**Building a Product API with ReactJS and Next.js**

Contents

[1. Project Overview 1](#_Toc186412860)

[2. Setting Up the Project 1](#_Toc186412861)

[3. Creating the API Endpoints 2](#_Toc186412862)

[Create an In-Memory Data Structure 2](#_Toc186412863)

[4. Building the Frontend 3](#_Toc186412864)

[Create Product List Component 3](#_Toc186412865)

[Create Product Form Component 3](#_Toc186412866)

[Integrate Components in the Home Page 5](#_Toc186412867)

[5. Testing React Components 6](#_Toc186412868)

[Create Test for ProductList 6](#_Toc186412869)

[Create Test for ProductForm 7](#_Toc186412870)

[6. Running the Application 8](#_Toc186412871)

[Start the Next.js Development Server 8](#_Toc186412872)

[Deploying the Product Management Application on Google Cloud Storage 8](#_Toc186412873)

[Prepare Your Application for Static Deployment 9](#_Toc186412874)

[Complete Workflow 11](#_Toc186412875)

[Commands Overview 11](#_Toc186412876)

# 1. Project Overview

This project involves building a simple product management application using **Next.js** and **TypeScript**. It will feature an in-memory data structure for storing product information and provide a form to add new products. The application will be tested using **Jest** and **React Testing Library (RTL).**

# 2. Setting Up the Project

**Initialize Next.js Project with TypeScript**

First, we'll create a new Next.jsproject:

npx create-next-app@latest product-management –typescript

cd product-management

**Install Required Dependencies**

npm install @testing-library/react @testing-library/jest-dom jest ts-jest jest-environment-jsdom

**Set Up Jest Configuration**  
Create a jest.config.js file to configure Jest for TypeScript.

module.exports = {

  preset: 'ts-jest',

  testEnvironment: 'jest-environment-jsdom',

  transform: {

    '^.+\\.(ts|tsx)$': 'ts-jest',

  },

  setupFilesAfterEnv: ['@testing-library/jest-dom', '<rootDir>/jest.setup.ts'], // Correct the module path

};

**Create a jest.setup.ts file to import jest-dom**

import '@testing-library/jest-dom';

# 3. Creating the API Endpoints

### Create an In-Memory Data Structure

We'll keep our product data in an in-memory list in pages/api/products.js:

import { NextApiRequest, NextApiResponse } from 'next';

let products = [{ id: 1, name: 'Sample Product', price: '10.00' }];

export default function handler(req: NextApiRequest, res: NextApiResponse) {

  if (req.method === 'GET') {

    res.status(200).json(products);

  } else if (req.method === 'POST') {

    const { name, price } = req.body;

    const newProduct = { id: Date.now(), name, price };

    products.push(newProduct);

    res.status(201).json(newProduct);

  } else {

    res.status(405).json({ error: 'Method Not Allowed' });

  }}

# 4. Building the Frontend

### Create Product List Component

Create a components/ProductList.tsx file for the product list:

import React from 'react';

type Product = { id: number; name: string; price: string };

const ProductList: React.FC<{ products: Product[] }> = ({ products }) => {

  return (

    <ul>

      {products.map((product) => (

        <li key={product.id}>

          {product.name} - ${product.price}

        </li>

      ))}

    </ul>

  );

};

export default ProductList;

### Create Product Form Component

Create a components/ProductForm.tsx file for adding new products:

import React, { useState } from 'react';

type Product = { id: number; name: string; price: string };

interface ProductFormProps {

  onAddProduct: (product: Product) => void;

}

const ProductForm: React.FC<ProductFormProps> = ({ onAddProduct }) => {

  const [name, setName] = useState('');

  const [price, setPrice] = useState('');

  const handleSubmit = async (event: React.FormEvent) => {

    event.preventDefault();

    const response = await fetch('/api/products', {

      method: 'POST',

      headers: { 'Content-Type': 'application/json' },

      body: JSON.stringify({ name, price }),

    });

    const product = await response.json();

    onAddProduct(product);

    setName('');

    setPrice('');

  };

  return (

    <form onSubmit={handleSubmit}>

      <input

        type="text"

        placeholder="Name"

        value={name}

        onChange={(e) => setName(e.target.value)}

      />

      <input

        type="text"

        placeholder="Price"

        value={price}

        onChange={(e) => setPrice(e.target.value)}

      />

      <button type="submit">Add Product</button>

    </form>

  );

};

export default ProductForm;

### Integrate Components in the Home Page

Update pages/index.tsx to integrate the ProductList and ProductForm components:

import React, { useState, useEffect } from 'react';

import ProductForm from '../components/ProductForm';

import ProductList from '../components/ProductList';

interface Product {

  id: number;

  name: string;

  price: string;

}

const Home: React.FC = () => {

  const [products, setProducts] = useState<Product[]>([]);

  // Fetch existing products on initial load

  useEffect(() => {

    fetch('/api/products')

      .then((res) => res.json())

      .then((data) => setProducts(data));

  }, []);

  const handleAddProduct = (product: Product) => {

    setProducts([...products, product]);

  };

  return (

    <div>

      <h1>Product Management</h1>

      {/\* Add New Product \*/}

      <ProductForm onAddProduct={handleAddProduct} />

      {/\* List Existing Products \*/}

      <ProductList products={products} />

    </div>

  );

};

export default Home;

# 5. Testing React Components

### Create Test for ProductList

Create components/ProductList.test.tsx:

import { render, screen } from '@testing-library/react';

import ProductList from './ProductList';

import '@testing-library/jest-dom';

describe('ProductList Component', () => {

  it('renders a list of products', () => {

    const products = [

      { id: 1, name: 'Product 1', price: '10.00' },

      { id: 2, name: 'Product 2', price: '20.00' },

    ];

    render(<ProductList products={products} />);

    // Check if the products are rendered

    expect(screen.getByText(/product 1/i)).toBeInTheDocument();

    expect(screen.getByText(/10.00/i)).toBeInTheDocument();

    expect(screen.getByText(/product 2/i)).toBeInTheDocument();

    expect(screen.getByText(/20.00/i)).toBeInTheDocument();

  });

  it('matches snapshot', () => {

    const products = [

      { id: 1, name: 'Product 1', price: '10.00' },

      { id: 2, name: 'Product 2', price: '20.00' },

    ];

    const { asFragment } = render(<ProductList products={products} />);

    expect(asFragment()).toMatchSnapshot();

  });

});

### Create Test for ProductForm

Create components/ProductForm.test.tsx:

import { render, screen, fireEvent, waitFor } from '@testing-library/react';

import ProductForm from './ProductForm';

import @testing-library/jest-dom';

// Mock fetch globally

global.fetch = jest.fn().mockResolvedValue({

  json: jest

    .fn()

    .mockResolvedValue({ id: 1, name: 'Product 3', price: '30.00' }),

});

test('submits new product', async () => {

  const handleAddProduct = jest.fn();

  render(<ProductForm onAddProduct={handleAddProduct} />);

  fireEvent.change(screen.getByPlaceholderText(/Name/i), {

    target: { value: 'Product 3' },

  });

  fireEvent.change(screen.getByPlaceholderText(/Price/i), {

    target: { value: '30.00' },

  });

  fireEvent.click(screen.getByText(/Add Product/i));

  await waitFor(() => {

    expect(handleAddProduct).toHaveBeenCalledWith({

      id: 1,

      name: 'Product 3',

      price: '30.00',

    });

  });

});

# 6. Running the Application

### Start the Next.js Development Server

Run the following command to start the development server:

**npm run dev**

Your application should now be accessible at http://localhost:3000.

**Add "test": "jest" to scripts section in package.json.**

**Run Tests**

Run the following command to run the tests:

**npm test**

# Deploying the Product Management Application on Google Cloud Storage

**1: Set Up Google Cloud Storage**

**1.1 Create a Google Cloud Project**

1. Go to the Google Cloud Console.
2. Create a new project or select an existing project.

**1.2 Create a Storage Bucket**

1. Navigate to **Storage** > **Buckets**.
2. Click **Create Bucket**.
3. Provide a **bucket name** (must be globally unique).
4. Select **Region** for your bucket.
5. Set the **Storage Class** to Standard.
6. Choose **Uniform** access control and click **Create**.

### **Prepare Your Application for Static Deployment**

#### **2.1 Update** next.config.js

Ensure that next.config.js has the output: 'export' setting:

Exclude test files from the production build

const nextConfig = {

reactStrictMode: true,

output: 'export', // Enables static export

};

export default nextConfig;

**Build the Static Site**

1. Run the following command to build your application:

Next.js will generate a out directory containing static files.

**3: Upload Files to Google Cloud Storage**

**3.1 Install Google Cloud CLI**

If you haven't already installed the Google Cloud SDK, install it now.

**3.2 Authenticate with Google Cloud**

Log in to your Google Cloud account:

gcloud auth login

**3.3 Set Your Project**

Set the active project:

gcloud config set project [PROJECT\_ID]

**3.4 Upload Files**

Upload the files in the out directory to your bucket:

gsutil -m cp -r out/\* gs://[BUCKET\_NAME]/

Replace [BUCKET\_NAME] with the name of your bucket.

**4: Make the Bucket Public**

1. Navigate to **Storage** > **Buckets** in the Google Cloud Console.
2. Select your bucket and click **Permissions**.
3. Click **Grant Access**.
4. Add the following:
   * **New Principals**: allUsers
   * **Role**: Storage Object Viewer
5. Save the changes to make the bucket publicly accessible.

**5: Configure the Bucket for Static Hosting – (may not be necessary. Test first)**

1. In the **Buckets** page, click on your bucket.
2. Go to the **Edit Website Configuration** section.
3. Set the following:
   * **Main Page**: index.html
   * **404 Page**: 404.html
4. Save the configuration.

**6: Test Your Deployment**

**Open the public URL for your bucket**

https://storage.googleapis.com/[BUCKET\_NAME]/index.html

Verify that the application is working correctly.

**7: Update the Deployment**

If you make changes to your app:

1. Rebuild the project:

npm run build

1. Re-upload the out directory to the bucket:

gsutil -m cp -r out/\* gs://[BUCKET\_NAME]/

# Complete Workflow

### Commands Overview

# Install Google Cloud SDK and authenticate

gcloud auth login

# Set active project

gcloud config set project [PROJECT\_ID]

# Build the application

npm run build

# Upload files to the bucket

gsutil -m cp -r out/\* gs://[BUCKET\_NAME]/

# Make the bucket public (optional step in the UI)

# Test your application

# Open: https://storage.googleapis.com/[BUCKET\_NAME]/index.html

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*