TanStack Router: Official Getting Started Guide

An in-depth walkthrough to help you set up and utilize TanStack Router for modern web applications, based on the latest official documentation.

Table of Contents

What is TanStack Router?

Overview of TanStack Router

TanStack Router is a modern, flexible, and powerful routing library designed to handle navigation in web applications. Built with performance and simplicity in mind, it offers an intuitive API that makes defining and managing routes straightforward across various frameworks.

Features and Benefits

- **Framework Agnostic:** Works seamlessly with React, Vue, Svelte, and other JavaScript frameworks.
- **Declarative Routing:** Easily specify routes using a clear, declarative syntax.
- **Nested Routing:** Supports complex nested layouts and route hierarchies.
- **Data Loading:** Built-in support for data fetching with loaders and mutations.
- **Type Safety:** Strong TypeScript support to catch errors early.
- **Flexible Navigation:** Programmatic and declarative navigation options.
- **SSR & Static Generation:** Compatible with server-side rendering and static site generation.
- **Extensible & Customizable:** Hooks and APIs to tailor routing behavior to your needs.

Use Cases

TanStack Router is ideal for:

- Large-scale single-page applications (SPAs)
- Static websites with complex navigation
- Server-rendered applications that require hydration
- Multi-framework projects needing consistent routing solutions
- Apps requiring advanced features like route guards, transitions, and code splitting

In summary, TanStack Router aims to provide a robust routing solution that emphasizes developer experience, performance, and adaptability, making it a compelling choice for modern web development projects.

Why Choose TanStack Router?

Comparison with Other Routing Libraries

TanStack Router stands out among other routing solutions due to its modern architecture, comprehensive feature set, and flexibility. Unlike traditional routers, TanStack Router is built with TypeScript-first design, ensuring strong type safety and autocomplete support, which minimizes runtime errors and improves developer experience. Its declarative API aligns well with contemporary frameworks and allows for seamless integration across different environments.

Advantages in Modern Web Development

- **Type Safety**: Leverages TypeScript extensively, providing compile-time guarantees for route parameters, search params, and more.
- **Performance**: Optimized for minimal bundle size and fast navigation, supporting features like code splitting and deferred loading.
- **Flexibility**: Supports static, dynamic, nested, and layout routes with ease, accommodating complex application architectures.
- **Universal Compatibility**: Designed to work seamlessly with React, Vue, Svelte, and other frameworks, fostering cross-framework development.
- **Rich Data Layer**: Built-in support for data loading, mutations, and state management, reducing boilerplate and streamlining data flow.
- **Advanced Features**: Includes route guards, animations, transition states, and SSR capabilities, enabling feature-rich applications.

Supported Frameworks

TanStack Router's versatility is evident in its broad support for various frameworks:

- **React**: Fully integrated with React's component model, hooks, and context.
- **Vue.js**: Compatible with Vue 3's Composition API and reactive system.
- **Svelte**: Adapts well with Svelte's reactivity and component structure.
- **Others**: Can be integrated with other JavaScript frameworks or used in vanilla JS projects with custom adapters.

Choosing TanStack Router equips your projects with a modern, performant, and type-safe routing solution that scales with your application's complexity and framework preferences.

Prerequisites

Before you start using TanStack Router, ensure your development environment meets the following requirements:

Node.js and npm/Yarn

- **Node.js:** Make sure you have Node.js installed on your machine. We recommend using the latest LTS version. You can download it from Node.js official website.

- **Package Managers:** Have either npm (comes bundled with Node.js) or Yarn installed. For Yarn, visit Yarn's official site.

Framework Compatibility

TanStack Router supports various frontend frameworks. Currently, it is compatible with:

- **React (recommended version: v18.x.x or v19.x.x):** Ensure React and ReactDOM are installed.

- **Vue.js, Svelte, and others:** Support depends on the adapters and integration methods. Check the specific documentation for your framework.

Basic Routing Concepts

A fundamental understanding of routing principles is helpful:

- **Routes:** Mappings between URLs and components.

- **Navigation:** How users move between routes.

- **Dynamic Parameters:** URL parts that can change.

- **Query Parameters:** Key-value pairs appended to URLs.

- **Nested Routes:** Hierarchical route structures.

Familiarity with these concepts will facilitate a smoother setup and integration process.

Note: If you're new to routing in web applications, consider reviewing basic routing tutorials tailored to your framework before proceeding with TanStack Router integration.

Installation

Getting TanStack Router up and running is straightforward. Follow these steps to integrate it into your project and start building your routes.

Installing via npm/yarn

To install TanStack Router, run one of the following commands in your project directory:

```
npm install @tanstack/react-router
```

or

```
yarn add @tanstack/react-router
```

Make sure to also install any peer dependencies if prompted, such as React or other frameworks related packages.

Setting up a new project

If you're starting a new project, you can quickly set up using your preferred scaffolding tool like Create React App, Vite, or Next.js. Once your project environment is ready, install TanStack Router as shown above.

Installing peer dependencies

TanStack Router integrates with frameworks like React, Vue, or Svelte. Ensure that the relevant framework's dependencies are installed:

- For React projects:

```
npm install react react-dom
```

- For Vue projects:

```
npm install vue
```

- For Svelte projects:

```
npm install svelte
```

Always verify the latest peer dependencies required for your framework version from the official documentation.

Summary

- Use npm or yarn to install `@tanstack/react-router`.
- Set up your project with necessary framework dependencies.
- Follow your framework-specific setup guides to initialize the router and configure your routes.

Once installed, you can proceed to configure your router instance and define your routes to start managing navigation in your application efficiently.

Basic Setup

Setting up TanStack Router in your project is straightforward. This chapter guides you through initializing a router instance, creating your first routes, and understanding how to configure routes effectively.

Initializing a Router Instance

Begin by importing the necessary functions from TanStack Router and creating a router instance:

```
import { createRouter } from '@tanstack/react-router'

// Create a new router instance
const router = createRouter({
  // route configuration will go here
})
```

This `createRouter` function initializes your router with specified configurations.

Creating Your First Routes

Routes define the navigation paths in your application. Here's a simple example of setting up basic routes:

```
const routeTree = {
  id: 'root',
  children: [
    {
      id: 'home',
      path: '/',
      component: HomePage,
    },
    {
      id: 'about',
      path: '/about',
      component: AboutPage,
    },
  ],
}
```

Replace `HomePage` and `AboutPage` with your actual React component imports. To use these routes, pass the route tree during router creation:

```
const router = createRouter({ routeTree })
```

Understanding Route Configuration

Routes can be configured with various properties:

- `path`: URL pattern matching for the route.
- `component`: React component rendered for the route.
- `children`: Nested routes for hierarchical navigation.
- `loader`: Function to load data before rendering.
- `action`: Function to handle form submissions or mutations.

Example with additional configurations:

```
{
  id: 'profile',
  path: '/profile/:userId',
  component: UserProfile,
  loader: fetchUserData,
}
```

Tip: Always define a catch-all route for handling 404 errors or undefined paths.

Summary

- Import `createRouter` from TanStack Router.
- Define your route tree with paths and components.
- Instantiate the router with your route configuration.
- Use router-aware components to enable navigation.

By following these steps, you'll establish a solid foundation for route management in your application.

Route Definitions

Understanding route definitions is fundamental to building effective navigation structures with TanStack Router. This section outlines how to declare static, dynamic, and nested routes to suit various application needs.

Static Routes

Static routes map fixed URL paths directly to route configurations. These are best for pages with unchanging paths, such as the homepage or about page.

```
import { createRouter, RouterProvider, Route } from '@tanstack/react-router';

const router = createRouter({
  routes: [
    {
      path: '/',
      element: () => <HomePage />,
    },
    {
      path: '/about',
      element: () => <AboutPage />,
    },
  ],
});

function App() {
  return (
    <RouterProvider router={router} />
  );
}
```

- **Key Points:**
    - Use a fixed `path` string.
    - Associate each route with an `element` or component.
    - Ideal for static pages.

Dynamic Routes with Params

Dynamic routes include parameters, allowing URLs to contain variable segments. These are useful for user profiles, product pages, etc.

```
const productRoute = {
  path: '/product/:productId',
  element: () => <ProductPage />,
};
```

- **Usage:**
    - The `:parameterName` syntax defines a route parameter.
    - Access parameters via hooks or props.

```
function ProductPage() {
  const params = useParams();
  return <div>Product ID: {params.productId}</div>;
}
```

- **Example URLs:**
  - `/product/123`
  - `/product/abc`

Nested Routes

Nested routes enable hierarchical view structures, often used with layouts or shared components.

```
const dashboardRoute = {
  path: '/dashboard',
  element: () => <DashboardLayout />,
  children: [
    {
      path: 'stats',
      element: () => <StatsPage />,
    },
    {
      path: 'settings',
      element: () => <SettingsPage />,
    },
  ],
};
```

- **Features:**
  - Define child routes inside a parent.
  - Use layout components to wrap nested views.
  - URLs become `/dashboard/stats`, `/dashboard/settings`.

Route Configuration Tips

- Always use explicit `path` strings.
- For nested routes, specify `children`.
- Use route parameters for dynamic segments.
- Combine static and dynamic segments as needed.

Mastering route definitions allows you to craft intuitive and scalable navigation structures tailored to your application's architecture.

Navigation and Linking

Effective navigation and linking are essential for creating a seamless user experience in web applications using TanStack Router. This chapter explores the core concepts and techniques for programmatic navigation, declarative links, and styling active links.

Programmatic Navigation

Programmatic navigation allows you to change routes dynamically within your application based on user actions or other logic. TanStack Router provides hooks and functions to facilitate this:

```
import { useNavigate } from '@tanstack/react-router';

function MyComponent() {
  const navigate = useNavigate();

  const goToProfile = () => {
    navigate({ to: '/profile' });
  };

  return (
    <button onClick={goToProfile}>Go to Profile</button>
  );
}
```

- **useNavigate**: Hook that returns a function to trigger route changes.
- **navigate**: Function that accepts route parameters and performs navigation.

Declarative Links

Declarative links enable users to navigate through your app using `<Link>` components, which are similar to traditional anchor tags but optimized for single-page applications.

```
import { Link } from '@tanstack/react-router';

function Navigation() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/profile">Profile</Link>
    </nav>
  );
}
```

Benefits of Declarative Links

- Preloading routes on hover or focus.
- Managing active styling automatically.
- Handling client-side navigation seamlessly.

Active Link Styling

Highlighting the current active link enhances user orientation.

```
import { Link, useMatch } from '@tanstack/react-router';

function Navigation() {
  const isActive = (routePath) => useMatch({ to: routePath });

  return (
    <nav>
      <Link
        to="/"
        style={{ fontWeight: isActive('/') ? 'bold' : 'normal' }}
      >
        Home
      </Link>
      <Link
        to="/about"
        style={{ fontWeight: isActive('/about') ? 'bold' : 'normal' }}
      >
        About
      </Link>
    </nav>
  );
}
```

Alternatively, use the `active` class:

```
<Link
  to="/"
  className={({ isActive }) => (isActive ? 'active-link' : undefined)}
>
  Home
</Link>
```

And define CSS:

```
.active-link {
  font-weight: bold;
  text-decoration: underline;
}
```

Summary

Navigation and linking in TanStack Router empower developers to create intuitive and accessible web applications. Use programmatic navigation for dynamic route changes, declarative `<Link>` components for user navigation, and active styling to indicate current routes. Combining these techniques results in a coherent and user-friendly navigation experience.

Route Matching & Parameters

Understanding how TanStack Router matches routes and handles parameters is essential for creating dynamic and flexible navigation.

Matching Strategies

TanStack Router offers several strategies for route matching:

- **Exact Matching**: Routes match only when the URL path exactly corresponds to the route pattern.
- **Prefix Matching**: Routes match if the URL begins with the route pattern, useful for nested or layout routes.
- **Pattern Matching**: Using route patterns with dynamic segments to match multiple URLs.

The selection of a matching strategy depends on your application's structure and routing requirements. The router efficiently determines the active route based on the current URL using these strategies, enabling predictable navigation behavior.

Using Route Params

Route parameters allow you to create routes that respond to variable segments in URLs. You define params in your route configuration:

```
{
  path: "/user/:id",
  component: UserProfile
}
```

Accessing Params

Once matched, params are accessible within your components or loaders via hooks or context:

```
import { useParams } from '@tanstack/react-router'

function UserProfile() {
  const params = useParams()
  return <div>User ID: {params.id}</div>
}
```

Optional and Query Params

- **Optional Params**: Defined with a question mark (`?`) to handle optional segments.
- **Query/Search Params**: Accessed via URL's search parameters, often using built-in utilities or hooks.

```
// Example of retrieving query params
const searchParams = new URLSearchParams(location.search)
const filter = searchParams.get('filter')
```

Proper handling of params enables pattern matching for dynamic routes and supports complex URL structures, enhancing user experience and SEO.

Fetching Data

In TanStack Router, data fetching is primarily managed through the use of loaders. Loaders allow you to fetch data before a route is rendered, enabling seamless data-driven experiences within your application. This chapter covers how to effectively implement data fetching and optimize its performance.

Loaders

Loaders are functions associated with routes that run during navigation, providing data necessary for rendering the route's component. They can run on the server for SSR, or on the client during client-side navigation.

Defining a Loader

To define a loader, attach a `loader` property to your route configuration:

```
import { createRoute } from '@tanstack/react-router'

const userRoute = createRoute({
  path: '/user/:id',
  element: <UserComponent />,
  loader: async ({ params }) => {
    const response = await fetch(`/api/users/${params.id}`)
    const user = await response.json()
    return { user }
  },
})
```

Accessing Loader Data

Within your route components, data provided by loaders can be accessed via hooks or props. For example, using React hooks:

```
import { useLoaderData } from '@tanstack/react-router'

function UserComponent() {
  const data = useLoaderData()
  return <div>{data.user.name}</div>
}
```

Fetching on Route Load

Loaders are invoked when a route is initially loaded or when navigated to programmatically. This ensures your components receive up-to-date data.

Preloading Data

TanStack Router supports preloading data for better user experience:

```
router.prefetch('/user/123')
```

This fetches data ahead of time, reducing latency during navigation.

Data Caching

To optimize performance, loaders can implement caching strategies:

- **In-Memory Caching:** Store responses in memory to avoid redundant fetches.
- **Persistent Caching:** Use IndexedDB or localStorage for longer-term caching.
- **Stale-While-Revalidate:** Show cached data while fetching fresh data in the background.

Sample implementation of in-memory cache:

```
const cache = new Map()

const fetchWithCache = async (key, fetchFn) => {
  if (cache.has(key)) {
    return cache.get(key)
  }
  const data = await fetchFn()
  cache.set(key, data)
  return data
}
```

Use this within your loader to avoid unnecessary network requests.

Handling Load States

Managing loading and error states enhances user experience and robustness.

Loading Indicators

Display loading spinners or skeletons while data is being fetched:

```
import { useNavigation, useLoaderData } from '@tanstack/react-router'

function UserComponent() {
  const navigation = useNavigation()
  const data = useLoaderData()

  if (navigation.state.status === 'loading') {
    return <div>Loading...</div>
  }

  return <div>{data.user.name}</div>
}
```

Error Handling

Handle errors gracefully within loaders:

```
loader: async ({ params }) => {
  try {
    const response = await fetch(`/api/users/${params.id}`)
    if (!response.ok) throw new Error('Failed to fetch')
    const user = await response.json()
    return { user }
  } catch (error) {
    // Handle error, e.g., route to error page or show message
  }
}
```

Retry Logic

Implement retries with exponential backoff to improve resilience:

```
const fetchWithRetry = async (url, retries = 3) => {
  for (let i = 0; i < retries; i++) {
    try {
      const response = await fetch(url)
      if (response.ok) return response.json()
    } catch {}
    await new Promise((resolve) => setTimeout(resolve, Math.pow(2, i) * 1000))
  }
  throw new Error('Failed after retries')
}
```

Summary

Effective data fetching in TanStack Router involves:

- Defining loaders for routes
- Accessing loader data within components
- Preloading and caching responses
- Managing loading and error states gracefully
- Implementing retries for robustness

Mastering these techniques ensures your applications are performant, resilient, and deliver a smooth user experience.

Mutating Data

Mutating data in TanStack Router involves performing actions that change the application's state or backend data based on user interactions or other triggers. This process is essential for handling form submissions, updating records, toggling features, or any other operation that modifies data.

Core Concepts

Actions

Actions are functions responsible for executing data mutations. They can communicate with APIs, perform database updates, or manipulate local state. Actions are typically triggered through user events like form submissions or button clicks.

Form Submissions

Handling form data is a common use case for mutations. TanStack Router provides mechanisms to handle form submissions seamlessly, allowing you to associate forms with specific actions to update data accordingly.

Optimistic Updates

Optimistic updates improve user experience by immediately reflecting changes in the UI before receiving server confirmation. This approach requires careful handling of potential errors to maintain data consistency.

Implementing Mutations

Using Actions

To create and execute a mutation, define an action function and invoke it where needed:

```
import { createAction } from '@tanstack/react-router'

const updateUser = createAction(async (userData) => {
  const response = await fetch('/api/user', {
    method: 'POST',
    body: JSON.stringify(userData),
    headers: {
      'Content-Type': 'application/json',
    },
  })
  if (!response.ok) {
    throw new Error('Failed to update user')
  }
  return response.json()
})

// Trigger the action on form submit
function handleSubmit(event) {
  event.preventDefault()
  const formData = new FormData(event.target)
  const data = {
    name: formData.get('name'),
    email: formData.get('email'),
  }
  updateUser.submit(data)
}
```

Using Form.Submit

TanStack Router offers convenient components like `Form.Submit` for handling form submissions declaratively:

```
import { Form } from '@tanstack/react-router'

function UserForm() {
  return (
    <Form action={updateUser}>
      <input name="name" placeholder="Name" />
      <input name="email" placeholder="Email" />
      <button type="submit">Save</button>
    </Form>
  )
}
```

Handling Mutations and Feedback

You can listen to the success or error of actions to update the UI accordingly:

```
useEffect(() => {
  if (updateUser.status === 'success') {
    alert('User updated successfully!')
  } else if (updateUser.status === 'error') {
    alert(`Error: ${updateUser.error.message}`)
  }
}, [updateUser.status])
```

Best Practices

- **Validate Data Before Mutating**: Ensure data integrity by validating inputs client-side and server-side.
- **Handle Loading and Error States**: Provide user feedback during mutation processes.
- **Use Optimistic Updates Wisely**: Implement with caution when real-time feedback enhances user experience.
- **Secure Mutations:** Authenticate and authorize mutation requests to prevent unauthorized data changes.

Summary

Mutating data is a vital part of dynamic applications. With TanStack Router, actions and form components enable seamless data mutations, offering a declarative and intuitive way to handle state changes and backend updates. Proper handling of load states, error management, and user feedback ensures a smooth user experience.

Handling Load States

Effective management of load states is crucial for creating responsive and user-friendly applications with TanStack Router. This section covers strategies to handle loading indicators, errors, and retries to ensure a seamless user experience during data fetching and navigation.

Loading Indicators

Displaying a visual cue during data loading helps users understand that a process is underway. TanStack Router provides hooks and options to implement loading indicators:

- Use *loader* functions to fetch data asynchronously.
- Show spinners, progress bars, or skeleton screens based on the router's load state.
- Example:

```
import { useRouteLoaderData } from '@tanstack/react-router'

function DataDisplay() {
  const data = useRouteLoaderData()

  if (!data) {
    return <div className="spinner">Loading...</div>
  }

  return <div>{/* render data */}</div>
}
```

Error Handling

Errors during data fetching or route loading can disrupt the user experience. To manage errors:

- Use try-catch blocks within loader functions.
- Implement error boundaries or fallback components.
- TanStack Router allows defining *error elements* for routes:

```
const route = createRoute({
  path: '/data',
  loader: async () => {
    try {
      const response = await fetchData()
      return response
    } catch (error) {
      throw new Error('Failed to fetch data')
    }
  },
  errorElement: <ErrorComponent />,
})
```

- The `ErrorComponent` will render automatically when an error occurs.

Retry Logic

Automatic retries can enhance robustness when transient errors happen. Strategies include:

- Implementing retries within loader functions with exponential backoff.
- Using third-party libraries or custom hooks to retry fetch requests.
- Example:

```
function fetchDataWithRetry(retries = 3) {
  return async () => {
    for (let i = 0; i < retries; i++) {
      try {
        const response = await fetch('/api/data')
        if (!response.ok) throw new Error('Network response was not ok')
        return response.json()
      } catch (error) {
        if (i === retries - 1) throw error
        await new Promise(res => setTimeout(res, Math.pow(2, i) * 1000))
      }
    }
  }
}
```

Best Practices

- Always provide visual feedback during loading.
- Gracefully handle errors with user-friendly messages.
- Allow users to retry failed fetches.
- Avoid blocking navigation without indicating progress.
- Optimize for quick load times to reduce the occurrence of load states.

By thoughtfully managing load states, you ensure that users have a smooth and trustworthy experience, even when network conditions are unpredictable.

Route Guards & Authentication

Route guards are an essential feature in web applications to control access to certain routes based on authentication status, permissions, or other conditions. They help ensure that users can only access content they are authorized to see, enhancing the security and user experience of your application.

Protected Routes

Protected routes are routes that require users to be authenticated before access is granted. Implementing protected routes typically involves checking the user's authentication state within route guards or middleware.

```
import { createRoute } from '@tanstack/react-router'

// Example: Guarded route that only authenticated users can access
const ProtectedRoute = createRoute({
  path: '/dashboard',
  loader: async ({ context }) => {
    if (!context.user.isAuthenticated) {
      throw new Error('Unauthorized')
    }
    // Fetch additional data if needed
  },
  // Optional: redirect unauthenticated users
  conditionalRender: ({ context }) => {
    return context.user.isAuthenticated ? <DashboardPage /> : <Redirect to="/login" />
  }
})
```

Redirects

Redirects are useful for guiding users to appropriate pages based on their authentication status or other conditions. For example, redirecting unauthenticated users to a login page or redirecting authenticated users away from login pages.

```
import { createRoute } from '@tanstack/react-router'

const authGuard = createRoute({
  path: '/protected',
  loader: ({ context }) => {
    if (!context.user.isAuthenticated) {
      return { redirect: '/login' }
    }
  }
})
```

Conditional Rendering

Conditional rendering within routes allows dynamically displaying components based on authentication or other guard conditions. This provides flexibility in UI/UX, ensuring only authorized content is visible.

```
import { createRoute } from '@tanstack/react-router'
```

```
const ProfileRoute = createRoute({
  path: '/profile',
  conditionalRender: ({ context }) =>
    context.user.isAuthenticated ? <ProfilePage /> : <Redirect to="/login" />
})
```

Implementing Authentication Logic

In practice, route guards rely on an authentication context or state management. Often, middleware or loader functions check user credentials, tokens, or permission levels.

```
// Example: Using context to check auth
const checkAuth = (context) => {
  if (!context.user || !context.user.isAuthenticated) {
    return { redirect: '/login' }
  }
  return null
}
```

Best Practices

- Always verify authentication status both on the client-side (for user experience) and server-side (for security).
- Use clear redirect paths and messages for unauthorized access.
- Combine route guards with role-based access control for more granular permissions.
- Keep authentication logic centralized to simplify maintenance and updates.

Summary

Route guards and authentication mechanisms in TanStack Router enable you to secure routes effectively through:

- Protected routes that require authentication
- Redirect logic for unauthorized access
- Conditional rendering based on user state

Proper implementation of these features ensures a secure, seamless navigation experience for your users while maintaining robust access control.

Nested and Layout Routes

Nested and layout routes are powerful features in TanStack Router that enable building complex and reusable page structures. They allow you to compose multiple routes within each other, providing consistent layouts and shared UI components across different parts of your application.

Creating Layout Routes

A layout route serves as a wrapper for nested routes, often containing common elements such as headers, footers, navigation menus, or sidebars. To create a layout route, define a route with child routes, and include a placeholder for nested content using `<Outlet />` in your component.

```
// ExampleLayout.jsx
import { Outlet } from '@tanstack/react-router';

function ExampleLayout() {
  return (
    <div>
      <header>My Application Header</header>
      <main>
        <Outlet />  {/* Nested routes will render here */}
      </main>
      <footer>Footer content</footer>
    </div>
  );
}
```

```
// Routes configuration
const routes = createRouteConfig({
  path: '/',
  component: ExampleLayout,
  children: [
    {
      path: 'dashboard',
      component: DashboardPage,
    },
    {
      path: 'settings',
      component: SettingsPage,
    },
  ],
});
```

Nested Routing Patterns

Nested routing allows you to define routes inside other routes, enabling shared layouts and consistent navigation flows.

- **Deeply nested routes:** Build multi-level route hierarchies for complex applications.
- **Shared layouts:** Use layout routes to manage common UI elements across multiple nested routes.
- **Parameter sharing:** Pass route parameters efficiently through nested routes.

Example Structure:

```
/app
  /dashboard
    /stats
    /reports
  /settings
    /profile
    /preferences
```

Example Route Configuration:

```
const routes = createRouteConfig({
  path: 'app',
  component: AppLayout,
  children: [
    {
      path: 'dashboard',
      component: DashboardLayout,
      children: [
        { path: 'stats', component: StatsPage },
        { path: 'reports', component: ReportsPage },
      ],
    },
    {
      path: 'settings',
      component: SettingsLayout,
      children: [
        { path: 'profile', component: ProfilePage },
        { path: 'preferences', component: PreferencesPage },
      ],
    },
  ],
});
```

Shared Layouts

Reusable layout components help maintain a consistent look and feel across your app. You can embed shared UI elements such as navigation bars, side menus, and footers within layout routes, ensuring DRY (Don't Repeat Yourself) principles.

Best Practices

- Use layout routes to encapsulate UI elements common across multiple pages.
- Keep layout components simple and focused on structure.
- Use `<Outlet />` to render nested routes dynamically.
- Organize deeply nested routes thoughtfully to avoid overly complex hierarchies.

Summary

Nested and layout routes empower you to build modular, scalable, and maintainable applications with TanStack Router. By leveraging layout routes and nested patterns, you can create intuitive and reusable UI structures that enhance user experience and streamline development.

Transitions & Animations

Enhancing user experience through smooth transitions and engaging animations is a key aspect of modern web development. TanStack Router provides several features and best practices for implementing page transitions and animations to create seamless navigation flows.

Page Transitions

Page transitions help to visually indicate navigation changes, making the user interface feel more dynamic and responsive. You can implement transitions by leveraging CSS animations, JavaScript-based animation libraries, or built-in transition hooks provided by the routing library.

Using CSS Animations

Apply CSS classes during route changes to animate elements. For example:

```
/* Fade-in and fade-out effects */
.page-enter {
  opacity: 0;
  transform: translateY(-10px);
  transition: opacity 300ms, transform 300ms;
}

.page-enter-active {
  opacity: 1;
  transform: translateY(0);
}

.page-exit {
  opacity: 1;
  transform: translateY(0);
  transition: opacity 300ms, transform 300ms;
}

.page-exit-active {
  opacity: 0;
  transform: translateY(10px);
}
```

And toggle these classes based on route lifecycle hooks or route change states.

Using JavaScript Animations

For more complex animations, integrate with libraries like Framer Motion or GSAP. You can trigger animations in route lifecycle hooks or in response to navigation events.

```
import { motion } from 'framer-motion';

function PageComponent() {
  return (
    <motion.div initial={{ opacity: 0 }} animate={{ opacity: 1 }} exit={{ opacity: 0 }
      {/* page content */}
    </motion.div>
  );
}
```

Using CSS Animations

CSS animations are performant and straightforward for simple effects. Use keyframes for more intricate sequences.

Using CSS Animations

```
@keyframes fadeIn {
  from { opacity: 0; }
  to { opacity: 1; }
}
.element {
  animation: fadeIn 0.5s forwards;
}
```

Transition Hooks in TanStack Router

TanStack Router supports transition hooks such as `onEnter`, `onStay`, and `onLeave`, allowing you to trigger animations programmatically during navigation events.

```
import { createRouter, Route } from '@tanstack/router';

const router = createRouter({
  routeTree: [
    new Route({
      path: '/home',
      onEnter: () => {
        // start fade-in animation
      },
      onLeave: () => {
        // start fade-out animation
      },
    }),
  ],
});
```

Best Practices

- Use lightweight CSS transitions for performance.
- Coordinate animations with route lifecycle hooks for synchronized effects.
- Maintain accessibility by ensuring animations are non-intrusive and can be reduced if needed.
- Test animations on different devices and screen sizes to ensure smoothness.

Additional Resources

- CSS-Tricks: Animation and Transitions
- Framer Motion Documentation
- GSAP Documentation

Implementing well-crafted transitions and animations enhances the overall user experience, making your web application feel more polished and professional. Experiment with different effects and techniques to find what best suits your project's aesthetic.

Server-Side Rendering (SSR)

Server-Side Rendering with **TanStack Router** allows your application to render pages on the server before sending them to the client. This approach enhances performance, improves SEO, and provides a better user experience, especially for initial page loads.

SSR Setup Instructions

To enable SSR in your TanStack Router application, follow these general steps:

1. **Configure your server environment:**
   Set up a Node.js server or a serverless environment capable of rendering your app.

2. **Pre-render routes:**
   Use the router's API to prefetch data and generate HTML for each route during build time or on-demand.

3. **Hydration:**
   Once the server sends the rendered HTML, hydrate the React/Vue/Svelte app on the client to enable full interactivity.

Hydration Considerations

- Ensure the server-rendered content matches exactly with the client-side output.
- Use `useEffect` or equivalent hooks to manage client-specific code that shouldn't run on the server.
- Be cautious with API calls on the server; prefetch data during the rendering process to avoid flickering or inconsistencies.

Data Prefetching

Prefetch data during server rendering to improve performance and reduce load times for pages:

- Implement route loaders to fetch necessary data during SSR.
- Pass fetched data as initial state to the client to avoid duplicate requests.

Best Practices for SSR

- Keep the server rendering logic separate from client logic to prevent hydration mismatches.
- Use cache strategies to minimize server load and improve response times.
- Optimize your build process to support static export or incremental regeneration where applicable.

Example: Simplified SSR Flow

```
// Pseudocode for SSR setup
import { renderToString } from 'react-dom/server';
import { createMemoryHistory, RouterProvider } from '@tanstack/router';

const html = renderToString(
  <RouterProvider router={yourRouter} />
);

// Send `html` as response from your server
```

Conclusion

SSR with **TanStack Router** is flexible and powerful, supporting various backends and rendering strategies. Proper setup ensures faster load times, better SEO, and a smooth user experience. For detailed implementation tailored to your framework, consult the official documentation for React, Vue, Svelte, or your preferred environment.

Static Site Generation

Static Site Generation (SSG) is a powerful feature of TanStack Router that enables you to pre-render your web pages at build time. This approach results in faster load times, improved SEO, and better performance for your web application.

Pre-rendering Routes

With SSG, you can specify which routes should be statically generated. This is especially useful for sites with mostly static content, such as blogs, documentation, or marketing pages.

How it Works

During the build process, TanStack Router will generate static HTML files for the routes you configure for pre-rendering. These files are then served directly by your hosting environment, minimizing server load and eliminating the need for server-side rendering on each request.

Exporting Static Assets

To enable static site generation, you need to configure your build setup accordingly. Typically, this involves:

- Defining which routes to pre-render.
- Using the appropriate build commands or plugins that support static generation.
- Ensuring that your project outputs static assets like HTML, CSS, and JS files compatible with static hosting.

Optimization for Static Websites

To maximize the benefits of SSG, consider:

- Minimizing dynamic data requirements on pre-rendered pages.
- Leveraging incremental static regeneration or on-demand revalidation if your framework supports it.
- Using fallback strategies for routes that require server-side data or are generated dynamically.

Example Configuration

Here's a simplified example of how you might configure your router for static site generation:

```
import { createRouter, defineRoute } from '@tanstack/router'

const routes = [
  defineRoute({
    path: "/",
    prefetch: true,
  }),
  defineRoute({
    path: "/about",
    prefetch: true,
  }),
]
```

```
const router = createRouter({ routes })

export default router
```

Consult your framework's documentation for specific ways to integrate TanStack Router with static site generation workflows.

Benefits of Static Site Generation

- **Speed:** Serve pre-rendered pages instantly.
- **SEO:** Better search engine indexing due to static HTML.
- **Reliability:** Less dependency on backend servers.
- **Cost:** Reduced hosting costs by serving static files.

Considerations

While SSG offers many advantages, it may not be suitable for sites with highly dynamic content that needs real-time updates. For such cases, hybrid approaches combining static generation with client-side hydration or server-side rendering can be employed.

Explore the capabilities of TanStack Router's SSG features to optimize your specific project needs and deliver fast, reliable, and SEO-friendly websites.

Using Devtools

TanStack Router Devtools are an essential tool for debugging and inspecting your application's routing state and navigation flow. They provide real-time insights into your route tree, active routes, loaded data, and transition states, making it easier to troubleshoot and optimize your routing setup.

Installing Devtools

To begin using the Devtools:

- Install the plugin via npm or yarn:

```
npm install @tanstack/router-devtools
# or
yarn add @tanstack/router-devtools
```

- Import and initialize the Devtools in your application entry point:

```
import { RouterProvider } from '@tanstack/react-router'
import { Devtools } from '@tanstack/router-devtools'

function App() {
  return (
    <>
      <RouterProvider router={router} />
      <Devtools initialIsOpen={false} />
    </>
  )
}
```

Inspecting Routes and State

Once integrated, open your browser's developer tools and locate the TanStack Router Devtools panel. Here, you'll find:

- **Route Tree:** Visual representation of the current nested routes and their hierarchy.
- **Active Routes:** Highlighted routes that are currently active.
- **Route Data:** Loaded data for each route, with options to inspect request details.
- **Navigation Events:** Details on current and past navigation actions.
- **Transition States:** Visual cues indicating ongoing route transitions.

Debugging Navigation Flow

The Devtools provide valuable information about:

- The current URL and route parameters.
- Which routes are being matched and why.
- Data fetching status, including loading, success, or error states.
- Transition animations and effects.
- Errors or warnings in route configurations.

Use these features to:

- Verify route setup and parameter passing.

- Diagnose routing issues or unexpected behaviors.
- Optimize data loading strategies.
- Ensure smooth navigation experiences.

Best Practices

- Keep the Devtools panel open during development for real-time feedback.
- Use the inspection tools to simulate different route parameters or states.
- Combine Devtools insights with console logs for comprehensive debugging.

Additional Resources

For more detailed documentation and advanced usage tips, refer to the official TanStack Router Devtools documentation:

- Official Docs
- API References

By leveraging Devtools effectively, you can significantly enhance your debugging workflow and build more reliable, performant routing in your web applications.

Performance Tips

Optimizing the performance of your TanStack Router implementation is essential for delivering fast, smooth user experiences. This section covers practical strategies and best practices to enhance your application's routing efficiency.

Code Splitting

- **Lazy Loading Routes:** Implement route-based code splitting by dynamically importing route components. This reduces the initial bundle size, leading to faster load times.
- **Split at Logical Boundaries:** Divide your app into chunks aligned with feature sets or pages to optimize loading and caching.

Prefetching Strategies

- **Preload Future Routes:** Use prefetching to load route assets and data before the user navigates, reducing perceived latency.
- **Targeted Prefetching:** Prioritize prefetching for links likely to be followed, such as nearby or frequently accessed routes.
- **Idle Time Prefetching:** Leverage browser idle periods to prefetch resources without impacting critical rendering paths.

Optimizing Load Times

- **Prioritize Critical Resources:** Ensure essential route components and data are loaded first.
- **Use Cache Effectively:** Utilize browser cache, service workers, or state management to avoid redundant network requests.
- **Minimize Renders:** Reduce unnecessary re-renders by memoizing components and leveraging React's `useMemo` or `useCallback`.

Additional Best Practices

- **Limit the Number of Routes:** Keep the routing tree as simple and flat as possible to minimize matching overhead.
- **Avoid Heavy Computations in Loaders:** Perform lightweight operations in loaders to prevent blocking navigation.
- **Monitor Performance:** Use browser DevTools, Lighthouse, or analytics to identify bottlenecks and optimize accordingly.

By applying these performance tips thoughtfully, you can ensure that your application remains responsive and efficient, providing a seamless experience for end users.

Best Practices

Implementing effective routing strategies is essential for building maintainable, performant, and user-friendly web applications with TanStack Router. This page outlines some of the best practices to help you maximize the potential of your routing setup.

Routing Architecture

- **Plan your route hierarchy carefully:** Use nested routes and layout components to create a clear, logical structure that reflects your application's UI and data flow.
- **Keep your route definitions concise**: Avoid overly complex route configurations. Break down large routes into smaller, reusable components.

State Management and Data Loading

- **Leverage loaders efficiently**: Fetch data in loaders to keep components cleaner and to utilize caching strategies effectively.
- **Use React Query for server state**: Integrate with React Query to handle data caching, background updates, and retries seamlessly.
- **Prefetch data proactively**: Use preloading strategies to fetch data for routes that the user is likely to visit next, improving perceived performance.

Navigation and User Experience

- **Use declarative links**: Prefer `<Link>` components for navigation to ensure consistent behavior and accessibility.
- **Highlight active links**: Provide visual cues for the current route using `activeProps` for better navigation clarity.
- **Handle load and error states gracefully**: Show loading spinners, skeletons, or error messages to keep users informed.

Security and Access Control

- **Implement route guards**: Protect sensitive routes using guards that check authentication or permissions before rendering.
- **Use redirects for unauthorized access**: Redirect users to login pages or error pages as necessary to prevent unauthorized access.

Accessibility

- **Ensure accessible navigation**: Use semantic HTML and ARIA attributes where applicable.
- **Manage focus states**: Properly handle focus during route transitions for a better screen reader experience.

Performance Optimization

- **Implement code splitting:** Split your routes into chunks to reduce initial load time.
- **Optimize route transitions:** Use CSS animations or transitions for smooth visual navigation effects.
- **Minimize re-renders:** Memoize components and avoid unnecessary state updates during navigation.

Maintenance and Scalability

- **Document your routing logic:** Maintain clear documentation for complex routes and nested layouts.
- **Use meaningful route names and paths:** Follow consistent naming conventions for better developer experience.
- **Regularly review and refactor routes:** Keep your route structure aligned with evolving application requirements.

Community Resources

- Collaborate with the community through GitHub discussions, forums, and chat channels.
- Stay updated with the latest features and best practices by following official documentation and tutorials.

By adopting these best practices, you'll ensure that your application not only functions well but also provides a seamless and accessible user experience while being easier to develop and maintain.

Migrating from Other Routers

Switching to TanStack Router from another routing library involves understanding the core differences and adapting your existing routing setup. Whether you're migrating from React Router, Vue Router, or an older custom solution, this guide will help you transition smoothly.

React Router Migration Guide

Key Differences

- **Declarative Configuration:** TanStack Router emphasizes route configuration as objects, enabling more flexible and modular setups.
- **Route Tree Structure:** Routes are defined as a nested tree, allowing for more structured layouts.
- **Data Loading:** Built-in support for loaders and actions facilitates data fetching and mutations directly within route definitions.
- **Hooks and APIs:** Uses React hooks like `useMatches`, `useNavigate`, and `useParams`, which may differ from React Router hooks.

Migration Steps

1. **Replace Route Definitions:** Convert your `<Route>` components or route configuration objects to TanStack Router's route tree format.

2. **Update Navigation:** Replace `<Link>`, `<NavLink>`, and programmatic navigation with TanStack's `Link` component and `navigate` function.

3. **Refactor Data Fetching:** Migrate data fetching from `useEffect` or React Router's `loader` functions to TanStack Router's `loader` API within route objects.

4. **Adjust Route Guards:** Implement route protection using route conditions or wrapper components provided by TanStack Router.

5. **Test Thoroughly:** Ensure all routes, navigation, and data interactions work as intended after migration.

Example

React Router:

```
<Route path="/dashboard" element={<Dashboard />} />
```

TanStack Router:

```
const routeTree = createReactRouter({
  routes: [
    {
      path: "/dashboard",
      element: <Dashboard />,
    },
  ],
});
```

Vue Router Migration Guide

- Convert Vue Router route definitions into TanStack's route configuration format.
- Replace `<router-link>` and `<router-view>` with TanStack's `<Link>` and `<Outlet>`.
- Adjust navigation logic to use TanStack's `useNavigate` hook.
- Migrate route-specific data fetching and guards to TanStack's loader and condition APIs.

Legacy Apps and Other Routers

Migrating from legacy or less common routers involves:

- **Mapping route paths:** Translate route paths and nested structures to TanStack's route tree syntax.
- **Handling parameters and queries:** Adjust route parameters and query management within TanStack's API.
- **Replacing navigation patterns:** Update all navigation-related code to use TanStack's components and hooks.
- **Refactoring data loading:** Migrate any data fetching logic to TanStack's loader system.

General Migration Tips

- **Incremental Migration:** Start by replacing route definitions, then gradually adopt TanStack Router's features.
- **Leverage Compatibility Layers:** Use adapters or wrappers if available to ease transition.
- **Read the Documentation:** Familiarize yourself with the official migration guides and API references.
- **Community Support:** Engage with community forums or GitHub discussions for complex scenarios.

By carefully planning and implementing these steps, you can transition to TanStack Router smoothly, unlocking its powerful features for your application's routing needs.

Compatibility & Framework Support

TanStack Router is designed to be highly versatile and compatible across multiple frameworks, enabling developers to integrate modern routing solutions regardless of their chosen environment. Below is an overview of the supported frameworks and considerations for integration.

React

- **Official Support:** Fully supported with dedicated hooks, components, and utilities.
- **Features:** Seamless integration with React Suspense, Context API, and hooks for data loading, navigation, and route management.
- **Usage:** Ideal for SPAs, SSR, and static site generation within React ecosystems.

Vue.js

- **Community Support:** Available through community-maintained packages and adapters.
- **Features:** Compatibility with Vue's reactivity system, Vue Router patterns, and composition API.
- **Usage:** Suitable for Vue SPA and SSR applications.

Svelte

- **Community Support:** Integration via community adapters.
- **Features:** Utilizes Svelte's reactive features to provide smooth routing, transitions, and nested route management.
- **Usage:** Ideal for Svelte apps needing advanced routing capabilities.

Other Frameworks

- **SvelteKit, Astro, and more:** Developers can create custom adapters for other frameworks following the abstraction principles provided by TanStack Router.
- **Pure JavaScript:** Can be integrated into vanilla JS projects, with manual DOM management for route changes.

Adapter Architecture

TanStack Router uses an adapter pattern to support various frameworks. This approach encapsulates framework-specific logic, providing a consistent API for route definitions, navigation, and data loading across environments.

Considerations for Framework Integration

- **Hydration:** When implementing SSR, ensure proper hydration strategies are in place to prevent mismatch issues.
- **Reactivity:** Leverage framework-specific reactive features to synchronize route state with UI.
- **Performance:** Utilize code-splitting, prefetching, and caching strategies compatible with each framework for optimized performance.

Future Support and Community Contributions

While official support is currently focused on React, ongoing development and community contributions aim to expand support to additional frameworks continually. Developers are encouraged to participate in creating adapters or extending support for their preferred environments.

Summary

Framework	Support Status	Notes
React	Fully supported	Official package, hooks, and components
Vue.js	Community-supported	Compatibility with Vue Router patterns
Svelte	Community-supported	Integration via community adapters
Others (SvelteKit, Astro, vanilla JS)	Possible with custom adapters	Flexibility through adapter pattern

For detailed instructions on integrating TanStack Router into your specific framework, refer to the respective guides and community resources.

Official Documentation

Welcome to the official documentation for **TanStack Router**. This comprehensive guide provides you with everything you need to get started, understand the core concepts, explore advanced features, and utilize the full potential of the router in your projects.

Links to Documentation

- Getting Started
- API Reference
- Guides and Tutorials

Core Concepts and Features

Routing System

- **Route Definitions**: Learn how to define static, dynamic, and nested routes.
- **Navigation**: Programmatic and declarative navigation techniques.
- **Route Matching**: Strategies for matching URLs, handling params, and query parameters.

Data Management

- **Loaders**: Fetch data on route load with built-in caching.
- **Actions**: Handle form submissions and mutations.
- **Handling Load States**: Manage loading indicators, errors, and retries.

Advanced Features

- **Route Guards & Auth**: Implement protected routes, redirects, and conditional rendering.
- **Layouts and Nesting**: Create shared layouts with nested routes.
- **Transitions & Animations**: Add page transitions with CSS or JS animations.

SSR & Static Sites

- **Server-Side Rendering**: Setup instructions, hydration, and data prefetching.
- **Static Site Generation**: Pre-rendering, exporting assets, and optimizations.

Development & Debugging

- **Devtools**: Inspect routes, state, and navigation flow.
- **Performance**: Utilize code-splitting, prefetching, and load-time optimizations.
- **Best Practices**: Architect routing with accessibility, scalability, and maintainability.

Migration & Compatibility

- Guides for migrating from React Router, Vue Router, and legacy apps.
- Compatibility support for React, Vue.js, Svelte, and other frameworks.

Using the Documentation

This documentation is designed to be your one-stop resource for all things related to **TanStack Router**. It includes:

- Step-by-step setup guides
- API references with detailed descriptions
- Real-world examples and best practices
- Troubleshooting tips
- Community resources and support channels

Community and Support

Join the conversation or get help through:

- GitHub Discussions
- Stack Overflow
- Discord/Slack Channels

Contributing and Feedback

Your feedback helps improve **TanStack Router**. To contribute:

- Review the Contributing Guidelines
- Report issues on GitHub Issues
- Request features via Discussions or GitHub

Frequently Asked Questions (FAQs)

Visit the FAQ section for answers to common questions, troubleshooting tips, and recommendations.

Final Notes

Keep the official documentation bookmarked for quick reference, and stay tuned for updates, new features, and tutorials. Happy routing!

Community & Support

GitHub Discussions

Engage with the community through GitHub Discussions. Here, you can ask questions, share ideas, and collaborate on improving TanStack Router. It's a great place to find official answers and participate in ongoing conversations with both maintainers and fellow users.

Stack Overflow

For quick troubleshooting and specific implementation questions, Stack Overflow is an excellent resource. Use the tag `tanstack-router` to find related questions or to ask your own. Clear, well-structured questions help you get the most effective answers from the community.

Discord and Slack Channels

Join our vibrant community on Discord or Slack. These channels offer real-time support, discussions, and live collaboration opportunities. They're ideal for interactive help, learning tips & tricks, and connecting with developers using TanStack Router across different frameworks.

Community Contribution

We welcome contributions from the community! Whether you want to submit a bug fix, improve documentation, or add new features, your input helps make TanStack Router better for everyone. Check our contributing guidelines on the official repository for details on how to get involved.

Reporting Issues

Encounter a bug or have a feature request? Use the GitHub Issues page to report problems or suggest enhancements. Provide detailed descriptions, reproduction steps, and relevant code snippets to help us understand and address your concerns efficiently.

Stay Updated

Follow our official channels for the latest updates, announcements, and resources. Subscribing to newsletters or following social media accounts will ensure you stay in the loop with all things TanStack Router.

Get Support

If you need personalized support or have questions not covered in the documentation or community channels, consider reaching out via our support email or through official contact forms available on the website. We're here to help you succeed!

Contributing & Feedback

Thank you for your interest in contributing to the TanStack Router project! Your feedback, bug reports, suggestions, and code contributions are invaluable to help improve the library and the overall community.

How to Contribute

- **Reporting Issues:** If you encounter bugs or unexpected behaviors, please open an issue on the GitHub repository. Please provide detailed steps to reproduce the problem, your environment details, and any relevant code snippets.
- **Feature Requests:** Have an idea for a new feature or improvement? Submit a feature request through the GitHub issues page, describing your proposal and its benefits.
- **Submitting Pull Requests:** We welcome pull requests for bug fixes, new features, or improvements. Please ensure your code adheres to the project's style and passes all tests. Read the contribution guidelines in the repository for detailed instructions.

Guidelines

- **Code Quality:** Write clear, maintainable, and well-documented code.
- **Test Your Changes:** Ensure your modifications are tested thoroughly.
- **Community Respect:** Be respectful and constructive in your comments and interactions.
- **Stay Updated:** Keep your fork up-to-date with the main repository to facilitate smooth integrations.

Support and Communication

- Join discussions on our GitHub Discussions or Slack/Discord channels to collaborate with other developers and ask questions.
- Follow updates on the Official Documentation and community channels.

Feedback

Your feedback helps shape the future of TanStack Router. Please share your thoughts, report issues, or suggest improvements via the designated channels. We appreciate your contributions and look forward to building a robust, flexible routing library together!