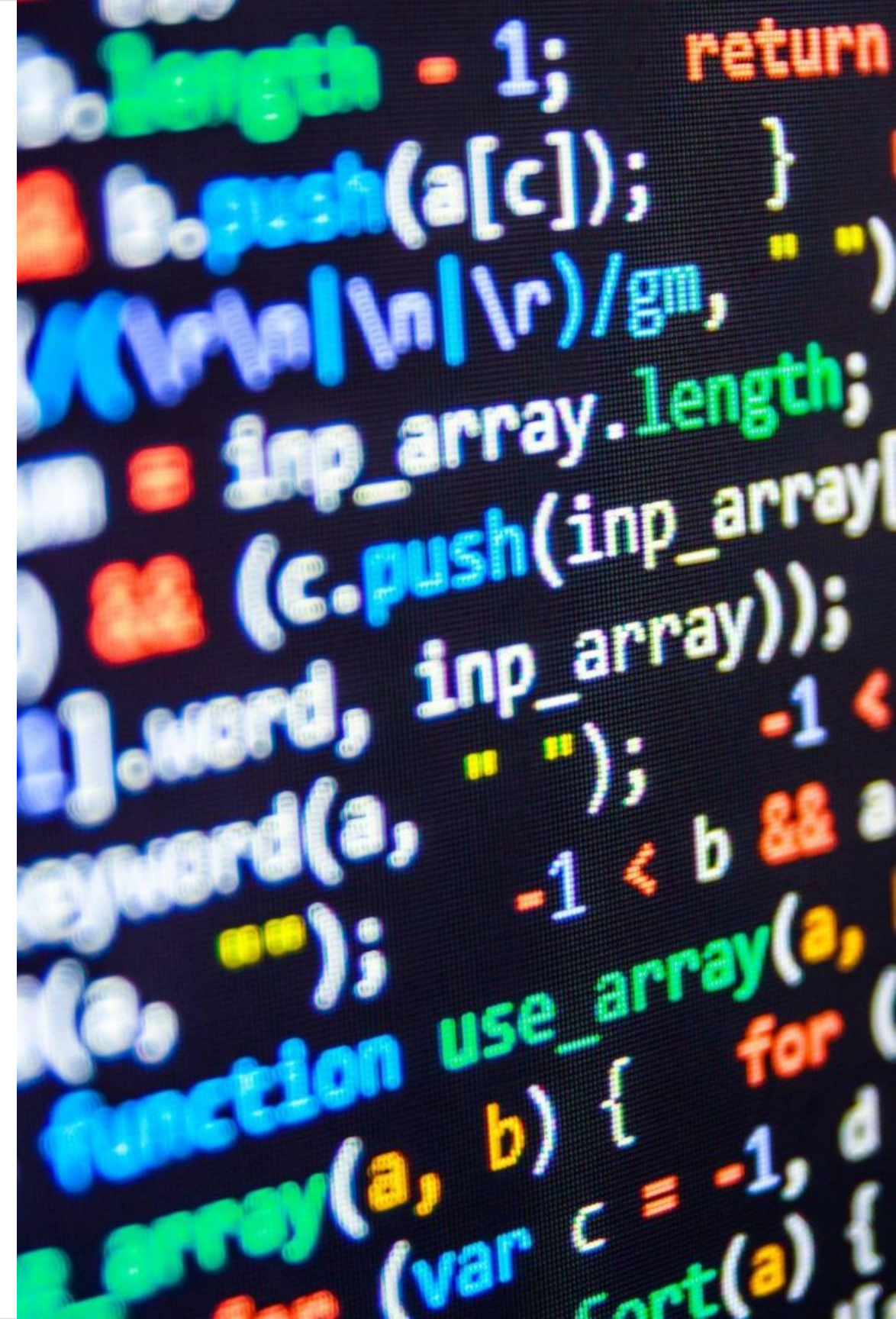


NEXT JS: Building Dynamic Web Applications

Welcome to the world of NEXT JS! In this presentation, we will explore how to install NEXT JS, create a simple router for handling various requests, and build a "Hello World" page.



What is Next.js?

The Next.js logo, featuring the word "NEXT" in a large, white, sans-serif font, followed by ".JS" in a smaller font. The logo is set against a dark, abstract background with blue and green light trails, suggesting a tunnel or a futuristic theme.

Next.js Unveiled

Next.js is an open-source React framework designed to empower you in building dynamic web applications with finesse

Incredible Features

Server-side Rendering (SSR): Next.js excels at SSR, delivering faster page loads and improved SEO.

Automatic Code Splitting: Say goodbye to tedious configuration. Next.js automatically splits your code, optimizing performance.

Seamless Routing: Its routing system is a breeze to work with, simplifying navigation in your app.

Industry Trust

Next.js isn't just a theoretical concept; it's a production-ready tool used by a multitude of companies, from startups to tech giants.

Why Next.js?

Performance, SEO, and Productivity

Why choose Next.js for your web projects? The answer is in the incredible benefits it offers:

1. **Improved Performance:** Next.js shines with its server-side rendering, reducing page load times and providing users with a smoother experience.
2. **SEO-Friendliness:** Better SEO results are achieved effortlessly, thanks to Next.js's built-in server-side rendering. Search engines love it!
3. **Developer Productivity:** Say goodbye to the complexities of configuring and managing routes. With Next.js, developers can focus on coding and creating, not on setting up routing systems.

Community and Support

Next.js boasts a large and thriving community. With extensive support and a library of extensions and packages, you're never alone in your journey. Join a network of like-minded developers, and let's build amazing things together!

How to Install Next.js

Node.js and npm

Ensure you have Node.js and npm installed on your machine. If not, download and install them from the official website.

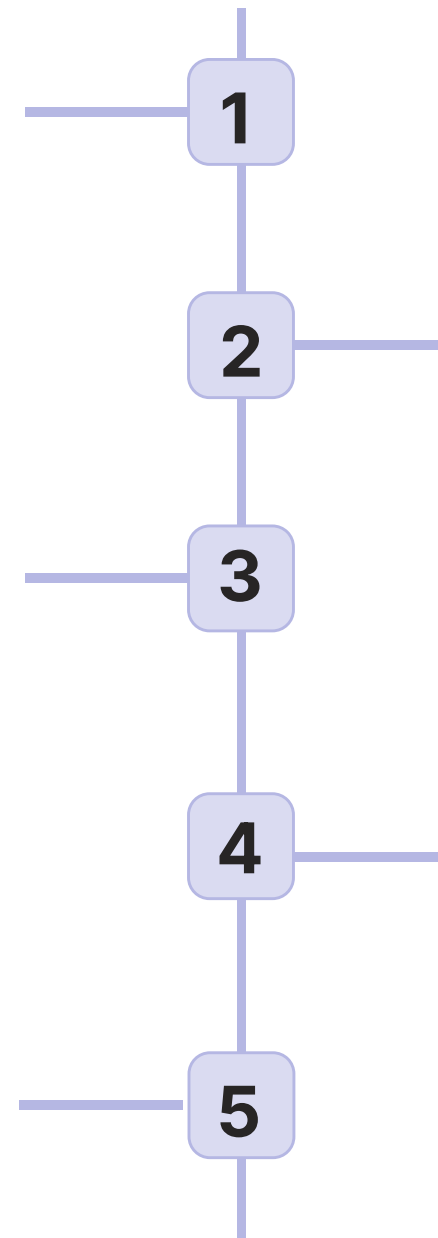
Create a New Next.js Project:

In your terminal, run the following command:

npx create-next-app my-next-app. This will create a new Next.js project named "my-next-app."

Launch the Development Server:

Start the development server with **npm run dev**. Your Next.js project is now up and running, ready for your creativity.



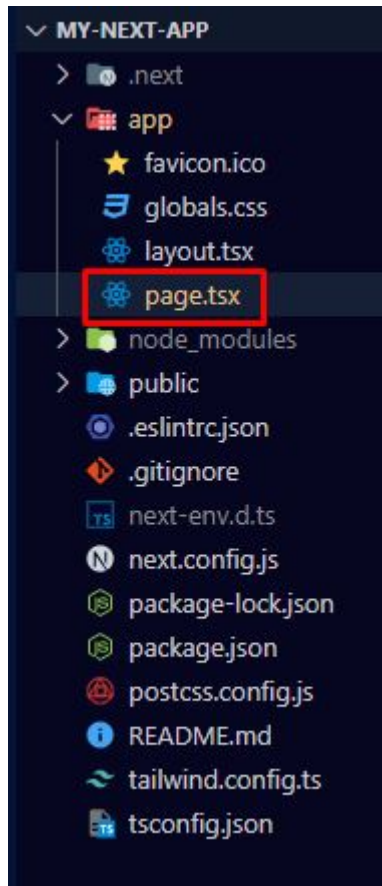
Open Your Terminal:

Open your command line interface or terminal.

Navigate to Your Project:

Use the **cd my-next-app** command to change into the project directory.

Transforming Home Page to "Hello World" Page



1

First, locate the page.tsx file inside your app directory. This is the current home page of your Next.js application.

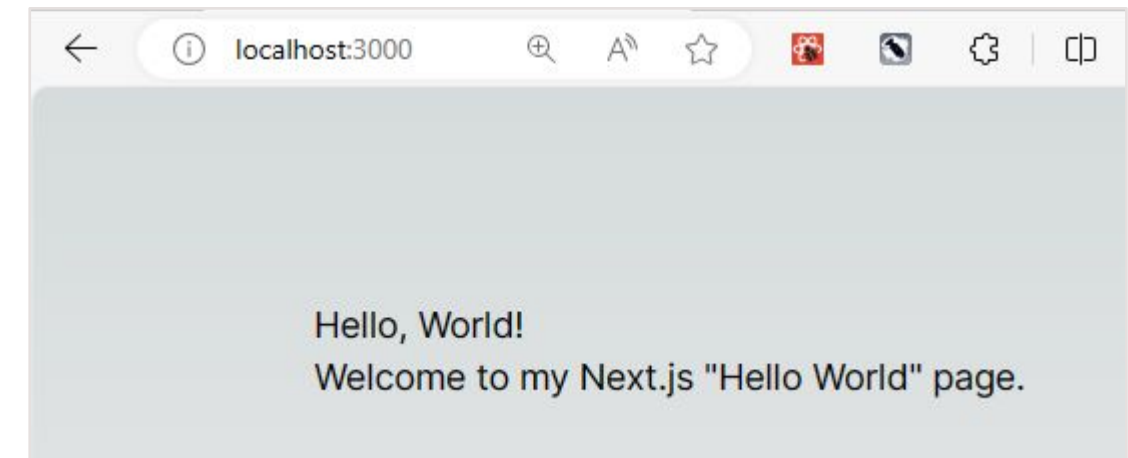
2

Update the Content

```
app > page.tsx > ...
1  export default function Home() {
2    return (
3      <main className="flex min-h-screen flex-col items-center justify-between p-24">
4        <div>
5          <h1>Hello, World!</h1>
6          <p>Welcome to my Next.js "Hello World" page.</p>
7        </div>
8      </main>
9    )
10 }
11
```

3

Go to <http://localhost:3000/>



Defining Core HTTP Methods

HTTP, or Hypertext Transfer Protocol, operates with a variety of methods to manage communication between clients and servers. These methods define the type of action that should be performed on a resource. The core HTTP methods include:



GET:

Used to retrieve data from a specified resource.



POST:

Sends data to the server for creating a new resource.



PUT:

Updates or replaces an existing resource.



DELETE

Removes a specified resource.

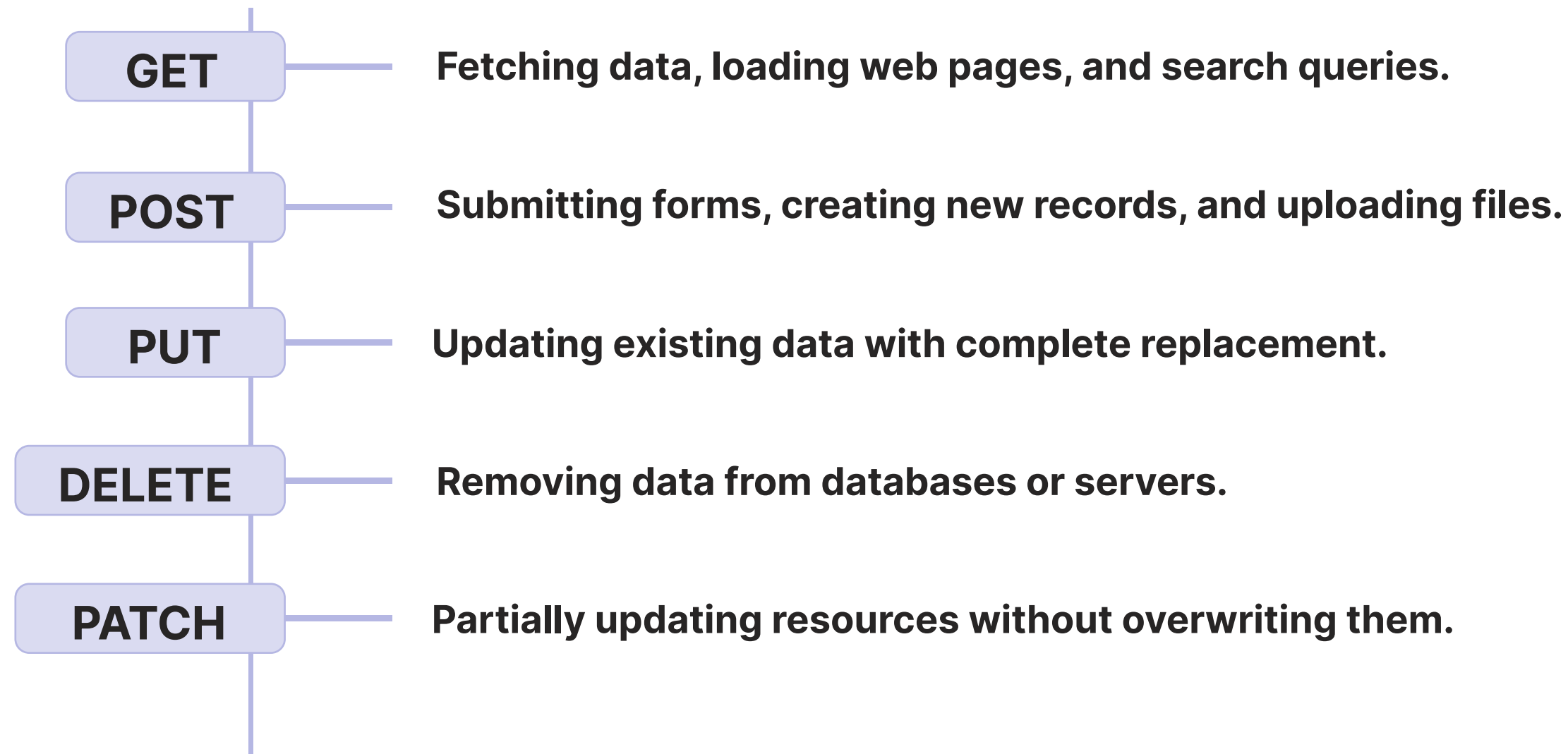


PATCH:

Partially updates a resource.

Significance in Web Development

Understanding these methods is fundamental in web development:



Properly using these methods ensures efficient, secure, and well-organized web applications.

Support for Handling HTTP Methods

One of Next.js's standout features is its exceptional support for handling various HTTP methods. This includes core methods like GET, POST, PUT, DELETE, and PATCH.

Next.js simplifies the process of managing these methods, allowing developers to create dynamic and responsive web applications effortlessly.

Building a Dynamic Blog Website with Next.js

We will explore the development of a dynamic blog website using Next.js for building web applications. Our project exemplifies the real-world application of Next.js, showcasing its features and capabilities.



How to Install Next.js

Structuring Our Project

Discuss the project's file structure, demonstrating how Next.js' file-based routing system organizes our code.

Explaining Next.js' file-based routing system.

Creating API Routes

Highlight the routes.ts files within the api directory.

Explaining how these files handle various HTTP methods (GET, PUT, DELETE, PATCH) to interact with the data functions defined in data.ts.

1

2

3

4

Data Management with data.ts

Dive into the data.ts file, which serves as our data management hub.

Describe the TypeScript type Post for structuring our blog posts.

Explaining functions like getPosts, addPost, deletePost, updatePost, and getByid for managing blog data.

Testing and Quality Assurance

API Routes Testing with Postman

The File-Based Routing System

Highlighting Next.js' Routing System

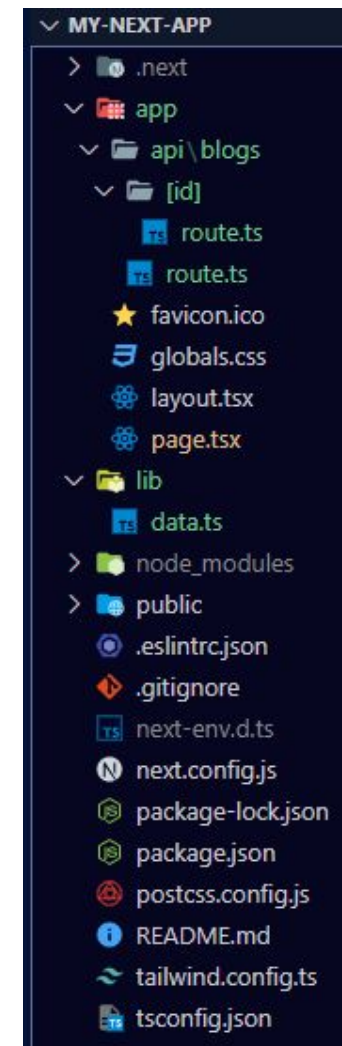
Next.js boasts a unique and intuitive file-based routing system, setting it apart from many other frameworks. This system simplifies how your application handles routes, making it both user-friendly and efficient.

File Structure Determines Routes

In Next.js, the structure of our project's files and directories plays a pivotal role in determining the routes. This means that the arrangement of your files directly influences how your application handles different paths.

For instance, consider the following structure of your project:

Here, the organization of files under the `api` and `blogs` directories defines the API routes, including dynamic routes with IDs like `[id]`.



Coding Example

To demonstrate the power of file-based routing in Next.js, we'll look at the code snippets you've provided for defining routes, such as GET, PUT, DELETE, and PATCH, in the `routes.ts` files.

These files contain the logic to handle various HTTP methods with Next.js, and they perfectly align with the file-based routing system.

Description of data.ts

The data.ts file provides the data management logic for our application. It defines a TypeScript type `Post` representing the structure of a blog post, including an id, title, description (desc), and a date.

It also initializes an array `posts` to store blog post objects.

Four functions are provided for data management:

- **getPosts**: Retrieves all posts from the `posts` array.
- **addPost**: Adds a new post to the `posts` array.
- **deletePost**: Removes a post by filtering the array based on the provided id.
- **updatePost**: Updates the title and desc of a post with a specified id.
- **getById**: Retrieves a post based on the id.

In the context of the entire scenario:

We are building a web application, presumably a blog system, using Next.js.

The data.ts file manages our blog post data, allowing you to retrieve, add, delete, update, and retrieve posts by their id.

```
lib > data.ts > ...
1  type Post = {
2      id: string;
3      title: string;
4      desc: string;
5      date: Date;
6  };
7
8  let posts: Post[] = [];
9
10 //handlers
11 export const getPosts = () => posts;
12
13 export const addPosts = (post: Post) => {
14     posts.push(post);
15 };
16
17 export const deletePost = (id: string) => {
18     posts = posts.filter((post) => post.id !== id);
19 };
20
21 export const updatePost = (id: string, title: string, desc: string) => {
22     const post = posts.find((post) => post.id === id );
23
24     if (post) {
25         post.title = title;
26         post.desc = desc;
27     } else {
28         throw new Error("NO POST FOUND");
29     }
30 };
31
32 export const getById = (id: string) => {
33     return posts.find((post) => post.id === id );
34 }
35
```


Explaining the GET Request Handling (GET All)

This code snippet represents a GET request API route in Next.js designed to retrieve all blog posts.

Upon receiving a GET request, the route calls the **getPosts** function from your data module to fetch all available blog posts.

If the operation is successful, it responds with a JSON object containing the fetched posts and a status code of 200, indicating a successful response.

In the event of an error, such as a database connection issue, it returns a JSON response with an "Error" message and a 500 status code, indicating a server error.

This route showcases how Next.js efficiently handles GET requests to retrieve all resources and provides meaningful responses for both success and error scenarios.

```
app > api > blogs > route.ts > ...
1  import { getPosts, addPosts } from "@lib/data";
2  import { NextResponse } from "next/server"
3
4  export const GET = async (req: Request, res: Response) => {
5    try {
6      const posts = getPosts();
7      return NextResponse.json({ message: "Success", posts }, { status: 200 });
8    } catch (err) {
9      return NextResponse.json({ message: "Error", err }, { status: 500 });
10   }
11 };
12
```


Code Snippet for POST Request API Route

The provided code snippet illustrates a POST request API route in Next.js, responsible for adding a new blog post.

Upon receiving a POST request, the route extracts data from the request body, specifically the title and desc of the new blog post.

It then attempts to create a new post object with a unique ID, timestamp, and the received data.

If the creation is successful, it responds with a JSON object containing the newly created post and a status code of 201, indicating a successful resource creation.

In the event of an error, it returns a JSON response with an "Error" message and a 500 status code, signifying a server error.

This route exemplifies how Next.js effectively handles POST requests to add new resources and delivers meaningful responses for both success and error scenarios.

```
app > api > blogs > route.ts > ...
1  import { getPosts, addPosts } from "@lib/data";
2  import { NextResponse } from "next/server"
3
4  export const POST = async (req: Request, res: Response) => {
5      const { title, desc } = await req.json();
6
7      try {
8          const post = { title, desc, date: new Date(), id: Date.now().toString() };
9          addPosts(post);
10         return NextResponse.json({ message: "OK", post }, { status: 201 });
11     } catch (err) {
12         return NextResponse.json({ message: "Error", err }, { status: 500 });
13     }
14 };
15
```

Code Snippet for GET by ID

Request API Route

This code snippet represents a GET request API route in Next.js designed to retrieve a specific blog post by its unique ID.

Upon receiving a GET request, the route extracts the ID from the request URL by splitting it, ensuring it captures the specific blog post's identifier.

It then utilizes the **getById** function from your data module to fetch the blog post associated with the provided ID.

If the operation is successful and a matching post is found, it responds with a JSON object containing the post and a status code of 200, indicating a successful response.

In case no post is found (i.e., an incorrect or non-existent ID is provided), it returns a JSON response with an "ERROR" message and a 404 status code, signifying a resource not found.

This route exemplifies how Next.js efficiently handles GET requests for retrieving specific resources, delivering meaningful responses for both success and error scenarios.

```
app > api > blogs > [id] > TS route.ts > ...
1  import { deletePost, getById, updatePost } from "@lib/data";
2  import { NextResponse } from "next/server"
3
4  export const GET = async (req: Request) => {
5      try {
6          const id = req.url.split("blogs/")[1];
7          const post = getById(id);
8          console.log(id);
9          if (!post){
10             return NextResponse.json({ message: "ERROR", post }, { status: 404 });
11         }
12         return NextResponse.json({ message: "OK", post }, { status: 200 });
13     } catch(err){
14         return NextResponse.json({ message: "ERROR", err }, { status: 404 });
15     }
16 };
17
```

Code Snippet for PUT Request API Route

This code snippet represents a PUT request API route in Next.js designed to update an existing blog post.

When a PUT request is received, the route extracts the ID and the updated data (i.e., title and desc) from the request body.

It then uses the **updatePost** function from your data module to modify the existing post with the provided ID using the new data.

If the update operation is successful, it responds with a JSON object containing the "OK" message and a status code of 200, indicating a successful response.

In case of an error, such as an invalid ID or failed update, it returns a JSON response with an "ERROR" message and a 404 status code, signifying a resource not found.

This route showcases how Next.js effectively handles PUT requests for updating resources and provides meaningful responses for both success and error scenarios.

```
app > api > blogs > [id] > route.ts > DELETE
1  import { deletePost, getById, updatePost } from "@lib/data";
2  import { NextResponse } from "next/server"
3
4  > export const GET = async (req: Request) => { ...
16 };
17
18 export const PUT = async (req: Request) => {
19   try {
20     const {title, desc} = await req.json();
21     const id = req.url.split("blogs/")[1];
22     updatePost(id, title, desc);
23     return NextResponse.json({ message: "OK" }, { status: 200 });
24   } catch(err){
25     return NextResponse.json({ message: "ERROR", err }, { status: 404 });
26   }
27 }
28 };
29
```


Code Snippet for DELETE Request API Route

This code snippet represents a DELETE request API route in Next.js designed to delete a specific blog post.

Upon receiving a DELETE request, the route extracts the ID from the request URL by splitting it, ensuring it identifies the specific blog post to be deleted.

It then utilizes the **deletePost** function from your data module to remove the blog post associated with the provided ID.

If the deletion is successful, it responds with a JSON object containing the "OK" message and a status code of 200, indicating a successful response.

In the event of an error, such as an incorrect or non-existent ID, it returns a JSON response with an "ERROR" message and a 404 status code, signifying a resource not found.

This route exemplifies how Next.js efficiently handles DELETE requests for removing specific resources, delivering meaningful responses for both success and error scenarios.

```
app > api > blogs > [id] > route.ts > ...
1  import { deletePost, getById, updatePost } from "@lib/data";
2  import { NextResponse } from "next/server"
3
4  > export const GET = async (req: Request) => { ...
16 };
17
18 > export const PUT = async (req: Request) => { ...
28 };
29
30 export const DELETE = async (req: Request) => {
31   try {
32     const id = req.url.split("blogs/")[1];
33     deletePost(id);
34     console.log(id);
35     return NextResponse.json({ message: "OK" }, { status: 200 });
36   } catch(err){
37     return NextResponse.json({ message: "ERROR", err }, { status: 404 });
38   }
39 };
40
```

Code Snippet for PATCH Request API Route

This code snippet represents a PATCH request API route in Next.js designed to update an existing blog post.

When a PATCH request is received, the route extracts the ID and the updated data (i.e., title and desc) from the request body.

It then uses the **updatePost** function from your data module to modify the existing post with the provided ID using the new data.

If the update operation is successful, it responds with a JSON object containing the "OK" message and a status code of 200, indicating a successful response.

In case of an error, such as an invalid ID or a failed update, it returns a JSON response with an "ERROR" message and a 404 status code, signifying a resource not found.

This route showcases how Next.js effectively handles PATCH requests for updating resources and provides meaningful responses for both success and error scenarios.

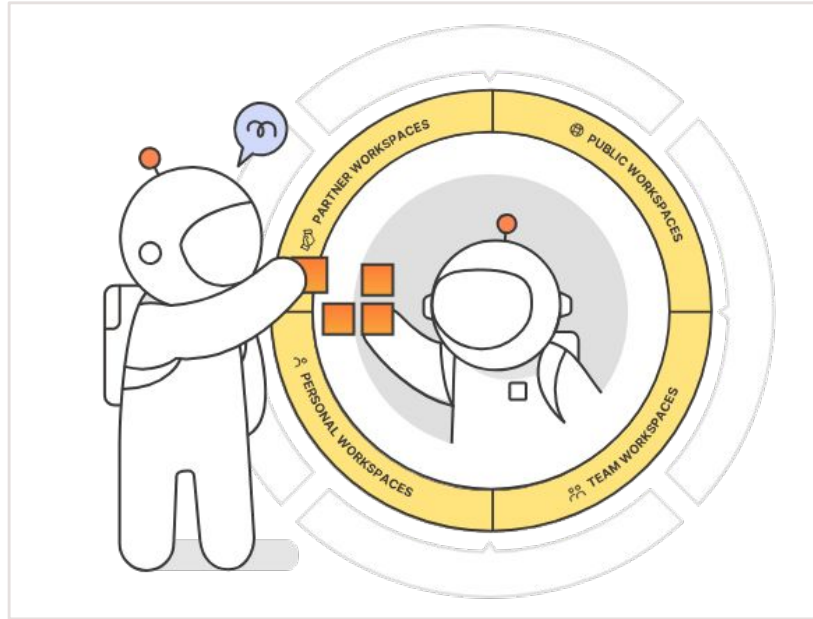
```
app > api > blogs > [id] > route.ts > ...
1  import { deletePost, getById, updatePost } from "@lib/data";
2  import { NextResponse } from "next/server"
3
4  > export const GET = async (req: Request) => { ...
16  };
17
18  > export const PUT = async (req: Request) => { ...
28  };
29
30  > export const DELETE = async (req: Request) => { ...
39  };
40
41  export const PATCH = async (req: Request) => {
42    try {
43      const { title, desc } = await req.json();
44      const id = req.url.split("blogs/")[1];
45      updatePost(id, title, desc);
46      return NextResponse.json({ message: "OK" }, { status: 200 });
47    } catch (err) {
48      return NextResponse.json({ message: "ERROR", err }, { status: 404 });
49    }
50  };
51
```


Testing and Quality Assurance

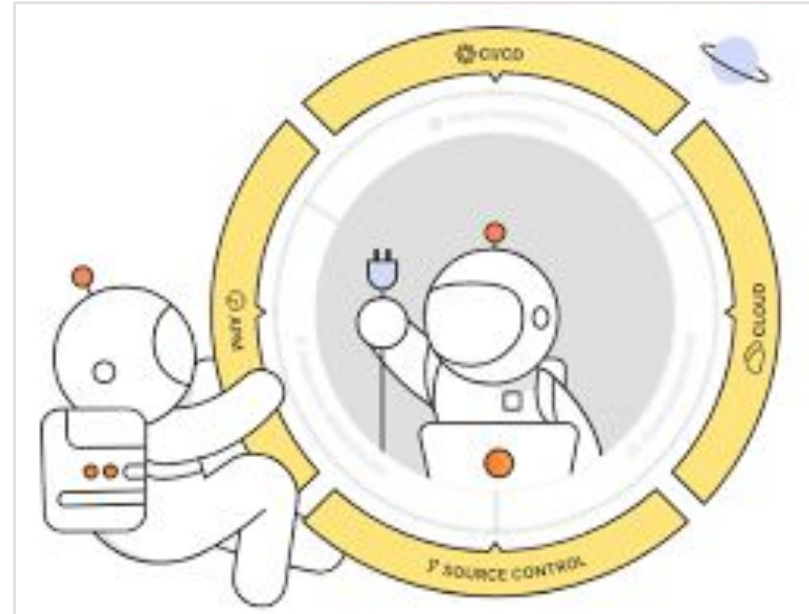
API Routes Testing with Postman



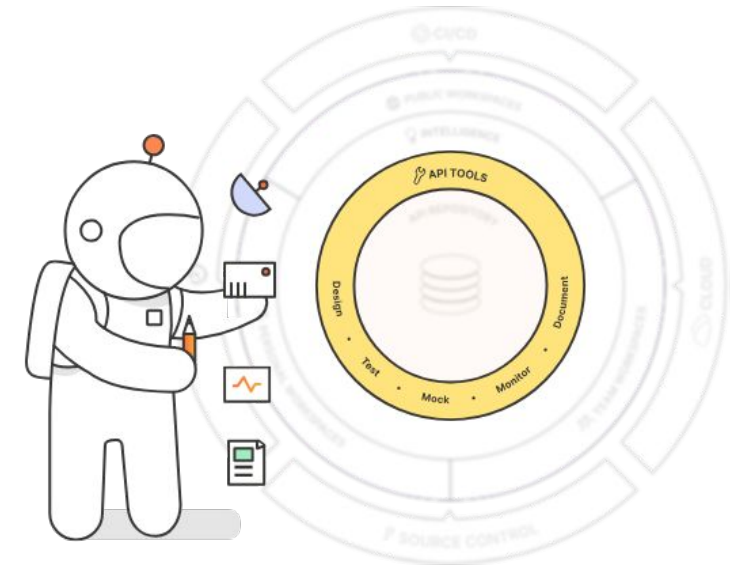
POSTMAN



Efficient testing is vital to ensure the reliability and functionality of your API routes.



Postman is a popular and powerful API testing tool that simplifies the testing process.



Postman allows you to create and manage requests, organize them into collections, and automate testing workflows.

Testing a POST Request

The screenshot displays a REST client interface for testing a POST request. The top section shows the request configuration:

- Method:** POST (highlighted with a red box)
- URL:** `http://localhost:3000/api/blogs ...` (highlighted with a red box)
- Body Tab:** Selected (highlighted with a red box)
- Body Type:** raw (highlighted with a red box)
- Content Type:** JSON (highlighted with a red box)
- Send Button:** A blue button with a red arrow pointing to it.

The request body is defined as follows:

```
1 {  
2   "title": "My First Post",  
3   "desc": "SAMPLE"  
4 }
```

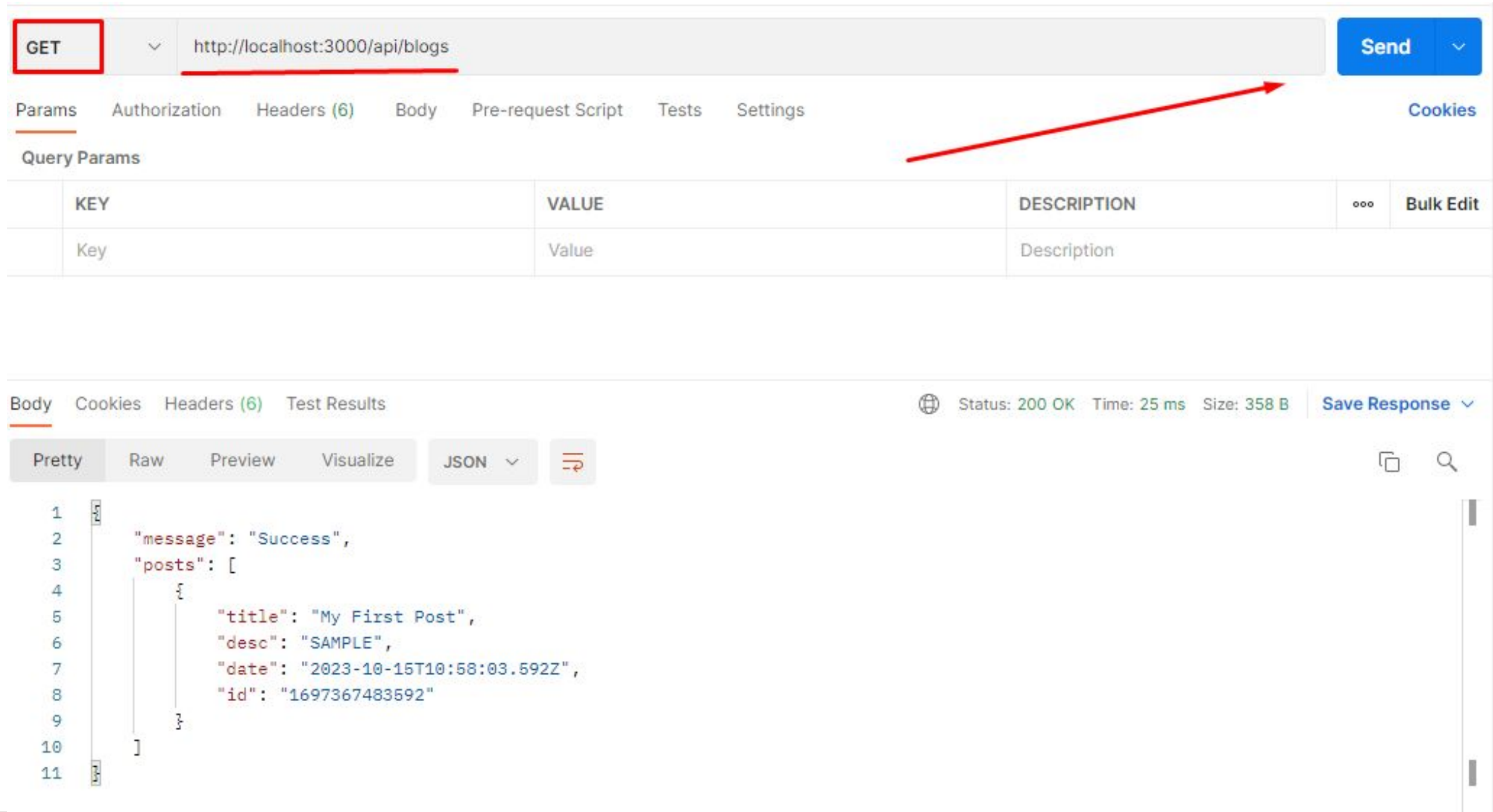
The bottom section shows the response details:

- Status:** 201 Created
- Time:** 21 ms
- Size:** 355 B
- Save Response:** A blue button with a dropdown arrow.

The response body is displayed in the 'Pretty' tab:

```
1 {  
2   "message": "OK",  
3   "post": {  
4     "title": "My First Post",  
5     "desc": "SAMPLE",  
6     "date": "2023-10-15T10:58:03.592Z",  
7     "id": "1697367483592"  
8   }  
9 }
```

Testing a GET Request



The screenshot displays a REST client interface for testing a GET request. The request method is **GET** (highlighted with a red box) and the URL is http://localhost:3000/api/blogs. The **Send** button is visible on the right. Below the URL bar, tabs for **Params**, **Authorization**, **Headers (6)**, **Body**, **Pre-request Script**, **Tests**, and **Settings** are shown. The **Query Params** section is active, displaying a table with columns **KEY**, **VALUE**, and **DESCRIPTION**. A red arrow points from the **Send** button to this table.

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Below the query params, the **Body** tab is selected, showing the response in **JSON** format. The response status is **200 OK** with a time of **25 ms** and size of **358 B**. The response body is:

```
1 {
2   "message": "Success",
3   "posts": [
4     {
5       "title": "My First Post",
6       "desc": "SAMPLE",
7       "date": "2023-10-15T10:58:03.592Z",
8       "id": "1697367483592"
9     }
10  ]
11 }
```

Testing a GET by ID Request

NextJS_blog / Get By ID Copy

Save

GET Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (6) Test Results

Status: 200 OK Time: 34 ms Size: 350 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": "OK",
3   "post": {
4     "title": "My First Post",
5     "desc": "SAMPLE",
6     "date": "2023-10-15T11:02:48.461Z",
7     "id": "1697367768461"
8   }
9 }
```


Testing a PUT Request

The screenshot displays a REST client interface with the following configuration and results:

- Method:** PUT
- URL:** `http://localhost:3000/api/blogs/1697367768461`
- Body Tab:** Selected, showing a JSON payload:

```
1 {  
2   "title": "My new Post",  
3   "desc": "SAMPLE"  
4 }
```
- Content Type:** raw
- Format:** JSON
- Send Button:** A red arrow points to the 'Send' button.
- Response Section:**
 - Status:** 200 OK
 - Time:** 39 ms
 - Size:** 246 B
 - Save Response:** Button available
 - Body Tab:** Selected, showing the response in 'Pretty' format:

```
1 {  
2   "message": "OK"  
3 }
```

Testing a PATCH Request

The screenshot displays a REST client interface with the following components:

- Request Method:** PATCH (highlighted with a red box).
- URL:** `http://localhost:3000/api/blogs/1697355590691` (underlined with a red line).
- Body Tab:** Selected and highlighted with a red box.
- Body Type:** raw (highlighted with a red box).
- Body Format:** JSON (highlighted with a red box).
- Request Body:**

```
1 {  
2   "desc": "SAMPLEEE"  
3 }
```

 (entire block highlighted with a red box).
- Send Button:** A blue button to execute the request, with a red arrow pointing to it.
- Response Section:**
 - Status:** 200 OK, Time: 34 ms, Size: 246 B.
 - Save Response:** A blue button.
 - Response Body:**

```
1 {  
2   "message": "OK"  
3 }
```

Testing a DELETE Request

The screenshot shows a REST client interface with a DELETE request configured. The URL is `http://localhost:3000/api/blogs/1697354576155`. The request is sent, resulting in a 200 OK status. The response body is displayed in JSON format, showing `"message": "OK"`.

Request Configuration:

- Method: DELETE
- URL: `http://localhost:3000/api/blogs/1697354576155`
- Params: Query Params
- Body: (Empty)
- Pre-request Script: (Empty)
- Tests: (Empty)
- Settings: (Empty)

KEY	VALUE	DESCRIPTION
Key	Value	Description

Response Details:

- Status: 200 OK
- Time: 32 ms
- Size: 246 B
- Save Response

Response Body (JSON):

```
1 {
2   "message": "OK"
3 }
```

Summarizing Key Points

In today's presentation, we've covered several crucial aspects of Next.js and web application development:

- Structuring our project efficiently with Next.js' file-based routing system.
- Effective data management using the data.ts module.
- Creating versatile API routes for handling HTTP methods.
- Streamlining testing processes with Postman.

The Power of Next.js



Next.js has emerged as a powerful and flexible framework for modern web development.

It simplifies project organization with its file-based routing system.

Offers robust data management capabilities with TypeScript.

Provides a seamless way to create API routes.

Enables efficient testing and quality assurance.

Next Steps

1

I encourage you to explore Next.js further, whether you're a seasoned developer or just beginning your journey in web development.

2

Start building your web applications with Next.js and experience its capabilities firsthand.

Thank You

The Next.js logo is displayed in white text against a dark, futuristic background. The background features a series of concentric, glowing blue and green rings, resembling a tunnel or a high-tech interface. The text "NEXT" is in a large, sans-serif font, and ".JS" is in a smaller font to the right. A thin white diagonal line cuts across the "X" in "NEXT".

NEXT.js

Thank you for your attention and engagement. I'm here to answer any questions and assist you in your journey with Next.js and web development.

Let's continue to create innovative and exciting web applications with Next.js!