

@volley/vgf

Volley Games Framework (VGF)

A powerful, opinionated framework for building multiplayer, real-time games with synchronized state and seamless client-server communication.

Table of Contents

- [Volley Games Framework \(VGF\)](#)
 - [Table of Contents](#)
 - [Overview](#)
 - [Quick Start](#)
 - [Installation](#)
 - [Basic Server Setup](#)
 - [Basic Client Setup](#)
 - [Documentation](#)
 - [Key Resources](#)
 - [Core Concepts](#)
 - [Game Ruleset](#)
 - [Phases](#)
 - [Actions](#)
 - [React Integration](#)
 - [Advanced Usage](#)
 - [Custom Transport Implementation](#)
 - [Custom Storage Implementation](#)
 - [Middleware Integration](#)
 - [Custom Routes](#)
 - [Best Practices](#)
 - [Error Handling](#)
 - [State Validation](#)
 - [Performance Optimization](#)


- [Examples](#)
 - [Learning Examples](#)
 - [CI/CD & Release Process](#)
 - [Conventional Commits](#)
 - [Commit Types](#)
 - [Breaking Changes](#)
 - [Examples](#)
 - [Automated Release Workflow](#)
 - [1. Commit Analysis](#)
 - [2. Release Notes Generation](#)
 - [3. Version Bumping](#)
 - [4. Publishing](#)
 - [5. Tagging](#)
 - [Merge Queue & Validation](#)
 - [Required Checks](#)
 - [Merge Queue](#)
 - [Branch Protection](#)
 - [Requirements](#)
 - [License](#)
-

Overview

VGF abstracts away the complexity of multiplayer game development by providing:

- **Real-time state synchronization** across all connected clients
- **Phase-based game logic** with declarative transitions
- **Action-based architecture** with server-side validation
- **Transport layer abstraction** supporting multiple networking implementations
- **Built-in session management** with Redis-backed persistence
- **React integration** with hooks and providers for seamless UI updates

Quick Start

 **Want a complete tutorial?** Check out our [Getting Started Guide](#) for a comprehensive step-by-step walkthrough of building your first VGF game.

Installation

```
npm install @volley/vgf
# or
pnpm add @volley/vgf
```

Basic Server Setup

```
import { VGFServer, SocketIOTransport, RedisStorage } from
"@volley/vgf/server"
import { createServer } from "http"
import express from "express"
import Redis from "ioredis"

// Define your game ruleset
const game = {
  setup: () => ({ phase: "lobby", players: [] }),
  // Global actions available in any phase
  actions: {
    someAction: {
      action: (ctx) => ctx.session.state,
    },
  },
  phases: {
    lobby: {
      actions: {
        startGame: {
          action: (ctx) => ({
            ...ctx.session.state,
            phase: "playing",
          }),
        },
      },
      next: "playing",
    },
  },
}

// Create server infrastructure
const app = express()
const httpServer = createServer(app)
const redisClient = new Redis()
```

```
const storage = new RedisStorage({ redisClient })

// Initialize VGF server
const server = new VGFServer({
  game,
  port: 3000,
  app,
  httpServer,
  transport: new SocketIOTransport({
    httpServer,
    redisClient,
    storage,
  }),
  storage,
})

server.start()
```

Basic Client Setup

```
import React from "react"
import { VGFPProvider, useTransport } from "@volley/vgf/client"

function App() {
  const transport = useTransport({
    url: "http://localhost:3000",
    userId: "user-123",
    autoCreateSession: {
      // If true, a session will be created automatically if one doesn't
      exist
      // only use when running games outside of the hub
      enabled: true
    },
    // Optional: Configure reconnection behavior
    reconnectionAttempts: 5,
    reconnectionDelay: 1000,
    reconnectionDelayMax: 5000,
    // Optional: Handle connection events
    onConnect: () => console.log("Connected to game server"),
    onDisconnect: (reason, active) => console.log(`Disconnected: ${reason}
    ${active ? "will attempt to reconnect" : "will not attempt to reconnect"}`),
    onReconnect: () => console.log("Reconnected to game server"),
  })
}
```


[Copy](#)

```
if (!transport) return <div>Connecting...</div>

return (
  <VGFPProvider transport={transport}>
    <GameComponent />
  </VGFPProvider>
)
```

Documentation

For comprehensive guides, API reference, and examples:

 **Complete Documentation** - Start here for tutorials, API reference, and examples

Key Resources

- **Getting Started Guide** - Complete step-by-step tutorial for building your first VGF game
- **API Reference** - Detailed documentation of all exports, classes, and interfaces
- **Observability Guide** - Production monitoring, logging, and performance optimization

Core Concepts

Game Ruleset

The `GameRuleset` is the central configuration object that defines your entire game:

```
import type { GameRuleset } from "@volley/vgf/server"

interface MyGameState {
  phase: string
  score: number
  currentPlayer: string
}

const game: GameRuleset<MyGameState> = {
  // Initialize game state
  setup: (initialData = {}) => ({
    phase: "lobby",
    score: 0,
    currentPlayer: "",
  })
```

```

    ...initialData,
  }),

  // Global actions available in any phase
  actions: {
    resetGame: {
      action: (ctx) => game.setup(),
    },
  },

  // Phase-specific logic
  phases: {
    lobby: {
      actions: {
        startGame: {
          action: (ctx, playerName: string) => ({
            ...ctx.session.state,
            phase: "playing",
            currentPlayer: playerName,
          }),
        },
      },
      endIf: (ctx) => ctx.session.state.currentPlayer !== "",
      next: "playing",
    },

    playing: {
      actions: {
        updateScore: {
          action: (ctx, points: number) => ({
            ...ctx.session.state,
            score: ctx.session.state.score + points,
          }),
        },
      },
      endIf: (ctx) => ctx.session.state.score >= 100,
      next: "gameOver",
    },
  },
}

```

Phases

Phases represent distinct states in your game's lifecycle. Each phase can:

- Define available actions
- Specify transition conditions (endIf)
- Include lifecycle hooks (onBegin, onEnd)
- Determine the next phase (next)

```
import type { Phase } from "@volley/vgf/server"

const lobbyPhase: Phase<MyGameState> = {
  // Actions only available in this phase
  actions: {
    joinGame: {
      action: (ctx, playerName: string) => {
        // Action logic here
        return {
          ...ctx.session.state,
          players: [...players, playerName],
        }
      },
    },
  },

  // Called when entering this phase
  onBegin: (ctx) => {
    ctx.logger.info("Entering lobby phase")
    return ctx.session.state
  },

  // Called when leaving this phase
  onEnd: (ctx) => {
    ctx.logger.info("Leaving lobby phase")
    return ctx.session.state
  },

  // Condition to automatically transition
  endIf: (ctx) => ctx.session.state.players.length >= 4,

  // Next phase to transition to
  next: "playing",
}
```

Actions

Actions are pure functions that receive the current game state and return a new state:

```
import type { GameAction, IGameActionContext } from "@volley/vgf/server"

const joinGameAction: GameAction<MyGameState, [string]> = (
  ctx: IGameActionContext<MyGameState>,
  ...args: string[]
) => {
```

@volley/vgf



```
// Access current state
const currentState = ctx.session.state

// Access session info
const playerId = ctx.sessionMemberId

// Use logger
ctx.logger.info({ playerName, playerId }, "Player joining")

// Return new state
return {
  ...currentState,
  players: [...currentState.players, { id: playerId, name: playerName }],
}
}
```

React Integration

VGF provides React hooks for seamless client-side integration:

```
import { getVGFFunctions } from "@volley/vgf/client"
import type { MyGameRuleset, MyGameState } from "../game"

// Generate typed hooks for your specific game
const {
  useStateSync,
  useDispatchAction,
  usePhase,
  useEvents
} = getVGFFunctions<MyGameRuleset, MyGameState, "lobby" | "playing" | "gameOver">()

function GameComponent() {
  // Get current game state
  const state = useStateSync<MyGameState>()

  // Get action dispatcher
```



```

const dispatchAction = useDispatchAction()

// Get current phase
const currentPhase = usePhase<"lobby" | "playing" | "gameOver">()

// Get lifecycle events
const { setPhase, endPhase } = useEvents()

return (
  <div>
    <h1>Current Phase: {currentPhase}</h1>
    <p>Score: {state.score}</p>

    <button onClick={() => dispatchAction("updateScore", 10)}>
      Add 10 Points
    </button>

    <button onClick={() => setPhase("gameOver")}>
      End Game
    </button>
  </div>
)
}

```

Advanced Usage

Custom Transport Implementation

Implement the `ITransport` interface to create custom networking layers:

```

import type { ITransport, IConnection } from "@volley/vgf/server"

class CustomTransport implements ITransport {
  onConnection(callback: (connection: IConnection) => Promise<void>): void
{
  // Implement connection handling
}
}

```

Custom Storage Implementation

Implement the `IStorage` interface for different persistence layers:

```
import type { IStorage, Session } from "@volley/vgf/server"

class DatabaseStorage implements IStorage {
  async create(session: Session): Promise<void> {
    // Implement session creation
  }

  async get(sessionId: string): Promise<Session | null> {
    // Implement session retrieval
  }

  // ... implement other required methods
}
```

Middleware Integration

Add custom middleware for observability, authentication, etc:

```
import { datadogServerMiddleware } from "@volley/vgf/server"
import tracer from "dd-trace"

const transport = new SocketIOTransport({ httpServer, redisClient, storage })

// Add Datadog tracing
transport.registerMiddleware(datadogServerMiddleware(tracer))

// Add custom middleware
transport.registerMiddleware((socket, next) => {
  // Custom logic here
  next()
})
```

Custom Routes

Register additional HTTP endpoints:

```
import { HTTPMethod } from "@volley/vgf/server"

server.registerRoute({
  path: "/api/custom",
  method: HTTPMethod.GET,
  handler: (req, res) => {
```

```
    res.json({ message: "Custom endpoint" })  
  },  
})
```

Best Practices

Error Handling

Handle errors gracefully in your actions:

```
const riskyAction: GameAction<MyGameState> = async (ctx) => {  
  try {  
    const data = await fetchExternalData()  
    return { ...ctx.session.state, data }  
  } catch (error) {  
    ctx.logger.error({ error }, "Failed to fetch data")  
    return {  
      ...ctx.session.state,  
      error: "Failed to load data",  
    }  
  }  
}
```

State Validation

Validate state before applying changes:

```
const updateScoreAction: GameAction<MyGameState, [number]> = (ctx, points) => {  
  if (points < 0) {  
    ctx.logger.warn({ points }, "Attempted to subtract points")  
    return ctx.session.state // No change  
  }  
  
  return {  
    ...ctx.session.state,  
    score: ctx.session.state.score + points,  
  }  
}
```

Performance Optimization

- Keep game state serializable and minimal
 - Monitor and optimize action execution time
-

Examples

Learning Examples

- [Getting Started Tutorial](#) - Build a complete number guessing game from scratch
- [Turn-Based Game](#) - Advanced example with complex game flow

The turn-based game example shows:

- Server setup with phases and actions
 - Client integration with React
 - State synchronization
 - Testing strategies
-

CI/CD & Release Process

VGF uses an automated release process powered by [semantic-release](#) to ensure consistent versioning, changelog generation, and deployments.

Conventional Commits

All commits must follow the [Conventional Commits](#) specification:

```
<type>[optional scope]: <description>
```

```
[optional body]
```

```
[optional footer(s)]
```

Commit Types

Type	Description	Release Impact
feat	New features or enhancements	Minor version

Type	Description	Release Impact
fix	Bug fixes	Patch version
docs	Documentation changes	No release
style	Code style changes (formatting, semicolons, etc.)	No release
refactor	Code refactoring without functional changes	No release
test	Adding or updating tests	No release
chore	Maintenance tasks, dependency updates	No release
ci	CI/CD configuration changes	No release
perf	Performance improvements	Patch version

Breaking Changes

To trigger a major version release, use one of these patterns:

```
# Exclamation mark after type/scope  
feat!: remove deprecated API methods
```

```
# BREAKING CHANGE footer  
feat: add new authentication system
```

```
BREAKING CHANGE: The old auth tokens are no longer supported
```

Examples

```
# Minor release (new feature)  
feat(client): add useGameTimer hook for phase timing
```

```
# Patch release (bug fix)  
fix(server): resolve memory leak in session cleanup
```

```
# Patch release (performance improvement)  
perf(transport): optimize WebSocket message batching
```

```
# No release (documentation)  
docs: update API reference with new examples
```

```
# Major release (breaking change)  
feat(server)!: redesign GameRuleset interface for better type safety
```

BREAKING CHANGE: `GameRuleset.setup()` now requires explicit return type

Automated Release Workflow

Every merge to the `main` branch triggers the automated release process:

1. Commit Analysis

- Analyzes commit messages since the last release
- Determines the next version number based on commit types
- Skips release if no relevant changes are found

2. Release Notes Generation

- Automatically generates changelog entries from commit messages
- Groups changes by type (Features, Bug Fixes, Breaking Changes)
- Links to related issues and pull requests

3. Version Bumping

- Updates `package.json` version
- Updates `pnpm-lock.yaml` lockfile
- Commits version changes back to `main`

4. Publishing

- Publishes new version to npm registry
- Creates GitHub release with generated notes
- Updates `CHANGELOG.md` with new entries




5. Tagging



- Creates Git tag for the new version
- Pushes tag to trigger any tag-based workflows

Merge Queue & Validation

To ensure code quality and proper commit formatting:

Required Checks

-  **Lint:** ESLint validation passes
-  **Type Check:** TypeScript compilation succeeds
-  **Tests:** All unit and integration tests pass

-  **Build:** Package builds successfully
-  **Commit Format:** Commit messages follow Conventional Commits

Merge Queue

- All PRs must pass through the merge queue
- Queue validates that the final squashed commit message is properly formatted
- Failed validation blocks the merge and requires commit message fixes

Branch Protection

- Direct pushes to `main` are blocked
 - All changes must go through pull requests
 - Status checks must pass before merging
-

Requirements

- **Node.js:** `>=22.0.0`
 - **pnpm:** `>=9.0.0`
 - **React:** `^19.0.0` (for client-side usage)
 - **Redis:** For session storage (recommended)
-

License

MIT - See LICENSE file for details.

Generated using [TypeDoc](#)