HANDS-ON API TESTING

Roodie API Test Guide

Test two real HTTP APIs from your terminal with curl. Every command in this guide was executed against the live server, and the "Expect" lines show what it really returned.

Built by **Jonathan (Roodie77)** · github.com/Roodie77/jonathan-portfolio

Roodie Insuretech API

- File an insurance claim and see it assessed, scored for fraud and routed
- Adjuster decisions, payouts and appeals
- Roles, ETag concurrency and idempotent payments

Xeno Shield API

- Screen user input for injection, XSS and evasion; quarantine and review it
- Detect brute force, credential stuffing and password spraying
- Escalating bans and tunable policies

Quick start (2 minutes)

1. Install **Node 18+** and **curl**. On Windows use **Git Bash** or WSL (PowerShell's `curl` is a different tool).
2. Terminal 1, start the server (no `npm install` needed):

```
git clone https://github.com/Roodie77/jonathan-portfolio.git
cd jonathan-portfolio
node server.js --fresh
```

3. Terminal 2, load the helpers and make the first call:

```
cd jonathan-portfolio
source scripts/curl-helpers.sh
api $INS
```

The server listens on **localhost only**. Nothing is exposed to the internet. Data is saved in `./data`; `--fresh` resets it.

What the helpers do

`api` is plain curl plus the status code plus pretty JSON. Swap in `curl` any time to see the raw response:

```
curl -i "$INS/claims?limit=1"
```

`field` reads a value from JSON, `etag` fetches a record's version, `day -3` prints a date three days ago.

Variables and tokens

- `$INS` Insuretech base URL
- `$SHD` Shield base URL
- `$CLAIMANT` · `$ADJUSTER` Insuretech roles
- `$SHIELD_KEY` Shield writes
- `$JSON` · `$MERGE` content types

Reads need no token. Run the steps in order: later steps reuse `$ID`, `$ETAG` and friends.

Part 1 Roodie Insuretech API

An insurance claims engine. Base URL `$INS` = `http://localhost:3000/insuretech/v1`

1 Look around

Read endpoints are open, so no token yet. `api` is the helper from `curl-helpers.sh`: curl plus the status code plus pretty JSON.

1.1.1 Meet the API

```
api $INS
```

EXPECT HTTP 200. The index lists the roles, workflow and conventions.

1.1.2 List the policies, picking only some fields

```
api "$INS/policies?fields=holder,product,status,limit"
```

EXPECT HTTP 200. Six seeded policies, including one lapsed and one still inside its waiting period.

1.1.3 Filter and sort: high fraud risk first

```
api "$INS/claims?risk=high&sort=-fraud_score&fields=claim_number,status,amount_claimed"
```

EXPECT HTTP 200. Claims in the high fraud band, highest score first, with only the fields you asked for.

2 Issue a clean policy (adjuster)

Every claim below uses a brand-new policy, so results are the same no matter how many times you run this guide. Notice the role: policies are an adjuster action.

1.2.1 Create a policy and keep its id

The `'$(day -120)'` pattern drops a real date into the JSON (a policy that started 120 days ago).

```
PID=$(curl -s -X POST $INS/policies -H "$ADJUSTER" -H "$JSON" -d '{
  "holder": "Test Driver", "product": "auto",
  "coverages": ["collision", "theft", "glass"],
  "limit": 20000, "deductible": 500, "premium_annual": 1100,
  "start_date": "'$(day -120)'", "end_date": "'$(day 245)'",
  "auto_approve_limit": 1500
}' | field .id)
echo "policy: $PID"
```

EXPECT Prints the new policy id, for example `policy: pol_0mvn2m…` .

3 File claims (claimant) and watch the routing

Four claims, four different outcomes. Each response contains the full *assessment*: eligibility checks, coverage maths, fraud flags and the reason it was routed.

1.3.1 A small glass claim: straight-through approval

```
api -X POST $INS/claims -H "$CLAIMANT" -H "$JSON" -d '{
  "policy_id": "'$PID'", "type": "glass", "incident_date": "'$(day -1)'",
  "amount_claimed": 900,
  "description": "A stone chipped the windshield on the motorway."
}'
```

EXPECT HTTP 201, `"status": "approved"`, `"approved_amount": 400` ($900 minus the $500 deductible), routed `straight-through` with no human involved.

1.3.2 A bigger collision claim: needs an adjuster

```
ID=$(curl -s -X POST $INS/claims -H "$CLAIMANT" -H "$JSON" -d '{
  "policy_id": "'$PID'", "type": "collision", "incident_date": "'$(day -3)'",
  "amount_claimed": 2600,
  "description": "Hit a pothole and bent the front wheel and suspension."
}' | field .id)
echo "claim: $ID"
curl -s $INS/claims/$ID | field .assessment.routing
```

EXPECT The routing is `adjuster`: the amount ($2,600) exceeds the policy's $1,500 auto-approve limit. It is now `under_review`.

1.3.3 A suspicious theft claim: special investigations

```
SID=$(curl -s -X POST $INS/claims -H "$CLAIMANT" -H "$JSON" -d '{
  "policy_id": "'$PID'", "type": "theft", "incident_date": "'$(day -2)'",
  "amount_claimed": 19000,
  "description": "Car stolen overnight, total loss, no police report, need cash urgently."
}' | field .id)
curl -s $INS/claims/$SID | field '.assessment.fraud.flags.map(f => f.code + " +" + f.weight).join(",
")'
curl -s $INS/claims/$SID | field 'd.status + " / score " + d.assessment.fraud.score'
```

EXPECT Four red flags add up to 55 (high band): near-limit +20, amount-outlier +15, suspicious-language +12, round-amount +8. The claim lands in `siu_review`.

1.3.4 A claim on a lapsed policy: automatic denial

```
DID=$(curl -s -X POST $INS/claims -H "$CLAIMANT" -H "$JSON" -d '{
  "policy_id": "pol_auto_lapsed", "type": "collision", "incident_date": "'$(day -2)'",
  "amount_claimed": 1200,
  "description": "Scraped the side of the car on a gate post."
}' | field .id)
curl -s $INS/claims/$DID | field 'd.status + " " + JSON.stringify(d.decision.reason_codes)'
```

EXPECT Denied by the system with a machine-readable reason: `denied ["policy.active"]`.

4 Adjuster decisions: roles, ETags and safe payments

This is the part real claims systems get wrong. Watch the API refuse a blind write, a wrong role and a stale read, then refuse to pay twice.

1.4.1 Read the claim's ETag

An ETag is a version fingerprint. Writes must send it back in `If-Match`.

```
ETAG=$(etag $INS/claims/$ID)
echo "etag: $ETAG"
```

EXPECT Prints something like `etag: "1-6b2f9c1a"`.

1.4.2 Approve without If-Match

```
api -X POST $INS/claims/$ID/decision -H "$ADJUSTER" -H "$JSON" \
  -d '{"decision": "approve", "reason": "Photos match the estimate."}'
```

EXPECT HTTP 428 Precondition Required. The API will not let two people act on the same claim blindly.

1.4.3 Approve as a claimant

```
api -X POST $INS/claims/$ID/decision -H "$CLAIMANT" -H "$JSON" -H "If-Match: $ETAG" \
  -d '{"decision": "approve", "reason": "I would like my money."}'
```

EXPECT HTTP 403 Forbidden. The body says which role is required and which one you have.

1.4.4 Approve properly

```
api -X POST $INS/claims/$ID/decision -H "$ADJUSTER" -H "$JSON" -H "If-Match: $ETAG" \
  -d '{"decision": "approve", "reason": "Photos match the estimate."}'
```

EXPECT HTTP 200, status `approved`, `"approved_amount": 2100` ($2,600 minus the $500 deductible). The claim is now at version 2.

1.4.5 A second adjuster still holding the old ETag tries to deny

```
api -X POST $INS/claims/$ID/decision -H "$ADJUSTER" -H "$JSON" -H "If-Match: $ETAG" \
  -d '{"decision": "deny", "reason": "Looks suspicious to me."}'
```

EXPECT HTTP 412 Precondition Failed: "this record changed since you read it (now at version 2)". Nobody silently overwrites anybody.

1.4.6 Pay the claim, with an Idempotency-Key

The key makes retries safe: the same key replays the first response instead of paying again.

```
ETAG=$(etag $INS/claims/$ID)
api -X POST $INS/claims/$ID/pay -H "$ADJUSTER" -H "If-Match: $ETAG" -H "Idempotency-Key: pay-$ID"
```

> **EXPECT** HTTP 200, status `paid`, with a `payment_reference` like PAY-7K2QX9ML.

1.4.7 Retry the exact same payment

```
curl -s -D - -o /dev/null -X POST $INS/claims/$ID/pay \
  -H "$ADJUSTER" -H "If-Match: $ETAG" -H "Idempotency-Key: pay-$ID" | grep -iE '^(HTTP|idempotent-replayed)'
```

> **EXPECT** `HTTP/1.1 200 OK` and `idempotent-replayed: true`. Same answer, no second payout.

1.4.8 Pay again with no key (an impatient double-click)

```
api -X POST $INS/claims/$ID/pay -H "$ADJUSTER" -H "If-Match: *"
```

> **EXPECT** HTTP 409 Conflict: "already paid", with the original reference. A second layer of protection.

5 Claimant actions

Claimants can add documents, edit while a claim is pending, appeal a denial once, and withdraw.

1.5.1 Attach a document to the investigated claim

```
api -X POST $INS/claims/$SID/documents -H "$CLAIMANT" -H "$JSON" \
  -d '{"kind": "police-report", "name": "police-report.pdf"}'
```

> **EXPECT** HTTP 200 and the document appears in `"documents"`.

1.5.2 Edit the amount with a JSON Merge Patch

```
ETAG=$(etag $INS/claims/$SID)
api -X PATCH $INS/claims/$SID -H "$CLAIMANT" -H "$MERGE" -H "If-Match: $ETAG" \
  -d '{"amount_claimed": 6000}'
```

> **EXPECT** HTTP 200. Only the field you sent changed, and the whole assessment was recomputed.

1.5.3 Try to change something that is locked after filing

```
api -X PATCH $INS/claims/$SID -H "$CLAIMANT" -H "$MERGE" -H "If-Match: *" \
  -d '{"type": "fire"}'
```

> **EXPECT** HTTP 422: `"type" cannot be changed after creation`. The policy, type and incident date define what was assessed.

1.5.4 Appeal the automatic denial

```
api -X POST $INS/claims/$DID/appeal -H "$CLAIMANT" -H "$JSON" \
  -d '{"reason": "The policy was active when the loss happened and I can prove it."}'
```

EXPECT HTTP 200, status `appealed`.

1.5.5 Appeal a second time

```
api -X POST $INS/claims/$DID/appeal -H "$CLAIMANT" -H "$JSON" \
  -d '{"reason": "Appealing again just to see what happens here."}'
```

EXPECT HTTP 409: only denied claims can be appealed, and only once.

1.5.6 Withdraw a claim, then look for it

```
api -X DELETE $INS/claims/$SID -H "$CLAIMANT" -H "If-Match: *"
api $INS/claims/$SID
```

EXPECT HTTP 204, then HTTP 410 Gone with a link to restore it. Claims are soft-deleted, never erased.

6 Errors you should see

Every error is RFC 9457 `application/problem+json`: a type, a title, the status, a human detail and a request id.

1.6.1 No token

```
api -X POST $INS/claims -H "$JSON" -d '{}'
```

EXPECT HTTP 401 Unauthorized.

1.6.2 A bad body: every problem reported at once, with JSON pointers

```
api -X POST $INS/claims -H "$CLAIMANT" -H "$JSON" \
  -d '{"policy_id": "pol_nope", "type": "meteor", "amount_claimed": -5, "description": "short",
"status": "approved"}'
```

EXPECT HTTP 422 with an `errors` list. It also tells you that `status` is read-only and which endpoint moves a claim.

1.6.3 Try to permanently delete a financial record

```
api -X DELETE "$INS/claims/$ID?purge=true" -H "$CLAIMANT" -H "If-Match: *"
```

EXPECT HTTP 409 `retention-required`. Claims are never purged.

1.6.4 A method that does not exist here

```
api -X PUT $INS/claims/$ID -H "$CLAIMANT" -H "$JSON" -d '{}'
```

> **EXPECT** HTTP 405 Method Not Allowed, with the allowed methods listed.

7 The business view

1.7.1 Straight-through rate, money paid and loss ratio per product

```
api $INS/stats
```

> **EXPECT** HTTP 200. `loss_ratio` is (paid + reserved) / premium written, the number an insurer watches.

Part 2 Xeno Shield API

Quarantines bad data and stops abuse. Base URL `$SHD` = `http://localhost:3000/shield/v1`

1 Look around

2.1.1 Meet the API

```
api $SHD
```

EXPECT HTTP 200. Lists the conventions and links.

2.1.2 Read a policy

```
api $SHD/policies/login
```

EXPECT HTTP 200. The login guard allows 5 failures per minute before it bans.

2 Screen user input (quarantining bad data)

Send the fields of a form to `/inspections`. Each field is scored by detectors; anything over the threshold is quarantined for a human to review. A random `source` keeps every run independent.

2.2.1 A clean form

```
SRC=curl-$RANDOM
api -X POST $SHD/inspections -H "$SHIELD_KEY" -H "$JSON" -d '{
  "source": "'$SRC'",
  "fields": { "name": "Ada Lovelace", "email": "ada@example.com", "message": "Loved the demo!" }
}'
```

EXPECT HTTP 200, `"verdict": "allow"`. Nothing suspicious, nothing stored.

2.2.2 SQL injection hiding in one field

```
api -X POST $SHD/inspections -H "$SHIELD_KEY" -H "$JSON" -d '{
  "source": "'$SRC'",
  "fields": {
    "name": "Ada Lovelace",
    "id": "1 UNION SELECT username, password FROM users"
  }
}'
```

EXPECT HTTP 201 Created: `"verdict": "quarantine"`. The `name` field is allowed, `id` is quarantined (rule `sqli.union-select`) and gets a `case_id`.

2.2.3 Cross-site scripting

```
api -X POST $SHD/inspections -H "$SHIELD_KEY" -H "$JSON" -d '{
  "source": "'$SRC'",
  "fields": { "comment": "<script>alert(document.cookie)</script>" }
}'
```

EXPECT HTTP 201, category `xss`.

2.2.4 The same attack, double URL-encoded to dodge filters

```
api -X POST $SHD/inspections -H "$SHIELD_KEY" -H "$JSON" -d '{
  "source": "'$SRC'",
  "fields": { "q": "%253Cscript%253Ealert(1)%253C%252Fscript%253E" }
}'
```

EXPECT HTTP 201. Input is decoded before matching, and an extra `evasion.encoding` finding is added.

2.2.5 Dry run: report findings but store nothing

```
api -X POST $SHD/inspections -H "$SHIELD_KEY" -H "$JSON" \
  -d '{"source": "'$SRC'", "mode": "monitor", "payload": "../../../../etc/passwd"}'
```

EXPECT HTTP 200, `"mode": "monitor"`, verdict `quarantine` but `"case_id": null`. Safe for rolling the screening out gradually.

3 Review the quarantine

A human reviews each held item. The review workflow is a small state machine, and the API enforces it.

2.3.1 Find the case you just created

```
CASE=$(curl -s "$SHD/quarantine?source=$SRC&q=union&limit=1" | field .data[0].id)
echo "case: $CASE"
```

EXPECT Prints `case: qr_…`. Notice the filter and full-text search on the list endpoint.

2.3.2 Confirm it as malicious (needs the ETag)

```
ETAG=$(etag $SHD/quarantine/$CASE)
api -X PATCH $SHD/quarantine/$CASE -H "$SHIELD_KEY" -H "$MERGE" -H "If-Match: $ETAG" \
  -d '{"status": "confirmed", "labels": ["sqli"], "reviewer_note": "Verified by hand."}'
```

EXPECT HTTP 200, status `confirmed`, version 2.

2.3.3 Try to jump straight from confirmed to released

```
api -X PATCH $SHD/quarantine/$CASE -H "$SHIELD_KEY" -H "$MERGE" -H "If-Match: *" \
  -d '{"status": "released"}'
```

EXPECT HTTP 409 `illegal-transition`. The response lists the transitions that *are* allowed.

2.3.4 Try to edit the captured evidence

```
api -X PATCH $SHD/quarantine/$CASE -H "$SHIELD_KEY" -H "$MERGE" -H "If-Match: *" \
  -d '{"payload": "perfectly harmless"}'
```

EXPECT HTTP 422: the payload is immutable once captured.

2.3.5 Send the same attack again

```
api -X POST $SHD/inspections -H "$SHIELD_KEY" -H "$JSON" -d '{
  "source": "'$SRC'",
  "fields": { "id": "1 UNION SELECT username, password FROM users" }
}'
```

EXPECT HTTP 200 with reason `known-malicious`. No duplicate case: the existing one just counts another hit.

4 Stop abuse (brute force, spraying)

Call `/guard/attempts` on every login. The answer is allow (200), throttle (429, with Retry-After) or block (403). These two loops fire real attacks at the guard.

2.4.1 Brute force: 8 password guesses at one account from one IP

```
IP=203.0.113.$((100 + RANDOM % 100))
for i in 1 2 3 4 5 6 7 8; do
  curl -s -o /dev/null -w "attempt $i -> HTTP %{http_code}\n" \
    -X POST $SHD/guard/attempts -H "$SHIELD_KEY" -H "$JSON" \
    -d '{"scope": "login", "ip": "'$IP'", "identity": "admin", "outcome": "failure"}'
done
```

EXPECT Two `200`s, two `429` throttles (back off!), then `403` from the 5th attempt on: the IP is banned.

2.4.2 Ask whether that subject is blocked

```
api "$SHD/guard/status?scope=login&ip=$IP"
```

EXPECT HTTP 200, `"blocked": true`, with the ban record. This is the read-only check you would call from middleware.

2.4.3 Password spraying: one common password, many accounts, many IPs

Per-IP counting would never catch this. The guard tracks the password *hash* across accounts (never the real password).

```
FP=pw_summer2026_$RANDOM
for i in 1 2 3 4 5 6; do
  curl -s -o /dev/null -w "account $i -> HTTP %{http_code}\n" \
    -X POST $SHD/guard/attempts -H "$SHIELD_KEY" -H "$JSON" \
    -d '{"scope": "login", "ip": "198.51.100.'$((100 + RANDOM % 100))'", "identity": "user'$i'",
"secret_fingerprint": "'$FP'"}'
done
```

EXPECT Four `200` s, then `403` from the 5th account: the *password fingerprint* itself is banned, whichever IP tries it.

2.4.4 See what the guard banned

```
api "$SHD/bans?active=true&sort=-created_at&limit=2&fields=kind,value,reason"
```

EXPECT HTTP 200. Newest first: the spray ban (`fingerprint`) then the brute-force ban (`ip`).

5 Errors and totals

2.5.1 No token

```
api -X POST $SHD/inspections -H "$JSON" -d '{"source": "x", "payload": "hi"}'
```

EXPECT HTTP 401 with a `WWW-Authenticate` challenge.

2.5.2 Contradictory input

```
api -X POST $SHD/inspections -H "$SHIELD_KEY" -H "$JSON" \
  -d '{"source": "x", "fields": {"a": 1}, "payload": "both"}'
```

EXPECT HTTP 422 listing each problem with a JSON pointer.

2.5.3 The totals

```
api $SHD/stats
```

EXPECT HTTP 200: cases by status and category, active bans, and how many attempts were allowed, throttled and blocked.

Appendix

Status codes you will meet

CODE	MEANING	WHEN
200	OK	Read or update succeeded.
201	Created	A claim, policy, case or ban was created (see the Location header).
204	No Content	Success with nothing to return (withdraw, lift).
304	Not Modified	Your If-None-Match ETag is still current.
400	Bad Request	Malformed JSON, bad cursor or invalid query parameter.
401	Unauthorized	Missing or unknown bearer token.
403	Forbidden	Insuretech: your role cannot do this. Shield guard: the attempt was blocked (banned).
404	Not Found	No such route or id.
405	Method Not Allowed	The path exists but not for that method (see Allow).
409	Conflict	Valid request, wrong state: duplicate, illegal transition, already paid.
410	Gone	Soft-deleted. The body links to the restore endpoint.
412	Precondition Failed	Stale ETag: someone changed it after you read it.
413	Payload Too Large	Request body over 1 MB (local server).
415	Unsupported Media Type	Wrong Content-Type for the body.
422	Unprocessable Content	Well-formed but breaks a field or business rule.
428	Precondition Required	A write is missing its If-Match header.
429	Too Many Requests	API rate limit, or a Shield guard throttle (see Retry-After).

Insuretech endpoints tokens: claimant_demo_key · adjuster_demo_key

METHOD	PATH (UNDER $INS)	WHO	WHAT
GET	/policies · /policies/{id}	anyone	Policies with live usage
POST PUT PATCH	/policies · /policies/{id}	adjuster	Issue and amend policies
GET POST	/claims	anyone · claimant	List with filters · file a claim
GET PATCH DELETE	/claims/{id}	anyone · claimant	Read · edit while pending · withdraw
POST	/claims/{id}/decision	adjuster	approve · deny · request_info · escalate
POST	/claims/{id}/pay	adjuster	Pay an approved claim
POST	/claims/{id}/appeal	claimant	Appeal a denial, once
POST	/claims/{id}/documents	claimant	Attach a document
POST	/claims/{id}/restore	adjuster	Undo a withdrawal
GET	/claims/{id}/history	anyone	Audit trail
GET	/stats	anyone	KPIs and loss ratio

Shield endpoints token: xeno_demo_key

METHOD	PATH (UNDER $SHD)	TOKEN	WHAT
POST	/inspections	key	Screen fields; quarantines suspicious ones
GET POST	/quarantine	key (write)	List · add a case manually
GET PUT PATCH DELETE	/quarantine/{id}	key (write)	Read · replace · review · soft delete
POST	/guard/attempts	key	Report a login/signup/API attempt
GET	/guard/status	open	Is this subject blocked?
GET POST	/bans	key (write)	List · ban manually
GET PUT PATCH DELETE	/bans/{id}	key (write)	Read · replace · extend · lift
GET PATCH	/policies/{scope}	key (write)	Read and tune thresholds
GET	/stats	open	Cases, bans, guard decisions

Troubleshooting

`api: command not found`	Run `source scripts/curl-helpers.sh` in this terminal.
`curl: (7) Failed to connect`	The server is not running. Start it with `node server.js`. If port 3000 is taken: `PORT=4000 node server.js` and `export HOST=http://localhost:4000`.
A step returns a different result	Restart with `node server.js --fresh`, then run the steps in order.
`HTTP 429` on every call	You exceeded the API's own rate limit (60-request burst, 10 per second). Wait a second.
Empty `$ID` / `$ETAG`	An earlier step failed. Re-run it and read its output.
Dates are rejected	Incident dates must be real, recent dates: use `$(day -3)`.

These are sandbox demos. State lives in local files, the tokens are public demo keys, and the fraud and detection rules are transparent heuristics, not production models.