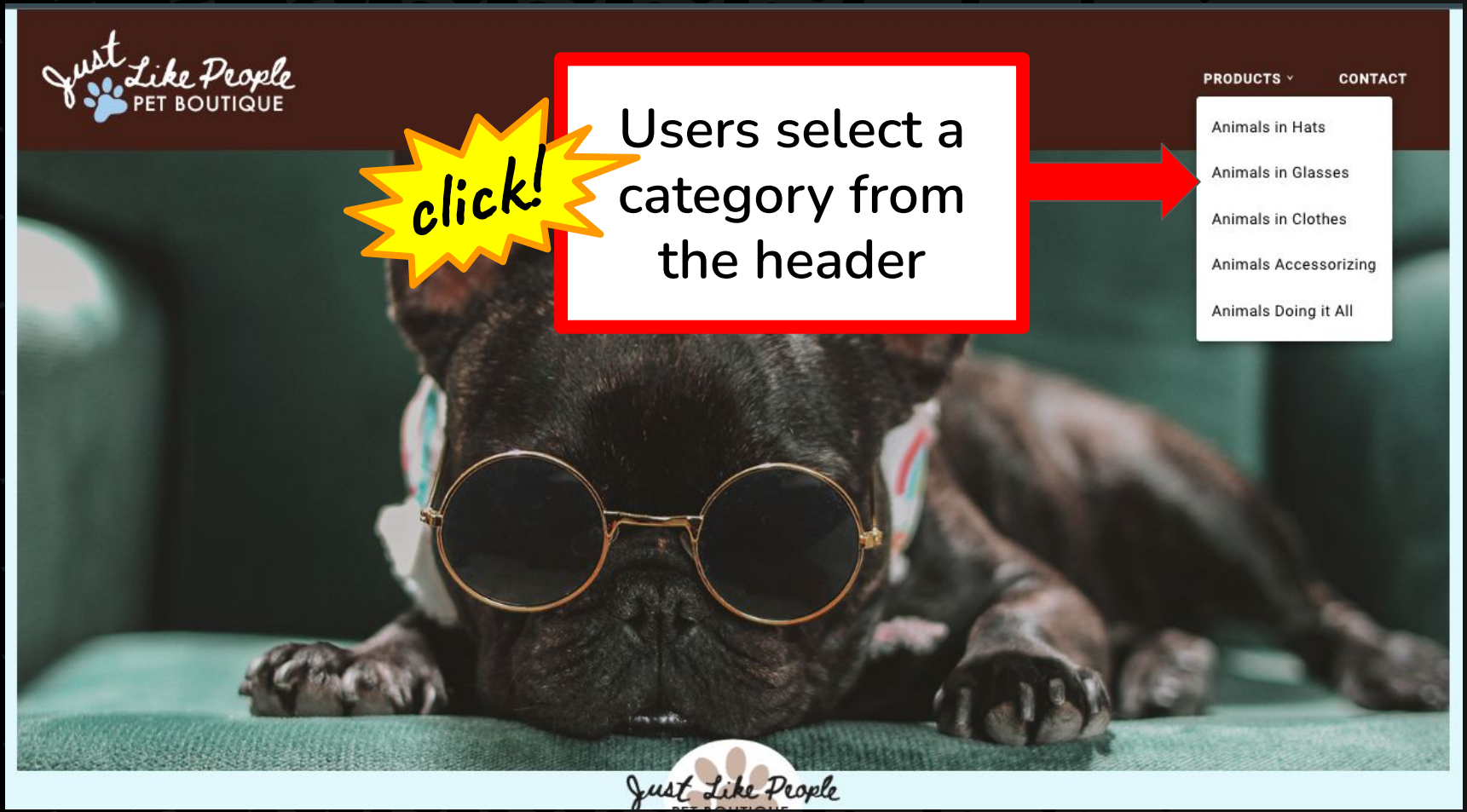
We want to avoid mixing responsibilities

```
selectProduct(id: string){
  this.productsService.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


Just Like People
PET BOUTIQUE
click!
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

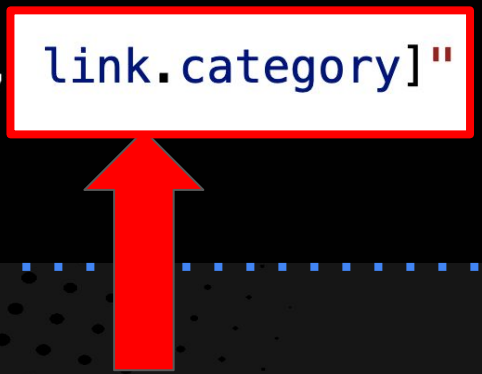
```
<button
  *ngFor="let link of LINKS"
  mat-menu-item
  [routerLink]="['/' + ROUTE_TOKENS.products, link.category]"
  routerLinkActive="router-link-active"
>{{ link.description }}</button>
```

In the header component template

```
<button
  *ngFor="let link of LINKS"
  mat-menu-item
  [routerLink]="['/' + ROUTE_TOKENS.products, link.category]"
  routerLinkActive="router-link-active"
>{{ link.description }}</button>
```

We change the route directly
from the template using routerLink

```
<button
  *ngFor="let link of LINKS"
  mat-menu-item
  [routerLink]="['/' + ROUTE_TOKENS.products, link.category]"
  routerLinkActive="router-link-active"
>{{ link.description }}</button>
```

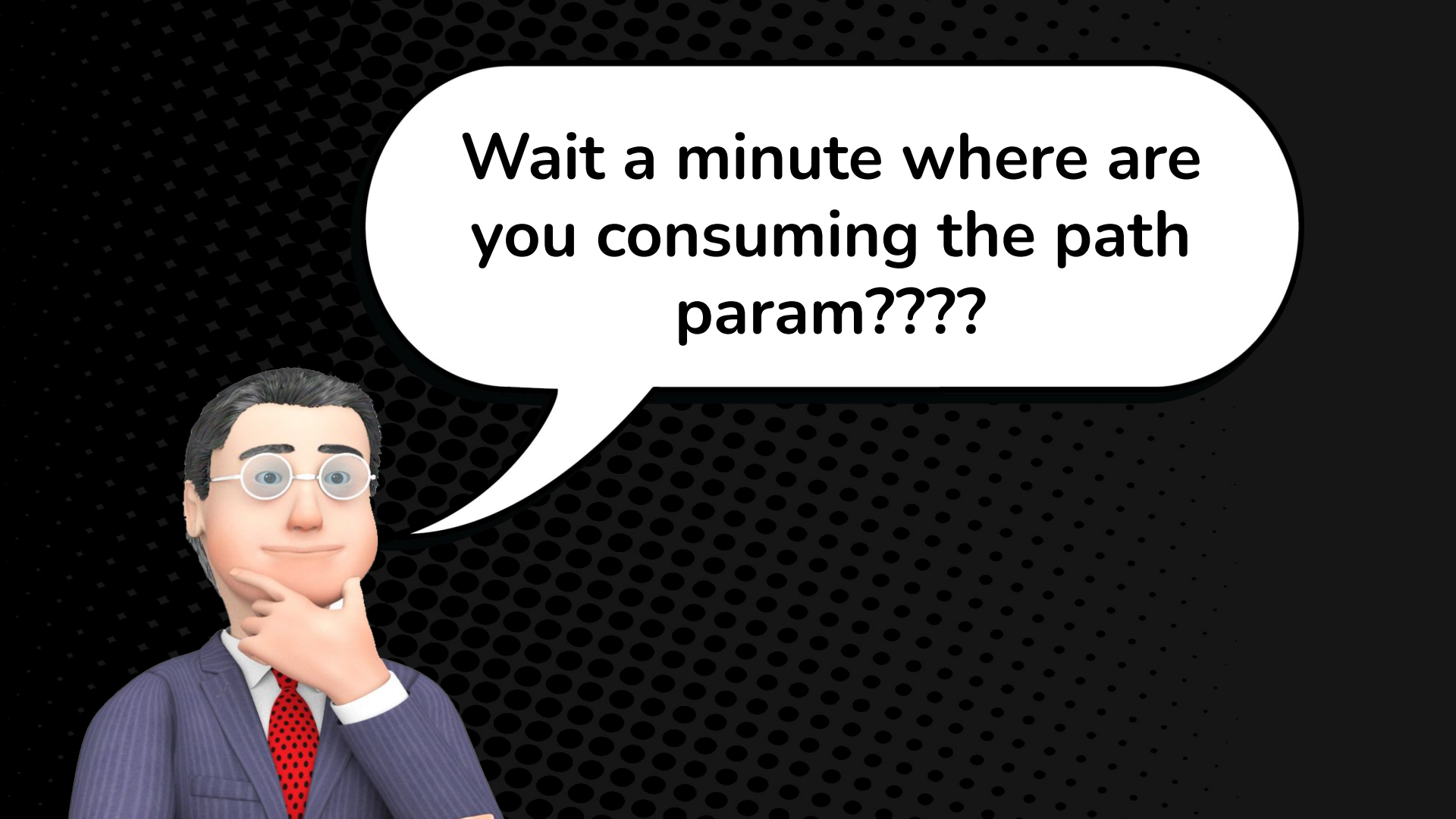


Here we are passing a path parameter called categoryId.

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);
}
```

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);
}
```

Oh hey a setter!

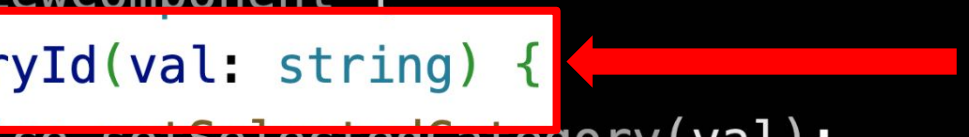
As of version 16 we can consume all route params, and resolver data in inputs!

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

To use this new feature update
the Router configuration

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);
}
```



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);
}
```

Inside the Component

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);
  }
```

Inside the ProductsService

setSelectedProduct acts as a setter on the private selectedCategory subject.

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

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

Inside the ProductsService

We chose behavior subject because it will always have a value selectedCategory.

Subjects are multicast observables.

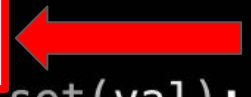
```
export class ProductService {
  private readonly selectedCategory = new BehaviorSubject<string>(Category.ALL);
  readonly selectedCategory$ = this.selectedCategory.asObservable();

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

Inside the ProductsService

We want the subject to be private so we expose the observable on a public property

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);
```



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);
```

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

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

Inside the ProductService

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

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.

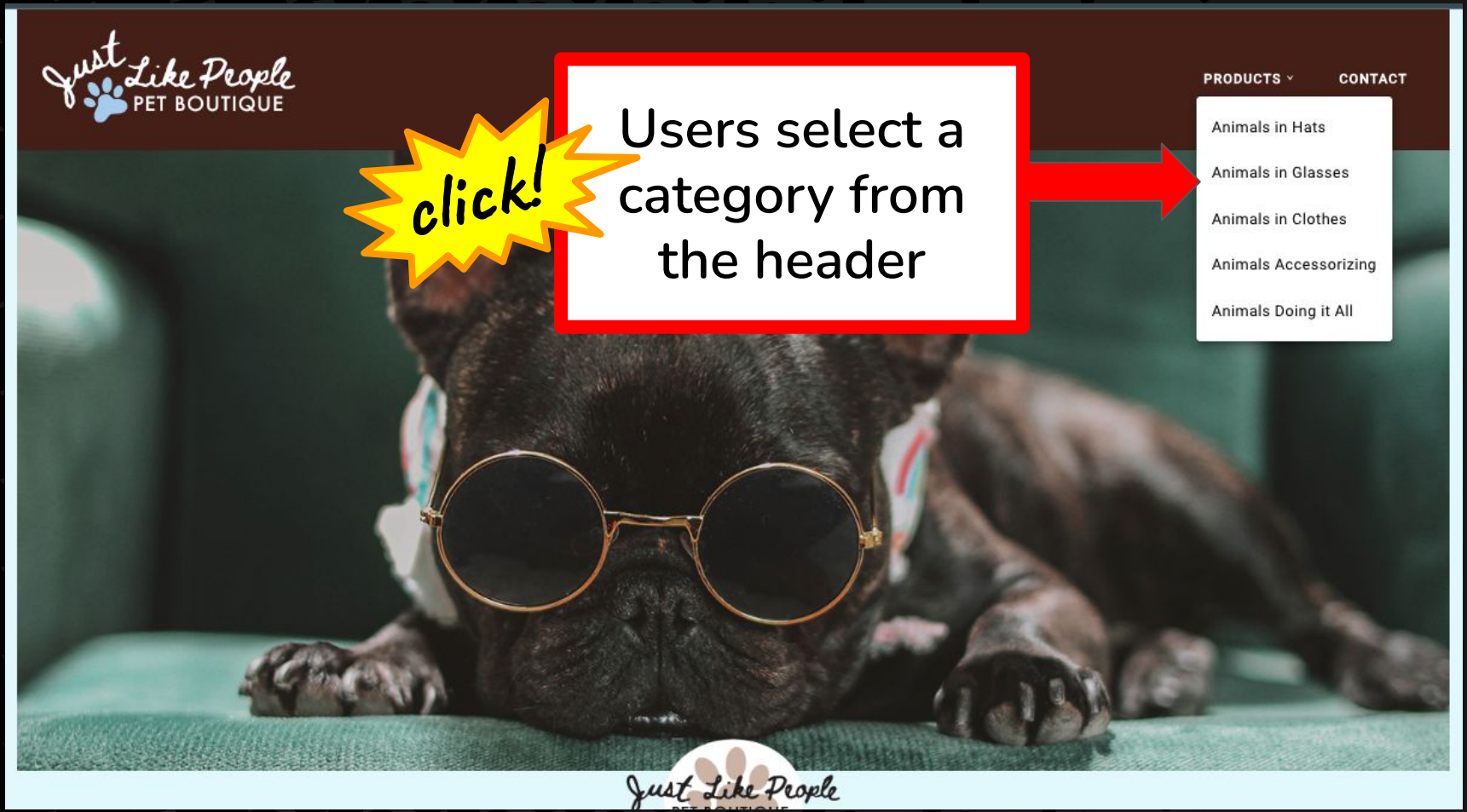
Let’s combine our patterns!

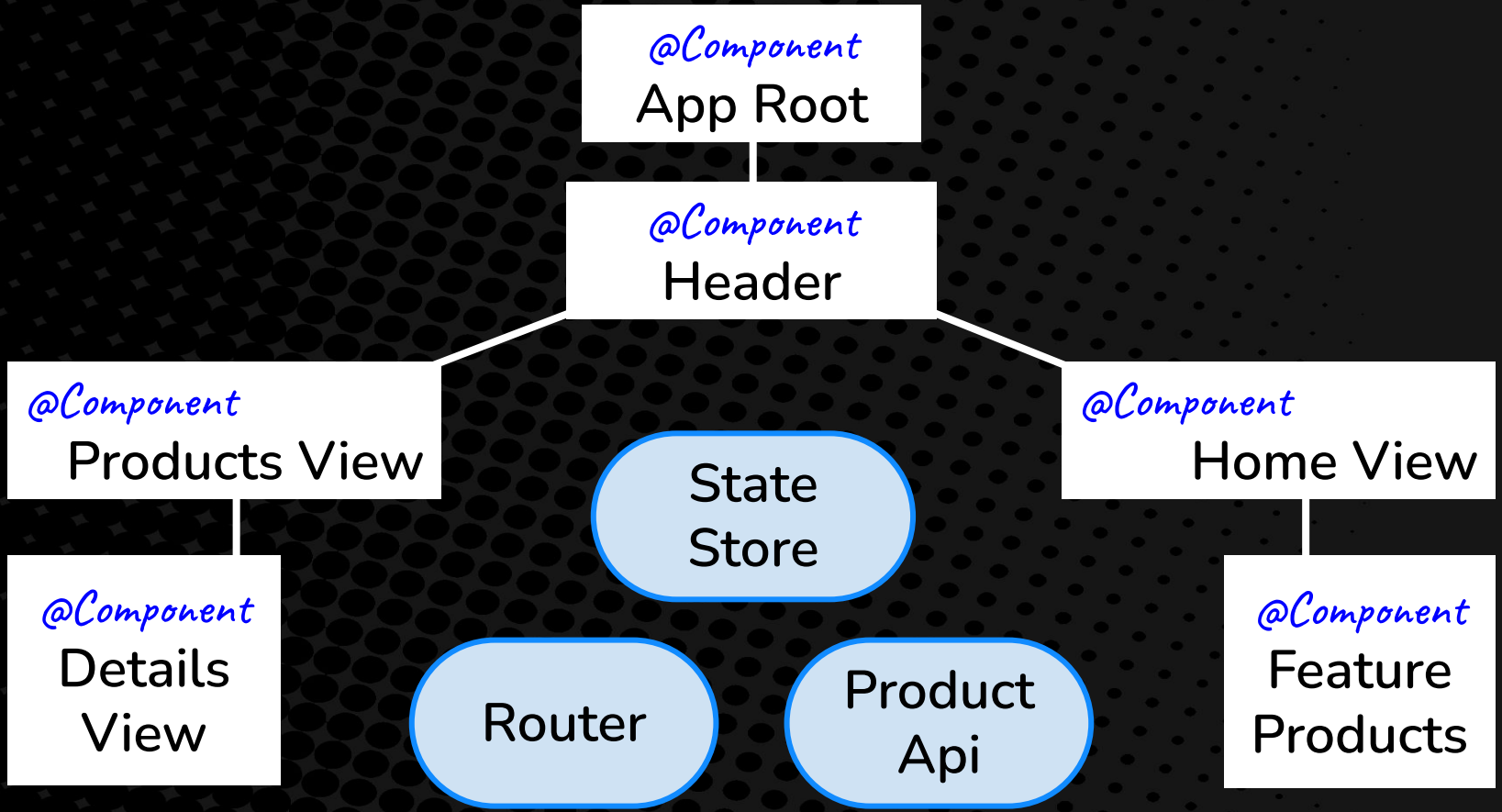
Input setters

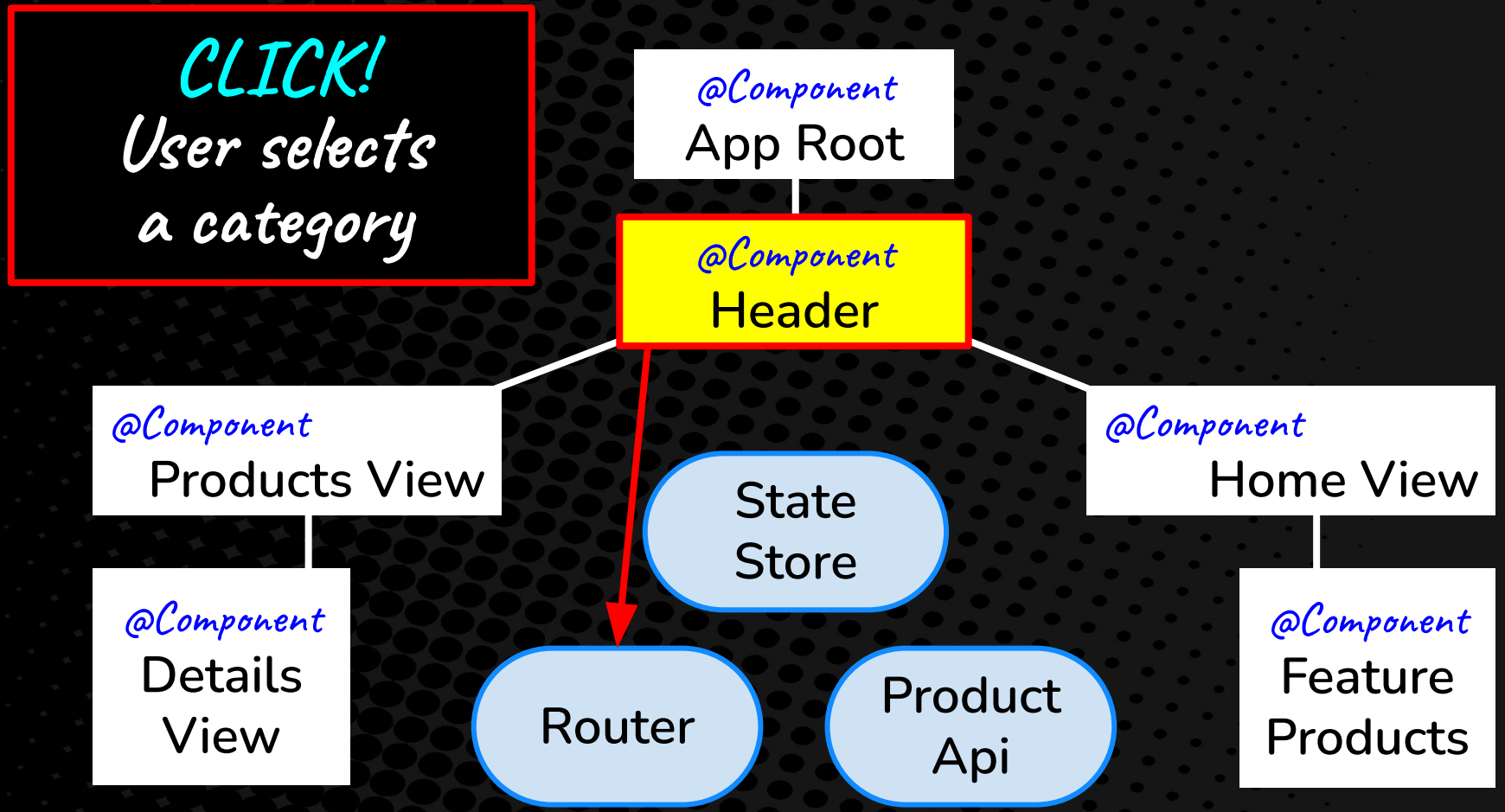
Emit events
where they happen

Compose data before it gets
to the consumers

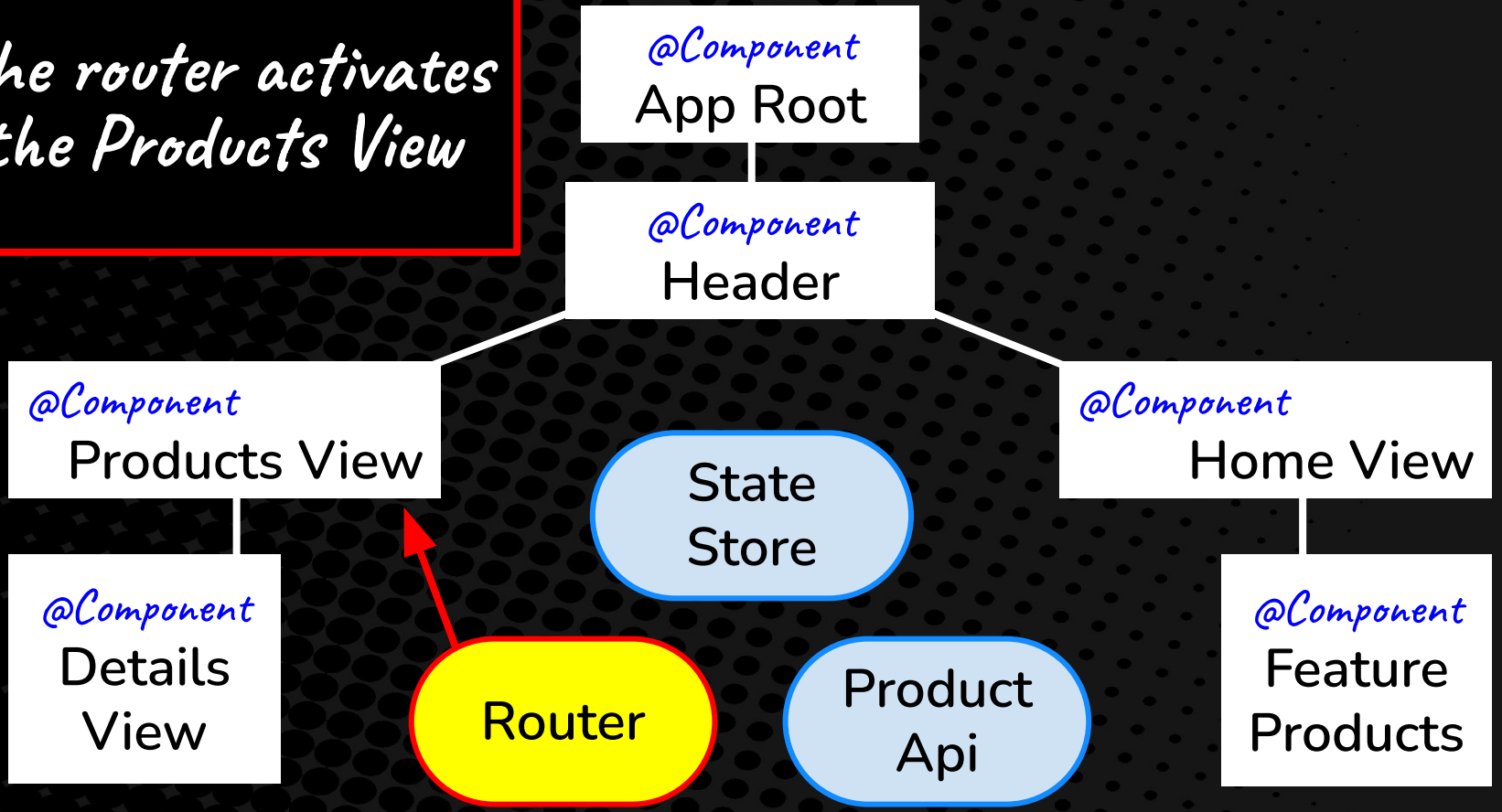
Let's see the state flow for selecting a category

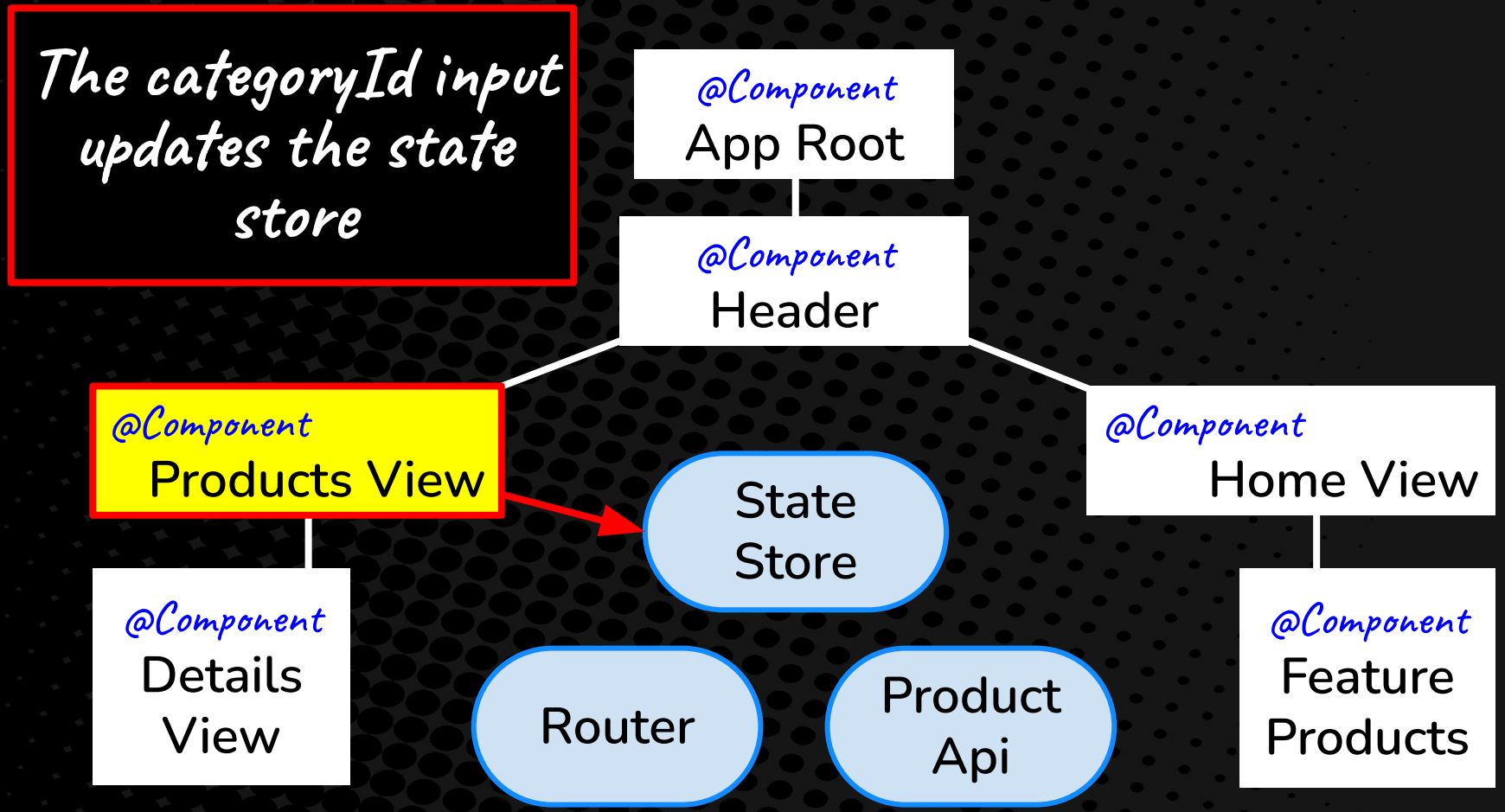

Just Like People
PET BOUTIQUE
click!
Users select a category from the header
PRODUCTS
CONTACT
Animals in Hats
Animals in Glasses
Animals in Clothes
Animals Accessorizing
Animals Doing it All
Just Like People


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

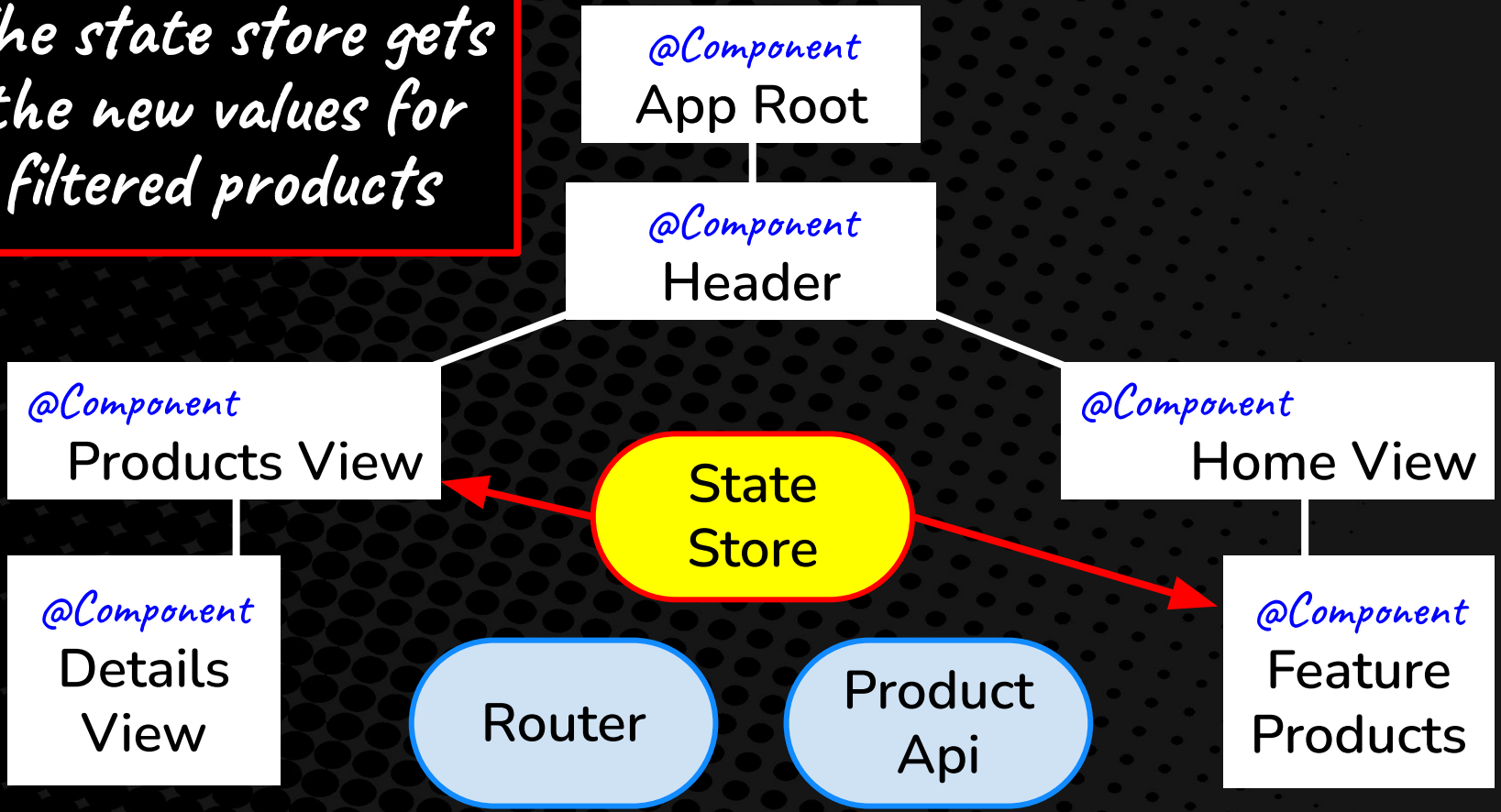

CLICK!
User selects a category
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

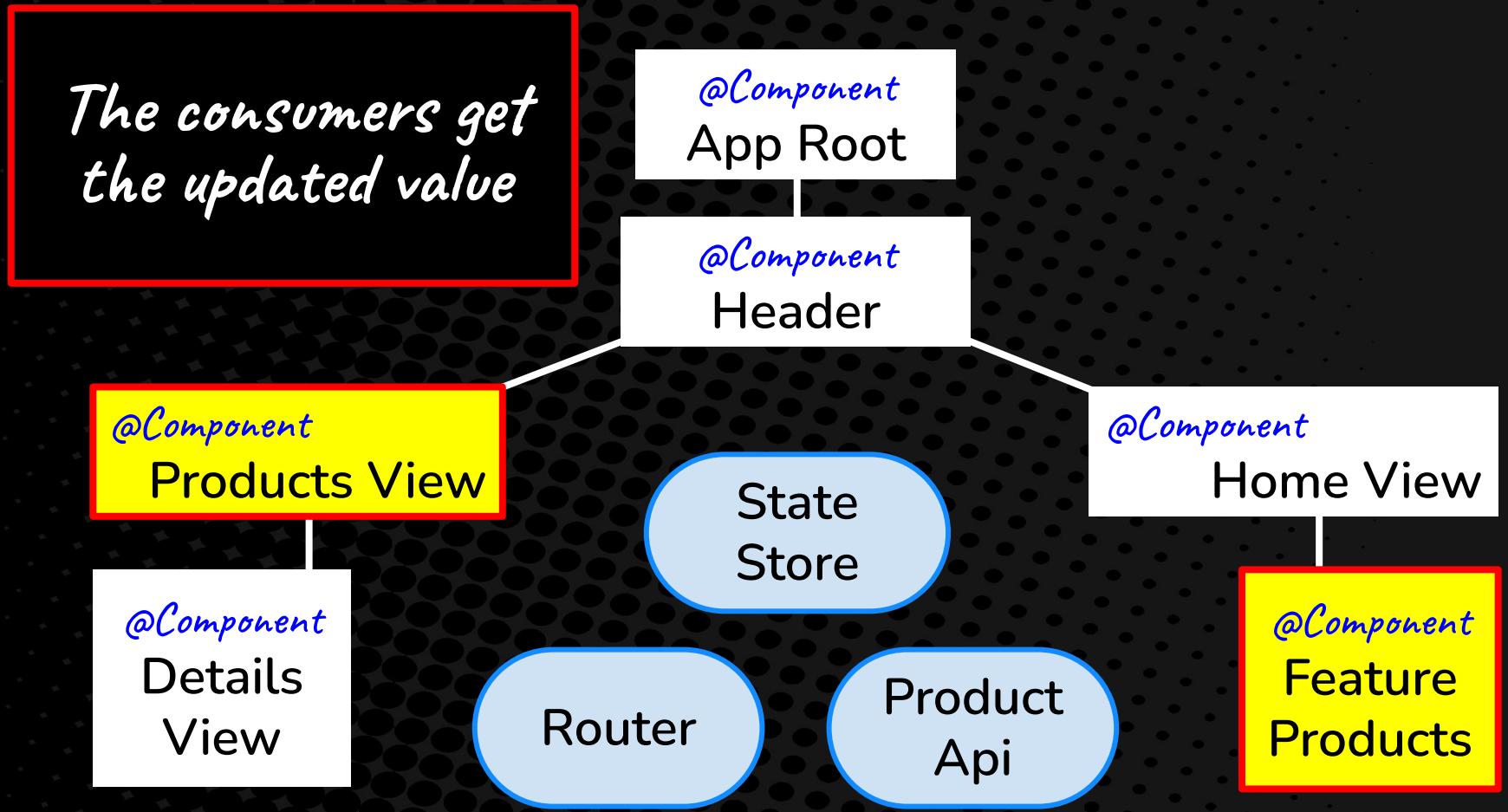
The router activates the Products View



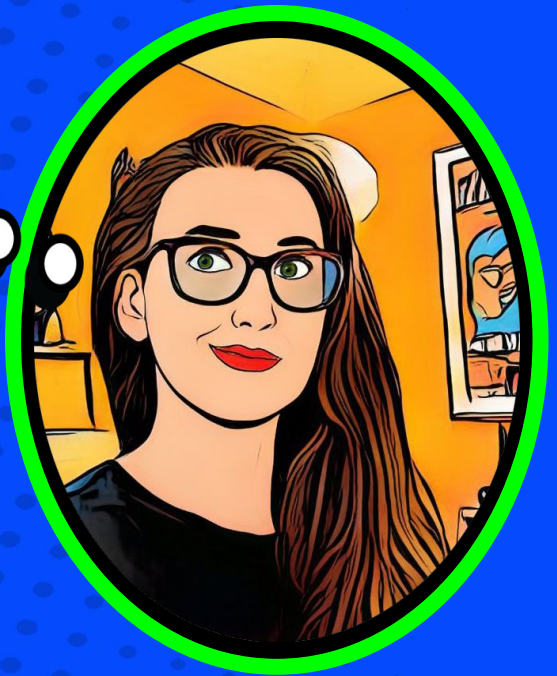

The categoryId input updates the state store
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

The state store gets the new values for filtered products




The consumers get the updated value
@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



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

Inside the ProductService

We compose the data inside a declared stream

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

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

Inside the ProductService

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

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

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

Inside the ProductService

Products are fetched and stored on initialization

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

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

Inside the Component

Declare a local property equal to the filteredProducts$ observable

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

Inside the Template

featuredProducts$ is subscribed on init using the Async pipe and new values are bound to the template

Show me the Signals

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

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

Inside the ProductSignalService

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;
```

Inside the Component

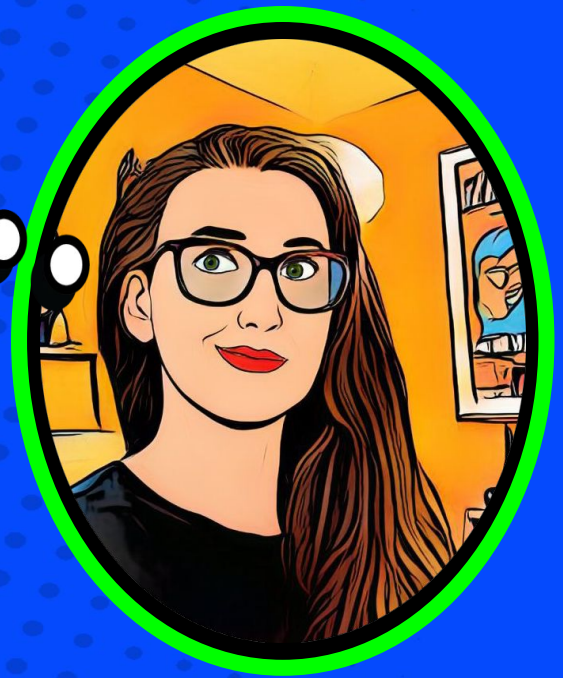
Declare a local property equal to the filteredProducts signal

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

Inside the Template

Use the value of the featuredProducts signal directly. New values automatically update the template

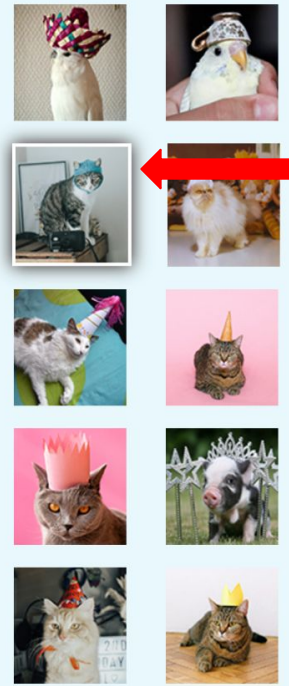
Let's practice again this time with the selected product flow



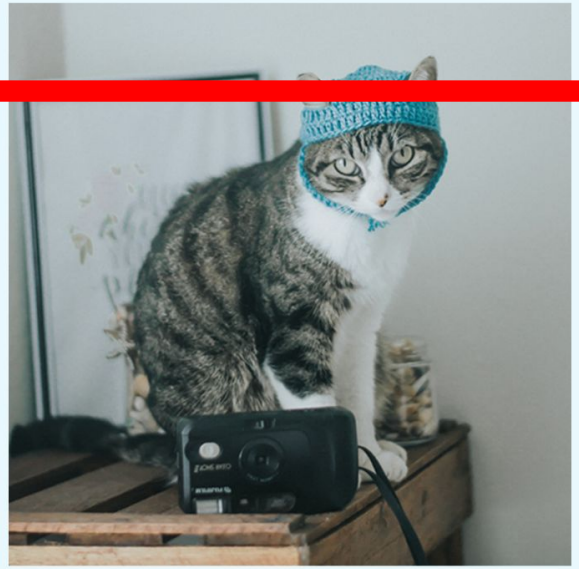
Just Like People PET BOUTIQUE

PRODUCTS CONTACT

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



Baby Blue Baby Bonnet for Big Baby Kitties

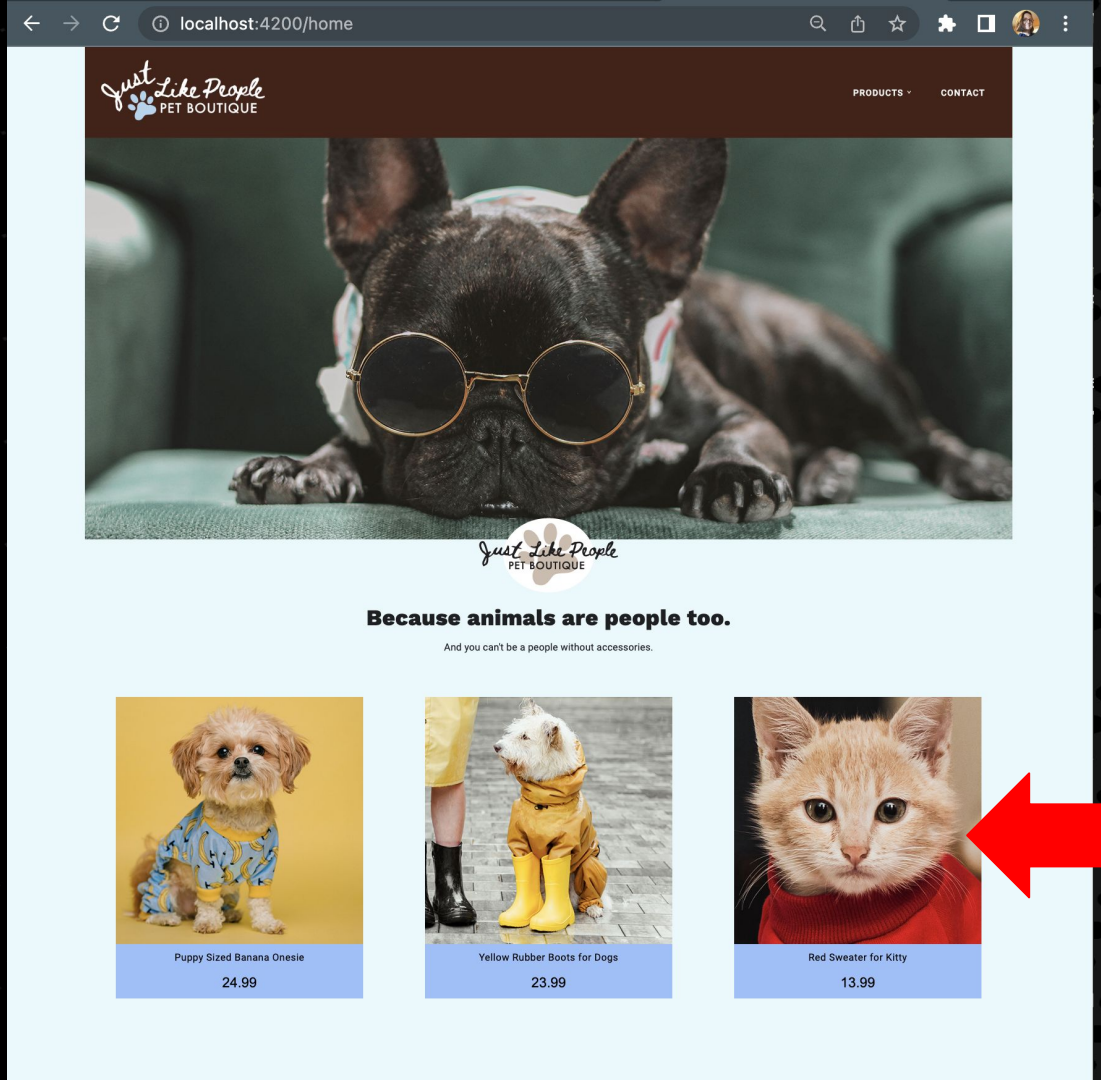


Users can select products from the side menu

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

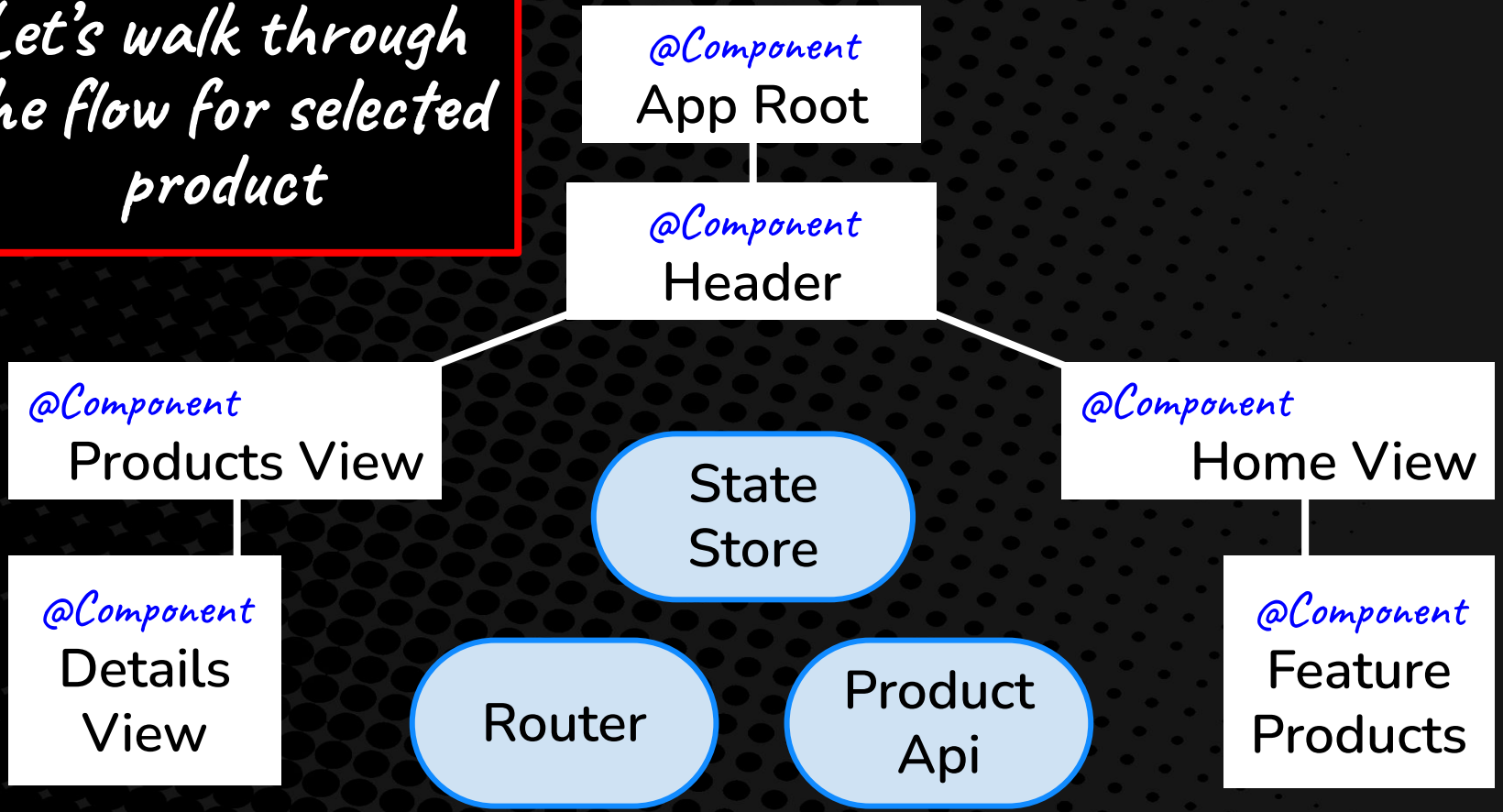
The header and the detail view need the selected product

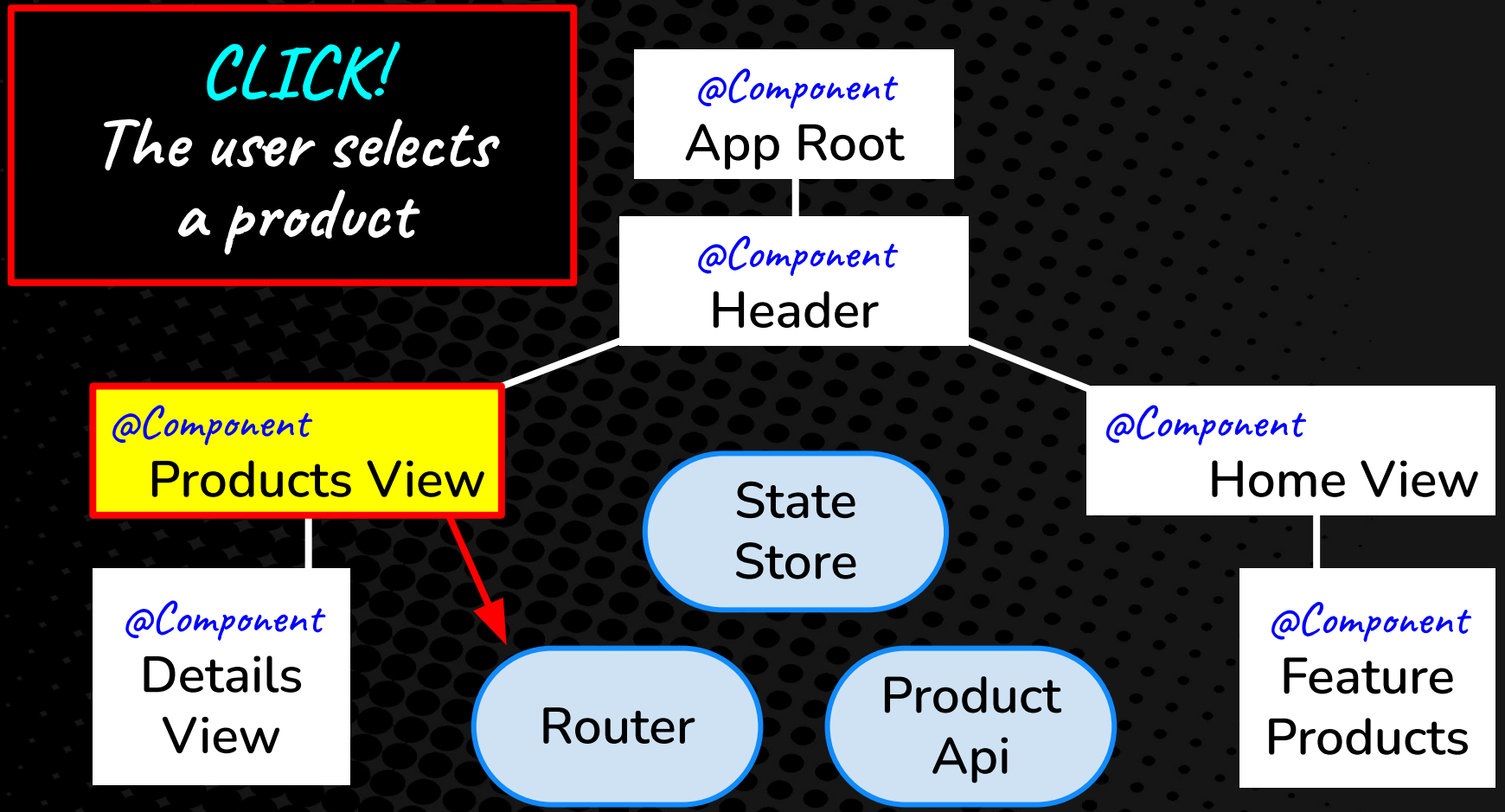
These rugged and durable overalls are great for little poochies who work hard and play harder! Now with 100% more pockets!

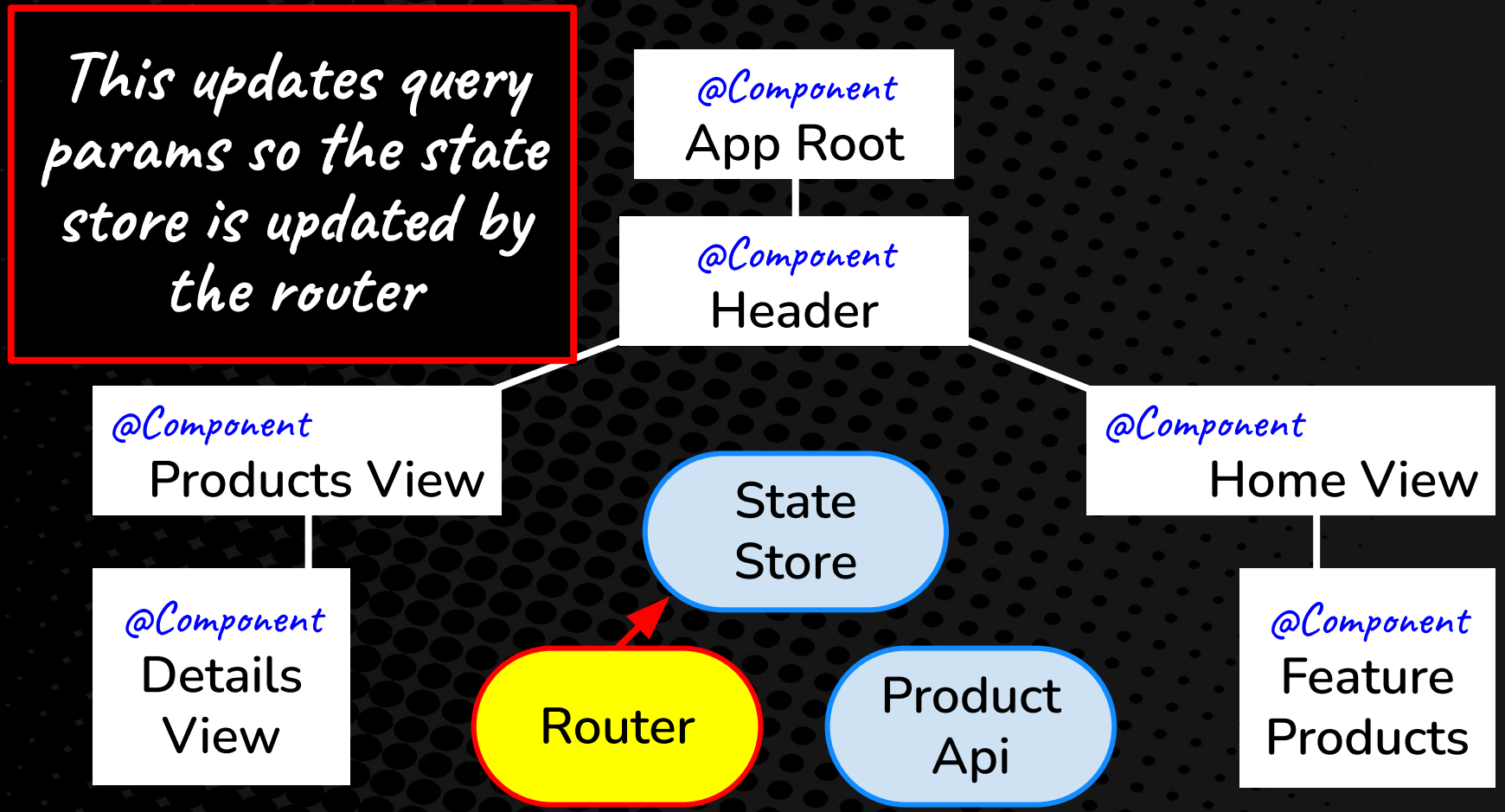
Tags: dog, work pants, overalls, denim, small dogs

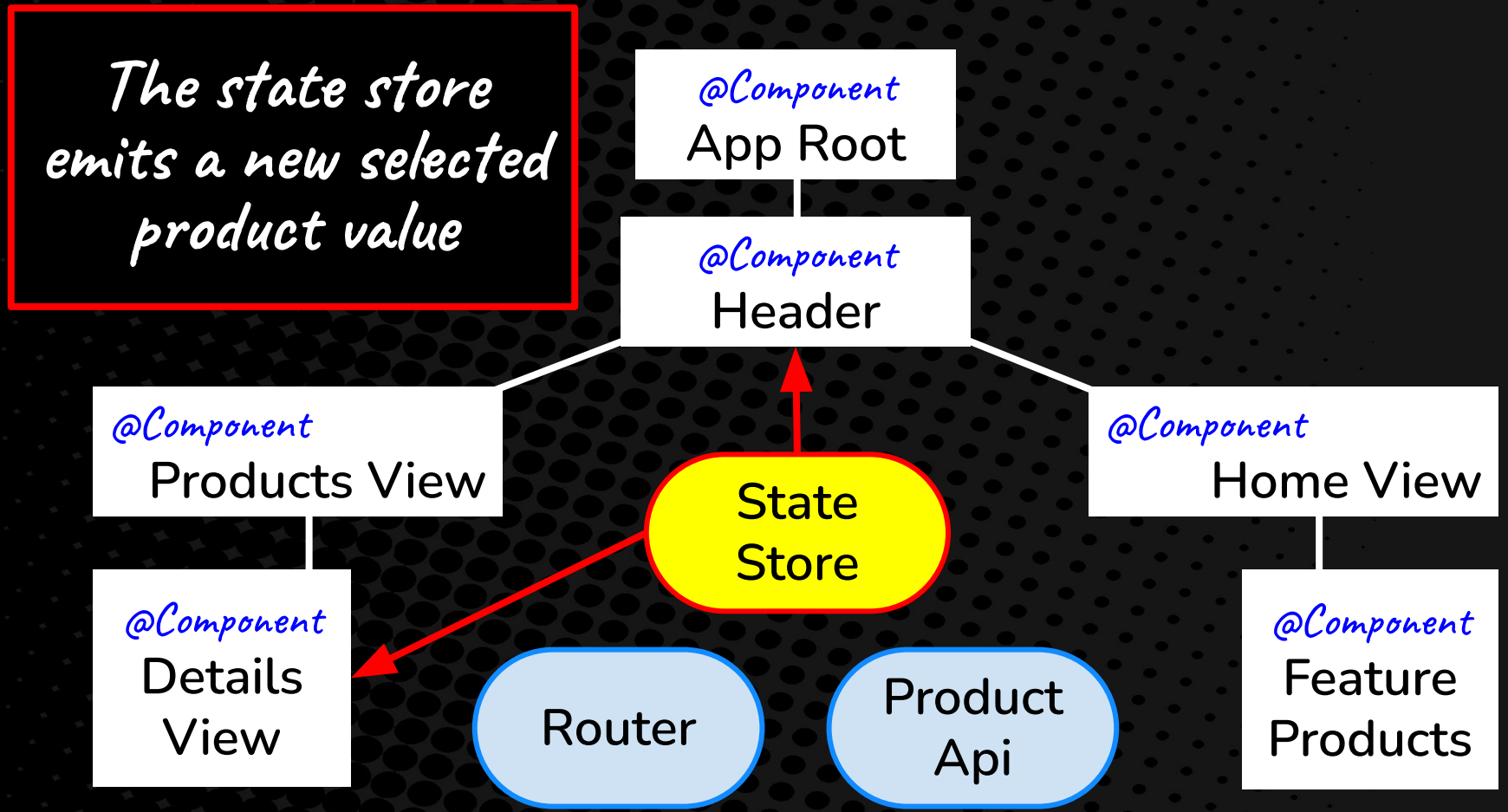
Price: 32.99 ***ON SALE!!***

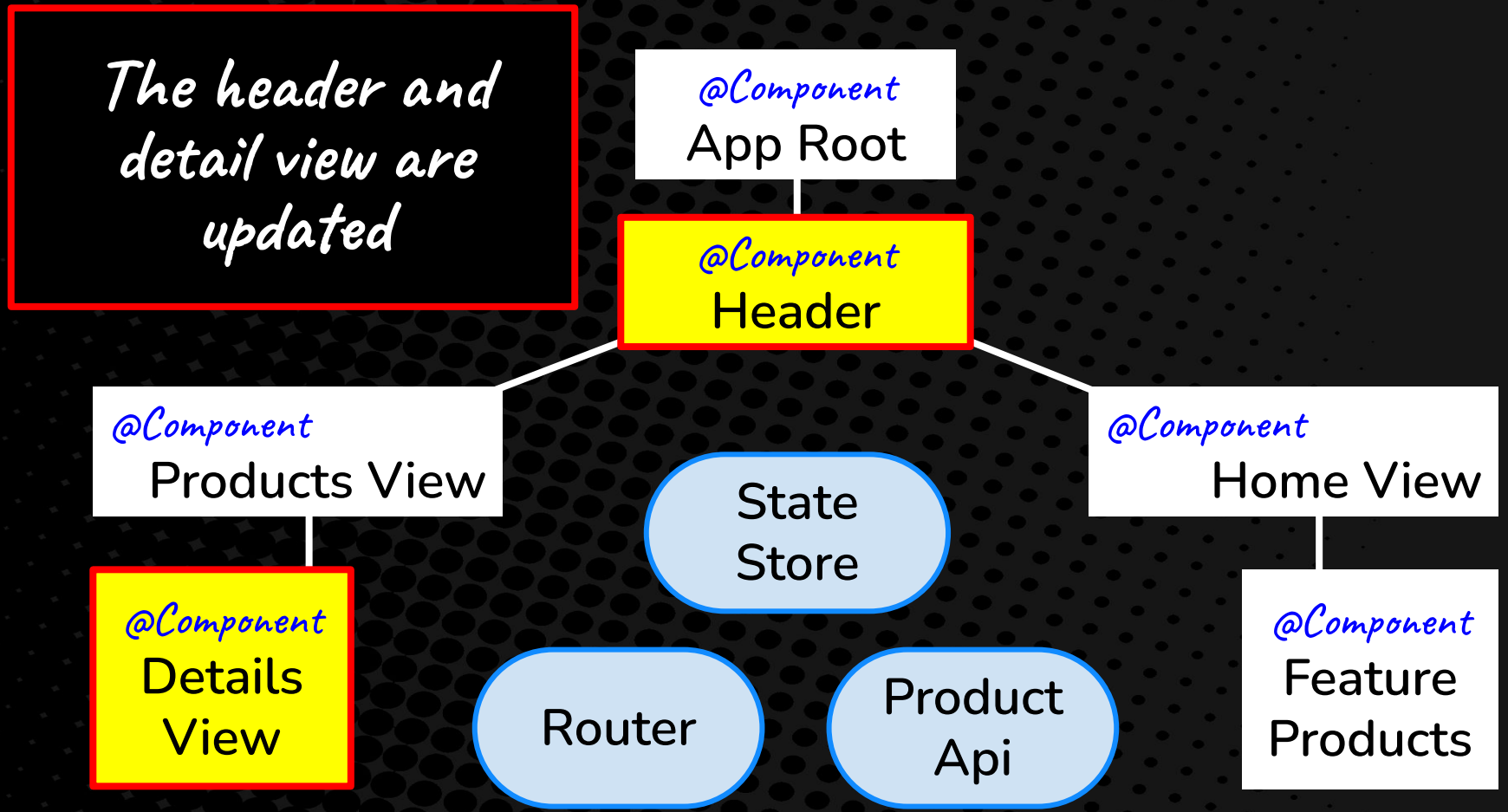
Let's walk through the flow for selected product



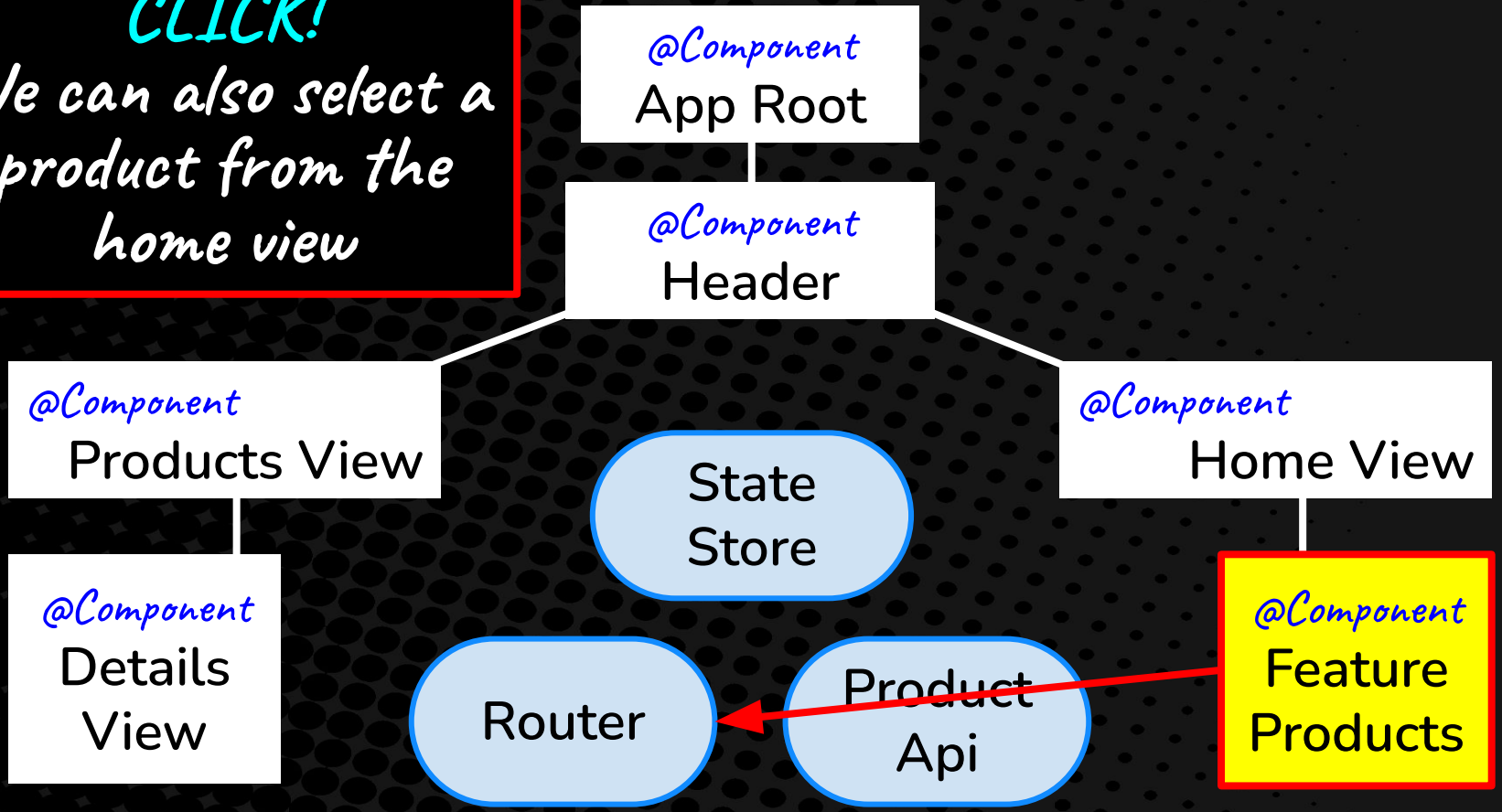

CLICK!
The user selects a product
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

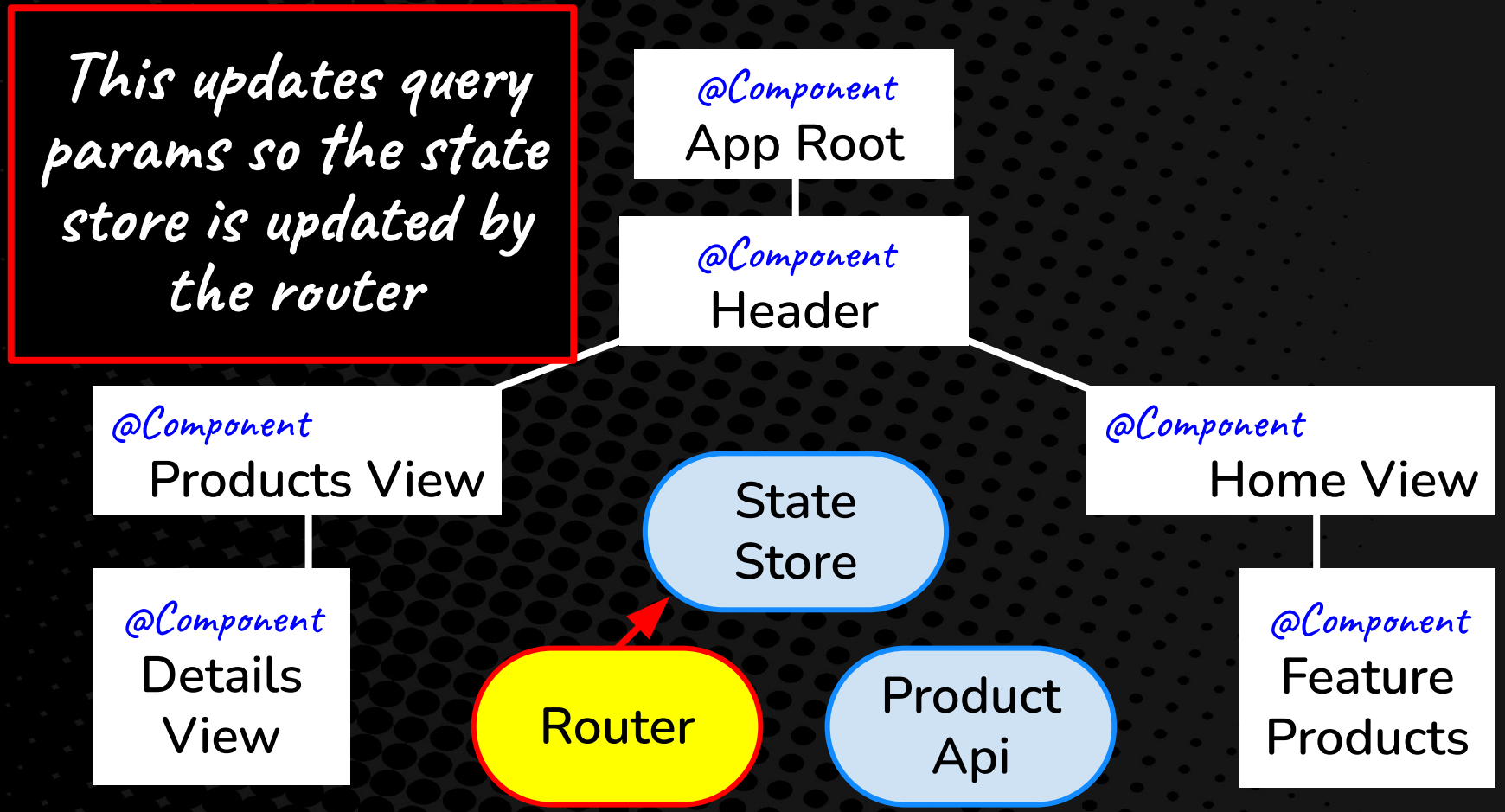

This updates query params so the state store is updated by the router
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
@Component
Details View
State Store
Router
Product Api
@Component
Feature Products

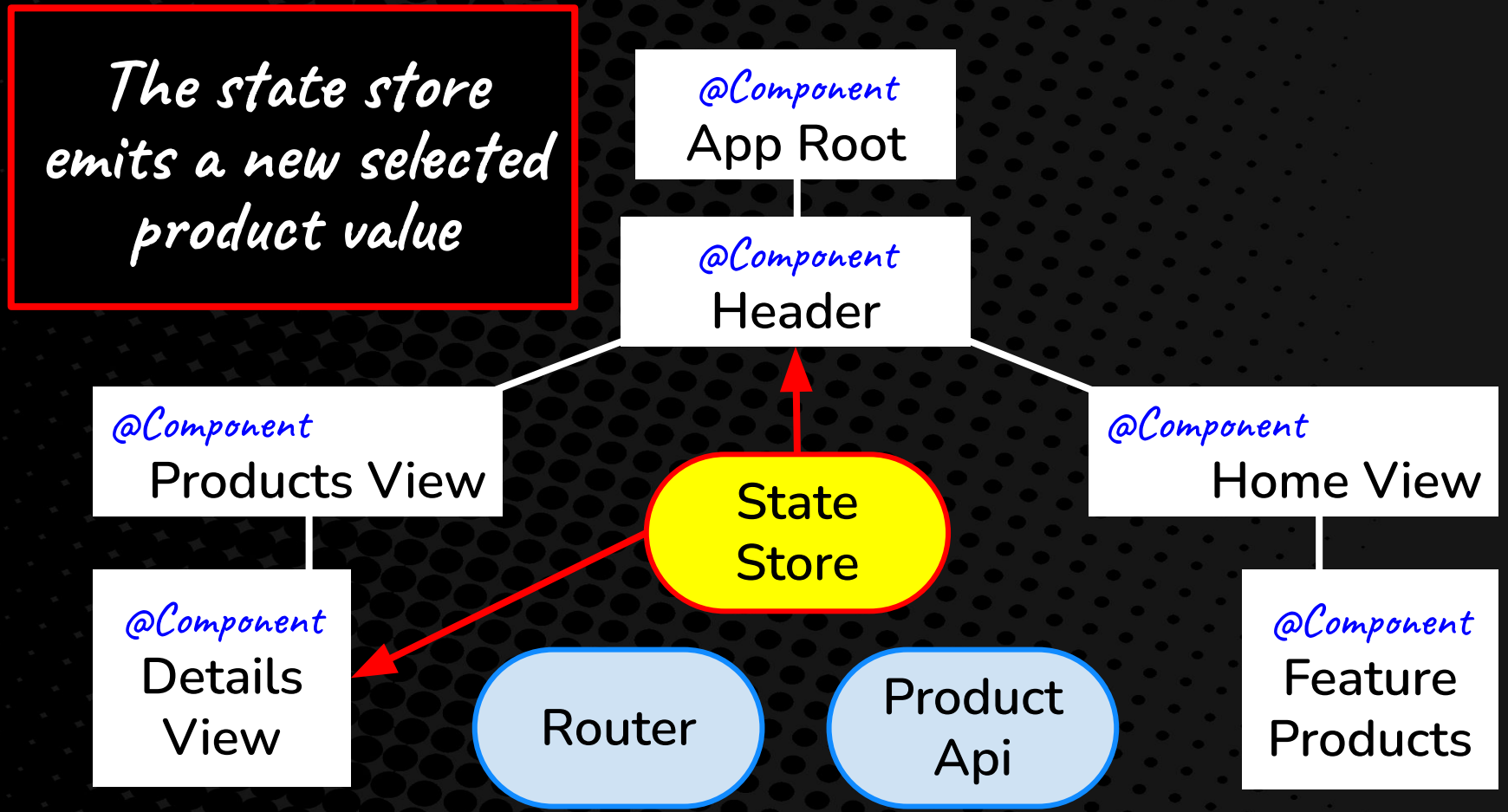

The state store emits a new selected product value
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

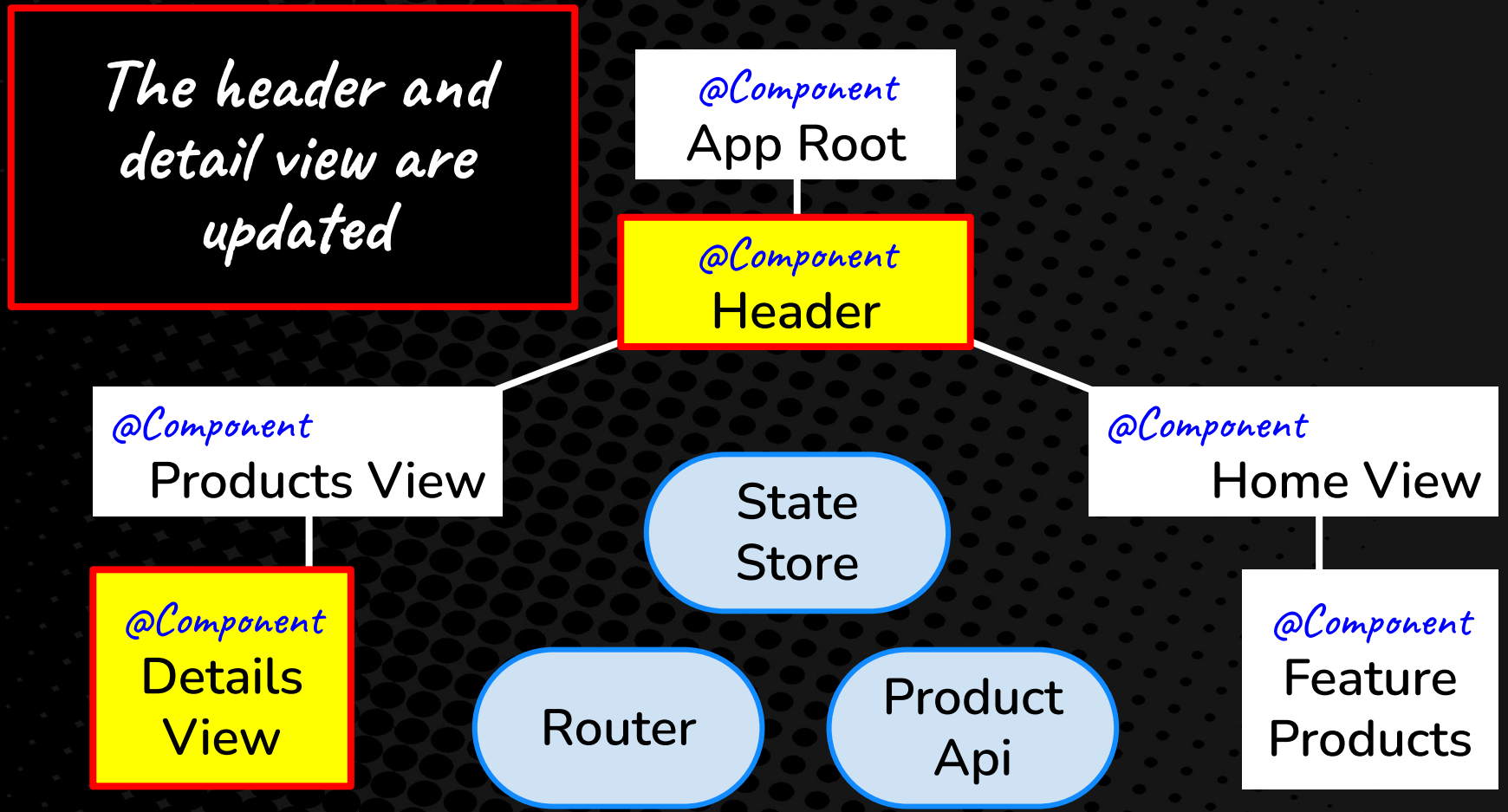

The header and detail view are updated
@Component
App Root
@Component
Header
@Component
Products View
@Component
Home View
State Store
@Component
Details View
Router
Product Api
@Component
Feature Products

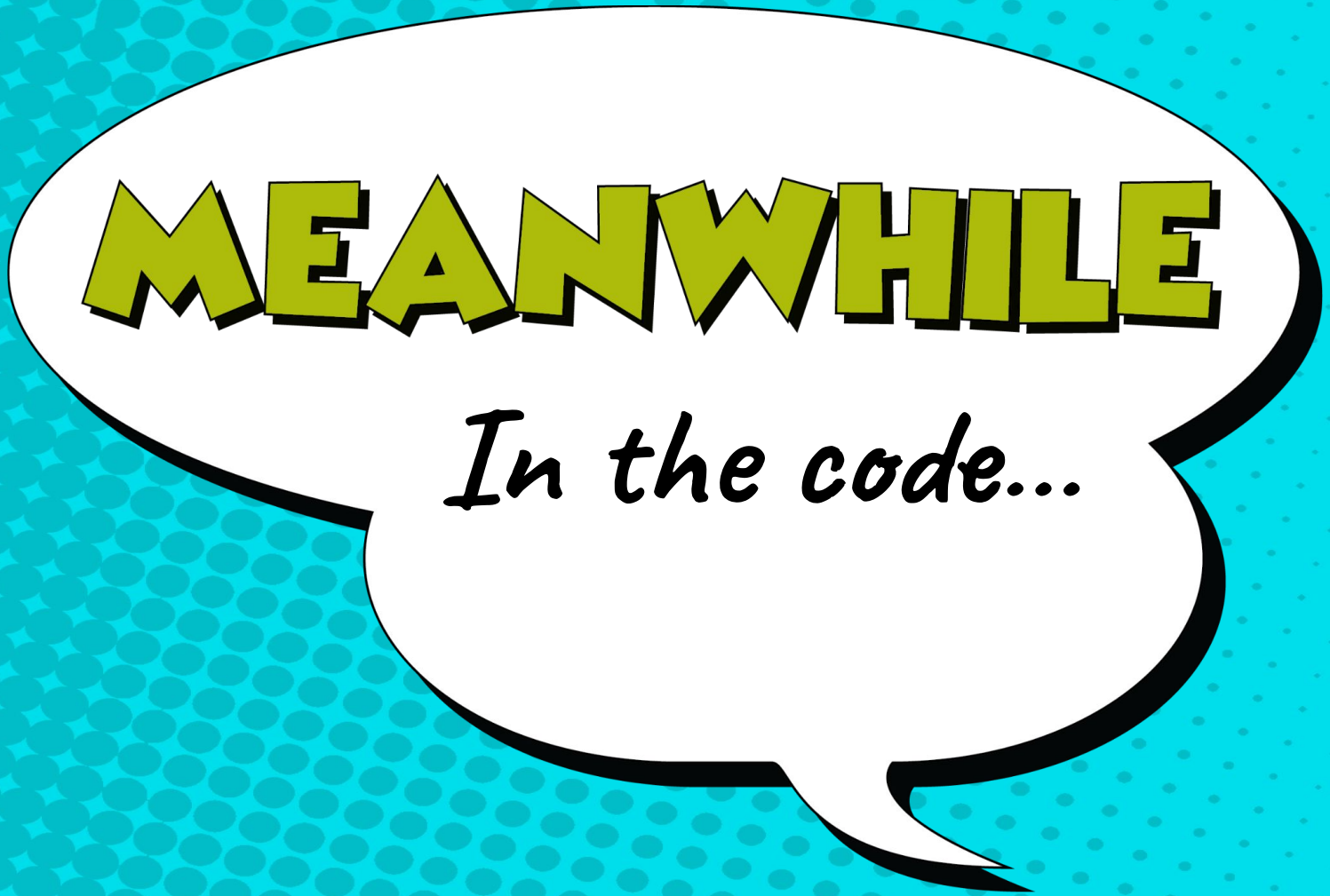
CLICK!
We can also select a product from the home view




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


MEANWHILE
In the code...

Query params are already globally available, so they are our single source of truth for our selected product

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

Inside either Template

Update the route with query params

Inside the Service

We access the query params directly off of the activated route

```
export class ProductSignalsService {
  private readonly productApiService = inject(ProductApiService);
  readonly products = toSignal(
    this.productApiService.getProducts$,
    { initialValue: [] }
  );

  private readonly activatedRoute = inject(ActivatedRoute);
  readonly selectedProductId = toSignal(
    this.activatedRoute.queryParams.pipe(
      map((params) => params['productId'])
      ));

  readonly selectedProduct = computed(() => {
      return this.products().find((product) =>
        product.id === this.selectedProductId());
  });
```

Inside the Service

toSignal handles subscribing

```
export class ProductSignalsService {
  private readonly productApiService = inject(ProductApiService);
  readonly products = toSignal(
    this.productApiService.getProducts$,
    { initialValue: [] }
  );

  private readonly activatedRoute = inject(ActivatedRoute);
  readonly selectedProductId = toSignal(
    this.activatedRoute.queryParams.pipe(
      map((params) => params['productId'])
      ));

  readonly selectedProduct = computed(() => {
      return this.products().find((product) =>
        product.id === this.selectedProductId());
  });
```

Inside the Service

Any time the selected id changes the call back function reruns

```
export class ProductSignalsService {
  private readonly productApiService = inject(ProductApiService);
  readonly products = toSignal(
    this.productApiService.getProducts$,
    { initialValue: [] }
  );

  private readonly activatedRoute = inject(ActivatedRoute);
  readonly selectedProductId = toSignal(
    this.activatedRoute.queryParams.pipe(
      map((params) => params['productId'])
      ));

  readonly selectedProduct = computed(() => {
      return this.products().find((product) =>
        product.id === this.selectedProductId());
  });
```

```
private readonly productService = inject(ProductSignalsService);

readonly selectedProduct = this.productService.selectedProduct;
```

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

```
<ng-container *ngIf="selectedProduct() as selectedProduct">
  <div *ngIf="selectedProduct.onSale" class="sale-banner">
    <span>{{ selectedProduct.title }}</span>
    <span>{{ '  On Sale Now!!!' | uppercase }}</span>
  </div>
```

We use the value of the selectedProduct signal property in both templates

```
<div class="detail" *ngIf="selectedProduct() 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: <span class="tag" *ngFor="let tag of selectedProduct.tags">{{
```

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 get the same data 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!!*

Together, all three patterns, define a holistic data flow that can be followed throughout the application.

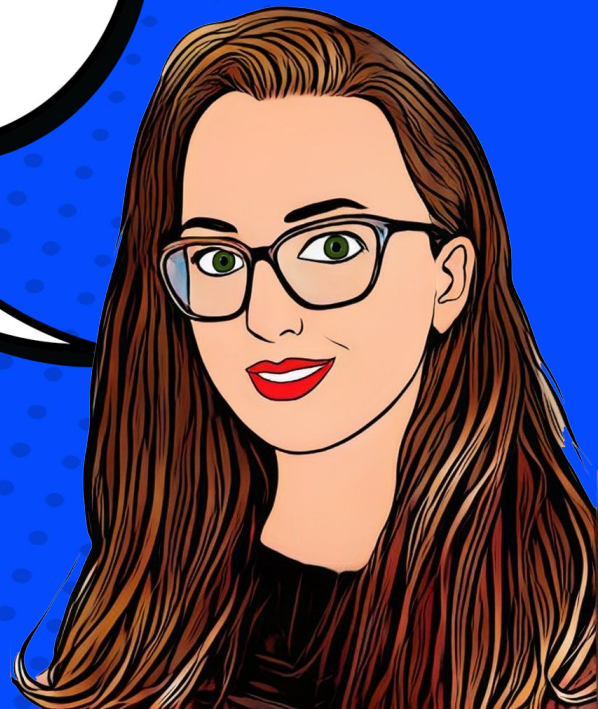
To put it another way...

Orchestrating a clear flow of data through the application

Creating and maintaining 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