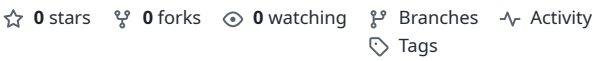
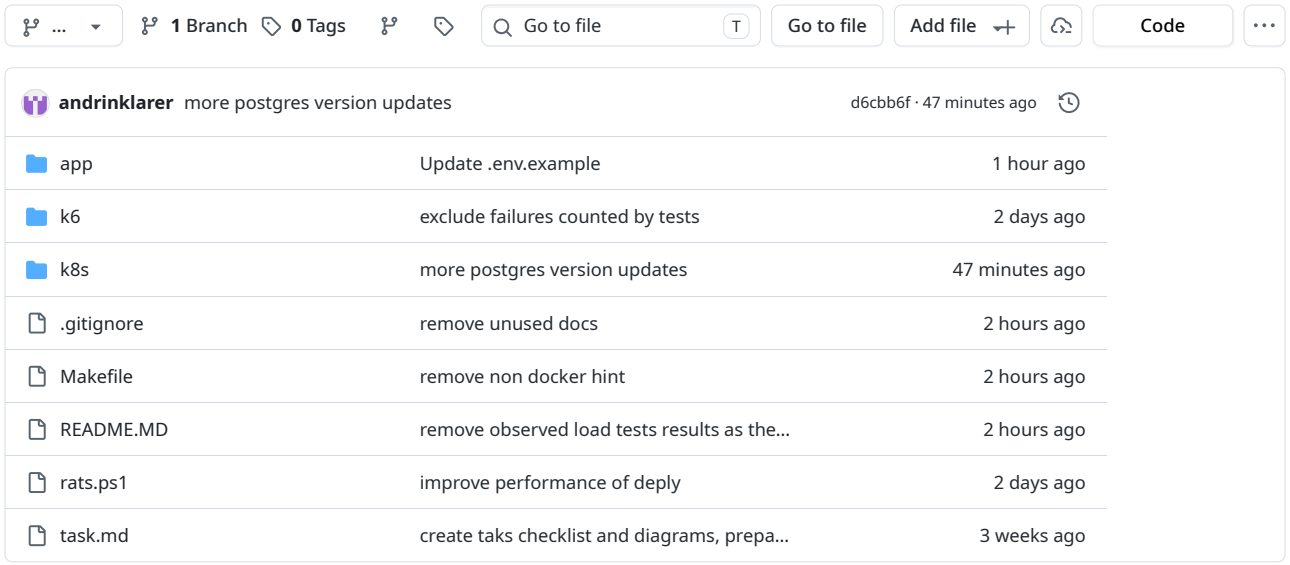
Reliable & Available Ticketing System for Rattenfest



Private repository



README

R.A.T.S

Reliable & Available Ticketing System for Rattenfest

Project Idea

R.A.T.S. is a distributed, containerized flash-sale ticketing system designed for the real-world event Rattenfest.

During the hours shortly before the event starts the ticketing system experiences a massive spike in traffic. This project simulates that and handles it appropriately.

Tech Stack

- **Frontend & Backend:** Next.js + tRPC
- **Authentication:** NextAuth.js, JWT-based (credentials, httpOnly cookies, stateless sessions)
- **Database:** PostgreSQL (Drizzle ORM)
- **Orchestration:** Kubernetes (Docker Desktop)
- **Load Balancer:** NGINX Ingress Controller
- **Load Testing:** k6

Getting Started

Prerequisites

- Docker Desktop with Kubernetes enabled
- `kubectl` available in your terminal

Kubernetes UI (optional)

To visually inspect pods, services, and the load balancer — and kill resources from a UI:

- **Kubernetes Dashboard** (web UI):

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml

# Grant access to all namespaces (run once)
kubectl create clusterrolebinding kubernetes-dashboard-admin `
  --clusterrole=cluster-admin `
  --serviceaccount=kubernetes-dashboard:kubernetes-dashboard

kubectl proxy

# Token:
kubectl -n kubernetes-dashboard create token kubernetes-dashboard
```

Link to WebUI

Start the full stack

```
.\rats.ps1 start
```

This builds the Docker image, applies all Kubernetes manifests, runs DB migrations, and seeds the Rattenfest event data — all in one command.

- **App:** http://localhost (via NGINX Ingress)
- **Database:** `postgresql://postgres:postgres@localhost:30432/rats`

Other commands

```
.\rats.ps1 stop       # tear down all K8s resources
.\rats.ps1 restart    # full rebuild and redeploy
.\rats.ps1 logs       # stream app logs
.\rats.ps1 status     # show pod status
```

Re-seed the database (e.g. after a load test)

The startup script always truncates and re-seeds on every deploy. To trigger it without a full restart, re-run the migrate Job:

```
kubectl delete job rats-migrate -n rats
kubectl apply -k k8s/
```

Failover Demo

Shows that the system keeps serving traffic while a pod is killed and K8s respawns it.

Terminal 1 — watch pod lifecycle:

```
kubectl get pods -n rats -w
```

Terminal 2 — stream logs with pod names:

```
kubectl logs -n rats -l app=rats-app -f --prefix=true
```

Terminal 3 — kill one pod:

```
# Get the pod names first
kubectl get pods -n rats

# Delete one of the rats-app pods
kubectl delete pod -n rats <pod-name>
```

During the respawn window the remaining pod handles all traffic. Kill both -> app wont respond

Load Testing (k6)

Simulates the Rattenfest flash-sale spike: 200 users ramping up simultaneously and all trying to purchase tickets at once.

Prerequisites

Install k6: https://grafana.com/docs/k6/latest/set-up/install-k6/

Run the load test

```
# Default: 200 VUs against http://localhost
k6 run k6/load-test.js
```

What it does

1. **Setup** — registers 50 test users (`loadtest-0@rats.test` … `loadtest-49@rats.test`) and authenticates each one via NextAuth, collecting session tokens. 200 VUs share the 50 sessions round-robin.
2. **Ramp-up (10s)** — VUs increase from 0 → 200, simulating users rushing in at event start.
3. **Sustained peak (30s)** — all 200 VUs hammer the `orders.purchase` endpoint concurrently.
4. **Ramp-down (5s)** — traffic subsides.

Bottleneck analysis

The bottleneck is **PostgreSQL**. Every ticket purchase runs a serialised DB transaction (`SELECT FOR UPDATE` + `UPDATE` + `INSERT`), which means concurrent purchases queue at the row lock rather than executing in parallel. Horizontal scaling of the app pods cannot improve purchase throughput beyond what the single PostgreSQL instance can sustain.

Evidence from the results: response time climbs during peak load, which is probably because of lock-wait time inside the DB, not app processing. To push throughput higher, the next step would be read replicas (for non-purchase queries) or sharding the event table.

Reset event seats between runs

```
$env:PGPASSWORD="postgres"; echo 'UPDATE rats_event SET "availableSeats" = 3000; DELETE FROM rats_order;' | psql -h l
```

Verify no overselling after the test

```
$env:PGPASSWORD="postgres"; echo 'SELECT COUNT(*) FROM rats_order WHERE status = ''confirmed'';' | psql -h localhost
```

The count must not exceed `totalSeats` (3000).

JWT Authentication & OWASP Compliance

Authentication uses NextAuth.js v5 with a JWT session strategy. Tokens are never exposed to JavaScript, they live exclusively in an `httpOnly` cookie set by the server.

OWASP Guideline	Implementation
Tokens stored server-side only	`httpOnly` cookie — inaccessible to JavaScript
Short token lifetime	15-min access token + 7-day refresh token (rotated on use, stored as SHA-256 hash)
Strong signing secret	256-bit random `AUTH_SECRET` in K8s Secret
Password hashing	bcryptjs with cost factor 12
Input validation	Zod schema on every auth endpoint (email format, password ≥ 8 chars)
CSRF protection	NextAuth CSRF token required on all credential submissions
SQL injection prevention	Drizzle ORM parameterised queries throughout
Sensitive data in token	Only `userId` and `email` — no passwords or roles
Protected API enforcement	`protectedProcedure` middleware rejects any request without a valid token

Stateless design: Because the JWT is verified cryptographically on every request, both app pods can authenticate any user without shared session state or a session store.

Docker Security

The container image follows the OWASP Docker Security Cheat Sheet:

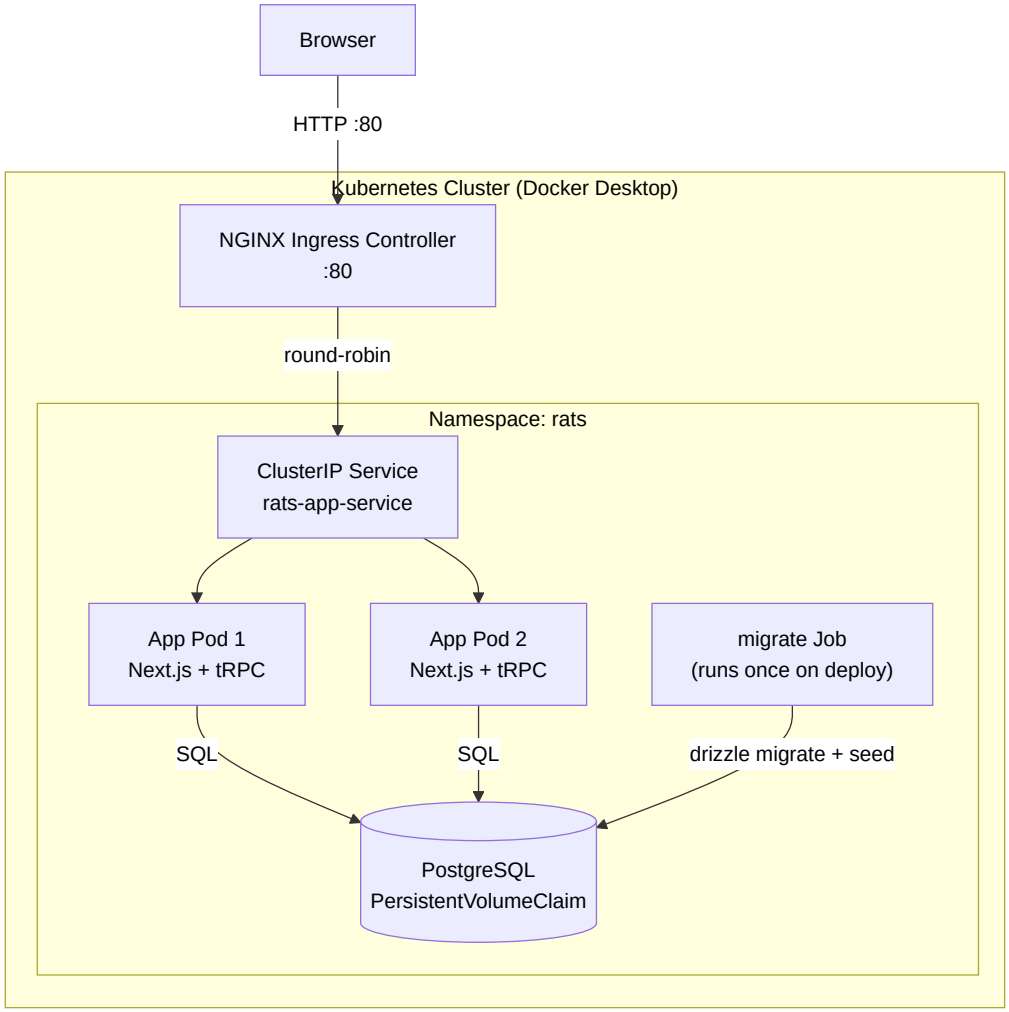
Measure	Implementation
Non-root user	Runner stage creates a `nextjs` user and drops root before `CMD`
Small attack surface	`node:25-alpine` base image — no shell tools or package manager in production
Multi-stage build	Only compiled output and production `node_modules` are copied to the runner stage
No daemon socket exposure	`/var/run/docker.sock` is never mounted
Secrets via K8s Secrets	`AUTH_SECRET`, `DATABASE_URL`, and `POSTGRES_PASSWORD` are injected from K8s Secrets at runtime, never baked into the image

Verify the non-root user:

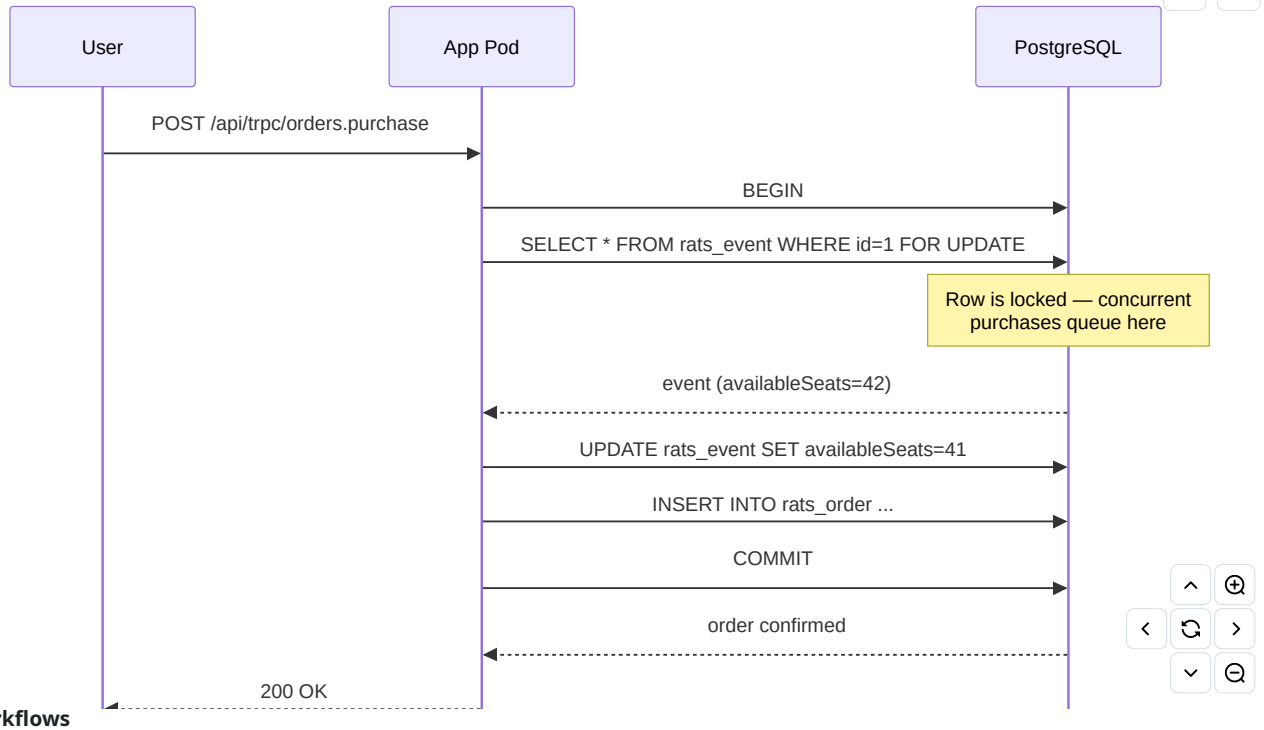
```
docker build app/ -t rats-app:latest
docker run --rm rats-app:latest whoami  # → nextjs
```

Architecture

System topology



Purchase flow (row-level locking)



Releases

No releases p
Create a new

Packages

No packages
Publish your f

Contribut

andri

Language

TypeScri

Suggested workflows

Based on your tech stack

Deno

Test your Deno project.

Publish Node.js Package to GitHub Packages

Publishes a Node.js package to GitHub Packages.

SLSA Generic generator

Generate SLSA provenance for your existing release workflows

More workflows

Key design decisions

Decision	Why
`SELECT FOR UPDATE` in a transaction	Prevents overselling under concurrent load - the DB serialises conflicting purchases at the row level
Migration as a K8s Job	With 2 replicas, running migration as an initContainer would race (it would run the twice, once for each pod), the job runs once
Readiness probe on port 3000	NGINX only routes to a pod once Next.js is actually serving - no cold-start 502s
`wait-for-migration` initContainer	App pods poll for the `rats_event` table before starting, so they never boot against an empty schema

Configure

Configure

Configure

Dismiss suggestions