**Vanilla JS Game Framework - Developer Guide**

**🎯 Core Architecture Pattern**

**Request from LLM:** "Build a Vanilla JavaScript Game Architecture using the Modular Game Controller Pattern with Component-Based Game Screens"

**File Structure**

game/

├── app.js // Main game controller

├── config.js // All game data/templates

├── screen-name.js // Each screen = separate module

├── index.html // Static HTML structure

└── styles.css // CSS with animation classes

**🏗️ 1. Main Game Controller (app.js)**

class GameName {

constructor() {

// Centralized mutable state - everything lives here

this.gameState = {

currentScreen: "start-screen",

// Add all game data properties here

};

// Module composition - inject 'this' into each module

this.screenName = new ScreenName(this);

// Add more modules here

this.init();

}

init() {

this.setupEventListeners();

this.showScreen("start-screen");

}

// Screen state machine - only one screen active

showScreen(screenId) {

document.querySelectorAll(".screen").forEach(s => s.classList.remove("active"));

document.getElementById(screenId).classList.add("active");

this.gameState.currentScreen = screenId;

}

// Manual state synchronization - call after state changes

updateElement(elementId, content) {

const el = document.getElementById(elementId);

if (el) el.textContent = content;

}

}

**🖥️ 2. Screen Module Pattern (screen-name.js)**

class ScreenName {

constructor(game) {

this.game = game; // Dependency injection

}

// Template method pattern - override in each screen

setupEventListeners() {

document.querySelectorAll(".btn").forEach(btn => {

btn.addEventListener("click", (e) => {

this.handleClick(e.target.dataset.action);

});

});

}

handleClick(action) {

// Update state first

this.game.gameState.someProperty = newValue;

// Then update UI manually

this.updateUI();

}

// Manual lifecycle management

updateUI() {

this.game.updateElement("score", this.game.gameState.score);

// Update all relevant UI elements

}

}

**📊 3. Configuration-Driven Data (config.js)**

// Externalize ALL game data

const GAME\_CONFIG = {

MAX\_HEALTH: 100,

LEVEL\_COUNT: 5

};

// Template pattern for entities

const ENTITY\_TEMPLATES = {

player: {

name: "Hero",

health: 100,

// All entity properties

}

};

// Factory pattern utilities

const CONFIG\_UTILS = {

getRandomElement(array) {

return array[Math.floor(Math.random() \* array.length)];

},

createEntity(templateName) {

return { ...ENTITY\_TEMPLATES[templateName] };

}

};

**🎨 4. Dynamic UI Generation Patterns**

**Template String Rendering**

// For popups and dynamic content

showPopup(title, content) {

const popup = document.createElement("div");

popup.innerHTML = `

<div class="popup">

<h2>${title}</h2>

<p>${content}</p>

<button onclick="this.parentElement.remove()">Close</button>

</div>

`;

document.body.appendChild(popup);

}

**Data-Driven UI Rendering**

// For lists and grids

renderList(container, dataArray) {

container.innerHTML = ""; // Clear first

dataArray.forEach(item => {

const element = document.createElement("div");

element.innerHTML = `<span>${item.name}</span>`;

container.appendChild(element);

});

}

**Dynamic Entity Management**

// For game objects that spawn/despawn

spawnEntity(x, y, type) {

const entity = document.createElement("div");

entity.className = "game-entity";

entity.style.left = x + "px";

entity.style.top = y + "px";

gameArea.appendChild(entity);

// Auto-cleanup

setTimeout(() => entity.remove(), 5000);

}

**🎵 5. Common Game Systems**

**Audio Manager Pattern**

class AudioManager {

constructor() {

this.audioEnabled = false;

}

initializeAudio() {

this.audioEnabled = true;

}

playSound(soundName) {

if (!this.audioEnabled) return;

const audio = new Audio(`sounds/${soundName}.mp3`);

audio.play().catch(e => console.warn("Audio failed:", e));

}

}

**Animation Controller Pattern**

addAnimationClass(element, className) {

element.classList.add(className);

setTimeout(() => {

element.classList.remove(className);

}, 1000);

}

**📱 6. HTML Structure Pattern**

<!DOCTYPE html>

<html>

<head>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<!-- Screen State Machine - only one .active at a time -->

<div class="screen" id="start-screen">

<h1>Game Title</h1>

<button data-action="start">Start Game</button>

</div>

<div class="screen" id="game-screen">

<div id="game-area"></div>

<div id="ui-panel">

<span id="score">0</span>

</div>

</div>

<!-- Load order matters! -->

<script src="config.js"></script>

<script src="screen-name.js"></script>

<script src="app.js"></script>

</body>

</html>

**🎨 7. CSS Animation Integration**

/\* Animation classes for JavaScript \*/

.screen { display: none; }

.screen.active { display: block; }

.damage-animation {

animation: damageFlash 0.5s ease-out;

}

@keyframes damageFlash {

0% { filter: brightness(1); }

50% { filter: brightness(2) hue-rotate(180deg); }

100% { filter: brightness(1); }

}

**⚡ 8. Key Patterns & When to Use**

| **Pattern** | **When to Use** | **LLM Request Term** |
| --- | --- | --- |
| Template String Rendering | Dynamic popups, dialogs | "Template String UI Generation" |
| Data-Driven Rendering | Lists, grids, menus | "Data-Driven UI Rendering" |
| Dynamic Entity Management | Game objects, particles | "Dynamic Entity Management" |
| Manual State Sync | After any state change | "Manual State Synchronization" |
| Component Screens | Multi-screen games | "Component-Based Game Screens" |

**🚨 9. Critical Rules**

**State Management:**

1. **All state in gameState** - Never store state in DOM
2. **Update state first** - Then call UI updates
3. **Manual sync required** - Call update methods after changes

**DOM Manipulation:**

1. **Check element exists** - Use if (element) before accessing
2. **Clean up events** - Remove listeners when switching screens
3. **Clear before rebuild** - innerHTML = "" before adding new content

**Performance:**

1. **Batch DOM updates** - Multiple changes in one function
2. **Use CSS animations** - Not JavaScript for smooth effects
3. **Clean up timers** - Clear intervals/timeouts on screen changes

**📋 10. Quick Start Template**

**Request from LLM:** "Create a Vanilla JavaScript Game using this exact structure"

// app.js - Copy this structure

class MyGame {

constructor() {

this.gameState = { currentScreen: "menu" };

this.menuScreen = new MenuScreen(this);

this.init();

}

init() { this.setupEventListeners(); this.showScreen("menu"); }

showScreen(id) { /\* screen switching logic \*/ }

updateElement(id, content) { /\* safe DOM updates \*/ }

}

// menu-screen.js - Copy this pattern

class MenuScreen {

constructor(game) { this.game = game; }

setupEventListeners() { /\* event binding \*/ }

updateUI() { /\* manual UI sync \*/ }

}

// Initialize

document.addEventListener("DOMContentLoaded", () => {

new MyGame();

});

**🎯 Usage with LLMs**

**Example Request:**

"Build a turn-based battle game using the Vanilla JavaScript Game Architecture. Use the Modular Game Controller Pattern with Component-Based Game Screens. Include Dynamic Entity Management for battle animations and Template String UI Generation for battle results. Follow the Manual State Synchronization pattern."

This will give you a game that follows this exact structure and patterns!