Ziro: A Modular SSR Framework

Nariman Movaffaghi
Supervisor: Soroush Vedaei

December 6, 2024

Contents

1 Abstract

Web development has evolved to require frameworks that are both flexible and efficient. Existing server-side rendering (SSR) frameworks, such as Next.js and Nuxt.js, offer powerful solutions but can be less suited for building modular systems with pre-integrated features like authentication and administrative tools. This project presents Ziro, an SSR framework designed to explore the feasibility of creating a plugin-driven architecture within the JavaScript ecosystem.

Ziro provides a practical foundation for building dynamic web applications by emphasizing modularity and extensibility. Its architecture enables developers to integrate pre-built plugins, reducing repetitive tasks and focusing on core functionality. At its core, the `zirorouter` provides a cross-environment compatible router, the `zirogenerator` automates TypeScript type generation to offer robust type safety and `ziroreact` brings React.js compability to the router. The framework also supports hybrid rendering modes, combining server-side and client-side rendering to enhance performance and user experience.

Developed iteratively with rigorous testing using Vitest, Ziro incorporates features refined through real-world scenarios and edge case handling. By blending flexibility with a plugin-driven architecture, Ziro addresses common development pain points and establishes a foundation for future advancements in modular web frameworks. This proposal details the development process, key technical components, and the insights gained from building Ziro.

2 Introduction

Web development frameworks have evolved significantly in recent years, yet many still lack a cohesive approach to combining scalability, modularity, and developer-friendly abstractions. Frameworks like Next.js and Remix provide powerful tools for building web applications, but they do not expose certain critical layers, such as the router, for plugin-based customization. This limitation often forces developers to repeatedly create essential features like authentication, which involves setting up routes, middleware, actions, and components manually for every project.

Ziro addresses this gap by introducing a thoughtful approach that abstracts multiple parts of a full-stack framework while allowing them to scale independently. Ziro enables plugins to dynamically add custom pages, endpoints, middleware, and actions. This approach significantly reduces repetitive work and accelerates the development process, making it ideal for tasks like implementing reusable authentication systems or integrating predefined pages across projects.

Additionally, Ziro sticks to web standards to ensure compatibility across various runtimes, including Node.js, browsers and others like Bun[1] or Deno[2]. The framework is structured into separate packages to maintain modularity and framework-agnostic design:

- `ziro/router`: Contains the core routing concepts, designed to work independently of any specific framework.
- `ziro/generator`: A tool for generating TypeScript type-safe definitions to ensure robust developer workflows.
- `ziro/react`: A React-compatible implementation for seamless integration with React-based projects.

Ziro's features are based on modern web development principles. It offers a fully typesafe routing system, ensuring that all actions and data derived from middleware or layouts maintain type safety throughout the application. With its sequential data flow, loaders from nested pages run in order, allowing data fetched at higher levels to cascade to lower ones. The plugin system further enhances flexibility by simplifying the process of extending applications with custom functionalities.

[1] https://bun.sh
[2] https://deno.com

Targeted primarily at developers, Ziro also opens the door to building tools for broader audiences, such as WordPress users who may not have a strong technical background. By introducing modularity and preconfigured components, it streamlines the creation of dynamic, scalable applications like e-commerce platforms, personal blogs, and full-stack applications.

Developing Ziro has presented several challenges, including implementing robust type safety - a feature still relatively novel in modern frameworks - and ensuring seamless integration across different rendering modes: SSR, CSR, and a partially SSR mode that streams data to users. The result is a proof of concept that not only demonstrates the feasibility of combining full-stack JavaScript apps with plugin-based scaling but also provides a modular foundation for scalable, dynamic application development.

3 Literature Review

3.1 Evolution of Web Technologies

In the early days of framework-based web development, frameworks and technologies like `Symphony`[3], `Laravel`[4] and `Ruby on Rails`[5] were popular (and still popular), allowing developers to deliver full HTML pages generated on the server. These server-rendered applications have clear benefits. Users receive complete, pre-generated content, making initial page loads quick and ensuring that content was easy for search engines to index. This method effectively addressed early concerns around user experience and web visibility.

However, things changed with the introduction of Single Page Applications (SPAs). SPAs revolutionized development by providing smoother client-side experiences. They shifted the server's role to mainly delivering static assets while JavaScript managed the content rendering on the client side. This model was attractive due to its improved maintainability and streamlined development. Frameworks like Angular[6] and later React[7] and Vuejs[8] became popular for their client-side rendering (CSR) that delivered a richer and more dynamic user experience.

While SPAs brought many advantages, they also had their challenges. One major weakness of SPAs was their initial page load performance. Since the server only served a minimal HTML content and relied on JavaScript to render content, the time taken for the browser to download and execute the JavaScript bundle impacted the initial load speed. This delay affected user experience and search engine optimization (SEO), as search engines struggled to index content that was rendered dynamically in the browser. To address these issues, SSR frameworks emerged as a practical solution.

3.2 The Rise of Server-Side Rendering (SSR) Frameworks

SSR frameworks have changed the landscape of web development, offering benefits that meet the increasing demand for better user experiences and search

[3] https://symfony.com/
[4] https://laravel.com/
[5] https://rubyonrails.org/
[6] https://angularjs.org/
[7] https://react.dev/
[8] https://vuejs.org

engine optimization (SEO). These frameworks pre-render content on the server, allowing for faster initial page loads, improved discoverability by search engines, and a smoother user experience. With SSR, the server processes the requests and delivers fully rendered HTML pages to the client, reducing the browser's responsibility of generating content. This approach has proven valuable as applications have become more interactive and data-heavy.

A significant advantage of SSR frameworks is their positive impact on `Core Web Vitals`[9], a set of essential metrics used to measure user experience. These include:

- **Largest Contentful Paint (LCP)**: Measures loading performance; SSR improves LCP by ensuring that the initial content loads quickly.
- **First Input Delay (FID)**: Evaluates interactivity; SSR can reduce FID by delivering fully rendered pages that are interactive sooner.
- **Cumulative Layout Shift (CLS)**: Assesses visual stability; SSR reduces layout shifts by ensuring content is structured and rendered on the server before the client receives it.

By improving initial page load times, SSR frameworks contribute to better performance in Core Web Vitals, which directly impacts user satisfaction and search engine ranking.

Another benefit of SSR is the ability to generate dynamic Open Graph[10] (OG) tags before the HTML is sent to the user. OG tags are crucial for sharing rich and structured content on social media platforms, enhancing how a web page appears when shared. These tags must exist in the initial response for social media crawlers to read them and display previews accurately. This ensures that shared content looks appealing and informative.

Overall, the evolution from traditional server-rendered pages to SPAs and back to SSR frameworks has shaped modern web development. While SPAs have their benefits, particularly in terms of client-side interactivity and reduced server load, SSR frameworks offer practical solutions for better SEO, improved performance, and enhanced user experience. By implementing SSR, developers can leverage server-side processing to deliver fully rendered pages with optimal loading times, making web applications faster and more accessible.

[9] https://web.dev/articles/vitals#core-web-vitals
[10] https://ogp.me/

3.3 Review of Key SSR Frameworks

Next.js[11] has become one of the most widely adopted SSR frameworks within the JavaScript ecosystem. Built around React, it offers a strong set of features, including file-based routing, and advanced built-in SSR capabilities. It provides developers flexibility through hybrid rendering options that combine both SSR and Static Site Generation (SSG). Next.js supports two different routing systems: the Page Router[12] and the App Router[13]. The Page Router has limited TypeScript support, which can lead to challenges in maintaining type safety. In contrast, the App Router offers much better TypeScript integration, making it easier for developers to write type-safe code. However, while Next.js shines in out-of-the-box features, it lacks a comprehensive plugin system for extensive customization. This can make large-scale projects more complex as developers need to extend all middleware rules in a single file[14] and create routes manually, resulting in code repetition and in some cases inefficiency.

Remix[15] which was made on top of **React Router**[16] by developers behind it, provides a unique approach to handling routing and data fetching. It focuses heavily on nested routing and uses some functions called loaders for data management. This model improves data handling within complex page structures. Remix by introducing delightful APIs for the data fetching flow has improved its developer experience. However, lack of middlewares and having multiple actions on a single route hurts the developer experience. Additionally, Remix doesn't have an easy solution for type-safety, but it works with better developer experience than Next.js in terms of routing and data fetching.

Nuxt.js[17] is the go-to SSR framework for `Vue.js` applications and offers similar capabilities to Next.js, but tailored for the Vue ecosystem. It has built-in SSR, a modular structure, and supports customizability. In compared to Next.js and Remix, Nuxt offers more APIs to customize the logic behind the app. Of course modifying or extending core logic requires a deep understanding of the framework's internals but Nuxt has a more comprehensive plugin system than Next.js and Remix which makes it easier to extend the core functionalities. Additionally, Nuxt.js has great support for TypeScript, making it easier for developers to write type-safe code and improve maintainability. In case of developer experience, Nuxt shines!

[11] https://nextjs.org/
[12] https://nextjs.org/docs/pages
[13] https://nextjs.org/docs/app
[14] https://nextjs.org/docs/app/building-your-application/routing/middleware
[15] https://remix.run/
[16] https://reactrouter.com/
[17] https://nuxt.com

Astro[18] is a newer framework that has gained attention for its support of SSR and hybrid rendering, which blends SSR and Client-Side Rendering (CSR). Its minimal JavaScript footprint helps optimize performance by allowing developers to choose which frameworks to use for different parts of their project. Astro also has first-class support for TypeScript, making it easier for developers to write type-safe code. Although Astro's modular design provides significant customization, integrating various components can sometimes feel fragmented and may require developers to piece together solutions for a cohesive application. Additionally, Astro offers a revolutionary solution for handling server actions[19] and data validation, streamlining the process and enhancing developer productivity. One of the standout features of Astro is its framework-agnostic nature. Developers can write components using React, Vue.js, Svelte, Markdown, etc, allowing for greater flexibility and reusability of existing code. This capability makes Astro an attractive option for teams that work with multiple front-end technologies or are transitioning between frameworks. By supporting a variety of component models, Astro enables developers to leverage the strengths of different frameworks within a single project, optimizing performance and maintainability.

3.4 The Market Landscape of SSR Frameworks

The growth of SSR frameworks has been notable in recent years, driven by their potential to enhance user experience and SEO. In the context of market adoption, it's important to understand how frameworks have been received by the development community and the proportion they represent in web development projects.

Next.js stands out as the most widely used SSR framework, supported by its robust integration with React. According to the State of Frontend 2023[20] report, Next.js commands a significant share of the market, with over 56% of developers who work in web development using it for their SSR needs. The growth can be attributed to its ease of use, stability, and large community, which makes it a good choice for a wide range of projects.

Nuxt.js, continues to be a key player within the Vue.js ecosystem. While its adoption is smaller compared to Next.js, it remains a preferred choice for Vue developers. Its modular architecture and focus on simplifying SSR actions make it an efficient tool for building dynamic, server-rendered applications. According

[18] https://astro.build/
[19] https://docs.astro.build/en/guides/actions/
[20] https://2023.stateofjs.com/en-US/libraries/meta-frameworks/

to the State of Frontend 2023 survey, almost 23% of developers prefer using Nuxt.js, and this percentage grows each year based on the charts. Nuxt.js has a strong presence in the market, with a significant number of developers leveraging its capabilities for SSR projects.

Remix, despite being newer than Next.js and Nuxt.js, has shown rapid growth in popularity, especially among developers seeking greater control over routing and data fetching. The State of Frontend 2023 survey mentioned that while Remix's adoption is still below of Next.js and Nuxt.js, it has gained significant attention for its modern approach to data management.

Astro has emerged as a promising framework for projects prioritizing performance and reduced JavaScript load. According to the State of Frontend 2023 survey, Astro is used by more developers than Remix, highlighting its growing popularity. Its unique hybrid approach appeals to developers who want the flexibility of combining SSR and CSR efficiently. The use of Astro is increasing, particularly in projects where minimal JavaScript and high performance are key considerations.

3.5 Why Market Metrics Matter

Market size or community size alone is not always the most crucial factor when evaluating frameworks. Instead, their progress and the timing of their introduction reveal important insights. For instance, Next.js, launched in 2016, holds a substantial market share due to its established presence. Meanwhile, newer frameworks like Astro, introduced in 2021, have already captured nearly 19% of the market, reflecting their rapid adoption to developers. This suggests that Astro delivers features that resonate strongly with modern needs.

Rather than selecting a framework outright, we analyze usage metrics and growth trends to identify why these frameworks succeed. This evaluation helps us understand features that can be beneficial to Ziro. For example, Next.js demonstrates reliability with its long-term adoption, but newer frameworks like Astro showcase innovation and efficiency that might inspire Ziro's design.

3.6 Evolution of Hosting Solutions and Their Impact on SSR-based Projects

The growth of server-side rendering (SSR) frameworks has been accompanied by significant advancements in hosting technologies. In the past, web applica-

tions were often hosted on virtual private servers (VPS), which, while flexible, required significant manual configuration and maintenance. This setup brought challenges in terms of scalability and performance, making it less suitable for SSR, which demands efficient server processing.

Recently, a shift has occurred with the emergence of modern serverless and edge hosting solutions. Platforms such as Vercel[21], Netlify[22], Cloudflare Pages[23], Deno Deploy[24], and Fly.io[25] have transformed how SSR frameworks are deployed. These services offer affordable, scalable hosting that makes it easier for developers to manage applications without worrying about infrastructure complexity.

The idea of running servers "on the edge," closer to the user, has been a game-changer. Platforms like Cloudflare Pages, Deno Deploy, and Fly.io make it possible to execute server-side code from multiple locations worldwide. This approach reduces latency, speeds up load times, and addresses many of the traditional challenges of server-side rendering (SSR), like slow response times and scalability concerns. By being more accessible and cost-effective, edge computing has also made SSR frameworks a go-to solution for developers building fast, efficient web applications.

3.7 Ziro's Position: A Foundation for Modern Web Development

The modern landscape of web development demands frameworks that not only streamline the development process but also empower developers with flexible and powerful tools. While existing frameworks like Next.js, Remix, Nuxt.js, and Astro offer impressive capabilities, they often lack the foundational elements necessary for building comprehensive, feature-rich web applications similar to the advanced content management systems (CMS) seen in the PHP and Laravel ecosystems, such as WordPress[26] and Laravel Nova[27]. This is where Ziro steps in.

[21] https://vercel.com
[22] https://netlify.com
[23] https://pages.cloudflare.com/
[24] https://deno.com/deploy
[25] https://fly.io/deploy
[26] https://wordpress.org
[27] https://nova.laravel.com/

The Vision Behind Ziro

The primary motivation for developing Ziro was to create a framework that enables developers to build applications with the simplicity and extensibility similar to established CMS platforms but within the JavaScript ecosystem. For instance, Laravel Nova is a powerful admin panel for Laravel that integrates seamlessly into the framework, providing a ready-to-use, configurable interface for managing data and content. By installing a plugin or package in Laravel, developers can extend their applications with new functionalities, such as authentication pages or admin dashboards, that are automatically configured through a single configuration file.

Ziro aims to bring this level of seamless integration and extensibility to the JavaScript world, enabling developers to build complex, dynamic web applications without having to reinvent the wheel for common features. With Ziro, adding essential components like authentication pages, administrative panels, or custom workflows should be as simple as including a package and configuring it through a simple config file.

Ziro's Key Features and Advantages

1. **Modular Plugin Architecture**: Ziro's most distinguishing feature is its modular plugin architecture. Unlike existing frameworks that often require developers to build complex middleware or extend core logic from scratch, Ziro allows developers to add new functionalities with pre-built plugins that can be installed and activated with minimal effort. Each plugin in Ziro is designed to integrate seamlessly with the core framework, reducing manual coding and enabling rapid development.

   For example, when a developer wants to add an authentication system or a content management interface, they can install a Ziro plugin that handles all related routes, controllers (loaders/actions), and UI components. This system's modularity allows developers to customize these features through a single configuration file, hugely reduces the time spent on setup and maintenance.

2. **Type Safety and Predictability**: A major pain point with many current frameworks is the difficulty of maintaining type safety across various parts of the application. Ziro addresses this with a type-safe system that extends throughout its routing and data-handling mechanisms. By leveraging TypeScript, Ziro ensures that data passed between components is

consistently validated, reducing the risk of runtime errors and making code more predictable. This helps developers build complex applications confidently, knowing that type mismatches will be caught during development rather than after deployment.

3. **Comprehensive Configuration Management**: In frameworks like Laravel, configuration management is centralized, allowing developers to control application behavior from a single point. Ziro adopts a similar approach, providing an easy-to-use configuration file where developers can enable or disable plugins, set default values, and manage application settings. This makes it easier to adjust core features and tailor the framework's behavior to the specific needs of a project.

4. **Sequential Data Handling and Flow**: Ziro's data flow is designed to be both predictable and sequential, ensuring that data loading and state management propagate logically from parent to child routes. This sequential approach facilitates better coordination between different application parts and simplifies managing shared state, making complex data structures easier to handle.

5. **Extensibility for Advanced Features**: One of Ziro's goals is to support the development of advanced features with minimal friction. For instance, when building an application similar to WordPress or Laravel Nova, developers may need to create custom pages or modules that fit within the broader application structure. Ziro's built-in plugin system and flexible routing make this process straightforward, supporting extensions that behave like native parts of the framework and can extend the Typescript declarations.

The Need for a Comprehensive Foundation

While frameworks like Next.js, Nuxt.js, Astro or Remix have gained popularity for their robust capabilities, they are often focused on providing a solid base for general-purpose applications. Ziro's focus is different; it aims to provide a comprehensive, foundational platform that facilitates building developer-friendly tools and applications directly out of the box. With Ziro, developers can focus on building features that matter without having to spend an excessive amount of time setting up repetitive, common components.

Ziro in Context with the Existing Ecosystem

When compared with other frameworks, Ziro's position stands out because it bridges the gap between advanced, feature-rich platforms like Laravel and the modern JavaScript ecosystem. While tools like Astro offer efficient rendering and optimized performance, their modular approach can sometimes leave developers piecing together components manually. Ziro's approach is more comprehensive. It combines powerful, built-in features with the ability to enhance the framework with additional modules without complexity.

The modular and type-safe nature of Ziro brings new opportunities for web developers who seek the fluency of modern JavaScript frameworks while enjoying the satisfaction and extensibility of established CMS platforms. By enabling developers to build feature-rich applications quickly, Ziro opens a new door in the JavaScript ecosystem.

3.8 Conclusion

The landscape of web development has seen remarkable evolution, moving from server-rendered content to dynamic client-side applications and now to the hybrid models supported by modern frameworks. Frameworks like Next.js, Nuxt.js, Remix and Astro have cleared the way, each addressing different aspects of performance, SEO, and developer experience. However, as the demand for comprehensive, feature-rich, and scalable applications grows, so too does the need for a framework that combines the flexibility of modern JavaScript tools with the extensibility and ease-of-use found in mature CMS platforms like WordPress and Laravel Nova.

Ziro comes up as a compelling solution in this context. With its type-safe architecture, modular plugin system, and centralized configuration management, it promises to streamline the development of complex applications while promoting rapid deployment and ease of maintenance. Ziro's approach to sequential data handling, combined with its ability to integrate advanced features seamlessly, positions it uniquely within the ecosystem.

As development teams navigate a choice among existing frameworks, understanding their strengths and limitations is key. While established players like Next.js offer reliability and a standard path, newer frameworks such as Astro showcase the potential for innovative, minimalistic approaches. Ziro, however, sets itself apart by providing new paradigm to developers who need both foundational structure and modular extendibility. It fills the gaps left by existing

solutions, providing a practical, developer-centric platform that encourage rapid application development without sacrificing flexibility or depth.

In conclusion, Ziro wants to take the next step in the SSR frameworks era. An inclusive framework that blends the best of JavaScript with the tried-and-tested practices of established web development platforms. Its unique features support developers in building powerful, scalable applications that meet the demands of today's web environment, while remaining adaptable to future technological shifts. As the landscape continues to evolve, frameworks like Ziro will be at the forefront, inspiring a new era of development that is both practical and innovative.

4 Methodology

4.1 Building the Core of Ziro

The development of Ziro started with a focus on its most important part—the router. This main component, called `ziro/router`, forms the base of the framework and takes care of handling routes and interacting with browser events. The initial step of creating a solid and reliable router was crucial for setting the groundwork for the rest of the framework, ensuring a strong and consistent foundation for future development.

4.2 Creating `ziro/router`

The `ziro/router` package was designed to be as independent as possible from specific frameworks and was built entirely in JavaScript to ensure compatibility with different environments and libraries. Its main goal is to offer an API that can handle browser events smoothly, such as updating the route when a user clicks on a link or adjusting the router state when they use the back and forward buttons on the browser. This independence also ensured that the package could be reused and extended for different UI libraries like React, Vue, etc, making Ziro versatile and adaptable.

4.3 Using Test-Driven Development

The development of `ziro/router` began with a focus on rigorous testing, although it did not follow a formal test-driven development (TDD) approach. Without a fully operational environment to test the router's performance in action, Vitest[28] was used as a critical tool for writing and running tests. These tests helped verify that the code functioned correctly and covered edge cases, contributing to a more robust codebase and preventing potential issues. Vitest, known for its speed and ease of use in JavaScript and TypeScript projects, played an essential role in ensuring the reliability of the router's functionality and streamlining the verification process. The use of Vitest, a fast and user-friendly testing tool for JavaScript and TypeScript, contributed greatly to the reliability of the codebase and streamlined the process of verifying the router's functionality.

[28] https://vitest.dev

This rigorous testing approach was especially crucial because `ziro/router` was initially developed as a client-only router. Achieving comprehensive code coverage was a top priority, given the absence of a user interface (UI) for visual confirmation of features. Consequently, testing became the primary method for verifying the robustness and dependability of the router's functionality, ensuring it could handle real-world scenarios reliably. This part will be moved to the 'Theses' section to support the idea of Ziro's reliability and commitment to high-quality development.

4.4 Moving to `ziro/react`

After establishing a stable `ziro/router`, the next step was developing a version for React applications, called `ziro/react`. This transition allowed developers to integrate the router seamlessly within React projects, providing the benefits of Ziro's routing capabilities while maintaining the ease of use that React developers expect.

4.5 Workflow for Development

The project was structured as a monorepo, meaning that the `ziro/react` package could share dependencies and directly use the development version of `ziro/router`. This monorepo approach not only made it easier to manage dependencies but also facilitated synchronized development across packages. Any changes made to `ziro/router` could be tested and updated alongside `ziro/react`, ensuring consistency and reducing the risk of compatibility issues.

While building `ziro/react`, new needs arose that required adjustments to the `ziro/router` core. This back-and-forth process of refining both packages together was crucial for the evolution of Ziro. It ensured that enhancements and new APIs were added in a way that supported the overall goal of creating a cohesive and powerful framework.

The in-depth discussion about the benefits of using a monorepo will be moved to the 'Technical Specifications' section to provide a more detailed look at this technical setup.

4.6 Improving Type Safety with TypeScript

A detailed explanation of the `ziro/generator` package and how it worked with Vite is better suited for the 'Technical Specifications' section. This package was developed to create type-safe routes by automatically generating TypeScript types, which greatly improved code safety and developer confidence.

4.7 Adding Server-Side Compatibility

Once `ziro/router`, `ziro/react`, and `ziro/generator` were working smoothly on the client side, the next challenge was adding server-side compatibility. This involved integrating the client-side router with a server-side version to facilitate data sharing between the two. This feature was crucial for enabling Ziro to support both server-side rendering (SSR) and client-side rendering (CSR) within a unified framework. This integration allowed developers to build applications that could leverage the strengths of both rendering types, providing a seamless experience and better performance for end-users.

4.8 Improving and Refining the Framework

The process of developing `ziro/router`, followed by `ziro/react`, `ziro/generator`, and adding server-side support, was carried out in a step-by-step manner. Each phase played an important role in refining the framework and ensuring its core components remained strong, flexible, and aligned with the overall goals. The iterative development approach meant that as new challenges arose, the framework could be adapted to meet them, ensuring long-term robustness and scalability.

4.9 Results of the Development Process

This careful and step-by-step approach led to the development of Ziro's main packages with:

- Stability: The use of rigorous testing and TDD created a dependable codebase that can be trusted for reliable performance.
- Flexibility: The `ziro/router` package was built to be adaptable and can be used across different projects and environments, not just React.

- Scalability: The modular architecture allows for easy additions of new features and plugins, making it straightforward to expand Ziro's capabilities.
- Type Safety: The integration of TypeScript and automated type generation minimized errors and boosted developer confidence in building and maintaining applications.

5 Theses

Web development frameworks have evolved significantly, yet they often fall short when it comes to addressing the repetitive and time-consuming tasks that developers face. Ziro proposes a solution to this challenge by introducing a modular, plugin-driven framework tailored for the JavaScript ecosystem. Its aim is to simplify the development process for modern websites by offering ready-to-use, extensible components that streamline repetitive tasks, while retaining the flexibility and customization developers need.

5.1 The Problem: Repetitive and Time-Consuming Tasks

In the modern era of software-as-a-service (SaaS) hosting and deployment platforms, JavaScript frameworks have become the backbone of scalable and efficient web applications. However, developers often find themselves repeatedly implementing features like authentication systems, dashboards, and layouts—components that are essential but not central to the unique value of their projects.

For example, in an e-commerce application, the storefront design and functionality are critical, while elements like the dashboard or role-based authentication system are secondary but necessary. Building these repetitive components from scratch can consume valuable time and resources, diverting focus from the core features that define the application. Existing frameworks like Next.js and Remix address general development needs but lack the flexibility to add such components as pre-built, easily configurable plugins. This gap leads to inefficiencies and a steeper learning curve for developers who need to create these features repeatedly.

Moreover, this challenge extends to projects like personal websites or smaller-scale applications where the goal is to have basic functionality without spending excessive time on implementation. For instance, a developer creating a blog or portfolio website may need a dashboard for managing content but does not require a highly customized or unique design. Ziro seeks to address these gaps by enabling developers to focus on what matters most—the unique aspects of their projects—while reducing time spent on repetitive tasks.

5.2 Modularity and Plugin-Driven Design

Ziro introduces a new paradigm by prioritizing modularity and a plugin-based architecture. Developers can use plugins to integrate features like authentication pages, dashboards, or layouts directly into their applications with minimal configuration. These plugins not only save time but also allow customization, enabling developers to tailor the functionality to their specific needs. This approach mirrors the extendability of systems like WordPress, where plugins provide essential functionality while maintaining flexibility for user-defined customization.

The framework is designed to work standalone, meaning it does not rely heavily on external dependencies for its core functionality. Instead, Ziro offers developers a lightweight, adaptable system that can be extended as needed. For example, by simply installing a plugin, developers can add a fully functional authentication system or a dashboard layout to their applications without building these components from scratch. This reduces development time while ensuring that core functionality remains intact.

In addition, Ziro leverages TypeScript[29] to ensure type safety across the entire development process. TypeScript has become an essential tool in modern development, providing better data management, accurate IDE suggestions, and minimizing runtime errors. By making type safety a central feature, Ziro enhances developer productivity and reduces common pitfalls associated with dynamic JavaScript programming. This ensures that plugins and core functionalities integrate seamlessly without introducing unexpected behaviors or breaking existing code.

5.3 Addressing Today's Web Development Needs

Modern frameworks like Next.js and Remix are excellent for general-purpose applications but do not offer native solutions for repetitive tasks. For instance, creating a personal website often involves building a dashboard for managing content. While the dashboard's design might not be critical, implementing it from scratch is unnecessarily time-consuming. Ziro's plugin system allows developers to quickly add such components, freeing them to focus on unique features that make their applications stand out. This approach reduces development overhead and enables faster project delivery without compromising flexibility.

[29] https://www.typescriptlang.org/

Ziro's hybrid rendering model—combining server-side rendering (SSR) and client-side rendering (CSR)—further supports the needs of modern web applications. Hybrid rendering enhances performance by improving initial page load times, which is crucial for search engine optimization (SEO) and user experience. By adopting a partially on-demand streaming approach, Ziro ensures that both developers and end-users benefit from optimized performance without sacrificing interactivity or responsiveness.

Demonstrating Plugin Integration and Flexibility

The goal of Ziro is to achieve a level of extensibility comparable to platforms like WordPress, where plugins play a central role in extending functionality. Ziro's plugin architecture enables developers to:

- Add custom pages and routes to their applications.
- Extend existing features with minimal configuration.
- Build modular and reusable components that align with their specific requirements.

For example, a developer building an e-commerce platform could use a plugin to integrate a role-based authentication system and a dashboard for order management. The developer could then customize the dashboard by adding specific pages, such as sales analytics or customer feedback, without disrupting the overall application structure. This level of flexibility ensures that plugins provide a solid foundation while allowing developers to make adjustments that fit their needs.

By supporting a hybrid rendering model, Ziro further enhances the performance and SEO of applications, improving initial page load times and user experience. This combination of modularity, flexibility, and performance optimization provides developers with a framework that adapts to their needs while minimizing repetitive work.

5.4 Ziro's Contribution to Web Development

Ziro contributes to the web development ecosystem by filling a gap between general-purpose frameworks and fully extendable platforms. By focusing on

plugin-driven development, Ziro empowers developers to create scalable applications faster and with fewer resources. Its emphasis on type safety and hybrid rendering ensures a modern, efficient development process that aligns with the demands of today's web applications.

Additionally, Ziro's architecture promotes collaboration and innovation within the developer community. The ability to create, share, and reuse plugins fosters a collaborative ecosystem where developers can build upon each other's work. This approach not only accelerates development but also encourages the creation of high-quality, standardized components that benefit the entire community.

Ultimately, Ziro seeks to redefine how developers approach repetitive tasks, offering a framework that prioritizes ease of use, extensibility, and flexibility. This makes it an ideal choice for developers looking to streamline their workflows while maintaining the freedom to innovate and customize. Through its modular design and focus on modern web development needs, Ziro aims to establish itself as a valuable tool for building dynamic, scalable applications efficiently.

6 Technical Specifications

6.1 Understanding the Router and Route Structure

Ziro's router is a fundamental component that handles routing and the organization of routes and endpoints in an application. It plays a crucial role in navigating between different parts of an application, ensuring data flows seamlessly, and integrating various elements such as middleware, actions, and components. Understanding the router and route structure in Ziro is essential for developing and maintaining applications efficiently.

6.1.1 Overview of the Router's Role and Structure

The router in Ziro manages the definitions of all routes, making sure that each route is properly configured to handle requests and load data. The router tree is built on top of the **rou3**[30] library, a lightweight and fast router for JavaScript based on a radix tree structure. This library provides an efficient way to manage and traverse routes.

6.1.2 What is a Radix Tree?

A radix tree (or prefix tree) is a type of data structure that is particularly well-suited for handling route paths. Each route is stored as a sequence of characters in the tree, allowing for efficient prefix matching and lookup. This structure is ideal for route trees as it reduces the number of comparisons needed when finding a specific route and supports hierarchical relationships.

6.1.3 Main Properties of a Route

- **id**: The path of the route also acts as the identifier because it is unique.
- **parent**: The parent route.
- **loader**: A function or method responsible for loading data needed by the route.

[30] https://unjs.io/packages/radix3

- **middlewares**: Functions that run before the route loader and actions to perform tasks such as authentication, data validation, or request modification.
- **actions**: Functions that handle post requests, often used in handling form requests.
- **meta**: Metadata associated with the route, used for SEO.
- **props**: Properties that relates to the route. Often used for storing the rendering-related data, including the main `Component`, `ErrorBoundary`, and `Loading` components (in React concept).

6.1.4 Parent-Child Route Relationships

Routes in Ziro can have parent-child relationships that help structure the application in a nested way. For example, a layout route has some sub-routes that inherit the data from the parent, allowing for shared layouts, middleware and loading data. This nesting supports complex page structures and maintains consistent data flow across different parts of the application.

6.1.5 Example of a Basic Router and Route Definition

Consider the following example of how a router with different routes are defined in Ziro:

```
1  import { Route, Router } from 'ziro/router';
2
3  const router = new Router()
4
5  const rootRoute = new Route("_root", {
6    loader: async () => {
7      return fetch("https://api.example.com/data").then(res => res.json())
8    }
9  });
10
11 const indexRoute = new Route("/", {
12   parent: rootRoute,
13   loader: async () => {
14     return fetch("https://api.example.com/index").then(res => res.json())
15   }
16 });
17 router.addRoute(indexRoute)
18
19 export default router;
```

Figure 1: Example of a Basic Router and Route Definition

In this example, the router is created (line 3), and two routes (`rootRoute` and `indexRoute`) are defined. Only the `indexRoute` is added to the router as it is the main route visible to users. The routes ending with `_root` and `_layout` are considered non-viewable routes and serve as wrappers around other routes. When the router is called to find the route tree for a specific request, such as the "/" endpoint, it should return an array with the `rootRoute` as the first item and the `indexRoute` as the second item. This calculation occurs via the `addRoute` method, which builds the route tree recursively, traversing the parent property until it reaches undefined and adds the route array as the value of the radix tree for the given endpoint.

Minimal Example of How the Router Creates the Radix Tree

```
import * as rou3 from 'rou3'

class Router {
  radixTree;
  constructor(){
    this.radixTree = rou3.createRouter();
  }

  addRoute(route){
    let tree = [route]
    let routePath = route.id;

    while (true) {
      if (!tree[0].getParent()) break
      const parentRoute = tree[0].getParent()
      if (parentRoute) tree.unshift(parentRoute)
    }

    rou3.addRoute(this.radixTree, '', routePath, tree)
  }
}
```

Figure 2: Minimal Example of How the Router Creates the Radix Tree

This code snippet shows how the router class initializes a **rou3** radix tree and adds routes by traversing their parent routes recursively to build a comprehensive route tree structure.

6.1.6 How the Router Handles Requests

When the `router.handleRequest(request)` method is called, the router follows a systematic process to find and handle the appropriate routes. Here is how the process works:

1. **Request Matching**: The handleRequest() method uses the radix tree to match the incoming request path to a specific sequence of routes. This is done by calling `rou3.findRoute(this.radixTree, '', String(request.url.pathname))`
2. **Load Routes**: Once a route tree is matched, the routes should be start handling the incoming request. The middlewares and the route handlers (loaders or actions) of each route will be called in order from the parent

to the child sequentially. The middlewares and route handles (loaders or actions) can intercept the request, and prevent it from proceeding by throwing an early response.

3. **Store the loaded data in the cache**: After calling each loader or middleware the returned data will be cached in a Cache object to be used in the rendering process and prevent data loss.

After router find and loads a route tree, it's ready to be rendered on the page using a rendering library. In this project we have the `ziro/react` to render the router in the React environment.

6.1.7 Web standards

The Ziro Router is built on top of web standards, specifically the `Request` and `Response` objects. These objects are part of the Fetch API, which is a standard way to handle HTTP requests and responses in JavaScript. By leveraging these web standards, Ziro ensures compatibility and consistency across different environments, whether it's running on the client-side in a browser or server-side in a Node.js environment. Every loader, action or middleware described bellow has access to the request/response objects (based on their responsibility) to handle the incoming request and the response object to intercept the generated response and do proper action if needed.

6.1.8 Loaders

Loaders are essential functions for building dynamic and data-driven applications with Ziro. They help manage data fetching and preparation before a route's main content is rendered. Understanding how to use loaders is key to building applications that are both efficient and maintainable.

What are Loaders in Ziro?

Loaders in Ziro are functions associated with routes that handle data fetching and preparation. They are especially valuable for server-side data loading, enabling data to be fetched from external APIs, databases, or other sources. Loaders are executed after the middlewares registered on the route and before starting the process of loading the nested routes.

Example of a Simple Loader

Here is a simplified example to illustrate how a loader function works in Ziro:

```
import { Router, Route } from 'ziro/router';

const router = new Router()

const usersRoute = new Route('/users/:userId', {
  loader: async ({ params }) => {
    // Fetch user data from an API
    const response = await fetch(`https://api.com/users/${params.userId}`);
    if (!response.ok) {
      throw new Error('Failed to load user data');
    }
    const userData = await response.json();

    // Return the user data to store in the cache
    return {
      name: userData.name,
      email: userData.email,
      avatar: userData.avatar,
    };
  };
});

router.addRoute(router);
```

Figure 3: Example of a Simple Loader

6.1.9 Middlewares

Ziro middleware is a powerful tool used to handle various tasks before or after a route's request and response cycle. Middleware functions run in sequence and can be used to modify requests, handle authentication, log information, or perform other pre- and post-processing actions. Here, we will discuss the purpose of middleware and present a simple example to illustrate how it works.

What is Middleware in Ziro?

Middleware in Ziro is a function that can intercept and modify requests and responses at different points in the lifecycle of a request. It allows for reusable

logic that can be shared across multiple routes or endpoints. Middleware can be executed at various stages, such as before a request is processed (`onRequest`) or before a response is sent (`onBeforeResponse`).

Simple Middleware Example

To understand how middleware works in Ziro, let's look at a simplified example:

```
import { Middleware } from 'ziro/router';

// Simple middleware that logs the request URL and method
export const simpleLogger = new Middleware('simple-logger', {
  async onRequest({ request }) {
    console.log(`Request received: ${request.method} ${request.url}`);
  },
  async onBeforeResponse({ response }) {
    console.log(`Response status: ${response.status}`);
  },
});
```

Figure 4: Simple Middleware Example

Explanation:

- **Import Statement**: We import Middleware from the `ziro/router` module.
- **Middleware Creation**: We create a new instance of Middleware, passing in a name ('simple-logger') and an object containing the `onRequest` and `onBeforeResponse` functions.
- `onRequest` **Function**: This function runs before the request is processed. In this case, it logs the HTTP method and the request URL to the console.
- `onBeforeResponse` **Function**: This function runs before the response is sent. It logs the status code of the response.

How Middleware Fits into the Router Lifecycle?

Middleware functions in Ziro can be attached to routes and are executed in the order they are added. When a request is made, the following sequence occurs:

1. The `onRequest` method of each middleware is executed in the order they are defined.
2. The route's handler (the loader or action) is executed to process the request and prepare a response. During this step, the current route's handler (the loader or action) runs, followed by any handler from nested routes. Middleware functions that are defined for these routes are also executed, running in sequence as per the route's configuration. This ensures that all necessary data is fetched and any pre-processing needed for the request is completed before generating the final response.
3. The `onBeforeResponse` method of each middleware runs in the reverse order before the response is sent to the client.

This enables developers to control and modify both the incoming request and outgoing response with ease.

More Complex Middleware Example

For more advanced use cases, middleware can be used to implement features such as logging, authentication, or performance tracking. Here is a more detailed example:

```
import { Middleware } from 'ziro/router';

// Middleware that logs request details and response time
export const requestLogger = new Middleware('request-logger', {
  async onRequest({ dataContext }) {
    dataContext.responseTime = Date.now();
  },
  async onBeforeResponse({ request, dataContext }) {
    const responseTime = Date.now() - dataContext.responseTime;
    const pathname = new URL(request.url).pathname;
    console.log(`Request to ${pathname} took ${responseTime}ms`);
  },
});
```

Figure 5: More Complex Middleware Example

In this example, the response time of the request will be logged by storing the current time on the `dataContext` (in the `onRequest` method) and will be logged right after the child routes loading completed (the `onBeforeResponse` method). The `dataContext` object is used to be a shared data layer through the loading the routes sequentially.

Example of Integrating Loader and Middlewares in a route

Here is an integrated example of how loaders and middlewares can be used in a route definition:

```
import { Router, Route } from 'ziro/router';
import { requestLogger } from './middlewares/request-logger'

const router = new Router()

const usersRoute = new Route('/users/:userId', {
  middlewares: [requestLogger],
  loader: async ({ params }) => {
    // Fetch user data from an API
    const response = await fetch(`https://api.com/users/${params.userId}`);
    if (!response.ok) {
      throw new Error('Failed to load user data');
    }
    const userData = await response.json();

    // Return the user data to store in the cache
    return {
      name: userData.name,
      email: userData.email,
      avatar: userData.avatar,
    };
  };
});

router.addRoute(router);
```

Figure 6: Example of Integrating Loader and Middlewares in a route

This example demonstrates how to use middleware on a route that has a loader. The middleware, named `requestLogger` (Figure 5), logs the request details and calculates the time taken by the loader to fetch data from the API. This is achieved by defining an `onRequest` function that stores the start time and an `onBeforeResponse` function that logs the request URL, method, and the total response time.

6.1.10 Actions

Ziro actions are an essential part of building dynamic applications, allowing developers to handle complex logic that goes beyond simple data fetching or

rendering. Actions in Ziro provide a mechanism to handle server-side operations, such as form submissions, data updates, and other tasks that require request handling and processing. By using actions, developers can create robust, interactive applications that manage data more efficiently and ensure that certain operations are secure and well-structured.

What are Actions in Ziro?

Actions in Ziro are special functions that can be associated with specific routes and used to handle mutation requests like POST HTTP requests. These functions are designed to handle tasks such as modifying data, performing validations, or managing interactions that involve user input. An action can be defined with an input validation schema and a handler function to process the request body.

Benefits of Using Actions

- **Input Validation**: Actions use schemas to validate the input data before processing. This ensures that only valid data is handled, preventing errors and enhancing security.
- **Centralized Logic**: Actions help in maintaining clear and modular code by placing related logic in one place, making the codebase easier to understand and maintain.
- **Customizable Behavior**: Each action can be customized to handle different types of requests and data, providing flexibility in how data is managed and manipulated.

What is a Schema in Actions?

A schema in actions is a data validation structure that defines the shape and constraints of the input data for actions. Schemas are crucial for ensuring that only valid data is processed, which helps prevent errors and enhances the security and reliability of the application. In Actions, schemas should be defined using **Zod**[31], a TypeScript-first schema declaration and validation library.

[31] https://zod.dev

What is Zod?

Zod is a TypeScript-first schema validation library that allows developers to define and validate data structures in a type-safe manner. With Zod, developers can create complex validation logic that is easy to read and maintain. It supports various data types and provides built-in methods for validating strings, numbers, arrays, objects, and more. The integration of Zod in actions makes it easier to enforce data integrity and ensure that actions receive correctly formatted input.

Benefits of Using Schemas in Actions

- **Type Safety**: Schemas are defined in TypeScript, providing strong type checking at compile time, which helps catch errors early in the development process.
- **Input Validation**: Using Zod schemas, actions can validate incoming data to make sure it meets the required conditions before processing.
- **Error Handling**: Schemas can specify detailed error messages, making it easier to understand why an input is invalid and guiding developers to fix issues efficiently.
- **Consistency**: Schemas ensure that data structures are consistent across different parts of the application, enhancing code maintainability.

You can find an example of actions in the figure 7

```
import { Route, Action } from 'ziro/router';
import { z } from 'zod';

let todos = [];

export const actions = {
  addTodo: new Action({
    input: z.object({
      title: z.string().min(3, 'Title must be at least 3 characters'),
    }),
    async handler(body, ctx) {
      todos.push({
        ...body,
        isDone: false,
      });
      return {
        ok: true,
      };
    },
  }),
  toggleTodo: new Action({
    input: z.object({
      index: z.coerce.number().min(0, 'This field is required'),
    }),
    async handler(body) {
      todos[body.index].isDone = !todos[body.index].isDone;
      return {
        ok: true,
      };
    },
  }),
  deleteTodo: new Action({
    input: z.object({
      index: z.coerce.number().min(0, 'This field is required'),
    }),
    async handler(body) {
      todos.splice(body.index, 1);
      return { ok: true };
    },
  }),
};

export const todoRoute = new Route("/todo", {
  loader: async () => todos,
  actions: actions
})
```

Figure 7: Example of Actions in a To-Do List App

6.1.11 Meta Functions

Meta functions are an important part of creating well-rounded web applications with Ziro. They help manage the meta-information of routes, such as page titles, descriptions, and other SEO-related data. By defining meta functions for routes, developers can enhance the discoverability and ranking of their web pages, improve social media sharing, and provide a better overall user experience.

What are Meta Functions in Ziro?

Meta functions in Ziro are functions that are associated with specific routes and are responsible for setting metadata in the HTML head. These functions receive data from the route's loader, allowing them to dynamically generate metadata based on the content of the page. Meta functions can be defined to return various pieces of meta-information, such as the page title, description, and other elements needed for SEO and social sharing.

Ziro's integration with the **unhead**[32] library makes it possible to manage meta tags in a reactive and dynamic manner, ensuring that the content in the HTML head is always up-to-date with the current route.

Benefits of Using Meta Functions

- **Improved SEO**: Properly configured meta functions ensure that search engines can better understand the content of each page, improving search rankings.
- **Enhanced Social Sharing**: Meta tags help generate rich previews on social media platforms when links are shared.
- **Dynamic and Context-Aware**: Meta functions can use data loaded by route loaders to generate context-specific metadata.
- **Consistency**: Using meta functions ensures that each route has consistent and up-to-date metadata.

[32] https://unhead.unjs.io/

Example of a Meta Function

```
import { Route, Action } from 'ziro/router';
import z from 'zod'

const todos = []

const meta = async ({ loaderData }) => {
  return {
    title: `${loaderData.todos.length} items in list | To-Do App`,
    description: `Track your tasks for free!`,
  };
};

const loader = async () => {
  return { todos };
}

const actions = {
    addTodo: new Action({
      input: z.object({
        title: z.string().min(1, 'Title is required')
      }),
      async handler(body){
        todos.push(body);
      }
    })
}

const todoRoute = new Route("/todo", {
  meta,
  loader,
  actions
})
```

Figure 8: Example of a Meta Function

6.1.12 Transitioning to Framework Development

While Ziro's core libraries provide powerful capabilities for handling routing, data loading, and meta management, building a complete framework requires integrating these functionalities within a cohesive development environment. This is where a modern bundler like **Vite** comes into play. Vite allows Ziro to efficiently handle dynamic module loading, and streamline development workflows. One of the key advantages of using Vite is the ability to implement a file-based routing system, eliminating the need to define routes manually. This approach simplifies the development process by allowing routes to be automatically generated based on the directory structure of route files. By leveraging Vite, Ziro evolves from a routing libary into a comprehensive framework capable of delivering web applications.

6.2 Vite Overview

Vite is a next-generation frontend tooling system that provides a fast and efficient development environment for modern web applications. By leveraging features like on-demand module loading, hot module replacement (HMR), and support for server-side rendering (SSR), Vite creates a seamless experience for developers building frameworks like Ziro.

Why Vite Enables Framework Development

- **Dynamic Module Loading**: Vite's `ssrLoadModule` method is a game-changer for frameworks. It allows modules to be dynamically loaded during SSR, making it possible to execute Javascript on-demand and generate updated values on the server based on latest file changes.
- **Hot Module Replacement (HMR)**: HMR drastically reduces development time by enabling instant updates to modules without requiring a full page reload. This feature is invaluable when working on a project, where frequent changes to files need immediate feedback.
- **File Watching and Auto Reload**: Vite's built-in file-watching mechanism ensures that any changes to the project's codebase are automatically detected. Combined with its optimized reload system, Vite provides a smooth development experience even for large-scale applications.
- **Enhanced SSR Capabilities**: Vite simplifies the implementation of server-side rendering by providing tools to manage server and client-side

modules separately. This separation ensures that Ziro's SSR functionality remains maintainable.

- **Plugin System**: Vite's extensible plugin system allows Ziro to integrate additional functionality, such as generating route-related files (`ziro/generator`) and handling file-based routing.
- **Vite is Fast**: By utilizing native ES modules (ESM), Vite removes the need for traditional bundling during development. This results in lightning-fast startup times, even for complex projects.

In summary, Vite's robust feature set and modern approach to tooling make it an ideal choice for bootstrapping a development server for Ziro. It not only accelerates the development process but also ensures that the resulting framework is scalable and maintainable.

6.3 Vite Plugins and Their Role in Ziro

Vite plugins are essential tools that extend the functionality of Vite, allowing developers to customize and automate parts of the development and build process. In the context of Ziro, Vite plugins play an important role in detecting file changes, generating route-related files and generating server-side rendered pages.

6.3.1 How Vite Plugins Work

A Vite plugin is essentially a JavaScript function or object that hooks into various stages of the Vite lifecycle[33], such as server startup, file changes, and builds. These plugins use Vite's robust API to perform tasks like file processing, module transformation, and custom behavior during development or production builds.

6.3.2 File Watching and Route Detection

Vite comes with a file-watching system based on the chokidar[34] library, which monitors the file system for changes. This feature allows Ziro's Vite plugin to:

[33] https://vite.dev/guide/api-plugin
[34] https://github.com/paulmillr/chokidar

- **Track Route Files**: By watching specific directories (like `pages/`), the plugin can detect when a new route file is added, modified, or removed.
- **Trigger Updates**: When changes are detected, the plugin triggers the generation of files necessary for the router to work correctly.

6.3.3 Generating Route-Related Files

When a change is detected in the route files, Ziro's Vite plugin takes the following steps to ensure the router remains up to date:

1. **Manifest Generation**: A `manifest.json` file is created, containing information about each route, such as its path, associated middleware, loaders, actions, etc...
2. **Server and Client Router Files**: Separate router files are generated for server-side (`router.server.ts`) and client-side(`router.client.ts`) usage. The server router handles SSR-specific logic, while the client router is optimized for browser execution, including lazy loading[35] for route components.
3. **TypeScript Definitions (`routes.d.ts`)**: The plugin generates a TypeScript file that defines types for all routes. This ensures type safety, making it easier for developers to work with route data in a consistent way.

6.3.4 Benefits of Vite Plugins in Ziro

- **Automation**: Developers don't need to manually update route configurations or related files when they add or modify routes. The plugin handles everything automatically. It generates the route objects (`new Router(...)`) based on the exports of the route files.
- **Efficiency**: By using Vite's built-in file watching and caching mechanisms, updates are fast and incremental, meaning only the affected files are regenerated.
- **Consistency**: The plugin ensures that all generated files, such as `routes.d.ts`, are always in sync with the file system. This reduces the risk of bugs caused by mismatched configurations.

[35] https://react.dev/reference/react/lazy

- **Dynamic File Transformation**: During the build or development process, the plugin can modify or transform route modules to exclude server-only logic or optimize for the client environment.

6.4 Manifest Generation

To maintain a single source of truth and support server/client-specific routers, Ziro generates a manifest file that encapsulates all route-related information. This manifest serves as a centralized repository, containing details about each route and its associated exports. It simplifies route management, type generations, and ensures that both server-side and client-side routers built on single source of truth.

6.4.1 Why a Manifest is Necessary

In file-based routing systems, routes are defined dynamically based on the structure of route files and their exported variables. The manifest acts as the backbone of this system by:

- Storing metadata about each route file, including its path and exported properties.
- Facilitating the generation of server and client router files with consistent data.

6.4.2 Structure of the Manifest

A manifest is represented as a **JSON** object. Below is an example of a manifest for a simple application with a root, layout and a single page at `/`:

```
{
  "/_root": {
    "id": "/_root",
    "routeInfo": {
      "filepath": "pages/_root.tsx",
      "index": false,
      "hasActions": false,
      "hasComponent": true,
      "hasErrorBoundary": true,
      "hasLoader": true,
      "hasLoadingComponent": true,
      "hasMeta": true,
      "hasMiddleware": true,
      "hasLayout": true
    }
  },
  "/_layout": {
    "id": "/_layout",
    "routeInfo": {
      "filepath": "pages/_layout.tsx",
      "index": false,
      "hasActions": false,
      "hasComponent": true,
      "hasErrorBoundary": true,
      "hasLoader": true,
      "hasLoadingComponent": true,
      "hasMeta": true,
      "hasMiddleware": false,
      "hasLayout": true
    },
    "parentId": "/_root"
  },
  "/": {
    "id": "/",
    "routeInfo": {
      "filepath": "pages/index.tsx",
      "index": true,
      "hasActions": false,
      "hasComponent": true,
      "hasErrorBoundary": false,
      "hasLoader": true,
      "hasLoadingComponent": true,
      "hasMeta": true,
      "hasMiddleware": false,
      "hasLayout": false
    },
    "parentId": "/_layout"
  }
}
```


Figure 9: Example of manifest with 1 route at `/`

6.4.3 Components of the Manifest

1. **Route ID (`id`)**: A unique identifier for the route, corresponding to its path.
2. **Route Information (`routeInfo`)**: Details about the route file, such as:
   - `filepath`: The relative path to the file in the project structure.
   - `index`: Indicates if the route is an index route.
   - `hasComponent`: Indicates if the file has a default export.
   - `hasLayout`: Indicates if the file has a export called 'layout'.
   - `hasErrorBoundary`: Indicates if the file exports a variable called `ErrorBoundary` (a React specific rule).
   - `hasLoader`: Indicates if the file exports a variable called `Loading` (a React specific rule).
   - `hasLoader`, `hasActions` `hasMeta`, `hasMiddleware`: Flags for other exported properties like loaders, actions, meta functions, and middleware.
3. **Parent ID (`parentId`)**: Specifies the parent route for nested routes, enabling hierarchical routing.

6.4.4 How the Manifest is Generated

1. **File System Analysis**: During the development server's startup, Ziro uses Vite's file system watcher and the `es-module-lexer`[36] library to scan the project directory for route files.
2. **Export Inspection**: For each route file, `es-module-lexer` identifies the exported variables (e.g., `loader`, `meta`, `actions`, `middleware`, `Loading`, `ErrorBoundary`, and the `default` component).
3. **Manifest Construction**: The gathered data is structured into a JSON object following the manifest format.

[36] https://github.com/guybedford/es-module-lexer

6.4.5 Output of Manifest Generation

Once the manifest is created, it serves as the input for generating `router.server.ts` and `router.client.ts`. These files contain optimized routing logic tailored for the server and client environments, respectively.

For example:

- `router.server.ts` runs only on the server and handles server-side routing, including loaders, actions and middleware.
- `router.client.ts` runs on the browser and focuses on client-side navigation and lazy loading components.

The manifest ensures that both files are generated consistently, reflecting the same route structure.

6.4.6 Why This Approach Works

By using a manifest file, Ziro achieves:

- Both server and client routers share a single source of truth.
- The manifest is generated automatically during development, reducing manual configuration.
- New routes and modifications are detected dynamically, updating the manifest and routers in real time.

This process forms the foundation of Ziro's file-based routing system, enabling seamless integration of server and client functionality.

Hint: By extending the `manifest.json` we can add custom pages or layouts out of the project directory to our app. It happens because the server/client router will be generated based on this manifest file. This is actually the basis of the Ziro Plugin which will be discussed further in the next sections.

6.4.7 Example of a route serves at `/`

```tsx
// pages/index.tsx

import { MetaFn } from 'ziro/router'

export const loader = async () => {
  return fetch('https://api.example.com/data').then(r => r.json())
}

export const meta: MetaFn<'/'> = async () => {
  return {
    title: 'Homepage',
  }
}

export default function Index() {
  return (
    <div>
      <h1>Home page</h1>
    </div>
  )
}

export const Loading = () => {
  return <span>Loading...</span>
}

export const ErrorBoundary = () => {
  return <span>Something went wrong, please reload the page.</span>
}
```

In this example, when this route is going to to be rendered on the browser, it shows the `Loading` component while the data is being fetched and the `ErrorBoundary` component if an error occurs during the loading process. The `meta` function is used to set the page title to "Homepage" when the route is rendered on the browser. The `loader` function fetches data from an API endpoint and returns it to be used in the component. The `default` export is the main component that will be rendered when the route is loaded successfully.

6.5 Router Generation

As described in the previous section, to ensure a seamless routing experience tailored for both client and server environments, Ziro generates two distinct router files based on the manifest: `router.client.ts` and `router.server.ts`. These files contain the necessary routing logic optimized for their respective environments.

6.5.1 Route Naming Conventions

The naming conventions for the route files in the pages directory determine how routes are generated and how layouts and contexts are applied. This structured convention is important on generating the routers. Below are the key conventions:

Special Files

- `_root.tsx`: This file represents the root of the application. It is used to define the top-level contexts and layouts that will wrap the entire app. All routes in the application will inherit from the contexts or layouts defined here.
- `_layout.tsx`: This file acts as a layout wrapper for every route within the specific directory and its child directories. It is useful for defining shared components like headers, footers, or navigation menus that need to appear across multiple routes in the directory.
- `index.tex`: This file represents the index route of the directory. It serves as the default route for the directory when no specific route is provided.
- **Dynamic routes**: Any file that begins with a colon (`:`) generates a dynamic named route. For example, `:slug.tsx` defines a route that matches URLs like `/blog/post-slug`, where `post-slug` can be any dynamic value. Dynamic routes are particularly useful for handling paths based on unique identifiers or variables in the URL.

6.5.2 Example File Structure and Generated Routes

Consider the following directory structure:

```
pages/
- _layout.tsx
- index.tsx
- blog/
  - _layout.tsx
  - index.tsx
  - :slug.tsx
```

This structure generates the following index routes:

1. `/` : Wrapped by the root layout defined in `pages/_layout.tsx` .
2. `/blog` : Wrapped by both `pages/_layout.tsx` and `pages/blog/_layout.tsx` .
3. `/blog/:slug` A dynamic route where `slug` is replaced with a specific value, such as `post-1` or `my-first-post` . Also wrapped by `pages/_layout.tsx` and `pages/blog/_layout.tsx` .

The `_layout.tsx` files allow for nested layouts, where a directory-level layout wraps its routes and those in child directories. For instance, `pages/_layout.tsx` serves as the outermost layout for the app (after `_root.tsx` file, if defined), while `pages/blog/_layout.tsx` defines a layout for routes within the `blog` directory, inheriting the root layout. This modular and scalable approach makes it easy to handle shared components, manage complex route hierarchies, and implement reusable layouts, ensuring a clean and predictable structure for building applications. Dynamic routes further enhance flexibility by allowing URL patterns to adapt to varying data.

6.5.3 Client-Side Router (`router.client.ts`)

The client-side router is designed to handle navigation and dynamic component loading in the browser. It leverages React's `lazy` for code splitting and ensures that components, meta functions, and loading/error boundaries are loaded only when needed.

```
import { Router, Route } from 'ziro/router';
import { lazy } from 'react';

const router = new Router({ "mode": "csr" });

const dynImport = () => import("../pages/index")

const pages_indexRoute = new Route("/", {
  meta: (...args) => dynImport().then(m => m.meta(...args)),
  loader: (...args) => dynImport().then(m => m.loader(...args)),
  props: {
    component: lazy(dynImport),
    ErrorBoundary: lazy(() => dynImport().then(m => ({ default: m.ErrorBoundary }))),
    LoadingComponent: lazy(() => dynImport().then(m => ({ default: m.Loading }))),
    Layout: lazy(() => dynImport().then(m => ({ default: m.Layout }))),
  },
});
router.addroute(pages_indexRoute);

// Similar logic for other routes...

export router;
```

Figure 10: Example of router.client.ts

Details:

- Components and route parts (`meta`, `loader`) are lazily loaded using `import()` to optimize performance so only the required parts of the app are downloaded as users navigate between routes.
- Props defines the Component, Loading, and ErrorBoundary for each route.

6.5.4 Server-Side Router (`router.server.ts`)

The server-side router focuses on server-specific logic, such as loaders and middleware. This file is responsible for handling requests and preparing the data needed for server-side rendering.

```
import { Router, Route } from 'ziro/router';
import * as pagesIndex from '../pages/index';

const router = new Router({ "mode": "csr" });

const pagesIndexRoute = new Route("/", {
  meta: pagesIndex.meta,
  loader: pagesIndex.loader,
  props: {
    component: pagesIndex.default,
    ErrorBoundary: pagesIndex.ErrorBoundary,
    LoadingComponent: pagesIndex.Loading,
    Layout: pagesIndex.Layout,
  },
});
router.addroute(pages_indexRoute);

// Similar logic for other routes...

export router;
```

Figure 11: Example of router.server.ts

Details:

- All components and metadata are directly imported, as the server does not benefit from lazy loading.
- When router mode has set to `ssr` or `partially-ssr`, the server router file includes server-only logic like loaders for data fetching and middleware for request handling. In this case, the `router.client.ts` doesn't include the server methods (loaders, middlewares or actions).

6.5.5 Why Two Separate Router Files?

The client router needs to be optimized for browser environment by importing the route modules dynamically, while the server router focuses on loading all required resources to handle SSR in a single file. To achieve this, we need to have 2 seperated files with totally different syntaxes. Also server and client have different requirements for resource loading and routing logic. Separating these concerns ensures a safer architecture as the server-logic code doesn't get leaked to the browser. For example, when the router mode is set to `ssr`, the client

router file doesn't include server-only logic like `loaders` and `middlewares`, which should be run on the server.

6.6 TypeScript Integration

During file generation chain of Ziro, a typescript declaration file will be generated called `routes.d.ts`. It acts as the type declaration file for all routes in the application, ensuring that developers can rely on accurate type information for each route.

Here's a sample of a generated routes.d.ts file:

```ts
import {
  Route,
  LoaderReturnType,
  IntersectionOfMiddlewaresResult,
  GetRouteDataContext
  } from 'ziro/router';
import * as pages_root from '../pages/_root';
import * as pagesIndex from '../pages/index';

declare module 'ziro/router' {
  interface RouteFilesByRouteId {
    "/_root": {
      route: Route<
        "/_root",
        LoaderReturnType<typeof pages_root.loader>,
        {},
        typeof pages_root.middlewares,
        undefined
      >;
      dataContext: {} & IntersectionOfMiddlewaresResult<
        typeof pages_root.middlewares
      >;
    };
    "/": {
      route: Route<
        "/",
        LoaderReturnType<typeof pagesIndex.loader>,
        {},
        [],
        RouteFilesByRouteId["/_root"]["route"]
       >;
      dataContext: GetRouteDataContext<"/_root"> &
        IntersectionOfMiddlewaresResult<[]>;
    };
  }
  interface RoutesByRouteId {
    routes: "/";
  }
}
```

Figure 12: Example of routes.d.ts

Explanation Key Components

- `RouteFilesByRouteId` Interface:
  - Maps each `route ID` (e.g., "_root") to its corresponding route configuration.

  - Includes:
    * `route`: Defines the route structure using the Route type, including its loader, actions, middlewares, meta function, and parent route.
    * `dataContext`: Specifies the combined data context derived from middlewares and parent routes.

- `RoutesByRouteId` Interface:
  - Lists all **index-only** `route IDs` in the application for quick reference.
  - Useful for navigating user to a destination within the app.

- Generic Types:
  - Route: The core generic type defining the structure of a route. Includes:
    * Route ID.
    * Return type of the loader.
    * Types for actions and middlewares.
    * Parent route structure.

- `LoaderReturnType`: Infers the return type of a loader, ensuring accurate typing for loaded data.

- `IntersectionOfMiddlewaresResult`: Merges the output of all middlewares in the route hierarchy into a single type.

- `GetRouteDataContext`: Retrieves the data context for a specific `route ID`, combining parent and middleware data. It's added in a seperated property to avoid loop in type definition.

How This Helps in Development?

Having a fully typed version of the router, makes it possible to write some utility types around it to make the whole app typesafe. For example, the `loader` function in each route, should be typesafe and aware of it's parent data (As explained, parent loaded data should be accessible in the child loader). But how is that possible? The `routes.d.ts` file is the answer. It owns the necessary types to make the loaders typesafe. For example, the `LoaderArgs` is a utility type that has been written around `RouteFilesByRouteId` interface (which is overridden in the `routes.d.ts` file) that provides necessary types to the loader. Here is an example of how to use it:

```tsx
// pages/_root.tsx
  import { LoaderArgs } from 'ziro/router'

  export const loader = (ctx: LoaderArgs<'/_root'>) => {
    // the root's loader returns some random data like:
    return {
      version: '1.0'
    }
  }

// pages/index.tsx
  import { LoaderArgs } from 'ziro/router'

  export const loader = (ctx: LoaderArgs<'/todo'>) => {
    // ctx is fully type safe and it's parent data is accessible and typesafe
    // here ctx.dataContext.version is available and fully typesafe.
    return {
      appName: `My App v${ctx.dataContext.version}`
    }
  }
```

6.7 Ziro Plugins

Ziro's plugin system is a powerful feature that allows developers to extend their applications by adding custom routes, layouts, and functionality in a modular and reusable way. By integrating directly with the `manifest.json` file, plugins can introduce new pages or layouts that seamlessly integrate with the existing application's routing structure. This system is particularly useful for building scalable applications where features can be added or modified without directly changing the core codebase.

6.7.1 How Ziro Plugins Work

Ziro plugins are registered in the `vite.config.js` file, because they have to integrate with the framework's manifest generation process. For example:

```
import { dashboard } from './plugins/dashboard';
import { defineConfig } from 'vite'

export default defineConfig({
  plugins: [
    ziro({
      plugins: [dashboard],
    }),
  ],
});
```

When the development server starts, Ziro detects the registered plugins and triggers their `registerRoutes` method during the manifest generation process. The plugins then add custom routes or layouts to the `manifest.json` file. This ensures that both server and client routers include the new additions during the file generation process.

6.7.2 Simplified Example of a Ziro Plugin

```
import { Plugin } from 'ziro/generator';

type Config = {
  dashboardPath: string; // The route path for the dashboard page
};

export const dashboardPlugin = new Plugin<Config>(
  'dashboard', // Plugin name
  {
    registerRoutes(config) {
      return [
        {
          routeId: config.dashboardPath || '/dashboard',
          filePath: new URL(
            './dashboard.tsx',
            import.meta.url
          ).pathname,
        },
      ];
    },
  },
  {
    configPath: 'configs/dashboard.ts', // Path to the plugin's configuration file
  },
);
```

Figure 13: Simplified Example of a Ziro Plugin

6.7.3 Key Components of the Plugin

The Ziro plugin system includes several core components that work together to extend the application's functionality. The plugin's `name`, such as dashboard, uniquely identifies it and ensures it can be reused and distinguished from other plugins. A configuration object (`Config`) allows developers to type the plugin's config. The `registerRoutes` method defines the custom routes introduced by the plugin, including a unique identifier (`routeId` or `path`) and the location of the file that implements the route's component (`filePath`). Additionally, the `configPath` provides a pointer to an external configuration file (`configs/dashboard.ts`), which makes it possible to config the plugins based on project needs.

6.7.4 How Plugins Modify the Manifest

When a plugin is added, it extends the `manifest.json` file with additional route definitions. For example:

```
{
  "/dashboard": {
    "id": "/dashboard",
    "routeInfo": {
      "filepath": "./node_modules/installed-plugin/dashboard.tsx",
      "index": true,
      "hasActions": false,
      "hasComponent": true,
      "hasErrorBoundary": false,
      "hasLoader": false,
      "hasLoadingComponent": false,
      "hasMeta": false,
      "hasMiddleware": false,
      "hasLayout": false
    },
    "plugin": "dashboard"
  }
}
```

This manifest entry ensures that the server and client routers include the new `/dashboard` route when they are generated. The plugin system effectively decouples feature development from the core project, making it easier to add and manage new functionality.

Ziro plugins can be used to address a variety of practical needs, such as adding authentication systems, admin dashboards, or analytics tools. For example, a plugin can introduce login, registration, and authentication-related routes while also providing logic for securing specific pages. Similarly, admin dashboards can be added as pre-built pages and layouts, complete with tools like rate-limiting middleware for sensitive sections and their child routes. For e-commerce applications, plugins can enable features like shopping carts, product pages, and checkout workflows. These use cases demonstrate the flexibility and power of Ziro plugins to simplify development and enhance application functionality.

6.8 Request Handling on the Server

In Ziro, request handling on the server is a critical part of activating server-side rendering (SSR). When a request is made, the server must load the router and prepare a response based on the application's routing logic and data-loading mechanisms. This process ensures that users receive the appropriate response, whether it's a fully-rendered page or a partially-streamed HTML document.

6.8.1 SSR Activation and Router Loading

To handle requests in SSR mode, the server must render the router. This involves loading all routes by executing their associated middlewares and loaders sequentially, starting from the parent routes and proceeding to their children. The sequential execution ensures that:

- **Middlewares** process the request, performing tasks such as authentication, validation, or request transformation.
- **Loaders** fetch the necessary data for each route, building the context needed for rendering the page.

If any middleware or loader throws a response (e.g., a redirect or error response), the server immediately sends this response back to the user without further processing. This mechanism guarantees precise handling of redirects, errors, and other special cases.

6.8.2 Handling SSR and Partially-SSR Modes

- **SSR Mode**: In SSR mode, the server waits for all routes to finish their loading processes. This ensures that the full HTML document is ready before sending it to the user. While this approach ensures a complete response, it may increase the initial response time.
- **Partially-SSR Mode**: In partially-SSR mode, the server begins streaming the HTML to the user as soon as possible. Instead of waiting for all routes to complete their loaders, it sends the initial HTML structure and streams additional content as it becomes available. This approach significantly reduces the perceived loading time for users, improving their experience.

In both modes, if no error or custom response is thrown during the loading process, the server proceeds to render the router using `renderToReadableStream` (from React library). This method generates a stream of HTML that is sent to the user incrementally.

6.8.3 Integration with the Web Server

To manage these processes, Ziro integrates with a web server. The chosen server framework, **H3**, handles the HTTP requests and responses while coordinating with Ziro's router logic. This integration ensures smooth and efficient delivery of server-rendered pages and guarantees that the server works regardless of the Vite's development server as in the production environment, the server is responsible for handling all requests and responses without Vite.

6.8.4 Why H3 as the Web Server?

H3 was chosen as the web server for Ziro due to its modern and lightweight design, making it an ideal fit for handling the needs of a high-performance SSR framework. H3 is a runtime-agnostic web framework built on top of popular technologies that offer both flexibility and performance.

A key feature of H3 is its runtime-agnostic nature, which provides support for different types of request handlers. In Ziro, this means that H3 can operate seamlessly on top of the standard Web `Request` and `Response` interfaces, ensuring full compatibility and a consistent handling experience. This allows Ziro to process HTTP requests and responses efficiently while supporting SSR logic, middleware execution, and data loading in a streamlined manner.

Another significant advantage of using H3 is its capability to deploy applications across different runtimes without the need for multiple adapters. Whether deploying on `Cloudflare workers/pages`, `Bun`, or `Deno`, H3 handles the compatibility, simplifying the deployment process and making it easier to maintain cross-runtime consistency.

Overall, H3's efficient handling of HTTP requests, runtime flexibility, and compatibility with modern web standards make it the perfect choice for supporting Ziro's request handling and server-side rendering capabilities.

6.9 React integrations

Ziro's core router library, `ziro/router`, is designed as a headless routing solution, which means it handles routing logic without any built-in user interface (UI). To bring this routing logic to life in a web application, a UI integration is necessary. This is where `ziro/react` comes into play, making it possible to connect Ziro's routing capabilities with React applications.

6.9.1 Setting Up React Integration

To integrate Ziro with React, the client-side entry point of the application is typically a JSX file that renders the router to the DOM. Here is an example of how the main entry file might look:

```
1  import { hydrateRoot } from 'react-dom/client';
2  import { Router } from 'ziro/react';
3  import router from '/.ziro/router.client.ts';
4
5  hydrateRoot(document, <Router router={router} />);
```

In this setup, the `hydrateRoot` method from React is used to render the router into the existing HTML structure. The Router component from `ziro/react` takes the router prop, which is imported from the generated `router.client.ts` file. Once the router is loaded and ready, it will handle the rendering of the route components on the page, making the application fully interactive.

6.9.2 Key Utilities for React Integration

To make Ziro compatible with the React ecosystem, `ziro/react` provides several utilities that enhance its integration:

- `Link` **Component**: This component allows for seamless navigation between pages within the application. It functions similarly to the Link component from **React Router**[37], making it intuitive for developers familiar with React.

[37] https://reactrouter.com/start/library/navigating#link

- `useAction` **Method**: The useAction method provides a way to call route actions directly from React components. This utility helps handle user interactions that need to trigger route-based actions, such as form submissions or other state-changing events fully type-safe.

- `Outlet` Component: To render a child route within its parent route, Ziro uses the `<Outlet />` component, similar to React Router's implementation. When a request is made, the framework identifies the route tree, which includes all parent and child routes from the top-level (_root.tsx or _layout.tsx) to the specific leaf route. The `<Outlet />` component determines where the child routes should be rendered within the parent route's layout. This ensures a seamless rendering hierarchy, allowing developers to define layouts and nest routes effectively.

6.10 Client-Only Server-Only Transformation

When using SSR (Server-Side Rendering) in Ziro, it's essential to ensure that server-only code, such as database connections and private API calls, is not exposed to the client-side JavaScript. Ziro achieves this by integrating Vite and **Babel**[38] to transform route modules and control what is sent to the client.

Vite, which is based on ECMAScript modules (ESM), provides a fast and efficient development server by transforming modules on demand. This means that Vite only processes code when a module is requested by the browser, leading to quicker development and optimized builds. Ziro's Vite plugin takes advantage of this by adding a custom step to the transformation process. When a route module is requested, the plugin uses Babel to scan and remove any server-specific code, such as loaders, actions, and middlewares, ensuring that only client-compatible code remains.

This approach protects sensitive server-side operations, such as database queries or internal API interactions, from being exposed in the client-side bundle. It maintains the security of the application while delivering an optimized, lightweight JavaScript bundle for the browser. By leveraging Vite's ESM-based transformation and adding custom filtering logic with the Ziro plugin, this method enhances both performance and security.

[38] https://babeljs.io/

6.10.1 Caching, the Shared Layer Between Server and Client

Data caching plays a crucial role in synchronizing server-side rendered (SSR) routes with the client to ensure efficient data transfer and rendering. In Ziro, this mechanism is designed to optimize data flow between the server and client, providing a seamless and fast user experience.

In SSR mode, the server is responsible for running the middlewares, loaders and actions to load data. When a client makes a request for a route, the client router sends a request to the server to retreive the necessary data. The server processes this request, executes the necessary middlewares and loaders and loads the relevant data using the route loaders. During this data loading phase, the server generates a cache object containing the route data and responses. This cache object is then serialized as JSON and sent back to the client router. The client router receives this cached data and directly loads it into its own router cache. The client router can then use the data stored in its cache to render the components.

In contrast, in Client-Side Rendering (CSR) mode, data handling is managed entirely on the client. Since the server only serves a minimal HTML shell and static assets, all loaders, middlewares, and data-fetching operations are executed within the client's browser. The client fetches data as needed using JavaScript running in the browser, so there is no need for a data synchronization process from the server. This approach simplifies the interaction and reduces server load but may result in longer initial load times compared to SSR.

This data caching strategy in SSR mode ensures consistency between the server-rendered output and the client's rendered state, providing a smooth transition when the page is loaded. By reusing the data already fetched and processed by the server, the client can render routes quickly without additional network requests.

6.11 How to setup a project using Ziro

Setting up a project with Ziro involves creating a React-based Vite project, installing the Ziro package, and configuring both Vite and TypeScript to integrate Ziro's routing and rendering system. Below, we'll walk you through the necessary steps to get your Ziro project up and running.

1. **Create a React-Vite Project** First, create a new Vite project with the React template using the following command:

```
npm create vite@latest my-ziro-app -- --template react-ts
```

This command will create a new React project with TypeScript support.

2. **Install the Ziro Package** Navigate to your project directory and install the Ziro package:

```
npm i ziro
```

3. **Configure vite.config.ts** Set up your `vite.config.ts` file to include Ziro as a plugin:

```
import react from '@vitejs/plugin-react';
import { defineConfig } from 'vite';
import ziro from 'ziro/vite';

export default defineConfig({
  plugins: [
    react(),
    ziro(),
  ],
});
```

This configuration ensures that Vite runs with Ziro's plugin.

4. **Update tsconfig.json** To make sure TypeScript recognizes Ziro's type declarations, modify your `tsconfig.json` to include the Ziro type definitions:

```
  {
  "compilerOptions": {
    "strict": true,
    "esModuleInterop": true,
    "jsx": "react-jsx",
    "module": "esnext",
    "types": [
      "vite/client",
      "./.ziro/routes.d.ts"
    ],
    "moduleResolution": "Bundler"
  }
}
```

5. **Create Route Files**: Create the pages directory and add the following route files.

```tsx
// _root.tsx
import { FC, PropsWithChildren } from 'react';
import { Body, Head, Html, Outlet, RouteProps } from 'ziro/react';
import baseStyle from './styles.css?url';

export default function Root(props: RouteProps<'/_root'>) {
  return <Outlet />;
}

export const Layout: FC<PropsWithChildren> = ({ children }) => {
  return (
    <Html>
      <Head>
        <meta charSet="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <link href={baseStyle} rel="stylesheet" />
      </Head>
      <Body>{children}</Body>
    </Html>
  );
};
```

```tsx
    // index.tsx
import { MetaFn } from 'ziro/router';

export const loader = async () => {
  return {};
};

export const Loading = () => {
  return <span>Loading home page...</span>;
};

export const meta: MetaFn<'/'> = async () => {
  return {
    title: 'Homepage',
  };
};

export default function Index() {
  return (
    <div>
      <h1>Hello World!</h1>
    </div>
  );
}
```

These files define the main route layout and the homepage, including components and metadata.

6. **Update package.json** Scripts Modify the scripts section in your package.json to run the project with Ziro:

```
"scripts": {
  "dev": "ziro dev",
  "build": "ziro build",
  "preview": "vite preview"
}
```

This configuration ensures that running `npm run dev` starts the development server with Ziro, `npm run build` generates a production build, and `npm run preview` previews the built application.

With these steps, you now have a React-Vite project set up to work with Ziro, enabling server-side rendering, routing, and type safety out of the box. This setup provides an efficient development experience with modern tools and seamless server-client interactions.

7 Results

The Ziro package has been successfully published to the npm registry, with the latest version being `0.0.16`[39] as of now. The package is actively being developed and will likely see future releases with additional features and improvements.

Current Features and Capabilities

The core functionality of Ziro is well-developed and includes several key features that have been tested and implemented:

- **SSR, CSR, and Partially-SSR Modes**: The three rendering modes—Server-Side Rendering (SSR), Client-Side Rendering (CSR), and partially-SSR—are fully developed and tested. This allows developers to choose the rendering strategy that best fits their project needs.
- **Server-Client Integration**: The integration between the server and client has been successfully implemented, enabling smooth communication and data transfer between the two.
- **Type Safety**: Loaders, middlewares, and actions are now type-safe, ensuring that developers can leverage TypeScript for more reliable code in addition to better developer experience.
- **Plugin System**: A basic plugin system has been implemented, allowing users to add custom pages and layouts. This system is designed to be extendable, with more flexibility and additional features planned in the near future.
- **Example Project**: A simple example application is available in the examples directory of the repository, demonstrating basic usage and providing a starting point for developers. The example can be accessed at: https://github.com/narixius/ziro-vite
- **Testing**: The `ziro/router` has been tested using Vitest, ensuring that the core router works as expected and that future changes can be made with confidence.

It is important to note that only the development environment has been implemented so far. The production build process for Ziro-based projects has

[39] https://www.npmjs.com/package/ziro

not yet been completed. Future development will focus on extending the framework to support production builds, ensuring that projects built with Ziro can be deployed efficiently on various environments.

8 Future Works

The future works listed bellow can help Ziro evolve into a comprehensive and powerful framework that meets the needs of modern web development, making it more robust, flexible, and suitable for a wider range of applications. Here are some potential directions and enhancements that can be included to expand the capabilities of Ziro and make it more robust and useful:

- **Improved Plugin System**: Enhance the plugin system to support more complex functionalities, such as dynamic route generation, general middlewares, general actions and having more controll over the requests for the middlewares, loaders and actions. This would make it easier for developers to create and integrate more sophisticated plugins.
- **CLI Tool for Project Initialization**: Develop a command-line interface (CLI) tool similar to Vite's, allowing developers to create pre-configured Ziro projects easily. This tool could be invoked with commands like `npm create ziro`, streamlining the setup process and ensuring best practices are followed from the start.
- **HMR Support for Ziro in Vite**: Adding first-class Hot Module Replacement (HMR) integration to Ziro's Vite plugin will enable real-time updates to routes and components without a full page reload, improving development speed and maintaining app state.
- **Production Build Support**: Develop and integrate the production build functionality to ensure that Ziro-based projects can be deployed for production environments. This would include features like code minification, asset optimization, and efficient handling of production-specific configurations.
- **Cross-Platform Compatibility**: Expand Ziro's compatibility to support at least one additional runtimes, such as serverless environments and various cloud hosting providers (e.g., `Cloudflare workers/pages`, `Netlify`). This would enable developers to deploy Ziro-based applications more flexibly.
- **React 19 Support**: Ensure compatibility with the latest version of React (React 19). This includes updating dependencies, testing for breaking changes, and leveraging new features and improvements introduced in React 19 to enhance the overall developer experience.

- **Server-Side Caching**: Implement server-side caching mechanisms to improve performance, reduce redundant data fetching, and enhance the overall response time of Ziro-powered applications.
- **Testing `ziro/react` Integration**: Since `ziro/react` is crucial for connecting the Ziro router with React applications, thorough testing should be conducted.
- **Real-World Solutions**: Implementing solutions and examples for common needs like authentication, dashboards, and i18n will make Ziro more practical. These examples will help developers quickly integrate essential features into their projects.
- **Documentation and Community Support**: Provide more comprehensive guides, tutorials, and examples that cater to both beginners and advanced users.
- **Performance Benchmarking**: Conduct performance benchmarking and optimization to ensure that Ziro performs well under various loads and use cases. This would help identify potential bottlenecks and refine the performance of the framework.

9 Acknowledgements

I would like to express my deepest gratitude to my supervisor, Soroush Vedaei, for their invaluable guidance, support, and mentorship throughout this project. Their expertise and encouragement have been instrumental in helping me navigate the complexities of developing Ziro. I am also grateful to all the professors and faculty members at BIHE who have provided me with a strong foundation in computer engineering and valuable insights that contributed to the success of this work. Their dedication to teaching and research has been inspiring, and I am thankful for the knowledge and skills I have gained under their guidance.

Figure 14: A moment of a memorable night