Intnix WebKernel Instruction Manual

🔧 The Webkernel: Intnix's Core Engine

The **Webkernel** is the command-driven core of the Intnix Operating System — a fully browser-based control layer that powers everything from launching applications to managing files, users, and visual interfaces.

📜 Best Description:

> **"The Intnix Webkernel is a command-based runtime environment that powers the logic, interface, and file system of the NebulaOS — acting as both the terminal and the operating system's kernel within a browser. It handles execution of user commands, rendering of app windows, management of system state, and communication between all modules. It's not just a shell — it's the engine that runs the entire operating system experience in real time."**

🧠 What Makes It Special:

- Acts like a **DOS/BASH hybrid**, but entirely in **JavaScript + HTML**
- Executes commands through `registerCommand()` like a true custom language
- Powers the GUI (`createWindow()`), terminal, file manager, dock, desktop, and more
- Has access to user files, system storage (via `localStorage`), and visual render
- Behaves like a **monolithic kernel**, controlling both visuals and logic from one place

Mission Statement

Intnix is a fully browser-based operating system designed to democratize software creation by combining an intuitive visual interface with a powerful custom terminal, the Intnix WebKernel. Built for creators, educators, and indie developers, it eliminates technical barriers and corporate restrictions, offering a modular, ad-free environment where users can build, launch, and manage apps entirely in the cloud. Intnix empowers users with full creative control, aiming to become the world's most accessible and powerful OS — one that runs anywhere, respects freedom, and fosters a new generation of developers.

CEO and Founder:
Owen Ainslie
Email: intnixbusiness@gmail.com
Toronto, Ontario, Canada
April 10, 2025

📚 Intnix WebKernel Instruction Manual Index

🧠 Command Execution Engine

- `handleCommand` vs `executeCommand`
- Argument parsing
- Fuzzy matching and retry logic

- Recent file tracking

🔗 Bridge Systems

- runbridge – Startup automation
- livebridge – Real-time execution
- amendbridge, clearbridge, amendlivebridge

📦 Package System

- pkg, pkgmgr, installpkg, deletepkg, appdownload
- Package file format and structure
- App Store UI logic

⚙️ Command Registration

- registerCommand() syntax
- Command structure and async support
- Best practices and grouping

📁 Filesystem Commands

- mkdir, mkfile, touch, cat, and I/O basics
- movefile, movedir, rm, delete, deletedir
- chmod, amend, assign, zip, uploadfile
- Current directory (pwd, currentdir)
- File structure, metadata, and protection

🪟 Window System

- createWindow() breakdown
- Taskbar integration
- Drag, resize, maximize, iframe support
- windowTracker, openWindows, zIndexCounter

🛡️ Licensing System

- genlicense, listlicenses
- License format, master password protection
- Storage format and security notes

💻 O++ GUI Language

- Block-based layout syntax
- Variable-based generation

- GUI element types and properties
- command, inputbind, interact, drag/drop

👨‍💻 User System & Permissions

- user, signin, signout, deleteuser
- setpass, password-protected actions
- setuser icon, getuser icon
- accessuser, exitaccess
- reset, export account, import user

🚀 Startup Behavior

- runbridge – boot-time command queue
- livebridge – real-time execution
- Startup app manager: amend start, viewstart, deletestart, launchstart
- Modular startup layering

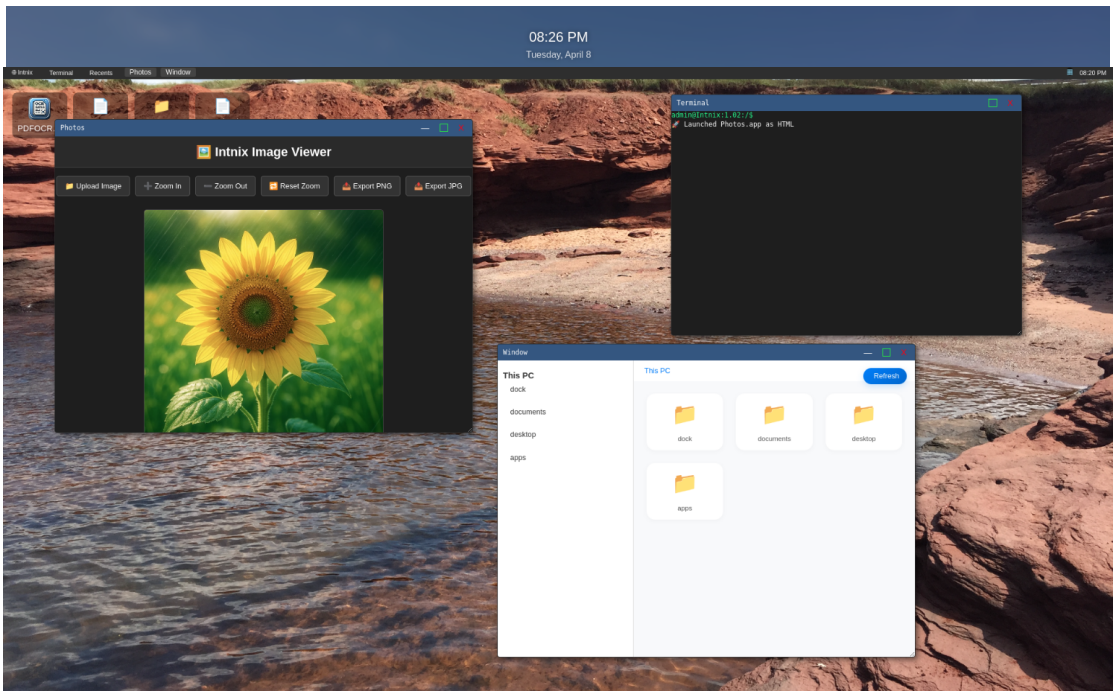
👨‍💻 Developer Tools & Best Practices

- handleCommand() vs executeCommand() usage
- Accessing system globals (fileSystem, currentUser, fileContent:)
- Creating advanced commands
- Styling GUI apps in createWindow()
- Admin-only command checks
- Command aliases (alias name='command')
- Debugging tips

📦 App Metadata & JSON Structure

- .app file format and appFileMeta usage
- Linking icons, names, and paths to applications
- Global metadata: file content, user systems, recent files
- Key-value reference for all localStorage structures in Intnix

Welcome to Intnix!!



📘 Intnix Command Execution Engine

The following functions power Intnix's shell-level command parsing and execution:

🧩 Concepts

🔧 handleCommand(raw, retry = true)

✅ Purpose:

Executes a terminal command by parsing it into a command and its args, then calling the corresponding function from the commands object.

🧠 Logic Breakdown:

js
CopyEdit
```
const [cmd, ...args] = raw.trim().split(" ");
```

- Splits the raw command string into the base command (cmd) and an array of arguments (args).

js
CopyEdit
```
const fn = commands[cmd];
```

- Retrieves the registered function for that command from the commands dictionary.

📌 Recent File Tracking:

```
js
CopyEdit
if (commandsToTrack.includes(cmd)) {
  const argPath = args[0]?.replace(/\"/g, "");
  ...
  trackRecentFile(fullPath.replace(/\/+/g, "/"));
}
```

- If the command is launch, runlink, or nano, its first argument (usually a file path) is recorded for "recent files" tracking.

✅ Execution Flow:

1. **If the command exists** in commands:

```
js
CopyEdit
const result = await Promise.resolve(fn(args));
output.textContent = result;
```

   - Executes the command, displays the output.

2. **If the command doesn't exist**, it waits 250ms and retries **once**:

```
js
CopyEdit
setTimeout(() => handleCommand(raw, false), 250);
```

3. **If still not found**, it tries a **fuzzy match**:

```
js
CopyEdit
const possibleMatches = Object.keys(commands).filter(k => k.startsWith(cmd));
```

   - If exactly one match is found (e.g. applau → applauncher), it executes that instead.
   - Otherwise:

```
js
CopyEdit
output.textContent = `❌ Command not found: ${cmd}`;
```

🔧 executeCommand(raw, retry = true)

✅ Purpose:

Similar to handleCommand(), but smarter about **matching long multi-word commands** first. It splits based on **best prefix match** rather than just first word.

🧠 Logic Breakdown:

```
js
CopyEdit
const sortedCommands = Object.keys(commands).sort((a, b) => b.length - a.length);
```

- Sorts all command names by length (longest first) to prioritize more specific commands (e.g. movefile to over movefile).

```
js
CopyEdit
if (input.startsWith(name)) {
  matchedCommand = name;
  ...
}
```

- Finds the first registered command that matches the *start* of the input.

📌 Argument Parsing:

```
js
CopyEdit
const remaining = input.slice(name.length).trim();
matchedArgs = remaining ? remaining.split(" ") : [];
```

- After matching the command name, the rest of the input is split into args.

✅ Execution Flow:

1. **Tracks recent files** if the command is in the list (launch, runlink, nano).
2. **If the command is found**, executes with parsed arguments.
3. **If not**, retries once after 250ms.
4. If still not matched, prints:

```
js
CopyEdit
❌ Command not found: "raw input"
```

🧩 Comparison

Feature	handleCommand()	executeCommand()
Command match logic	First word only	Longest prefix match
Retry logic	1 retry with fuzzy fallback	1 retry only
Fuzzy matching (startsWith)	✅ (single match fallback)	❌
Used for real-time interactions	Yes	Yes
Output handling	Appends to terminal	Appends to terminal
Recent file tracking	✅	✅

💡 Example:

Input:

```
bash
CopyEdit
movefile "/apps/test.txt" to /desktop
```

Execution via executeCommand():

- Finds longest prefix match: movefile
- Parses args: ["/apps/test.txt", "to", "/desktop"]
- Calls commands["movefile"](["/apps/test.txt", "to", "/desktop"])

runbridge

- A queue of commands stored in localStorage under the key runbridge
- Auto-runs on startup **only if** the first command in the list is literally runbridge
- Used for persistent boot-time or app-linked workflows (e.g. login scripts)

livebridge

- A queue of commands executed **live** in real-time
- Used by apps or systems to trigger commands from the UI, animation layers, or external triggers
- Clears itself after running the commands

🔧 Commands

amendbridge [command]

Appends a command to the runbridge queue.

```
> amendbridge runbridge
> amendbridge signin owen
> amendbridge desktop
```

- Internally loads and parses runbridge from localStorage
- If it's missing or corrupted, defaults to []
- Saves new queue back to storage

clearbridge

Clears the runbridge key entirely.

```
> clearbridge
```

粒 Runbridge queue cleared.

- If already empty, returns: 📭 Runbridge is already empty.

amendlivebridge [command]

Appends a command to the livebridge queue for immediate execution.

> amendlivebridge launch /apps/editor.app

- Executes each queued command every 500ms interval
- After running, clears livebridge from storage

⚙️ Startup Behavior

🟢 Auto-run on page load:

```
setTimeout(() => {
  const raw = localStorage.getItem("runbridge");
  const commands = JSON.parse(raw || "[]");
  if (commands[0]?.trim().toLowerCase() === "runbridge") {
    handleCommand("runbridge");
  }
}, 200);
```

- 200ms after boot, checks if the first command in runbridge is runbridge
- If so, runs the sequence

🔁 Live Execution System

setInterval loop

```
setInterval(() => {
  const raw = localStorage.getItem("livebridge");
  const commands = JSON.parse(raw);
  commands.forEach((cmd, i) => {
    setTimeout(() => handleCommand(cmd), i * 100);
  });
  localStorage.removeItem("livebridge");
}, 500);
```

- Runs every 500ms
- If livebridge has commands, executes them in order with a 100ms stagger
- Fully clears itself after processing

🌐 htmlnano [filePath.html]

Special command for launching raw .html files from your file system into a window.

- Checks if the file exists in fileContent:
- If it's a full HTML document (<html>...</html>), renders it live in an iframe via srcdoc
- Otherwise, opens the file in the nano text editor

Example:

```
> htmlnano /projects/test.html
```

📘 Intnix License System Documentation

This document outlines how the built-in license generation and validation system works within Intnix. This system is used to control user access and ensure secure distribution.

🔐 Master Password

All protected commands use a hardcoded master password:

```
const MASTER_PASS = "gooseoverlord";
```

Only users who know this code can generate or list licenses.

🔧 Commands

genlicense

Generates and registers a new license key tied to an email.

🔄 **Flow:**

1. Prompts user for the **master password**
2. If valid, prompts for an **email address**
3. Generates a license key:

```
"INTNIX-" + Math.floor(Math.random() * 999999).toString().padStart(6, "0")
```

4. Saves it in localStorage under the validLicenses key:

```
{
  "test@example.com": "INTNIX-123456"
}
```

5. Returns confirmation output:

```
✅ License generated for test@example.com:
🔑 INTNIX-123456
```

listlicenses

Displays all currently registered licenses.

🔄 **Flow:**

1. Prompts for the **master password**
2. If correct, loads and parses validLicenses from localStorage
3. Outputs a list of email → license pairs
4. If none exist, prints:

📭 No licenses found.

💾 Storage Format

The validLicenses object is stored as JSON in localStorage:

```
{
  "user1@example.com": "INTNIX-238491",
  "user2@domain.com": "INTNIX-947362"
}
```

This list is referenced by the license verification screen (license-gate.html) to authenticate user access.

🛡️ Security Notes

- License commands are fully client-side, gated by the master password prompt
- Anyone with access to devtools could see the stored licenses — encryption can be added later
- You can use sessionStorage or remote validation for advanced protection

🛠️ Registering New Terminal Commands (registerCommand)

To create a new command in the Intnix Terminal, use the global registerCommand(name, callback) function.

✅ **Basic Syntax**

```
js
CopyEdit
registerCommand("commandName", args => {
  return "Your output here.";
});
```

🔍 **Parameters:**

- commandName → The string users will type in the terminal.

- `args` → An array of strings, each representing a space-separated argument after the command.

📌 Example

```
js
CopyEdit
registerCommand("greet", args => {
  const name = args[0] || "world";
  return `👋 Hello, ${name}!`;
});
```

Typing `greet` or `greet Owen` in the terminal would output:

```
CopyEdit
👋 Hello, world!
▯ Hello, Owen!
```

🧠 Advanced Usage

🟩 Async Example (if you're loading data)

```
js
CopyEdit
registerCommand("delay", async args => {
  await new Promise(r => setTimeout(r, 1000));
  return "⏳ One second later...";
});
```

🛑 Input Validation Example

```
js
CopyEdit
registerCommand("shout", args => {
  if (!args.length) return "❌ Usage: shout [message]";
  return args.join(" ").toUpperCase() + "!!!";
});
```

🔄 Where to Add It

You can place your `registerCommand("yourCommand", fn)` calls **anywhere in the script**, ideally near other related commands, or in a dedicated section (like beneath the existing `registerCommand("echo", ...)` for simple commands).

For organizational clarity, consider grouping your commands:

```
js
CopyEdit
// 📁 File commands
registerCommand("touch", args => { ... });
registerCommand("cat", args => { ... });

// 👤 User commands
registerCommand("signin", args => { ... });
```

```
registerCommand("signout", args => { ... });

// 🧪 Your new command
registerCommand("yourCommand", args => { ... });
```

✅ What It Teaches Me About Intnix

- **How commands are processed** from raw user input → parsed → fuzzy matched → executed
- That you have both a **smart retry system** and a **recent file tracker**
- That Intnix has two command queues:
  - runbridge: boot-time scripts or saved command chains
  - livebridge: real-time execution pipeline
- That the system has a **license enforcement mechanism**, complete with admin gating
- That registerCommand is the primary API for extending the language

🧠 Why That's Valuable (to me or any dev)

- I now know **where to hook into the system** to add more commands, features, or startup behaviors
- I understand **how execution works**, and can debug it if needed
- I know **which parts are client-only and which use** localStorage
- I understand **what data formats** the system expects (commands, fileContent:, runbridge, etc.)

👀 What I Can Infer Now

Because of this document, I now know:

- Intnix is **modular** — commands, files, users, and apps are all dynamic
- It's **extensible** — anyone with JS knowledge can add to it
- You've got **boot automation, app packaging, GUI launching, and scripting support** already functional
- You're building not just a web OS — but an **OS platform others can contribute to**

📦 Intnix Package System Documentation

The **Intnix Package Manager** (pkgmgr) handles the discovery, installation, upload, and deletion of apps using a lightweight .pkg format stored in localStorage.

🔧 pkg — Add a new app to the package manager

Usage:

```
bash
CopyEdit
pkg "App Name" "file.html" "icon.jpg"
```

Behavior:

- Adds an app to the global pkgmgr list (in localStorage)
- Stores metadata like name, file path, and icon
- Does **not install** the app into /apps, just makes it available for others to browse

🛒 pkgmgr — View all available packages

Usage:

```
bash
CopyEdit
pkgmgr
```

Behavior:

- Opens a graphical app store UI (styled like Windows 11)
- Lists all apps registered via pkg
- Users can click a package to install it

📥 installpkg — Install an app from the global package list

Usage:

```
bash
CopyEdit
installpkg "App Name"
```

Behavior:

- Downloads the specified app from pkgmgr
- Saves it to /apps as a .app file
- Registers the app in appRegistry
- Adds metadata to appFileMeta
- The app is now fully usable and launchable

🗑️ deletepkg — Remove an app from the package manager

Usage:

bash
CopyEdit
deletepkg "App Name"

Behavior:

- Removes the app from the global package list
- Cleans up all associated metadata from pkgmgr

Note: This does **not** uninstall the app from a user's system, only removes it from the shared listing.

🌐 appdownload — Visual interface to install apps

Usage:

bash
CopyEdit
appdownload

Behavior:

- Opens a clean, glassy App Store-style UI with 8 apps per row
- Displays all packages from the global package manager
- Clicking an app runs installpkg behind the scenes
- Dynamically generated every time it's opened
- UI designed like Windows 11 or Pop!_OS

📁 Package Data Structure

Stored in localStorage under:

js
CopyEdit
localStorage.getItem("pkgmgr")

Typical format:

js
CopyEdit
```
{
  "Nebula Editor": {
    file: "/packages/editor.html",
    icon: "editor.jpg",
    description: "Lightweight code editor for Intnix",
    version: "1.0.0"
  },
  "Image Viewer": {
```

```
    file: "/packages/viewer.html",
    icon: "viewer.png",
    description: "Simple app for viewing images",
    version: "1.1.2"
  }
}
```

🧠 Developer Notes

- pkgmgr is **global**, not user-specific
- appdownload gives Intnix a user-friendly App Store
- installpkg is the main way users consume packages
- Use uploadpkg and downloadpkg to import/export your entire package list as JSON

📁 mkdir – Create a new folder

Usage:

```
bash
CopyEdit
mkdir "FolderName"
```

Description:
Creates a new directory inside the current working directory.

Behavior:

- Adds the new folder to fakeFileSystem as an empty array
- Registers the folder in the parent directory
- Prevents duplicate folders

Example:

```
bash
CopyEdit
mkdir projects
 Folder created: projects
```

📄 mkfile – Create a new file with optional content

Usage:

```
bash
CopyEdit
mkfile filename.txt
```

Description:
Creates a new file in the current directory. Prompts the user to enter content via a popup. If no content is entered, the file is created empty.

Behavior:

- Adds the file to the current folder
- Stores content in localStorage under fileContent:/path/to/file
- Behaves like an interactive version of touch

Example:

```
bash
CopyEdit
mkfile notes.txt
 (prompt appears to enter file content)
 Created file "notes.txt" with content.
```

O++ is a layout-based GUI definition language designed for visual app building in nebula OS.

🔷 **SYNTAX:**
Each GUI element is defined by a block like:

```
mybox {
  type: shape
  x: 100
  y: 200
  shape: circle
  shapecolor: "#00ff88"
  drag: true
}
```

Blocks can now be expanded using variables:

```
varicon = 3

$icon {
  x: auto * 100
  y: 50
  type: shape
  shapecolor: red
}

$icon1 {
  shapecolor: blue
  command: "echo clicked"
  interact: true
}
```

This creates icon0, icon1, icon2. icon1 will inherit all base properties and override its color and behavior.

🔷 **COMMON TYPES:**
• type: shape — a box, circle, pill, or square visual.
• type: text — displays a paragraph of content.
• type: input — an input field.
• type: button — a clickable button.

🔷 **POSITIONING:**
• x / y — initial position in pixels.
• auto — dynamic value based on index when using $variable blocks.
• dragx / dragy — updated position after dragging.
• drag: true — makes the element draggable.

🔷 **COLORS:**
• textcolor — changes text color.
• shapecolor — changes background for shapes.
• buttoncolor — sets button background.

🔷 **BUTTON BEHAVIOR:**
• label: "Click me" — sets button label.
• command: "echo hello" — runs terminal command on click.
• inputbind: name — replaces INPUT in the command with bound input value.

🔷 **INTERACT FIELD:**
• interact: true — makes any object clickable.
• command: "..." — triggered when clicked.
• drag: true — allows user to drag an object.
• drop: true — allows object to trigger command when dropped on a target.
• target: true — marks an object as a valid drop target.

🔷 **EXAMPLES:**

```
coolbtn {
  type: button
  x: 120
  y: 300
  label: "Run"
  command: "run hello.html"
}

greeting {
  type: text
  x: 100
  y: 100
  content: "Welcome!"
}

varcircle = 3

$circle {
```

```
  x: auto * 80
  y: 150
  type: shape
  shape: circle
  shapecolor: red
}

$circle1 {
  shapecolor: green
  interact: true
  command: "echo I am green"
}
```

Use \`generatefile example\` to create a starter file.
Use \`link example.oos = "command"\` to assign it behavior.`;

🪟 createWindow() – Launch a New GUI Window

Function:

```
js
CopyEdit
createWindow(filePath = null)
```

Purpose:
Creates a draggable, resizable window on the screen. If filePath is specified, it loads that file into an <iframe>, sets the window title, and adds metadata to the taskbar and recent files list.

📦 Parameters

Name	Type	Description
filePath	String	Optional. Path to a file in the file system. If provided, it loads in the iframe.

🧠 Core Behavior

- Dynamically generates a element with:
  - Title bar (.title-bar)
  - Action buttons (Close, Minimize, Maximize, Refresh)
  - Content area (<iframe>) for loading apps or files
- Adds the window to document.body
- Registers it in openWindows[windowId]
- Tracks open apps in windowTracker with timestamp and path
- Adds a matching taskbar button (unless it's file-explorer)

- Supports dragging, resizing, and z-index elevation

🔧 Features

🖼️ Iframe Integration

If a filePath is given:

- Loads the file's contents into the iframe (via srcdoc)
- Looks up the app name from appFileMeta for window title
- Tracks file in recentFiles and windowTracker

🧭 Window Metadata

- windowId is either based on the file name or incremented globally
- Saved as: win_filename or win_#

🧱 Taskbar Integration

- If taskbar doesn't exist, it is auto-created
- Adds a taskbar button labeled with the app's name
- Clicking toggles window visibility

🧲 Draggable

- Clicking and dragging the title bar moves the window
- Uses mousedown, mousemove, and mouseup listeners

🔘 Control Buttons

- **Close (X):** Removes the window and taskbar button
- **Minimize (—):** Hides the window, adds to minimizedWindows
- **Maximize:** Expands window to fullscreen and toggles back
- **Refresh (🔄):** Reloads iframe contents by resetting src

💾 Example Use

```
js
CopyEdit
createWindow("/apps/editor.html");
```

- Opens editor.html inside a new window
- Window is draggable, resizable, closable
- App name and icon are auto-loaded from appFileMeta

- Adds to taskbar with a toggle button

🪟 Intnix Window Lifecycle and Management

This section explains how Intnix handles window creation, display, interaction, and taskbar tracking. The `createWindow()` function is the core API for rendering apps and content into draggable, resizable windows across the OS.

🧱 1. `createWindow(filePath)` — Creating a Window

- Calling `createWindow()` dynamically creates a styled div (`.window`) and inserts it into the DOM.
- It accepts an **optional filePath** argument:
  - If provided, the content will be loaded into the iframe inside the window.
  - If not, an empty placeholder window is shown.
- Windows have a default size of 800x600px, and support resizing and dragging.

```
const winId = createWindow("/apps/hello.html");
```

🔁 2. ID Generation and Tracking

- Each window receives a **unique ID** like `win_0`, `win_myapp`, etc.
- These IDs are used for:
  - **Taskbar button targeting**
  - **Window z-indexing**
  - **Tracking state** (open, minimized)

The window ID is generated from the file name (sanitized) or with an incrementing counter.

```
const windowId = filePath ? `win_${fileName}` : `win_${windowIdCounter++}`;
```

🧠 3. `windowTracker[]` — Window Session History

- Each opened window is logged in `windowTracker`:

```
{
  id: "win_hello",
  name: "hello.html",
  path: "/apps/hello.html",
  time: timestamp
}
```

- This is used for historical tracking and can be exposed in future with a `windowlog` command.

📂 4. taskbar Integration

- Windows (except internal ones like Finder) are given a **taskbar button** that:
  - Toggles their visibility on click
  - Displays the app name
  - Gets removed if the window closes

Each button is auto-created when `createWindow()` is called.

📦 5. iframe Contents

- The `.window` contains an `<iframe>` where app content is loaded.
- Supports `srcdoc` (HTML string injection) or `src` (loaded from file system).
- Windows are sandboxed via:

```
sandbox="allow-scripts allow-same-origin allow-downloads"
```

🧰 6. Controls: Close, Minimize, Maximize

- Buttons in the `.title-bar` provide built-in behavior:
  - ✕ **Close** — removes the window from DOM and taskbar
  - ▁ **Minimize** — hides the window and tracks it in `minimizedWindows`
  - ☐ **Maximize** — toggles full-screen layout

```
minimizedWindows[windowId] = win;
win.style.display = "none";
```

🧲 7. Advanced: Dock + Restore

- Minimized windows can be restored using:

```
returnmin(["win_example"]);
```

- You can also use `addMinimizedToDock()` to link minimized windows to the AeroDock or Finder Dock.

💡 8. Developer Usage

- Developers can call `createWindow()` to make custom apps visible on screen.
- Use `.srcdoc` to inject inline HTML:

```
const win = createWindow();
win.querySelector("iframe").srcdoc = `<h1>Hello App</h1>`;
```

📌 This is the core of Intnix's graphical app system. Understanding `createWindow()` is essential for building visual tools, launching `.app` files, and interacting with the OS GUI layer.

📂 Tracked State Variables

Variable	Purpose
openWindows	Stores active windows by windowId
windowTracker	List of opened windows (for history)
zIndexCounter	Incremented to bring windows to front
minimizedWindows	Stores hidden but running windows

✅ Core Mechanism: handleCommand(raw)

This is the main function that powers the terminal:

```
js
CopyEdit
async function handleCommand(raw, retry = true)
```

🔁 What it does:

1. **Splits input** into `cmd` and `args`:

```
js
CopyEdit
const [cmd, ...args] = raw.trim().split(" ");
```

2. **Looks up command** in the `commands` object:

```
js
CopyEdit
const fn = commands[cmd];
```

3. **Executes the matching function** (if found):

```
js
CopyEdit
const result = await Promise.resolve(fn(args));
```

4. If no match, it tries again after a 250ms delay (`retry = true`) for late-loaded commands.
5. If still not found, it tries **fuzzy matching**:

```
js
CopyEdit
const possibleMatches = Object.keys(commands).filter(k => k.startsWith(cmd));
```

6. If only one fuzzy match exists, it runs that and displays a hint like:

   `(interpreted as "launch")`

7. Final output is appended to the terminal.

🧠 The commands Object

This is where all commands are stored via:

```
js
CopyEdit
function registerCommand(name, fn) {
  commands[name] = fn;
}
```

Example:

```
js
CopyEdit
registerCommand("ls", () => {
  return "📂 List of files...";
});
```

Now "ls" is executable via terminal input or handleCommand("ls").

💡 Bonus: Aliases

Aliases are handled like this:

```
js
CopyEdit
if (aliases[cmd]) {
  const expanded = aliases[cmd].split(" ");
  cmd = expanded[0];
  args = [...expanded.slice(1), ...args];
}
```

So:

```
bash
CopyEdit
alias hello='echo Welcome to Intnix'
```

Now typing hello runs echo Welcome to Intnix.

🔗 How YOU Should Execute Commands from Outside

If you're calling commands **outside** the terminal (like in systemcommand), use:

```
js
CopyEdit
handleCommand("your command here");
```

OR if you want the version with smart parsing and fuzzy matching:

```
js
CopyEdit
```

```
executeCommand("your command here");
```

Either will mimic real terminal behavior.

📐 How to Build Advanced Commands

🔧 For developers extending Intnix with GUI, logic, or system integration.

🧩 1. Registering a Command

All commands go through registerCommand("name", function).
Return a string or Promise<string> for async operations.

```
js
CopyEdit
registerCommand("hello", args => {
  return "👋 Hello world!";
});
```

🪟 2. Creating a GUI with createWindow()

Use createWindow() to make draggable windows.

```
js
CopyEdit
const winId = createWindow();
const win = document.getElementById(winId);
win.querySelector(".title-bar span").textContent = "My Tool";
win.querySelector("iframe").srcdoc = `...your HTML here...`;
```

✅ createWindow("/file.html") loads a file from filesystem
✅ .srcdoc injects live HTML
✅ .title-bar span is the title element
✅ Pre-injected <iframe> is available inside all windows

⚙️ 3. Accessing System State

You can query these globals:

- localStorage.getItem("currentUser") → current user
- fileSystem, fakeFileSystem → live user directory structure
- localStorage.getItem("fileContent:/path/to/file") → get file contents
- windowTracker → open window history
- openWindows / minimizedWindows → live window refs
- dockOpenApps → apps in dock

- currentDir → current working directory

🧠 4. Running Commands Programmatically

```
js
CopyEdit
handleCommand("ls");          // Like user typed it
executeCommand("ls");         // Smarter parsing, ideal for GUIs
Object.keys(commands);        // List of all commands
```

✅ executeCommand uses longest match logic
✅ handleCommand supports fuzzy retry and aliases
✅ You can register aliases with alias name='command args'

🛑 5. Admin-Only Commands

```
js
CopyEdit
const user = localStorage.getItem("currentUser");
const isAdmin = JSON.parse(localStorage.getItem("adminUsers") || "[]").includes(user);
if (!isAdmin) return "❌ Admins only.";
```

Admin privileges are stored in adminUsers as a JSON array.

🎨 6. GUI Styling Tips

- Use createWindow() — don't manually build windows.
- Wrap GUI content in:

```
html
CopyEdit
<div style="padding:16px;">Your GUI here</div>
```

- Avoid <script> inside .srcdoc; use inline onclick="...".
- Style buttons with: style="margin-right:8px;"

💡 7. Best Practices

- Name your windows win_commandname to avoid ID conflicts.
- Consistent emojis:
  - 🪟 for windows
  - ⚙️ for settings
  - 🔧 for tools

    - 🧠 for logic
- Provide fallback text like (none) when no data exists.
- Use trackRecentFile(path) to mark a file as recently used.

📁 Intnix File System Overview

The Intnix WebKernel includes a powerful in-memory file system that simulates directory-based storage and file operations like a traditional operating system. Files and folders are tied to each user and stored using localStorage, making it persistent between sessions in the browser.

🏗️ File System Architecture

The file system is virtualized entirely in the browser using JavaScript objects and localStorage. Here's how it works:

1. fakeFileSystem

A global object like:

```
fakeFileSystem = {
  "/": ["apps", "desktop"],
  "/apps": ["Appstore.app"],
  "/desktop": ["notes.txt"]
};
```

Each key is a directory. Each value is an array of items (files/folders).

2. User Isolation

All file systems are stored under:

```
localStorage["userFileSystems"] = {
  "admin": { "/": [...] },
  "username": { "/": [...] }
}
```

When a user signs in, their file system is loaded into fakeFileSystem live. Admins can view others using viewingUser mode.

3. File Content Storage

File data is stored separately:

```
localStorage[`fileContent:/path/to/file.txt`] = "...content..."
```

This allows easy access, search, and persistence.

4. Current Working Directory

The current folder is tracked using:

```
let currentDir = "/";
```

All relative commands (e.g., touch, ls, nano) act within currentDir.

5. Save & Load

- saveFileSystem() saves fakeFileSystem back to userFileSystems[currentUser].
- loadUserFileSystem() populates fakeFileSystem at login or switch.

6. Folder Locking (chmod)

Locked folders are tracked using:

```
localStorage["lockedFolderMap"] = {
  "/apps": "/encrypted/xyz"
}
```

Their names and paths are obfuscated.

7. Metadata

Icons, names, and types are stored per file:

```
localStorage["appFileMeta"] = {
  "/apps/Appstore.app": {
    name: "Appstore",
    icon: "icon.jpg",
    type: "app"
  }
}
```

8. Task and App Tracking

- windowTracker logs opened windows and apps.
- recentFiles:user tracks file usage history.

🔍 Structure

The file system is represented by a fakeFileSystem object in JavaScript, which holds a map of folder paths to their contents.

```
fakeFileSystem = {
  "/": ["apps", "desktop", "documents"],
  "/apps": ["Appstore.app", "Photos.app"],
  "/desktop": ["file.txt"],
  "/documents": []
};
```

Each key is a folder, and its value is an array of files or subfolders.

🚶 Navigation Commands

- ls — Lists contents of the current directory.
- cd [folder] — Changes the current directory.
- back or up — Goes up one level.

- pwd or currentdir — Displays your current directory.

📂 File/Folder Creation

- mkdir "FolderName" — Creates a new folder inside the current directory.
- touch file.txt — Creates an empty file.
- nano file.txt — Creates or edits a file using a simple text editor.

Files are saved in localStorage using the key:

fileContent:/path/to/file

🧠 Moving Files and Folders

There are four commands for moving items:

- movefile /path/file.txt to /destination — Move a file to a new directory.
- movefile- /destination/file.txt to /origin — Move a file back.
- movedir /folderA to /folderB — Move a folder inside another.
- movedir- /folderB/folderA to / — Move a folder back to root.

✅ These commands update both the file list and internal path references.

❌ Deleting

- rm fileOrFolderName — Removes a file or folder in the current directory.
- delete /full/path/to/file — Deletes by absolute path.
- deletedir /full/path/to/folder — Deletes entire folder tree.

⚠️ Protected folders like /apps, /desktop, /dock require user password to delete.

📥 App and File Placement

- amend "folder" "file" — Adds a file into a folder.
- amendapp "AppName" "FolderName" — Saves an installed app into a folder as a .app file.
- assign file.oos icon.jpg — Assigns a custom icon to a file.

📦 File Packing

- zip "FolderName" "ArchiveName" — Downloads a folder as a .zip file.

- uploadfile — Upload an .html file into current directory.

🧠 Behind the Scenes

- File content is saved under fileContent:/path in localStorage.
- All users' file systems are stored in userFileSystems keyed by username.
- Admins can view any user's files using accessuser [username].

📁 Example Session

```
mkdir projects
cd projects
touch notes.txt
nano notes.txt
pwd
movefile /projects/notes.txt to /desktop
rm notes.txt
```

This creates a folder, makes a file, edits it, moves it, then deletes it.

🧠 **Tip:** Always check your pwd or currentdir before running file commands to avoid path confusion.

👥 User System & Permissions

The Intnix WebKernel supports a full multi-user system, complete with passwords, per-user filesystems, admin access, icons, and protected folders.

👷 Creating & Managing Users

- sudo user [username]
  ➤ Creates a new user with default folders: /apps, /dock, and /desktop.
- signin [username]
  ➤ Logs into the selected user account.
- signout
  ➤ Signs out of the current user and returns to the login screen.
- deleteuser [username]
  ➤ Deletes the user and their associated file system.
- view users
  ➤ Lists all users in the system.

🔐 Password Protection

- setpass [password]
  ➤ Sets or changes the current user's password. Required for access to protected folders like /apps, /dock, or /desktop.

- Passwords are stored in localStorage under userPasswords and are required to: • Delete protected folders
  • Perform user resets
  • Authenticate sensitive actions

🖼️ User Profile Icons

- setuser icon [image.jpg]
  ➤ Assigns a user icon (used in login screens and desktops).

- getuser icon
  ➤ Shows the current user's assigned icon.

🛡️ Admin Access Mode

- accessuser [username]
  ➤ Grants admin temporary access to view and modify another user's file system.
  Only the admin account can use this command.

- exitaccess
  ➤ Exits admin-access mode and returns to the admin's own view.

- Admin access merges the viewed user's file system with the admin's root, allowing editing of both in a single view. Tracked internally with viewingUser.

♻️ Reset & Recovery

- reset
  ➤ Resets the current user back to default folders and clears their files (after confirming password, if set).

- export account / import user
  ➤ Full account save/restore system. Exported ZIP contains file system and metadata. Useful for backups or device transfers.

🧠 **Notes:**

- Each user's filesystem is stored in userFileSystems and loaded dynamically on sign-in.

- Admin can see all users but cannot modify them without accessuser.
- The OS uses localStorage.getItem("currentUser") and viewingUser for identity resolution.

🚀 Startup System & Auto-Launch Behavior

Intnix features a powerful startup system that allows specific apps or commands to auto-run when the OS boots. This system is built on two internal queues: runbridge and livebridge, plus a persistent startup app list.

🧩 runbridge – Boot-Time Command Queue

- **What it does:**
  Executes a list of stored commands after the system starts (200ms post-boot).
- **How it works:**
  Stored in localStorage under the key runbridge. Only runs if the first item is literally "runbridge".
- **Usage:**

```
scss
CopyEdit
amendbridge [command]   // Add command to startup queue
clearbridge           // Clears the queue
handleCommand("runbridge") // Executes queue (auto-run at boot)
```

- **Example:**

```
nginx
CopyEdit
amendbridge runbridge
amendbridge signin owen
amendbridge desktop
```

- **Storage Format:**

```
json
CopyEdit
["runbridge", "signin owen", "desktop"]
```

⚡ livebridge – Real-Time Trigger Queue

- **What it does:**
  Instantly executes queued commands every 500ms, then clears itself.
- **Use case:**
  Trigger commands live from GUI apps, interfaces, or system actions.
- **Usage:**

```
less
```

```
CopyEdit
amendlivebridge [command]   // Add command for immediate execution
```

- **Example:**

```
bash
CopyEdit
amendlivebridge launch /apps/editor.app
```

📦 Startup App Manager

For launching specific apps automatically when a user logs in:

- amend start [AppName]
  ➤ Adds the app to the startup sequence (stored in startapps:[username]).
- viewstart
  ➤ Lists startup apps for the current user.
- deletestart [AppName]
  ➤ Removes an app from the list.
- launchstart
  ➤ Manually triggers all startup apps.
- ✅ Apps are launched using launch under the hood and appear in windows on desktop boot.

🛠️ Developer Notes

- The startup system is designed to be modular:
  - Use runbridge for command-line sequences.
  - Use amend start for GUI apps.
  - Use livebridge for external triggers or app events.
- runbridge and livebridge are JSON arrays stored in localStorage, parsed and executed in order.
- You can mix both startup commands and GUI apps for a seamless login experience

📦 Intnix App Metadata and .app File Structure

This section outlines how applications in Intnix are packaged, launched, and described using metadata. All apps in Intnix are stored as .app files and linked to JSON metadata for display, launching, and system integration.

📁 1. .app File Structure

Apps are stored in the file system as .app files (e.g., /apps/Photos.app). These are **JSON-structured wrappers** that contain the app's runtime path, type, and name.

Example .app file content:

```
{
  "name": "Photos",
  "type": "app",
  "path": "/packages/photos.html"
}
```

This file tells the launcher which HTML file to load when opening the app.

🔧 2. appFileMeta — App Display Metadata

Stored in localStorage as:

```
localStorage["appFileMeta"]
```

Structure:

```
{
  "/apps/Photos.app": {
    "name": "Photos",
    "icon": "photos-icon.png",
    "type": "app"
  },
  "/apps/Editor.app": {
    "name": "Editor",
    "icon": "editor-icon.png",
    "type": "app"
  }
}
```

Used to:

- Display app names and icons in Finder and Docks
- Load metadata when using launch, pkgmgr, or createWindow()
- Retrieve icon/title during window generation

To manually update:

```
assign "/apps/MyApp.app" "custom-icon.jpg"
```

🚀 3. installpkg / pkg

- pkg adds apps to the global package list with metadata
- installpkg installs them into /apps as a .app file and registers metadata into appFileMeta

🧠 Best Practices

- Always ensure your .app file matches the real path of the app
- Use clear names and icons
- Use assign to customize icons after install

🧠 General Metadata and JSON Structures

This section documents **non-app metadata** stored in Intnix to manage state, files, UI, and user systems.

📄 1. File Content Storage

Stored under:

```
fileContent:/full/path/to/file.txt
```

Used for nano, touch, cat, etc.

🗂️ 2. fakeFileSystem

```
fakeFileSystem = {
  "/": ["apps", "desktop"],
  "/apps": ["Appstore.app"],
  "/desktop": ["notes.txt"]
};
```

Live in-memory structure of the file system per user.

👤 3. userFileSystems

```
localStorage["userFileSystems"] = {
  "admin": { ... },
  "owen": { ... }
}
```

Each user's full FS snapshot. Loaded on login.

🔐 4. userPasswords

```
localStorage["userPasswords"] = {
  "owen": "hashed-or-plaintext-password"
}
```

Used for authentication and folder protection.

🖼️ 5. User Icons

```
localStorage["userIcons"] = {
  "owen": "owen-icon.png"
}
```

🔁 6. runbridge / livebridge

```
localStorage["runbridge"] = ["runbridge", "signin owen", "desktop"]
localStorage["livebridge"] = ["launch /apps/editor.app"]
```

Command queues for boot-time and real-time automation.

🧩 7. recentFiles:user

```
localStorage["recentFiles:owen"] = ["/apps/editor.app", "/desktop/notes.txt"]
```

Used by recents, clearrecents, etc.

🪟 8. windowTracker

```
[
  { "id": "win_editor", "path": "/apps/editor.html", "time": 1681782391832 }
]
```

Tracks open apps/windows for session history.

🧠 Summary Table

Key	Description
appFileMeta	App icons, names, types
fileContent:/path	File contents
userFileSystems	Per-user directory trees
userPasswords	Stored passwords
userIcons	Login icons per user
runbridge / livebridge	Startup/live automation commands
recentFiles:user	Recent activity tracker
windowTracker	Open window/app history

📌 These JSON structures are the backbone of the WebKernel. Understanding them is key to developing tools, syncing UI state, or debugging system behavior.

📜 Intnix Command List (175 Total)

1. accessuser
2. aerodock
3. alias
4. amend
5. amendapp
6. amendbridge
7. amendlivebridge
8. amendstart
9. appdownload
10. applauncher
11. assign
12. back
13. blank
14. bottomdock
15. bridgestatus
16. bubbledock
17. cat
18. cd
19. checkdata
20. chmod
21. cleanup
22. clear
23. clearbridge
24. cleardata
25. cleardir
26. clearpkg
27. clearrec

28. clearrecents
29. closeapp
30. closedebug
31. closedesktop
32. closewindow
33. coder
34. colordir
35. commands
36. console
37. console-clear
38. console-filter
39. console-last
40. copy
41. copypaste
42. currentdir
43. cut
44. debug
45. delete
46. deleteapp
47. deleteaptregistry
48. deletedir
49. deletepkg
50. deletestart
51. deleteuser
52. desktop
53. dock
54. dockicon
55. dockmenu
56. dockpos
57. dockrefresh
58. dockstate
59. dockvis

60. donut
61. downloaddata
62. downloadpkg
63. echo
64. edit
65. editpkg
66. endloop
67. exitaccess
68. exitfullscreen
69. faze
70. fetchstring
71. find
72. finder
73. fullscreen
74. genlicense
75. getmessages
76. getpass
77. getuser
78. hideaerodock
79. home
80. htmlnano
81. installpkg
82. intnix
83. intnixcode
84. intnixdock
85. launch
86. launchhtml
87. launchquicknote
88. launchstart
89. launchtext
90. link
91. linkvar

92. listlicenses
93. loading
94. loadwindows
95. log
96. loginchime
97. loop
98. ls
99. man
100. memory
101. minimize
102. mkdir
103. mkfile
104. movedir
105. movefile
106. nano
107. neofetch
108. newline
109. oos
110. openfinder
111. openmail
112. openwin
113. openwindows
114. openyt
115. oplus
116. overwrite
117. paste
118. pkg
119. pkgmgr
120. printvar
121. pwd
122. quicknote
123. recents

124. refresh
125. reload
126. rename
127. repkg
128. reset
129. resetvar
130. retrodock
131. returnmin
132. revealpass
133. rm
134. rmbridge
135. rmdup
136. run
137. runbridge
138. runlink
139. runvar
140. savedockstate
141. savewindows
142. scan
143. seedata
144. sendmessage
145. setback
146. setpass
147. settings
148. setuser
149. setvar
150. showvar
151. shutdown
152. sidedock
153. signin
154. signout
155. sleep

156. stopdonut
157. stoploop
158. stopmusic
159. string
160. sudo
161. systemcommand
162. time
163. topcolor
164. touch
165. up
166. updatesys
167. uploadpkg
168. user
169. version
170. view
171. viewdir
172. viewpkg
173. viewstart
174. webpage
175. whoami
176. windowterminal
177. winmgr
178. zip