Welcome to OopisOS v4.0

So, What Is This Thing?

You've got OopisOS. At its core, it's a self-contained operating system that runs entirely on your local machine. That's the most important part. There's no cloud, no server, no telemetry. All your files, your users, and your data are stored locally and only locally.

The whole OS is designed to be truly portable. You can stick it on a USB drive and carry your entire environment with you, leaving nothing behind on the host machine.

What It Actually Does: The Features That Matter

Forget the buzzwords. Here's what you can actually do with it.

- **Local AI, Your Rules:** The `gemini` command is your gateway to AI. You can chat with local models you run yourself via Ollama or LM Studio, or use the default provider which is smart enough to use other commands to answer questions about your files. It's a tool, not a gimmick.
- **A Real Filesystem (in a Box):** It's a persistent virtual filesystem running on IndexedDB, but you don't need to care about that. What you care about are the commands, and you've got the standard toolkit: `ls`, `cp`, `mv`, `rm`, `find`, `zip`—the works.
- **A Shell That Doesn't Suck:** It's a proper command-line environment. You get command history, tab-completion, piping (`|`), redirection (`>`), background tasks (`&`), and environment variables. You can even customize your prompt. All the basics you'd expect are here.
- **Actual Multi-User Security:** You can create users (`useradd`) and groups (`groupadd`), manage file permissions (`chmod`, `chown`), and escalate privileges correctly with `sudo`. This isn't just for show; permissions are enforced everywhere.
- **A Suite of Built-in Apps:** There's a set of useful, full-screen applications for when the command line isn't enough:
    - `edit`: A text editor with live Markdown preview.
    - `paint`: A character-based art studio. Yes, really.
    - `chidi`: An AI tool for analyzing collections of documents.
    - `adventure`: A surprisingly powerful engine for playing and building text adventure games.
    - `basic`: A complete IDE for the BASIC programming language.
    - `log`: A personal, timestamped journaling system.

Command-Line Crash Course

You're going to live in the terminal. You might as well get good at it. Here are the absolute essentials. Type `man [command]` if you need more than this.

Command	What It *Really* Does
`help [cmd]`	Shows you a list of commands or the basic use of one. Start here.
`ls`	Lists what's in a directory. Use it constantly.
`cd [directory]`	Changes your current directory. Your primary way of moving around.
`pwd`	"Print Working Directory." Shows you where you are.
`cat [file]`	Dumps a file's entire content to the screen. Good for a quick look.
`mkdir [dir_name]`	Makes a new, empty directory.
`touch [file]`	Creates an empty file or updates its timestamp.
`mv [source] [dest]`	Moves or renames a file. Same command, different result based on `dest`.
`cp [source] [dest]`	Copies a file.
`rm [item]`	Deletes a file or directory. **There is no trash can. This is permanent.**
`grep [pattern] [file]`	Finds lines containing a text pattern within a file. Incredibly useful.

The Application Suite: Tools for Getting Things Done

These aren't your typical bloated applications. They are focused, full-screen tools for specific jobs.

- **`explore [path]`**: For when you're tired of `ls` and `cd`. This gives you a graphical, two-pane file explorer. A directory tree on the left, file listing on the right. It's an intuitive way to see the lay of the land. Use it, get your bearings, then get back to the real work in the terminal.
- **`edit [file]`**: Your workhorse for text. It's smarter than `cat` and more powerful than `echo`. It handles plain text, but its real value is the live preview for Markdown and HTML (`Ctrl+P`). Stop guessing what your markup looks like and just *see* it.
- **`paint [file.oopic]`**: Look, sometimes you need to draw. This is a character-based art studio. It's got a canvas, colors, and tools. You can make icons, title screens for your scripts, or maps for the `adventure` engine. It's surprisingly capable. Don't knock it 'til you've tried it.
- **`chidi [path]`**: The "AI Librarian". Point it at a directory of Markdown files, and it gives you a dedicated interface to read and analyze them. You can ask it to summarize a document or find specific information across the entire collection. It's for deep-diving into a known set of files, unlike `gemini` which is for general discovery.
- **`adventure [file.json]`**: A data-driven text adventure engine. You can play the built-in game or, more importantly, create your own worlds by defining rooms, items, and logic in a JSON file. It's a powerful way to see how a complex, stateful application works within the OS.
- **`gemini "<prompt>"`**: This is your AI multitool. By default, it's smart enough to use other commands (`ls`, `cat`, `grep`) to answer questions about your files. Or, you can point it at a local model (`-p ollama`) for direct chat. It bridges the gap between natural language and shell commands.
- **`basic [file.bas]`**: A full IDE for the BASIC programming language. It's a throwback, but it's a complete, sandboxed environment where you can `LIST`, `RUN`, `SAVE`, and `LOAD` old-school, line-numbered programs. It even has a secure bridge to interact with the main OS (`SYS_CMD`, `SYS_READ`, `SYS_WRITE`) so your BASIC programs are still subject to the system's permission model.
- **`log`**: A simple, secure journaling system. It creates timestamped Markdown files in `~/.journal/`. The app gives you a timeline and an editor, but at the end of the day, they're just text files. You can `grep` them, `cat` them, or back them up like anything else.

Advanced Course: How to Actually Control the Thing

Now we're talking. These are the tools for administrators and people who want to make the system their own.

Shell Customization: The `PS1` Variable

Your command prompt doesn't have to be boring. The `PS1` environment variable controls its appearance. Use `set` to change it.

Sequence	What It Does
`\u`	Your username.
`\h`	The hostname (`OopisOs`).
`\w`	The full path of the current working directory.
`\W`	Just the basename of the current directory.
`\$`	A `#` if you're `root`, otherwise a `$`.

Example: `set PS1="[\u@\h \W]\$ “`
This changes the prompt to `[Guest@OopisOs ~]$`.

Privilege Escalation: `sudo` and `visudo`

Don't run as `root` all the time. It's stupid and dangerous. When you need administrative power, borrow it with `sudo`.

- **`sudo [command]`**: Executes a single command with `root` privileges. You'll be prompted for *your own* password, not the root password. The system checks if you're allowed to run the command based on the `/etc/sudoers` file.
- **`visudo`**: The *only* way you should edit `/etc/sudoers`. It locks the file and checks for syntax errors on save, which stops you from making a typo that locks everyone (including yourself) out of `sudo`. A very, very good idea.

Command Chaining: `&&` and `||`

The shell is smart enough to handle basic logic. This is essential for scripting.

- **`&&` (AND)**: The command on the right runs **only if** the command on the left succeeds.
  - **Use case:** `mkdir new_dir && cd new_dir` (Only change into the directory if the creation was successful).
- **`||` (OR)**: The command on the right runs **only if** the command on the left *fails*.
  - **Use case:** `grep "ERROR" log.txt || echo "No errors found."` (Print a success message only if grep finds nothing and returns an error code).

For Developers: How Not to Make a Mess

If you want to contribute, you need to understand the architecture. It's not complicated, but it is deliberate. The entire system is designed around a few core ideas. Don't fight them.

1. **It's All On The Client.** There is no backend. There is no server to save you. The OS is 100% self-reliant, and all data lives and dies in the user's browser. This is a hard constraint.
2. **Modularity is Not Optional.** Features are built as discrete, isolated components. The command executor *orchestrates* the filesystem manager; it doesn't get tangled up in its internals. This separation is what keeps the system from turning into a bowl of spaghetti.
3. **Security is the Foundation, Not a Feature.** The permission model is not a suggestion. All I/O, without exception, goes through a single gatekeeper (`FileSystemManager.hasPermission()`). Passwords are never stored in plaintext; they're hashed with the Web Crypto API. There are no shortcuts.
4. **Execution is Contained.** Every command follows a strict lifecycle: Lex, Parse, Validate, Execute. We validate everything—arguments, paths, permissions—*before* a single line of the command's core logic is run. This prevents a badly written command from taking down the whole system.

The "El Código del Taco" Architectural Model

This is the simplest way to understand the system. It's a taco. Each layer has one job.

Layer	Ingredient	Responsibility	OopisOS Implementation
1	**The Protein**	**Core Logic.** The actual work.	`commexec.js`, `fs_manager.js`
2	**The Lettuce**	**Presentation Layer.** The UI.	`terminal_ui.js`, `editor_manager.js`
3	**The Cheese**	**Features.** Valuable, but not essential.	`BasicInterpreter.js`, `gemini.js`
4	**The Salsa**	**API & Data.** The single source of truth for storage.	`storage.js` (IndexedDB)
5	**The Onions**	**Utilities.** Helpers that don't fit elsewhere.	`utils.js`
6	**The Jalapeño**	**Security.** The gatekeepers.	`sudo_manager.js`, `fs_manager.js`
7	**The Fold**	**Build & Deploy.** The final assembly.	`index.html`

The point is, you don't mix the ingredients. The presentation layer doesn't know how the filesystem works, it just knows how to display what it's given.

Adding a New Command: The Command Contract

This is the most important part for contributors. Adding a command is a declarative process. You don't just write code; you write a *contract* that tells the `CommandExecutor` what your command needs to run safely.

The executor handles all the tedious and error-prone validation *for you*.

Step 1: Create the Command File

Make a new file in `/scripts/commands/`. The filename must match the command name (e.g., `mycommand.js`).

Step 2: Define the Contract

Create an object that defines your command's requirements.

```
// scripts/commands/mycommand.js
const myCommandDefinition = {
  commandName: "mycommand",
  // What flags does it accept?
  flagDefinitions: [
    { name: "force", short: "-f" },
    { name: "output", short: "-o", takesValue: true }
  ],
  // How many arguments are required?
  argValidation: {
    min: 1,
    max: 2,
    error: "Usage: mycommand [-f] [-o file] <source> [destination]"
  },
  // Which arguments are file paths?
  pathValidation: [
    { argIndex: 0, options: { expectedType: 'file' } },
    { argIndex: 1, options: { allowMissing: true } }
  ],
  // What permissions are needed for those paths?
  permissionChecks: [
    { pathArgIndex: 0, permissions: ["read"] }
  ],
  // Finally, the logic.
  coreLogic: async (context) => { /* ... */ }
};
```

Step 3: Write the Core Logic

Your `coreLogic` function receives a `context` object. By the time your code runs, you can *trust* that everything in this object has already been validated according to your contract.

```
coreLogic: async (context) => {
  const { args, flags, currentUser, validatedPaths } = context;

  // No need to check permissions or if the path is valid.
  // The CommandExecutor already did it. Just do the work.
  const sourceNode = validatedPaths[0].node;
  const content = sourceNode.content;

  // ... your logic here ...

  return { success: true, output: "Execution complete." };
}
```

This design makes the system robust. It's hard to write an insecure command because the security is handled for you before your code even runs.

The Testing Environment: Don't Ship Broken Code

A new OS is an empty canvas. That's boring and hard to test. Use these scripts.

- **`run /extras/inflate.sh`**: This builds a whole world for you. It creates a complex directory structure with different file types, permissions, and even some secrets to find. Use it to test your commands in a realistic environment.
- **`run /extras/diag.sh`**: This is the gauntlet. It's a comprehensive stress test that runs a barrage of commands to check every corner of the OS. If your change breaks `diag.sh`, fix it. No excuses.

The Text Editors: `edit` and `code`

`edit`: The Main Workhorse

This is your primary text editor. You launch it by typing `edit [filepath]`. If the file exists, it opens it. If it doesn't, it'll create it when you save (`Ctrl+S`).

Its main purpose is to be adaptive. It's not just a dumb text box; it changes based on what you're working on.

- **Markdown Mode (`.md`)**: This is where `edit` shines. It has a live preview. You can cycle through views: editor only, a split view with your code and the rendered output side-by-side, or just the preview. Stop writing markup blind.
- **HTML Mode (`.html`)**: Same deal as Markdown. You get a live, *sandboxed* preview. The sandboxing is important—it means the HTML is rendered in a clean `iframe` so it doesn't mess with the rest of the OS UI.
- **Text Mode (everything else)**: For any other file (`.txt`, `.js`, `.sh`), it's a clean, standard text editor. It has a word-wrap toggle that actually saves its state, which is a nice touch.

Key Shortcuts (Memorize these):

- `Ctrl+S`: Save
- `Ctrl+O`: Exit
- `Ctrl+P`: Cycle through view modes (Edit/Split/Preview)
- `Ctrl+Z`: Undo
- `Ctrl+Y`: Redo

`code`: The Scalpel

The `code` command is for quick, surgical edits. It's not a feature-rich IDE. It's for when you know exactly what line you need to change in a script and you want to get in and get out without any fuss.

Type `code [filepath]` and it pops a simple modal with a text area. It has basic JavaScript syntax highlighting—it knows about comments, keywords, and strings. That's it. It's not meant to be pretty; it's meant to be clear.

Don't use `code` to write a novel. Use it to fix a typo in a script. Use `edit` for anything more substantial. One is a scalpel, the other is a workshop.

So, now that you know how to write and edit files, shall we dig into how the AI tools (`chidi` and `gemini`) actually use them?

The AI Tools: `chidi` and `gemini`

You have two primary AI commands. They are not the same. Don't use them for the same tasks.

`chidi`: The AI Librarian for Deep Dives

Think of `chidi` as a specialist. Its job is to perform deep analysis on a set of documents you already have. You give it a directory, and it launches a dedicated reading application where you can work with that specific collection of files.

```
chidi /path/to/your/docs
```

Once you're in the app, you can ask the AI to **summarize** a specific file, generate **study questions**, or—most importantly—**ask a question across all the documents** you loaded.

This is its key feature. It doesn't just dumbly send everything to the cloud. It uses a Retrieval-Augmented Generation (RAG) strategy:

1. It performs a local keyword search across your files to find the most relevant ones.
2. It then constructs a focused prompt containing *only* the content of those relevant files.
3. Finally, it sends that lean, relevant context to the AI for an answer.

This is a smart design. It's faster, cheaper, and gives you better answers because it's not wading through irrelevant garbage. You can even pipe file paths into it (e.g., `find . -name "*.md" | chidi`) to create a dynamic knowledge base on the fly.

`gemini`: The General-Purpose AI Assistant

If `chidi` is a specialist, `gemini` is your generalist. It's a conversational assistant that lives in your terminal and can use other OS tools to figure things out. It's designed to answer questions when you *don't* know where the information is.

It operates in two modes:

1. **Tool-Using Agent (Default)**: This is the interesting part. When you ask a question like `"Which of my text files contain the word 'OopisOS'?"`, it doesn't just guess. It follows a two-step process:
    - **Planner**: First, the AI looks at your question and your current directory and formulates a plan of whitelisted shell commands (`ls`, `grep`, `cat`, etc.) to find the answer. This is a critical security feature; it can't just run `rm -rf`.
    - **Synthesizer**: After the OS securely executes those commands, the AI takes the output and formulates a final, human-readable answer.
2. **Direct Chat (`-p` flag)**: If you just want to talk to an AI for creative tasks or general knowledge, you can use the `-p` flag to specify a provider, like a local Ollama instance (`gemini -p ollama "write a story"`). In this mode, it's just a direct conversation without the tool-use logic.

The Bottom Line: Which One Do I Use?

- **Use `chidi` when:** You have a collection of documents and you want to analyze, summarize, or query their content. You know *where* the knowledge is, you just need help processing it.
- **Use `gemini` when:** You have a question and you want the OS to figure out how to answer it. You want to find files, search for content across the whole system, or get a summary of things it discovers on its own.

One is a librarian for your personal library; the other is a research assistant you can send out into the stacks.

`paint`: The Character-Based Art Studio

Yes, it's a paint program. For the terminal. Get over it. It exists to solve a real problem: creating visual assets within the OS without needing to import external files. It's for making title screens for your scripts, icons for your files, or maps and character portraits for the `adventure` engine. It's a tool that embraces the aesthetic of the system instead of fighting it.

How It Works

You invoke it with `paint [filename.oopic]`. If the file exists, it loads it. Otherwise, you get a blank canvas that saves to that name.

The interface is simple and gets the job done:

- **Toolbar:** Has everything you need and nothing you don't. Pencil, eraser, line and shape tools, a color palette, brush size, and a grid toggle.
- **Canvas:** A fixed 80x24 grid. You "draw" by placing characters with specific colors.
- **Status Bar:** Gives you real-time feedback. No guessing what tool you have selected.

It has multi-level undo/redo (`Ctrl+Z`/`Ctrl+Y`), which frankly makes it more robust than some "professional" tools I've seen.

The Technical Part (The Only Part That Really Matters)

The implementation is a clean example of the OS's design philosophy.

- **Separation of Concerns:** The logic is split cleanly. `PaintManager.js` is the brain; it handles the application state—the canvas data (which is just a 2D array), the selected tool, the color, and the undo stack. `PaintUI.js` is the hands; its only job is to touch the DOM. It renders the canvas and sends user input back to the manager. They don't meddle in each other's business.
- **The Canvas Isn't a `<canvas>`:** This is the clever bit. The canvas is not a single `<canvas>` element. It's a CSS grid of individual `<span>` elements. This is far more efficient for this use case. Updating one character means changing one `<span>`, not redrawing a whole bitmap. It's also what makes the "ANSI" style art possible, with a foreground color for each character cell.
- **The `.oopic` File Format:** The artwork is saved to a custom `.oopic` format. It's just JSON. It stores the dimensions and a 2D array of the cells (character and color). This is intentional. It's transparent, human-readable, and you could even edit the art with `cat` and `edit` if you were so inclined. It's an open format for an open system.

`paint` isn't an afterthought. It's a first-class citizen of the OS that proves you can build powerful, creative tools within a limited, text-based paradigm. It's integrated, it's useful, and its architecture is sound.

The Adventure Engine: A Study in Data-Driven Design

The `adventure` command launches a powerful engine for interactive fiction. You can play the built-in game, but the real value is in understanding how to build your own worlds.

How to Play

This is the simple part.

- `adventure`: Starts the default game.
- `adventure /path/to/my_game.json`: Loads a custom game from a JSON file.

Inside the game, type `help` for a list of commands (`look`, `take`, `use`, etc.). Type `quit` to exit.

How to Build

This is what matters. You can build your own adventure without writing a single line of JavaScript. You just need to create a `.json` file that describes your world.

To start, use the creation tool: `adventure --create my_game.json`. This drops you into an interactive editor that helps you build the JSON structure.

The Anatomy of an Adventure (.json)

The entire game state is defined by a handful of key objects in your JSON file.

Rooms

These are the locations in your world. Each room has a unique ID, a name, a description, and a list of `exits` that link to the IDs of other rooms.

```
"test_chamber": {
  "name": "Test Chamber",
  "description": "You are in a room that feels... unfinished.",
  "exits": { "north": "server_closet" },
  "isDark": false
}
```

You can also add sensory details like `onListen` or `onSmell` for extra flavor.

Items

These are the objects that populate your rooms. The engine's parser is smart enough to understand nouns and adjectives, so you can `take brass key` instead of just `take key`.

The `location` property determines where an item starts: in a room, in another item (a container), or directly in the `"player"`'s inventory.

```
"key": {
  "id": "key",
  "name": "brass key",
  "noun": "key",
  "adjectives": ["brass", "small"],
  "description": "A small, plain brass key.",
  "location": "test_chamber",
  "canTake": true,
  "unlocks": "chest"
}
```

Stateful and Interactive Items

Items aren't just static. They can have states (`"on"`/`"off"`) with different descriptions for each state. They can be containers (`isContainer: true`) that can be opened and closed.

Most importantly, you can define complex interactions using `onPush`, `onUse`, etc. This is how you build puzzles. Pushing a lever can change its own state and trigger an `effect` that changes the state of another item, like turning on a terminal.

```
"power_box": {
  "id": "power_box",
  "state": "off",
  "onPush": {
    "newState": "on",
    "message": "You push the heavy lever. It clunks into the 'ON' position.",
    "effects": [
      { "targetId": "terminal", "newState": "on" }
    ]
  }
}
```

NPCs and Daemons

You can add Non-Player Characters (`npcs`) with branching `dialogue` trees and reactions to items you `show` them. You can also create `daemons`—timed events that trigger messages or actions after a certain number of turns, which is perfect for providing hints to a stuck player.

The Point

The adventure engine is a self-contained system that demonstrates how to build a complex, interactive application with a clean separation between the engine's code and the world's data. It’s a practical example of the design philosophy that underpins the entire OS. Study it.

Oopis Basic: A Sandboxed Scripting Environment

Before you ask, yes, it's an implementation of the BASIC programming language. It's not here for nostalgia; it's here to demonstrate a core architectural principle: **secure, sandboxed code execution**.

The IDE (`basic` command)

When you type `basic [file.bas]`, you're not just running a script. You're launching a full-screen Integrated Development Environment (IDE). This is a presentation-layer component (`basic_app.js`) that handles the UI, manages the program you're writing (`LIST`, `SAVE`, `LOAD`), and gives you the commands to control execution (`RUN`, `NEW`, `EXIT`).

It's a focused environment. When you're in it, you're just dealing with your BASIC program.

The Language and The Engine

The language itself is simple, line-numbered BASIC. You have `PRINT`, `INPUT`, `GOTO`, `GOSUB`/`RETURN`, and `IF...THEN`. It's straightforward.

The interesting part is the engine, `BasicInterpreter.js`. This is a self-contained parser and executor. Crucially, it has **no direct access to the file system or the main command executor**. This is the sandbox. A `GOTO 10` loop in a BASIC program can't crash the host OS because it's running in an isolated environment.

The Secure Bridge: `SYS_` Functions

So how does a sandboxed program do anything useful? Through a secure, controlled bridge to the host OS. The interpreter provides a specific set of system functions that a BASIC program can call:

- **`SYS_CMD("command")`**: Executes an OopisOS shell command and returns its output as a string.
- **`SYS_READ("filepath")`**: Reads the content of a file from the virtual file system.
- **`SYS_WRITE("filepath", "content")`**: Writes content to a file.

When your BASIC program uses one of these, the interpreter doesn't just execute it. It bundles up the request and passes it through the main `CommandExecutor` and `FileSystemManager`. This means any action your BASIC program attempts is still subject to the OS's fundamental permission model. A `Guest` user running a BASIC script still can't read files in the `/home/root` directory, because the request is ultimately handled and validated by the secure core of the OS.

That is the entire point. It's not just a feature; it's an architectural statement on how to safely grant power to a subsystem without compromising the integrity of the whole.

`log`: The Captain's Journal

The `log` command is your personal journal. It's simple, secure, and built on the principle that your data is your own. When you run `log`, it opens a clean, two-pane application: a timeline of your entries on the left and an editor on the right.

The most important architectural point is that there is no complex database behind it. Each entry is just a separate Markdown file stored in `~/.journal/`. This is a deliberate design choice. It means you can use the entire OopisOS toolchain on your journal. You can `grep` for entries, `cat` them to the terminal, or back them up with `zip`. The application is just a convenient front-end for managing a directory of text files. It's a perfect example of leveraging the core OS features instead of reinventing the wheel.

You can also make a quick entry directly from the command line:

```
log "This is a new entry."
```

It's a simple tool that does one job well, and it does it by building on the foundation of the filesystem. That's good design.

`explore`: The Graphical File Explorer

The `explore` command is for when a visual overview is faster than typing `ls -R`. It opens a two-pane file explorer: a directory tree on the left, and the contents of the selected directory on the right.

It's a read-only tool. Its purpose is orientation. Use it to quickly navigate a complex directory structure, find what you're looking for, and then get back to the command line where the real work happens. It respects all file permissions, so you'll only see what you're allowed to see.

OopisOS: Security by Design

OopisOS is architected on a principle of zero-trust, ensuring security by default, not by effort. The system's security is not a single feature, but a series of interlocking components that govern every action from authentication to file access.

The Bedrock

The security model is built on three pillars: **client-side sandboxing**, **explicit user permissions**, and **architected containment**. The system has no servers, collects no user data, and has no access to local files beyond what the user explicitly provides.

The Core Model: How It Works

Every security-sensitive action is funneled through audited, single-purpose managers.

- **Authentication (`UserManager` & `passwd`):** User passwords are never stored in plaintext. They are hashed using the browser's native Web Crypto API with SHA-256, as seen in `user_manager.js`'s `_secureHashPassword` function. The `passwd` command provides the user-facing interface for changing passwords, invoking `UserManager.changePassword` to orchestrate the secure update process.
- **Authorization (`FileSystemManager`):** This component is the sole gatekeeper for all file system operations. As implemented in `fs_manager.js`, every attempt to read, write, or execute a file is validated through the `FileSystemManager.hasPermission(node, username, permissionType)` function. This function rigorously checks the file's owner and group against its octal permissions (`rwx`). The only exception is the `root` user, who bypasses these checks.
- **Privilege Escalation (`SudoManager` & `visudo`):** The `sudo` command allows for temporary, controlled privilege escalation. Access is governed by the `/etc/sudoers` file, which is parsed by the `SudoManager`. The `visudo` command ensures this file is edited safely, setting its mode to a read-only `0o440` upon saving to prevent unauthorized modification. Privileges granted via `sudo` are temporary and scoped to the specific command executed.

Your Security Toolkit: Data Verification and Protection

OopisOS provides a suite of command-line tools for data integrity and security.

Command	Role in Security	Implementation Details
`cksum`	**Verification**	Calculates a 32-bit CRC checksum and byte count for a file's content. This is used to verify that a file has not been altered or corrupted since its last check.
`base64`	**Transformation**	Encodes and decodes data using the Base64 standard, utilizing the browser's native `btoa()` and `atob()` functions. This is essential for safely transmitting binary data through text-only systems.
`ocrypt`	**Encryption**	Provides strong, password-based file encryption using the modern AES-GCM standard, implemented via the Web Crypto API. This is the recommended tool for securing sensitive files.
`xor`	**Obscurity**	A simple password-based XOR cipher. **This is not secure encryption.** It is included as an educational tool to demonstrate basic data transformation principles.
`sync`	**Persistence**	Manually forces all pending filesystem changes to be written from memory to the underlying IndexedDB database by calling `FileSystemManager.save()`.

This suite embodies the OopisOS philosophy: providing the user with transparent and robust tools to manage their own data security.

Command Reference: The Toolbox

This is not an exhaustive guide. It's a quick reference. For the full, excruciating detail on any command, use `man [command_name]`.

1. Observation & Security: Look Before You Leap

You can't manage what you can't see. These are the foundational tools for observing the state of the system and its rules.

Command	What It *Actually* Does
`ls`	**Lists directory contents.** Use it constantly. `ls -l` provides a detailed long format, while other flags sort by time (`-t`), size (`-S`), or reverse order (`-r`).
`tree`	**Lists contents in a tree-like format.** It's `ls` for people who like diagrams. Use `-L <level>` to limit depth or `-d` for directories only.
`pwd`	**Prints the working directory.** Tells you where you are. If you're lost, use this.
`diff`	**Compares two files line by line.** Shows you exactly what changed. Invaluable.
`df`	**Reports filesystem disk space usage.** Shows you how much space you've got left in the virtual disk. Use `-h` for human-readable sizes.
`du`	**Estimates file space usage.** Shows you how much space a specific file or directory is taking up. Supports `-h` for human-readable and `-s` for a summary.
`chmod`	**Changes the permission mode of a file.** The core of file security. Use 3-digit octal modes (e.g., `755`). If you don't know what that means, you shouldn't be using it.
`find`	**Searches for files.** A powerful tool to find files based on name, type, permissions (`-perm`), and modification time (`-mtime`). The all-seeing eye of the filesystem.
`cksum`	**Calculates a checksum for a file.** Verifies that a file hasn't been corrupted. If the numbers match, the file is the same.

2. User & Group Management: The Social Contract

Now that you can see, you need to manage who's who. This is about defining the actors in your security model.

Command	What It *Actually* Does
`useradd`	**Creates a new user account.** Also creates their home directory. Prompts for a password via a modal input.
`removeuser`	**Deletes a user account.** Use `-r` to also delete their home directory and all their files. Be careful with this one.
`groupadd`	**Creates a new user group.** For managing permissions for multiple users at once. Requires `root` privileges.
`groupdel`	**Deletes a group.** You can't delete a group if it's the primary group for any user. Requires `root` privileges.
`usermod`	**Modifies a user's group memberships.** Its only supported use here is `usermod -aG <group> <user>` to add a user to a supplementary group.
`passwd`	**Changes a user's password.** If you're not `root`, you can only change your own, and you'll need to provide the old one.
`chown`	**Changes the user ownership of a file.** Only the file's owner or `root` can do this.
`chgrp`	**Changes the group ownership of a file.** Same rules as `chown`.
`sudo`	**Executes a command as root.** The safe way to get administrative privileges for a single command, governed by `/etc/sudoers`.
`visudo`	**Safely edits the `/etc/sudoers` file.** The *only* way you should touch this file. It prevents you from locking yourself out and secures the file on save.
`login`	**Logs in as a different user.** This ***replaces*** your current session.
`logout`	**Logs out of a stacked session.** This is the counterpart to `su`. Use it to return to your original user.
`su`	**Switches to another user.** Stacks a new session on top of your current one. Default is `root`.
`whoami`	**Prints your current username.** In case you forget.
`groups`	**Displays group memberships.** Shows you which groups a user belongs to.
`listusers`	**Lists all registered users.** A quick way to see who has an account on the system.

3. The Workshop: Fundamental File Operations

These are the tools you'll use every day. They are simple, sharp, and do exactly what they say they do.

Command	What It *Actually* Does
`mkdir`	**Makes a new directory.** Use the `-p` flag to create parent directories as needed, which saves you from creating them one by one.
`rmdir`	**Removes *empty* directories.** If there's anything in it, this command will fail. This is a safety feature. Use it.
`touch`	**Creates an empty file** or updates the timestamp of an existing one using `-d` for date strings or `-t` for stamps.
`echo`	**Writes its arguments to the output.** Its main purpose is to write text into files using redirection (`>`). Supports `-e` to enable backslash escapes like `\n` and `\t`.
`cat`	**Concatenates and displays file content.** Dumps the entire contents of a file to the screen. Use `-n` to number the output lines.
`head`	**Outputs the first part of a file.** By default, the first 10 lines. Use `-n` for lines or `-c` for bytes.
`tail`	**Outputs the last part of a file.** The opposite of `head`. Essential for checking the end of log files. Also supports `-n` and `-c`.
`cp`	**Copies files or directories.** Use `-r` for directories, `-p` to preserve metadata, and `-i` to be prompted before overwriting.
`mv`	**Moves or renames files and directories.** Same command, different result based on whether the destination exists. Supports `-i` for interactive and `-f` to force overwrites.
`rm`	**Removes (deletes) files or directories. There is no undelete.** Use `-r` for directories and `-f` to force deletion without prompting. Be extremely careful. You've been warned.
`zip`	**Creates a simulated `.zip` archive.** Bundles a file or directory into a single JSON file representing the file structure.
`unzip`	**Extracts a simulated `.zip` archive.** The counterpart to `zip`, recreating the archived directory structure.
`upload`	**Uploads files from your real machine** into the OS. Opens a file dialog. Use `-r` to upload an entire directory.
`export`	**Downloads a file from the OS** to your real machine.

4. The Assembly Line: Text Processing & Automation

This is where the real power of a command-line OS comes from. These tools are designed to be chained together with pipes (`|`) to perform complex data manipulation.

Command	What It *Actually* Does
`grep`	**Finds lines that match a pattern.** The single most useful text processing tool you have. Supports `-i`, `-v`, `-n`, `-c`, and recursive (`-R`) search.
`sort`	**Sorts lines of text.** Alphabetically by default, or numerically with `-n`. Use `-r` to reverse and `-u` for unique lines.
`uniq`	**Filters out adjacent repeated lines.** Use `-c` to count, `-d` for only repeated lines, and `-u` for only unique lines. Useless without `sort` first.
`wc`	**Counts lines (`-l`), words (`-w`), and bytes (`-c`).** Good for sanity checks.
`awk`	**A powerful pattern-scanning and text-processing language.** Use `-F` to specify a field separator and `{print $N}` to extract columns.
`more` / `less`	**Pagers to display content one screen at a time.** `less` is better because it lets you scroll backward (`b` or ArrowUp) and forward (`f` or Space).
`bc`	**An arbitrary-precision calculator.** Pipe an expression to it or provide it as an argument. Handles basic arithmetic and parentheses.
`xargs`	**Builds and executes command lines from standard input.** The glue that lets you use the output of one command as the arguments for another. Use `-I <str>` to replace a placeholder.
`run`	**Executes a shell script (`.sh` file).** The foundation of all automation, with argument support (`$1`, `$@`, `$#`).
`delay`	**Pauses execution** for a number of milliseconds. Essential for scripting demonstrations.
`base64`	**Encodes or decodes data.** For making binary data safe for text-based systems. Use `-d` to decode.

5. The Bridge: Networking & System Integrity

These commands are for interacting with the outside world and managing the state of the OS itself. Some of these are dangerous. Pay attention.

Command	What It *Actually* Does
`wget`	**Downloads a file from a URL.** A non-interactive downloader. Specify an output file with `-O`.
`curl`	**A more versatile data transfer tool.** Use it for API interaction, `-i` to see headers, `-o` for output file, and `-L` to follow redirects.
`ps`	**Lists current background processes** that you started with `&`. Shows the Process ID (PID) and the command.
`kill`	**Terminates a background process** by its Job ID (PID) from `ps`.
`backup`	**Creates a full, downloadable backup of the entire OS state.** This is your escape hatch. It includes a checksum for integrity.
`restore`	**Restores the OS from a backup file.** Wipes the current state completely. It will ask for confirmation.
`sync`	**Commits filesystem caches to persistent storage.** Forces a save of the in-memory `fsData` object to IndexedDB.
`reboot`	**Reboots the virtual machine** by reloading the page. All your data persists.
`reset`	**Wipes ALL OopisOS data and performs a factory reset.** This is the "burn it all down" command. It is permanent. It is destructive. Use it when you want to start over from nothing.

6. The Cockpit: High-Level Applications

These are the full-screen modal apps. We've covered them in detail, so this is just a quick reference.

Command	What It *Actually* Does
`edit`	**Opens the main text editor** with live previews for Markdown/HTML.
`paint`	**Launches the character-based art studio.** For creating assets with the `.oopic` extension.
`explore`	**Opens the graphical file explorer.** For a quick visual overview of your files.
`chidi`	**The "AI Librarian."** Launches a dedicated app for analyzing a collection of documents using a configured LLM provider.
`gemini`	**The general-purpose AI assistant.** Uses system tools to answer questions about your files or can interact directly with local LLMs.
`adventure`	**Starts the text adventure engine.** Lets you play or create interactive fiction using `.json` files. Use `--create` to enter the editor.
`basic`	**Launches the Oopis Basic IDE.** A sandboxed environment for writing and running `.bas` programs.
`log`	**Opens the personal journaling application.** A simple front-end for managing timestamped text files. Can also create quick entries from the command line.

7. The Environment: Shell & Session Control

These commands control the shell itself. Use them to customize your environment and manage your workflow.

Command	What It *Actually* Does
`help`	**Displays a list of commands** or a command's basic syntax. A quick reminder.
`man`	**Displays the detailed manual page** for a command. This is `help` with more words.
`history`	**Displays or clears the command history.** Use `-c` to clear it.
`clear`	**Clears the terminal screen.** Doesn't clear your history, just the clutter.
`alias`	**Creates command shortcuts.** `alias ll='ls -l'` is a classic for a reason.
`unalias`	**Removes an alias**.
`set`	**Sets or displays environment variables.** Use it to customize your `PS1` prompt.
`unset`	**Removes an environment variable**.
`date`	**Displays the current system date and time**.

FAQ: The Real Questions

What *is* this thing, really?

It's a simulated operating system that runs entirely in your browser. It's a persistent, private sandbox for you to work, play, and experiment in. There is no cloud. There is no server. All your data is stored on your machine, and only on your machine. We don't have it, we don't want it.

So my data is actually private?

Yes. See the point above. All files, user accounts, and session information are stored exclusively in your browser's `localStorage` and `IndexedDB`. It never leaves your computer unless you explicitly `export` or `backup` a file.

What's the `root` password?

It's `mcgoopis`. This is set during the initial user setup in `UserManager.initializeDefaultUsers`. Don't lose it. And don't do everything as `root`. That's just asking for trouble.

Why are some commands so slow?

Because your disk might be slow, or you're doing a lot at once. OopisOS's performance is directly tied to the read/write speed of the browser's `IndexedDB` storage. Operations that hit the virtual disk hard—like `find`, `unzip`, or `grep -R`—will be limited by the speed of this storage. That's not a bug; it's physics.

My Gemini API key disappeared when I switched computers. What gives?

That's a security feature, not a bug. Your API key is stored in your browser's `localStorage` using the key `oopisGeminiApiKey`, which is specific to that machine. It is *intentionally* not included in system backups created by the `backup` command to prevent you from accidentally sharing your private key. You'll need to re-enter your key the first time you use an AI command on a new machine.

Why can't `wget` or `curl` download from every website?

CORS (Cross-Origin Resource Sharing). Because OopisOS runs in a browser, it's subject to the browser's security rules, which are enforced via the `fetch` API. If a website's server doesn't explicitly send a header that allows your browser to request its data, the browser will block the request. This isn't a limitation of OopisOS; it's a fundamental security policy of the web.

Is this just Linux?

No. It's a *simulation* of a Unix-like operating system, implemented in JavaScript. Many of the commands are designed to behave like their real-world counterparts (`ls`, `grep`, `chmod`, etc.), but it is not a Linux kernel. It's a self-contained environment that provides a similar *experience* and enforces similar security principles, such as user and group permissions managed by `chown` and `chgrp`.

Who is this for?

It's for people who are curious. It's for developers who want to see how an OS can be architected in a non-traditional environment. It's for students who want a safe environment to learn command-line fundamentals without the risk of damaging a real system. It's for anyone who wants a private, portable sandbox to play in. If you're looking for a production-grade server to run your company's database, you're in the wrong place.

About & Credits

The Creators

This project was a collaboration between two entities:

- **Andrew Edmark**: The Human. The Curator. The one who provided the direction, did the testing, and assembled the final work.
- **Gemini**: The AI Assistant. The one that generated code, drafted documentation, and was generally responsible for the bugs that later became features.

The Social Contract (aka The Boring Legal Bit)

This is the MIT License, more or less. It lays out what you can and can't do. The important part is that this software is provided "AS IS", without warranty. If it breaks, you get to keep both pieces.

This Software, OopisOS, represents a collaborative endeavor between human direction and artificial intelligence.

Now, what are you still here for?

Go.
Explore.
Create.