

Params vs searchParams in Next.js

Working and Uses of params and searchParams in Next.js 15.

In **Next.js 15**, routing has evolved with the **App Router**, and handling **dynamic routes**, **route parameters** (`params`), and **query parameters** (`searchParams`) is an essential part of building modern applications.

Let's break it down:

`params` in Next.js 15

✓ What is `params` ?

`params` refers to **dynamic route segments** in your URL — values extracted from the file or folder names defined using `[param]` syntax in the `/app` directory.

Folder Structure Example

```
bash

/app
  /product
    /[id]
      page.tsx
```

Example URL

```
bash

/product/42
```

Accessing `params`

In `page.tsx` or a layout/server component, you can access `params` like this:

```
tsx

// app/product/[id]/page.tsx
import { FC } from 'react';
```

```
interface ProductPageProps {
  params: { id: string };
}

const ProductPage: FC<ProductPageProps> = ({ params }) => {
  return <h1>Product ID: {params.id}</h1>;
};

export default ProductPage;
```

params.id → 42

searchParams in Next.js 15

✓ What is searchParams ?

searchParams refers to **query parameters** in the URL after the `?`. These are accessible in server components or page components.

Example URL

bash

/product/42?sort=asc&filter=popular

Accessing searchParams

tsx

```
// app/product/[id]/page.tsx
interface ProductPageProps {
  params: { id: string };
  searchParams: { [key: string]: string | string[] | undefined };
}

const ProductPage = ({ params, searchParams }: ProductPageProps) => {
  return (
    <>
      <h1>Product ID: {params.id}</h1>
      <p>Sort: {searchParams.sort}</p>
    </>
  );
}
```

```
<p>Filter: {searchParams.filter}</p>
</>
);
};

export default ProductPage;
```

`searchParams.sort` → `"asc"`

`searchParams.filter` → `"popular"`

✓ Use Cases

Use Case	Use <code>params</code>	Use <code>searchParams</code>
Product ID, User ID, Slugs in URL	✓	✗
Filters, Sorting, Pagination	✗	✓
Blog Post Slug: <code>/blog/[slug]</code>	✓	✗
Search query: <code>/search?q=nextjs</code>	✗	✓

🧠 Notes

- `params` and `searchParams` are only available in **App Router** (not in `pages` directory).
- They are available in:
 - `page.tsx`
 - `layout.tsx`
 - `loading.tsx` (as of newer versions)
 - Server components
- They are not automatically available in client components. You must use hooks like:

💡 For Client Components

Use:

```
tsx

'use client'
import { useSearchParams } from 'next/navigation';

const MyComponent = () => {
  const searchParams = useSearchParams();
  const sort = searchParams.get('sort');

  return <div>Sort Order: {sort}</div>;
};
```

✅ Summary

Feature	params	searchParams
Type	Dynamic route segments	URL query parameters (?key=value)
Access in	Server components, pages/layouts	Server components, pages/layouts
Client support	❌ (use hook for searchParams)	✅ (with useSearchParams)
Common use	/product/[id]	/search?q=nextjs&sort=asc

Would you like a practical project example using both?

yes

Great! Let's walk through a **mini-project** in **Next.js 15** using both `params` and `searchParams` .



Project: Product Detail Page with Filtering



Features:

- Dynamic product route using `params` (`/products/[id]`)
 - Filtering reviews by rating using `searchParams` (`?rating=5`)
-



Folder Structure

```
bash

/app
/products
/[id]
  page.tsx
```



Example URLs

- `/products/101` → Show product 101
 - `/products/101?rating=5` → Show product 101 with only 5-star reviews
-



Code: `app/products/[id]/page.tsx`

```
tsx

// app/products/[id]/page.tsx

interface PageProps {
  params: { id: string };
  searchParams: { rating?: string };
}

const fakeReviews = [
  { id: 1, rating: 5, comment: "Excellent!" },
```

```

{ id: 2, rating: 4, comment: "Very good!" },
{ id: 3, rating: 5, comment: "Perfect!" },
{ id: 4, rating: 3, comment: "Okay" },
];

export default function ProductPage({ params, searchParams }: PageProps) {
  const { id } = params;
  const ratingFilter = searchParams.rating ? parseInt(searchParams.rating) : null;

  const filteredReviews = ratingFilter
    ? fakeReviews.filter((r) => r.rating === ratingFilter)
    : fakeReviews;

  return (
    <main>
      <h1>Product ID: {id}</h1>

      <h2>Reviews {ratingFilter ? `(Rating: ${ratingFilter} stars)` : ""}</h2>
      <ul>
        {filteredReviews.map((review) => (
          <li key={review.id}>
            <strong>{review.rating}★</strong> — {review.comment}
          </li>
        ))}
      </ul>
    </main>
  );
}

```

Testing the Page

✓ URL: `/products/101`

Output:

diff

Product ID: 101

Reviews:

- 5★ Excellent!

- 4★ Very good!

- 5★ Perfect!

- 3★ Okay

✓ URL: `/products/101?rating=5`

Output:

diff

Product ID: 101

Reviews (Rating: 5 stars):

- 5★ Excellent!

- 5★ Perfect!

💡 Bonus: Add a client component to change filters dynamically

Create a simple dropdown:

tsx

```
// app/products/[id]/Filter.tsx
```

```
'use client'
```

```
import { useRouter, useSearchParams } from 'next/navigation';
```

```
export default function Filter() {
```

```
  const router = useRouter();
```

```
  const searchParams = useSearchParams();
```

```
  const handleChange = (e: React.ChangeEvent<HTMLSelectElement>) => {
```

```
    const value = e.target.value;
```

```
    const current = new URLSearchParams(Array.from(searchParams.entries()));
```

```
    if (value) {
```

```
      current.set('rating', value);
```

```
    } else {
```

```
      current.delete('rating');
```

```
    }
```

```
    router.push(`?${current.toString()}`);
```

```
  };
```

```

return (
  <select onChange={handleChange} defaultValue={searchParams.get('rating') || ''}>
    <option value="">All Ratings</option>
    <option value="5">5 Stars</option>
    <option value="4">4 Stars</option>
    <option value="3">3 Stars</option>
  </select>
);
}

```

Then include it in `page.tsx` :

```

tsx

import Filter from './Filter';
// Inside return:
<Filter />

```

✓ Summary

- `params.id` : gets product ID from URL
- `searchParams.rating` : filters reviews by star rating
- `useSearchParams` and `useRouter` in client components allow for dynamic interaction

Would you like to see this project extended with API routes or real database support (e.g. Prisma + Postgres)?

yes

Perfect — let's **extend this project** to include:

1. ✓ A real backend using **API routes**
2. ✓ A PostgreSQL database using **Prisma**
3. ✓ Full-stack setup with **Next.js 15 App Router**

Project Stack

Part	Technology
Backend	Next.js API Routes
Database	PostgreSQL
ORM	Prisma
Frontend	Next.js 15 App Router (Server + Client Components)

Step 1: Set up Prisma with PostgreSQL

1. Install dependencies

```
bash

npm install prisma @prisma/client
npx prisma init
```

2. Configure `.env` (PostgreSQL connection)

```
env

DATABASE_URL="postgresql://user:password@localhost:5432/yourdbname"
```

3. Define your Prisma schema

```
prisma

// prisma/schema.prisma

model Product {
  id    Int    @id @default(autoincrement())
  name  String
  reviews Review[]
}

model Review {
  id    Int    @id @default(autoincrement())
```


```
rating Int
comment String
product Product @relation(fields: [productId], references: [id])
productId Int
}
```

4. Push to your database

```
bash
```

```
npx prisma db push
```

Step 2: Create API route to fetch product with reviews

 app/api/products/[id]/route.ts

```
ts
```

```
// app/api/products/[id]/route.ts
import { PrismaClient } from '@prisma/client';
import { NextResponse } from 'next/server';

const prisma = new PrismaClient();

export async function GET(req: Request, { params }: { params: { id: string } }) {
  const url = new URL(req.url);
  const rating = url.searchParams.get('rating');

  const product = await prisma.product.findUnique({
    where: { id: parseInt(params.id) },
    include: {
      reviews: {
        where: rating ? { rating: parseInt(rating) } : {},
      },
    },
  });

  if (!product) {
    return NextResponse.json({ error: 'Product not found' }, { status: 404 });
  }
}
```

```
return NextResponse.json(product);  
}
```

Step 3: Frontend page with data fetching

 app/products/[id]/page.tsx

tsx

```
import Filter from './Filter';  
  
interface PageProps {  
  params: { id: string };  
  searchParams: { rating?: string };  
}  
  
interface Review {  
  id: number;  
  rating: number;  
  comment: string;  
}  
  
interface Product {  
  id: number;  
  name: string;  
  reviews: Review[];  
}  
  
export default async function ProductPage({ params, searchParams }: PageProps) {  
  const res = await fetch(  
    `http://localhost:3000/api/products/${params.id}${searchParams.rating ? '?  
rating=${searchParams.rating}': ''}`,  
    { cache: 'no-store' }  
  );  
  
  if (!res.ok) {  
    return <h1>Product not found</h1>;  
  }  
  
  const product: Product = await res.json();
```

```

return (
  <main>
    <h1>{product.name} (ID: {product.id})</h1>
    <Filter />

    <h2>Reviews:</h2>
    <ul>
      {product.reviews.map((review) => (
        <li key={review.id}>
          <strong>{review.rating}★</strong> — {review.comment}
        </li>
      ))}
    </ul>
  </main>
);
}

```

Step 4: Client-side Filter Component

 app/products/[id]/Filter.tsx

tsx

'use client'

import { useRouter, useSearchParams } from 'next/navigation';

export default function Filter() {

const router = useRouter();

const searchParams = useSearchParams();

const handleChange = (e: React.ChangeEvent<HTMLSelectElement>) => {

const rating = e.target.value;

const current = new URLSearchParams(Array.from(searchParams.entries()));

if (rating) current.set('rating', rating);

else current.delete('rating');

router.push(`?\${current.toString()}`);

};

```

return (
  <select onChange={handleChange} defaultValue={searchParams.get('rating') || ''}>
    <option value="">All Ratings</option>
    <option value="5">5 Stars</option>
    <option value="4">4 Stars</option>
    <option value="3">3 Stars</option>
  </select>
);
}

```

Final Result

✓ URL: `/products/1?rating=5`

- Fetches product with ID = 1 and only 5-star reviews
- Works using:
 - `params.id` (for product)
 - `searchParams.rating` (for review filter)
 - **API route** as backend endpoint
 - **PostgreSQL via Prisma**

Extra: Seeding Some Test Data (Optional)

Add a file like `prisma/seed.ts`:

```

ts

import { PrismaClient } from '@prisma/client';
const prisma = new PrismaClient();

async function main() {
  const product = await prisma.product.create({
    data: {
      name: 'Test Product',
      reviews: {
        create: [
          { rating: 5, comment: 'Amazing!' },
          { rating: 4, comment: 'Pretty good' },
          { rating: 3, comment: 'Okay' },

```

```
    ],  
    },  
  },  
});  
  
    console.log('Seeded product:', product);  
  }  
  
  main();
```

Run it:




```
bash
```

```
ts-node prisma/seed.ts
```

✅ What You Now Have:

- Full-stack Next.js 15 app
- API route using dynamic `params` and `searchParams`
- PostgreSQL + Prisma integration
- Client-side and server-side data usage




Would you like:

-  Create/Update/Delete review functionality?
-  Add user auth (e.g. with NextAuth)?
-  Unit or E2E testing with Jest/Playwright?

Let me know how you want to extend this!

Create/Update/Delete review functionality?

Great! Let's extend the project to support **Create**, **Update**, and **Delete** operations for reviews. We'll use:

-  **Next.js API routes** (`app/api`)
-  **Prisma** to interact with the PostgreSQL database
-  **Client components** for forms & actions

-  `params` and `searchParams` where appropriate

Review CRUD API Structure

bash

`/app/api/reviews`

`[id]/`

`route.ts` *# PUT (update), DELETE*

`route.ts` *# POST (create)*

Step 1: Create Review API (POST)

 `app/api/reviews/route.ts`

ts

```
import { NextResponse } from 'next/server';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export async function POST(req: Request) {
  const body = await req.json();
  const { productId, rating, comment } = body;

  if (!productId || !rating || !comment) {
    return NextResponse.json({ error: 'Missing fields' }, { status: 400 });
  }

  const review = await prisma.review.create({
    data: {
      productId,
      rating,
      comment,
    },
  });
}
```


```
return NextResponse.json(review);  
}
```

Step 2: Update Review API (PUT)

 app/api/reviews/[id]/route.ts

```
ts  
  
import { NextResponse } from 'next/server';  
import { PrismaClient } from '@prisma/client';  
  
const prisma = new PrismaClient();  
  
export async function PUT(req: Request, { params }: { params: { id: string } }) {  
  const reviewId = parseInt(params.id);  
  const body = await req.json();  
  const { rating, comment } = body;  
  
  const updated = await prisma.review.update({  
    where: { id: reviewId },  
    data: { rating, comment },  
  });  
  
  return NextResponse.json(updated);  
}  
  
export async function DELETE(_: Request, { params }: { params: { id: string } }) {  
  const reviewId = parseInt(params.id);  
  
  await prisma.review.delete({ where: { id: reviewId } });  
  
  return NextResponse.json({ message: 'Review deleted' });  
}
```

Step 3: Add Review Form (Client Component)

 app/products/[id]/AddReview.tsx


```

'use client';

import { useState } from 'react';
import { useRouter } from 'next/navigation';

export default function AddReview({ productId }: { productId: number }) {
  const [rating, setRating] = useState(5);
  const [comment, setComment] = useState("");
  const router = useRouter();

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();

    await fetch('/api/reviews', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ productId, rating, comment }),
    });

    setComment("");
    setRating(5);
    router.refresh(); // reload data from server
  };

  return (
    <form onSubmit={handleSubmit}>
      <h3>Add Review</h3>
      <label>
        Rating:
        <select value={rating} onChange={(e) => setRating(Number(e.target.value))}>
          {[5, 4, 3, 2, 1].map((r) => (
            <option key={r} value={r}>{r} Stars</option>
          ))}
        </select>
      </label>
      <br />
      <label>
        Comment:
        <input value={comment} onChange={(e) => setComment(e.target.value)} required />
      </label>
      <br />
    </form>
  );
}

```

```
    <button type="submit">Submit</button>
  </form>
);
}
```

Step 4: Add Delete/Update UI

Inside your `page.tsx` review list:

```
tsx

'use client'

import { useRouter } from 'next/navigation';
import { useState } from 'react';

export function ReviewItem({ review }: { review: { id: number, rating: number, comment: string } }) {
  const router = useRouter();
  const [editing, setEditing] = useState(false);
  const [newComment, setNewComment] = useState(review.comment);
  const [newRating, setNewRating] = useState(review.rating);

  const deleteReview = async () => {
    await fetch(`/api/reviews/${review.id}`, { method: 'DELETE' });
    router.refresh();
  };

  const updateReview = async () => {
    await fetch(`/api/reviews/${review.id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ rating: newRating, comment: newComment }),
    });
    setEditing(false);
    router.refresh();
  };

  return (
    <li>
      {editing ? (
```

```

    <>
      <select value={newRating} onChange={(e) => setNewRating(Number(e.target.value))}>
        {[5, 4, 3, 2, 1].map((r) => (
          <option key={r} value={r}>{r} Stars</option>
        ))}
      </select>
      <input value={newComment} onChange={(e) => setNewComment(e.target.value)} />
      <button onClick={updateReview}>Save</button>
    </>
  ) : (
    <>
      <strong>{review.rating}★</strong> — {review.comment}
    </>
  )}
  <button onClick={() => setEditing((prev) => !prev)}>{editing ? 'Cancel' : 'Edit'}</button>
  <button onClick={deleteReview}>Delete</button>
</li>
);
}

```

 **Update** `page.tsx` to use `AddReview` + `ReviewItem`

```

tsx

import AddReview from './AddReview';
import { ReviewItem } from './ReviewItem';

...

return (
  <main>
    <h1>{product.name} (ID: {product.id})</h1>
    <Filter />
    <AddReview productId={product.id} />

    <h2>Reviews:</h2>
    <ul>
      {product.reviews.map((review) => (
        <ReviewItem key={review.id} review={review} />
      ))}
    </ul>
  )

```

```
</main>  
);
```

✅ Features You Now Have

Feature	Type	Route	Method
Create Review	API + Form	/api/reviews	POST
Update Review	API + Client	/api/reviews/[id]]	PUT
Delete Review	API + Client	/api/reviews/[id]]	DELETE
Fetch Reviews	Server Render	/products/[id]	GET

Would you like to:

- 🔒 Add **authentication** so only logged-in users can create/edit/delete?
- 🛡️ Add **validation** using Zod or Joi?
- 🧪 Add **unit or E2E tests** for this flow?

Let me know what direction you'd like to take this next.