Web-based Terminal For Docker

I have docker running on my server. I want someone to be able to access a container on my server by using a web browser. Within the web browser, there should be a terminal that allows a user to interact with a brand new container each time they open a new session.

The user should be able to type commands into the terminal, which should be executed by the container. The container responses should be visible in the terminal session. When the user types exit in the terminal, it should end the session and release the underlying resources on the server.

Assume that I have a Linux server. The end user should be able to use any modern web browser.

Please provide a detailed design and implementation plan for such a system. Please include an architecture diagram in your output.

To design the best solution for your use case, could you please clarify a few things:

1. What kind of container image should each new session use? (e.g., Ubuntu, Alpine, Python runtime, etc.)
2. Is authentication required for users accessing the terminal, or should anyone with the URL be able to start a session?
3. Are there resource constraints or limits you want on each container (CPU, memory, etc.)?
4. Do you have a preference for open source tools vs building from scratch?
5. How long should a session be kept alive if the user doesn't exit explicitly?

Once I have this info, I'll provide a full [...] architecture diagram and step-by-step implementation plan.

Skip to content

1. Ubuntu
2. Yes, username and password authentication
3. Yes, .5 CPU, 200mb RAM
4. Use open source tools to simplify and limit custom build
5. 15 minutes

Thanks! I'll put together a detailed design and implementation plan for a web-based terminal system that authenticates users, launches a new Ubuntu-based Docker container for each session with specific resource limits, and destroys the container upon exit or after 15 minutes. I'll include an architecture diagram and all setup instructions.

Web Terminal System Design for Docker-Backed Sessions

Overview

This guide describes a **web-based terminal system** that allows authenticated users to spawn ephemeral shell sessions in isolated Docker containers. Each user's browser connects to a terminal interface, backed by a temporary Ubuntu container with restricted resources. The design uses open-source components (Docker, web terminal libraries, etc.) to minimize custom development. Key features include:

- **On-Demand Containers:** A fresh Docker container (Ubuntu base image) launches for each new web terminal session.
- **Resource Limits:** Each container is limited to **0.5 CPU cores** and **200 MB RAM** using Docker's cgroup settings.
- **Browser Terminal UI:** Users interact with a real-time **terminal emulator** in their web browser, which streams input/output to the container.

- **Auto-Cleanup:** Containers termir Skip to content on user exit or **15 minutes of inactivity**, ensuring no orphaned containers.
- **User Authentication:** A login (username/password) is required before any terminal access, preventing unauthorized use.

By combining a web-based terminal front-end (like **Xterm.js**) with a back-end that manages Docker containers, we can deliver a secure, responsive shell experience accessible from any modern browser. The following sections detail the system architecture, component choices, setup steps, and security considerations.

Architecture Overview

Architecture Diagram – The web terminal system consists of a web server (with authentication and terminal proxy logic) connecting user browsers to on-demand Docker containers.

Components:

- **User Browser:** Runs the terminal front-end (JavaScript terminal emulator) over HTTPS.
- **Web Server & Terminal Gateway:** Handles user authentication and orchestrates Docker containers. It proxies WebSocket traffic between the browser's terminal and the container's shell.
- **Docker Engine:** Runs on the Linux server to launch and manage containers via Docker API/CLI.
- **Ephemeral Container:** An Ubuntu-based container providing a shell (`/bin/bash`). Each session gets its own container with enforced CPU and memory limits. The container is removed when the session ends.

Data Flow:

1. **User Login:** The user visits the web application. The **web server** (or a reverse proxy like Nginx) authenticates the user via **HTTP basic auth** or a login form (backed by an auth database). Only authenticated users can proceed to the terminal interface.

2. **Session Initialization:** After login [illegible]es the terminal page. The web front-end opens a WebSocket connection to the server (e.g., wss://server/terminal). The server then triggers Docker to **launch a new container** (using `docker run`) with the specified resource limits and an **interactive shell process**. The Docker **–cpus** and **–memory** flags enforce the 0.5 CPU and 200MB RAM limits GITHUB.COM . The container is started in attached mode (with a pseudo-TTY) but not yet connected to the user.
3. **Terminal Attachment:** The server attaches the WebSocket to the container's stdio. From this point, anything the user types in the browser is relayed to the container's STDIN, and the container's STDOUT/STDERR is streamed back to the browser in real-time. This is achieved using a terminal library (like **Xterm.js** on the client and Docker's attach API on the server). The Xterm.js library is a full-featured JavaScript terminal emulator that renders the shell session in the browser. Communication runs over WebSockets so that input/output is **real-time and interactive**.
4. **Inactivity Timer:** The server monitors activity. If **15 minutes** pass with no input (and optionally no output) on the session's WebSocket, the server will automatically terminate the container (as if the user typed "exit"). This uses an inactivity timer that resets on any user keystroke or command.
5. **Session Termination:** If the user types the `exit` command or the inactivity timer triggers, the shell process ends and Docker stops the container. A cleanup routine then removes the container (using `docker rm` or by running it with `--rm` so Docker auto-removes it). The WebSocket to the browser is closed, and the browser terminal UI is informed that the session ended. The user can then start a new session if needed (which creates a new container).

Chosen Tools and Technologies

Docker: Containerization platform to i… ons. We use the official Ubuntu base image for the container environment. Docker's resource limitation flags are used for CPU/RAM constraints. The Docker Engine API (or CLI) is invoked by the web server to spawn and manage containers. This ensures each terminal session is isolated at the OS level (namespaces, cgroups).

Web Terminal Front-End: Xterm.js is used to provide a terminal emulator in the browser. Xterm.js is an open-source JavaScript library that emulates an xterm-compatible terminal entirely in the browser. It can handle typical shell interactions (control characters, terminal resizing, etc.) and is widely used in projects like VS Code and Wetty. The front-end will use Xterm.js to display the shell and a small JavaScript to connect it via WebSocket to the back-end.

WebSocket Proxy Server: A lightweight web server (e.g. built with **Node.js** using libraries like `express` and `ws`, or Python with frameworks like FastAPI or Flask-SocketIO) acts as the bridge between Xterm.js and Docker. This server:

- Authenticates users (if not offloaded to a proxy)
- Launches a Docker container for the session (via Docker SDK or CLI)
- Attaches to the container's PTY (pseudo-terminal)
- Relays data between the container and browser over a WebSocket.

Using a WebSocket is crucial for interactive, bidirectional communication – it carries keystrokes from the browser to the container and shell output back to the browser in a low-latency fashion. Standard HTTP is insufficient for this real-time stream; WebSockets enable full-duplex communication suited for a terminal.

Authentication Layer: We want to **enforce login** before granting access. Options include:

- **Nginx Reverse Proxy** with HTTP Basic Auth: Nginx can sit in front of the web socket server. It serves the static HTML/JS for the terminal UI and requires a basic auth login for the `/terminal` path (or whole site). Only on correct credentials will it forward the connection upgrade to the backend. This is simple and leverages Nginx's proven auth module.

- **Built-in Auth in Web App:** Altern t a login page in the Node/Python server that checks credentials (pos... ...tpasswd file or a database). On success, issue a session cookie and allow access to the terminal page. This requires more custom code, but can integrate with existing user directories or OAuth if needed.

Open-Source Projects Inspiration: This solution draws on ideas from existing tools:

- *Wetty* – a web terminal that normally connects to a local shell or SSH. We adapt the concept by hooking it to a Docker container instead of a fixed shell. Wetty's use of **Xterm.js** and Node.js websockets is a reference model.
- *ttyd* – a command-line tool that exposes a terminal over web. We use a similar approach of combining a pty with websockets, but we need dynamic container spawning per session (ttyd alone doesn't manage new Docker containers per user).
- *JupyterHub's DockerSpawner* – an open-source system that spawns a Docker container per user for Jupyter notebooks. Our design is analogous but much lighter; instead of full Jupyter notebooks, we only run a shell. This demonstrates the feasibility of per-user containers managed by an authentication hub.

By using these well-tested components (Docker, Nginx, Xterm.js, etc.), we avoid writing everything from scratch. Next, we detail how to set up and configure each part of the system.

Setup and Configuration Steps

Follow these steps to implement and deploy the web-based terminal system:

1. Prepare the Linux Server and Docker

- **Install Docker:** Make sure Docker Engine is installed on the Linux server and the service is running. Also, ensure the user account running the web server has permission to access the Docker socket (e.g., add to the `docker` group or use a secure socket).
- **Pull Base Image:** Pull the Ubuntu image that will be used for containers (for example, `docker pull ubuntu:22.04`). This ensures the image is cached and ready. You may also create a custom image if you want to pre-install certain packages. For a basic shell, vanilla Ubuntu is fine.

- **Test Resource Limits:** It's good ' Skip to content 'ontainer launch to verify the CPU/RAM limits syntax, e.g.:

bash Copy

```bash
docker run -it --rm --cpus=0.5 --memory=200m ubuntu:22.04 /bin/bash
```

This should start an interactive bash shell with the limits applied (you can check with `docker stats` in another terminal). The `--rm` flag means Docker will clean up the container on exit. Use `exit` to quit the test container.

2. Set Up the Web Server (Terminal Gateway)

We will use **Node.js** for the implementation (Python could also be used similarly). The server will handle both HTTP(s) requests and the WebSocket connections for terminal data. Key steps:

a. Initialize Node Project:

- Install Node.js (≥ 18.x recommended) on the server.
- Create a project directory and run `npm init -y` .
- Install needed libraries:

bash Copy

```bash
npm install express express-ws dockerode xterm jsdom
```

  - `express` – to serve web pages and handle HTTP routes.
  - `express-ws` – to add WebSocket (ws) support to Express easily.
  - `dockerode` – a Node library for Docker API, to programmatically manage containers. (Alternatively, you could use the Docker CLI via `child_process.spawn` , but dockerode is cleaner.)
  - *xterm.js* (the front-end library) doesn't run on the server, but we might install its npm package or simply include via CDN in the HTML.

b. Write the Server Script: (High-level outline)

- **Import modules** and create an E Skip to content
- **Static Content:** Serve a static HTML page (and JS/CSS) for the terminal UI at the root or `/terminal`. This HTML will include a `<div id="terminal">` and load Xterm.js, plus a small script to connect to the WebSocket.
- **Authentication Middleware:** If not using an external auth (like Nginx), implement basic auth middleware. For example, use `express.basicAuth` or a custom check: compare `req.headers.authorization` to expected credentials (if using Basic Auth header) and respond with `401 Unauthorized` if not matching. Alternatively, use express-session and a login route (this is more code – offloading to Nginx Basic Auth is simpler).
- **WebSocket route:** Set up an Express-ws route, e.g. `app.ws('/terminal', (ws, req) => { ... });`. Inside this callback, you will spawn/attach to the Docker container for the session.
  - **Launch Container:** Use Dockerode to create and start a container. For example:

    js Copy

    ```js
    const container = await docker.createContainer({
      Image: 'ubuntu:22.04',
      Cmd: ['/bin/bash'],
      Tty: true,  // Allocate a pseudo-TTY
      AttachStdin: true,
      AttachStdout: true,
      AttachStderr: true,
      OpenStdin: true,
      HostConfig: {
        AutoRemove: true,       // auto-remove on exit (similar to --rm)
        CpuQuota: 50000,        // 50% of a single CPU (in microseconds of CPU time)
        Memory: 200 * 1024 * 1024 // 200MB in bytes
      }
    });
    await container.start();
    ```

The above sets up a contain Skip to content ɔash . We use **AutoRemove** so Docker cleans it up on stop (this is equivalent to `--rm`) – this covers orphan cleanup. CPU and Memory limits are set in HostConfig. (Note: Alternatively, use `docker.run()` in dockerode or spawn the `docker run` command; either approach works.)

- **Attach Streams:** After starting, attach to the container's STDIN/STDOUT. With Dockerode:

js Copy

```js
const attachOpts = { stream: true, stdin: true, stdout: true,
stderr: true };
const stream = await container.attach(attachOpts);
```

If `Tty: true` , Docker multiplexes stdout/stderr together similar to a real terminal GITHUB.COM , so we can treat it as a single stream. Now set up event handlers:

  - On WebSocket message from client: write data to `stream` (which goes into container's stdin).
  - On stream data from container: send it out over the WebSocket to the browser.
  - On stream end/close (container exited): close the WebSocket.
    For example:

js Copy

```js
ws.on('message', (msg) => {
  stream.write(msg);
});
stream.on('data', (chunk) => {
  ws.send(chunk);
});
stream.on('end', () => {
  ws.close();
});
```

You may also handle `error` Skip to content ctively **pipes the container I/O to the WebSocket**. The user will see the shell prompt as soon as the container is attached, and can start typing commands.

- **Resize PTY (Optional):** If you want to support resizing the terminal window, you can have the front-end send resize events (xterm.js can do this) and call the Docker API to resize the container's TTY (`container.resize({ h, w })`). Not mandatory, but improves UX.
- **Inactivity Timer:** Implement a timer to track inactivity. One simple way: on any `ws.on('message')` (user input), reset a `lastActivity = Date.now()` . Use `setInterval` to periodically check if `Date.now() - lastActivity > 15*60*1000` (15 min). If so, call `container.stop()` or `container.kill()` and send a message to client or simply close the socket. The container's AutoRemove ensures it gets removed. Also, when the WebSocket closes (e.g., user closed browser without typing exit), make sure to stop the container to avoid it running headless.

- **HTTP Server:** Finally, have the Express app listen on a port (e.g., 8080). Use HTTPS if possible (or behind a reverse proxy with TLS).

c. Terminal HTML/JS:

Create an `index.html` to serve at the root (or /terminal) that sets up the front-end:

- Include Xterm.js (you can use a CDN script tag for xterm.js CSS and JS, or serve local files from node_modules/xterm). For example:

html Copy

```html
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/xterm/css/xterm.css" />
<script src="https://cdn.jsdelivr.net/npm/xterm/lib/xterm.js">
</script>
<script src="https://cdn.jsdelivr.net/npm/xterm-addon-fit/lib/xterm-
addon-fit.js"></script>
```

- In a script, initialize the terminal and open a WebSocket:

js Copy

```
const term = new Terminal(
term.open(document.getElementById('terminal'));
const socket = new
WebSocket(`${location.origin.replace('http','ws')}/terminal`);
socket.binaryType = 'arraybuffer';  // use arraybuffer for binary data
(if needed)
term.onData(data => socket.send(data));  // send input to server
socket.onmessage = (event) => {
  const message = typeof event.data === 'string' ? event.data : new
Uint8Array(event.data);
  term.write(message);
};
term.focus();
```

This connects to the back-end WS and binds the terminal's input/output. Xterm.js will handle rendering the data it receives (which includes control sequences for colors, etc.).

d. Launch Script as a Service:
For production, run the Node app as a service or via a process manager. For example, create a systemd service file at `/etc/systemd/system/webterminal.service` with:

```ini
[Unit]
Description=Web Terminal Service
After=network.target docker.service
[Service]
ExecStart=/usr/bin/node /opt/webterminal/index.js
Restart=always
Environment=NODE_ENV=production
[Install]
WantedBy=multi-user.target
```

Enable and start it: `sudo systemctl enable --now webterminal`. Ensure the service user has Docker access (e.g., add `User=ubuntu` in the service file if running as a specific user, and that user is in `docker` group or use root for simplicity, though root is less safe).

3. Configure Nginx as Reverse Proxy (with Authentication)

Using Nginx in front is optional but ad Skip to content nvenient features:

- Install Nginx and ensure it's running. Obtain an SSL certificate for your domain (e.g., via Let's Encrypt) if using HTTPS.
- Create an **htpasswd file** for basic auth credentials (unless you have another auth method). For example: `htpasswd -c /etc/nginx/htpasswd.users admin` (you'll be prompted to set a password for user "admin").
- **Nginx Site Config:** Point a server block to the Node app. For instance, at `/etc/nginx/sites-available/webterminal.conf`:

nginx Copy

```nginx
server {
    listen 443 ssl http2;
    server_name yourdomain.com;
    ssl_certificate
/etc/letsencrypt/live/yourdomain.com/fullchain.pem;
    ssl_certificate_key
/etc/letsencrypt/live/yourdomain.com/privkey.pem;

    location / {
        auth_basic "Protected Terminal";
        auth_basic_user_file /etc/nginx/htpasswd.users;

        proxy_pass http://localhost:8080;  # Node app
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;   # Support WebSocket
upgrade
        proxy_set_header Connection "Upgrade";
        proxy_set_header Host $host;
    }
}
```

This configuration enforces basic Skip to content c) on all requests. It proxies both HTTP and WebSocket traffic to the Node server. The `proxy_set_header Upgrade` and `Connection "Upgrade"` directives are crucial for WebSocket connections to be forwarded properly. Without them, the WebSocket upgrade request may not work, resulting in 400 Bad Request or similar issues. We also ensure HTTP/1.1 is used and the Host header is preserved. If the Node app is on the same server, `localhost:8080` is fine. Adjust the port if different.

- Enable this site and reload Nginx:

```bash
ln -s /etc/nginx/sites-available/webterminal.conf /etc/nginx/sites-enabled/
nginx -t && sudo systemctl reload nginx
```

Now, Nginx will require the set username/password to access the site. After auth, it serves the HTML and proxies the WebSocket for the terminal.

4. Finalize and Test

- **Start the system:** Ensure Docker is running. Start the Node service (if not already). Access the web terminal URL (e.g., `https://yourdomain.com/`) from a browser.
- **Login Prompt:** You should see a basic auth prompt (if configured) – enter the credentials. If a custom login page is used, you'll see that instead.
- **Terminal Interface:** After authentication, the browser should load the terminal page. The Xterm.js terminal area should appear (possibly just a blank black screen initially). Within a second or two, you should see a shell prompt (e.g., `root@container-id:/#`) indicating the container's bash shell is ready. If you see this, the WebSocket connection and container attach were successful.
- **Try Commands:** Run some commands in the browser terminal (e.g., `uname -a` , `uptime` , etc.) to verify output is coming back. The commands should execute inside the isolated container. You can create files, install packages (`apt update` then `apt install` something) – remembering the container is transient. Also, verify the CPU and memory limits by trying a stress test (for example, run a CPU intensive loop and observe `top` from the host to see it throttled to 50%).

- **Exit Behavior:** Type `exit` in the [illegible]ell should close and the WebSocket disconnect. On the server side, Docker should remove the container. Check `docker ps -a` to ensure the session container is gone.
- **Inactivity Timeout:** To test the 15-minute inactivity timeout, you might reduce the timer to 1 minute for a quick test. Let the terminal sit idle; after the timeout, it should terminate. The browser might show a message like “Connection closed” or simply freeze until you refresh – indicating the session ended. Check that the container was removed.
- **Multi-User:** (If applicable) test simultaneous sessions by logging in from two different browsers or accounts. Each should get its own container (you can verify by `docker ps` showing two different container IDs). Ensure they operate independently.

Security Considerations

Security is paramount since this system allows shell access on a server (albeit in containers). Consider the following:

- **Authentication & Encryption:** We enabled user auth to restrict access. Use strong, unique passwords for each user, or integrate with LDAP/SSO for enterprise setups if needed. All traffic (especially the WebSocket carrying terminal data) should be over **HTTPS** to prevent sniffing. Nginx with SSL (or the Node server using HTTPS) is necessary.
- **Container Isolation:** Docker provides a good degree of isolation, but it’s wise to run the containers with **reduced privileges**:
  - Run the container as a **non-root user** if possible. By default Ubuntu’s `bash` will run as root in the container. You could create a user in the Docker image and use the `--user` flag or Dockerfile `USER` directive to drop privileges. This prevents a user from potentially damaging the container host if a Docker breakout vulnerability exists.
  - Consider adding Docker security options: e.g. `--cap-drop=ALL --security-opt=no-new-privileges` to drop Linux capabilities, and maybe use AppArmor or seccomp profiles to limit syscalls. This can mitigate the impact if someone tries to escape the container or exploit the kernel.

    - Limit networking if appropriat[illegible]eed internet access from the container, you could attach containers to a none or internal network, or use firewall rules to restrict outbound access. This reduces abuse risk (like launching attacks from the container).

- **Resource Limits:** We set CPU and RAM limits to prevent excessive resource use. 0.5 CPU ensures no single session hogs the processor entirely, and 200MB RAM contains memory usage (the container will be OOM-killed if it exceeds this). Depending on your server's capacity and expected load, you might tune these values. You can also limit disk usage or run containers in an LVM thin pool or use filesystem quotas if disk abuse is a concern.
- **Idle Timeout and Cleanup:** The 15-minute inactivity timeout helps ensure abandoned sessions don't linger. Orphaned containers (from crashes or unexpected disconnects) are removed thanks to the `--rm` /AutoRemove setting. It's good practice to also implement a periodic check (maybe a cron or systemd timer) that uses `docker ps` to find any containers older than a threshold and remove them, as a failsafe cleanup.
- **Audit Logging:** Consider logging user commands for auditing. This can be done by capturing the input stream server-side. However, be mindful of privacy and inform users if you implement this. At minimum, logging session start/stop times and user ID is useful.
- **Testing for Escape:** Before deploying to untrusted users, thoroughly test that a user in the web terminal truly cannot affect the host outside their container. For example, ensure they cannot docker exec into other containers (don't map the Docker socket into the container!), cannot see host file systems (no inadvertent volume mounts), and cannot use packet sniffers to capture traffic from the host. Each container should be a sandbox.
- **Denial of Service:** The system should handle multiple concurrent users. Use Docker's cgroup settings (as done) to prevent resource exhaustion. The Node server itself should also be protected – for instance, limit the number of concurrent WebSockets or use a reverse proxy to buffer against slow clients. Fail2ban can watch auth logs to block brute-force login attempts if using basic auth.

By adhering to these security practices, the system will be resilient and safe for providing shell access to users without exposing the host system or other users' sessions.

Deployment and Usage Instructions

In summary, to deploy the system:

1. **Deploy Docker and Web Server:** Set up Docker on the host and deploy the Node.js service (or alternative implementation) that manages the WebSocket and container lifecycle. Use systemd or a container for the service itself. Confirm it can launch containers on demand.
2. **Configure Reverse Proxy:** Put Nginx (or Apache/Caddy) in front for TLS and auth. Test that you can't access the terminal page without logging in. Ensure websockets upgrade correctly (you can check Nginx logs or use the browser dev console to see the WebSocket connection status).
3. **Environment Variables:** Configure any needed env vars, e.g., to set the inactivity timeout duration, allowed users, etc., either via a config file or environment. This makes it easier to adjust without code changes.
4. **Monitoring:** Set up monitoring for the service. For instance, ensure the Node process is running and not using excessive memory (it mostly will be idle, just relaying data). Monitor Docker metrics to see how many containers are running. You could also add an admin page or endpoint to list active sessions if needed.
5. **Documentation for Users:** Provide end-users with the URL and login credentials, and maybe some guidelines (e.g., remind them that sessions are temporary and will auto-terminate after idle period, and not to expect files to persist).

Once deployed, users can navigate to the site, authenticate, and will be greeted with a fully functional Linux shell in their browser. They can run commands as if on an Ubuntu system, with the comfort of knowing each session is isolated. This setup can be useful for training environments, demoing Linux tools, or any scenario where you want to provide shell access without giving out SSH keys or spinning up full VMs for each user.

Sources

References and Further Reading