Go Language

# 1: Introduction & Setup

## 1. Overview of Go

Go (or Golang) is an open-source, statically typed, compiled programming language designed for simplicity, efficiency, and scalability. Go is a **statically typed** language, which means that **variable types are checked at compile time** rather than at runtime.

### History of Go

* Created at **Google** in 2007 by **Robert Griesemer, Rob Pike, and Ken Thompson**
* Officially released in **2009**
* Designed to improve **speed, concurrency, and ease of development** compared to older languages like C++

### Key Features of Go

* **Simplicity** – Minimal syntax, easy to read
* **Performance** – Compiled language with fast execution
* **Garbage Collection** – Automatic memory management
* **Concurrency** – Built-in support for goroutines (lightweight threads)
* **Cross-Platform** – Works on Windows, macOS, Linux
* **Strong Typing** – Statically typed but less verbose than Java or C++

### Why Use Go?

* Used by **Google, Uber, Dropbox, Kubernetes, Docker**
* Ideal for **backend services, cloud applications, and microservices**
* Easy to learn and adopt for developers coming from C, Python, or Java

## 2. Setting Up Go

### Installation Steps:

1. Download Go from the official website: https://go.dev/dl/
2. Install it and verify by running:

|  |
| --- |
| go version |

If correctly installed, it will output something like:

1. Check the Go environment variables:

|  |
| --- |
| go env |

This displays the paths and settings for Go.

## 3. Understanding the Go Workspace & GOPATH

### 1. Go Workspace Structure in Windows

When Go is installed, it sets up a **default workspace** in:

|  |
| --- |
| C:\Users\YourUsername\go\ # GOPATH (workspace)  ├── src\ # Contains Go source code (your projects)  ├── bin\ # Stores compiled executable files  ├── pkg\ # Stores compiled packages |

### 2. Understanding GOPATH and GOROOT

* **GOPATH**: Defines where Go projects are stored (**default: C:\Users\YourUsername\go**).
* **GOROOT**: Points to the Go installation directory (**default: C:\Go**).

### 3. Checking GOPATH and GOROOT in Windows

Open **Command Prompt (cmd)** or **PowerShell** and run:

|  |
| --- |
| echo %GOPATH% # Check Go workspace path  go env GOROOT # Check Go installation path |

### 4. Setting GOPATH Manually (If Needed)

If GOPATH is not set, you can configure it:

|  |
| --- |
| setx GOPATH "C:\Users\YourUsername\go" |

Restart your terminal for changes to take effect.

### 5. Using Go Modules (No Need to Stay in GOPATH)

With **Go Modules**, projects can be created **anywhere**, not just inside GOPATH.  
Example of a project outside GOPATH:

|  |
| --- |
| C:\Projects\MyGoApp\ # Project folder  ├── main.go # Main Go file  ├── go.mod # Go module file |

To initialize a Go module:

|  |
| --- |
| cd C:\Projects\MyGoApp  go mod init mygoapp |

## 4. Writing and Running a Simple "Hello, World!" Program

**Create a new Go file:**

1. Open a terminal and create a folder for your project:

|  |
| --- |
| mkdir myproject  cd myproject |

1. Create a new file called main.go:

|  |
| --- |
| touch main.go # Linux/macOS  notepad main.go # Windows |

1. Add the following Go code to main.go:

|  |
| --- |
| package main  import "fmt"  func main() {  fmt.Println("Hello, World!")  } |

**Run the program:**

|  |
| --- |
| go run main.go |

# Running Go in VS Code

You can set up and run Go programs in **VS Code** by following these steps:

### 1. Install VS Code & Go Extension

* **Download and install VS Code**: <https://code.visualstudio.com/>
* **Install the Go extension** in VS Code:
  + Open VS Code
  + Go to **Extensions (Ctrl + Shift + X)**
  + Search for **Go** and install the official extension by **Go Team at Google**

### 2. Set Up a Go Project in VS Code

1. **Create a new Go project folder:**  
   Open a terminal and run:

|  |
| --- |
| mkdir myproject  cd myproject |

1. **Initialize the project:**

|  |
| --- |
| go mod init myproject |

This creates a go.mod file, which helps manage dependencies.

1. **Create a new Go file:**

* Inside the myproject folder, create a file named main.go.
* Open main.go in VS Code and add:

|  |
| --- |
| package main  import "fmt"  func main() {  fmt.Println("Hello, World!")  } |

### 3. Running the Go Program in VS Code

**Method 1: Run Using Terminal**

* Open the **VS Code terminal** (**Ctrl + `** or go to **View > Terminal**)
* Run:

go run main.go

**Method 2: Using the VS Code Run Button**

* Click on the **"Run" ▶ button** at the top-right corner of VS Code.
* Select **"Run Without Debugging" (Ctrl + F5)**
* You should see the output in the **VS Code terminal**

### 4. Optional: Enable Autoformatting & Linting

* The **Go extension** provides automatic formatting and error checking.
* If prompted, click **"Install all dependencies"** to enable features like IntelliSense, debugging, and code completion.

## Understanding go.mod and go.sum in Go Modules

Go uses **modules** for dependency management, and these files help manage them efficiently.

### 1. What is go.mod?

* It is the **module definition file** in a Go project.
* It **tracks the module name** and **dependencies**.
* Created when you run:

|  |
| --- |
| go mod init myproject |

### 2. What is go.sum?

* It **records checksums (hash values)** for each dependency.
* Ensures that dependencies **have not been tampered with**.
* Created automatically when you install dependencies (go get).
* **Do not edit go.sum manually**—Go manages it automatically.

### 3. How They Work Together

* go.mod defines **what dependencies** your project needs.
* go.sum verifies **dependency integrity** to prevent changes.

### ****Difference Between**** go.mod ****and**** go.sum

|  |  |  |
| --- | --- | --- |
| **Feature** | **go.mod** | **go.sum** |
| **Purpose** | Defines the Go module and its dependencies | Stores checksums to verify dependencies |
| **Created When?** | Running go mod init <module\_name> | When a dependency is first downloaded (go get) |
| **Updated When?** | Adding, updating, or removing dependencies | Any time dependencies are downloaded, verified, or updated |
| **Contains** | Module name, required dependencies, Go version | Hash values (checksums) for dependencies |
| **Example** | require github.com/gin-gonic/gin v1.9.0 | github.com/gin-gonic/gin v1.9.0 h1:AbCdEf... |
| **Can Be Edited?** | Yes, but managed by Go | No, Go updates it automatically |
| **Role in Security** | Specifies which package versions to use | Prevents tampering by verifying integrity |

# 2. Input and Output in Go

Go provides the fmt package for basic input and output operations. Below are examples of how to handle **user input and output** in Go.

## 1. Printing Output (fmt.Println, fmt.Printf)

Use fmt.Println() and fmt.Printf() to display output.

|  |
| --- |
| package main  import "fmt"  func main() {  fmt.Println("Hello, Go!") // Prints with a newline  fmt.Print("Hello") // Prints without a newline  fmt.Printf("\nMy age is %d\n", 25) // Formatted output  } |

* **fmt.Println()** adds a newline automatically.
* **fmt.Print()** does not add a newline.
* **fmt.Printf()** allows formatted output (e.g., %d for integers).

## 2. Taking User Input (fmt.Scan, fmt.Scanf, fmt.Scanln)

Go uses fmt.Scan(), fmt.Scanf(), and fmt.Scanln() for user input.

|  |
| --- |
| package main  import "fmt"  func main() {  var name string  fmt.Print("Enter your name: ")  fmt.Scan(&name) // Takes input from user  fmt.Println("Hello,", name)  } |

**Example: Taking Multiple Inputs**

|  |
| --- |
| package main  import "fmt"  func main() {  var name string  var age int  fmt.Print("Enter your name and age: ")  fmt.Scan(&name, &age) // Takes two inputs  fmt.Printf("Name: %s, Age: %d\n", name, age)  } |

**Example: Using fmt.Scanf() (Formatted Input)**

|  |
| --- |
| package main  import "fmt"  func main() {  var name string  var age int  fmt.Print("Enter name and age: ")  fmt.Scanf("%s %d", &name, &age) // Takes formatted input  fmt.Printf("Hello %s, you are %d years old.\n", name, age)  } |

# 3. Basic Syntax & Control Structures

## 1. Variables in Go

A variable is a **named storage location** in memory that holds a value. In Go, variables must be declared before use.

### Declaring Variables in Go

1. **Explicit Declaration (with type)**

The var keyword is used. The type is explicitly mentioned.

|  |
| --- |
| var age int = 25  var name string = "Alice"  var isActive bool = true |

1. **Implicit Declaration (Type Inference using :=)**

Type is automatically inferred. Can only be used **inside functions**.

|  |
| --- |
| age := 25 // Go infers age as int  name := "Alice" // Go infers name as string  isActive := true // Go infers isActive as bool |

1. **Multiple Variable Declaration**

Variables can be declared together.

|  |
| --- |
| var a, b, c int = 1, 2, 3  x, y, z := "Go", 3.14, false |

## 2. Constants in Go

A **constant** is a variable whose value **cannot be changed** after being assigned. In Go, if a constant is **untyped**, it adopts the type where it is used.

**Declaring Constants in Go**

|  |
| --- |
| const pi = 3.14  const appName string = "Task Manager" |

pi and appName **cannot** be modified later in the program.

## 3. Data Types in Go

A **data type** defines the kind of values a variable can hold. Go is **statically typed**, meaning the type of a variable is checked at compile-time.

### 1. Basic Data Types

|  |  |  |
| --- | --- | --- |
| **Type** | **Example** | **Description** |
| bool | true, false | Boolean values |
| string | "Hello" | Sequence of characters |
| int | 10 | Integer (depends on OS: int32 or int64) |
| int8 | -128 to 127 | 8-bit integer |
| int16 | -32768 to 32767 | 16-bit integer |
| int32 | -2^31 to 2^31-1 | 32-bit integer |
| int64 | -2^63 to 2^63-1 | 64-bit integer |
| uint8 | 0 to 255 | Unsigned 8-bit integer |
| uint16 | 0 to 65535 | Unsigned 16-bit integer |
| uint32 | 0 to 2^32-1 | Unsigned 32-bit integer |
| uint64 | 0 to 2^64-1 | Unsigned 64-bit integer |
| float32 | 3.14 | 32-bit floating-point number |
| float64 | 3.14159265 | 64-bit floating-point number |
| complex64 | 1 + 2i | Complex number with float32 real and imaginary parts |
| complex128 | 2 + 5i | Complex number with float64 real and imaginary parts |
| byte | uint8 alias | Used for raw binary data |
| rune | int32 alias | Represents a Unicode character |

### 2. Composite Data Types (Collections & Structures)

|  |  |  |
| --- | --- | --- |
| **Type** | **Example** | **Description** |
| array | [3]int{1, 2, 3} | Fixed-size collection of elements |
| slice | []int{1, 2, 3} | Dynamic-size collection |
| map | map[string]int{"age": 30} | Key-value pairs (like dictionaries) |
| struct | type Person struct { Name string } | Custom data structure |
| pointer | \*int | Stores memory address of a variable |
| interface | interface{} | Defines method sets for polymorphism |
| channel | make(chan int) | Used for goroutine communication |

### 3. Example Usage of Data Types in Go

|  |
| --- |
| package main  import "fmt"  func main() {  // Basic types  var a int = 10  var b float64 = 3.14  var c string = "GoLang"  var d bool = true  // Composite types  arr := [3]int{1, 2, 3} // Array  slc := []string{"apple", "banana"} // Slice  dict := map[string]int{"age": 30} // Map  fmt.Println(a, b, c, d)  fmt.Println(arr, slc, dict)  } |

## 4. Operators in Go

Operators perform **mathematical, comparison, and logical operations**.

### Arithmetic Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition | a + b |
| - | Subtraction | a - b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (Remainder) | a % b |

Example:

|  |
| --- |
| package main  import "fmt"  func main() {  a, b := 10, 3  fmt.Println(a + b) // 13  fmt.Println(a - b) // 7  fmt.Println(a \* b) // 30  fmt.Println(a / b) // 3 (integer division)  fmt.Println(a % b) // 1  } |

### Comparison Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Equal to | a == b |
| != | Not equal to | a != b |
| > | Greater than | a > b |
| < | Less than | a < b |
| >= | Greater than or equal to | a >= b |
| <= | Less than or equal to | a <= b |

### Logical Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Logical AND | a > 0 && b > 0 |
| || | Logical OR | a > 0 || b > 0 |
| ! | Logical NOT | !isTrue |

Example:

|  |
| --- |
| package main  import "fmt"  func main() {  x, y := true, false  fmt.Println(x && y) // false  fmt.Println(x || y) // true  fmt.Println(!x) // false  } |

## 5. Control Structures (Decision Making & Loops)

### If-Else Statement

Used for conditional execution.

|  |
| --- |
| package main  import "fmt"  func main() {  num := 10  if num > 0 {  fmt.Println("Positive")  } else if num < 0 {  fmt.Println("Negative")  } else {  fmt.Println("Zero")  }  } |

### Switch-Case

Alternative to multiple if-else conditions.

|  |
| --- |
| package main  import "fmt"  func main() {  day := 3  switch day {  case 1:  fmt.Println("Monday")  case 2:  fmt.Println("Tuesday")  case 3:  fmt.Println("Wednesday")  default:  fmt.Println("Other Day")  }  } |

## 6. Loops in Go

### For Loop

|  |
| --- |
| package main  import "fmt"  func main() {  for i := 1; i <= 5; i++ {  fmt.Println(i)  }  } |

### For Loop as While Loop

|  |
| --- |
| package main  import "fmt"  func main() {  i := 0  for i < 5 {  fmt.Println(i)  i++  }  } |

### Range Loop (For Iterating Over a Slice)

|  |
| --- |
| package main  import "fmt"  func main() {  nums := []int{10, 20, 30}  for index, value := range nums {  fmt.Println(index, value)  }  } |

# 4. Functions & Error Handling

## 1. Defining Functions in Go

A **function** is a block of reusable code that performs a specific task.

|  |
| --- |
| func functionName(parameters) returnType {  // Function body  return value  } |

**Example: A Simple Function**

|  |
| --- |
| package main  import "fmt"  // Function to add two numbers  func add(a int, b int) int {  return a + b  }  func main() {  sum := add(5, 3)  fmt.Println("Sum:", sum) // Output: Sum: 8  } |

func add(a int, b int) int:

* a and b are function parameters of type int.
* The function **returns an int** (int after the parameter list).
* The function **returns** the sum using return a + b.

## 2. Returning Multiple Values

Go allows functions to return **multiple values**.

**Example: Returning Multiple Values**

|  |
| --- |
| package main  import "fmt"  // Function that returns two values  func divide(numerator, denominator int) (int, int) {  quotient := numerator / denominator  remainder := numerator % denominator  return quotient, remainder  }  func main() {  q, r := divide(10, 3)  fmt.Println("Quotient:", q) // Output: 3  fmt.Println("Remainder:", r) // Output: 1  } |

* This function **returns two values**: quotient and remainder.
* The function call divide(10, 3) stores values into q and r.

## 3. The defer Statement

* defer **delays** the execution of a function until **the surrounding function returns**.
* **Common use case:** Cleaning up resources (e.g., closing files, unlocking mutexes).

**Example: Using defer**

|  |
| --- |
| package main  import "fmt"  func main() {  fmt.Println("Start")  defer fmt.Println("Deferred Execution")  fmt.Println("End")  } |

* defer fmt.Println("Deferred Execution") **executes last**, even though it appears earlier in the code.
* defer is **useful for cleanup** tasks like closing a file.

**Example: Closing a File Using defer**

|  |
| --- |
| package main  import (  "fmt"  "os"  )  func main() {  file, err := os.Open("test.txt")  if err != nil {  fmt.Println("Error opening file:", err)  return  }  defer file.Close() // Ensures file is closed at the end  fmt.Println("File opened successfully")  } |

## 4. Basic Error Handling

Error handling in Go is **explicit**. Instead of exceptions (like in Python or Java), Go functions return **error values**.

### Using errors.New to Create Errors

The errors package provides errors.New to create **simple error messages**.

|  |
| --- |
| package main  import (  "errors"  "fmt"  )  // Function that returns an error  func divide(a, b int) (int, error) {  if b == 0 {  return 0, errors.New("division by zero is not allowed")  }  return a / b, nil  }  func main() {  result, err := divide(10, 0)  if err != nil {  fmt.Println("Error:", err)  } else {  fmt.Println("Result:", result)  }  } |

* If b == 0, an **error is returned** (errors.New("division by zero is not allowed")).
* If there’s **no error**, nil is returned as the second value.

### Using fmt.Errorf for Formatted Errors

The fmt.Errorf function allows **formatted error messages**.

|  |
| --- |
| package main  import (  "fmt"  )  // Function that generates an error  func greet(name string) (string, error) {  if name == "" {  return "", fmt.Errorf("invalid input: name cannot be empty")  }  return "Hello, " + name, nil  }  func main() {  message, err := greet("")  if err != nil {  fmt.Println("Error:", err)  } else {  fmt.Println(message)  }  } |

If name is empty, fmt.Errorf("invalid input: name cannot be empty") returns an error.

## 5. Structuring Go Programs into Packages

A **package** in Go is a **collection of related functions** in separate files.

* The main package is required for executable programs.
* Other packages can be imported to **reuse code**.

### Creating a Custom Package

**Step 1: Create a Package (mathutils)**

1. Inside your project, create a **new folder** named mathutils.
2. Inside the folder, create mathutils.go.
3. Write the following code:

|  |
| --- |
| package mathutils  // Function to add two numbers  func Add(a, b int) int {  return a + b  } |

* **package mathutils** defines this as a package.
* **func Add(a, b int) int** defines a function inside the package.

**Step 2: Use the Package in main.go**

1. Create a main.go file in the root directory.
2. Import your custom package and use it.

|  |
| --- |
| package main  import (  "fmt"  "mathutils"  )  func main() {  result := mathutils.Add(5, 3)  fmt.Println("Sum:", result)  } |

The mathutils.Add(5, 3) function is **called from another file**.

**Step 3: Run the Code**

To run the program, ensure the package is correctly placed and use:

|  |
| --- |
| go run main.go |

# 4. Structs, Interfaces & Concurrency Basics

## 1. Structs in Go

* A **struct** is a custom data type that groups related variables together.
* It is similar to a class but does not support inheritance.

### Defining a Struct

|  |
| --- |
| package main  import "fmt"  // Define a struct  type Person struct {  Name string  Age int  }  func main() {  // Creating a struct instance  p1 := Person{"Alice", 25}  fmt.Println("Name:", p1.Name) // Accessing fields  fmt.Println("Age:", p1.Age)  } |

* type Person struct {} defines a **struct** with fields Name and Age.
* p1 := Person{"Alice", 25} creates an **instance** of the struct.
* Fields are accessed using p1.Name and p1.Age.

### Using Named Fields (Better Readability)

|  |
| --- |
| p2 := Person{Name: "Bob", Age: 30}  fmt.Println(p2) |

## 2. Methods on Structs

**A method** is a function with a **receiver** (a struct it operates on).

### Example: Method on a Struct

|  |
| --- |
| package main  import "fmt"  // Define a struct  type Rectangle struct {  Width float64  Height float64  }  // Method to calculate area  func (r Rectangle) Area() float64 {  return r.Width \* r.Height  }  func main() {  rect := Rectangle{Width: 5, Height: 3}  fmt.Println("Area:", rect.Area()) // Calls the method  } |

* func (r Rectangle) Area() float64 defines a **method** with r as the **receiver**.
* Methods **can access struct fields**.
* Call methods using rect.Area().

### Pointer Receiver (Modifying Structs)

If a method **modifies the struct**, use a **pointer receiver** (\*).

|  |
| --- |
| package main  import "fmt"  // Define a struct  type Counter struct {  Value int  }  // Method with pointer receiver (modifies struct)  func (c \*Counter) Increment() {  c.Value++  }  func main() {  c := Counter{Value: 10}  c.Increment()  fmt.Println("Updated Value:", c.Value) // 11  } |

* func (c \*Counter) Increment() uses \*Counter to modify Value.
* If we **don't use a pointer**, the struct **won’t be updated**.

## 3. Interfaces in Go

* An **interface** defines a set of methods a type must implement.
* It allows **polymorphism** (different types implementing the same behavior).

### Example: Creating an Interface

|  |
| --- |
| package main  import "fmt"  // Define an interface  type Shape interface {  Area() float64  }  // Struct implementing the interface  type Circle struct {  Radius float64  }  func (c Circle) Area() float64 {  return 3.14 \* c.Radius \* c.Radius  }  func main() {  var s Shape // Declare interface variable  s = Circle{Radius: 5} // Assign a struct implementing the interface  fmt.Println("Area:", s.Area()) // Calls the method  } |

* type Shape interface { Area() float64 } defines an interface.
* Circle **implements** Shape by defining an Area() method.
* **No need for explicit implements keyword** (Go detects it automatically).

### Interfaces with Multiple Implementations

|  |
| --- |
| type Rectangle struct {  Width, Height float64  }  func (r Rectangle) Area() float64 {  return r.Width \* r.Height  }  func main() {  var s Shape  s = Rectangle{Width: 4, Height: 5}  fmt.Println("Rectangle Area:", s.Area())  } |

**Any struct** with an Area() method **automatically implements** Shape.

## 4. Concurrency Basics

Go has **built-in support** for concurrency using **goroutines** and **channels**.

* A **goroutine** is a lightweight thread in Go.
* Use go before a function call to run it concurrently.

### Example: Running a Goroutine

|  |
| --- |
| package main  import (  "fmt"  "time"  )  // Function to run as a goroutine  func printMessage() {  for i := 1; i <= 5; i++ {  fmt.Println("Hello from Goroutine", i)  time.Sleep(time.Millisecond \* 500) // Simulate work  }  }  func main() {  go printMessage() // Start goroutine  fmt.Println("Main function running")  time.Sleep(time.Second \* 3) // Wait for goroutine to complete  } |

* go printMessage() runs the function **in parallel**.
* time.Sleep(time.Second \* 3) ensures the main function **doesn't exit immediately**.

## 5. Introduction to Channels (If Time Permits)

* A **channel** allows goroutines to communicate **safely**.
* Use chan keyword to create channels.

### Example: Using a Channel

|  |
| --- |
| package main  import "fmt"  func main() {  messages := make(chan string) // Create a channel  go func() {  messages <- "Hello, Go!" // Send message into channel  }()  msg := <-messages // Receive message from channel  fmt.Println(msg)  } |

* make(chan string) creates a **channel**.
* messages <- "Hello, Go!" **sends** data.
* <-messages **receives** data.

# Axios and useEffect in React

In your **Task Manager frontend**, we are using **Axios** to communicate with the backend and **useEffect** to handle data fetching when components load. Below, we will thoroughly explain **what Axios is, how HTTP requests work, and how useEffect functions in these scenarios**.

## 1. Understanding Axios

Axios is a **JavaScript library** used to send HTTP requests from the frontend (React) to the backend (Go API). It is commonly used for fetching, creating, updating, and deleting data from APIs.

**Why Use Axios Instead of Fetch?**

JavaScript has a built-in fetch() API for HTTP requests, but Axios is preferred because:

1. **Automatic JSON Handling** – Axios automatically parses JSON responses.
2. **Shorter Syntax** – Axios simplifies request code compared to fetch.
3. **Error Handling** – Axios has better error handling with catch().
4. **Request Timeout Handling** – Axios supports setting timeouts.
5. **Supports Interceptors** – Allows modifying requests or responses before they are handled.

### Installing Axios

If Axios is not already installed in your project, install it using:

|  |
| --- |
| npm install axios |

## 2. Understanding Axios HTTP Methods

HTTP methods define the type of request being sent to the backend. Axios provides different methods corresponding to common HTTP actions.

### Axios Methods and Their Purpose

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **Axios Function** | **Purpose** |
| GET | axios.get(url) | Fetch data from the server |
| POST | axios.post(url, data) | Send new data to the server |
| PUT | axios.put(url, data) | Update existing data on the server |
| DELETE | axios.delete(url) | Remove data from the server |

# 3. How Axios is Used in Task Manager

Below are examples of how Axios is being used in your Task Manager frontend.

## Fetching All Tasks (GET request)

### TaskList.jsx

|  |
| --- |
| useEffect(() => {  axios.get("http://localhost:8080/tasks") // Makes a GET request to fetch tasks  .then(response => setTasks(response.data)) // Sets the response data in state  .catch(() => setError("Failed to load tasks")); // Handles errors  }, []); |

### Breakdown

1. **axios.get("http://localhost:8080/tasks")** – Sends a GET request to fetch all tasks from the backend.
2. **.then(response => setTasks(response.data))** – Stores the retrieved tasks in the React state (setTasks).
3. **.catch(() => setError("Failed to load tasks"))** – If the request fails, it sets an error message.

## Fetching a Single Task (GET request)

### TaskView.jsx

|  |
| --- |
| const { id } = useParams(); // Extracts task ID from URL  useEffect(() => {  axios.get(`http://localhost:8080/tasks/${id}`)  .then(response => setTask(response.data)) // Sets the response data as the task  .catch(() => setError("Task not found")); // Handles errors  }, [id]); |

### Breakdown

1. **useParams()** – Retrieves the id from the URL.
2. **axios.get(http://localhost:8080/tasks/${id}`)`** – Fetches the task with the given id.
3. **.then(response => setTask(response.data))** – Updates the state with the task details.
4. **.catch(() => setError("Task not found"))** – Displays an error message if the task does not exist.

## Creating a New Task (POST request)

### TaskCreate.jsx

|  |
| --- |
| const handleSubmit = async (e) => {  e.preventDefault(); // Prevents page reload  try {  await axios.post("http://localhost:8080/tasks", task); // Sends POST request  alert("Task Created Successfully!");  navigate("/tasks/list"); // Redirects to task list  } catch (error) {  alert("Error creating task");  }  }; |

### Breakdown

1. **e.preventDefault()** – Stops the form from refreshing the page.
2. **await axios.post("http://localhost:8080/tasks", task)** – Sends a POST request with the task data.
3. **navigate("/tasks/list")** – Redirects the user to the task list page after creating the task.
4. **.catch(error => alert("Error creating task"))** – Shows an error message if the request fails.

## Updating a Task (PUT request)

### TaskEdit.jsx

|  |
| --- |
| const handleSubmit = async (e) => {  e.preventDefault();  try {  await axios.put(`http://localhost:8080/tasks/${id}`, task); // Sends PUT request  alert("Task Updated Successfully!");  navigate("/tasks/list"); // Redirects to task list  } catch (error) {  alert("Error updating task");  }  }; |

### Breakdown

1. **await axios.put("http://localhost:8080/tasks/${id}", task)** – Sends updated task data to the backend.
2. **.then(response => alert("Task Updated Successfully!"))** – Alerts success message.
3. **navigate("/tasks/list")** – Redirects to task list.

## Deleting a Task (DELETE request)

### TaskList.jsx

|  |
| --- |
| const deleteTask = async (id) => {  if (window.confirm("Are you sure you want to delete this task?")) { // Confirmation popup  try {  await axios.delete(`http://localhost:8080/tasks/${id}`); // Calls DELETE API  setTasks(tasks.filter(task => task.id !== id)); // Updates UI  } catch (error) {  alert("Error deleting task");  }  }  }; |

### Breakdown

1. **window.confirm("Are you sure?")** – Shows a confirmation popup before deleting.
2. **await axios.delete("http://localhost:8080/tasks/${id}")** – Sends a DELETE request to the backend.
3. **setTasks(tasks.filter(task => task.id !== id))** – Removes the deleted task from the UI.

## 4. Understanding useEffect()

useEffect() is a **React Hook** that allows running side effects (e.g., API calls) **when a component mounts or updates**.

### Syntax

|  |
| --- |
| useEffect(() => {  // Side effect (e.g., API call)  }, [dependencies]); |

* **Runs when the component mounts** (if dependencies array [] is empty).
* **Runs when dependencies change** (if values inside [dependencies] change).

### Fetching Data When Component Loads

|  |
| --- |
| useEffect(() => {  axios.get("http://localhost:8080/tasks")  .then(response => setTasks(response.data))  .catch(() => setError("Failed to load tasks"));  }, []); |

**Explanation**

1. The empty [] means this runs **only once** when the component mounts.
2. Calls axios.get() to **fetch tasks** from the backend.
3. Updates state (setTasks) with the retrieved tasks.

## 5. Summary

|  |  |  |
| --- | --- | --- |
| **Feature** | **Purpose** | **Example** |
| axios.get() | Fetch data from API | axios.get("http://localhost:8080/tasks") |
| axios.post() | Create a new resource | axios.post("http://localhost:8080/tasks", task) |
| axios.put() | Update an existing resource | axios.put("http://localhost:8080/tasks/${id}", task) |
| axios.delete() | Delete a resource | axios.delete("http://localhost:8080/tasks/${id}") |
| useEffect() | Run side effects like API calls | useEffect(() => { axios.get(...) }, []) |