Engineer

Contract testing

verifying interface expectations between components

What it is

Contract testing checks that two components agree about the requests, responses or messages they exchange. A parcel tracking screen and a tracking service may each behave correctly in isolation. The screen still breaks if it expects a field called `status` and the service sends `state`. A test can protect the interface by expressing what the screen actually uses and verifying that the service can supply it.

Pact describes one approach in which the consumer runs against a controlled substitute for the provider. Its test records interactions in a contract. The provider then verifies the contract by exercising its own implementation under specified conditions. For messages, the equivalent evidence concerns the observable message consumed and produced. The two sides can check their expectations while they work and release independently, as long as the evidence refers to the versions they intend to use together.

Other teams use contract testing to mean checking one provider against its published interface description, such as an OpenAPI document. That is useful when the written specification is the shared agreement. An interface description can state field shapes, status codes and allowed values. Consumer-driven examples also show which combinations a particular caller uses and what assumptions the caller made. A working arrangement can include both methods. A journey through the real integration still checks things outside the individual exchange, such as routing, credentials and environment configuration.

Use the concept in a decision

Put “What is contract testing?” to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

An interface often outlives the code that first implemented it. A service team can rename a field and update its own tests without changing a separately maintained caller. A mobile client may continue to run an older version for months. Even a small type change matters if a caller uses it to choose what to display. A contract test can make those dependencies visible before a change ships.

Separate release timing makes compatibility information useful. An old consumer might support only a handful of status values. A new provider may need to add an intermediate status. The teams can add support in the consumer first, continue to accept both forms during the transition and remove the old behavior after confirming the depended-on versions no longer use it. Which order is safe depends on the recorded contracts, real usage and the system's release policy. A successful test of one recent pair does not prove compatibility for all clients still in use.

Tests should express the meaningful points of agreement. If any well-formed reference is fine, match the reference format and avoid requiring an exact sample value. If a field determines customer-visible behavior, check its type and the values the client understands. A carefully chosen example preserves room for the provider to change internals and for the consumer to keep using agreed behavior.

What it looks like in practice

Consider a fictional parcel tracking app. It requests a parcel by its reference. When a parcel exists and is in transit, the screen expects a success status, the parcel reference and a text value it can map to an in-transit label. When a parcel is unknown, the screen expects a distinct response and shows a way to check the reference. Each interaction is tied to a precondition, such as "the requested parcel is in transit" or "there is no parcel with that reference".

Test the consumer through its actual request code against the controlled substitute. Confirm the application displays the appropriate result from each example. Then verify the same interactions on the provider's actual response path. Create deterministic fixtures for the relevant provider state and reset them between checks. If the test uses a stub for its database or for an external carrier, keep the request parsing, application logic and response serialization that the consumer depends on in the test path.

Try changing the provider's `status` field to `state` while leaving the current consumer unchanged. Verification should fail for the missing field and identify the affected interaction. A new optional field that the consumer ignores should be allowed if the agreement permits it. Keep the separate verification runs tied to their genuine versions, because combining a new provider result and an old contract sample without provenance makes it hard to know which pair was tested.

In our note on opening a dApp in Solana mobile wallets, several integrations expect different deep-link paths and query parameter encoding. The note records those actual differences and includes device checks. A small contract-style verification of generated URLs could protect the request shape for each wallet. An actual device run can show whether a browser passes the link to the installed app and whether the journey completes. The two observations answer different questions.

Where it goes wrong

Contracts can become too rigid when every response byte has to match one saved example. A provider may be free to add an unused field or change the formatting of an independent message. Make a field mandatory when a caller uses it or when the public specification requires it. Declare ordering, units or exact text only when the agreement needs them. Leave harmless variability available.

A test also loses value when it bypasses the real boundary. Verifying a hand-written example of a JSON object exercises the example and may never run the serializer. Testing only that the substitute returned the expected fixture can leave the consumer's parser untouched. Run the real request, serialization or parsing code on the side under test. Record which side was exercised and what its dependencies supplied.

Finally, one recorded happy path says little about missing data, unknown references or versions that still run in the field. Identify the error paths the consumer handles and the compatibility obligations the provider actually supports. Keep enough integration testing to exercise configuration and routing. A clear contract gives teams a smaller, faster piece of evidence they can use on every relevant change.

Further reading

- Pact documentation on what contract testing checks
- How Pact verifies consumers and providers
- Pact documentation on provider states
- OpenAPI Specification for describing HTTP interfaces

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is contract testing?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/contract-testing/