ChatGPT

Custom Client-Side Routing Specification

Overview

This document specifies a **custom client-side routing framework** for the ToddleToy application. The router is implemented in **pure vanilla JavaScript** with no external routing libraries, leveraging the browser's History API for navigation control. Key features and requirements include:

- **History API Integration:** Use `history.pushState` and `history.replaceState` for navigation without full page reloads [1] [2]. This allows the app to update the URL (for routes like `/`, `/toy`, `/admin`) and respond accordingly without navigating away from the single-page application.
- **Popstate Event Handling:** Listen for the `window.onpopstate` event to handle user-initiated navigation (browser back/forward buttons) [3]. The router must intercept these events and load the appropriate screen state when the history changes.
- **Route Storage:** Maintain a registry of routes using a simple **Map** from path strings to handler functions [4] [5]. This Map will map route paths (e.g. `'/'`, `'/toy'`, `'/admin'`) to the functions that render or activate those screens.
- **Protected Routes (Route Guards):** Implement route guard logic to prevent unauthorized access to certain routes. In particular, the main game route (`/toy`) should **only be accessible after the user has completed the configuration step**, not via direct URL entry [6]. The router will use a flag to determine if access to `/toy` is allowed, and redirect to the default route if not.
- **State Preservation on Refresh:** Ensure that if the user refreshes the page while on the game screen (`/toy`), the application preserves state and stays on the game screen (instead of kicking the user back to config). The router should detect a **refresh on the `/toy` route with existing game state** and allow the route to remain accessible [7] [8]. This prevents losing progress when refreshing or reopening the app on the toy screen.
- **Default and Special Routes:** Define the default route (`"/"`) to show the configuration screen, and an admin route (`"/admin"`) that always shows the configuration screen with admin options (bypassing any "skip config" setting). The admin route ensures administrators or advanced users can access config even if the app is set to skip it.

The routing system will be integrated into the broader front-end stack of ToddleToy. The application is built with **ES6 modules** and bundled by **Vite 5** (a modern build tool) [9]. Aside from the game engine (**Phaser 3.70.0** [10] for rendering the interactive toy), no front-end framework is used. Styling is handled via standard CSS (inline styles and dynamic style injection in JavaScript), and application state (configuration, game state) is managed through custom classes and persisted to `localStorage` [11]. The app is a **Progressive Web App (PWA)** with a service worker and manifest for offline use; users can be prompted to install it on their device [12]. Quality assurance is maintained through **unit tests** (Jest) and **end-to-end tests** (Playwright) [9]. The following sections detail the design of the router and its integration, including pseudocode for implementation and testing scenarios.

Router Class Implementation

The routing functionality is encapsulated in a `Router` class (defined in `src/routes/Router.js`). This class manages the collection of routes and controls navigation. Below is a breakdown of the `Router` class, its properties, and methods, described in pseudocode form for clarity.

Router Properties and Constructor

The `Router` class constructor initializes routing state and sets up event listeners for navigation events. Key properties include:

- `routes`: A Map storing route path strings to their handler functions.
- `currentRoute`: The path of the route that is currently active (e.g. `'/' or '/toy'`).
- `previousRoute`: The path of the last route that was active (used for reference or debugging).
- `defaultRoute`: The default route path (set to `'/' for the config screen by default).
- `allowDirectToyAccess`: A boolean flag indicating if navigating to the `'/toy'` route is currently permitted. This starts as `false` (so direct access to the toy is blocked) and is set to `true` only when the user goes through the proper flow (configuration) or in certain refresh scenarios.
- `isRefreshFromToy`: A boolean used to detect a page refresh on the toy route. If `true`, it signals that the user reloaded the game page (and had game state saved), so the router should preserve access to `/toy` on load.

Constructor Behavior (Pseudocode):

```
class Router:
    constructor():
        this.routes = new Map()
        this.currentRoute = null
        this.previousRoute = null
        this.defaultRoute = '/'
        this.allowDirectToyAccess = false

        // Determine if this is a refresh on the toy route
        this.isRefreshFromToy = false
        if window.location.pathname == '/toy':
            if localStorage.getItem('toddleToyGameState') exists AND
localStorage.getItem('toddleToyConfig') exists:
                // Both game state and config in local storage implies user was
playing and refreshed
                console.log("🔍 Refresh from toy detected")
                this.isRefreshFromToy = true

        // Listen for browser navigation events (back/forward)
        window.addEventListener('popstate', (event) => {
            // When history changes, load the appropriate route
            this.handleRouteChange(window.location.pathname)
```

```
        })

        // Note: We do NOT call init() here. Route definitions will be added
first,
        // then init() should be called to process the initial URL.
```

In this pseudocode, the constructor sets up the Map and initial flags, checks the current URL for a `/toy` refresh scenario, and attaches a `popstate` event listener so that clicking the back/forward buttons will trigger `handleRouteChange` with the correct path. The actual initialization of the route (rendering the current view) is deferred until after routes are registered (via a separate `init()` call).

Route Registration (`addRoute`)

Routes are registered by providing a path and a handler (callback function). The handler encapsulates what should happen when the route is navigated to (such as showing or hiding certain UI components).

`addRoute(path, handler)` **Pseudocode:**

```
function addRoute(path, handler):
    // Store the handler function for the given route path
    routes.set(path, handler)
```

This method simply adds an entry to the `routes` Map linking a URL path to a handler function. For example, `router.addRoute('/toy', someFunction)` will ensure that when navigation to `'/toy'` occurs, `someFunction` is called to update the UI.

Initialization (`init`)

After all routes have been registered, the `router.init()` method is called to handle the initial page load or refresh. It determines what the current URL path is and triggers the appropriate route logic.

`init()` **Pseudocode:**

```
function init():
    // Determine the starting route (use defaultRoute if none or root)
    let startPath = window.location.pathname
    if (!startPath or startPath == ''):
        startPath = this.defaultRoute  // default to '/' if pathname is empty

    // Process the initial route
    this.handleRouteChange(startPath)
```

This will cause the router to immediately handle the route corresponding to the current URL. For instance, if the app is loaded at `"/"` (the base URL), it will load the configuration screen. If loaded at `"/toy"` (e.g. user refreshed on the game screen), it will attempt to handle the toy route – and, due to the refresh detection earlier, allow access if appropriate.

Navigation Methods (`navigate` and `replace`)

The router provides two methods to programmatically change routes: - `navigate(path)` – push a new entry onto the history stack and go to the route. Use this for normal in-app navigation (e.g. when a user clicks a "Start Playing" button to go from config to toy).
- `replace(path)` – replace the current history entry and go to the route. Use this for redirects or immediate route changes that shouldn't create a back-button history entry (e.g. redirecting an unauthorized access attempt to the default page).

`navigate(path)` **Pseudocode:**

```
function navigate(path):
    if path != this.currentRoute:
        // Push new state into browser history
        history.pushState({}, '', path)
        // Handle the route change (load the new route)
        this.handleRouteChange(path)
```

This checks if the new path is different from the current route to avoid redundant navigation. If it is a new route, it updates the browser's address bar and history via `pushState` and then invokes the internal route handling logic to update the UI.

`replace(path)` **Pseudocode:**

```
function replace(path):
    // Replace current history entry (no new entry in back/forward stack)
    history.replaceState({}, '', path)
    // Immediately handle the route change
    this.handleRouteChange(path)
```

This method always triggers a route change (even if the path is the same) by replacing the URL. It's useful for redirect scenarios (like sending a user back to `'/'` if they try to access a protected route without permission).

Core Logic: Handling Route Changes

The heart of the router is the `handleRouteChange(path)` method. This function is responsible for executing the correct route handler when a route is navigated to (either via `navigate()`, `replace()`, or

browser navigation events). It also enforces route guards and manages state transitions such as entering or leaving the protected toy route.

`handleRouteChange(path)` **Pseudocode:**

```
function handleRouteChange(path):
    // Keep track of previous and current routes
    this.previousRoute = this.currentRoute
    this.currentRoute = path

    // Route Guard & Toy Access Logic:
    if (path == '/toy'):
        if (this.isRefreshFromToy):
            // Special case: user refreshed on /toy and had prior state
            console.log("  Refresh on /toy detected, preserving access")
            this.allowDirectToyAccess = true   // allow access without config
since it's a refresh
            this.isRefreshFromToy = false      // reset the refresh flag after
using it
        // Note: If path == '/toy' and it's not a refresh, we do NOT immediately
reset allowDirectToyAccess.
        // We want to keep whatever value it currently has (it might be true if
user just came from config).
        // We avoid resetting it here to not accidentally block a legitimate
navigation from config.

    else:
        // If navigating to any route that is not '/toy', revoke toy access
permission.
        // This prevents users from using back button to return to toy without
re-authorizing (unless it's a refresh scenario).
        this.resetToyAccess()  // sets allowDirectToyAccess = false

    // Find the route handler for the given path
    let handler = routes.get(path)
    if (handler exists):
        try:
            // Execute the route's handler function to render that route
            handler()
        catch (error):
            console.error(`Error handling route ${path}:`, error)
            this.handleNotFound()  // handle exceptions by redirecting to
default
    else:
        console.log(`No route handler found for ${path}`)
        if (path != this.defaultRoute):
            // Unknown path: redirect to default route (typically '/')
```

```
            console.warn(`Route not found: ${path}, redirecting to default`)
            this.replace(this.defaultRoute)
        else:
            // Already at default route but no handler (edge case)
            this.handleNotFound()
```

Explanation:
- When a route change is triggered, the router updates its `previousRoute` and `currentRoute` trackers.
- If the new route is `'/toy'` (entering the protected game screen), special logic applies. If `isRefreshFromToy` was set to true in the constructor (meaning the app loaded on `/toy` with saved state), the router logs this and explicitly allows toy access (`allowDirectToyAccess = true`) so that the user isn't kicked out. Then it resets the `isRefreshFromToy` flag for future navigations. Importantly, if `'/toy'` is reached through normal navigation (not a refresh), the router **does not reset** the `allowDirectToyAccess` flag here. This ensures that if the user came from the config screen (which will have set the flag to true), they remain authorized to view the toy.
- If the new route is anything other than `/toy`, the router calls `resetToyAccess()` to revoke any prior toy access. This is a security measure; whenever the user leaves the toy screen (e.g., going back to config or to any other route), direct toy access should no longer be considered valid until they explicitly go through config again.
- After handling the toy access logic, the router looks up the corresponding handler in the `routes` Map. If found, it invokes the handler inside a try/catch. If the handler throws an error (unexpected), a fallback `handleNotFound()` routine is called to redirect to a safe page (default route).
- If no handler is found for the given path (meaning an undefined route was requested), the router will also invoke a fallback. If the current path is not the default, it will **redirect to the default route** (via `replace` to avoid a new history entry) [13] [14]. If somehow the current path is already the default and there's no handler (which shouldn't happen since default `'/'` should be defined), it calls `handleNotFound()` to log a warning.

`handleNotFound()`: In this design, `handleNotFound` simply logs a warning and redirects to the default route if not already there. In pseudocode:

```
function handleNotFound():
    warn(`Route not found: ${this.currentRoute}, redirecting to default`)
    if (this.currentRoute != this.defaultRoute):
        this.replace(this.defaultRoute)
    // If already on default, do nothing (or could display an error message)
```

This ensures the user is always taken to a valid screen. In practice, since we define a handler for `'/'`, the user will end up on the config screen for any unknown route.

Utility and State Management Methods

The Router class includes a few simple utility methods to manage its state and enforce the route guard:

- `allowToyAccess()` – Sets the internal flag to allow navigating to the toy route. This should be called when the user has completed the required steps (configuration) to proceed to the toy.

```
function allowToyAccess():
    this.allowDirectToyAccess = true
```

- `isToyAccessAllowed()` – Returns the status of the toy access flag (true/false). Route handlers use this to decide if the toy route can be shown or if the user should be redirected.

```
function isToyAccessAllowed():
    return this.allowDirectToyAccess
```

- `resetToyAccess()` – Resets the toy access flag to false, revoking permission. This is typically called automatically by the router when leaving the toy route.

```
function resetToyAccess():
    this.allowDirectToyAccess = false
```

- `getCurrentRoute()` – Returns the current route path string. Useful if other parts of the app need to know which view is active.

```
function getCurrentRoute():
    return this.currentRoute
```

- `isCurrentRoute(path)` – Convenience check to compare a given path with the current route. Returns true if they match, false otherwise.

```
function isCurrentRoute(path):
    return (this.currentRoute == path)
```

These methods support the main routing logic by exposing the router state and controlling the guard flag.

Route Definitions and Screen Navigation

Route definitions are managed in a separate module (e.g. `src/routes/routes.js`) which sets up the specific routes for the application using the Router. The `AppRoutes` class (or similar) is responsible for creating a `Router` instance, registering all needed routes, and linking them to the functions that control the UI for each route. After defining routes, it calls `router.init()` to start the router.

Below is a specification of the routes and their handler behavior in pseudocode:

Initialization of AppRoutes and Router Setup:

```
class AppRoutes:
    constructor():
        this.router = new Router()
        this.configManager = new ConfigManager()
        this.configScreen = null
        this.game = null
        this.currentScreen = null

        this.setupRoutes()
        this.router.init()   // start routing on the current URL after setup
```

Route Handlers Specification

Within `AppRoutes.setupRoutes()`, each route is registered with `router.addRoute(path, handlerFunction)`. We have three primary routes in this application:

1. **Default Route (`"/"` – Configuration Screen):**
   Displays the configuration UI where the user selects content and settings. This is the home screen of the app.
   Handler behavior:

   ```
   router.addRoute('/', () => {
       // Default route handler: Show the configuration screen.
       showConfigScreen()
   })
   ```

   When the `'/'` route is triggered, we call `showConfigScreen()`. This function will ensure the config UI is visible (and create it if it hasn't been created yet). It also handles a special case: if the user had previously chosen to skip the config screen on startup, and this navigation isn't forced, the app might immediately redirect to `/toy` instead (see **Skip Config logic** below).

2. **Protected Toy Route (`"/toy"` – Main Game Screen):**
   This route should present the main interactive toy/game. It is **protected**, meaning the user normally cannot access it unless they have completed the configuration or have a saved configuration ready.
   Handler behavior (with guard):

   ```
   router.addRoute('/toy', () => {
       // Only allow if user came through config or refresh
       if (!router.isToyAccessAllowed()):
   ```

```
        console.log("Direct /toy access denied. Redirecting to
'/' (config)")
        router.replace('/')    // Redirect to default (config) if accessed
improperly
        return

    // User is allowed (flag was set), proceed to show the toy/game screen
    showToyScreen()
})
```

When `/toy` is requested, the handler first **checks the router's flag** for toy access. If `isToyAccessAllowed()` returns false, it means the user did not go through the config screen in this session (or at least the app hasn't granted permission). In that case, the handler immediately uses `router.replace('/')` to send the user back to the config screen, and returns without loading the game. If the check passes (true), the handler calls `showToyScreen()` to initialize or reveal the game. The `showToyScreen` function will create a new game instance if needed and update the UI accordingly. It may also double-check that a config exists in `localStorage` as a safety net, and if not, it will also redirect to config [15]. This double-check ensures that even if a flag was somehow improperly set, an actual missing configuration will still cause a redirect to protect the app's stability.

State Preservation on Refresh: If the user refreshes the page while on `/toy` with a saved state, the router's constructor will have set `isRefreshFromToy = true` (as described earlier). This causes `router.allowToyAccess()` effectively to be called during route handling (in `handleRouteChange`), meaning `isToyAccessAllowed()` will return true in the handler. Thus, even on a direct page reload of `/toy`, the guard check passes (assuming a valid config was in storage) and `showToyScreen()` executes. The game can then restore state (e.g., reloading the last shown objects) from `localStorage` after load.

1. **Admin Route (`"/admin"` – Admin Configuration):**
   This route also shows the configuration screen, but is intended for admin or advanced use. When accessed, it will **bypass the "skip config" setting** and also enable any admin-specific features (for example, showing a PWA install prompt or additional debug options).
   Handler behavior:

   ```
   router.addRoute('/admin', () => {
       // Always show the config screen, ignoring any skip preferences.
       showConfigScreen(forceShow = true, isAdmin = true)
   })
   ```

   The admin handler calls `showConfigScreen` with parameters indicating that the config should be shown even if the user normally skips it (`forceShow=true`), and that this is an admin context (`isAdmin=true`). Inside `showConfigScreen`, these flags ensure that it does not redirect to `/toy` even if skip is enabled, and it might trigger admin-specific UI changes. For instance, in admin mode the application could display a message or prompt to install the app as a PWA [12].

Skip Config Logic:
If the user has previously opted to "⚡ Skip this screen next time" on the config screen, the application should normally bypass the config on startup. The `showConfigScreen` logic (for the `'/'` route) checks the skip setting via `configManager.shouldSkipConfig()`. In pseudocode, the relevant part is:

```
function showConfigScreen(forceShow=false, isAdmin=false):
    if (!forceShow AND configManager.shouldSkipConfig() == true):
        // User wants to skip config on normal startup
        if (router.getCurrentRoute() != '/toy'):
            console.log("Skipping config, navigating directly to /toy")
            // Before redirecting, allow toy access since user has a saved
config
            router.allowToyAccess()  // ensure guard will allow entry
            router.replace('/toy')   // go directly to toy route
        else:
            console.log("Already on /toy, no need to redirect")
        return  // exit, so config screen is not shown now
    }
    // ... otherwise, proceed to display config UI ...
```

The above pseudocode includes an **important implementation detail**: if skip is enabled and we're not forcing the config to show, we grant toy access (`router.allowToyAccess()`) **before** navigating to `/toy`. This adaptation ensures that the route guard will not immediately kick the user back to `/` in an endless loop. Essentially, if the user chose to skip config, we trust that a valid configuration is stored and pre-authorize the toy route on startup. The actual code should implement this logic (setting the flag then navigating), as the current implementation may need that adjustment to avoid redirect loops. Once on the toy screen, the app remains functional because a saved config is present in local storage and the game can load with those settings.

Screen Transition Functions (`showConfigScreen` and `showToyScreen`)

For completeness, here is a brief overview of what the screen management functions do:

- `showConfigScreen(forceShow=false, isAdmin=false)`**:** Responsible for displaying the configuration UI. It checks the skip logic as described, hides any currently active screen, and then either creates a new `ConfigScreen` instance or shows an existing one. It also sets `currentScreen` state and updates the page title (e.g., to "ToddleToy - Configure" or "ToddleToy - Admin Configuration"). If `isAdmin` is true, it might also trigger admin-specific notifications (like prompting the PWA installation).
- `showToyScreen()`**:** Responsible for switching to the game view. It first verifies that a configuration exists (`toddleToyConfig` is in localStorage) as a safety check [16]. If not, it redirects to config (this is a fallback in case someone somehow bypassed the intended flow). If the config is present, it hides the config screen (if one was showing), then either creates a new `ToddlerToyGame` instance or resets the existing game state for a fresh start [17]. It marks the current screen as 'toy' and updates the page title (e.g., "ToddleToy - Interactive Learning"). The actual game content is handled by the Phaser-powered `ToddlerToyGame` class, which is beyond the scope of this router specification.

Both of these functions (`showConfigScreen` and `showToyScreen`) are called by the router's route handlers defined above. They ensure a clear separation between the routing logic (which decides *what* should be shown) and the UI logic (which actually manipulates the DOM and game state to show it).

Frontend Tech Stack Context

(This section provides context about the environment and tools, which can influence how the router is implemented and tested.)

- **Framework-Free Implementation:** The entire front-end is built without frameworks – all DOM manipulation and event handling is done with plain JavaScript (ES6). The router and UI components are custom classes (e.g., `Router`, `ConfigScreen`, etc.), and modules are bundled using Vite (v5) [9]. The only major external library included is **Phaser 3.70.0** [10], used for the game/interactive portion of the toy.
- **Styling:** Styling is achieved via standard CSS. Some styles are defined inline or injected via JavaScript (for example, the config screen dynamically injects a `<style>` tag for certain animations). There is no CSS framework; styles are managed as needed in components.
- **State Management:** Application state such as configuration settings and game state is handled by custom classes (`ConfigManager` for config) and stored in `localStorage` for persistence. For example, when the user clicks "Start Playing", the config is saved to `localStorage` and later retrieved to configure the game. The router interacts with this by checking for `localStorage` entries to make decisions (like refresh detection and `showToyScreen` safety checks).
- **Progressive Web App (PWA):** ToddleToy is set up as a PWA with a web manifest and a service worker for offline support. In admin mode, the app nudges the user to install the app via an "Install as App" prompt [12]. This does not directly affect routing, but it means the app can be launched in standalone mode and URLs still need to route correctly even offline. The router's design using History API works offline since it doesn't require server round-trips for navigation.
- **Testing:** The project uses **Jest** for unit tests and **Playwright** for end-to-end/browser tests [9]. The routing system is tested to ensure it behaves as expected (e.g., the router correctly allows or denies access to routes, updates state on navigation, etc.). Tests are also in place for configuration skipping, refresh behavior, and integration between the router and the config/game components.

Testing Plan and Scenarios

To guarantee that the routing framework works as intended, the following key scenarios should be covered by tests. The tests can be implemented in Jest (for unit tests, simulating the Router behavior and using JSDOM for DOM interactions) and in Playwright (for full integration tests, where the app is run in a headless browser and user actions are simulated). Below we outline important test cases and their expected outcomes:

1. **Default Route Initialization:**
   Scenario: Launch the app at the base URL ("/").
   Steps: Construct the `AppRoutes` (which in turn constructs the Router and calls `init()`).
   Expected: The router should interpret the current path as `'/'` and automatically call the handler for `'/'`. The result should be that `showConfigScreen()` is invoked and the config screen is displayed. The router's `currentRoute` should be `'/'` after init.

Verification: The Router's `getCurrentRoute()` returns `'/'` [18], and any state indicating the config screen is visible can be checked (e.g., a DOM element with id `"config-screen"` becomes visible, though that might be checked in integration tests).

2. **Route Registration and Navigation:**
   Scenario: Add a new test route and navigate to it (unit test of Router functionality).
   Steps: Use `router.addRoute('/test', handlerFn)` with a mock function, then call `router.navigate('/test')`.
   Expected: The mock handler function should be called once as a result of navigation [19]. The router should update its `currentRoute` to `'/test'` and reflect that in `getCurrentRoute()`. Additionally, `window.history.pushState` should have been called with the new path (this can be verified by spying on `pushState` in a test environment).

3. **Replace State Navigation:**
   Scenario: Use `router.replace()` to redirect to a route.
   Steps: Register a route (e.g., `'/replace-test'`) with a mock handler. Call `router.replace('/replace-test')`.
   Expected: The handler should be invoked (meaning the route was handled) [20]. The router should set `currentRoute` to `'/replace-test'`, and the browser history should have replaced the current entry (verifiable via spy on `history.replaceState`). Importantly, if a back navigation is triggered after this, it should skip the replaced route as expected.

4. **Prevent Direct Toy Access (Guard):**
   Scenario: Simulate a user manually navigating to the `'/toy'` route without going through config first.
   Steps: Ensure `router.allowDirectToyAccess` is `false` (default). Call `router.handleRouteChange('/toy')` or simulate a fresh page load at `/toy` by setting `window.location.pathname='/toy'` before `router.init()`.
   Expected: The router should detect that `isToyAccessAllowed()` is false in the `'/toy'` handler and immediately redirect to `'/'` [21]. In a unit test, you can spy on `router.replace` to confirm it's called with `'/'`. After this operation, `router.getCurrentRoute()` should end up as `'/'` (the default route). No game screen should be shown. In integration testing, navigating to the `/toy` URL should result in the user seeing the config screen (and the URL might change to '/' due to the replace state).

5. **Allow Toy Access via Config Flow:**
   Scenario: User goes through the normal flow: opens app at `'/'`, configures settings, then clicks "Start Playing".
   Steps: In a test, simulate calling `configScreen.startPlaying()` which should internally call `router.allowToyAccess()` and then `router.navigate('/toy')` [22].
   Expected: After `allowToyAccess()`, the `isToyAccessAllowed()` flag returns true. The subsequent navigation to `/toy` should find that flag true and thus run `showToyScreen()` without redirecting. The game screen becomes active, and `router.currentRoute` is `'/toy'`. Any stored config remains in localStorage for this session. We verify that no redirect occurred (i.e., `router.replace('/')` was not called during this process) and that the toy screen's elements are present.

6. **Toy Access Reset on Leaving:**
   Scenario: After entering the toy, the user navigates back to the config screen (e.g., by hitting the back button or via the app UI).
   Steps: Starting from the toy route with access allowed, simulate `router.navigate('/')` or a `window.history.back()` triggering `popstate`.
   Expected: When leaving `/toy`, the router should call `resetToyAccess()` internally [23]. We can verify that `router.isToyAccessAllowed()` becomes false once the route has changed to `'/'`. This ensures that if the user tries to use the forward button to go to `/toy` again (without going through config again), the guard will treat it as a direct access and prevent it. In a UI test, one could navigate to toy, then back to config, then attempt to go forward to toy and expect to be redirected to config again (or simply that the forward navigation doesn't show toy unless config was done again).

7. **Refresh on Toy Route (State Preservation):**
   Scenario: The user hits refresh while on the toy screen (with a game in progress). There is a saved game state and config in localStorage from the ongoing session.
   Steps: Simulate a page reload at `'/toy'`. In a unit test, you can set `window.location.pathname = '/toy'` and populate `localStorage` with a dummy `toddleToyConfig` and `toddleToyGameState` (to mimic a session with state) before constructing the Router. Then call `router.init()`.
   Expected: The router constructor should detect the refresh scenario (since pathname is '/toy' and state exists) and set `isRefreshFromToy = true`. During `init()`, when `handleRouteChange('/toy')` runs, the router should preserve access by setting `allowDirectToyAccess = true` [8]. The route handler then sees access allowed and calls `showToyScreen()` as normal. The result is that after refresh, the app remains on the toy screen (the user does not get redirected to config). In a test, one can verify that `router.isToyAccessAllowed()` ends up true after init in this scenario, and possibly check that the game state was loaded (e.g., objects count in the game state remains the same). The unit test provided in the codebase indicated the intended behavior is to maintain toy access on refresh and to restore game objects from localStorage [24] [25].

8. **Skip Config Setting:**
   Scenario: The user enabled "skip config next time". Now the app is loaded fresh (not using the admin route).
   Steps: Simulate that `configManager.shouldSkipConfig()` returns true (perhaps by stubbing it or setting the underlying flag in localStorage). Load the app at `'/'` and call `router.init()` (which triggers the default route handler).
   Expected: The `showConfigScreen` logic should detect skip = true and automatically redirect to `/toy`. Ideally, it should also call `router.allowToyAccess()` before doing so, as noted. In the test, check that after initialization, the current route becomes `'/toy'` (meaning it navigated straight to toy without user intervention). The config screen should not be visible. If the implementation is corrected per spec, no redirect loop occurs and the toy screen is shown. (If the implementation lacked the allowToyAccess call, one might observe a loop or an immediate redirect back to `'/'`; part of testing is to catch such issues. This spec calls for adding the allowToyAccess step to prevent that loop.)

9. **Admin Route Bypass:**
   Scenario: The app is launched or navigated to `"/admin"`.

Steps: Call `router.handleRouteChange('/admin')` or simulate a user typing `.../admin` in the URL and then `router.init()`.
Expected: The admin route handler should invoke `showConfigScreen(forceShow=true, isAdmin=true)`. The result is that the config screen appears even if skip was enabled. In admin mode, one might verify that certain admin-only elements or messages are present (for example, the PWA install prompt banner should appear when `showConfigScreen` is called with `isAdmin=true` [26]). The `currentRoute` should be `'/admin'` (though the app might still visually be showing the config screen, it might differentiate admin mode by title or other UI changes). After this, if the user starts playing from admin, the flow should be similar: `router.allowToyAccess()` (likely called in the normal start playing process) and navigate to `/toy`.

Each of these scenarios helps ensure the router behaves correctly. Tests should cover both the internal logic (unit tests spying on methods and flags) and the end-to-end behavior (making sure the user sees the right screen and the URL updates appropriately). The programmer implementing according to this specification should use these test cases as a guide for writing automated tests and for manual verification.

Conclusion

By following this specification, the programmer will implement a robust client-side routing system tailored for the ToddleToy application. The router will provide smooth transitions between the configuration and game screens, enforce that the game is only accessed through the intended flow (unless explicitly skipped or on refresh), and integrate cleanly with browser navigation controls. The result will be a seamless user experience in a single-page application style, without the overhead of external libraries, and full control over the route logic to accommodate special cases like admin access and PWA behavior.

---

[1] [2] [3] [4] [5] [7] [8] [13] [14] [23] Router.js
https://github.com/truevox/toddletoy/blob/8a3123397b82b6c34140a16225ae846b81d0b1fc/src/routes/Router.js

[6] [15] [16] [17] [21] [26] routes.js
https://github.com/truevox/toddletoy/blob/8a3123397b82b6c34140a16225ae846b81d0b1fc/src/routes/routes.js

[9] [10] package.json
https://github.com/truevox/toddletoy/blob/8a3123397b82b6c34140a16225ae846b81d0b1fc/package.json

[11] [12] [22] ConfigScreen.js
https://github.com/truevox/toddletoy/blob/8a3123397b82b6c34140a16225ae846b81d0b1fc/src/config/ConfigScreen.js

[18] [19] [20] config-system.test.js
https://github.com/truevox/toddletoy/blob/8a3123397b82b6c34140a16225ae846b81d0b1fc/tests/config-system.test.js

[24] [25] refresh-state-preservation.test.js
https://github.com/truevox/toddletoy/blob/8a3123397b82b6c34140a16225ae846b81d0b1fc/tests/unit/refresh-state-preservation.test.js