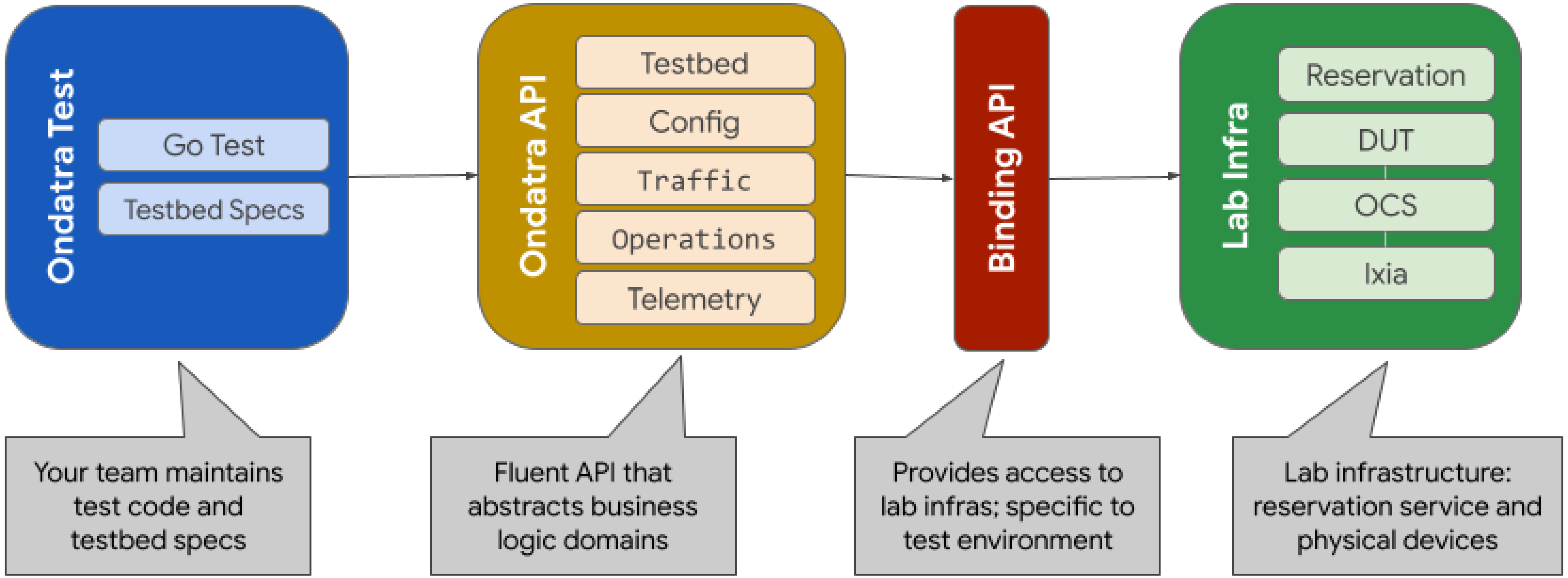
A Tour of Ondatra

What is Ondatra?

Open **N**etwork **D**evice **A**utomated **T**est **R**unner and **A**PI

Ondatra is a framework for writing and running tests against real network devices.



Your team maintains test code and testbed specs

Fluent API that abstracts business logic domains

Provides access to lab infras; specific to test environment

Lab infrastructure: reservation service and physical devices

Creating a testbed

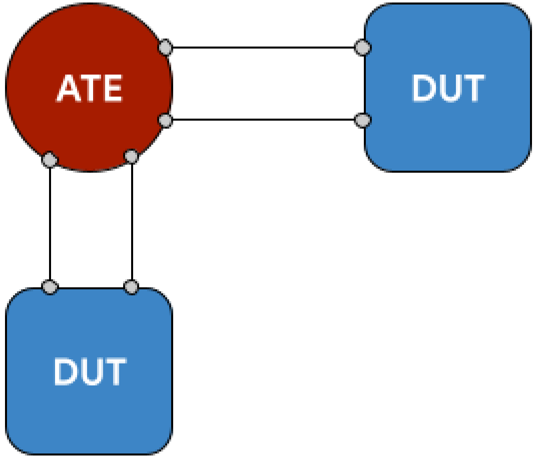
A **testbed** is a collection of network devices and the links between them.

Every device in a testbed is of **one of two types**:

- **DUT**: device under test, e.g. a router; or
- **ATE**: automated test equipment, e.g. Ixia

You specify the testbed in a simple proto (example)

- You choose unique IDs for the devices, not actual devices names
- *No* hard-coding of specific devices; only abstact criteria like the vendor



Tests typically send traffic from the ATE to the DUT and check the DUT behaves as expected in response.

Using the testbed

Every Ondatra test reserves a single testbed for exclusive use until the test completes.

That is accomplished by the following snippet, which must appear in every Ondatra test.

```
func TestMain(m *testing.M) {
  ondatra.RunTests(m, b2bindinit.Init)
}
```

Test cases then lookup DUTs and ATEs by their unique IDs given in the testbed proto, e.g.:

```
func TestGetDevicesFromTestbed(t *testing.T) {
  ate := ondatra.ATE(t, "ate")
  dut := ondatra.DUT(t, "dut")
  doSomethingWith(ate, dut)
}
```

API Principles

The Ondatra API **aims to be fluent**

- uses chained method calls that read like English prose
- influenced by Network Engineers interested in the Nokia Robot framework
- method chaining makes the API easily discoverable through code completion

```
gnoi.Execute(t, dut, system.NewPingOperation().Destination("8.8.8.8"))
```

The Ondatra API **aims to make it clear what is being tested**

- Ixia configuration is part of the test, not in a separate config file
- validation logic is part of the test, not in centralized validators

DUT Overview

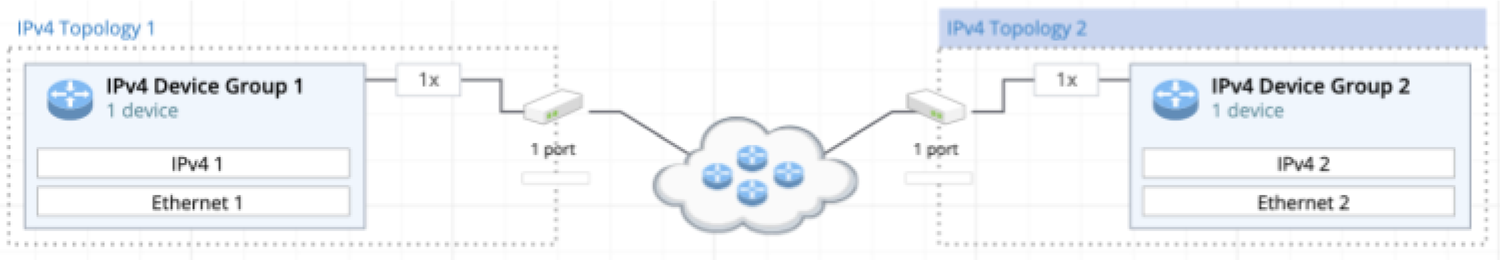
Ondatra offers vendor-neutral DUT APIs with contributions from Arista, Cisco, Nokia, and Juniper to ensure compatibility.

DUTs support the following APIs:

API Name	**Method to access**	**Purpose**
Config	`dut.Config()`	to push vendor config to the device
Operations	`gnoi.Execute()`	to execute gnoi commands on the device
Telemetry	`gnmi.<Func>`	to query values and statistics about the state of the device

ATE Overview

Ondatra supports one ATE vendor today: Ixia



ATEs support the following APIs:

API Name	Method to access	Purpose
Topology	`ate.Topology()`	to create a simulated network topology on the device
Traffic	`ate.Traffic()`	to generate traffic flows from the device
Actions	`ate.Actions()`	to execute operational commands on the device
Telemetry	`gnmi.<Func>`	to query values and statistics about the state of the device

DUT Configuration

Use the Config API to push config to a DUT.

Every device is initialized with a minimal **baseline config** that ensures the device is reachable and manageable.

Use the API to set different config for each vendor so that the **same test can be run against multiple vendors**.

You can **also set OpenConfig**, which will be pushed if the test doesn't set vendor-specific config for the DUT.

Every push does a **full config replace** of the existing config with the **baseline + test-specified config**.

```go
func TestPushConfig(t *testing.T) {
  dut := ondatra.DUT(t, "dut")
  dut.Config().New().
    WithAristaText(`
      interface {{ port "port1" }}
       description From Ixia
       no switchport
       ip address 192.168.31.1/30`).
    WithCiscoFile("path/to/cisco_config.txt").
    Push(t)
}
```

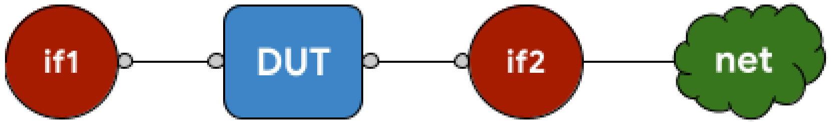
ATE Topology

Use the Topology API to create a simulated network topology on an ATE.

Add one or more logical *interfaces* to the toplogy.

You may optionally add any number of *networks*, simulated clouds of devices, behind the interfaces.

Each interface and network may be configured with any combination of the IPv4, IPv6, BGP, and ISIS protocols.



```go
func TestCreateTopology(t *testing.T) {
  ate := ondatra.ATE(t, "ate")
  ap1 := ate.Port(t, "port1")
  ap2 := ate.Port(t, "port2")
  top := ate.Topology().New()
  if1 := top.AddInterface("if1").WithPort(ap1)
  if2 := top.AddInterface("if2").WithPort(ap2)
  if1.IPv4().
    WithAddress("192.168.31.2/30").
    WithDefaultGateway("192.168.31.1")
  if2.IPv4().
    WithAddress("192.168.32.2/30").
    WithDefaultGateway("192.168.32.1")
  net := if2.AddNetwork("net")
  net.IPv4().
    WithAddress("192.168.40.0/30").
    WithCount(100)
  top.Push(t).StartProtocols(t)
}
```

ATE Traffic

Use the Traffic API to generate traffic from an ATE.

Create any number of traffic *flows* on the topology.
For each flow you may specify:

- src and dst endpoints: interface, network, or port
- packet headers: Ethernet, IPv4, IPv6, GRE, etc
- frame rate: BPS, FPS, or percent line rate
- frame size: fixed or random value, or IMIX presets

To run the traffic:

1. start the traffic flows
2. sleep for the desired traffic duration
3. stop the traffic
4. optionally get stats about the traffic from telemetry

```go
func TestGenerateTraffic(t *testing.T) {
  ate := ondatra.ATE(t, "ate")
  ap1 := ate.Port(t, "port1")
  ap2 := ate.Port(t, "port2")
  top := ate.Topology().New()
  if1 := top.AddInterface("if1").WithPort(ap1)
  if2 := top.AddInterface("if2").WithPort(ap2)
  flow := ate.Traffic().NewFlow("Flow1").
    WithSrcEndpoints(if1).
    WithDstEndpoints(if2).
    WithFrameRatePct(50)
  ate.Traffic().Start(t, flow)
  time.Sleep(3 * time.Minute)
  ate.Traffic().Stop(t)
}
```

Device Telemetry

Use the Telemetry API to query properties and statistics from DUTs and ATEs.

The API lets you construct a gNMI path to the value of interest and then query it in one of two ways:

- *Get*: retrieves the current value at the path
- *Collect*: retrieves the value over a period of time

```
ds := gnmi.Get(t, dut, gnmi.OC().Interface(dp.Name()).OperStatus().State())
as := gnmi.Get(t, ate, gnmi.OC().Interface(ap.Name()).OperStatus().State())
if want := oc.Interface_OperStatus_UP; ds != want {
  t.Errorf("Get(DUT port1 status): got %v, want %v", ds, want)
}
if want := oc.Interface_OperStatus_UP; as != want {
  t.Errorf("Get(ATE port1 status): got %v, want %v", as, want)
}

p := gnmi.Collect(t, dut, gnmi.OC().Interface(`{{ port "port2" }}`).Counters().InPkts().State(),
if got := netutil.MeanRate(t, p.Await(t)); got != expectedRate {
  t.Fatalf("Got unexpected input rate %.4f, want: %.4f", got, expectedRate)
```

Device Operations

Use the Operations API to execution operational commands on DUTs or ATEs.

The API is for I/O or side-effecting operations like:

- send a ping from a device
- turn up or down an interface on a device
- reload a linecard on a device

The Operations API is *not* for "show" commands that retrieve device state — use the Telemetry API instead.

```
func TestExecuteOperation(t *testing.T) {
  dut := ondatra.DUT(t, "dut")
  gnoi.Execute(t, dut, system.NewPingOperatior

  ate := ondatra.ATE(t, "ate")
  ap := dut.Port(t, "port1")
  ate.Actions().
    NewSetPortState().
    WithPort(ap).
    WithEnabled(false).
    Send(t)
}
```