Relative Path and Absolute Path

Paths in React (or any JavaScript application) are used to reference files, resources, or modules. Understanding the differences between relative and absolute paths is essential for clean, maintainable, and scalable code.

Relative Path: A relative path specifies the location of a file relative to the file that references it. It is based on the folder structure and uses . (current directory) or .. (parent directory) to navigate the file hierarchy.

How It Works:

./: Refers to the current directory.

../: Moves one level up in the folder structure.

Usage: Relative paths are typically used to import files that are close to the current file in the folder hierarchy.

Syntax Like,

```
import Component from './Component'; // Current directory
import Component from '../Component'; // Parent directory
import Component from '../../folder/Component'; // Two levels up, then into 'folder'
```

Advantages

- No Configuration Needed: Works out of the box in any React project.
- Clear Relationships: Easy to understand file relationships for small, simple projects.

Disadvantages

- Complexity in Deep Structures: Paths can become long and hard to manage in deeply nested directories.

Example

```
import Component from '../../../../common/Component';
```

Hard to Refactor: Moving files can break multiple imports, requiring significant updates.

Example: Let's consider the folder structure

```
src/
├──── components/
│     ├──── Header.js
│     ├──── Footer.js
├──── App.js
```

Import Header.js in App.js, use the syntax

```
import Header from './components/Header';
```

Import Footer.js in Header.js (same directory), use the syntax

```
import Footer from './Footer';
```

Absolute Path: An absolute path specifies the location of a file or resource starting from a fixed root directory, typically the src folder in React. Absolute paths are not dependent on the location of the referencing file.

How It Works: Absolute paths start from a preconfigured root directory, such as src.

Usage: Absolute paths are used to simplify imports, especially in large projects where files are deeply nested.

Syntax Like,

```
import Component from 'components/Header'; // Starts from 'src'
```

How to Enable Absolute Paths in React: React does not support absolute paths by default. You need to configure it:

- Create a jsconfig.json file in the root directory (or tsconfig.json for TypeScript) and add the following configuration:

```
{
  "compilerOptions": {
    "baseUrl": "src"  },
"include": ["src"]
}
```

- Restart the development server.

Advantages

- Clean Imports: Simplifies import statements, making them easier to read and manage.
- Easier Refactoring: Moving files does not require updating imports as long as the relative folder structure is maintained.
- Scalability: Ideal for large-scale applications with deeply nested directories.

Disadvantages

- Requires Setup: Needs configuration, which may be challenging for beginners.
- Potential Confusion: Developers unfamiliar with the setup may find it less intuitive.

Example: Let's consider the same folder structure, as previous

```
src/
├── components/
│   ├── Header.js
│   └── Footer.js
└── App.js
```

Import Header.js in App.js

```
import Header from 'components/Header';
```

Why Use tsconfig.json or jsconfig.json?

- **tsconfig.json**: Used for TypeScript projects to define TypeScript-specific settings like type checking, module resolution, and path aliases.
- **jsconfig.json**: Used for JavaScript projects with no TypeScript. It serves a similar purpose, focusing on enabling IDE support and path mapping.

Both files allow you to configure absolute imports for your project and improve development experience in editors like VS Code.

Creating and Configuring tsconfig.json or jsconfig.json

If Using TypeScript (tsconfig.json)

- Create a tsconfig.json in the root directory (if it doesn't already exist).
- Add the following configuration:

```
{
  "compilerOptions": {
    "baseUrl": "./",        // Sets the base directory for resolving non-relative paths
    "paths": {
      "@/*": ["src/*"]  } },    // Maps '@' to the 'src' folder
  "include": ["src"]         // Include the 'src' folder in the project
}
```

If Using JavaScript (jsconfig.json)

- Create a jsconfig.json in the root directory (if it doesn't already exist).
- Add the following configuration:

```
{
  "compilerOptions": {
    "baseUrl": "./",        // Sets the base directory for resolving non-relative paths
    "paths": {
      "@/*": ["src/*"]    } },     // Maps '@' to the 'src' folder
  "include": ["src"]         // Include the 'src' folder in the project
}
```

Integrating with Vite Configuration

To ensure the path alias works in both the build process and your editor, you need to add the alias in both vite.config.js (or vite.config.ts) and the tsconfig.json/jsconfig.json.

Vite Configuration (vite.config.js or vite.config.ts):

```
import { defineConfig } from 'vite';
import path from 'path';
export default defineConfig({
  resolve: {
    alias: {
      '@': path.resolve(__dirname, './src'), // Alias '@' to 'src'
    },  },
});
```

Restart the Development Server

After creating or modifying these configuration files, restart the Vite development server and your IDE (e.g., VS Code) to apply the changes.

Using the Alias in Imports

With the configuration in place, you can now use absolute paths in your imports.

Before (Relative Path)

```
import Header from '../../../components/Header';
```

After (Absolute Path)

```
import Header from '@/components/Header';
```

Verification

- **IDE Support:** Open your project in an IDE like VS Code. You should see proper autocompletion and path resolution for aliases.
- **Runtime:** Run your Vite project to ensure the paths resolve correctly.

Both tsconfig.json and jsconfig.json improve maintainability and scalability by simplifying imports and enhancing IDE support. Use tsconfig.json for TypeScript and jsconfig.json for JavaScript projects.

Getting error : error __dirname is not defined?

The error __dirname is not defined occurs because Vite uses ES Modules (type: "module" in package.json) by default, and __dirname is a CommonJS global variable that isn't available in ES Modules.
To fix this issue, you can define a custom equivalent for __dirname or use import.meta.url to resolve paths in an ES Modules environment.

Solution 1: Define __dirname in ES Modules

You can define __dirname manually using import.meta.url and the path module:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import path from 'path';
import { fileURLToPath } from 'url';
// Define __dirname for ES Modules
const __dirname = path.dirname(fileURLToPath(import.meta.url));

export default defineConfig({
  plugins: [react()],
  resolve: {
    alias: {
      '@': path.resolve(__dirname, './src'), // Alias '@' to 'src'
    },
  },
});
```

Solution 2: Use import.meta.url, directly

Instead of defining __dirname, you can use import.meta.url directly to resolve paths:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import path from 'path';
import { fileURLToPath } from 'url';

export default defineConfig({
  plugins: [react()],
  resolve: {
    alias: {
      '@': path.resolve(path.dirname(fileURLToPath(import.meta.url)), './src'),
          // Alias '@' to 'src'
    },
  },
});
```

useNavigate Hook

- The useNavigate hook is provided by React Router to programmatically navigate between routes in your application.
- It replaces the older useHistory hook (React Router v5) and provides a simplified API to control navigation.
- The useNavigate hook is one of the most versatile tools provided by React Router for programmatic navigation.
- It empowers developers to dynamically control navigation in response to application events, such as user interactions, API responses, or business logic conditions.
- The useNavigate hook is a robust and versatile tool that enables fine-grained control over navigation in React Router.
- Its ability to handle dynamic paths, state passing, and history stack manipulation makes it indispensable for building modern, interactive web applications.

Key Concepts of useNavigate

- **Programmatic Navigation**
  - useNavigate allows developers to navigate between routes programmatically without relying on declarative components like <Link> or <NavLink>.
  - It's particularly useful for handling navigation in scenarios such as form submissions, conditional redirects, or when specific business logic determines the next page.
- **Supports Navigation Control:** The useNavigate hook enables two primary forms of navigation:
  - Push Navigation (default): Adds a new entry to the browser history stack.
  - Replace Navigation: Replaces the current entry in the history stack.
- **Relative and Absolute Navigation**
  - Navigate to a route relative to the current URL or as an absolute path.
  - Relative navigation is especially useful in nested routing scenarios.
- **Pass State Between Pages**
  - Allows passing custom state data during navigation. The state can be accessed in the target route using the useLocation hook.
  - This eliminates the need for a global state or URL parameters for temporary data sharing.

- **Backward and Forward Navigation:** Enables navigating back and forth in the browser's history stack using integer values (e.g., -1 for one step back or 1 for one step forward).
- **Declarative Routing Alternative:** While <Link> components are declarative and preferred for static navigation, useNavigate is indispensable for scenarios where navigation depends on runtime conditions or user actions.

Use Cases

- **Redirect After Successful Actions:** Navigate to a confirmation page after successfully submitting a form.
  Example: Redirect to a dashboard page after user login or signup.
- **Conditional Navigation:** Navigate users to different pages based on their role, permissions, or specific app states.
- **Dynamic Routing:** Programmatically navigate based on user input or application state. Dynamically navigate to routes constructed at runtime (e.g., /products/${id}).
- **Custom State Passing:** Temporarily pass data between pages without exposing it in the URL (e.g., passing a confirmation message after a form submission).
- **Breadcrumb Navigation:** Implement backward navigation for multi-step forms or hierarchical pages.

Syntax

navigate(to, options)

where,

- to: A string (URL) or object specifying the target location. Can be relative or absolute.
- options:
  - replace: Boolean indicating whether to replace the current entry in the history stack. Default is false.
  - state: Custom state object to pass to the target route.

Example: Basic Navigation

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes, useNavigate } from 'react-router-dom';

// HomePage Component
function HomePage() {
  const navigate = useNavigate();
  const goToAbout = () => {
    navigate('/about');
  };
  return (
    <div>
      <h1>Home Page</h1>
      <button onClick={goToAbout}>Go to About</button>
    </div>
  );}
// AboutPage Component
function AboutPage() {
  return (
    <div>
      <h1>About Page</h1>
      <p>Welcome to the About Page!</p>
    </div>
  );}
// App Component with Routing
function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/about" element={<AboutPage />} />
      </Routes>
    </Router>
  );
}
export default App;
```

Example: Navigate with State

```
import { BrowserRouter as Router, Route, Routes, useNavigate, useLocation } from
'react-router-dom';
function LoginPage() {                    // LoginPage Component
  const navigate = useNavigate();
  const handleLogin = () => {
    navigate('/dashboard', { state: { user: 'Garima', role: 'Programmer' } });
  };
  return (
    <div>
      <h1>Login Page</h1>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}
function Dashboard() {                    // Dashboard Component
  const location = useLocation();
  const { user, role } = location.state || {};
  return (
    <div>
      <h1>Welcome, {user || 'Guest'}!</h1>
      <p>Role: {role || 'Only Visitor'}</p>
    </div>
  );
}
function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<LoginPage />} />
        <Route path="/dashboard" element={<Dashboard />} />
      </Routes>
    </Router>
  );
}
export default App;
```

Example: Relative Navigation

```
import { BrowserRouter as Router, Route, Routes, useNavigate } from 'react-router-dom';
function InstitutePage() {              // InstitutePage Component
  const navigate = useNavigate();
  const viewDetails = () => {
    navigate('details'); // Relative path navigation
  };
  return (
    <div>
      <h1>Information Page</h1>
      <button onClick={viewDetails}>View Institute Details</button>
    </div>
  );}

function InstituteDetails() {          // InstituteDetails Component
  return (
    <div>
      <h1>Learn2Earn Labs Training Institute</h1>
       <p>Best Training Institute for Job Oriented Training Programs and Career Development.</p>
    </div>
  );
}

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/institute" element={<InstitutePage />} />
        <Route path="/institute/details" element={<InstituteDetails />} />
      </Routes>
    </Router>
  );
}
export default App;
```

Example: Replace Navigation (confirmation and success workflow)

```
import { BrowserRouter as Router, Route, Routes, useNavigate } from 'react-router-dom';
function ConfirmationPage() {       // ConfirmationPage Component
  const navigate = useNavigate();
  const handleConfirm = () => {
    navigate('/success', { replace: true }); // Replaces the current entry in the history stack
  };
  return (
    <div>
      <h1>Confirmation Page</h1>
      <button onClick={handleConfirm}>Confirm</button>
    </div>
  );}
function SuccessPage() {            // SuccessPage Component
  return (
    <div>
      <h1>Success!</h1>
      <p>Your action has been successfully completed.</p>
    </div>
  );}
function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<ConfirmationPage />} />
        <Route path="/success" element={<SuccessPage />} />
      </Routes>
    </Router>
  );}
export default App;
```

Note: The ConfirmationPage serves as a step where the user can review or confirm an action before proceeding. After the user confirms, they are navigated to a SuccessPage, which displays a success message or confirmation of the completed action. The navigate('/success', { replace: true }) ensures that the user cannot go back to the ConfirmationPage after confirming.

Example: Replace Navigation

The example simulates a scenario where the user is redirected to a success page after submitting a form, but the navigation entry is replaced to prevent the user from going back to the form page using the browser's back button.

```
import { useState } from 'react';
import { BrowserRouter as Router, Routes, Route, useNavigate } from 'react-router-dom';

// Form Page Component
function FormPage() {
  const navigate = useNavigate();
  const [formData, setFormData] = useState({ name: '', email: '' });

  const handleSubmit = (e) => {
    e.preventDefault();
    // Perform form validation or submission logic here
    console.log('Form Submitted:', formData);
    // Navigate to the success page and replace the current entry
    navigate('/success', { replace: true, state: { name: formData.name } });
  };

  return (
    <div>
      <h1>Form Page</h1>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Name:</label>
          <input type="text" value={formData.name}
            onChange={(e) => setFormData({ ...formData, name: e.target.value })} />
        </div>
        <div>
          <label>Email:</label>
          <input type="email" value={formData.email}
            onChange={(e) => setFormData({ ...formData, email: e.target.value })} />
        </div>
```

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

// Success Page Component
function SuccessPage() {
  const navigate = useNavigate();
  const handleGoBack = () => {
    navigate('/', { replace: true }); // Redirect to home, replacing the current entry
  };

  return (
    <div>
      <h1>Success Page</h1>
      <p>Your form has been successfully submitted.</p>
      <button onClick={handleGoBack}>Go to Home</button>
    </div>
  );
}

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<FormPage />} />
        <Route path="/success" element={<SuccessPage />} />
      </Routes>
    </Router>
  );
}

export default App;
```

what happened if we use replace:false?

If you use replace: false (which is the default behavior of the useNavigate hook), the navigation adds a new entry to the browser's history stack instead of replacing the current entry.

replace: false (Default Behavior)

- A new entry is added to the history stack.
- The user can navigate back to the previous page using the browser's back button.

replace: true

- The current entry in the history stack is replaced by the new one.
- The user cannot navigate back to the replaced page using the browser's back button.

When to Use replace: true

- To prevent users from revisiting a page (e.g., a login or checkout form after submission).
- To ensure the navigation stack only contains meaningful pages (e.g., avoid keeping intermediate steps in history).
- For secure pages where you don't want users to access sensitive data by going back.

When to Use replace: false

- When users might need to go back to the previous page (e.g., a product page after adding an item to the cart).
- For multi-step forms where users may need to revisit previous steps.

Example: Navigate back in the history

```
import { BrowserRouter as Router, Route, Routes, useNavigate, Link } from 'react-router-dom';

function HomePage() {                // HomePage Component
 return (
  <div>
   <h1>Home Page</h1>
   <Link to="/details">  <button>Go to Details Page</button>  </Link>
  </div>
 );
}

function DetailsPage() {             // DetailsPage Component
 const navigate = useNavigate();
 const goBack = () => {
  navigate(-1);      // Navigate one step back in the history
 };
 return (
  <div>
   <h1>Details Page</h1>
   <button onClick={goBack}>Go Back</button>
  </div>
 );
}

function App() {
 return (
  <Router>
   <Routes>
    <Route path="/" element={<HomePage />} />
    <Route path="/details" element={<DetailsPage />} />
   </Routes>
  </Router>
 );
}
export default App;
```

Example: Go Back 3 Steps

```
import { BrowserRouter as Router, Route, Routes, useNavigate, Link } from 'react-router-dom';

// HomePage Component
function HomePage() {
  return (
    <div>
      <h1>Home Page</h1>
      <nav>
        <Link to="/aboutus"><button>About Us</button></Link>
        <Link to="/contactus"><button>Contact Us</button></Link>
        <Link to="/services"><button>Services</button></Link>
      </nav>
    </div>
  );
}

// AboutUs Component
function AboutUs() {
  return (
    <div>
      <h1>About Us</h1>
        <p>Welcome to Learn2Earn Labs Training Institute! We are dedicated to empowering individuals with job-oriented training programs to help them achieve their career goals.</p>
      <Link to="/"><button>Go Back Home</button></Link>
    </div>
  );
}

// ContactUs Component
function ContactUs() {
  return (
    <div>
      <h1>Contact Us</h1>
```

```
      <p>If you have any questions or want to learn more, feel free to reach out to us:</p>
      <ul>
        <li>Email: team@learn2earnlabs.com</li>
        <li>Phone: +91-9548868337</li>
        <li>website: www.learntoearnlabs.com</li>
       <li>Address: Learn2Earn Labs Training Institute, Sikandra, Agra, Uttar Pradesh, India.</li>
      </ul>
      <Link to="/"><button>Go Back Home</button></Link>
    </div>
  );
}

// Services Component
function Services() {
  const navigate = useNavigate();

  const goBackThreeSteps = () => {
    navigate(-3); // Navigate 3 steps back in the history stack
  };
  return (
    <div>
      <h1>Our Services</h1>
      <h2>Learn2Earn Labs Training Institute</h2>
      <p>
        We offer job-oriented training programs designed to help you build a successful career. Our programs include:
      </p>
      <ul>
        <li>Full Stack Web Development</li>
        <li>MERN Stack Development</li>
        <li>Java Full Stack Development</li>
        <li>Cloud Computing & DevOps</li>
        <li>Design Thinking & Innovation</li>
        <li>Digital Marketing</li>
      </ul>
```

```
      <h3>Our Specialities</h3>
      <ul>
        <li>Practical-based training</li>
        <li>Hands-on project work</li>
        <li>Real-world working experience</li>
        <li>Valid certification upon completion</li>
        <li>Job package guarantee</li>
      </ul>
      <Link to="/"><button>Go Back Home</button></Link>
      <button onClick={goBackThreeSteps}>Go Back 3 Steps</button>
    </div>
  );
}

// App Component with Routing
function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/aboutus" element={<AboutUs />} />
        <Route path="/contactus" element={<ContactUs />} />
        <Route path="/services" element={<Services />} />
      </Routes>
    </Router>
  );
}

export default App;
```

Benefits of useNavigate

- **Declarative to Imperative Transition:** Transforms navigation from declarative (static links) to imperative (dynamic logic), providing greater flexibility.
- **Simplified API:** Combines functionality that was previously scattered across older APIs like useHistory into a single, unified interface.
- **Cleaner Code:** Encourages a separation of concerns by handling navigation logic explicitly where needed, rather than coupling it with presentation components.
- **State Isolation:** Prevents global state pollution by allowing temporary state to be passed directly during navigation.
- **Backward Compatibility:** Retains the ability to handle navigation similar to older methods (like history.push) while modernizing the API.

Best Practices

- **Use Declarative Navigation When Possible:** Prefer <Link> or <NavLink> for static navigation, reserving useNavigate for scenarios that require dynamic routing.
- **Validate Navigation Paths:** Ensure that dynamically constructed paths are valid and handle invalid paths gracefully.
- **Combine with Error Handling:** Use try-catch blocks or conditional logic to gracefully handle failed navigation or invalid states.
- **Optimize State Passing:** Only pass essential state data during navigation to reduce unnecessary memory usage.
- **Avoid Overusing History Manipulation:** Use backward/forward navigation sparingly to maintain a predictable user experience.