**Room-Driven Multi-Chain Architecture Specification**

**Executive Summary**

This document defines the comprehensive architecture for multi-chain wallet integration in the quiz application. The system is **room-driven**, meaning chain selection and wallet context are determined by room configuration rather than global app state. This approach supports the core user journey: hosts select a blockchain during room creation, and all subsequent interactions (dashboard management, asset uploads, prize distribution, player joining) automatically use the appropriate chain based on the room's configuration.

**1. Architectural Principles**

**1.1 Room-Centric Chain Determination**

**Core Principle**: The room ID is the source of truth for blockchain context throughout the application.

**Implementation**:

* Chain selection occurs during room setup and is stored in room configuration
* All components derive their chain context from room configuration, not global state
* Chain information flows through the config persistence layer (Zustand stores and socket events)

**Why This Approach**:

* Multiple users can create rooms on different chains simultaneously
* No global chain state conflicts between different room contexts
* Natural isolation between different blockchain environments
* Aligns with quiz application's room-based architecture

**1.2 Store-Synchronized Provider Architecture**

**Core Principle**: Multiple DynamicChainProvider instances sync wallet state to a shared Zustand store.

**Implementation**:

* Each provider instance manages chain-specific wallet connections
* All providers sync their state changes to useWalletStore
* Components read wallet state from store, not directly from providers
* Store acts as persistence layer between navigation and provider instances

**Why Multiple Providers Are Necessary**:

* Chain detection happens at different times in different flows
* Setup wizard: User selects chain explicitly
* Dashboard: Chain determined from room config after socket connection
* Player join: Chain determined from room verification API response

**1.3 Chain-Agnostic Component Design**

**Core Principle**: Components contain zero hardcoded chain-specific logic.

**Implementation**:

* All components use generic hooks (useQuizChainIntegration, useWalletActions, useContractActions)
* No direct imports of chain-specific providers or hooks in components
* Chain-specific logic encapsulated in hook implementations
* Components work identically regardless of which chain is active

**Benefits**:

* Adding new chains requires no component modifications
* Consistent user experience across all supported blockchains
* Simplified testing and maintenance
* Clear separation of concerns

**2. Data Flow Architecture**

**2.1 Room Configuration Persistence**

The chain information flows through multiple persistence layers throughout the application lifecycle:

**Setup Phase (Room Creation)**

// User selects chain in wizard

setupConfig.web3Chain = 'stellar'

// Room creation stores complete config

const roomData = {

config: {

...setupConfig,

web3Chain: 'stellar',

// ... other properties

},

roomId: generatedRoomId,

hostId: generatedHostId

}

// API stores room with config

POST /quiz/api/create-web3-room { roomData }

**Dashboard Phase (Room Management)**

// Dashboard navigates to /host-dashboard/{roomId}

// Socket connects and emits room state

socket.emit('request\_current\_state', { roomId })

// Backend responds with full room config

socket.on('room\_config', (config) => {

// useQuizConfig store populated

setFullConfig(config)

// config.web3Chain now available to components

})

// DynamicChainProvider reads from config

const selectedChain = config?.web3Chain

**Player Join Phase (Runtime Detection)**

// Player enters room ID

// Room verification API call

const response = await fetch(`/quiz/api/verify-room/${roomId}`)

const roomConfig = await response.json()

// Chain detected from room config

const detectedChain = roomConfig.web3Chain

// JoinRoomFlow sets provider chain

<DynamicChainProvider selectedChain={detectedChain}>

**2.2 State Synchronization Flow**

The wallet state synchronization follows this pattern across all provider instances:

// In any chain provider (e.g., StellarWalletProvider)

useEffect(() => {

// Sync local wallet state to global store

useWalletStore.getState().updateStellarWallet({

address: wallet.address,

isConnected: wallet.isConnected,

// ... other properties

})

// Update active chain when connected

if (wallet.isConnected) {

useWalletStore.getState().setActiveChain('stellar')

}

}, [wallet.address, wallet.isConnected, /\* other dependencies \*/])

All components read from the store:

// Components never read from providers directly

const { stellar, activeChain } = useWalletStore()

const currentWallet = activeChain === 'stellar' ? stellar : null

**2.3 Configuration Timing Dependencies**

The architecture handles asynchronous configuration loading:

// Dashboard component lifecycle

1. Component mounts → DynamicChainProvider initializes

2. selectedChain = null initially (config not loaded)

3. Socket connects → room\_config event received

4. useQuizConfig updated → config.web3Chain available

5. DynamicChainProvider re-evaluates → Correct provider loads

6. Provider syncs wallet state to store

This timing pattern is consistent across all room-context components.

**3. Component Architecture**

**3.1 Provider Strategy by Context**

**Context 1: Room Setup (Web3Wizard)**

**Provider Placement**: None required **Chain Source**: setupConfig.web3Chain (user selection) **State Management**: Components read from store, no provider wrapper needed **Rationale**: Chain is known from user selection, wallet state persists in store

**Context 2: Room Management (HostDashboard)**

**Provider Placement**: Conditional wrapping for Web3 rooms **Chain Source**: config.web3Chain (from socket-loaded room config) **State Management**: Provider syncs to store, components read from store **Rationale**: Chain determined after config loads, provider ensures wallet functionality

**Context 3: Player Join (JoinRoomFlow)**

**Provider Placement**: Always wrapped **Chain Source**: Room verification API response **State Management**: Provider syncs to store for duration of join flow **Rationale**: Chain unknown until API response, temporary provider for join process

**Context 4: Game Controls (HostControlsPage)**

**Provider Placement**: None required (relies on store state) **Chain Source**: Inherited from dashboard context via store **State Management**: Components read from store **Rationale**: Wallet already connected in dashboard context, state persists via store

**3.2 Hook Architecture Hierarchy**

The hook system provides progressive abstraction layers:

**Layer 1: Chain-Specific Hooks (Provider Layer)**

* useStellarWallet() - Direct Stellar wallet management
* useStellarQuizContract() - Stellar contract interactions
* useEvmWallet() - Future EVM wallet management
* useSolanaWallet() - Future Solana wallet management

**Responsibility**: Direct blockchain integration, specific to each chain

**Layer 2: Generic Action Hooks (Abstraction Layer)**

* useWalletActions() - Chain-agnostic wallet operations (connect/disconnect)
* useContractActions() - Chain-agnostic contract operations (join/create/end rooms)

**Responsibility**: Provide generic interfaces, delegate to appropriate chain-specific hooks based on active chain

**Layer 3: Integration Hook (Component Interface)**

* useQuizChainIntegration() - Single hook for all component wallet needs

**Responsibility**: Bridge between quiz configuration and wallet state, provide chain-agnostic component interface

**3.3 Chain-Agnostic Component Pattern**

All components follow this pattern:

const MyComponent: FC = () => {

// Single integration hook provides all chain context

const {

selectedChain, // 'stellar' | 'evm' | 'solana' | null

isWalletConnected, // boolean

walletAddress, // string | null

needsWalletConnection,// boolean

walletReadiness, // { status, message, canProceed }

getChainDisplayName, // () => string

getFormattedAddress // () => string | null

} = useQuizChainIntegration()

// Generic action hooks (always called, handle chain internally)

const { connect, disconnect } = useWalletActions()

const { joinRoom, createPoolRoom, endRoom } = useContractActions()

// Component logic uses generic properties and functions

const handleConnect = () => connect() // No chain-specific code

const handleJoinRoom = () => joinRoom({ roomId, playerAddress: walletAddress })

// Render UI using generic properties

return (

<div>

<p>Chain: {getChainDisplayName()}</p>

<p>Status: {walletReadiness.message}</p>

{needsWalletConnection ? (

<button onClick={handleConnect}>Connect Wallet</button>

) : (

<p>Connected: {getFormattedAddress()}</p>

)}

</div>

)

}

**Key Characteristics**:

* No imports of chain-specific hooks or providers
* No conditional logic based on chain type
* All functionality accessed through generic interfaces
* Identical code pattern works for all chains

**4. State Management Architecture**

**4.1 Zustand Store Structure**

The wallet state is managed in a structured Zustand store:

interface WalletState {

// Chain-specific wallet states (separate branches for isolation)

stellar: StellarWalletConnection

evm: EvmWalletConnection

solana: SolanaWalletConnection

// Current active chain (determined by room context)

activeChain: SupportedChain | null

// Store actions

setActiveChain: (chain: SupportedChain | null) => void

updateStellarWallet: (updates: Partial<StellarWalletConnection>) => void

updateEvmWallet: (updates: Partial<EvmWalletConnection>) => void

updateSolanaWallet: (updates: Partial<SolanaWalletConnection>) => void

resetWallet: (chain: SupportedChain) => void

resetAllWallets: () => void

}

**4.2 Store Synchronization Pattern**

Each chain provider implements this synchronization pattern:

// Example: StellarWalletProvider synchronization

const StellarWalletProvider: FC = ({ children }) => {

const stellarWallet = useStellarWallet()

// Sync all wallet state changes to store

useEffect(() => {

useWalletStore.getState().updateStellarWallet({

address: stellarWallet.address,

isConnected: stellarWallet.isConnected,

isConnecting: stellarWallet.isConnecting,

isDisconnecting: stellarWallet.isDisconnecting,

error: stellarWallet.error,

balance: stellarWallet.balance,

publicKey: stellarWallet.publicKey,

networkPassphrase: stellarWallet.networkPassphrase,

})

// Update active chain when connection state changes

if (stellarWallet.isConnected) {

useWalletStore.getState().setActiveChain('stellar')

} else if (stellarWallet.error || !stellarWallet.isConnected) {

// Only clear active chain if no other chains are connected

const currentState = useWalletStore.getState()

const hasOtherConnection = currentState.evm.isConnected || currentState.solana.isConnected

if (!hasOtherConnection) {

useWalletStore.getState().setActiveChain(null)

}

}

}, [

stellarWallet.address,

stellarWallet.isConnected,

stellarWallet.isConnecting,

stellarWallet.isDisconnecting,

stellarWallet.error,

stellarWallet.balance,

stellarWallet.publicKey,

stellarWallet.networkPassphrase,

])

// Provider implementation continues...

}

**4.3 Store vs Configuration State Integration**

The wallet store integrates with existing configuration stores:

// useQuizChainIntegration bridges stores

const useQuizChainIntegration = () => {

// Configuration sources (room-based chain detection)

const { setupConfig } = useQuizSetupStore()

const { config } = useQuizConfig()

// Wallet state (synchronized from providers)

const { stellar, evm, solana, activeChain } = useWalletStore()

// Determine selected chain from configuration

const selectedChain = useMemo(() => {

// Priority order: active provider > dashboard config > setup config

if (activeChain) return activeChain

if (config?.web3Chain) return config.web3Chain

if (setupConfig?.web3Chain) return setupConfig.web3Chain

return null

}, [activeChain, config?.web3Chain, setupConfig?.web3Chain])

// Get current wallet based on selected chain

const currentWallet = useMemo(() => {

switch (selectedChain) {

case 'stellar': return stellar

case 'evm': return evm

case 'solana': return solana

default: return null

}

}, [selectedChain, stellar, evm, solana])

// Return unified state for components

return {

selectedChain,

currentWallet,

isWalletConnected: currentWallet?.isConnected ?? false,

// ... other derived state

}

}

**5. User Journey Specifications**

**5.1 Host Room Creation Journey**

**Step 1: Chain Selection** (StepWeb3QuizSetup)

* User selects blockchain from dropdown
* Chain stored in setupConfig.web3Chain
* No provider needed (selection only)

**Step 2: Configuration** (Subsequent wizard steps)

* Components read setupConfig.web3Chain via useQuizChainIntegration()
* Currency and prize configuration based on selected chain
* No wallet connection required during configuration

**Step 3: Review & Deploy** (StepWeb3ReviewLaunch)

* Component wrapped with DynamicChainProvider for deployment
* Provider loads based on setupConfig.web3Chain
* Wallet connection initiated for contract deployment
* Provider syncs connection state to store

**Step 4: Navigation to Dashboard**

* User navigates to /host-dashboard/{roomId}
* Wallet connection state persists in store
* Dashboard loads room config via socket
* DynamicChainProvider recognizes existing connection from store

**Step 5: Asset Management** (if applicable)

* Asset upload components use same wallet connection
* Contract interactions use persisted wallet state
* No reconnection required

**5.2 Host Dashboard Access Journey**

**Step 1: Direct Navigation**

* User navigates directly to /host-dashboard/{roomId} (e.g., from bookmark)
* No existing wallet state in store

**Step 2: Configuration Loading**

* Socket connects and loads room configuration
* config.web3Chain determined from room data
* DynamicChainProvider loads appropriate provider

**Step 3: Wallet Connection**

* If room requires Web3 functionality, wallet connection UI appears
* User connects wallet through provider
* Connection state synced to store

**Step 4: Persistent State**

* Further navigation within dashboard maintains wallet state
* Asset upload, prize distribution use same wallet connection
* No repeated connection prompts

**5.3 Player Room Join Journey**

**Step 1: Room ID Entry**

* Player enters room ID in join form
* Room verification API called

**Step 2: Chain Detection**

* API response includes room configuration with web3Chain
* JoinRoomFlow detects chain requirement
* DynamicChainProvider loads appropriate provider

**Step 3: Payment Processing**

* If room requires entry fee, wallet connection initiated
* Player connects wallet for payment
* Contract interaction processes payment

**Step 4: Game Entry**

* Player redirected to game interface
* Wallet state available if needed for in-game transactions
* Connection persists for game duration

**5.4 Prize Distribution Journey**

**Step 1: Game Completion**

* Host completes quiz in HostControlsPage
* Final leaderboard calculated with winners

**Step 2: Distribution Initiation**

* Host clicks "Distribute Prizes" button
* System uses existing wallet connection from dashboard context

**Step 3: Contract Execution**

* Prize distribution contract called using persisted wallet state
* Winners receive prizes automatically via smart contract
* Transaction hash provided for verification

**Step 4: Completion**

* Prize distribution status updated
* Room marked as completed
* Blockchain transaction links available

**6. Technical Implementation Specifications**

**6.1 DynamicChainProvider Implementation**

interface DynamicChainProviderProps {

selectedChain: SupportedChain | null

children: React.ReactNode

}

export const DynamicChainProvider: FC<DynamicChainProviderProps> = ({

selectedChain,

children

}) => {

const { setActiveChain } = useWalletStore()

// Update store when chain changes

useEffect(() => {

if (selectedChain) {

setActiveChain(selectedChain)

}

}, [selectedChain, setActiveChain])

// Provider selection logic

const ProviderComponent = useMemo(() => {

switch (selectedChain) {

case 'stellar': return StellarWalletProvider

case 'evm': return EvmWalletProvider

case 'solana': return SolanaWalletProvider

default: return null

}

}, [selectedChain])

// No provider mode (no wallet needed)

if (!selectedChain || !ProviderComponent) {

return <div data-testid="no-wallet-mode">{children}</div>

}

// Chain-specific provider mode

return (

<ErrorBoundary chain={selectedChain}>

<Suspense fallback={<ChainProviderSuspense chain={selectedChain} />}>

<ProviderComponent>

<div data-testid={`${selectedChain}-wallet-mode`}>

{children}

</div>

</ProviderComponent>

</Suspense>

</ErrorBoundary>

)

}

**6.2 Generic Action Hooks Implementation**

// useWalletActions.ts - Chain-agnostic wallet operations

export const useWalletActions = () => {

const { activeChain } = useWalletStore()

// Always call all hooks (no conditional calls)

const stellarWallet = useStellarWalletContext()

const evmWallet = useEvmWalletContext()

const solanaWallet = useSolanaWalletContext()

const connect = useCallback(async () => {

switch (activeChain) {

case 'stellar':

return stellarWallet.connect()

case 'evm':

return evmWallet.connect()

case 'solana':

return solanaWallet.connect()

default:

throw new Error('No chain selected')

}

}, [activeChain, stellarWallet, evmWallet, solanaWallet])

const disconnect = useCallback(async () => {

switch (activeChain) {

case 'stellar':

return stellarWallet.disconnect()

case 'evm':

return evmWallet.disconnect()

case 'solana':

return solanaWallet.disconnect()

default:

throw new Error('No chain selected')

}

}, [activeChain, stellarWallet, evmWallet, solanaWallet])

return {

connect,

disconnect,

isReady: !!activeChain,

}

}

// useContractActions.ts - Chain-agnostic contract operations

export const useContractActions = () => {

const { activeChain } = useWalletStore()

// Always call all contract hooks

const stellarContract = useStellarQuizContract()

const evmContract = useEvmQuizContract()

const solanaContract = useSolanaQuizContract()

const joinRoom = useCallback(async (params: JoinRoomParams) => {

switch (activeChain) {

case 'stellar':

return stellarContract.joinRoom(params)

case 'evm':

return evmContract.joinRoom(params)

case 'solana':

return solanaContract.joinRoom(params)

default:

throw new Error('No chain selected')

}

}, [activeChain, stellarContract, evmContract, solanaContract])

const createPoolRoom = useCallback(async (params: CreateRoomParams) => {

switch (activeChain) {

case 'stellar':

return stellarContract.createPoolRoom(params)

case 'evm':

return evmContract.createPoolRoom(params)

case 'solana':

return solanaContract.createPoolRoom(params)

default:

throw new Error('No chain selected')

}

}, [activeChain, stellarContract, evmContract, solanaContract])

const endRoom = useCallback(async (params: EndRoomParams) => {

switch (activeChain) {

case 'stellar':

return stellarContract.endRoom(params)

case 'evm':

return evmContract.endRoom(params)

case 'solana':

return solanaContract.endRoom(params)

default:

throw new Error('No chain selected')

}

}, [activeChain, stellarContract, evmContract, solanaContract])

return {

joinRoom,

createPoolRoom,

endRoom,

isReady: activeChain !== null,

}

}

**6.3 Chain Integration Hook Implementation**

// useQuizChainIntegration.ts - Component interface

export const useQuizChainIntegration = () => {

// Configuration sources

const { setupConfig } = useQuizSetupStore()

const { config } = useQuizConfig()

// Wallet state from store

const { stellar, evm, solana, activeChain } = useWalletStore()

// Determine selected chain with priority order

const selectedChain: SupportedChain | null = useMemo(() => {

// 1. Active provider chain (highest priority)

if (activeChain) return activeChain

// 2. Dashboard room config

const dashboardChain = config?.web3Chain || config?.web3ChainConfirmed

if (dashboardChain === 'stellar' || dashboardChain === 'evm' || dashboardChain === 'solana') {

return dashboardChain

}

// 3. Setup config (lowest priority)

const setupChain = setupConfig?.web3Chain

if (setupChain === 'stellar' || setupChain === 'evm' || setupChain === 'solana') {

return setupChain

}

return null

}, [activeChain, config?.web3Chain, config?.web3ChainConfirmed, setupConfig?.web3Chain])

// Get current wallet based on selected chain

const currentWallet = useMemo(() => {

switch (selectedChain) {

case 'stellar': return stellar

case 'evm': return evm

case 'solana': return solana

default: return null

}

}, [selectedChain, stellar, evm, solana])

// Derive wallet state

const isWalletConnected = currentWallet?.isConnected ?? false

const isWalletConnecting = currentWallet?.isConnecting ?? false

const walletError = currentWallet?.error ?? null

const walletAddress = currentWallet?.address ?? null

// Check if wallet is required for current configuration

const isWalletRequired = useMemo(() => {

if (!selectedChain) return false

// Check if room has entry fee or other payment requirements

const entryFee = config?.entryFee || setupConfig?.entryFee

return hasPositiveAmount(entryFee)

}, [selectedChain, config?.entryFee, setupConfig?.entryFee])

// Calculate wallet readiness status

const walletReadiness = useMemo(() => {

if (!isWalletRequired) {

return {

status: 'not-required' as const,

message: 'No wallet required',

canProceed: true

}

}

if (!selectedChain) {

return {

status: 'no-chain' as const,

message: 'No blockchain selected',

canProceed: false

}

}

if (isWalletConnecting) {

return {

status: 'connecting' as const,

message: `Connecting ${getChainDisplayName()} wallet...`,

canProceed: false

}

}

if (walletError) {

return {

status: 'error' as const,

message: walletError.message,

canProceed: false

}

}

if (!isWalletConnected) {

return {

status: 'disconnected' as const,

message: `${getChainDisplayName()} wallet not connected`,

canProceed: false

}

}

return {

status: 'ready' as const,

message: `${getChainDisplayName()} wallet ready`,

canProceed: true

}

}, [isWalletRequired, selectedChain, isWalletConnecting, walletError, isWalletConnected])

// Helper functions

const getChainDisplayName = useCallback((chain?: SupportedChain | null): string => {

const targetChain = chain ?? selectedChain

switch (targetChain) {

case 'stellar': return 'Stellar'

case 'evm': return 'Ethereum'

case 'solana': return 'Solana'

default: return 'No blockchain'

}

}, [selectedChain])

const getFormattedAddress = useCallback((short = true): string | null => {

if (!walletAddress) return null

if (!short) return walletAddress

if (walletAddress.length > 10) {

return `${walletAddress.slice(0, 6)}...${walletAddress.slice(-4)}`

}

return walletAddress

}, [walletAddress])

// Return unified interface for components

return {

// Chain information

selectedChain,

activeChain,

// Wallet state

currentWallet,

isWalletConnected,

isWalletConnecting,

walletError,

walletAddress,

// Quiz-specific logic

isWalletRequired,

walletReadiness,

needsWalletConnection: isWalletRequired && !isWalletConnected,

// Convenience flags

canProceed: walletReadiness.canProceed,

canProcessPayments: isWalletConnected && selectedChain !== null,

// Helper functions

getChainDisplayName,

getFormattedAddress,

// Debug information

debugInfo: {

configSource: activeChain ? 'active-provider' :

config?.web3Chain ? 'dashboard-config' :

setupConfig?.web3Chain ? 'setup-config' : 'none',

hasEntryFee: hasPositiveAmount(config?.entryFee || setupConfig?.entryFee),

}

}

}

**7. Security and Error Handling**

**7.1 Provider Error Boundaries**

Each DynamicChainProvider instance includes comprehensive error handling:

class ChainProviderErrorBoundary extends React.Component<

{ children: React.ReactNode; chain: SupportedChain | null },

{ hasError: boolean; error?: Error }

> {

constructor(props) {

super(props)

this.state = { hasError: false }

}

static getDerivedStateFromError(error: Error) {

return { hasError: true, error }

}

componentDidCatch(error: Error, errorInfo: React.ErrorInfo) {

console.error(`Failed to load ${this.props.chain} provider:`, error, errorInfo)

// Clear potentially corrupted wallet state

if (this.props.chain) {

useWalletStore.getState().resetWallet(this.props.chain)

}

}

render() {

if (this.state.hasError) {

return (

<div className="flex min-h-32 items-center justify-center">

<div className="space-y-2 text-center">

<div className="text-red-600">

⚠️ Failed to load {this.props.chain} wallet provider

</div>

<button

onClick={() => this.setState({ hasError: false })}

className="rounded bg-blue-600 px-4 py-2 text-white hover:bg-blue-700"

>

Retry

</button>

</div>

</div>

)

}

return this.props.children

}

}

**7.2 Wallet Connection Security**

All wallet connections implement security best practices:

// Connection validation in providers

const validateConnection = async (wallet: WalletConnection): Promise<boolean> => {

try {

// Verify wallet address format

if (!isValidAddress(wallet.address, wallet.chain)) {

throw new Error('Invalid wallet address format')

}

// Verify network consistency

if (wallet.chain === 'stellar' && !wallet.networkPassphrase) {

throw new Error('Stellar network passphrase required')

}

// Test connectivity

const balance = await wallet.getBalance()

if (balance === null) {

throw new Error('Unable to verify wallet connectivity')

}

return true

} catch (error) {

console.error('Wallet connection validation failed:', error)

return false

}

}

**7.3 Contract Interaction Security**

All contract interactions include security validations:

// Contract parameter validation

const validateContractParams = (params: ContractParams): void => {

// Validate room ID format

if (!params.roomId || !/^[a-zA-Z0-9\_-]+$/.test(params.roomId)) {

throw new Error('Invalid room ID format')

}

// Validate addresses

if (!isValidAddress(params.hostAddress, params.chain)) {

throw new Error('Invalid host address')

}

// Validate amounts

if (params.entryFee && (parseFloat(params.entryFee) < 0 || parseFloat(params.entryFee) > 1000000)) {

throw new Error('Entry fee out of acceptable range')

}

// Validate percentages

if (params.hostFeePct && (params.hostFeePct < 0 || params.hostFeePct > 5)) {

throw new Error('Host fee percentage must be between 0-5%')

}

}

**8. Performance Characteristics**

**8.1 Lazy Provider Loading**

The system implements efficient provider loading:

// Provider loading strategy

const providerLoadingStrategy = {

// Setup wizard: No provider loaded until deployment step

setupPhase: 'lazy-on-deployment',

// Dashboard: Provider loaded after room config determines chain

dashboardPhase: 'conditional-on-web3-room',

// Player join: Provider loaded immediately after chain detection

joinPhase: 'immediate-on-detection',

// Game controls: No provider, uses store state

gamePhase: 'store-only'

}

**8.2 State Synchronization Performance**

Provider-to-store synchronization is optimized:

// Optimized synchronization with debouncing

const useDebouncedWalletSync = (wallet: WalletState, delay = 100) => {

const debouncedUpdate = useMemo(

() => debounce((updates: Partial<WalletState>) => {

useWalletStore.getState().updateStellarWallet(updates)

}, delay),

[delay]

)

useEffect(() => {

debouncedUpdate({

address: wallet.address,

isConnected: wallet.isConnected,

// ... other properties

})

return () => debouncedUpdate.cancel()

}, [wallet.address, wallet.isConnected, debouncedUpdate])

}

**8.3 Memory Management**

The system implements proper cleanup strategies:

// Provider cleanup on unmount

useEffect(() => {

return () => {

// Clear provider-specific state on unmount

if (selectedChain) {

useWalletStore.getState().resetWallet(selectedChain)

}

}

}, [selectedChain])

// Store cleanup for completed rooms

const cleanupCompletedRooms = () => {

// Clear wallet state after prize distribution completes

const { resetAllWallets } = useWalletStore.getState()

// Allow time for final transactions to complete

setTimeout(() => {

resetAllWallets()

}, 5000)

}

**9. Testing Strategy**

**9.1 Component Testing Approach**

Components are tested in isolation using the store:

// Test setup with mock store

const createMockWalletStore = (overrides = {}) => ({

stellar: { isConnected: false, address: null, ...overrides.stellar },

evm: { isConnected: false, address: null, ...overrides.evm },

solana: { isConnected: false, address: null, ...overrides.solana },

activeChain: overrides.activeChain || null,

setActiveChain: jest.fn(),

updateStellarWallet: jest.fn(),

// ... other actions

})

// Component test example

describe('StepWeb3ReviewLaunch', () => {

it('should display wallet connection UI when wallet required', () => {

// Mock store state

useWalletStore.mockReturnValue(createMockWalletStore({

stellar: { isConnected: false, address: null },

activeChain: 'stellar'

}))

// Mock configuration

useQuizSetupStore.mockReturnValue({

setupConfig: {

web3Chain: 'stellar',

entryFee: '5.0',

// ... other config

}

})

render(<StepWeb3ReviewLaunch {...defaultProps} />)

expect(screen.getByText('Connect Stellar Wallet')).toBeInTheDocument()

})

it('should show connected state when wallet connected', () => {

useWalletStore.mockReturnValue(createMockWalletStore({

stellar: {

isConnected: true,

address: 'GTEST...WALLET'

},

activeChain: 'stellar'

}))

render(<StepWeb3ReviewLaunch {...defaultProps} />)

expect(screen.getByText('Stellar Wallet Connected')).toBeInTheDocument()

expect(screen.getByText('GTEST...LET')).toBeInTheDocument()

})

})

**9.2 Integration Testing Strategy**

Integration tests validate the complete user flows:

// End-to-end flow testing

describe('Room Creation to Dashboard Flow', () => {

it('should maintain wallet connection across navigation', async () => {

// 1. Setup wizard - select chain

const { user } = renderWithProviders(<Web3QuizWizard />)

await user.selectOptions(screen.getByLabelText('Blockchain'), 'stellar')

// ... complete wizard steps

// 2. Deploy contract

await user.click(screen.getByText('Deploy Web3 Quiz'))

// Verify wallet connection in store

expect(useWalletStore.getState().stellar.isConnected).toBe(true)

// 3. Navigate to dashboard

renderWithProviders(<HostDashboard />, {

initialEntries: [`/host-dashboard/${roomId}`]

})

// Verify wallet state persists

await waitFor(() => {

expect(screen.getByText('Stellar Wallet Connected')).toBeInTheDocument()

})

// 4. Test asset upload functionality

await user.click(screen.getByText('Upload Assets'))

// Should use existing wallet connection

expect(screen.queryByText('Connect Wallet')).not.toBeInTheDocument()

})

})

**9.3 Provider Testing Strategy**

Provider synchronization is tested with dedicated test suites:

// Provider sync testing

describe('StellarWalletProvider Store Sync', () => {

it('should sync wallet state to store on connection', async () => {

const mockWallet = createMockStellarWallet({

isConnected: true,

address: 'GTEST...WALLET'

})

renderWithProviders(

<StellarWalletProvider>

<TestComponent />

</StellarWalletProvider>

)

// Trigger connection

act(() => {

mockWallet.connect()

})

await waitFor(() => {

const store = useWalletStore.getState()

expect(store.stellar.isConnected).toBe(true)

expect(store.stellar.address).toBe('GTEST...WALLET')

expect(store.activeChain).toBe('stellar')

})

})

it('should clear store state on disconnection', async () => {

// ... test disconnection sync

})

})

**10. Migration and Deployment Strategy**

**10.1 Phased Migration Approach**

The migration from current implementation to this architecture follows a safe, incremental approach:

**Phase 1: Store Infrastructure (Week 1)**

* Implement store synchronization in existing providers
* Add generic action hooks
* Update useQuizChainIntegration to read from store
* Test with existing components

**Phase 2: Component Updates (Week 2)**

* Fix hook violations in StepWeb3ReviewLaunch and HostControlsPage
* Update components to use generic hooks
* Remove conditional hook calls
* Validate user flows work end-to-end

**Phase 3: Testing and Validation (Week 3)**

* Comprehensive testing of all user journeys
* Performance validation and optimization
* Error handling verification
* Documentation updates

**Phase 4: Future Chain Preparation (Week 4)**

* Create EVM and Solana provider templates
* Implement generic hook patterns for new chains
* Document chain addition process
* Prepare for multi-chain expansion

**10.2 Rollback Strategy**

Each migration phase includes rollback procedures:

// Feature flagging for safe deployment

const useRoomDrivenArchitecture = () => {

const isEnabled = process.env.REACT\_APP\_ROOM\_DRIVEN\_WALLET === 'true'

return {

useStoreSync: isEnabled,

useGenericHooks: isEnabled,

enableChainAgnosticComponents: isEnabled

}

}

**10.3 Backward Compatibility**

The migration maintains backward compatibility:

* Existing Stellar integration continues working during transition
* Components gracefully degrade if store sync unavailable
* Provider contexts remain functional as fallback
* User data and room configurations preserved

**11. Future Extensibility**

**11.1 Adding New Blockchains**

The architecture enables easy blockchain addition:

// Adding a new chain requires:

// 1. Chain-specific provider implementation

export const PolkadotWalletProvider: FC = ({ children }) => {

const polkadotWallet = usePolkadotWallet()

// Implement store sync pattern

useEffect(() => {

useWalletStore.getState().updatePolkadotWallet({

address: polkadotWallet.address,

isConnected: polkadotWallet.isConnected,

// ... sync all properties

})

}, [/\* polkadot wallet dependencies \*/])

// ... provider implementation

}

// 2. Update store interface

interface WalletState {

// ... existing chains

polkadot: PolkadotWalletConnection

updatePolkadotWallet: (updates: Partial<PolkadotWalletConnection>) => void

}

// 3. Update DynamicChainProvider

const ProviderComponent = useMemo(() => {

switch (selectedChain) {

// ... existing cases

case 'polkadot': return PolkadotWalletProvider

}

}, [selectedChain])

// 4. Update generic action hooks

const { polkadotWallet } = useAllWalletContexts()

const connect = useCallback(async () => {

switch (activeChain) {

// ... existing cases

case 'polkadot': return polkadotWallet.connect()

}

}, [activeChain, /\* all wallet contexts \*/])

**No component changes required** - all components automatically support the new chain through generic interfaces.

**11.2 Enhanced Features**

The architecture supports future enhancements:

**Multi-Chain Rooms**

// Future: Rooms supporting multiple chains

interface EnhancedRoomConfig {

supportedChains: SupportedChain[]

primaryChain: SupportedChain

crossChainBridging: boolean

}

**Advanced Wallet Features**

// Future: Multi-signature and advanced wallet features

interface AdvancedWalletFeatures {

multiSig: boolean

hardwareWallet: boolean

socialRecovery: boolean

crossChainSwaps: boolean

}

**Cross-Chain Prize Distribution**

// Future: Distribute prizes across multiple chains

interface CrossChainPrizeConfig {

distributions: Array<{

chain: SupportedChain

percentage: number

recipients: string[]

}>

}

**12. Monitoring and Analytics**

**12.1 Wallet Connection Analytics**

The system includes comprehensive analytics:

// Wallet connection tracking

const trackWalletConnection = (chain: SupportedChain, success: boolean) => {

analytics.track('wallet\_connection\_attempt', {

chain,

success,

timestamp: Date.now(),

userAgent: navigator.userAgent,

roomContext: getCurrentRoomContext()

})

}

// Provider loading performance

const trackProviderLoad = (chain: SupportedChain, loadTime: number) => {

analytics.track('provider\_load\_performance', {

chain,

loadTime,

timestamp: Date.now()

})

}

**12.2 Error Monitoring**

Comprehensive error tracking across all components:

// Error boundary reporting

const reportProviderError = (chain: SupportedChain, error: Error) => {

errorReporting.captureException(error, {

tags: {

component: 'wallet\_provider',

chain,

context: 'provider\_loading'

},

extra: {

walletState: useWalletStore.getState()[chain],

roomId: getCurrentRoomId(),

timestamp: Date.now()

}

})

}

**12.3 Performance Monitoring**

Real-time performance tracking:

// Store synchronization performance

const monitorStoreSyncPerformance = () => {

const startTime = performance.now()

useWalletStore.getState().updateStellarWallet(updates)

const syncTime = performance.now() - startTime

if (syncTime > 50) { // Alert if sync takes > 50ms

performance.mark(`slow-store-sync-${syncTime}ms`)

}

}

**13. Conclusion**

This room-driven multi-chain architecture provides a robust foundation for blockchain integration in the quiz application. The design prioritizes:

**User Experience**: Seamless wallet connections that persist across navigation, with chain selection determined by room context rather than global configuration.

**Developer Experience**: Chain-agnostic components that work identically across all supported blockchains, with clear patterns for adding new chains.

**Performance**: Lazy provider loading, optimized state synchronization, and efficient memory management.

**Reliability**: Comprehensive error handling, graceful degradation, and robust testing strategies.

**Extensibility**: Clear patterns for adding new blockchains and advanced features without breaking existing functionality.

The architecture acknowledges that the quiz application's room-based structure naturally leads to room-driven chain selection, and embraces this pattern rather than fighting against it. This results in a more intuitive user experience and a more maintainable codebase that aligns with the application's existing patterns and user mental models.

The next step is implementing the specific fixes outlined in the companion implementation document, which will transform the current problematic implementation into this well-architected, room-driven system.