NIG Workshop Stack: Multi-Instance Node-RED, InfluxDB & Grafana

This page documents how we built our own small “cloud” for hosting multiple isolated instances of a **Node-RED + InfluxDB + Grafana** stack (“NIG”) behind a single NGINX reverse proxy. The setup is what we use to give every workshop participant their own private playground on one shared server.

This is a workshop / development setup. Do *not* use this in production without a proper security review. Passwords are reused across services, the InfluxDB image is the unmaintained 1.x line, and there is no rate limiting or per-user network isolation. It is intentionally simple so it stays understandable.

What this page is (and isn't)

The goal here is to **explain the architecture** so you understand *what* is running, *why* it is running, and *how the pieces talk to each other*. The intended reader is a workshop participant — possibly a future lab staff member or someone from industry — who wants to understand the system well enough to either operate it, adapt it, or build something similar later.

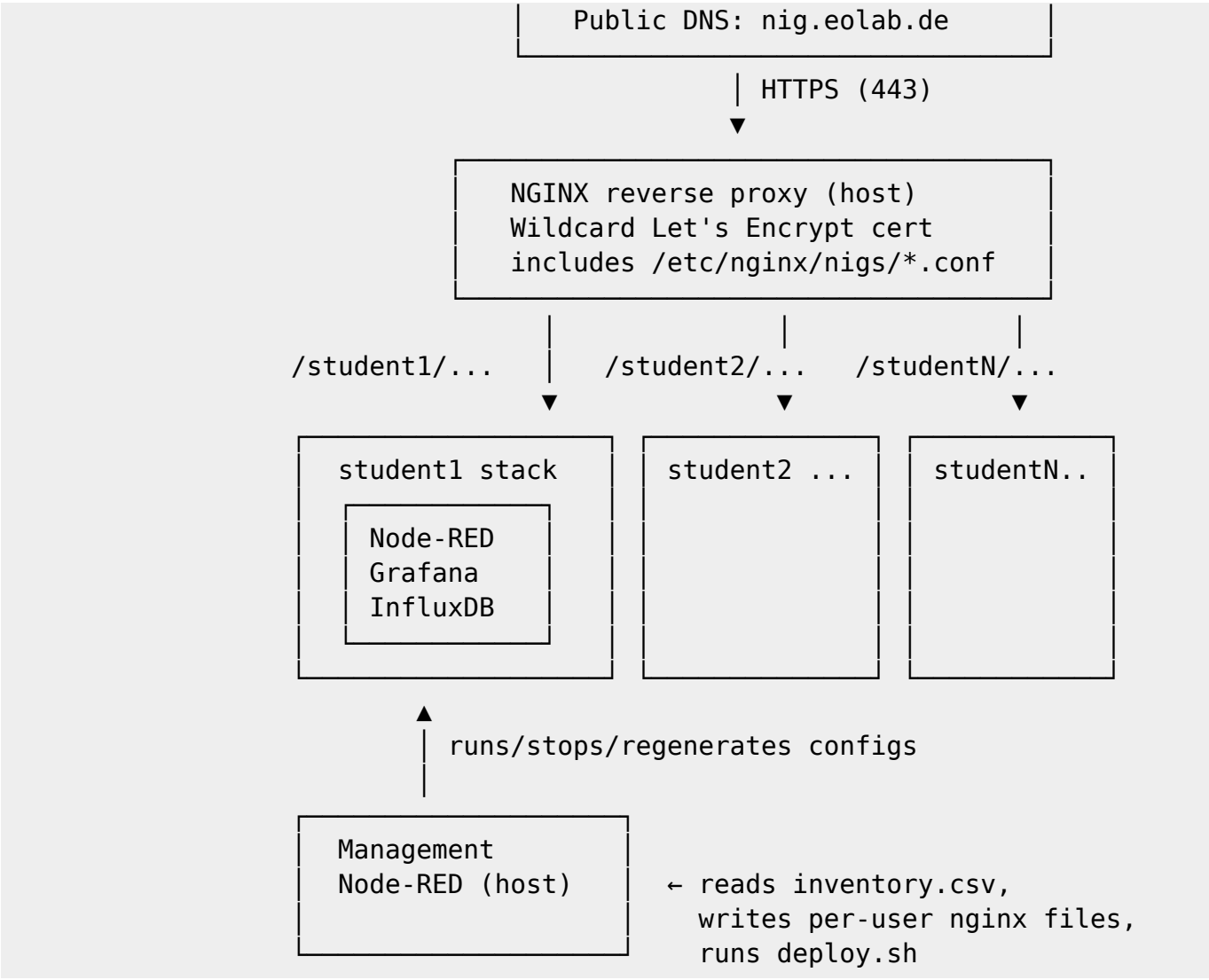
You are not expected to copy this 1:1. The multi-instance, NGINX-fronted, automated-deployment version is overkill for a single person learning Node-RED. If you just want a local NIG stack on your own laptop, jump to Running a local NIG instance at the end of this page. The middle sections are reference material for the people maintaining the workshop server.

All scripts, configs and the Node-RED management flow shown below live in our public repository:

- **Repository:** github.com/EOLab-HSRW/nig-workshop-stack

Architecture overview

The big idea is that **every workshop user gets their own isolated stack** — three Docker containers (Node-RED, InfluxDB, Grafana) — all sitting behind a *single* public NGINX entrypoint. Users do not need to know any port numbers; they reach their tools through clean URL paths like `/student1/node-red/` and `/student1/grafana/`.



There are two distinct “layers” of Node-RED in this setup, and it's important not to confuse them:

- **The student stacks** — one per user, containerised, isolated. This is what workshop participants actually use.
- **The management Node-RED** — a *single* Node-RED instance running on the host (not in Docker) that we use as a simple ops UI. It reads the user inventory, regenerates NGINX configs, reloads NGINX, and triggers the deploy script. We use it as a convenient front-end for the shell commands. It is also a nice (slightly meta) example of “what Node-RED is good for”.

Server setup

The following steps are roughly the order in which we set up the host. They assume a freshly provisioned Ubuntu server with SSH access and a domain you control (in our case `nig.eolab.de`).

Base packages

```
sudo apt update
sudo apt install -y nano snapd fuse nginx
sudo snap install core
```

```
sudo snap refresh core
```

We install **Snap** because we use it to install Certbot — the Snap version is the one Certbot maintainers currently recommend, and it gives us up-to-date plugins.

TLS certificates via Certbot (with IONOS DNS challenge)

We want a **wildcard certificate** for `*.example.com` so that every workshop URL is reachable over HTTPS without needing a separate certificate per student. Wildcards can only be issued via the **DNS-01 challenge** (HTTP-01 cannot prove control over a wildcard), which means Certbot has to be able to create temporary TXT records on our DNS provider.

Provider-specific section.
We use **IONOS** as our DNS provider, so we install the `certbot-dns-ionos` plugin. If you use a different provider (Cloudflare, Route53, …), you'll install a different plugin but the rest of the workflow is the same.

Install Certbot and the IONOS plugin:

```
sudo snap install --classic certbot
sudo ln -s /snap/bin/certbot /usr/bin/certbot
sudo snap set certbot trust-plugin-with-root=ok

# IONOS DNS plugin
sudo snap install certbot-dns-ionos
sudo snap connect certbot:plugin certbot-dns-ionos
```

Create an IONOS API user (in the IONOS control panel, under *System → Remote Users*, with rights for *Client Functions*, *DNS zone functions* and *DNS txt functions*) and store the credentials in an INI file readable only by root:

/root/ionos.ini

```
dns_ionos_prefix    = <PLACEHOLDER: your IONOS API key prefix>
dns_ionos_secret    = <PLACEHOLDER: your IONOS API key secret>
dns_ionos_endpoint  = https://api.hosting.ionos.com
```

```
sudo chmod 600 /root/ionos.ini
```

Make sure the following DNS records exist for your domain:

Record	Name	Value
A	*	server public IP
A	@	server public IP
CNAME	www	@

Now request the wildcard certificate:

```
sudo certbot certonly \
  --authenticator dns-ionos \
  --dns-ionos-credentials /root/ionos.ini \
  -d '*.example.com' -d 'example.com'
```

For the **manual** variant (no plugin), the equivalent invocation is:

```
sudo certbot --server https://acme-v02.api.letsencrypt.org/directory \
  -d '*.example.com' --manual --preferred-challenges dns-01 certonly
```

You will be asked to create a `_acme-challenge` TXT record by hand. Use the manual flow only if a plugin is not available — automatic renewal won't work for the manual challenge.

NGINX site configuration

We replace the default NGINX site with our own. Two files matter:

1. **`/etc/nginx/nginx.conf`** — the global config. Lightly customised: the only line that matters for this setup is that `sites-enabled` is *included*, which lets us drop in our virtual hosts as separate files.
2. **`/etc/nginx/sites-available/nig.eolab.de`** — our virtual host. Symlinked from `sites-enabled`. This is the file that terminates TLS and proxies into the student stacks.

Remove the default site and copy our template in its place:

```
sudo rm -f /etc/nginx/sites-enabled/default /etc/nginx/sites-
available/default
sudo nano /etc/nginx/sites-available/nig.eolab.de
sudo ln -s /etc/nginx/sites-available/nig.eolab.de /etc/nginx/sites-
enabled/nig.eolab.de
```

Generate Diffie-Hellman parameters once (used for older clients):

```
sudo openssl dhparam -out /etc/ssl/certs/dhparam.pem 2048
```

The key trick is that our site config does **not** list every student location explicitly. Instead, it `includes` a directory we manage with our Node-RED management flow:

```
# inside the server { ... } block of /etc/nginx/sites-available/nig.eolab.de
include /etc/nginx/nigs/*.conf;
```

Every time the inventory changes, the management flow writes one `<username>.conf` file into `/etc/nginx/nigs/` and reloads NGINX. This is what lets us add or remove students without touching the main site file.

After every change, validate and reload:

```
sudo nginx -t
sudo systemctl reload nginx
```

Docker

Install Docker and Docker Compose using the official Docker repository (instructions: <PLACEHOLDER: link to https://docs.docker.com/engine/install/ubuntu/>). Then enable the services so they come back after a reboot:

```
sudo systemctl enable docker.service
sudo systemctl enable containerd.service
```

Cloning the stack repository

```
cd /root
git clone https://github.com/EOLab-HSRW/nig-workshop-stack
cd nig-workshop-stack
```

This gives you `deploy.sh`, `compose.yml`, `gen_instances_files.py`, `globals.env` and `inventory.csv` — everything the next sections will use.

How a stack is built: the repository

The repo is small on purpose. Here is what each file does and why it exists.

inventory.csv — the source of truth

This is the *only* file you edit during a workshop. One row per user, with their credentials and the host ports their containers will bind to.

inventory.csv

```
USER_NAME,USER_PASSWORD,NODE_RED_PORT,NODE_RED_EXTRA_PORT,GRAFANA_PORT
student1,dump_password,17601,1881,17701
student2,dump_password,17602,1882,17702
```

- `NODE_RED_PORT` — host port mapped to Node-RED's editor (container port 1880).
- `NODE_RED_EXTRA_PORT` — host port mapped to Node-RED's “extra” port (container 1881). We expose this so students can spin up their own HTTP listeners, MQTT brokers, etc. on a port that doesn't collide with anyone else's.
- `GRAFANA_PORT` — host port mapped to Grafana (container 3000).
- **Every port must be unique** across all rows. The generator script enforces this.

globals.env — shared defaults

Values that apply to every instance: which images to use, which timezone, the public hostname for absolute URL generation.

globals.env

```
ROOT_URL=nig.eolab.de
NODE_RED_IMAGE=nodered/node-red:4.1.10-22
INFLUXDB_IMAGE=influxdb:1.12.4
GRAFANA_IMAGE=grafana/grafana:13.0.1
TZ=Europe/Berlin
INFLUXDB_DB=db
INFLUXDB_ADMIN_USER=admin
BCRYPT_ROUNDS=8
```

We pin every image to an exact tag. Workshops are easier to support when the version doesn't shift under you.

gen_instances_files.py — turning rows into env files

For each row in `inventory.csv`, this Python script writes one file `instances/<username>.env` that Docker Compose can later consume with `–env-file`. The interesting bits:

- **Port collision detection.** Two rows that share a port are rejected up front instead of producing a Docker error later.
- **Stable credential secret.** Node-RED encrypts saved credentials (e.g. MQTT passwords inside flows) with `NODE_RED_CREDENTIAL_SECRET`. If this changes, all existing encrypted credentials become unreadable. The generator therefore *preserves* an existing secret across regenerations, and only creates a fresh one (via `secrets.token_urlsafe(32)`) the first time around.
- **Sensible defaults for derived fields.** The InfluxDB user/password default to the workshop user's own credentials, the Compose project name is derived from `USER_NAME` (so `docker ps` shows readable container names), and so on.
- **Quoting safe for bcrypt hashes.** Bcrypt hashes start with sequences like `$2b$08$…`. Docker Compose interprets `$` as variable interpolation. The generator wraps such values in single quotes, which Compose treats as literal.

deploy.sh — the orchestrator

This is the script you actually invoke. It composes three steps:

1. Run `gen_instances_files.py` to (re)generate `instances/*.env`.
2. For each instance file, **hash the user's Node-RED password with bcrypt** by running a one-shot container against the pinned Node-RED image. This guarantees the hash is generated by the *same* bcrypt build that Node-RED itself will use to verify it.
3. For each instance file, call `docker compose -p <user> –env-file`

```
    instances/<user>.env -f compose.yml up -d.
```

Supported commands:

```
./deploy.sh up                # generate, hash, start all instances
./deploy.sh up student1       # same, but only for student1
./deploy.sh down              # stop and remove (including volumes) all
instances
./deploy.sh down student1     # stop and remove a single instance
./deploy.sh generate          # only regenerate the .env files
./deploy.sh hash              # only regenerate the bcrypt password hashes
./deploy.sh config            # render the merged Compose config for
inspection
```

Always use `./deploy.sh up`, not `docker compose up` directly.
A raw `docker compose` call won't generate the bcrypt hash, and Node-RED will refuse to start because `NODE_RED_PASSWORD_HASH` is empty.

By default `down` removes the named Docker volumes (`-v`). Pass `KEEP_VOLUMES=1 ./deploy.sh down student1` if you want to preserve a user's flows and InfluxDB data across a restart.

compose.yml — the per-user stack

A textbook three-service Compose file. The only things worth highlighting:

- **Strict required variables.** Most variables use the `${VAR:?error message}` form, which fails fast if Compose was invoked without the corresponding env file. This is what prevents "I forgot to run deploy.sh" from silently producing a broken stack.
- **Per-user URL prefix.** `NODE_RED_HTTP_ADMIN_ROOT` and `NODE_RED_HTTP_NODE_ROOT` are set to `/<user>/node-red/` and `/<user>/node-red/api` respectively. This is how Node-RED knows to render all its asset URLs underneath the per-user path — required for the reverse proxy to work without rewriting HTML.
- **Grafana sub-path.** `GF_SERVER_ROOT_URL` and `GF_SERVER_SERVE_FROM_SUB_PATH=true` do the same job for Grafana.
- **Mounted settings.** `./node-red/settings.js` is bind-mounted read-only into `/data/settings.js` so we can override Node-RED's admin auth and root paths without baking a custom image.
- **No published port for InfluxDB.** InfluxDB is only reachable via the internal Compose network as `influxdb:8086`. Students cannot point Grafana at someone else's database — there is no host port to point at.

node-red/settings.js — Node-RED runtime config

A trimmed Node-RED `settings.js`. The important parts:

- Refuses to start if `NODE_RED_PASSWORD_HASH` or `NODE_RED_CREDENTIAL_SECRET` are

missing, instead of starting with insecure defaults.
- Configures `adminAuth` with a single user whose password is the bcrypt hash injected by `deploy.sh`.
- Sets `httpAdminRoot` and `httpNodeRoot` from environment variables so the same image works for any user without rebuilding.
- Disables the projects feature — we don't want students accidentally bumping into git workflows during a 90-minute workshop.

grafana/provisioning/datasources/influxdb.yml — auto-wiring Grafana

Grafana supports "provisioning": YAML files dropped into `/etc/grafana/provisioning` are read at startup and used to declaratively create data sources, dashboards, etc. We use it to pre-create the InfluxDB data source so students don't have to do it by hand:

```
apiVersion: 1
datasources:
  - name: InfluxDB
    type: influxdb
    access: proxy
    url: http://influxdb:8086
    database: ${INFLUXDB_DB}
    user: ${INFLUXDB_USER}
    isDefault: true
    editable: true
    jsonData:
      httpMode: GET
    secureJsonData:
      password: ${INFLUXDB_USER_PASSWORD}
```

The variables `${INFLUXDB_DB}`, `${INFLUXDB_USER}`, `${INFLUXDB_USER_PASSWORD}` are resolved by Grafana from its *container* environment at startup — that's why the Compose file sets them on the Grafana service even though Grafana itself doesn't need to know the DB credentials at runtime, only to write the data source.

The management Node-RED flow

On the host we run an additional Node-RED instance (not in Docker, just installed with `npm install -g –unsafe-perm node-red` and started via `systemd`). Its single flow is our deployment console.

This is a deliberate example of "what is Node-RED for". For us it acts as a tiny ops UI that wires together "read CSV → write file → run shell command", which would otherwise be a Bash script. The wins are: anyone in the lab can click *Deploy* without remembering shell incantations, every step has its own debug output in the sidebar, and the flow itself documents the deployment pipeline visually.

Flow: Deploy / Redeploy

Trigger ⟶ *inject* node labelled **Deploy / Redeploy**.

1. **file in** reads `/root/nig-workshop-stack/inventory.csv`. It fans out to two branches:
   - a parallel branch that runs `rm /etc/nginx/nigs/*` once, to wipe any stale per-user NGINX snippets before regenerating them;
   - the main branch, which is delayed 5 seconds (giving the cleanup time to finish) and then parses the CSV.
2. **csv** parses the file with headers, emitting one message per row.
3. **change** stashes the parsed row under `msg.compose_info` so it survives the next steps.
4. **function: Generate NGINX Config** builds the per-user NGINX location blocks and assigns the target path:

```
msg.compose_info.nginx_path =
`/etc/nginx/nigs/${msg.compose_info.USER_NAME}.conf`;

msg.payload = `
    location /${msg.payload.USER_NAME}/grafana/ {
        proxy_set_header Host              $host;
        proxy_set_header X-Real-IP         $remote_addr;
        proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_pass http://localhost:${msg.payload.GRAFANA_PORT};
    }

    location /${msg.payload.USER_NAME}/grafana/api/live/ {
        proxy_http_version 1.1;
        proxy_set_header Upgrade    $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
        proxy_set_header Host       $host;
        proxy_pass http://localhost:${msg.payload.GRAFANA_PORT};
    }

    location /${msg.payload.USER_NAME}/node-red {
        proxy_pass http://localhost:${msg.payload.NODE_RED_PORT};
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header X-Real-IP $remote_addr;
    }
`;
return msg;
```

   The Grafana `/api/live/` block is a separate location because Grafana Live uses WebSockets, which need the `Upgrade` / `Connection` headers and HTTP/1.1. The Node-RED block needs the same WebSocket headers for the editor's live status updates.

5. **file** writes the generated snippet to the path stored in `msg.compose_info.nginx_path`, overwriting any previous file for that user.
6. **exec: systemctl reload nginx.service** picks up the new file. Reload (not restart) keeps

existing connections alive.
7. **change** sets `msg.payload` to the user name, which is then appended as an argument to:
8. **exec: bash /root/nig-workshop-stack/deploy.sh up** — the same script described above. Stdout and stderr are routed to separate debug nodes for visibility.

Flow: Delete all

Trigger ⟶ *inject* node labelled **Delete all**. Two parallel branches:

- `bash /root/nig-workshop-stack/deploy.sh down` — stops and removes every Compose stack.
- `rm /etc/nginx/nigs/*` followed by `systemctl reload nginx.service` — wipes the per-user NGINX configs.

Why have this at all?

For two users you could absolutely just SSH in and run `./deploy.sh up`. For 20 students during a workshop where the inventory changes a few times an hour, a single-click “regenerate everything” button is much harder to mess up — and the visual flow gives a less-experienced operator a fighting chance at understanding what is being done on their behalf.

Access URLs

Given the example `inventory.csv`, after `./deploy.sh up` the URLs are:

Service	URL
Node-RED editor	`https://nig.example.com/student1/node-red/`
Node-RED HTTP endpoints	`https://nig.example.com/student1/node-red/api/…`
Grafana	`https://nig.example.com/student1/grafana/`
Node-RED extra port (direct, no proxy)	