

Verifier Starter Kit

W3PATTERNS

FREE STARTER EDITION

Build and run your first business verifier

A practical introduction to independent business verification for developers, operators, and business process owners.

Copyright (c) 2026 W3Patterns. All rights reserved.

START WITH THE BUSINESS QUESTION

Why business verification?

Infrastructure monitoring and business verification solve different, complementary problems. Infrastructure monitoring tells you whether systems are available and healthy. Business verification asks whether the capability people depend on is actually producing the expected outcome.

Infrastructure healthy != business healthy

Infrastructure can be healthy while the business is not:

- The customer portal loads, but customers cannot log in.
- The checkout application is running, but payments cannot be completed.
- The nightly report job ran, but the expected file was not delivered.
- An API responds successfully, but the business data it returns is wrong.
- A database is online, but the record required by an application was not created.

A different question

Infrastructure monitoring remains essential. A BRS verifier adds direct evidence about the workflow or capability the business cares about. It executes observable checks and reports whether the expected business outcome occurred.

The fastest way to understand business verification is to run one, inspect what it observed, and see how it compared reality with expectation.

HANDS-ON DEMONSTRATION

Run a real business verifier

The included public-api example is a real working reference verifier shipped with BRS-Core. It runs independently and uses only Node.js built-in capabilities. BRS-Core, Prometheus, Grafana, and systemd are not required.

Before you begin

Requirement: Node.js 24 or newer and outbound HTTPS access to the public W3Patterns reference endpoint.

From the extracted w3patterns-verifier-starter directory:

```
cd examples
cp public-api/.env.example public-api/.env
node public-api.js
```

No npm install is required. The copied .env file supplies the read-only public endpoint and a 10-second request timeout. Existing process environment values take precedence.

What happens

The verifier makes one request to the Customer Care endpoint, checks the technical response, reads the JSON, and then checks whether the published Customer Care information matches the business expectation encoded in the verifier.

That final comparison is the important step. First, look at the complete result.

LOOK AT WHAT THE VERIFIER RETURNED

One complete successful result

This is the complete JSON from a successful run of the included public-api verifier. Durations and timestamps naturally vary between runs.

```
{
  "summary": "The NYMG public Customer Care API was verified successfully.",
  "steps": [
    {
      "name": "reach_customer_care_service",
      "label": "Reach Customer Care Service",
      "success": true,
      "durationSeconds": 0.268,
      "message": "The NYMG Customer Care service is reachable.",
      "checkedAt": "2026-09-28T23:25:30.557Z"
    },
    {
      "name": "receive_customer_care_response",
      "label": "Receive Customer Care Response",
      "success": true,
      "durationSeconds": 0,
      "message": "The NYMG Customer Care service responded successfully.",
      "checkedAt": "2026-09-28T23:25:30.564Z"
    },
    {
      "name": "read_customer_care_data",
      "label": "Read Customer Care Data",
      "success": true,
      "durationSeconds": 0.002,
      "message": "The NYMG Customer Care response contains valid JSON.",
      "checkedAt": "2026-09-28T23:25:30.566Z"
    },
    {
      "name": "verify_customer_care_information",
      "label": "Verify Customer Care Information",
      "success": true,
      "durationSeconds": 0,
      "message": "The expected NYMG Customer Care information is available.",
      "checkedAt": "2026-09-28T23:25:30.566Z"
    }
  ]
}
```

READ THE EVIDENCE

What did those four steps prove?

The steps deliberately move from technical availability toward business correctness.

1. Reach Customer Care Service

The network request completed and the service could be reached. This is useful technical evidence, but it does not yet prove that the service returned a successful or meaningful response.

2. Receive Customer Care Response

The service returned HTTP 200. The application answered successfully at the HTTP layer, but HTTP 200 alone says nothing about whether the returned Customer Care information is correct.

3. Read Customer Care Data

The response could be parsed as a JSON object. The payload is structurally readable, but readable data can still contain the wrong business values.

4. Verify Customer Care Information

The verifier compared the returned status and Customer Care fields with the values the business expects to publish. This is where ordinary availability checking becomes business verification.

Reachable -> HTTP 200 -> readable JSON -> expected business information

A PASS means every reported step succeeded. A FAIL stops after the first unmet expectation and identifies the business step that needs investigation.

LOOK AT WHAT THE APPLICATION RETURNED

The Customer Care data being verified

The current public endpoint returns this JSON:

```
{
  "status": "ok",
  "customerCare": {
    "eyebrow": "Customer care",
    "heading": "We're here to help.",
    "message": "Call us Monday through Friday, 8:00 a.m.-6:00 p.m. Eastern.",
    "phoneDisplay": "1-800-555-0191",
    "phoneHref": "tel:+18005550191"
  }
}
```

What if the endpoint still returned HTTP 200, but the phone number was wrong?

The server could be up. The API could be reachable. The request could return HTTP 200. Infrastructure monitoring could remain green.

But the business information could still be wrong. A customer could see or call the wrong number even though every technical availability signal looked healthy.

The verifier therefore needs an explicit statement of what the correct business information should be.

DEFINE EXPECTED BUSINESS REALITY

What does the verifier expect?

The public-api implementation defines the expected Customer Care information in one actual JavaScript object:

```
const EXPECTED_CUSTOMER_CARE = Object.freeze({
  eyebrow: 'Customer care',
  heading: 'We're here to help.',
  message: 'Call us Monday through Friday, 8:00 a.m.-6:00 p.m. Eastern.',
  phoneDisplay: '1-800-555-0191',
  phoneHref: 'tel:+18005550191'
});
```

This is not a general health check. It is the expected business reality for this example: the exact heading, service-hours message, display phone number, and telephone link that customers should receive.

The application response is observed reality. The object above is expected reality. The verifier becomes meaningful when it compares the two.

Application response = observed reality. Verifier expectation = expected business reality.

COMPARE REALITY WITH EXPECTATION

Where the business validation happens

This is the actual validation logic used by public-api:

```
function validateCustomerCarePayload(payload) {
  if (payload.status !== 'ok') {
    throw new Error('The API status was not ok.');
```

```
  }

  const customerCare = payload.customerCare;

  if (!customerCare || typeof customerCare !== 'object') {
    throw new Error('The customerCare object was missing.');
```

```
  }

  for (const [field, expectedValue] of Object.entries(
    EXPECTED_CUSTOMER_CARE
  )) {
    if (customerCare[field] !== expectedValue) {
      throw new Error(`The customerCare.${field} value was unexpected.`);
    }
  }
}
```

The function first requires the business status to be ok. It then requires a Customer Care object. Finally, it compares each returned field with the expected value.

PASS means the observed evidence matched the expected business outcome. HTTP 200 was necessary evidence, but it was not sufficient proof.

TURN THE IDEA TOWARD YOUR BUSINESS

What would be worth verifying for you?

Start with curiosity, not code:

- What does your business assume is working right now?
- What must customers or employees be able to do?
- What important process could silently fail?
- Is there a file or report that must arrive every day?
- Could an API return HTTP 200 while returning the wrong data?
- Must a database record exist after an important transaction?
- Is there a payment, order, claim, enrollment, or login whose success matters?
- How do you know that outcome actually happened?
- Would existing monitoring distinguish “the system responded” from “the business outcome was correct”?

Pick one thing

Do not model your entire business. Choose one business capability that matters, then ask:

What observable evidence would prove that it worked?

Evidence should be specific enough that a verifier can observe it and decide PASS or FAIL without guessing.

A SIMPLE WORKED EXAMPLE

From requirement to verification steps

Business requirement

Our partner claims file must arrive every morning.

Observable evidence

- The expected file exists in the approved location.
- Its identity or date matches today's expected delivery.
- It is not empty.

Possible verifier steps

1. Connect to the approved file location.
2. Locate the expected file.
3. Verify its expected identity or date.
4. Verify that it contains data.

Business result

PASS means the observed evidence supports the requirement that today's partner claims file arrived. FAIL means somebody should investigate which expectation was not met.

Business requirement -> observable evidence -> verification steps -> PASS / FAIL

These rules are illustrative, not universal. Your workflow may require a different filename, delivery window, signature, record count, checksum, or other evidence. The important move is translating a business requirement into observable facts.

CANONICAL PATTERN

Put the evidence into a verifier

A verifier is an independently executable CommonJS JavaScript program that implements the BRS verifier contract. The top-level entry file provides dependable standalone behavior and exports the same verifier object that BRS-Core can later operate.

Verifier object

- id - stable lower-snake-case identifier
- label - stable human-readable name
- schedule - optional interval and timeout declarations in seconds
- async verify() - runs the business verification once and returns a result
- summary - non-empty stable result summary
- steps - non-empty array of meaningful business checks
- observations - optional numeric gauges with bounded labels

Each step records

- name and label - stable machine identifier and readable business-check name
- success - a JavaScript Boolean
- durationSeconds - finite, non-negative execution duration
- message - stable business-facing outcome message
- checkedAt - ISO-8601 timestamp

Use the included skeleton

Open `examples/skeleton.js` and `examples/skeleton/skeleton.js`. Together they are the canonical implementation pattern: standalone execution in the entry file, business logic in the same-named directory, local configuration loading, structured results, and PASS/FAIL behavior.

FROM EXPERTISE TO EVIDENCE

Build your verifier with AI

You do not need to be an expert developer. You do need to understand the business process, what must happen, which observable evidence proves it happened, and what constitutes success or failure. AI can help translate that knowledge into verifier code.

Requirement -> evidence -> steps -> implementation

Copy/paste prompt - create a verifier

I'm providing the W3Patterns BRS skeleton.js verifier as the canonical implementation pattern. Preserve its verifier contract, standalone execution, structured JSON output, and exit-code behavior.

I need to verify this business requirement:
[DESCRIBE ONE BUSINESS OUTCOME - for example, our daily partner file arrived]

Before writing code, identify the observable conditions that would prove this requirement is working and propose the verification steps. After the evidence and steps are clear, create the verifier using the supplied skeleton. Keep each meaningful business check separate. Do not change the BRS verifier contract.

Attach or paste both skeleton files. Ask the AI to explain the proposed evidence before it writes code. "I need to verify that our daily partner file arrived" gives the AI a real requirement; "make me a verifier" does not.

PROVE THE PROOF

Review and test what AI built

A technically functioning script is not automatically a useful business verifier. Review whether its checks genuinely establish the intended outcome and whether it preserves the machine contract.

Copy/paste prompt - review a verifier

Review this verifier against the supplied W3Patterns BRS skeleton and my stated business requirement. Determine whether it proves the business outcome rather than merely proving that infrastructure is reachable. Check that it preserves the verifier contract, clean standalone JSON, correct exit status, safe failure handling, and secret protection. Identify gaps before suggesting code changes.

Test it directly

1. Run the verifier and confirm expected PASS behavior.
2. Parse stdout as JSON and confirm diagnostics are not mixed into it.
3. Confirm PASS exits 0.
4. Safely create or simulate a business failure when practical.
5. Confirm FAIL still returns valid JSON and exits 1.
6. Restore the normal condition and confirm PASS again.

```
node examples/your-verifier.js > verification.json
verifier_status=?
node -e "JSON.parse(require('node:fs').readFileSync('verification.json','utf8'))"
printf 'exit=%s\n' "$verifier_status"
```

AUTOMATION

Use the same verifier in CI/CD

The verifier remains the same independently executable program. Only the caller changes. A shell script or pipeline can trust the process exit status while preserving the JSON result as release evidence.

Pre-deployment release gate

Run against an approved test or staging target. Exit 0 permits promotion; exit 1 blocks it.

Post-deployment smoke test

Run once against the deployed target before acceptance. Exit 1 stops acceptance or invokes your rollback procedure.

Manual diagnostic

Run on demand to identify which business step failed and retain the result for investigation.

```
node examples/public-api.js > verification.json
verifier_status=$?
jq -e . verification.json
exit "$verifier_status"
```

Verifier passes -> exit 0. Verifier fails -> exit 1.

Store credentials in the deployment system's approved secret mechanism. Keep a pipeline job-level timeout as final protection. Direct execution runs once; it is not a scheduler.

CI/CD can prove a capability before and immediately after deployment. It does not continuously verify the business afterward.

CONTINUOUS OPERATION

From one verifier to continuous business verification

The verifier you just ran does not require BRS-Core. You can use independently executable verifiers manually, from shell scripts, in CI/CD, and as deployment smoke tests.

BRS-Core becomes useful when you want to operate many approved verifiers continuously as a system.

BRS-Core provides four focused runtime responsibilities

1. Discover top-level verifier entry files.
2. Schedule and isolate their executions.
3. Store the latest result for each verifier in memory.
4. Export those results through Prometheus metrics and a read-only JSON API.

Product boundary

BRS-Core does not install or manage Prometheus, Grafana, Alertmanager, dashboards, alert delivery, TLS termination, or infrastructure monitoring. It adds a business-verification layer alongside the monitoring environment you already operate.

Learn more: <https://w3patterns.com/brs-core.html>

Your next step

Pick one business capability. Define the observable evidence. Build and test one verifier. Use it manually or in CI/CD, then evaluate BRS-Core when recurring isolated execution and metrics become useful.

Starter Kit offer

Starter Kit users may qualify for a BRS-Core purchase discount.

Visit w3patterns.com for the current offer.

W3Patterns - Business verification that starts with the outcome.