

NEWS Free Terraform Enterprise tier for practitioners and small teams, affordable paid tiers for businesses. Read more →
(<https://www.hashicorp.com/blog/terraform-collaboration-for-everyone>)



Write, Plan, and Create Infrastructure as Code

[GET STARTED \(/INTRO/INDEX.HTML\)](#)

[DOWNLOAD 0.11.11 \(/DOWNLOADS.HTML\)](#)

[FIND MODULES \(HTTPS://REGISTRY.TERRAFORM.IO/\)](#)

SIMPLE AND POWERFUL

HashiCorp Terraform enables you to safely and predictably create, change, and improve infrastructure. It is an open source tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned.

WRITE
INFRASTRUCTURE AS CODE



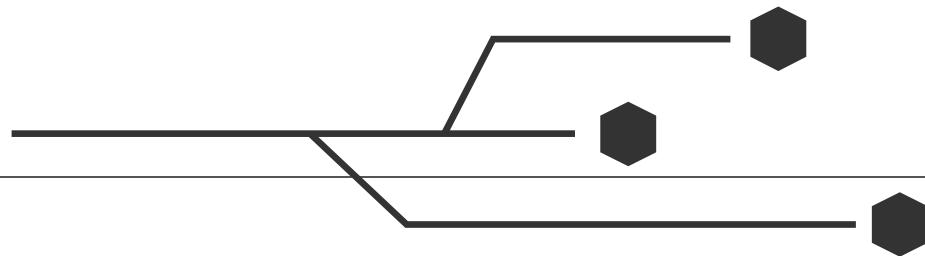
PLAN

PREVIEW CHANGES BEFORE APPLYING



CREATE

REPRODUCIBLE INFRASTRUCTURE



WRITE

INFRASTRUCTURE AS CODE

Define infrastructure as code to increase operator productivity and transparency.

COLLABORATE & SHARE

Terraform configuration can be stored in version control, shared, and collaborated on by teams of operators.

EVOLVE YOUR INFRASTRUCTURE

Track the complete history of infrastructure versions.

AUTOMATION FRIENDLY

If it can be codified, it can be automated.

PLAN

ONE SAFE WORKFLOW ACROSS PROVIDERS

Terraform provides an elegant user experience for operators to safely and predictably make changes to infrastructure.

MAP RESOURCE DEPENDENCIES

Understand how a minor change could have potential cascading effects across an infrastructure before executing that change. Terraform builds a dependency graph from the configurations, and walks this graph to generate plans, refresh state, and more.

SEPARATION OF PLAN & APPLY

Separating plans and applies reduces mistakes and uncertainty at scale. Plans show operators what would happen, applies execute changes.

ONE SAFE WORKFLOW

Use Terraform to create resources across all major infrastructure providers (AWS, GCP, Azure, OpenStack, VMware, and more).

CREATE

REPRODUCIBLE INFRASTRUCTURE

Terraform lets operators easily use the same configurations in multiple places to reduce mistakes and save time.

ENVIRONMENT PARITY

Use the same Terraform configuration to provision identical staging, QA, and production environments.

SHAREABLE MODULES

Common Terraform configurations can be packaged as modules and used across teams and organizations.

COMBINE MULTIPLE PROVIDERS CONSISTENTLY

Terraform allows you to effortlessly combine high-level system providers. Launch a server from one cloud provider, add a DNS entry with its IP with a different provider. Built-in dependency resolution means things happen in the right order.

LATEST NEWS

Terraform 0.11.0 Released

Terraform 0.11.0 is out! The latest update includes improvements to Terraform Registry integration, per-module provider configuration, and improvements to CLI workflow. There have also been improvements to a number of major providers. See the blog post for all the details!

[READ MORE \(HTTPS://WWW.HASHICORP.COM/BLOG/HASHICORP-TERRAFORM-0-11\)](https://www.hashicorp.com/blog/hashicorp-terraform-0-11)

Webinar: Controlling Your Organization With HashiCorp Terraform and Google Cloud Platform

Watch our recent webinar with Seth Vargo and Google Cloud. Learn how to build your entire infrastructure across Google Cloud with one command.

[WATCH NOW \(HTTPS://WWW.YOUTUBE.COM/WATCH?V=YM6DTUX5REG\)](https://www.youtube.com/watch?v=YM6DTUX5REG)

Webinar: Multi-Cloud, One Command with Terraform

Watch our recent webinar with Mitchell Hashimoto to learn how Terraform provisions infrastructure across different clouds using a consistent workflow.

[WATCH NOW
\(HTTPS://YOUTU.BE/NLG7FPVCIV4\)](https://youtu.be/nLG7FPVCIV4)

EXAMPLES

COMPOSING RESOURCES

Use attributes from other resources to create an infrastructure composed of resources across multiple providers.

• • •

```
resource "digitalocean_droplet" "web" {
  name      = "tf-web"
  size      = "512mb"
  image     = "centos-5-8-x32"
  region    = "sfo1"
}

resource "dnsimple_record" "hello" {
  domain   = "example.com"
  name     = "test"
  value    = "${digitalocean_droplet.web.ipv4_address}"
  type     = "A"
}
```

FAST, SIMPLIFIED INTERACTION

Simple and intuitive configuration makes even the most complicated services approachable: no more web consoles, loading bars, or confusing CLI clients.

• • •

```
resource "aws_elb" "frontend" {
  name = "frontend-load-balancer"
  listener {
    instance_port      = 8000
    instance_protocol = "http"
    lb_port           = 80
    lb_protocol       = "http"
  }
  instances = ["${aws_instance.app.*.id}"]
}

resource "aws_instance" "app" {
  count = 5

  ami          = "ami-408c7f28"
  instance_type = "t1.micro"
}
```

The intro contains a walkthrough guide, introductory literature, and a range of examples to experiment with Terraform.

GET STARTED
(/INTRO/INDEX.HTML)

Collaborative Infrastructure Automation for organizations. Collaborate on Terraform configurations, validate changes, and automate provisioning across providers.

LEARN MORE
([HTTPS://WWW.HASHICORP.COM/TERRAFORM.HTML](https://www.hashicorp.com/terraform.html))

```

(function() {
var __customevents = {}, __protocol = {}, __connector = {}, __timer = {}, __options = {}, __reloader = {},
__livereload = {}, __less = {}, __startup = {};
// customevents
var CustomEvents;
CustomEvents = {
bind: function(element, eventName, handler) {
  if (element.addEventListener) {
    return element.addEventListener(eventName, handler, false);
  } else if (element.attachEvent) {
    element[eventName] = 1;
    return element.attachEvent('onpropertychange', function(event) {
      if (event.propertyName === eventName) {
        return handler();
      }
    });
  } else {
    throw new Error("Attempt to attach custom event " + eventName + " to something which isn't a DOMElement");
  }
},
fire: function(element, eventName) {
  var event;
  if (element.addEventListener) {
    event = document.createEvent('HTMLEvents');
    event.initEvent(eventName, true, true);
    return document.dispatchEvent(event);
  } else if (element.attachEvent) {
    if (element[eventName]) {
      return element[eventName]++;
    }
  } else {
    throw new Error("Attempt to fire custom event " + eventName + " on something which isn't a DOMElement");
  }
}
};
__customevents.bind = CustomEvents.bind;
__customevents.fire = CustomEvents.fire;

// protocol
var PROTOCOL_6, PROTOCOL_7, Parser, ProtocolError,
var __indexOf = Array.prototype.indexOf || function(item) {
  for (var i = 0, l = this.length; i < l; i++) {
    if (this[i] === item) return i;
  }
  return -1;
},
__protocol.PROTOCOL_6 = PROTOCOL_6 = 'http://livereload.com/protocols/official-6';
__protocol.PROTOCOL_7 = PROTOCOL_7 = 'http://livereload.com/protocols/official-7';
__protocol.ProtocolError = ProtocolError = (function() {
  function ProtocolError(reason, data) {
    this.message = "LiveReload protocol error (" + reason + ") after receiving data: \\" + data + "\(".");
  }
  return ProtocolError;
})();
__protocol.Parser = Parser = (function() {
  function Parser(handlers) {
    this.handlers = handlers;
    this.reset();
  }
  Parser.prototype.reset = function() {
    return this.protocol = null;
  };
  Parser.prototype.process = function(data) {
    var command, message, options, _ref;
    try {
      if (!(this.protocol != null)) {
        if (data.match(/^!ver:([\d.]+)$/)) {
          this.protocol = 6;
        } else if (message = this._parseMessage(data, ['hello'])) {
          if (!message.protocols.length) {
            throw new ProtocolError("no protocols specified in handshake message");
          } else if (__indexOf.call(message.protocols, PROTOCOL_7) >= 0) {
            this.protocol = 7;
          } else if (__indexOf.call(message.protocols, PROTOCOL_6) >= 0) {
            this.protocol = 6;
          } else {
            throw new ProtocolError("no supported protocols found");
          }
        }
        return this.handlers.connected(this.protocol);
      } else if (this.protocol === 6) {
        message = JSON.parse(data);
        if (!message.length) {
          throw new ProtocolError("protocol 6 messages must be arrays");
        }
        command = message[0], options = message[1];
        if (command !== 'refresh') {
          throw new ProtocolError("unknown protocol 6 command");
        }
        return this.handlers.message({
          command: 'reload',
          path: options.path,
          liveCSS: (_ref = options.apply_css_live) != null ? _ref : true
        });
      } else {
        message = this._parseMessage(data, ['reload', 'alert']);
        return this.handlers.message(message);
      }
    } catch (e) {

```

```

        if (e instanceof ProtocolError) {
            return this.handlers.error(e);
        } else {
            throw e;
        }
    }
};

Parser.prototype._parseMessage = function(data, validCommands) {
    var message, _ref;
    try {
        message = JSON.parse(data);
    } catch (e) {
        throw new ProtocolError('unparsable JSON', data);
    }
    if (!message.command) {
        throw new ProtocolError('missing "command" key', data);
    }
    if (_ref = message.command, indexOf.call(validCommands, _ref) < 0) {
        throw new ProtocolError("invalid command '" + message.command + "'", only valid commands are: " +
(validCommands.join(', ')) + ")", data);
    }
    return message;
};
return Parser;
})();

// connector
// Generated by CoffeeScript 1.3.3
var Connector, PROTOCOL_6, PROTOCOL_7, Parser, Version, _ref;

_ref = __protocol, Parser = _ref.Parser, PROTOCOL_6 = _ref.PROTOCOL_6, PROTOCOL_7 = _ref.PROTOCOL_7;
Version = '2.0.8';

__connector.Connector = Connector = (function() {

    function Connector(options, WebSocket, Timer, handlers) {
        var _this = this;
        this.options = options;
        this.WebSocket = WebSocket;
        this.Timer = Timer;
        this.handlers = handlers;
        this._uri = "ws://" + this.options.host + ":" + this.options.port + "/livereload";
        this._nextDelay = this.options.mindelay;
        this._connectionDesired = false;
        this.protocol = 0;
        this.protocolParser = new Parser({
            connected: function(protocol) {
                _this.protocol = protocol;
                _this._handshakeTimeout.stop();
                _this._nextDelay = _this.options.mindelay;
                _this._disconnectionReason = 'broken';
                return _this.handlers.connected(protocol);
            },
            error: function(e) {
                _this.handlers.error(e);
                return _this._closeOnError();
            },
            message: function(message) {
                return _this.handlers.message(message);
            }
        });
        this._handshakeTimeout = new Timer(function() {
            if (!_this._isSocketConnected()) {
                return;
            }
            _this._disconnectionReason = 'handshake-timeout';
            return _this.socket.close();
        });
        this._reconnectTimer = new Timer(function() {
            if (!_this._connectionDesired) {
                return;
            }
            return _this.connect();
        });
        this.connect();
    }

    Connector.prototype._isSocketConnected = function() {
        return this.socket && this.socket.readyState === this.WebSocket.OPEN;
    };

    Connector.prototype.connect = function() {
        var _this = this;
        this._connectionDesired = true;
        if (this._isSocketConnected()) {
            return;
        }
        this._reconnectTimer.stop();
        this._disconnectionReason = 'cannot-connect';
        this.protocolParser.reset();
        this.handlers.connecting();
        this.socket = new this.WebSocket(this._uri);
        this.socket.onopen = function(e) {
            return _this._onopen(e);
        };
        this.socket.onclose = function(e) {
            return _this._onclose(e);
        };
        this.socket.onmessage = function(e) {

```

```

        return _this._onmessage(e);
    };
    return this.socket.onerror = function(e) {
        return _this._onerror(e);
    };
};

Connector.prototype.disconnect = function() {
    this._connectionDesired = false;
    this._reconnectTimer.stop();
    if (!this._isSocketConnected()) {
        return;
    }
    this._disconnectionReason = 'manual';
    return this.socket.close();
};

Connector.prototype.scheduleReconnection = function() {
    if (!this._connectionDesired) {
        return;
    }
    if (!this._reconnectTimer.running) {
        this._reconnectTimer.start(this._nextDelay);
        return this._nextDelay = Math.min(this.options.maxdelay, this._nextDelay * 2);
    }
};

Connector.prototype.sendCommand = function(command) {
    if (this.protocol == null) {
        return;
    }
    return this._sendCommand(command);
};

Connector.prototype._sendCommand = function(command) {
    return this.socket.send(JSON.stringify(command));
};

Connector.prototype._closeOnError = function() {
    this._handshakeTimeout.stop();
    this._disconnectionReason = 'error';
    return this.socket.close();
};

Connector.prototype._onopen = function(e) {
    var hello;
    this.handlers.socketConnected();
    this._disconnectionReason = 'handshake-failed';
    hello = {
        command: 'hello',
        protocols: [PROTOCOL_6, PROTOCOL_7]
    };
    hello.ver = Version;
    if (this.options.ext) {
        hello.ext = this.options.ext;
    }
    if (this.options.extver) {
        hello.extver = this.options.extver;
    }
    if (this.options.snipver) {
        hello.snipver = this.options.snipver;
    }
    this._sendCommand(hello);
    return this._handshakeTimeout.start(this.options.handshake_timeout);
};

Connector.prototype._onclose = function(e) {
    this.protocol = 0;
    this.handlers.disconnected(this._disconnectionReason, this._nextDelay);
    return this._scheduleReconnection();
};

Connector.prototype._onerror = function(e) {};

Connector.prototype._onmessage = function(e) {
    return this.protocolParser.process(e.data);
};

return Connector;
})();

// timer
var Timer;
var __bind = function(fn, me){ return function(){ return fn.apply(me, arguments); }; };
__timer.Timer = Timer = (function() {
    function Timer(func) {
        this.func = func;
        this.running = false;
        this.id = null;
        this._handler = __bind(function() {
            this.running = false;
            this.id = null;
            return this.func();
        }, this);
    }
    Timer.prototype.start = function(timeout) {
        if (this.running) {
            clearTimeout(this.id);
        }
    }
});

```

```

this.id = setTimeout(this._handler, timeout);
return this.running = true;
};
Timer.prototype.stop = function() {
  if (this.running) {
    clearTimeout(this.id);
    this.running = false;
    return this.id = null;
  }
};
return Timer;
})();
Timer.start = function(timeout, func) {
  return setTimeout(func, timeout);
};

// options
var Options;
_options.Options = Options = (function() {
  function Options() {
    this.host = null;
    this.port = RACK_LIVERELOAD_PORT;
    this.snipver = null;
    this.ext = null;
    this.extver = null;
    this.mindelay = 1000;
    this.maxdelay = 60000;
    this.handshake_timeout = 5000;
  }
  Options.prototype.set = function(name, value) {
    switch (typeof this[name]) {
      case 'undefined':
        break;
      case 'number':
        return this[name] = +value;
      default:
        return this[name] = value;
    }
  };
  return Options;
})();
Options.extract = function(document) {
  var element, keyAndValue, m, mm, options, pair, src, _i, _j, _len, _len2, _ref, _ref2;
  _ref = document.getElementsByTagName('script');
  for (_i = 0, _len = _ref.length; _i < _len; _i++) {
    element = _ref[_i];
    if ((src = element.src) && (m = src.match(/^[:]+:\/\/(.*)\z?livereload\.js(?:\?:(.*)?)$/))) {
      options = new Options();
      if (mm = m[1].match(/^\:[^/:]+(:(\d+))?\$/)) {
        options.host = mm[1];
        if (mm[2]) {
          options.port = parseInt(mm[2], 10);
        }
      }
      if (m[2]) {
        _ref2 = m[2].split('&');
        for (_j = 0, _len2 = _ref2.length; _j < _len2; _j++) {
          pair = _ref2[_j];
          if ((keyAndValue = pair.split('=')).length > 1) {
            options.set(keyAndValue[0].replace(/\-/g, '_'), keyAndValue.slice(1).join('='));
          }
        }
      }
      return options;
    }
  }
  return null;
};

// reloader
// Generated by CoffeeScript 1.3.1
(function() {
  var IMAGE_STYLES, Reloader, numberOfMatchingSegments, pathFromUrl, pathsMatch, pickBestMatch, splitUrl;

  splitUrl = function(url) {
    var hash, index, params;
    if ((index = url.indexOf('#')) >= 0) {
      hash = url.slice(index);
      url = url.slice(0, index);
    } else {
      hash = '';
    }
    if ((index = url.indexOf('?')) >= 0) {
      params = url.slice(index);
      url = url.slice(0, index);
    } else {
      params = '';
    }
    return {
      url: url,
      params: params,
      hash: hash
    };
  };

  pathFromUrl = function(url) {
    var path;
    url = splitUrl(url).url;
    if (url.indexOf('file://') === 0) {
      path = url.replace(/^file:\/\/\/(localhost)?/, '');
    }
  };
});

```

```

} else {
  path = url.replace(/^([:]+:)?\//\/([:^\/]+)(:\d*)?\//, '/');
}
return decodeURIComponent(path);
};

pickBestMatch = function(path, objects, pathFunc) {
var bestMatch, object, score, _i, _len;
bestMatch = {
  score: 0
};
for (_i = 0, _len = objects.length; _i < _len; _i++) {
  object = objects[_i];
  score = numberOfMatchingSegments(path, pathFunc(object));
  if (score > bestMatch.score) {
    bestMatch = {
      object: object,
      score: score
    };
  }
}
if (bestMatch.score > 0) {
  return bestMatch;
} else {
  return null;
}
};

numberOfMatchingSegments = function(path1, path2) {
var comps1, comps2, eqCount, len;
path1 = path1.replace(/\//g, '').toLowerCase();
path2 = path2.replace(/\//g, '').toLowerCase();
if (path1 === path2) {
  return 10000;
}
comps1 = path1.split('/').reverse();
comps2 = path2.split('/').reverse();
len = Math.min(comps1.length, comps2.length);
eqCount = 0;
while (eqCount < len && comps1[eqCount] === comps2[eqCount]) {
  ++eqCount;
}
return eqCount;
};

pathsMatch = function(path1, path2) {
  return numberOfMatchingSegments(path1, path2) > 0;
};

IMAGE_STYLES = [
{
  selector: 'background',
  styleNames: ['backgroundImage']
}, {
  selector: 'border',
  styleNames: ['borderImage', 'webkitBorderImage', 'MozBorderImage']
}
];

_reloader.Reloader = Reloader = (function() {

Reloader.name = 'Reloader';

function Reloader(window, console, Timer) {
  this.window = window;
  this.console = console;
  this.Timer = Timer;
  this.document = this.window.document;
  this.importCacheWaitPeriod = 200;
  this.plugins = [];
}

Reloader.prototype.addPlugin = function(plugin) {
  return this.plugins.push(plugin);
};

Reloader.prototype.analyze = function(callback) {
  return results;
};

Reloader.prototype.reload = function(path, options) {
  var plugin, _base, _i, _len, _ref;
  this.options = options;
  if (_base = this.options.stylesheetReloadTimeout == null) {
    _base.stylesheetReloadTimeout = 15000;
  }
  _ref = this.plugins;
  for (_i = 0, _len = _ref.length; _i < _len; _i++) {
    plugin = _ref[_i];
    if (plugin.reload && plugin.reload(path, options)) {
      return;
    }
  }
  if (options.liveCSS) {
    if (path.match(/\.\css$/i)) {
      if (this.reloadStylesheet(path)) {
        return;
      }
    }
  }
}

```

```

if (options.liveImg) {
    if (path.match(/\.(jpe?g|png|gif)$/i)) {
        this.reloadImages(path);
        return;
    }
}
return this.reloadPage();
};

Reloader.prototype.reloadPage = function() {
    return this.window.document.location.reload();
};

Reloader.prototype.reloadImages = function(path) {
    var expando, img, selector, styleNames, styleSheet, _i, _j, _k, _l, _len, _len1, _len2, _len3, _ref,
    _ref1, _ref2, _ref3, _results;
    expando = this.generateUniqueString();
    ref = this.document.images;
    for (_i = 0, _len = _ref.length; _i < _len; _i++) {
        img = _ref[_i];
        if (pathsMatch(path, pathFromUrl(img.src))) {
            img.src = this.generateCacheBustUrl(img.src, expando);
        }
    }
    if (this.document.querySelectorAll) {
        for (_j = 0, _len1 = IMAGE_STYLES.length; _j < _len1; _j++) {
            _ref1 = IMAGE_STYLES[_j], selector = _ref1.selector, styleNames = _ref1.styleNames;
            _ref2 = this.document.querySelectorAll("[style*=" + selector + "]");
            for (_k = 0, _len2 = _ref2.length; _k < _len2; _k++) {
                img = _ref2[_k];
                this.reloadStyleImages(img.style, styleNames, path, expando);
            }
        }
    }
    if (this.document.styleSheets) {
        _ref3 = this.document.styleSheets;
        _results = [];
        for (_l = 0, _len3 = _ref3.length; _l < _len3; _l++) {
            styleSheet = _ref3[_l];
            _results.push(this.reloadStylesheetImages(styleSheet, path, expando));
        }
        return _results;
    }
};

Reloader.prototype.reloadStylesheetImages = function(styleSheet, path, expando) {
    var rule, rules, styleNames, _i, _j, _len, _len1;
    try {
        rules = styleSheet != null ? styleSheet.cssRules : void 0;
    } catch (e) {}

    if (!rules) {
        return;
    }
    for (_i = 0, _len = rules.length; _i < _len; _i++) {
        rule = rules[_i];
        switch (rule.type) {
            case CSSRule.IMPORT_RULE:
                this.reloadStylesheetImages(rule.styleSheet, path, expando);
                break;
            case CSSRule.STYLE_RULE:
                for (_j = 0, _len1 = IMAGE_STYLES.length; _j < _len1; _j++) {
                    styleNames = IMAGE_STYLES[_j].styleNames;
                    this.reloadStyleImages(rule.style, styleNames, path, expando);
                }
                break;
            case CSSRule.MEDIA_RULE:
                this.reloadStylesheetImages(rule, path, expando);
        }
    }
};

Reloader.prototype.reloadStyleImages = function(style, styleNames, path, expando) {
    var newValue, styleName, value, _i, _len,
        this = this;
    for (_i = 0, _len = styleNames.length; _i < _len; _i++) {
        styleName = styleNames[_i];
        value = style[styleName];
        if (typeof value === 'string') {
            newValue = value.replace(/\burl\s*\(((^)]*)\)\//, function(match, src) {
                if (pathsMatch(path, pathFromUrl(src))) {
                    return "url(" + (_this.generateCacheBustUrl(src, expando)) + ")";
                } else {
                    return match;
                }
            });
            if (newValue !== value) {
                style[styleName] = newValue;
            }
        }
    }
};

Reloader.prototype.reloadStylesheet = function(path) {
    var imported, link, links, match, style, _i, _j, _k, _l, _len, _len1, _len2, _len3, _ref, _ref1,
        this = this;
    links = (function() {
        var _i, _len, _ref, _results;
        _ref = this.document.getElementsByTagName('link');

```

```

_results = [];
for (_i = 0, _len = _ref.length; _i < _len; _i++) {
  link = _ref[_i];
  if (link.rel === 'stylesheet' && !link.__LiveReload_pendingRemoval) {
    _results.push(link);
  }
}
return _results;
}).call(this);
imported = [];
_ref = this.document.getElementsByTagName('style');
for (_i = 0, _len = _ref.length; _i < _len; _i++) {
  style = _ref[_i];
  if (style.sheet) {
    this.collectImportedStylesheets(style, style.sheet, imported);
  }
}
for (_j = 0, _len1 = links.length; _j < _len1; _j++) {
  link = links[_j];
  this.collectImportedStylesheets(link, link.sheet, imported);
}
if (this.window.StyleFix && this.document.querySelectorAll) {
  ref1 = this.document.querySelectorAll('style[data-href]');
  for (_k = 0, _len2 = ref1.length; _k < _len2; _k++) {
    style = ref1[_k];
    links.push(style);
  }
}
this.console.log("LiveReload found " + links.length + " LINKed stylesheets, " + imported.length + " imported stylesheets");
match = pickBestMatch(path, links.concat(imported), function(l) {
  return pathFromUrl(_this.linkHref(l));
});
if (match) {
  if (match.object.rule) {
    this.console.log("LiveReload is reloading imported stylesheet: " + match.object.href);
    this.reattachImportedRule(match.object);
  } else {
    this.console.log("LiveReload is reloading stylesheet: " + (this.linkHref(match.object)));
    this.reattachStylesheetLink(match.object);
  }
} else {
  this.console.log("LiveReload will reload all stylesheets because path '" + path + "' did not match any specific one");
  for (_l = 0, _len3 = links.length; _l < _len3; _l++) {
    link = links[_l];
    this.reattachStylesheetLink(link);
  }
}
return true;
};

Reloader.prototype.collectImportedStylesheets = function(link, styleSheet, result) {
  var index, rule, rules, _i, _len;
  try {
    rules = styleSheet != null ? styleSheet.cssRules : void 0;
  } catch (e) {}

  if (rules && rules.length) {
    for (index = _i = 0, _len = rules.length; _i < _len; index = ++_i) {
      rule = rules[index];
      switch (rule.type) {
        case CSSRule.CHARSET_RULE:
          continue;
        case CSSRule.IMPORT_RULE:
          result.push({
            link: link,
            rule: rule,
            index: index,
            href: rule.href
          });
          this.collectImportedStylesheets(link, rule.styleSheet, result);
          break;
        default:
          break;
      }
    }
  }
};

Reloader.prototype.waitUntilCssLoads = function(clone, func) {
  var callbackExecuted, executeCallback, poll,
    this = this;
  callbackExecuted = false;
  executeCallback = function() {
    if (callbackExecuted) {
      return;
    }
    callbackExecuted = true;
    return func();
  };
  clone.onload = function() {
    console.log("onload!");
    _this.knownToSupportCssOnLoad = true;
    return executeCallback();
  };
  if (!this.knownToSupportCssOnLoad) {
    (poll = function() {
      if (clone.sheet) {

```

```

        console.log("polling!");
        return executeCallback();
    } else {
        return _this.Timer.start(50, poll);
    }
})(());
}
return this.Timer.start(this.options.stylesheetReloadTimeout, executeCallback);
};

Reloader.prototype.linkHref = function(link) {
    return link.href || link.getAttribute('data-href');
};

Reloader.prototype.reattachStylesheetLink = function(link) {
    var clone, parent,
        _this = this;
    if (link.__LiveReload_pendingRemoval) {
        return;
    }
    link.__LiveReload_pendingRemoval = true;
    if (link.tagName === 'STYLE') {
        clone = this.document.createElement('link');
        clone.rel = 'stylesheet';
        clone.media = link.media;
        clone.disabled = link.disabled;
    } else {
        clone = link.cloneNode(false);
    }
    clone.href = this.generateCacheBustUrl(this.linkHref(link));
    parent = link.parentNode;
    if (parent.lastChild === link) {
        parent.appendChild(clone);
    } else {
        parent.insertBefore(clone, link.nextSibling);
    }
    return this.waitUntilCssLoads(clone, function() {
        var additionalWaitingTime;
        if (/AppleWebKit/.test(navigator.userAgent)) {
            additionalWaitingTime = 5;
        } else {
            additionalWaitingTime = 200;
        }
        return _this.Timer.start(additionalWaitingTime, function() {
            var _ref;
            if (!link.parentNode) {
                return;
            }
            link.parentNode.removeChild(link);
            clone.onreadystatechange = null;
            return (_ref = _this.window.StyleFix) != null ? _ref.link(clone) : void 0;
        });
    });
};

Reloader.prototype.reattachImportedRule = function(_arg) {
    var href, index, link, media, newRule, parent, rule, tempLink,
        _this = this;
    rule = _arg.rule, index = _arg.index, link = _arg.link;
    parent = rule.parentStyleSheet;
    href = this.generateCacheBustUrl(rule.href);
    media = rule.media.length ? [].join.call(rule.media, ', ') : '';
    newRule = "@import url(\"" + href + "\") " + media + ";";
    rule.__LiveReload_newHref = href;
    tempLink = this.document.createElement("link");
    tempLink.rel = 'stylesheet';
    tempLink.href = href;
    tempLink.__LiveReload_pendingRemoval = true;
    if (link.parentNode) {
        link.parentNode.insertBefore(tempLink, link);
    }
    return this.Timer.start(this.importCacheWaitPeriod, function() {
        if (tempLink.parentNode) {
            tempLink.parentNode.removeChild(tempLink);
        }
        if (rule.__LiveReload_newHref !== href) {
            return;
        }
        parent.insertRule(newRule, index);
        parent.deleteRule(index + 1);
        rule = parent.cssRules[index];
        rule.__LiveReload_newHref = href;
        return _this.Timer.start(_this.importCacheWaitPeriod, function() {
            if (rule.__LiveReload_newHref !== href) {
                return;
            }
            parent.insertRule(newRule, index);
            return parent.deleteRule(index + 1);
        });
    });
};

Reloader.prototype.generateUniqueString = function() {
    return 'livereload=' + Date.now();
};

Reloader.prototype.generateCacheBustUrl = function(url, expando) {
    var hash, oldParams, params, _ref;
    if (expando == null) {
        expando = this.generateUniqueString();
    }
}
```

```

    }
    _ref = splitUrl(url), url = _ref.url, hash = _ref.hash, oldParams = _ref.params;
    if (this.options.overrideURL) {
      if (url.indexOf(this.options.serverURL) < 0) {
        url = this.options.serverURL + this.options.overrideURL + "?url=" + encodeURIComponent(url);
      }
    }
    params = oldParams.replace(/(\?|&)livereload=(\d+)/, function(match, sep) {
      return "" + sep + expando;
    });
    if (params === oldParams) {
      if (oldParams.length === 0) {
        params = "?" + expando;
      } else {
        params = "" + oldParams + "&" + expando;
      }
    }
    return url + params + hash;
  };
  return Reloader;
};

})();

}).call(this);

// livereload
var Connector, LiveReload, Options, Reloader, Timer;

Connector = __connector.Connector;
Timer = __timer.Timer;
Options = __options.Options;
Reloaders = __reloader.Reloaders;

__livereload.LiveReload = LiveReload = (function() {

  function LiveReload(window) {
    var _this = this;
    this.window = window;
    this.listeners = {};
    this.plugins = [];
    this.pluginIdentifiers = {};
    this.console = this.window.location.href.match(/LR-verbose/) && this.window.console &&
    this.window.console.log && this.window.console.error ? this.window.console : {
      log: function() {},
      error: function() {}
    };
    if (!(this.WebSocket = this.window.WebSocket || this.window.MozWebSocket)) {
      console.error("LiveReload disabled because the browser does not seem to support web sockets");
      return;
    }
    if (!(this.options = Options.extract(this.window.document))) {
      console.error("LiveReload disabled because it could not find its own <SCRIPT> tag");
      return;
    }
    this.reloader = new Reloader(this.window, this.console, Timer);
    this.connector = new Connector(this.options, this.WebSocket, Timer, {
      connecting: function() {},
      socketConnected: function() {},
      connected: function(protocol) {
        var _base;
        if (_base = _this.listeners).connect === "function") {
          _base.connect();
        }
        _this.log("LiveReload is connected to " + _this.options.host + ":" + _this.options.port + " (" + protocol + ")");
        return _this.analyze();
      },
      error: function(e) {
        if (e instanceof ProtocolError) {
          return console.log("") + e.message + ".";
        } else {
          return console.log("LiveReload internal error: " + e.message);
        }
      },
      disconnected: function(reason, nextDelay) {
        var _base;
        if (_base = _this.listeners).disconnect === "function") {
          _base.disconnect();
        }
        switch (reason) {
          case 'cannot-connect':
            return _this.log("LiveReload cannot connect to " + _this.options.host + ":" + _this.options.port +
", will retry in " + nextDelay + " sec.");
          case 'broken':
            return _this.log("LiveReload disconnected from " + _this.options.host + ":" + _this.options.port +
", reconnecting in " + nextDelay + " sec.");
          case 'handshake-timeout':
            return _this.log("LiveReload cannot connect to " + _this.options.host + ":" + _this.options.port +
(handshake timeout), will retry in " + nextDelay + " sec.");
          case 'handshake-failed':
            return _this.log("LiveReload cannot connect to " + _this.options.host + ":" + _this.options.port +
(handshake failed), will retry in " + nextDelay + " sec.");
          case 'manual':
            break;
          case 'error':
            break;
        }
      }
    });
  }
});

```

```

        default:
          return _this.log("LiveReload disconnected from " + _this.options.host + ":" + _this.options.port + "
(" + reason + "), reconnecting in " + nextDelay + " sec.");
      }
    },
    message: function(message) {
      switch (message.command) {
        case 'reload':
          return _this.performReload(message);
        case 'alert':
          return _this.performAlert(message);
      }
    }
  });
}

LiveReload.prototype.on = function(eventName, handler) {
  return this.listeners[eventName] = handler;
};

LiveReload.prototype.log = function(message) {
  return this.console.log("") + message);
};

LiveReload.prototype.performReload = function(message) {
  var _ref, _ref2;
  this.log("LiveReload received reload request for " + message.path + ".");
  return this.reloader.reload(message.path, {
    liveCSS: (_ref = message.liveCSS) != null ? _ref : true,
    liveImg: (_ref2 = message.liveImg) != null ? _ref2 : true,
    originalPath: message.originalPath || '',
    overrideURL: message.overrideURL || '',
    serverURL: "http://" + this.options.host + ":" + this.options.port
  });
};

LiveReload.prototype.performAlert = function(message) {
  return alert(message.message);
};

LiveReload.prototype.shutDown = function() {
  var _base;
  this.connector.disconnect();
  this.log("LiveReload disconnected.");
  return typeof (_base = this.listeners).shutdown === "function" ? _base.shutdown() : void 0;
};

LiveReload.prototype.hasPlugin = function(identifier) {
  return !this.pluginIdentifiers[identifier];
};

LiveReload.prototype.addPlugin = function(pluginClass) {
  var plugin;
  var _this = this;
  if (this.hasPlugin(pluginClass.identifier)) return;
  this.pluginIdentifiers[pluginClass.identifier] = true;
  plugin = new pluginClass(this.window, {
    livereload: this,
    reloader: this.reloader,
    connector: this.connector,
    console: this.console,
    Timer: Timer,
    generateCacheBustUrl: function(url) {
      return _this.reloader.generateCacheBustUrl(url);
    }
  });
  this.plugins.push(plugin);
  this.reloader.addPlugin(plugin);
};

LiveReload.prototype.analyze = function() {
  var plugin, pluginData, pluginsData, _i, _len, _ref;
  if (!(this.connector.protocol >= 7)) return;
  pluginsData = {};
  _ref = this.plugins;
  for (_i = 0, _len = _ref.length; _i < _len; _i++) {
    plugin = _ref[_i];
    pluginData[plugin.constructor.identifier] = pluginData = (typeof plugin.analyze === "function" ?
    plugin.analyze() : void 0) || {};
    pluginData.version = plugin.constructor.version;
  }
  this.connector.sendCommand({
    command: 'info',
    plugins: pluginsData,
    url: this.window.location.href
  });
};

return LiveReload;
})());
// less
var LessPlugin;
less = LessPlugin = (function() {
  LessPlugin.identifier = 'less';
  LessPlugin.version = '1.0';
  function LessPlugin(window, host) {
    this.window = window;
    this.host = host;
  }
});

```

```

}
LessPlugin.prototype.reload = function(path, options) {
  if (this.window.less && this.window.less.refresh) {
    if (path.match(/\.\less$/i)) {
      return this.reloadLess(path);
    }
    if (options.originalPath.match(/\.\less$/i)) {
      return this.reloadLess(options.originalPath);
    }
  }
  return false;
};
LessPlugin.prototype.reloadLess = function(path) {
  var link, links, _i, _len;
  links = (function() {
    var _i, _len, _ref, _results;
    _ref = document.getElementsByTagName('link');
    _results = [];
    for (_i = 0, _len = _ref.length; _i < _len; _i++) {
      link = _ref[_i];
      if (link.href && link.rel === 'stylesheet/less' || (link.rel.match(/stylesheet/) &&
link.type.match(/^text\/(x-)?less$/))) {
        _results.push(link);
      }
    }
    return _results;
  })();
  if (links.length === 0) {
    return false;
  }
  for (_i = 0, _len = links.length; _i < _len; _i++) {
    link = links[_i];
    link.href = this.host.generateCacheBustUrl(link.href);
  }
  this.host.console.log("LiveReload is asking LESS to recompile all stylesheets");
  this.window.less.refresh(true);
  return true;
};
LessPlugin.prototype.analyze = function() {
  return {
    disable: !(this.window.less && this.window.less.refresh)
  };
};
return LessPlugin;
})();
// startup
var CustomEvents, LiveReload, k;
CustomEvents = __customevents;
LiveReload = window.LiveReload = new (__livereload.LiveReload)(window);
for (k in window) {
  if (k.match(/^LiveReloadPlugin/)) {
    LiveReload.addPlugin(window[k]);
  }
}
LiveReload.addPlugin(__less);
LiveReload.on('shutdown', function() {
  return delete window.LiveReload;
});
LiveReload.on('connect', function() {
  return CustomEvents.fire(document, 'LiveReloadConnect');
});
LiveReload.on('disconnect', function() {
  return CustomEvents.fire(document, 'LiveReloadDisconnect');
});
CustomEvents.bind(document, 'LiveReloadShutDown', function() {
  return LiveReload.shutDown();
});
})();
})();

```

```
{  
  "name": "HashiCorp Terraform",  
  "icons": [  
    {  
      "src": "/assets/images/favicons/android-chrome-192x192.png",  
      "sizes": "192x192",  
      "type": "image/png"  
    },  
    {  
      "src": "/assets/images/favicons/android-chrome-512x512.png",  
      "sizes": "512x512",  
      "type": "image/png"  
    }  
  ],  
  "theme_color": "#ffffff",  
  "background_color": "#ffffff",  
  "display": "standalone"  
}
```

Y













Terraform Documentation

Welcome to the Terraform documentation! This documentation is more of a reference guide for all available features and options of Terraform. If you're just getting started with Terraform, please start with the introduction and getting started guide ([/intro/index.html](#)) instead.

Backends

A "backend" in Terraform determines how state is loaded and how an operation such as `apply` is executed. This abstraction enables non-local file state storage, remote execution, etc.

By default, Terraform uses the "local" backend, which is the normal behavior of Terraform you're used to. This is the backend that was being invoked throughout the introduction (</intro/index.html>).

Here are some of the benefits of backends:

- **Working in a team:** Backends can store their state remotely and protect that state with locks to prevent corruption. Some backends such as Terraform Enterprise even automatically store a history of all state revisions.
- **Keeping sensitive information off disk:** State is retrieved from backends on demand and only stored in memory. If you're using a backend such as Amazon S3, the only location the state ever is persisted is in S3.
- **Remote operations:** For larger infrastructures or certain changes, `terraform apply` can take a long, long time. Some backends support remote operations which enable the operation to execute remotely. You can then turn off your computer and your operation will still complete. Paired with remote state storage and locking above, this also helps in team environments.

Backends are completely optional. You can successfully use Terraform without ever having to learn or use backends.

However, they do solve pain points that afflict teams at a certain scale. If you're an individual, you can likely get away with never using backends.

Even if you only intend to use the "local" backend, it may be useful to learn about backends since you can also change the behavior of the local backend.

Backend Configuration

Backends are configured directly in Terraform files in the `terraform` section. After configuring a backend, it has to be initialized (/docs/backends/init.html).

Below, we show a complete example configuring the "consul" backend:

```
terraform {
  backend "consul" {
    address = "demo.consul.io"
    scheme  = "https"
    path    = "example_app/terraform_state"
  }
}
```

You specify the backend type as a key to the backend stanza. Within the stanza are backend-specific configuration keys. The list of supported backends and their configuration is in the sidebar to the left.

Only one backend may be specified and the configuration **may not contain interpolations**. Terraform will validate this.

First Time Configuration

When configuring a backend for the first time (moving from no defined backend to explicitly configuring one), Terraform will give you the option to migrate your state to the new backend. This lets you adopt backends without losing any existing state.

To be extra careful, we always recommend manually backing up your state as well. You can do this by simply copying your `terraform.tfstate` file to another location. The initialization process should create a backup as well, but it never hurts to be safe!

Configuring a backend for the first time is no different than changing a configuration in the future: create the new configuration and run `terraform init`. Terraform will guide you the rest of the way.

Partial Configuration

You do not need to specify every required argument in the backend configuration. Omitting certain arguments may be desirable to avoid storing secrets, such as access keys, within the main configuration. When some or all of the arguments are omitted, we call this a *partial configuration*.

With a partial configuration, the remaining configuration arguments must be provided as part of the initialization process (/docs/backends/init.html#backend-initialization). There are several ways to supply the remaining arguments:

- **Interactively:** Terraform will interactively ask you for the required values, unless interactive input is disabled. Terraform will not prompt for optional values.
- **File:** A configuration file may be specified via the `init` command line. To specify a file, use the `-backend-config=PATH` option when running `terraform init`. If the file contains secrets it may be kept in a secure data store, such as Vault (<https://www.vaultproject.io/>), in which case it must be downloaded to the local disk before running Terraform.

- **Command-line key/value pairs:** Key/value pairs can be specified via the `init` command line. Note that many shells retain command-line flags in a history file, so this isn't recommended for secrets. To specify a single key/value pair, use the `-backend-config="KEY=VALUE"` option when running `terraform init`.

If backend settings are provided in multiple locations, the top-level settings are merged such that any command-line options override the settings in the main configuration and then the command-line options are processed in order, with later options overriding values set by earlier options.

The final, merged configuration is stored on disk in the `.terraform` directory, which should be ignored from version control. This means that sensitive information can be omitted from version control, but it will be present in plain text on local disk when running Terraform.

When using partial configuration, Terraform requires at a minimum that an empty backend configuration is specified in one of the root Terraform configuration files, to specify the backend type. For example:

```
terraform {  
  backend "consul" {}  
}
```

A backend configuration file has the contents of the backend block as top-level attributes, without the need to wrap it in another `terraform` or `backend` block:

```
address = "demo.consul.io"  
path    = "example_app/terraform_state"  
scheme  = "https"
```

The same settings can alternatively be specified on the command line as follows:

```
$ terraform init \  
  -backend-config="address=demo.consul.io" \  
  -backend-config="path=example_app/terraform_state" \  
  -backend-config="scheme=https"
```

Changing Configuration

You can change your backend configuration at any time. You can change both the configuration itself as well as the type of backend (for example from "consul" to "s3").

Terraform will automatically detect any changes in your configuration and request a reinitialization (/docs/backends/init.html). As part of the reinitialization process, Terraform will ask if you'd like to migrate your existing state to the new configuration. This allows you to easily switch from one backend to another.

If you're using multiple workspaces (/docs/state/workspaces.html), Terraform can copy all workspaces to the destination. If Terraform detects you have multiple workspaces, it will ask if this is what you want to do.

If you're just reconfiguring the same backend, Terraform will still ask if you want to migrate your state. You can respond "no" in this scenario.

Unconfiguring a Backend

If you no longer want to use any backend, you can simply remove the configuration from the file. Terraform will detect this like any other change and prompt you to reinitialize ([/docs/backends/init.html](#)).

As part of the reinitialization, Terraform will ask if you'd like to migrate your state back down to normal local state. Once this is complete then Terraform is back to behaving as it does by default.

Backend Initialization

Terraform must initialize any configured backend before use. This can be done by simply running `terraform init`.

The `terraform init` command should be run by any member of your team on any Terraform configuration as a first step. It is safe to execute multiple times and performs all the setup actions required for a Terraform environment, including initializing the backend.

The `init` command must be called:

- On any new environment that configures a backend
- On any change of the backend configuration (including type of backend)
- On removing backend configuration completely

You don't need to remember these exact cases. Terraform will detect when initialization is required and error in that situation. Terraform doesn't auto-initialize because it may require additional information from the user, perform state migrations, etc.

The `init` command will do more than just initialize the backend. Please see the `init` documentation ([/docs/commands/init.html](#)) for more information.

Backend & Legacy Remote State

Prior to Terraform 0.9.0 backends didn't exist and remote state management was done in a completely different way. This page documents how you can migrate to the new backend system and any considerations along the way.

Migrating to the new backends system is extremely simple. The only complex case is if you had automation around configuring remote state. An existing environment can be configured to use the new backend system after just a few minutes of reading.

For the remainder of this document, the remote state system prior to Terraform 0.9.0 will be called "legacy remote state."

Note: This page is targeted at users who used remote state prior to version 0.9.0 and need to upgrade their environments. If you didn't use remote state, you can ignore this document.

Backwards Compatibility

In version 0.9.0, Terraform knows how to load and continue working with legacy remote state. A warning is shown guiding you to this page, but otherwise everything continues to work without changing any configuration.

Backwards compatibility with legacy remote state environments will be removed in Terraform 0.11.0, or two major releases after 0.10.0. Starting in 0.10.0, detection will remain but users will be *required* to update their configurations to use backends. In Terraform 0.11.0, detection and loading will be completely removed.

For the short term, you may continue using Terraform with version 0.9.0 as you have been. However, you should begin planning to update your configuration very soon. As you'll see, this process is very easy.

Migrating to Backends

You should begin by reading the complete backend documentation ([/docs/backends](#)) section. This section covers in detail how you use and configure backends.

Next, perform the following steps to migrate. These steps will also guide you through backing up your existing remote state just in case things don't go as planned.

1. **With the older Terraform version (version 0.8.x),** run `terraform remote pull`. This will cache the latest legacy remote state data locally. We'll use this for a backup in case things go wrong.
2. Backup your `.terraform/terraform.tfstate` file. This contains the cache we just pulled. Please copy this file to a location outside of your Terraform module.
3. Configure your backend ([/docs/backends/config.html](#)) in your Terraform configuration. The backend type is the same backend type as you used with your legacy remote state. The configuration should be setup to match the same configuration you used with remote state.
4. Run the `init` command ([/docs/backends/init.html](#)). This is an interactive process that will guide you through migrating your existing remote state to the new backend system. During this step, Terraform may ask if you want to copy your old remote state into the newly configured backend. If you configured the identical backend location, you may say no since it should already be there.

5. Verify your state looks good by running `terraform plan` and seeing if it detects your infrastructure. Advanced users may run `terraform state pull` which will output the raw contents of your state file to your console. You can compare this with the file you saved. There may be slight differences in the serial number and version data, but the raw data should be almost identical.

After the above steps, you're good to go! Everyone who uses the same Terraform state should copy the same steps above. The only difference is they may be able to skip the configuration step if you're sharing the configuration.

At this point, **older Terraform versions will stop working**. Terraform will prevent itself from working with state written with a higher version of Terraform. This means that even other users using an older version of Terraform with the same configured remote state location will no longer be able to work with the environment. Everyone must upgrade.

Rolling Back

If the migration fails for any reason: your states look different, your plan isn't what you expect, you're getting errors, etc. then you may roll back.

After rolling back, please report an issue (<https://github.com/hashicorp/terraform>) so that we may resolve anything that may have gone wrong for you.

To roll back, follow the steps below using Terraform 0.8.x or earlier:

1. Remove the backend configuration from your Terraform configuration file.
2. Remove any "terraform.tfstate" files (including backups). If you believe these may contain important data, you may back them up. Going with the assumption that you started this migration guide with working remote state, these files shouldn't contain anything of value.
3. Copy the `.terraform/terraform.tfstate` file you backed up back into the same location.

And you're rolled back. If your backend migration worked properly and was able to update your remote state, **then this will not work**. Terraform prevents writing state that was written with a higher Terraform version or a later serial number.

If you're absolutely certain you want to restore your state backup then you can use `terraform remote push -force`. This is extremely dangerous and you will lose any changes that were in the remote location.

Configuration Automation

The `terraform remote config` command has been replaced with `terraform init`. The new command is better in many ways by allowing file-based configuration, automatic state migration, and more.

You should be able to very easily migrate `terraform remote config` scripting to the new `terraform init` command.

The new `terraform init` command takes a `-backend-config` flag which is either an HCL file or a string in the format of `key=value`. This configuration is merged with the backend configuration in your Terraform files. This lets you keep secrets out of your actual configuration. We call this "partial configuration" and you can learn more in the docs on configuring backends ([/docs/backends/config.html](#)).

This does introduce an extra step: your automation must generate a JSON file (presumably JSON is easier to generate from a script than HCL and HCL is compatible) to pass into `-backend-config`.

Remote Operations (plan, apply, etc.)

Most backends run all operations on the local system — although Terraform stores its state remotely with these backends, it still executes its logic locally and makes API requests directly from the system where it was invoked.

This is simple to understand and work with, but when many people are collaborating on the same Terraform configurations, it requires everyone's execution environment to be similar. This includes sharing access to infrastructure provider credentials, keeping Terraform versions in sync, keeping Terraform variables in sync, and installing any extra software required by Terraform providers. This becomes more burdensome as teams get larger.

Some backends can run operations (plan, apply, etc.) on a remote machine, while appearing to execute locally. This enables a more consistent execution environment and more powerful access controls, without disrupting workflows for users who are already comfortable with running Terraform.

Currently, the [remote backend](#) (</docs/backends/types/remote.html>) is the only backend to support remote operations, and Terraform Enterprise (</docs/enterprise/index.html>) is the only remote execution environment that supports it. For more information, see:

- The [remote backend](#) (</docs/backends/types/remote.html>)
- Terraform Enterprise's CLI-driven run workflow (</docs/enterprise/run/cli.html>)

State Storage and Locking

Backends are responsible for storing state and providing an API for state locking ([/docs/state/locking.html](#)). State locking is optional.

Despite the state being stored remotely, all Terraform commands such as `terraform console`, the `terraform state` operations, `terraform taint`, and more will continue to work as if the state was local.

State Storage

Backends determine where state is stored. For example, the local (default) backend stores state in a local JSON file on disk. The Consul backend stores the state within Consul. Both of these backends happen to provide locking: local via system APIs and Consul via locking APIs.

When using a non-local backend, Terraform will not persist the state anywhere on disk except in the case of a non-recoverable error where writing the state to the backend failed. This behavior is a major benefit for backends: if sensitive values are in your state, using a remote backend allows you to use Terraform without that state ever being persisted to disk.

In the case of an error persisting the state to the backend, Terraform will write the state locally. This is to prevent data loss. If this happens the end user must manually push the state to the remote backend once the error is resolved.

Manual State Pull/Push

You can still manually retrieve the state from the remote state using the `terraform state pull` command. This will load your remote state and output it to stdout. You can choose to save that to a file or perform any other operations.

You can also manually write state with `terraform state push`. **This is extremely dangerous and should be avoided if possible.** This will overwrite the remote state. This can be used to do manual fixups if necessary.

When manually pushing state, Terraform will attempt to protect you from some potentially dangerous situations:

- **Differing lineage:** The "lineage" is a unique ID assigned to a state when it is created. If a lineage is different, then it means the states were created at different times and its very likely you're modifying a different state. Terraform will not allow this.
- **Higher serial:** Every state has a monotonically increasing "serial" number. If the destination state has a higher serial, Terraform will not allow you to write it since it means that changes have occurred since the state you're attempting to write.

Both of these protections can be bypassed with the `-force` flag if you're confident you're making the right decision. Even if using the `-force` flag, we recommend making a backup of the state with `terraform state pull` prior to forcing the overwrite.

State Locking

Backends are responsible for supporting state locking ([/docs/state/locking.html](#)) if possible. Not all backend types support state locking. In the list of supported backend types ([/docs/backends/types](#)) we explicitly note whether locking is supported.

For more information on state locking, view the page dedicated to state locking (</docs/state/locking.html>).

Backend Types

This section documents the various backend types supported by Terraform. If you're not familiar with backends, please read the sections about backends (</docs/backends/index.html>) first.

Backends may support differing levels of features in Terraform. We differentiate these by calling a backend either **standard** or **enhanced**. All backends must implement **standard** functionality. These are defined below:

- **Standard:** State management, functionality covered in State Storage & Locking (</docs/backends/state.html>)
- **Enhanced:** Everything in standard plus remote operations (</docs/backends/operations.html>).

The backends are separated in the left by standard and enhanced.

artifactory

Kind: Standard (with no locking)

Stores the state as an artifact in a given repository in Artifactory (<https://www.jfrog.com/artifactory/>).

Generic HTTP repositories are supported, and state from different configurations may be kept at different subpaths within the repository.

Note: The URL must include the path to the Artifactory installation. It will likely end in /artifactory.

Example Configuration

```
terraform {
  backend "artifactory" {
    username = "SheldonCooper"
    password = "AmyFarrahFowler"
    url      = "https://custom.artifactoryonline.com/artifactory"
    repo     = "foo"
    subpath  = "terarafm-bar"
  }
}
```

Example Referencing

```
data "terraform_remote_state" "foo" {
  backend = "artifactory"
  config {
    username = "SheldonCooper"
    password = "AmyFarrahFowler"
    url      = "https://custom.artifactoryonline.com/artifactory"
    repo     = "foo"
    subpath  = "terraform-bar"
  }
}
```

Configuration variables

The following configuration options / environment variables are supported:

- `username` / `ARTIFACTORY_USERNAME` (Required) - The username
- `password` / `ARTIFACTORY_PASSWORD` (Required) - The password
- `url` / `ARTIFACTORY_URL` (Required) - The URL. Note that this is the base url to artifactory not the full repo and subpath.
- `repo` (Required) - The repository name
- `subpath` (Required) - Path within the repository

azurerm (formerly azure)

Kind: Standard (with state locking)

Stores the state as a given key in a given blob container on Microsoft Azure Storage (<https://azure.microsoft.com/en-us/documentation/articles/storage-introduction/>). This backend also supports state locking and consistency checking via native capabilities of Microsoft Azure Storage.

Example Configuration

```
terraform {
  backend "azurerm" {
    storage_account_name = "abcd1234"
    container_name        = "tfstate"
    key                  = "prod.terraform.tfstate"
  }
}
```

Note that for the access credentials we recommend using a partial configuration (/docs/backends/config.html).

Example Referencing

```
data "terraform_remote_state" "foo" {
  backend = "azurerm"
  config {
    storage_account_name = "terraform123abc"
    container_name       = "terraform-state"
    key                 = "prod.terraform.tfstate"
  }
}
```

Configuration variables

The following configuration options are supported:

- `storage_account_name` - (Required) The name of the storage account
- `container_name` - (Required) The name of the container to use within the storage account
- `key` - (Required) The key where to place/look for state file inside the container
- `access_key / ARM_ACCESS_KEY` - (Optional) Storage account access key
- `environment / ARM_ENVIRONMENT` - (Optional) The cloud environment to use. Supported values are:
 - `public` (default)
 - `usgovernment`
 - `german`

- china

The following configuration options must be supplied if access_key is not.

- resource_group_name - The resource group which contains the storage account.
- arm_subscription_id / ARM_SUBSCRIPTION_ID - The Azure Subscription ID.
- arm_client_id / ARM_CLIENT_ID - The Azure Client ID.
- arm_client_secret / ARM_CLIENT_SECRET - The Azure Client Secret.
- arm_tenant_id / ARM_TENANT_ID - The Azure Tenant ID.

consul

Kind: Standard (with locking)

Stores the state in the Consul (<https://www.consul.io/>) KV store at a given path.

This backend supports state locking (</docs/state/locking.html>).

Example Configuration

```
terraform {
  backend "consul" {
    address = "demo.consul.io"
    scheme  = "https"
    path     = "full/path"
  }
}
```

Note that for the access credentials we recommend using a partial configuration (</docs/backends/config.html>).

Example Referencing

```
data "terraform_remote_state" "foo" {
  backend = "consul"
  config {
    path = "full/path"
  }
}
```

Configuration variables

The following configuration options / environment variables are supported:

- **path** - (Required) Path in the Consul KV store
- **access_token / CONSUL_HTTP_TOKEN** - (Required) Access token
- **address / CONSUL_HTTP_ADDR** - (Optional) DNS name and port of your Consul endpoint specified in the format `dnsname:port`. Defaults to the local agent HTTP listener.
- **scheme** - (Optional) Specifies what protocol to use when talking to the given address, either `http` or `https`. SSL support can also be triggered by setting the environment variable `CONSUL_HTTP_SSL` to `true`.
- **datacenter** - (Optional) The datacenter to use. Defaults to that of the agent.
- **http_auth / CONSUL_HTTP_AUTH** - (Optional) HTTP Basic Authentication credentials to be used when communicating with Consul, in the format of either `user` or `user:pass`.
- **gzip** - (Optional) `true` to compress the state data using `gzip`, or `false` (the default) to leave it uncompressed.

- `lock` - (Optional) `false` to disable locking. This defaults to `true`, but will require session permissions with Consul to perform locking.
- `ca_file / CONSUL_CAFILE` - (Optional) A path to a PEM-encoded certificate authority used to verify the remote agent's certificate.
- `cert_file / CONSUL_CLIENT_CERT` - (Optional) A path to a PEM-encoded certificate provided to the remote agent; requires use of `key_file`.
- `key_file / CONSUL_CLIENT_KEY` - (Optional) A path to a PEM-encoded private key, required if `cert_file` is specified.

etcd

Kind: Standard (with no locking)

Stores the state in etcd 2.x (<https://coreos.com/etcd/docs/latest/v2/README.html>) at a given path.

Example Configuration

```
terraform {
  backend "etcd" {
    path      = "path/to/terraform.tfstate"
    endpoints = "http://one:4001 http://two:4001"
  }
}
```

Example Referencing

```
data "terraform_remote_state" "foo" {
  backend = "etcd"
  config {
    path      = "path/to/terraform.tfstate"
    endpoints = "http://one:4001 http://two:4001"
  }
}
```

Configuration variables

The following configuration options are supported:

- **path** - (Required) The path where to store the state
- **endpoints** - (Required) A space-separated list of the etcd endpoints
- **username** - (Optional) The username
- **password** - (Optional) The password

etcdv3

Kind: Standard (with locking)

Stores the state in the etcd (<https://coreos.com/etcd/>) KV store with a given prefix.

This backend supports state locking (</docs/state/locking.html>).

Example Configuration

```
terraform {
  backend "etcdv3" {
    endpoints = ["etcd-1:2379", "etcd-2:2379", "etcd-3:2379"]
    lock      = true
    prefix    = "terraform-state/"
  }
}
```

Note that for the access credentials we recommend using a partial configuration (</docs/backends/config.html>).

Example Referencing

```
data "terraform_remote_state" "foo" {
  backend = "etcdv3"
  config {
    endpoints = ["etcd-1:2379", "etcd-2:2379", "etcd-3:2379"]
    lock      = true
    prefix    = "terraform-state/"
  }
}
```

Configuration variables

The following configuration options / environment variables are supported:

- **endpoints** - (Required) The list of 'etcd' endpoints which to connect to.
- **username** / **ETCDV3_USERNAME** - (Optional) Username used to connect to the etcd cluster.
- **password** / **ETCDV3_PASSWORD** - (Optional) Password used to connect to the etcd cluster.
- **prefix** - (Optional) An optional prefix to be added to keys when storing state in etcd. Defaults to "".
- **lock** - (Optional) Whether to lock state access. Defaults to `true`.
- **cacert_path** - (Optional) The path to a PEM-encoded CA bundle with which to verify certificates of TLS-enabled etcd servers.
- **cert_path** - (Optional) The path to a PEM-encoded certificate to provide to etcd for secure client identification.

- `key_path` - (Optional) The path to a PEM-encoded key to provide to etcd for secure client identification.

gcs

Kind: Standard (with locking)

Stores the state as an object in a configurable prefix and bucket on Google Cloud Storage (<https://cloud.google.com/storage/>) (GCS).

Example Configuration

```
terraform {
  backend "gcs" {
    bucket  = "tf-state-prod"
    prefix  = "terraform/state"
  }
}
```

Example Referencing

```
data "terraform_remote_state" "foo" {
  backend = "gcs"
  config {
    bucket  = "terraform-state"
    prefix  = "prod"
  }
}

resource "template_file" "bar" {
  template = "${greeting}"

  vars {
    greeting = "${data.terraform_remote_state.foo.greeting}"
  }
}
```

Configuration variables

The following configuration options are supported:

- **bucket** - (Required) The name of the GCS bucket. This name must be globally unique. For more information, see [Bucket Naming Guidelines](https://cloud.google.com/storage/docs/bucketnaming.html#requirements) (<https://cloud.google.com/storage/docs/bucketnaming.html#requirements>).
- **credentials / GOOGLE_CREDENTIALS** - (Optional) Local path to Google Cloud Platform account credentials in JSON format. If unset, Google Application Default Credentials (<https://developers.google.com/identity/protocols/application-default-credentials>) are used. The provided credentials need to have the `devstorage.read_write` scope and `WRITER` permissions on the bucket.
- **prefix** - (Optional) GCS prefix inside the bucket. Named states for workspaces are stored in an object called `<prefix>/<name>.tfstate`.

- `path` - (Deprecated) GCS path to the state file of the default state. For backwards compatibility only, use `prefix` instead.
- `project / GOOGLE_PROJECT` - (Optional) The project ID to which the bucket belongs. This is only used when creating a new bucket during initialization. Since buckets have globally unique names, the project ID is not required to access the bucket during normal operation.
- `region / GOOGLE_REGION` - (Optional) The region in which a new bucket is created. For more information, see Bucket Locations (<https://cloud.google.com/storage/docs/bucket-locations>).
- `encryption_key / GOOGLE_ENCRYPTION_KEY` - (Optional) A 32 byte base64 encoded 'customer supplied encryption key' used to encrypt all state. For more information see Customer Supplied Encryption Keys (<https://cloud.google.com/storage/docs/encryption#customer-supplied>).

http

Kind: Standard (with optional locking)

Stores the state using a simple REST (https://en.wikipedia.org/wiki/Representational_state_transfer) client.

State will be fetched via GET, updated via POST, and purged with DELETE. The method used for updating is configurable.

When locking support is enabled it will use LOCK and UNLOCK requests providing the lock info in the body. The endpoint should return a 423: Locked or 409: Conflict with the holding lock info when it's already taken, 200: OK for success. Any other status will be considered an error. The ID of the holding lock info will be added as a query parameter to state updates requests.

Example Usage

```
terraform {
  backend "http" {
    address = "http://myrest.api.com/foo"
    lock_address = "http://myrest.api.com/foo"
    unlock_address = "http://myrest.api.com/foo"
  }
}
```

Example Referencing

```
data "terraform_remote_state" "foo" {
  backend = "http"
  config {
    address = "http://my.rest.api.com"
  }
}
```

Configuration variables

The following configuration options are supported:

- `address` - (Required) The address of the REST endpoint
- `update_method` - (Optional) HTTP method to use when updating state. Defaults to POST.
- `lock_address` - (Optional) The address of the lock REST endpoint. Defaults to disabled.
- `lock_method` - (Optional) The HTTP method to use when locking. Defaults to LOCK.
- `unlock_address` - (Optional) The address of the unlock REST endpoint. Defaults to disabled.
- `unlock_method` - (Optional) The HTTP method to use when unlocking. Defaults to UNLOCK.
- `username` - (Optional) The username for HTTP basic authentication

- `password` - (Optional) The password for HTTP basic authentication
- `skip_cert_verification` - (Optional) Whether to skip TLS verification. Defaults to `false`.

local

Kind: Enhanced

The local backend stores state on the local filesystem, locks that state using system APIs, and performs operations locally.

Example Configuration

```
terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

Example Reference

```
data "terraform_remote_state" "foo" {
  backend = "local"

  config {
    path = "${path.module}../../terraform.tfstate"
  }
}
```

Configuration variables

The following configuration options are supported:

- **path** - (Optional) The path to the `tfstate` file. This defaults to `"terraform.tfstate"` relative to the root module by default.

manta

Kind: Standard (with locking within Manta)

Stores the state as an artifact in Manta (<https://www.joyent.com/manta>).

Example Configuration

```
terraform {
  backend "manta" {
    path      = "random/path"
    object_name = "terraform.tfstate"
  }
}
```

Note that for the access credentials we recommend using a partial configuration (/docs/backends/config.html).

Example Referencing

```
data "terraform_remote_state" "foo" {
  backend = "manta"
  config {
    path      = "random/path"
    object_name = "terraform.tfstate"
  }
}
```

Configuration variables

The following configuration options are supported:

- account - (Required) This is the name of the Manta account. It can also be provided via the `SDC_ACCOUNT` or `TRITON_ACCOUNT` environment variables.
- user - (Optional) The username of the Triton account used to authenticate with the Triton API. It can also be provided via the `SDC_USER` or `TRITON_USER` environment variables.
- url - (Optional) The Manta API Endpoint. It can also be provided via the `MANTA_URL` environment variable. Defaults to `https://us-east.manta.joyent.com`.
- key_material - (Optional) This is the private key of an SSH key associated with the Triton account to be used. If this is not set, the private key corresponding to the fingerprint in `key_id` must be available via an SSH Agent. Can be set via the `SDC_KEY_MATERIAL` or `TRITON_KEY_MATERIAL` environment variables.
- key_id - (Required) This is the fingerprint of the public key matching the key specified in `key_path`. It can be obtained via the command `ssh-keygen -l -E md5 -f /path/to/key`. Can be set via the `SDC_KEY_ID` or `TRITON_KEY_ID` environment variables.

- `insecure_skip_tls_verify` - (Optional) This allows skipping TLS verification of the Triton endpoint. It is useful when connecting to a temporary Triton installation such as Cloud-On-A-Laptop which does not generally use a certificate signed by a trusted root CA. Defaults to `false`.
- `path` - (Required) The path relative to your private storage directory (`/$MANTA_USER/stor`) where the state file will be stored. **Please Note:** If this path does not exist, then the backend will create this folder location as part of backend creation.
- `objectName` - (Optional, Deprecated) Use `object_name` instead.
- `object_name` - (Optional) The name of the state file (defaults to `terraform.tfstate`)

remote

Kind: Enhanced

The remote backend stores state and runs operations remotely. When running `terraform plan` or `terraform apply` with this backend, the actual execution occurs in Terraform Enterprise, with log output streaming to the local terminal.

To use this backend you need a Terraform Enterprise account on app.terraform.io (<https://app.terraform.io>) or have a private instance of Terraform Enterprise (version v201809-1 or newer).

Preview Release: As of Terraform 0.11.8, the remote backend is a preview release and we do not recommend using it with production workloads. Please continue to use the existing Terraform Enterprise (</docs/backends/types/terraform-enterprise.html>) backend for production workspaces.

Command Support

Currently the remote backend supports the following Terraform commands:

- `apply`
- `fmt`
- `get`
- `init`
- `output`
- `plan`
- `providers`
- `show`
- `taint`
- `untaint`
- `validate`
- `version`
- `workspace`

Workspaces

The remote backend can work with either a single remote workspace, or with multiple similarly-named remote workspaces (like `networking-dev` and `networking-prod`). The `workspaces` block of the backend configuration determines which mode it uses:

- To use a single workspace, set `workspaces.name` to the remote workspace's full name (like `networking`).

- To use multiple workspaces, set `workspaces.prefix` to a prefix used in all of the desired remote workspace names.

For example, set `prefix = "networking-"` to use a group of workspaces with names like `networking-dev` and `networking-prod`.

When interacting with workspaces on the command line, Terraform uses shortened names without the common prefix. For example, if `prefix = "networking-"`, use `terraform workspace select prod` to switch to the `networking-prod` workspace.

The backend configuration requires either name or prefix. Omitting both or setting both results in a configuration error.

If previous state is present when you run `terraform init` and the corresponding remote workspaces are empty or absent, Terraform will create workspaces and/or update the remote state accordingly.

Example Configuration

```
# Using a single workspace:
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "company"

    workspaces {
      name = "my-app-prod"
    }
  }
}

# Using multiple workspaces:
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "company"

    workspaces {
      prefix = "my-app-"
    }
  }
}
```

Example Reference

```
data "terraform_remote_state" "foo" {
  backend = "remote"

  config {
    organization = "company"

    workspaces {
      name = "workspace"
    }
  }
}
```

Configuration variables

The following configuration options are supported:

- `hostname` - (Optional) The remote backend hostname to connect to. Defaults to `app.terraform.io`.
- `organization` - (Required) The name of the organization containing the targeted workspace(s).
- `token` - (Optional) The token used to authenticate with the remote backend. We recommend omitting the token from the configuration, and instead setting it as `credentials` in the CLI config file ([/docs/commands/cli-config.html#credentials](#)).
- `workspaces` - (Required) A block specifying which remote workspace(s) to use. The `workspaces` block supports the following keys:
 - `name` - (Optional) The full name of one remote workspace. When configured, only the default workspace can be used. This option conflicts with `prefix`.
 - `prefix` - (Optional) A prefix used in the names of one or more remote workspaces, all of which can be used with this configuration. The full workspace names are used in Terraform Enterprise, and the short names (minus the prefix) are used on the command line. If omitted, only the default workspace can be used. This option conflicts with `name`.

S3

Kind: Standard (with locking via DynamoDB)

Stores the state as a given key in a given bucket on Amazon S3 (<https://aws.amazon.com/s3/>). This backend also supports state locking and consistency checking via Dynamo DB (<https://aws.amazon.com/dynamodb/>), which can be enabled by setting the `dynamodb_table` field to an existing DynamoDB table name.

Warning! It is highly recommended that you enable Bucket Versioning (<http://docs.aws.amazon.com/AmazonS3/latest/UG/enable-bucket-versioning.html>) on the S3 bucket to allow for state recovery in the case of accidental deletions and human error.

Example Configuration

```
terraform {
  backend "s3" {
    bucket = "mybucket"
    key    = "path/to/my/key"
    region = "us-east-1"
  }
}
```

This assumes we have a bucket created called `mybucket`. The Terraform state is written to the key `path/to/my/key`.

Note that for the access credentials we recommend using a partial configuration ([/docs/backends/config.html](#)).

S3 Bucket Permissions

Terraform will need the following AWS IAM permissions on the target backend bucket:

- `s3>ListBucket` on `arn:aws:s3:::mybucket`
- `s3GetObject` on `arn:aws:s3:::mybucket/path/to/my/key`
- `s3PutObject` on `arn:aws:s3:::mybucket/path/to/my/key`

This is seen in the following AWS IAM Statement:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "s3>ListBucket",
      "Resource": "arn:aws:s3:::mybucket"
    },
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject", "s3:PutObject"],
      "Resource": "arn:aws:s3:::mybucket/path/to/my/key"
    }
  ]
}
```

DynamoDB Table Permissions

If you are using state locking, Terraform will need the following AWS IAM permissions on the DynamoDB table (`arn:aws:dynamodb:::table/mytable`):

- `dynamodb:GetItem`
- `dynamodb:PutItem`
- `dynamodb>DeleteItem`

This is seen in the following AWS IAM Statement:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb>DeleteItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/mytable"
    }
  ]
}
```

Using the S3 remote state

To make use of the S3 remote state we can use the `terraform_remote_state` data source ([/docs/providers/terraform/d/remote_state.html](#)).

```
data "terraform_remote_state" "network" {
  backend = "s3"
  config {
    bucket = "terraform-state-prod"
    key    = "network/terraform.tfstate"
    region = "us-east-1"
  }
}
```

The `terraform_remote_state` data source will return all of the root module outputs defined in the referenced remote state (but not any outputs from nested modules unless they are explicitly output again in the root). An example output might look like:

```
data.terraform_remote_state.network:
  id = 2016-10-29 01:57:59.780010914 +0000 UTC
  addresses.# = 2
  addresses.0 = 52.207.220.222
  addresses.1 = 54.196.78.166
  backend = s3
  config.% = 3
  config.bucket = terraform-state-prod
  config.key = network/terraform.tfstate
  config.region = us-east-1
  elb_address = web-elb-790251200.us-east-1.elb.amazonaws.com
  public_subnet_id = subnet-1e05dd33
```

Configuration variables

The following configuration options or environment variables are supported:

- `bucket` - (Required) The name of the S3 bucket.
- `key` - (Required) The path to the state file inside the bucket. When using a non-default workspace ([/docs/state/workspaces.html](#)), the state path will be `/workspace_key_prefix/workspace_name/key`
- `region / AWS_DEFAULT_REGION` - (Optional) The region of the S3 bucket.
- `endpoint / AWS_S3_ENDPOINT` - (Optional) A custom endpoint for the S3 API.
- `encrypt` - (Optional) Whether to enable server side encryption (<https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingServerSideEncryption.html>) of the state file.
- `acl` - Canned ACL (<https://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html#canned-acl>) to be applied to the state file.
- `access_key / AWS_ACCESS_KEY_ID` - (Optional) AWS access key.
- `secret_key / AWS_SECRET_ACCESS_KEY` - (Optional) AWS secret access key.
- `kms_key_id` - (Optional) The ARN of a KMS Key to use for encrypting the state.
- `lock_table` - (Optional, Deprecated) Use `dynamodb_table` instead.
- `dynamodb_table` - (Optional) The name of a DynamoDB table to use for state locking and consistency. The table must have a primary key named LockID. If not present, locking will be disabled.

- `profile` - (Optional) This is the AWS profile name as set in the shared credentials file.
- `shared_credentials_file` - (Optional) This is the path to the shared credentials file. If this is not set and a profile is specified, `~/aws/credentials` will be used.
- `token` - (Optional) Use this to set an MFA token. It can also be sourced from the `AWS_SESSION_TOKEN` environment variable.
- `role_arn` - (Optional) The role to be assumed.
- `assume_role_policy` - (Optional) The permissions applied when assuming a role.
- `external_id` - (Optional) The external ID to use when assuming the role.
- `session_name` - (Optional) The session name to use when assuming the role.
- `workspace_key_prefix` - (Optional) The prefix applied to the state path inside the bucket. This is only relevant when using a non-default workspace. This defaults to "env:"
- `skip_credentials_validation` - (Optional) Skip the credentials validation via the STS API.
- `skip_get_ec2_platforms` - (Optional) Skip getting the supported EC2 platforms.
- `skip_region_validation` - (Optional) Skip validation of provided region name.
- `skip_requesting_account_id` - (Optional) Skip requesting the account ID.
- `skip_metadata_api_check` - (Optional) Skip the AWS Metadata API check.

Multi-account AWS Architecture

A common architectural pattern is for an organization to use a number of separate AWS accounts to isolate different teams and environments. For example, a "staging" system will often be deployed into a separate AWS account than its corresponding "production" system, to minimize the risk of the staging environment affecting production infrastructure, whether via rate limiting, misconfigured access controls, or other unintended interactions.

The S3 backend can be used in a number of different ways that make different tradeoffs between convenience, security, and isolation in such an organization. This section describes one such approach that aims to find a good compromise between these tradeoffs, allowing use of Terraform's workspaces feature ([/docs/state/workspaces.html](#)) to switch conveniently between multiple isolated deployments of the same configuration.

Use this section as a starting-point for your approach, but note that you will probably need to make adjustments for the unique standards and regulations that apply to your organization. You will also need to make some adjustments to this approach to account for *existing* practices within your organization, if for example other tools have previously been used to manage infrastructure.

Terraform is an administrative tool that manages your infrastructure, and so ideally the infrastructure that is used by Terraform should exist outside of the infrastructure that Terraform manages. This can be achieved by creating a separate *administrative* AWS account which contains the user accounts used by human operators and any infrastructure and tools used to manage the other accounts. Isolating shared administrative tools from your main environments has a number of advantages, such as avoiding accidentally damaging the administrative infrastructure while changing the target infrastructure, and reducing the risk that an attacker might abuse production infrastructure to gain access to the (usually more privileged) administrative infrastructure.

Administrative Account Setup

Your administrative AWS account will contain at least the following items:

- One or more IAM user (http://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html) for system administrators that will log in to maintain infrastructure in the other accounts.
- Optionally, one or more IAM groups (http://docs.aws.amazon.com/IAM/latest/UserGuide/id_groups.html) to differentiate between different groups of users that have different levels of access to the other AWS accounts.
- An S3 bucket (<http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingBucket.html>) that will contain the Terraform state files for each workspace.
- A DynamoDB table
(<http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html#HowItWorks.CoreComponents.TablesItemsAttributes>) that will be used for locking to prevent concurrent operations on a single workspace.

Provide the S3 bucket name and DynamoDB table name to Terraform within the S3 backend configuration using the `bucket` and `dynamodb_table` arguments respectively, and configure a suitable `workspace_key_prefix` to contain the states of the various workspaces that will subsequently be created for this configuration.

Environment Account Setup

For the sake of this section, the term "environment account" refers to one of the accounts whose contents are managed by Terraform, separate from the administrative account described above.

Your environment accounts will eventually contain your own product-specific infrastructure. Along with this it must contain one or more IAM roles (http://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html) that grant sufficient access for Terraform to perform the desired management tasks.

Delegating Access

Each Administrator will run Terraform using credentials for their IAM user in the administrative account. IAM Role Delegation (http://docs.aws.amazon.com/IAM/latest/UserGuide/tutorial_cross-account-with-roles.html) is used to grant these users access to the roles created in each environment account.

Full details on role delegation are covered in the AWS documentation linked above. The most important details are:

- Each role's *Assume Role Policy* must grant access to the administrative AWS account, which creates a trust relationship with the administrative AWS account so that its users may assume the role.
- The users or groups within the administrative account must also have a policy that creates the converse relationship, allowing these users or groups to assume that role.

Since the purpose of the administrative account is only to host tools for managing other accounts, it is useful to give the administrative accounts restricted access only to the specific operations needed to assume the environment account role and access the Terraform state. By blocking all other access, you remove the risk that user error will lead to staging or production resources being created in the administrative account by mistake.

When configuring Terraform, use either environment variables or the standard credentials file `~/.aws/credentials` to provide the administrator user's IAM credentials within the administrative account to both the S3 backend *and* to Terraform's AWS provider.

Use conditional configuration to pass a different `assume_role` value to the AWS provider depending on the selected workspace. For example:

```
variable "workspace_iam_roles" {
  default = {
    staging   = "arn:aws:iam::STAGING-ACCOUNT-ID:role/Terraform"
    production = "arn:aws:iam::PRODUCTION-ACCOUNT-ID:role/Terraform"
  }
}

provider "aws" {
  # No credentials explicitly set here because they come from either the
  # environment or the global credentials file.

  assume_role = "${var.workspace_iam_roles[terraform.workspace]}"
}
```

If workspace IAM roles are centrally managed and shared across many separate Terraform configurations, the role ARNs could also be obtained via a data source such as `terraform_remote_state` ([/docs/providers/terraform/d/remote_state.html](#)) to avoid repeating these values.

Creating and Selecting Workspaces

With the necessary objects created and the backend configured, run `terraform init` to initialize the backend and establish an initial workspace called "default". This workspace will not be used, but is created automatically by Terraform as a convenience for users who are not using the workspaces feature.

Create a workspace corresponding to each key given in the `workspace_iam_roles` variable value above:

```
$ terraform workspace new staging
Created and switched to workspace "staging"!

...
$ terraform workspace new production
Created and switched to workspace "production"!

...
```

Due to the `assume_role` setting in the AWS provider configuration, any management operations for AWS resources will be performed via the configured role in the appropriate environment AWS account. The backend operations, such as reading and writing the state from S3, will be performed directly as the administrator's own user within the administrative account.

```
$ terraform workspace select staging
$ terraform apply
...
```

Running Terraform in Amazon EC2

Teams that make extensive use of Terraform for infrastructure management often run Terraform in automation ([/guides/running-terraform-in-automation.html](#)) to ensure a consistent operating environment and to limit access to the various secrets and other sensitive information that Terraform configurations tend to require.

When running Terraform in an automation tool running on an Amazon EC2 instance, consider running this instance in the administrative account and using an instance profile (http://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2_instance-profiles.html) in place of the various administrator IAM users suggested above. An IAM instance profile can also be granted cross-account delegation access via an IAM policy, giving this instance the access it needs to run Terraform.

To isolate access to different environment accounts, use a separate EC2 instance for each target account so that its access can be limited only to the single account.

Similar approaches can be taken with equivalent features in other AWS compute services, such as ECS.

Protecting Access to Workspace State

In a simple implementation of the pattern described in the prior sections, all users have access to read and write states for all workspaces. In many cases it is desirable to apply more precise access constraints to the Terraform state objects in S3, so that for example only trusted administrators are allowed to modify the production state, or to control *reading* of a state that contains sensitive information.

Amazon S3 supports fine-grained access control on a per-object-path basis using IAM policy. A full description of S3's access control mechanism is beyond the scope of this guide, but an example IAM policy granting access to only a single state object within an S3 bucket is shown below:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "s3>ListBucket",
      "Resource": "arn:aws:s3:::myorg-terraform-states"
    },
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject", "s3:PutObject"],
      "Resource": "arn:aws:s3:::myorg-terraform-states/myapp/production/tfstate"
    }
  ]
}
```

It is not possible to apply such fine-grained access control to the DynamoDB table used for locking, so it is possible for any user with Terraform access to lock any workspace state, even if they do not have access to read or write that state. If a malicious user has such access they could block attempts to use Terraform against some or all of your workspaces as long as locking is enabled in the backend configuration.

swift

Kind: Standard (with no locking)

Stores the state as an artifact in Swift (<http://docs.openstack.org/developer/swift/>).

Warning! It is highly recommended that you enable Object Versioning (https://docs.openstack.org/developer/swift/overview_object_versioning.html) by setting the `expire_after` (https://www.terraform.io/docs/backends/types/swift.html#archive_path) configuration. This allows for state recovery in the case of accidental deletions and human error.

Example Configuration

```
terraform {  
  backend "swift" {  
    path = "terraform-state"  
  }  
}
```

This will create a container called `terraform-state` and an object within that container called `tfstate.tf`.

Note: Currently, the object name is statically defined as 'tfstate.tf'. Therefore Swift pseudo-folders (<https://docs.openstack.org/user-guide/cli-swift-pseudo-hierarchical-folders-directories.html>) are not currently supported.

For the access credentials we recommend using a partial configuration (/docs/backends/config.html).

Example Referencing

```
data "terraform_remote_state" "foo" {  
  backend = "swift"  
  config {  
    path = "terraform_state"  
  }  
}
```

Configuration variables

The following configuration options are supported:

- `auth_url` - (Required) The Identity authentication URL. If omitted, the `OS_AUTH_URL` environment variable is used.
- `container` - (Required) The name of the container to create for storing the Terraform state file.

- `path` - (Optional) DEPRECATED: Use `container` instead. The name of the container to create in order to store the state file.
- `user_name` - (Optional) The Username to login with. If omitted, the `OS_USERNAME` environment variable is used.
- `user_id` - (Optional) The User ID to login with. If omitted, the `OS_USER_ID` environment variable is used.
- `password` - (Optional) The Password to login with. If omitted, the `OS_PASSWORD` environment variable is used.
- `token` - (Optional) Access token to login with instead of user and password. If omitted, the `OS_AUTH_TOKEN` variable is used.
- `region_name` (Required) - The region in which to store `terraform.tfstate`. If omitted, the `OS_REGION_NAME` environment variable is used.
- `tenant_id` (Optional) The ID of the Tenant (Identity v2) or Project (Identity v3) to login with. If omitted, the `OS_TENANT_ID` or `OS_PROJECT_ID` environment variables are used.
- `tenant_name` - (Optional) The Name of the Tenant (Identity v2) or Project (Identity v3) to login with. If omitted, the `OS_TENANT_NAME` or `OS_PROJECT_NAME` environment variable are used.
- `domain_id` - (Optional) The ID of the Domain to scope to (Identity v3). If omitted, the following environment variables are checked (in this order): `OS_USER_DOMAIN_ID`, `OS_PROJECT_DOMAIN_ID`, `OS_DOMAIN_ID`.
- `domain_name` - (Optional) The Name of the Domain to scope to (Identity v3). If omitted, the following environment variables are checked (in this order): `OS_USER_DOMAIN_NAME`, `OS_PROJECT_DOMAIN_NAME`, `OS_DOMAIN_NAME`, `DEFAULT_DOMAIN`.
- `insecure` - (Optional) Trust self-signed SSL certificates. If omitted, the `OS_INSECURE` environment variable is used.
- `cacert_file` - (Optional) Specify a custom CA certificate when communicating over SSL. If omitted, the `OS_CACERT` environment variable is used.
- `cert` - (Optional) Specify client certificate file for SSL client authentication. If omitted the `OS_CERT` environment variable is used.
- `key` - (Optional) Specify client private key file for SSL client authentication. If omitted the `OS_KEY` environment variable is used.
- `archive_container` - (Optional) The container to create to store archived copies of the Terraform state file. If specified, Swift object versioning (https://docs.openstack.org/developer/swift/overview_object_versioning.html) is enabled on the container created at `container`.
- `archive_path` - (Optional) DEPRECATED: Use `archive_container` instead. The path to store archived copied of `terraform.tfstate`. If specified, Swift object versioning (https://docs.openstack.org/developer/swift/overview_object_versioning.html) is enabled on the container created at `path`.
- `expire_after` - (Optional) How long should the `terraform.tfstate` created at `path` be retained for? Supported durations: `m` - Minutes, `h` - Hours, `d` - Days.

terraform enterprise

Kind: Standard (with no locking)

Reads and writes state from a Terraform Enterprise (/docs/enterprise/index.html) workspace.

Why is this called "atlas"? Before it was a standalone offering, Terraform Enterprise was part of an integrated suite of enterprise products called Atlas. This backend predates the current version Terraform Enterprise, so it uses the old name.

This backend is useful for uncommon tasks like migrating state into a Terraform Enterprise workspace, but we no longer recommend using it as part of your day-to-day Terraform workflow. Since it performs runs outside of Terraform Enterprise and updates state directly, it does not support Terraform Enterprise's collaborative features like workspace locking (/docs/enterprise/run/index.html). To perform Terraform Enterprise runs from the command line, use Terraform Enterprise's CLI-driven workflow (/docs/enterprise/run/cli.html) instead.

Example Configuration

```
terraform {  
  backend "atlas" {  
    name = "example_corp/networking-prod"  
    address = "https://app.terraform.io" # optional  
  }  
}
```

We recommend using a partial configuration (/docs/backends/config.html) and omitting the access token, which can be provided as an environment variable.

Example Referencing

```
data "terraform_remote_state" "foo" {  
  backend = "atlas"  
  config {  
    name = "example_corp/networking-prod"  
  }  
}
```

Configuration variables

The following configuration options / environment variables are supported:

- **name** - (Required) Full name of the workspace (<ORGANIZATION>/<WORKSPACE>).
- **ATLAS_TOKEN/ access_token** - (Required) A Terraform Enterprise user API token (/docs/enterprise/users-teams-organizations/users.html#api-tokens). We recommend using the ATLAS_TOKEN environment variable rather than setting access_token in the configuration.

- **address** - (Optional) The URL of a Terraform Enterprise instance. Defaults to the SaaS version of Terraform Enterprise, at "<https://app.terraform.io>"; if you use a private install, provide its URL here.

Administering Private Terraform Enterprise

Private Terraform Enterprise (PTFE) is software provided to customers that allows full use of Terraform Enterprise in a private, isolated environment.

Administration of a PTFE instance has two main domains:

- Installation, upgrades, and operational tasks like backups and monitoring, which take place outside the Terraform Enterprise application.
- Administrative tasks and configuration within the application itself.

This section is about in-application administration, including general settings, systemwide integration settings, and management of accounts and resources. Administration functions can be managed via user interface (the focus of this guide) or via the Admin API (/docs/enterprise/api/admin/index.html).

Accessing the Admin Interface

Only Private Terraform Enterprise users with the site-admin permission can access the administrative functions.

The initial user account for a PTFE instance is the first site admin. Site admins can grant admin permissions to other users in the "Users" section of the admin pages. See Promoting a User to Administrator (/docs/enterprise/private/admin/resources.html#promoting-a-user-to-administrator) for details.

To navigate to the site admin section of the UI, click your user icon in the upper right, then click **Site Admin**:

The screenshot shows the Private Terraform Enterprise web interface. At the top, there's a navigation bar with icons for user profile, organization dropdown ('example_corp'), and tabs for 'Workspaces', 'Modules', and 'Settings'. To the right of the tabs are links for 'Documentation' and 'Status' and a small user icon. Below the navigation is a 'Workspaces' section header with a '1 total' count. It lists one workspace: 'minimum-prod' with status 'NO CHANGES', last change '25 days ago', run 'run-e1Pp', and repo 'nfagerlund/terraform'. To the right of this list is a user dropdown menu. The menu items are: 'USER' (Signed in as 'nfagerlund'), 'User Settings', 'Site Admin' (which is highlighted with a red oval), and 'Log Out'. A 'Search by name' input field is also visible above the workspace list.

This will take you to the admin area. Currently, it defaults to showing the user management page; use the navigation on the left to access the other administrative functions.



Choose an organization ▾

Documentation | Status



Admin > Users

SITE ADMINISTRATION

[Users](#)[Organizations](#)[Workspaces](#)[Runs](#)[Migrations](#)[Terraform Versions](#)[Settings](#)[SAML](#)[SMTP](#)[Twilio](#)

Release: d6e4bd6

Users

You can search for all the users on this installation. Users can be suspended, granted site administration access, and impersonated to help with supporting the organization and workspaces.

5 total

[All \(5\)](#)[Suspended \(0\)](#)[Site Admins \(1\)](#)

Search by name or email



USER

STATUS



assistant



bb-webhooks-example_corp

bb-webhooks-example_corp@hashicorp.com



gh-webhooks-example_corp

gh-webhooks-example_corp@hashicorp.com



gh-webhooks-examplecorp_legacy

gh-webhooks-examplecorp_legacy@hashicorp.com



nfagerlund

ADMIN

Administration Tasks

- General settings (/docs/enterprise/private/admin/general.html)
- Service Integrations (/docs/enterprise/private/admin/integration.html)
- Managing Accounts and Resources (/docs/enterprise/private/admin/resources.html)

Administration: General Settings

General settings control global behavior in Private Terraform Enterprise. To access general settings, visit the site admin area and click **Settings** in the left menu. To save the settings, click **Save Settings** at the bottom of the page.

The screenshot shows the 'Settings' page under 'Site Administration'. The left sidebar lists 'Users', 'Organizations', 'Workspaces', 'Runs', 'Migrations', 'Terraform Versions', and 'Settings' (which is selected). Below the sidebar, a release identifier 'Release: d2a7cb2' is shown. The main content area has a heading 'Settings' and a section titled 'Contact info'. It includes a note about the support email address and a 'SUPPORT EMAIL ADDRESS' input field containing 'support@hashicorp.com'. A note below it states that this address is shown on bug/error pages. The next section is 'Member organization creation', with a note that only site admins can create organizations by default. It contains two radio button options: 'Only site admins can create organizations' (selected) and 'All users can create organizations'. The final section is 'API Rate Limiting', which notes that requests per second are limited based on the user's IP. It includes a checked checkbox for 'Enable API rate limiting' and a dropdown for 'NUMBER OF ALLOWED REQUESTS PER SECOND' set to '30'. A 'Save Settings' button is at the bottom.

API: See the Admin Settings API (</docs/enterprise/api/admin/settings.html>).

Contact Info

The support email address is used in system emails and other contact details. It defaults to support@hashicorp.com (mailto:support@hashicorp.com). If you'd like users of your instance to reach out to a specific person or team when they have issues, it can be changed to a local email address.

Organization Creation

Organization creation can be limited to site administrators or allowed for all users. Limiting organization creation to administrators means that the need for new organizations can be audited and their creation easily monitored.

When new user accounts are created, if they cannot create their own organizations, they will be unable to access any Terraform Enterprise resources until they are added to a team.

API Rate Limiting

By default, requests to the Terraform Enterprise API from a single user or IP address are limited to 30 requests per second (</docs/enterprise/api/index.html#rate-limiting>) to prevent abuse or hogging of resources. Since usage patterns may vary for a given instance, this can be updated to match local needs.

Administration: Service Integrations

Private Terraform Enterprise can integrate with several external services to send communications and authenticate users. Each of these integrations has a separate page in the site admin section.

At this time, the available site-wide integrations are:

- SAML Single Sign-On
- SMTP
- Twilio

API: See the Admin Settings API (</docs/enterprise/api/admin/settings.html>).

SAML Integration

The SAML integration settings allow configuration of a SAML Single Sign-On integration for Terraform Enterprise. To access the SAML settings, click **SAML** in the left menu.

Note: Since enabling SAML is an involved process, there is a separate SAML section of the documentation (</docs/enterprise/saml/index.html>). Consult those pages for detailed requirements and configuration instructions for both Terraform Enterprise and your IdP.

To enable SAML, click **Enable SAML single sign-on** under "SAML Settings". Configure the fields below, then click **Save SAML settings**. To update the settings, update the field values, and save.

The **Enable SAML debugging** option can be used if sign-on is failing. It provides additional debugging information during login tests. It should not be left on during normal operations.

SMTP Integration

SMTP integration allows Terraform Enterprise to send email-based notifications, such as new user invitations, password resets, and system errors. We strongly recommend configuring SMTP.

To access the SMTP settings, click **SMTP** in the left menu. To enable SMTP, check the **Enable email sending with SMTP** box on the settings page, configure the settings, and click "Save SMTP settings."



SITE ADMINISTRATION

Users

Organizations

Workspaces

Runs

Migrations

Terraform Versions

Settings

SAML

SMTP

Twilio

Release: d2a7cb2

SMTP Settings

Configuring SMTP allows Terraform Enterprise to send email-based notifications, such as new user invitations and system errors.

Enable email sending with SMTP

SENDER EMAIL

Support <support@example.com>

The email address emails will appear to come from.

SEND TEST EMAIL TO

yourname@example.com

Used to validate configuration, not stored permanently.

HOST

mail.example.com

PORT

587



AUTHENTICATION

none

USERNAME

PASSWORD

Leave empty to keep existing password.

Save SMTP Settings

- **Sender Email:** The address that system mails should come from. A plain email address; do not include a display name.
- **Send test email to:** A sample address to send a test email to. Used to validate the settings when configuring SMTP; not stored.
- **Host and Port:** The host and port details for the SMTP server that will be used.
- **Authentication:** The type of authentication used by the server. Options are none, login, and plain.
- **Username:** Username used to authenticate to the server. Not required if the authentication setting is none.
- **Password:** Password to authenticate to the server. Not required if the authentication setting is none.

Note: The SMTP server used with Terraform Enterprise must support connection via SSL with a valid certificate.

Twilio Integration

Twilio integration is used to send SMS messages for two-factor authentication. It is optional; application-based 2FA is also supported.

To access the Twilio settings, click **Twilio** in the left menu. To enable Twilio, check the **Enable SMS sending with Twilio** box on the settings page and configure the relevant settings:



Choose an organization ▾

Documentation | Status



Admin > Twilio Settings

SITE ADMINISTRATION

Users

Organizations

Workspaces

Runs

Migrations

Terraform Versions

Settings

SAML

SMTP

Twilio

Release: d2a7cb2

Twilio Settings

Configuring Twilio allows Terraform Enterprise to send SMS messages for two-factor authentication. Find your Twilio Account SID, auth token, and phone number in the [Twilio Console](#).

Enable SMS sending with Twilio

ACCOUNT SID

AC0123...

AUTH TOKEN

abcdef0123...

Leave empty to keep existing auth token.

FROM NUMBER

+15551234567

This number must be registered with Twilio.

Save Twilio Settings

Verify your Twilio account settings by sending a test SMS to a specified number:

+15551112345

Send Test SMS

- **Account SID:** The unique identifier for your Twilio application (<https://www.twilio.com/docs/usage/api/applications>).
- **Auth Token:** The token that allows authentication (<https://support.twilio.com/hc/en-us/articles/223136027-Auth-Tokens-and-How-to-Change-Them>) with your Account SID.
- **From Number:** The number the message should come from. Must be registered with Twilio.

You can also verify the Twilio settings by sending a test message. Enter a number in the bottom box and click **Send Test SMS**.

Administration: Managing Accounts and Resources

Site administrators have access to all organizations, users, runs, and workspaces. This visibility is intended to provide access to management actions such as adding administrators, updating Terraform versions or adding custom Terraform bundles, suspending or deleting users, and creating or deleting organizations. It also allows for "impersonation" to aid in assisting regular users with issues in Terraform Enterprise.

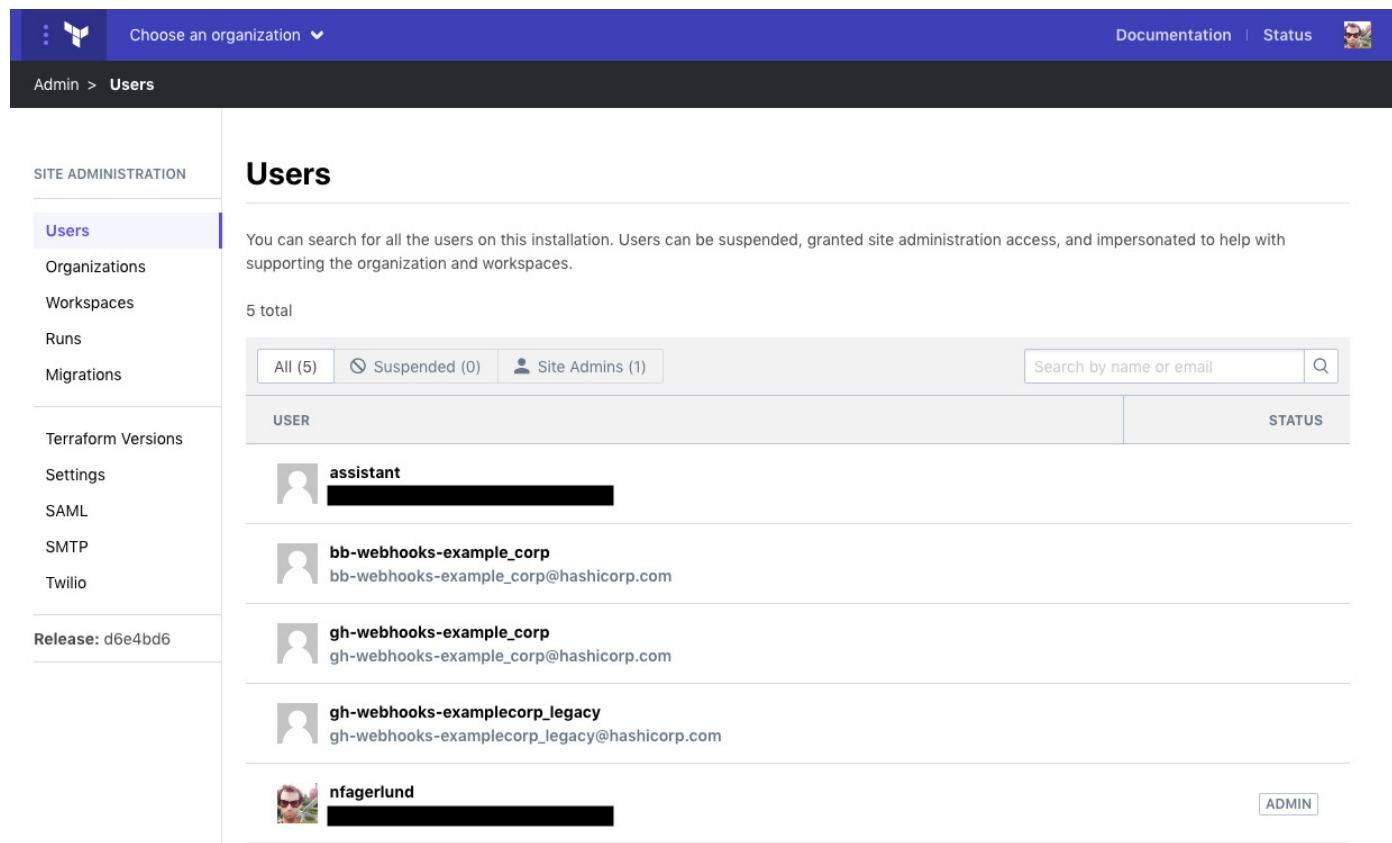
Viewing, Searching, and Filtering Lists

Each type of account or resource is initially presented as a searchable list, accessed by clicking the name of the resource on the left. In addition to searching or filtering (typically by email, name, or other relevant attribute), there are pre-existing filters to show useful sets, such as site administrators (users) or "Needs Attention" (workspaces, runs).

Managing Users

API: See the Admin Users API (</docs/enterprise/api/admin/users.html>).

To access the list of all users in the Terraform Enterprise instance, click **Users** in the left menu.



The screenshot shows the Terraform Enterprise administration interface. The top navigation bar includes a logo, a dropdown for 'Choose an organization', and links for 'Documentation' and 'Status'. The left sidebar has a 'SITE ADMINISTRATION' section with links for 'Users' (which is selected and highlighted in purple), 'Organizations', 'Workspaces', 'Runs', and 'Migrations'. Below that is a 'Terraform Versions' section with links for 'Settings', 'SAML', 'SMTP', and 'Twilio'. At the bottom of the sidebar, it says 'Release: d6e4bd6'. The main content area is titled 'Users'. It contains a search bar with placeholder 'Search by name or email' and a magnifying glass icon. Below the search bar are two buttons: 'All (5)' and 'Suspended (0)'. A third button, 'Site Admins (1)', has a small person icon next to it. The main list area has columns for 'USER' (listing names like 'assistant', 'bb-webhooks-example_corp', 'gh-webhooks-example_corp', 'gh-webhooks-examplecorp_legacy', and 'nfagerlund') and 'STATUS' (with one entry labeled 'ADMIN'). Each user row has a small profile picture on the left.

Selecting a user from the list shows their detail page, which includes their status and any organizations they belong to. The detail page offers four actions: promoting to administrator, suspending, deleting, and impersonating. For users with active two-factor authentication (2FA) (</docs/enterprise/users-teams-organizations/2fa.html>), it also offers an administrative option to disable their 2FA in the event that a reset is needed.

The screenshot shows the Terraform Enterprise Admin interface. On the left, there's a sidebar titled 'SITE ADMINISTRATION' with links like 'Users' (which is selected), 'Organizations', 'Workspaces', 'Runs', 'Migrations', 'Terraform Versions', 'Settings', 'SAML', 'SMTP', and 'Twilio'. Below the sidebar, it says 'Release: d2a7cb2'. The main content area shows a user profile for 'assistant' (ni.ck@hashicorp.com) with a 'Promote to admin' and 'Suspend user' button. Below the profile, there's a section for 'Impersonation' with a 'Impersonate' button. Another section shows 'Organizations (1)' with 'example.corp'. At the bottom, there's a 'Delete User' section with a 'Delete User' button.

Promoting a User to Administrator

This adds the user to the list of site administrators, which grants them access to this administrative area. Because admins have a very wide purview, if SMTP is configured, it will also send an email to the other site administrators notifying them that a user was added.

To promote a user, click **Promote to admin** at the top right of the user detail page.

Suspending or Deleting a User

Suspending a user retains their account, but does not allow them to access any Terraform Enterprise resources. Deleting a user removes their account completely; they would have to create a new account in order to log in again.

Suspended users can be unsuspended at any time. Deleted users cannot be recovered.

To suspend a user, click **Suspend user** at the upper right. To delete them, click **Delete User** in the "Delete User" section.

Impersonating a User

User impersonation allows Terraform Enterprise admins to access organization and workspace data and view runs. As an administrator, direct access to these resources only supports urgent interventions like deletion or force-canceling; to view and interact with resources, impersonation is required.

When impersonating a user, a reason is required and will be logged to the audit log. Any actions taken while impersonating will record both the impersonating admin and the impersonated user as the actor.

Impersonation can be performed from multiple places:

- From a user details admin page, by clicking the **Impersonate** button on the far right.
- From an organization, workspace, or run details admin page, all of which include a drop-down list of organization owners to impersonate.

- When a site admin encounters a 404 error for a resource that they do not have standard user access to.



Not Found

Sorry, the page requested could not be loaded.

This error could mean one of two things:

- The resource doesn't exist
- The resource exists, but you do not have permission to access it

If someone has linked you to this resource, ensure that they have given you proper permissions to access it.

Admin view

As an administrator, you can [view the admin organization page for demo_corp](#). You can also impersonate an owner of this organization to try loading this resource.

ORGANIZATION OWNERS

assistant



[Impersonate](#)

To view or modify the organization profile, settings and workspaces you must impersonate a user who is a member of the organization owner's team.

Resetting Two-Factor Authentication

If a user has lost access to their Terraform Enterprise 2FA device, a site admin can disable the configured 2FA and allow the user to log in using only their username and password or perform a standard password reset. If the user has active 2FA, a button labeled **Disable 2FA** will appear to the left of the admin promotion button.

Be sure that the user's identity and the validity of their request have been verified according to appropriate security procedures before disabling their configured 2FA.

Note: If the user belongs to an organization that requires 2FA, upon login, they will be redirected to set it up again ([/docs/enterprise/users-teams-organizations/2fa.html](#)) before they can view any other part of TFE.

Managing Organizations

API: See the Admin Organizations API ([/docs/enterprise/api/admin/organizations.html](#)).

If your institution uses multiple organizations in Terraform Enterprise, you can view the details of each organization by clicking it in the admin list of organizations. From the details page, you can impersonate an owner or delete an organization (using the red **Delete this organization** button at the bottom of the details page).

Typically, in private installations, all organizations will be granted "Premium" plan status for the purpose of providing access to all available features. However, it's also possible to set other statuses. An organization whose trial period is expired will be unable to make use of features in Terraform Enterprise.



Choose an organization ▾

Documentation | Status 

Admin > Organizations > **demo_corp**

SITE ADMINISTRATION

demo_corp

Users

Organizations

Workspaces

Runs

Migrations

Terraform Versions

Settings

SAML

SMTP

Twilio

Release: d2a7cb2

Manage the basic organization settings. For advanced settings impersonate an organization owner.

PLAN

premium

TRIAL EXPIRATION

09 / 02 / 2018

Sets the expiration of the organization. Default is 30 days from the time it is created. Expiration can only be set on organizations that are in "Trial" plan.

Update Organization

ORGANIZATION OWNERS

Owner to impersonate

Impersonate

To view or modify the organization profile, settings and workspaces you must impersonate a user who is a member of the organization owner's team.

Organization Delete

Deleting the **demo_corp** organization will permanently delete all workspaces associated with it. Please be certain you know what you're doing. This action cannot be undone.

Delete this organization

Managing Workspaces and Runs

API: See the Admin Workspaces API ([/docs/enterprise/api/admin/workspaces.html](#)) and Admin Runs API ([/docs/enterprise/api/admin/runs.html](#)).

The administrative view of workspaces and runs provides limited detail (name, status, and IDs) to avoid exposing sensitive data when it isn't needed. Site administrators can view and investigate workspaces and runs more deeply by impersonating a user with full access to the desired resource. (See Impersonating a User above.)

Deleting Workspaces

A workspace can be administratively deleted, using the **Delete this Workspace** button at the bottom of its details page, if it should not have been created, or is presenting issues for the application.

Force-Canceling Runs

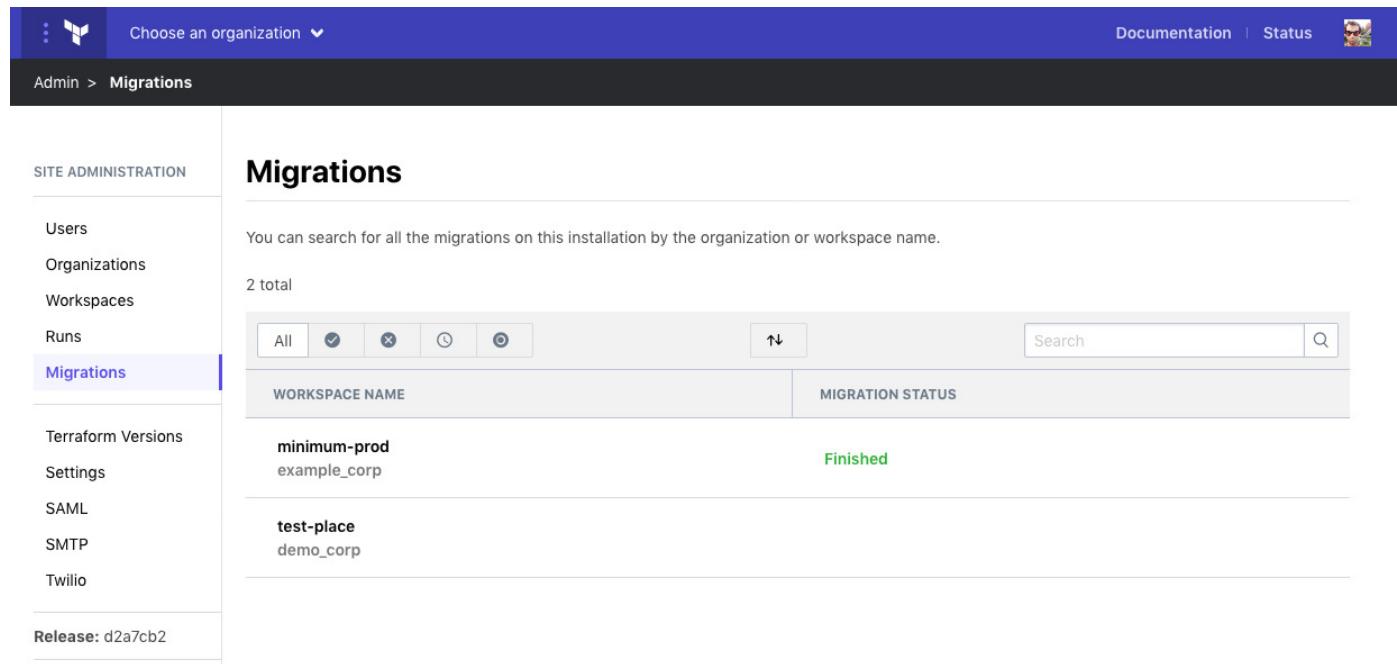
A run can be administratively force-canceled if it becomes stuck or is presenting issues to the application. Runs can be force-canceled from the run list or the run details page. The run details page also offers the option to impersonate an organization owner for additional details on the run.

We recommend impersonating a user (if necessary) to view run details prior to force-canceling a run, to ensure that graceful cancellation was attempted, and that the run is no longer progressing.

Migration of Workspace History from Legacy Environments

The Migrations section, accessed by clicking **Migrations** on the left, shows the progress of migrating history from legacy Terraform Enterprise environments to Terraform Enterprise workspaces. While all workspaces are listed in this area, history migration is only relevant for those workspaces created by setting up migration from older environments (</docs/enterprise/upgrade/index.html>).

During the initial setup, current data is migrated, and history data is migrated in the background, with the progress being visible in this admin area. This process is intended to have minimal impact on the running system, so if a large number of workspaces have been migrated, it may take some time before they're all complete.



The screenshot shows the Terraform Enterprise Admin interface. The top navigation bar includes a logo, a dropdown for 'Choose an organization', 'Documentation', 'Status', and a user profile icon. The left sidebar has a 'SITE ADMINISTRATION' heading with links for 'Users', 'Organizations', 'Workspaces', 'Runs', 'Migrations' (which is selected and highlighted in blue), 'Terraform Versions', 'Settings', 'SAML', 'SMTP', and 'Twilio'. Below the sidebar, a message says 'Release: d2a7cb2'. The main content area is titled 'Migrations' and contains a search bar and a table. The table has columns for 'WORKSPACE NAME' and 'MIGRATION STATUS'. It lists two workspaces: 'minimum-prod' (example_corp) with status 'Finished' and 'test-place' (demo_corp) which is still pending migration.

WORKSPACE NAME	MIGRATION STATUS
minimum-prod example_corp	Finished
test-place demo_corp	Pending

Note: Workspaces that weren't created by migrating a legacy environment have an empty "migration status" field on this page.

If history migration initially fails, the migration will be retried periodically. If you see a migration that continually errors, contact HashiCorp Support (</docs/enterprise/private/faq.html#support-for-private-terraform-enterprise>).

A future version of Terraform Enterprise will remove Legacy environment data and code completely. Be sure to review release notes and wait to upgrade to that version until migration of desired data is complete.

Managing Terraform Versions

API: See the Admin Terraform Versions API (</docs/enterprise/api/admin/terraform-versions.html>).

Private Terraform Enterprise ships with a default list of Terraform versions. However, the addition of new versions after installation is the responsibility of site administrators.


Choose an organization ▾
Documentation | Status

Admin > Terraform Versions

SITE ADMINISTRATION

Users
Organizations
Workspaces
Runs
Migrations

Terraform Versions

Settings

SAML
SMTP
Twilio

Release: d2a7cb2

Terraform Versions

Manage the available Terraform versions available in Terraform Enterprise when performing a run. New releases of Terraform are [available for download](#).

[Add Terraform Version](#)

VERSION	STATUS	USAGE
✓ 0.11.1	Enabled	in 1 workspace
✓ 0.11.0	Enabled	in 0 workspaces
⚠ 0.11.0-rc1	Beta	in 0 workspaces
⚠ 0.11.0-beta1	Beta	in 0 workspaces
✓ 0.10.8	Enabled	in 0 workspaces
✓ 0.10.7	Enabled	in 0 workspaces
✓ 0.10.6	Enabled	in 0 workspaces
✓ 0.10.5	Enabled	in 0 workspaces
✓ 0.10.4	Enabled	in 0 workspaces
✓ 0.10.3	Enabled	in 0 workspaces
✓ 0.10.2	Enabled	in 0 workspaces
✓ 0.10.1	Enabled	in 0 workspaces
✓ 0.10.0	Enabled	in 0 workspaces

To add a new version of Terraform, click **Terraform Versions** in the left menu and **Add Terraform Version** in the upper right, then provide the version number, Linux 64-bit download URL, and SHA256 checksum of the binary. Set the status to Beta to make the version available to site administrators, or Enabled to add it for everyone.


Choose an organization ▾
Documentation | Status

Admin > Terraform Versions > [Add Terraform Version](#)

SITE ADMINISTRATION

Users
Organizations
Workspaces
Runs
Migrations

Terraform Versions

Settings

SAML
SMTP
Twilio

Release: d2a7cb2

Add Terraform Version

New releases of Terraform are [available for download](#). The [Linux 64-bit](#) version download URL and SHA256 checksum must be provided.

VERSION

version

The numeric version of the terraform release (e.g. `0.11.7`). This should match the version number available in `terraform version` .

STATUS

Disabled

Enabled makes the Terraform version available to all users. Beta makes it available only to site admins. If any workspaces are using the Terraform version it can no longer be disabled.

URL

url

SHA256 CHECKSUM

sha

The SHA256 Checksum of the downloaded terraform version will be verified. The checksum can be found on the download page.

[Add Terraform Version](#)

The versions you add may be recent standard Terraform releases from HashiCorp, or custom Terraform versions. One common use for custom versions is to add a Terraform bundle that includes pre-installed providers ([/docs/enterprise/run/index.html#custom-and-community-providers](#)) commonly needed by the instance.

Versions of Terraform can also be modified by clicking them in the list. They can be set to disabled (unavailable for use) if no workspaces are currently using them. The list indicates how many workspaces are currently using a given version.

Private Terraform Enterprise Automated Recovery (Installer)

This guide explains how to configure automated recovery for a Private Terraform Enterprise installation. The goal is to provide a short Mean-Time-To-Recovery (MTTR) in the event of an outage. There are two steps in the automated recovery process:

1. Configure snapshots on your current install to backup data
2. Provision a new PTFE instance using the latest snapshot

This guide will walk through both of these steps.

1. Configure snapshots

Snapshots are taken on the Private TFE instance. That instance can have two types of data on it:

- Terraform Enterprise application data: The core product data such as run history, configuration history, state history. This data changes frequently.
- Terraform Enterprise installer data: The data used to configure Private TFE itself such as installation type, database connection settings, hostname. This data rarely changes.

In demo mode, both application data and installer data are stored on the PTFE instance. In mounted disk and external services mode, only installer data is stored on the instance. Application data is stored in the mounted disk or in an external Postgres.

Automated snapshots are more effective when using mounted disk or external services as the amount of backed up data is smaller and less risky.

Snapshots are configured in the dashboard under `Console Settings`, in the `Snapshot & Restore` section. We suggest you also select `Enable Automatic Scheduled Snapshots`. For the interval, it depends on the mode of operation you're using. If you're in Demo mode, one hour is recommended as that will minimize the data loss to one hour only. For Mounted Disk or External Services, Daily is recommended as the snapshots contain only configuration data, not application data.

2. Restore a snapshot in a new PTFE instance

Version Checking

Note: `replicatedctl` is located at `/usr/local/bin/replicatedctl`. On some operating systems, `/usr/local/bin` is not in the user's `$PATH`. In these cases, either add `/usr/local/bin` to the path or refer to `replicatedctl` with the full path.

Using the restore mechanism requires Replicated version 2.17.0 or greater. You can check the version using `replicatedctl version`.

Script

Below are examples of restore scripts that are run on machine boot. There are many mechanisms that can run a script on boot (cloud-init, systemd, /etc/init.d), which one is used is up to the user

This script is presented as an example. Anyone using it needs to understand what it's doing and is free to modify it to meet any additional needs they have.

S3

This example uses S3 as the mechanism to store snapshots on a debian-based system.

```

#!/bin/bash

set -e -u -o pipefail

bucket=your_bucket_to_store_snapshots
region=region_of_the_bucket
access_key=aws_access_key_id
secret_key=aws_secret_access_key

access="--store s3 --s3-bucket $bucket --s3-region $region --s3-key-id $access_key --s3-secret-key $secret_key"

# jq is used by this script, so install it. For other Linux distros, either preinstall jq # and remove these lines, or change to the mechanism your distro uses to install jq.

apt-get update
apt-get install -y jq

# Run the installer.

curl https://install.terraform.io/ptfe/stable | bash -s fast-timeouts

# This retrieves a list of all the snapshots currently available.
replicatedctl snapshot ls $access -o json > /tmp/snapshots.json

# Pull just the snapshot id out of the list of snapshots
id=$(jq -r 'sort_by(.finished) | .[-1].id // ""' /tmp/snapshots.json)

# If there are no snapshots available, exit out
if test "$id" = ""; then
    echo "No snapshots found"
    exit 1
fi

echo "Restoring snapshot: $id"

# Restore the detected snapshot. This ignores preflight checks to be sure the application # is booted.
replicatedctl snapshot restore $access --dismiss-preflight-checks "$id"

# Wait until the application reports itself as running. This step can be removed if # something upstream is prepared to wait for the application to finish booting.
until curl -f -s --connect-timeout 1 http://localhost/_health_check; do
    sleep 1
done

echo
echo "Application booted!"

```

SFTP

This example uses sftp to store the snapshots.

```

#!/bin/bash

set -e -u -o pipefail

key_file=path_to_your_ssh_key
key=$(base64 -w0 "$key_file")
host=sftp_server_hostname_or_ip
user=user_to_sftp_on_the_remote_server

access="--store sftp --sftp-host $host --sftp-user $user --sftp-key $key"

# jq is used by this script, so install it. For other Linux distros, either preinstall jq
# and remove these lines, or change to the mechanism your distro uses to install jq.

apt-get update
apt-get install -y jq

# Run the installer.

curl https://install.terraform.io/ptfe/stable | bash -s fast-timeouts

# This retrieves a list of all the snapshots currently available.
replicatedctl snapshot ls $access -o json > /tmp/snapshots.json

# Pull just the snapshot id out of the list of snapshots
id=$(jq -r 'sort_by(.finished) | .[-1].id // ""' /tmp/snapshots.json)

# If there are no snapshots available, exit out
if test "$id" = ""; then
    echo "No snapshots found"
    exit 1
fi

echo "Restoring snapshot: $id"

# Restore the detected snapshot. This ignores preflight checks to be sure the application
# is booted.
replicatedctl snapshot restore $access --dismiss-preflight-checks "$id"

# Wait until the application reports itself as running. This step can be removed if
# something upstream is prepared to wait for the application to finish booting.
until curl -f -s --connect-timeout 1 http://localhost/_health_check; do
    sleep 1
done

echo
echo "Application booted!"

```

Local directory

This example uses a local directory to store the snapshots. If this is intended to be run on a brand new system, the local directory would be either mounted block device or a network filesystem like NFS or CIFS.

```
#!/bin/bash

set -e -u -o pipefail

path=absolute_path_to_directory_of_snapshots

access="--store local --path '$path'

# jq is used by this script, so install it. For other Linux distros, either preinstall jq
# and remove these lines, or change to the mechanism your distro uses to install jq.

apt-get update
apt-get install -y jq

# Run the installer.

curl https://install.terraform.io/ptfe/stable | bash -s fast-timeouts

# This retrieves a list of all the snapshots currently available.
replicatedctl snapshot ls $access -o json > /tmp/snapshots.json

# Pull just the snapshot id out of the list of snapshots
id=$(jq -r 'sort_by(.finished) | .[-1].id // ""' /tmp/snapshots.json)

# If there are no snapshots available, exit out
if test "$id" = ""; then
    echo "No snapshots found"
    exit 1
fi

echo "Restoring snapshot: $id"

# Restore the detected snapshot. This ignores preflight checks to be sure the application
# is booted.
replicatedctl snapshot restore $access --dismiss-preflight-checks "$id"

# Wait until the application reports itself as running. This step can be removed if
# something upstream is prepared to wait for the application to finish booting.
until curl -f -s --connect-timeout 1 http://localhost/_health_check; do
    sleep 1
done

echo
echo "Application booted!"
```

Private Terraform Enterprise Automated Installation (Installer)

The installation of Private Terraform Enterprise can be automated for both online and airgapped installs. There are two parts to automating the install: configuring Replicated (<https://help.replicated.com/>) -- the platform which runs Terraform Enterprise -- and configuring Terraform Enterprise itself.

Before starting the install process, you must:

- prepare an application settings file, which defines the settings for the Terraform Enterprise application.
- prepare `/etc/replicated.conf`, which defines the settings for the Replicated installer.
- copy your license file to the instance.
- download the `.airgap` bundle to the instance (Airgapped mode only).

You may also need to provide additional flags (such as the instance's public and private IP addresses) in order to avoid being prompted for those values when running the installer (which may result in either a failure of the installer or an unbounded delay while waiting for input).

It's expected that the user is already familiar with how to do a manual install ([/docs/enterprise/private/install-installer.html#installation](#)).

Application settings

This file contains the values you would normally provide in the settings screen, which may be as simple as choosing the demo installation type or as complex as specifying the PostgreSQL connection string and S3 bucket credentials and parameters. You need to create this file first since it is referenced in the `ImportSettingsFrom` property in `/etc/replicated.conf`, which will be described below.

Format

The settings file is JSON formatted. All values must be strings. The example below is suitable for a demo installation:

```
{  
  "hostname": {  
    "value": "terraform.example.com"  
  },  
  "installation_type": {  
    "value": "poc"  
  },  
  "capacity_concurrency": {  
    "value": "5"  
  }  
}
```

Note: The JSON file must be valid JSON for the install to work, so it's best to validate it before using for an install.

The easiest way to check the application config is valid JSON would be with `python`, which will be present on most Linux installs:

```
$ python -m json.tool settings.json
Expecting property name enclosed in double quotes: line 8 column 5 (char 171)
```

After fixing the JSON file, the command will return the valid JSON:

```
$ python -m json.tool settings.json
{
  "hostname": {
    "value": "terraform.example.com"
  },
  "installation_type": {
    "value": "poc"
  },
  "capacity_concurrency": {
    "value": "5"
  }
}
```

Discovery

One the easiest ways to get the settings is to perform a manual install ([/docs/enterprise/private/install-installer.html#installation](#)) and configure all the settings how you want them. Then you can ssh in and request the settings in JSON format and use that file in a future automated install.

Note: `replicatedctl` is located at `/usr/local/bin/replicatedctl`. On some operating systems, `/usr/local/bin` is not in the user's `$PATH`. In these cases, either add `/usr/local/bin` to the path or refer to `replicatedctl` with the full path.

To extract the settings as JSON, run via ssh on the instance:

```
ptfe$ replicatedctl app-config export > settings.json
```

Here is an example app-config export for an instance configured in demo mode:

```

ptfe$ replicatedctl app-config export > settings.json
ptfe$ cat settings.json
{
    "aws_access_key_id": {},
    "aws_instance_profile": {},
    "aws_secret_access_key": {},
    "azure_account_key": {},
    "azure_account_name": {},
    "azure_container": {},
    "azure_endpoint": {},
    "ca_certs": {},
    "capacity_concurrency": {},
    "capacity_memory": {},
    "disk_path": {},
    "extern_vault_addr": {},
    "extern_vault_enable": {},
    "extern_vault_path": {},
    "extern_vault_role_id": {},
    "extern_vault_secret_id": {},
    "extern_vault_token_renew": {},
    "extra_no_proxy": {},
    "gcs_bucket": {},
    "gcs_credentials": {},
    "gcs_project": {},
    "hostname": {
        "value": "tfe.mycompany.com"
    },
    "installation_type": {
        "value": "poc"
    },
    "pg_dbname": {},
    "pg_extra_params": {},
    "pg_netloc": {},
    "pg_password": {},
    "pg_user": {},
    "placement": {},
    "production_type": {},
    "s3_bucket": {},
    "s3_endpoint": {},
    "s3_region": {},
    "s3_sse": {},
    "s3_sse_kms_key_id": {},
    "vault_path": {},
    "vault_store_snapshot": {}
}

```

Note that when you build your own settings file, you do not need to include parameters that do not have value keys, such as extra_no_proxy in the output above.

Available settings

A number of settings are available to configure and tune your installation. They are summarized below; it is expected the user will have completed a manual installation first and already be familiar with the nature of these parameters from the settings screen.

The following apply to every installation:

- **hostname** — (Required) this is the hostname you will use to access your installation

- `installation_type` — (Required) one of `poc` or `production`
- `enc_password` — Set the encryption password (<https://www.terraform.io/docs/enterprise/private/encryption-password.html>) for the install
- `capacity_concurrency` — number of concurrent plans and applies; defaults to 10
- `capacity_memory` — The maximum amount of memory (in megabytes) that a Terraform plan or apply can use on the system; defaults to 256
- `enable_metrics_collection` — whether PTFE's internal metrics collection (</docs/enterprise/private/monitoring.html#internal-monitoring>) should be enabled; defaults to true
- `extra_no_proxy` — (Optional) when configured to use a proxy, a , (comma) separated list of hosts to exclude from proxying. Please note that this list does not support whitespace characters. For example:
`127.0.0.1,tfe.myapp.com,myco.github.com.`
- `ca_certs` — (Optional) custom certificate authority (CA) bundle. JSON does not allow raw newline characters, so replace any newlines in the data with `\n`. For instance:

```
--- X509 CERT ---
aabbcdddeeff
--- X509 CERT ---
```

would become

```
--- X509 CERT ---\naabbcdddeeff\n--- X509 CERT ---\n
```

- `extern_vault_enable` — (Optional) Indicate if an external Vault cluster is being used. Set to 1 if so.
 - These variables are only used if `extern_vault_enable` is set to 1
 - `extern_vault_addr` — (Required) URL of external Vault cluster
 - `extern_vault_role_id` — (Required) AppRole RoleId to use to authenticate with the Vault cluster
 - `extern_vault_secret_id` — (Required) AppRole SecretId to use to authenticate with the Vault cluster
 - `extern_vault_path` — (Optional) Path on the Vault server for the AppRole auth. Defaults to `auth/approle`
 - `extern_vault_token_renew` — (Optional) How often (in seconds) to renew the Vault token. Defaults to 3600
- `vault_path` — (Optional) Path on the host system to store the vault files. If `extern_vault_enable` is set, this has no effect.
- `vault_store_snapshot` — (Optional) Indicate if the vault files should be stored in snapshots. Set to 0 if not, Defaults to 1.

`production_type` is required if you've chosen `production` for the `installation_type`:

- `production_type` — one of `external` or `disk`

`disk_path` is required if you've chosen `disk` for `production_type`:

- `disk_path` — path on instance to persistent storage

The following apply if you've chosen `external` for `production_type`:

- `pg_user` — (Required) PostgreSQL user to connect as
- `pg_password` — (Required) The password for the PostgreSQL user
- `pg_netloc` — (Required) The hostname and port of the target PostgreSQL server in the format `hostname:port`
- `pg_dbname` — (Required) The database name
- `pg_extra_params` — (Optional) Parameter keywords of the form `param1=value1¶m2=value2` to support additional options that may be necessary for your specific PostgreSQL server. Allowed values are documented on the PostgreSQL site (<https://www.postgresql.org/docs/9.4/static/libpq-connect.html#LIBPQ-PARAMKEYWORDS>). An additional restriction on the `sslmode` parameter is that only the `require`, `verify-full`, `verify-ca`, and `disable` values are allowed.

The system can use either S3 or Azure, so you only need to provide one set of the following variables.

Select which will be used, S3 or Azure:

- `placement` — (Required) Set to `placement_s3` for S3, `placement_azure` for Azure, or `placement_gcs` for GCS

For S3 (or S3-compatible storage providers):

- `aws_instance_profile` (Optional) When set, use credentials from the AWS instance profile. Set to 1 to use the instance profile. Defaults to 0. If selected, `aws_access_key_id` and `aws_secret_access_key` are not required.
- `aws_access_key_id` — (Required unless `aws_instance_profile` is set) AWS access key ID for S3 bucket access. To use AWS instance profiles for this information, set it to "".
- `aws_secret_access_key` — (Required unless `aws_instance_profile` is set) AWS secret access key for S3 bucket access. To use AWS instance profiles for this information, set it to "".
- `s3_endpoint` — (Optional) Endpoint URL (hostname only or fully qualified URI). Usually only needed if using a VPC endpoint or an S3-compatible storage provider.
- `s3_bucket` — (Required) the S3 bucket where resources will be stored
- `s3_region` — (Required) the region where the S3 bucket exists
- `s3_sse` — (Optional) enables server-side encryption of objects in S3; if provided, must be set to `aws:kms`
- `s3_sse_kms_key_id` — (Optional) An optional KMS key for use when S3 server-side encryption is enabled

For Azure:

- `azure_account_name` — (Required) The account name for the Azure account to access the container
- `azure_account_key` — (Required) The account key to access the account specified in `azure_account_name`
- `azure_container` — (Required) The identifier for the Azure blob storage container
- `azure_endpoint` — (Optional) The URL for the Azure cluster to use. By default this is the global cluster.

For GCS:

- `gcs_credentials` — (Required) JSON blob containing the GCP credentials document. **Note:** this is a string, so ensure value is properly escaped.
- `gcs_project` — (Required) The GCP project where the bucket resides.
- `gcs_bucket` — (Required) The storage bucket name.

Online

The following is an example `/etc/replicated.conf` suitable for an automated online install using a certificate trusted by a public or private CA. `ImportSettingsFrom` must be the full path to the application settings file. You also need to provide the full path to your license file in `LicenseFileLocation`.

See the full set of configuration parameters in the Replicated documentation (<https://help.replicated.com/docs/kb/developer-resources/automate-install/#configure-replicated-automatically>).

```
{  
    "DaemonAuthenticationType": "password",  
    "DaemonAuthenticationPassword": "your-password-here",  
    "TlsBootstrapType": "server-path",  
    "TlsBootstrapHostname": "server.company.com",  
    "TlsBootstrapCert": "/etc/server.crt",  
    "TlsBootstrapKey": "/etc/server.key",  
    "BypassPreflightChecks": true,  
    "ImportSettingsFrom": "/path/to/application-settings.json",  
    "LicenseFileLocation": "/path/to/license.rli"  
}
```

Invoking the installation

Once `/etc/replicated.conf` has been created, you can retrieve and execute the install script as `root`:

```
curl -o install.sh https://install.terraform.io/ptfe/stable  
bash ./install.sh \  
no-proxy \  
private-address=1.2.3.4 \  
public-address=5.6.7.8
```

Note the `private-address` and `public-address` flags provided to the installer. These may be left out, but the installer will prompt for them if it is unable to determine appropriate values automatically.

Airgapped

The following is an example `/etc/replicated.conf` suitable for an automated airgapped install, which builds on the online example above. Note the addition of `LicenseBootstrapAirgapPackagePath`, which is a path to the `.airgap` bundle on the instance.

```
{
  "DaemonAuthenticationType": "password",
  "DaemonAuthenticationPassword": "your-password-here",
  "TlsBootstrapType": "server-path",
  "TlsBootstrapHostname": "server.company.com",
  "TlsBootstrapCert": "/etc/server.crt",
  "TlsBootstrapKey": "/etc/server.key",
  "BypassPreflightChecks": true,
  "ImportSettingsFrom": "/path/to/application-settings.json",
  "LicenseFileLocation": "/path/to/license.rli",
  "LicenseBootstrapAirgapPackagePath": "/path/to/bundle.airgap"
}
```

Invoking the installation

Following on from the manual airgapped install ([/docs/enterprise/private/install-installer.html#run-the-installer-airgapped](#)) steps, you must also have the installer bootstrapper already on the instance. For illustrative purposes, it is assumed the installer bootstrapper has been unarchived in /tmp.

Once /etc/replicated.conf has been created, you can now execute the install script as root:

```
cd /tmp
./install.sh \
  airgap \
  no-proxy \
  private-address=1.2.3.4 \
  public-address=5.6.7.8
```

Waiting for Terraform Enterprise to become ready

Once the installer finishes, you may poll the /_health_check endpoint until a 200 is returned by the application, indicating that it is fully started:

```
while ! curl -ksFS --connect-timeout 5 https://tfe.example.com/_health_check; do
  sleep 5
done
```

References

- Replicated installer flags ([https://help.replicated.com/docs/distributing-an-application/installing-via-script/#flags](#))
- /etc/replicated.conf ([https://help.replicated.com/docs/kb/developer-resources/automate-install/#configure-replicated-automatically](#))
- application settings ([https://help.replicated.com/docs/kb/developer-resources/automate-install/#configure-app-settings-automatically](#))

Private Terraform Enterprise AWS Reference Architecture

This document provides recommended practices and a reference architecture for HashiCorp Private Terraform Enterprise (PTFE) implementations on AWS.

Required Reading

Prior to making hardware sizing and architectural decisions, read through the installation information available for PTFE (<https://www.terraform.io/docs/enterprise/private/install-installer.html>) to familiarise yourself with the application components and architecture. Further, read the reliability and availability guidance (<https://www.terraform.io/docs/enterprise/private/reliability-availability.html>) as a primer to understanding the recommendations in this reference architecture.

Infrastructure Requirements

Note: This reference architecture focuses on the *Production - External Services* operational mode.

Depending on the chosen operational mode (<https://www.terraform.io/docs/enterprise/private/preflight-installer.html#operational-mode-decision>), the infrastructure requirements for PTFE range from a single AWS EC2 instance for demo installations to multiple instances connected to RDS and S3 for a stateless production installation.

The following table provides high-level server guidelines. Of particular note is the strong recommendation to avoid non-fixed performance CPUs, or "Burstable CPU" in AWS terms, such as T-series instances.

PTFE Server (EC2 via Auto Scaling Group)

Type	CPU	Memory	Disk	AWS Instance Types
Minimum	2 core	8 GB RAM	50GB	m5.large
Recommended	4-8 core	16-32 GB RAM	50GB	m5.xlarge, m5.2xlarge

Hardware Sizing Considerations

- The minimum size would be appropriate for most initial production deployments, or for development/testing environments.
- The recommended size is for production environments where there is a consistent high workload in the form of concurrent terraform runs.

PostgreSQL Database (RDS Multi-AZ)

Type	CPU	Memory	Storage	AWS Instance Types
Minimum	2 core	8 GB RAM	50GB	db.m4.large

Type	CPU	Memory	Storage	AWS Instance Types
Recommended	4-8 core	16-32 GB RAM	50GB	db.m4.xlarge, db.m4.2xlarge

Hardware Sizing Considerations

- The minimum size would be appropriate for most initial production deployments, or for development/testing environments.
- The recommended size is for production environments where there is a consistent high workload in the form of concurrent terraform runs.

Object Storage (S3)

An S3 Standard (<https://aws.amazon.com/s3/storage-classes/>) bucket must be specified during the PTFE installation for application data to be stored securely and redundantly away from the EC2 servers running the PTFE application. This S3 bucket must be in the same region as the EC2 and RDS instances. It is recommended the VPC containing the PTFE servers be configured with a VPC endpoint for S3 (<https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc-endpoints.html>). Within the PTFE application, Vault is used to encrypt all application data stored in the S3 bucket. This allows for further server-side encryption (<https://docs.aws.amazon.com/AmazonS3/latest/dev/serv-side-encryption.html>) by S3 if required by your security policy.

Other Considerations

Additional AWS Resources

In order to successfully provision this reference architecture you must also be permitted to create the following AWS resources:

- VPC
- Subnet
- Route Table
- Route Table Association
- Security Group
- Load Balancer (Application, Network, or Classic Load Balancer)
- Launch Configuration
- Auto Scaling Group
- Target Group (if using Application or Network Load Balancer)
- CloudWatch Alarm
- IAM Instance Profile

- IAM Role
- IAM Role Policy
- Route 53 (optional)

Network

To deploy PTFE in AWS you will need to create new or use existing networking infrastructure. The below infrastructure diagram highlights some of the key components (VPC, subnets, DB subnet group) and you will also have security group, routing table and gateway requirements. These elements are likely to be very unique to your environment and not something this Reference Architecture can specify in detail. An example Terraform configuration (<https://github.com/hashicorp/private-terraform-enterprise/blob/master/examples/aws/network/main.tf>) is provided to demonstrate how these resources can be provisioned and how they interrelate.

DNS

DNS can be configured external to AWS or using Route 53 (<https://aws.amazon.com/route53/>). The fully qualified domain name should resolve to the Load Balancer (if using one) or the PTFE instance using a CNAME if using external DNS or an alias record set (<https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/resource-record-sets-choosing-alias-non-alias.html>) if using Route 53. Creating the required DNS entry is outside the scope of this guide.

Another approach would be to use an external registrar or DNS server to point to a Route 53 CNAME record using a canonical, but not necessarily public, domain name, which then forwards to the ALIAS record for the ELB. This pattern is required if using Route 53 Health Checks and failover pairs to automatically fail over to the standby instance. This is documented further below.

SSL/TLS Certificates and Load Balancers

An SSL/TLS certificate signed by a public or private CA is required for secure communication between clients, VCS systems, and the PTFE application server. The certificate can be specified during the UI-based installation or in a configuration file used for an unattended installation.

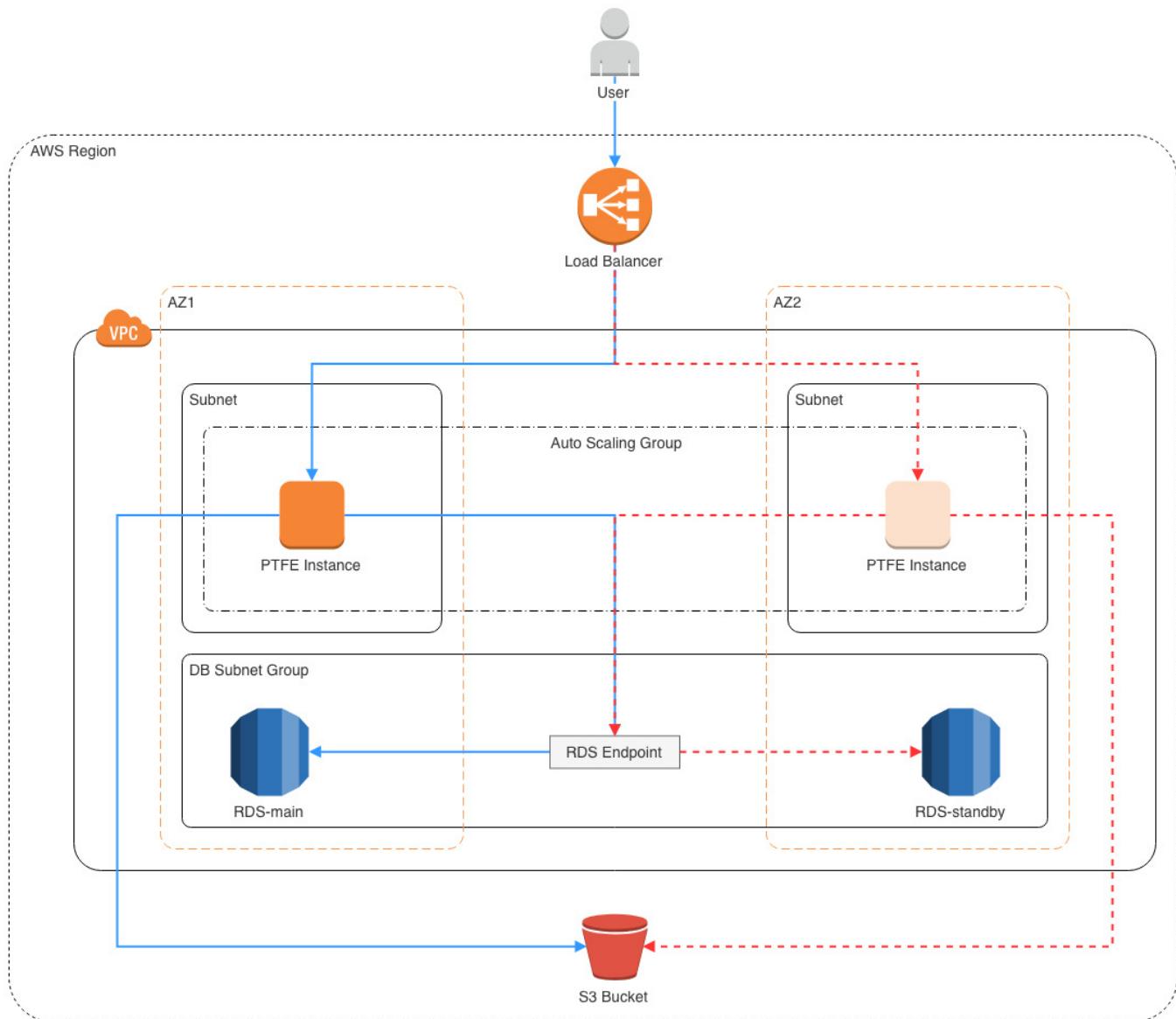
If a Classic or Application Load Balancer is used, SSL/TLS will be terminated on the load balancer. In this configuration, the PTFE instances should still be configured to listen for incoming SSL/TLS connections. If a Network Load Balancer is used, SSL/TLS will be terminated on the PTFE instance.

HashiCorp does not recommend the use of self-signed certificates on the PTFE instance unless you use a Classic or Application Load Balancer and place a public certificate (such as an AWS Certificate Manager certificate) on the load balancer. Note that certificates cannot be placed on Network Load Balancers.

If you want to use a Network Load Balancer (NLB) with PTFE, use either an internet-facing NLB or an internal NLB that targets by IP. An internal NLB that targets by instance ID cannot be used with PTFE since NLBs configured in this way do not support loopbacks. Amazon provides load balancer troubleshooting (<https://docs.aws.amazon.com/elasticloadbalancing/latest/network/load-balancer-troubleshooting.html>) information for Network Load Balancers.

A public AWS Certificate Manager (ACM) certificate cannot be used with a Network Load Balancer and PTFE since certificates cannot be placed on NLBs and AWS does not support exporting the private key for public ACM certificates. This means you cannot load the private key of a public ACM certificate on your PTFE instance.

Infrastructure Diagram



Application Layer

The Application Layer is composed of an Auto Scaling Group and a Launch Configuration providing an auto-recovery mechanism in the event of an instance or Availability Zone failure.

Storage Layer

The Storage Layer is composed of multiple service endpoints (RDS, S3) all configured with or benefiting from inherent resiliency provided by AWS.

Additional Information

- RDS Multi-AZ deployments (<https://aws.amazon.com/rds/details/multi-az/>).
- S3 Standard storage class (<https://aws.amazon.com/s3/storage-classes/>).

Infrastructure Provisioning

The recommended way to deploy PTFE is through use of a Terraform configuration that defines the required resources, their references to other resources, and dependencies. An example Terraform configuration (<https://github.com/hashicorp/private-terraform-enterprise/blob/master/examples/aws/pes/main.tf>) is provided to demonstrate how these resources can be provisioned and how they interrelate. This Terraform configuration assumes the required networking components are already in place. If you are creating networking components for this PTFE installation, an example Terraform configuration is available for the networking resources (<https://github.com/hashicorp/private-terraform-enterprise/blob/master/examples/aws/network/main.tf>) as well.

Normal Operation

Component Interaction

The Load Balancer routes all traffic to the *PTFE* instance, which is managed by an Auto Scaling Group with maximum and minimum instance counts set to one.

The PTFE application is connected to the PostgreSQL database via the RDS Multi-AZ endpoint and all database requests are routed via the RDS Multi-AZ endpoint to the *RDS-main* database instance.

The PTFE application is connected to object storage via the S3 endpoint for the defined bucket and all object storage requests are routed to the highly available infrastructure supporting S3.

Upgrades

See the Upgrades section (</docs/enterprise/private/upgrades.html>) of the documentation.

High Availability

Failure Scenarios

AWS provides availability and reliability recommendations in the Well-Architected framework (<https://aws.amazon.com/architecture/well-architected/>). Working in accordance with those recommendations the PTFE Reference Architecture is designed to handle different failure scenarios with different probabilities. As the architecture evolves it may provide a higher level of service continuity.

Single EC2 Instance Failure

In the event of the *PTFE* instance failing in a way that AWS can observe, the health checks on the Auto Scaling Group trigger, causing a new instance to be launched. Once the new EC2 instance is launched, it reinitializes the software and once that is complete, service would resume as normal.

Availability Zone Failure

In the event of the Availability Zone hosting the main instances (EC2 and RDS) failing, the Auto Scaling Group for the EC2 instance will automatically begin booting a new one in an operational AZ.

- Multi-AZ RDS automatically fails over to the RDS Standby Replica (*RDS-standby*). The AWS documentation provides more detail (<https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html>) on the exact behaviour and expected impact.
- S3 is resilient to Availability Zone failure based on its architecture.

See below for more detail on how each component handles Availability Zone failure.

PTFE Server

By utilizing an Auto Scaling Group, the PTFE instance automatically recovers in the event of any outage except for the loss of an entire region.

With external services (PostgreSQL Database, Object Storage) in use, there is still some application configuration data present on the PTFE server such as installation type, database connection settings, hostname. This data rarely changes. If the configuration on *PTFE* changes you should update the Launch Configuration to include this updated configuration so that any newly launched EC2 instance uses this new configuration.

PostgreSQL Database

Using RDS Multi-AZ as an external database service leverages the highly available infrastructure provided by AWS. From the AWS website:

In a Multi-AZ deployment, Amazon RDS automatically provisions and maintains a synchronous standby replica in a different Availability Zone. In the event of a planned or unplanned outage of your DB instance, Amazon RDS automatically switches to a standby replica in another Availability Zone. (source (<https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html>))

Object Storage

Using S3 as an external object store leverages the highly available infrastructure provided by AWS. S3 buckets are replicated to all Availability Zones within the region selected during bucket creation. From the AWS website:

Amazon S3 runs on the world's largest global cloud infrastructure, and was built from the ground up to deliver a customer promise of 99.99999999% of durability. Data is automatically distributed across a minimum of three physical facilities that are geographically separated within an AWS Region. (source (<https://aws.amazon.com/s3/>))

Disaster Recovery

Failure Scenarios

AWS provides availability and reliability recommendations in the Well-Architected framework. Working in accordance with those recommendations the PTFE Reference Architecture is designed to handle different failure scenarios that have different probabilities. As the architecture evolves it may provide a higher level of service continuity.

Region Failure

PTFE is currently architected to provide high availability within a single AWS Region. Using multiple AWS Regions will give you greater control over your recovery time in the event of a hard dependency failure on a regional AWS service. In this section, we'll discuss various implementation patterns and their typical availability.

An identical infrastructure should be provisioned in a secondary AWS Region. Depending on recovery time objectives and tolerances for additional cost to support AWS Region failure, the infrastructure can be running (Warm Standby) or stopped (Cold Standby). In the event of the primary AWS Region hosting the PTFE application failing, the secondary AWS Region will require some configuration before traffic is directed to it along with some global services such as DNS.

- RDS cross-region read replicas
(https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ReadRepl.html#USER_ReadRepl.XRgn) can be used in a warm standby architecture or RDS database backups
(https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_CommonTasks.BackupRestore.html) can be used in a cold standby architecture.
- S3 cross-region replication (<https://docs.aws.amazon.com/AmazonS3/latest/dev/crr.html>) must be configured so the object storage component of the Storage Layer is available in the secondary AWS Region.
- DNS must be redirected to the Load Balancer acting as the entry point for the infrastructure deployed in the secondary AWS Region.

Data Corruption

The PTFE application architecture relies on multiple service endpoints (RDS, S3) all providing their own backup and recovery functionality to support a low MTTR in the event of data corruption.

PTFE Servers

With external services (PostgreSQL Database, Object Storage) in use, there is still some application configuration data present on the PTFE server such as installation type, database connection settings, hostname. This data rarely changes. We recommend configuring automated snapshots (<https://www.terraform.io/docs/enterprise/private/automated-recovery.html#1-configure-snapshots>) for this installation data so it can be recovered in the event of data corruption.

PostgreSQL Database

Backup and recovery of PostgreSQL is managed by AWS and configured through the AWS management console on CLI. More details of RDS for PostgreSQL features are available here (<https://aws.amazon.com/rds/postgresql/>) and summarised below:

Automated Backups – The automated backup feature of Amazon RDS is turned on by default and enables point-in-time recovery for your DB Instance. Amazon RDS will backup your database and transaction logs and store both for a user-specified retention period.

DB Snapshots – DB Snapshots are user-initiated backups of your DB Instance. These full database backups will be stored by Amazon RDS until you explicitly delete them.

Object Storage

There is no automatic backup/snapshot of S3 by AWS, so it is recommended to script a bucket copy process from the bucket used by the PTFE application to a “backup bucket” in S3 that runs at regular intervals. The Amazon S3 Standard-Infrequent Access (<https://aws.amazon.com/s3/storage-classes/>) storage class is identified as a solution targeted more for DR backups than S3 Standard. From the AWS website:

Amazon S3 Standard-Infrequent Access (S3 Standard-IA) is an Amazon S3 storage class for data that is accessed less frequently, but requires rapid access when needed. S3 Standard-IA offers the high durability, high throughput, and low latency of S3 Standard, with a low per GB storage price and per GB retrieval fee. This combination of low cost and high performance make S3 Standard-IA ideal for long-term storage, backups, and as a data store for disaster recovery. (source (<https://aws.amazon.com/s3/storage-classes/>))

Private Terraform Enterprise Azure Reference Architecture

Introduction

This document provides recommended practices and a reference architecture for HashiCorp Private Terraform Enterprise (PTFE) implementations on Azure.

Required Reading

Prior to making hardware sizing and architectural decisions, read through the installation information available for PTFE (<https://www.terraform.io/docs/enterprise/private/install-installer.html>) to familiarise yourself with the application components and architecture. Further, read the reliability and availability guidance (<https://www.terraform.io/docs/enterprise/private/reliability-availability.html>) as a primer to understanding the recommendations in this reference architecture.

Infrastructure Requirements

Note: This reference architecture focuses on the *Production - External Services* operational mode.

Depending on the chosen operational mode (<https://www.terraform.io/docs/enterprise/private/preflight-installer.html#operational-mode-decision>), the infrastructure requirements for PTFE range from a single Azure VM instance (<https://azure.microsoft.com/en-us/services/virtual-machines/>) for demo or proof of concept installations, to multiple instances connected to Azure Database for PostgreSQL (<https://azure.microsoft.com/en-us/services/postgresql/>), Azure Blob Storage (<https://azure.microsoft.com/en-us/services/storage/blobs/>), and an external Vault cluster for a stateless production installation.

The following table provides high level server recommendations, and is meant as a guideline. Of particular note is the strong recommendation to avoid non-fixed performance CPUs, or "Burstable CPU" in Azure terms, such as B-series instances.

PTFE Servers (Azure VMs)

Type	CPU	Memory	Disk	Azure VM Sizes
Minimum	2 core	8 GB RAM	50GB	Standard_D2_v3
Recommended	4-8 core	16-32 GB RAM	50GB	Standard_D4_v3, Standard_D8_v3

Hardware Sizing Considerations

- The default osDisk size for most Linux images on Azure is 30GB. When increasing the size of the osDisk partition, there may be additional steps required to fully utilize the disk space, such as using a tool like fdisk. This process is documented in the Azure knowledge base article "How to: Resize Linux osDisk partition on Azure" (<https://blogs.msdn.microsoft.com/linuxonazure/2017/04/03/how-to-resize-linux-osdisk-partition-on-azure/>).

- The minimum size would be appropriate for most initial production deployments or for development/testing environments.
- The recommended size is for production environments where there is a consistently high workload in the form of concurrent terraform runs.

PostgreSQL Database (Azure Database for PostgreSQL)

Type	CPU	Memory	Storage	Azure DB Sizes
Minimum	2 core	4 GB RAM	50GB	General Purpose 2 vCores
Recommended	4-8 core	8-16 GB RAM	50GB	General Purpose 4 vCores, General Purpose 8 vCores

Hardware Sizing Considerations

- The minimum size would be appropriate for most initial production deployments, or for development/testing environments.
- The recommended size is for production environments where there is a consistent high workload in the form of concurrent terraform runs.

Object Storage (Azure Blob Storage)

An Azure Blob Storage container (<https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction#container>) must be specified during the PTFE installation for application data to be stored securely and redundantly away from the Azure VMs running the PTFE application. This Azure Blob Storage container must be in the same region as the VMs and Azure Database for PostgreSQL instance. It is recommended the virtual network containing the PTFE servers be configured with a Virtual Network (VNet) service endpoint (<https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-service-endpoints-overview>) for Azure Storage. Vault is used to encrypt all application data stored in the Azure Blob Storage container. This allows for further server-side encryption (<https://docs.microsoft.com/en-us/azure/storage/common/storage-service-encryption>) by Azure Blob Storage if required by your security policy.

Vault Cluster

In order to provide a fully stateless application deployment, PTFE must be configured to speak with an external Vault cluster (<https://www.terraform.io/docs/enterprise/private/vault.html>). This reference architecture assumes that a highly available Vault cluster is accessible at an endpoint the PTFE servers can reach.

Other Considerations

Additional Azure Resources

In order to successfully provision this reference architecture you must also be permitted to create the following Azure resources:

- Resource Group(s) (<https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview#resource-groups>)
- Load Balancer (<https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview>)
- Virtual Network (<https://azure.microsoft.com/en-us/services/virtual-network/>)
- Subnet (<https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-manage-subnet>)
- Public IP (<https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-ip-addresses-overview-arm#public-ip-addresses>)
- Managed Disk (<https://azure.microsoft.com/en-us/services/managed-disks/>)
- Network Interface (<https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-network-interface>)

Network

To deploy PTFE in Azure you will need to create new or use existing networking infrastructure. The infrastructure diagram highlights some of the key components. These elements are likely to be very unique to your environment and not something this Reference Architecture can specify in detail.

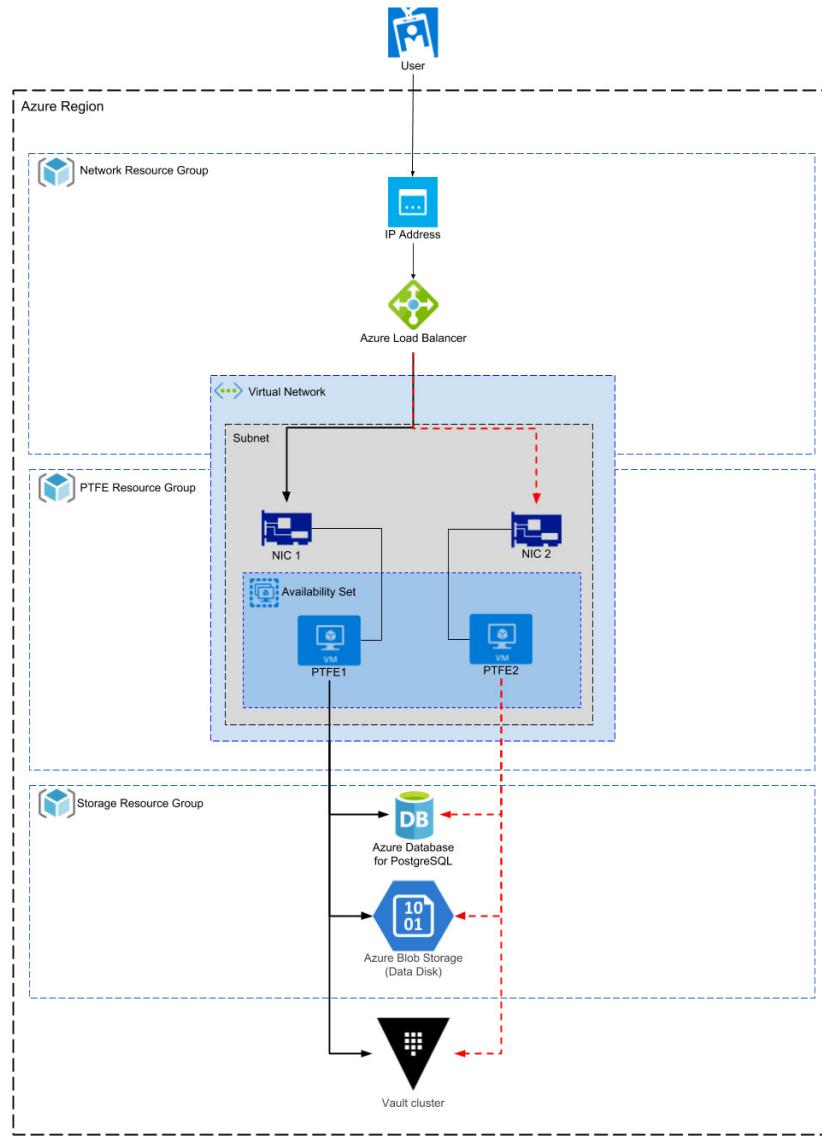
DNS

DNS can be configured outside of Azure or using Azure DNS (<https://azure.microsoft.com/en-gb/services/dns/>). The fully qualified domain name should resolve to the Load Balancer. Creating the required DNS entry is outside the scope of this guide.

SSL/TLS

An SSL/TLS certificate is required for secure communication between clients and the PTFE application server. The certificate can be specified during the UI-based installation or the path to the certificate codified during an unattended installation.

Infrastructure Diagram



The above diagram shows the infrastructure components at a high-level.

Application Layer

The Application Layer is composed of two PTFE servers (Azure VMs) running in different subnets and operating in an active/standby configuration. Traffic is routed to the active PTFE server via the Load Balancer rules and health checks. In the event that the active PTFE server becomes unavailable, the traffic will then route to the standby PTFE server, making it the new active server. Routing changes can also be managed by a human triggering by triggering a change in the Load Balancer configuration to switch between the PTFE servers.

Storage Layer

The Storage Layer is composed of multiple service endpoints (Azure Database for PostgreSQL, Azure Blob Storage, Vault) all configured with or benefitting from inherent resiliency provided by Azure (in the case of Azure Database for PostgreSQL and

Azure Blob Storage) or assumed resiliency provided by a well-architected deployment (in the case of Vault).

Additional Information

- Azure Database for PostgreSQL deployments (<https://docs.microsoft.com/en-us/azure/postgresql/concepts-business-continuity>)
- Azure Blob Storage (<https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction>)
- Highly available Vault deployments (<https://www.vaultproject.io/guides/operations/vault-ha-consul.html>)

Infrastructure Provisioning

The recommended way to deploy PTFE is through use of a Terraform configuration that defines the required resources, their references to other resources, and associated dependencies.

Normal Operation

Component Interaction

The Load Balancer routes all traffic to the active PTFE instance, which handles all requests to the PTFE application.

The PTFE application is connected to the PostgreSQL database via the Azure provided database server name endpoint. All database requests are routed to the highly available infrastructure supporting Azure Database for PostgreSQL.

The PTFE application is connected to object storage via the Azure Blob Storage endpoint for the defined container. All object storage requests are routed to the highly available infrastructure supporting Azure Storage.

The PTFE application is connected to the Vault cluster via the Vault cluster endpoint URL.

Monitoring

While there is not currently a monitoring guide for PTFE, information around logging (<https://www.terraform.io/docs/enterprise/private/logging.html>), diagnostics (<https://www.terraform.io/docs/enterprise/private/diagnostics.html>) as well as reliability and availability (<https://www.terraform.io/docs/enterprise/private/reliability-availability.html>) can be found on our website.

Upgrades

See the Upgrades section (<https://www.terraform.io/docs/enterprise/private/upgrades.html>) of the documentation.

High Availability

Failure Scenarios

The PTFE Reference Architecture is designed to handle different failure scenarios that have different probabilities. The ability to provide better service continuity will improve as the architecture evolves.

Component Failure

Single VM Failure

In the event of the active instance failing, the Load Balancer should be reconfigured (manually or automatically) to route all traffic to the standby instance.

Important: Active-active configuration is not supported due to a serialisation requirement in the core components of PTFE; therefore, all traffic from the Load Balancer *MUST* be routed to a single instance.

When using the *Production - External Services* deployment model (PostgreSQL Database, Object Storage, Vault), there is still some application configuration data present on the PTFE server such as installation type, database connection settings, and hostname; however, this data rarely changes. If the application configuration has not changed since installation, both PTFE1 and PTFE2 will use the same configuration and no action is required.

If the configuration on the active instance changes, you should create a snapshot (<https://www.terraform.io/docs/enterprise/private/automated-recovery.html#1-configure-snapshots>) via the UI or CLI and recover this to the standby instance so that both instances use the same configuration.

PostgreSQL Database

The Azure Database for PostgreSQL service provides a guaranteed high level of availability. The financially backed service level agreement (SLA) is 99.99% upon general availability. There is virtually no application down time when using this service. More information on Azure Database for PostgreSQL service redundancy is available in the Azure documentation (<https://docs.microsoft.com/en-us/azure/postgresql/concepts-high-availability>).

Object Storage

Using Azure Blob Storage as an external object store leverages the highly available infrastructure provided by Azure. More information on Azure Storage redundancy is available in the Azure documentation (<https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy>).

Vault Servers

For the purposes of this guide, the external Vault cluster is expected to be deployed and configured in line with the HashiCorp Vault Enterprise Reference Architecture (<https://www.vaultproject.io/guides/operations/reference-architecture.html>). This would provide high availability and disaster recovery support, minimising downtime in the event of an outage.

Disaster Recovery

Failure Scenarios

The PTFE Reference Architecture is designed to handle different failure scenarios that have different probabilities. The ability to provide better service continuity will improve as the architecture evolves.

Region Failure

PTFE is currently architected to provide high availability within a single Azure Region. Using multiple Azure Regions will give you greater control over your recovery time in the event of a hard dependency failure on a regional Azure service. In this section, we'll discuss various implementation patterns and their typical availability.

An identical infrastructure should be provisioned in a secondary Azure Region. In the event of the primary Azure Region hosting the PTFE application failing, the secondary Azure Region will require some configuration before traffic is directed to it along with some global services such as DNS.

- Azure Database for PostgreSQL's geo-restore feature (<https://docs.microsoft.com/en-us/azure/postgresql/concepts-business-continuity>) provides the ability to recover the database backup to the secondary Azure Region
- Geo-redundant storage (GRS) for Azure Storage (<https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy-grs>) must be configured so the object storage component of the Storage Layer is available in the secondary Azure Region.
- Vault Disaster Recovery (DR) Replication (<https://www.vaultproject.io/docs/enterprise/replication/index.html#performance-replication-and-disaster-recovery-dr-replication>) must be configured for a Vault cluster in the secondary Azure Region.
- DNS must be redirected to the Load Balancer acting as the entry point for the infrastructure deployed in the secondary Azure Region.

Data Corruption

The PTFE application architecture relies on multiple service endpoints (Azure DB, Azure Storage, Vault) all providing their own backup and recovery functionality to support a low MTTR in the event of data corruption.

PTFE Servers

When using the *Production - External Services* deployment model (PostgreSQL Database, Object Storage, Vault), there is still some application configuration data present on the PTFE server such as installation type, database connection settings, and hostname; however, this data rarely changes. We recommend configuring automated snapshots (<https://www.terraform.io/docs/enterprise/private/automated-recovery.html#1-configure-snapshots>) for this installation data so it can be recovered in the event of data corruption.

PostgreSQL Database

Backup and recovery of PostgreSQL is managed by Azure and configured through the Azure portal or CLI. More details of Azure DB for PostgreSQL features are available here (<https://docs.microsoft.com/en-us/azure/postgresql/concepts-backup>) and summarised below:

Automated Backups – Azure Database for PostgreSQL automatically creates server backups and stores them in user configured locally redundant or geo-redundant storage.

Backup redundancy – Azure Database for PostgreSQL provides the flexibility to choose between locally redundant or geo-redundant backup storage.

Object Storage

There is no automatic backup/snapshot of Azure Blob Storage by Azure, so it is recommended to script a container copy process from the container used by the PTFE application to a “backup container” in Azure Blob Storage that runs at regular intervals. It is important the copy process is not so frequent that data corruption in the source content is copied to the

backup before it is identified.

Vault Cluster

The recommended Vault Reference Architecture uses Consul for storage. Consul provides the underlying snapshot functionality (<https://www.consul.io/docs/commands/snapshot.html>) to support Vault backup and recovery. Vault Backup/Restore doc (https://docs.google.com/document/d/1_RzV5xqgjDG-krFht0T3Deehw37HIMdU24uc2tqRwRk/edit).

Private Terraform Enterprise Capacity and Performance (Installer)

The maximum capacity and performance of Private Terraform Enterprise (PTFE) is dependent entirely on the resources provided by the Linux instance it is installed on. There are a few settings that allow the capacity that PTFE uses to be adjusted to suit the instance.

Memory + Concurrency

The amount of memory to allocate to a Terraform run and the number of concurrent runs are the primary elements in understanding capacity above the base services.

By default, PTFE allocates 256 MB of memory to each Terraform run, with a default concurrency of 10 parallel runs. Therefore, by default PTFE requires 2.6 GB of memory reserved for runs.

After factoring in the memory needed to run the base services that make up the application, the default memory footprint of PTFE is approximately 4 GB.

Settings

The settings for per-run memory and concurrency are available in the dashboard on port 8800, on the Settings page, under the Capacity section. They can also be set via the application settings JSON file when using the automated install procedure (<https://www.terraform.io/docs/enterprise/private/automating-the-installer.html#available-settings>).

Increasing Capacity

To increase the number of concurrent runs, adjust the **Capacity** setting. Note that this setting is not limited by system checks; it depends on the operator to provide enough memory to the system to accommodate the requested concurrent capacity. For instance, if **Capacity** is set to 100, the instance would require, at a minimum, 26 GB of memory reserved for Terraform runs.

Adjusting Memory

The default memory limit of 256 MB per Terraform run is also configurable. Note that this setting is not limited by system checks; it depends on the operator to provide enough memory to the system to accommodate the requested limits. If the memory limit is adjusted to 512 MB with the default capacity of 10, the instance would require, at a minimum, 5.2 GB of memory reserved for Terraform runs.

Downward Adjustment

We do not recommend adjusting the memory limit below 256 MB. Memory is Terraform's primary resource and it becomes easy for it to go above smaller limits and be terminated mid-run by the Linux kernel.

CPU

The required CPU resources for an individual Terraform run vary considerably, but in general they are a much more minor factor than memory due to Terraform mostly waiting on IO from APIs to return.

Our rule of thumb is 10 Terraform runs per CPU core, with 2 CPU cores allocated for the base PTFE services. So an 8-core instance with 8GB of memory could comfortably run 20 terraform runs, if the runs are allocated the default 256 MB each.

Disk

The amount of disk storage available to a system plays a small role in the capacity of an instance. A root volume with 200GB of storage can sustain a capacity well over 100 concurrent runs.

Disk I/O

Because of the amount of churn caused by container creation as well as Terraform state management, highly concurrent setups will begin pushing hard on disk I/O. In cloud environments like AWS that limit disk I/O to IOPS that are credited per disk, it's important to provision a minimum number to prevent I/O related stalls.

This resource is harder to predict than memory or CPU usage because it varies per Terraform module, but in general we suggest 35 IOPS per concurrent terraform run. So if an instance is configured for 20 concurrent runs, the disk should have 700 IOPS allocated. For reference, on AWS, an EBS volume with an allocated size of 250 GB comes with a steady state of 750 IOPS.

Private Terraform Enterprise Installation (Installer) - CentOS Install Guide

This install guide is specifically for users of Private Terraform Enterprise installing the product on CentOS Linux.

Install Recommendations

- CentOS Linux version 7.1611+
- A suitable version of Docker:
 - Docker 1.13.1 (available in the [extras](#) repository)
 - Docker CE (<https://docs.docker.com/install/linux/docker-ce/centos/>) version 17.06 or later.
 - Docker EE (<https://docs.docker.com/install/linux/docker-ee/centos/>) version 17.06 or later.
 - Or you can allow the installer to install Docker for you.
- A properly configured docker storage backend, either:
 - Devicemapper configured for production usage, in accordance with the Docker documentation (<https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production>). This configuration requires a second block device available to the system to be used as a thin-pool for Docker. You may need to configure this block device before the host system is booted, depending on the hosting platform.
 - A system capable of using overlay2. The requires at least kernel version 3.10.0-693 and, if XFS is being used, the flag `f_type=1`. The full documentation on this configuration is on the Docker site (<https://docs.docker.com/storage/storagedriver/overlayfs-driver/>).

If you choose to have Docker installed via the install script, ensure that `/etc/docker/daemon.json` is set up correctly, first. The installer's default configuration sets up the devicemapper driver to use a loopback file, which is explicitly not supported, and the installation script will fail. Setting up the driver for direct-lvm usage before installation will help ensure a successful installation.

FAQ:

Can I use the Docker version in [extras](#)?

Sure! Just be sure to have at least 1.13.1.

Can an installation where `docker info` says that I'm using devicemapper with a loopback file work?

No. This is an installation that Docker provides as sample and is not supported by Private Terraform Enterprise due to the significant instability in it. Docker themselves do not suggest using this mode (<https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-loop-lvm-mode-for-testing>).

How do I know if an installation is in devicemapper loopback mode?

Run the command `docker info | grep dev/loop`. If there is any output, you're in devicemapper loopback mode. Docker may also print warning about loopback mode when you run the above command, which is another indicator.

Private Terraform Enterprise Configuration

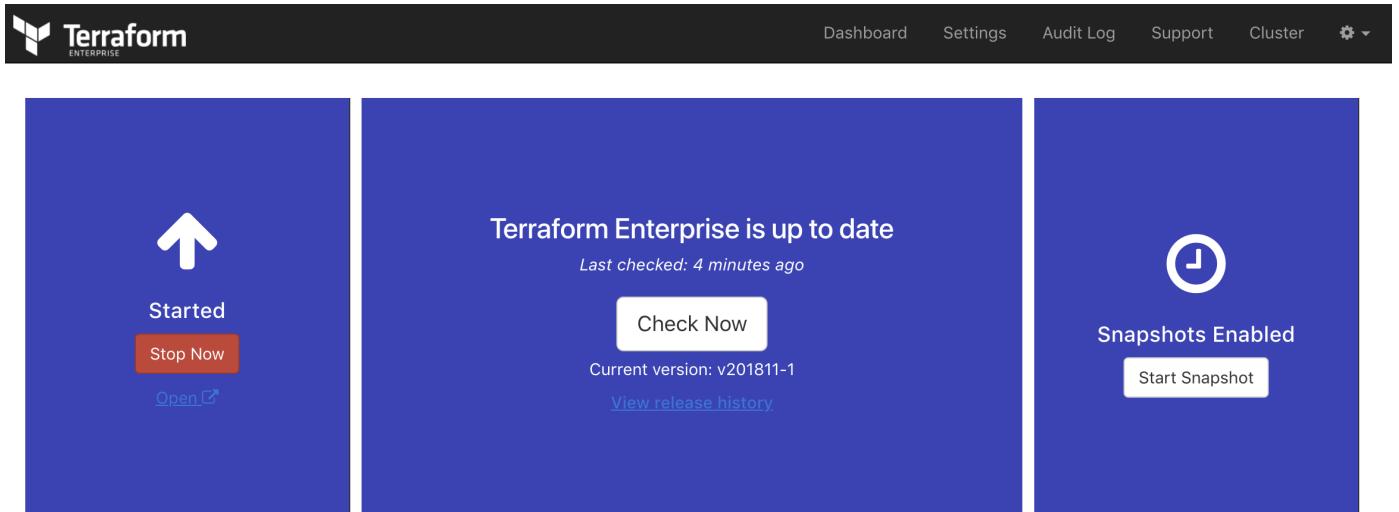
After you have completed the installation process you will need to create an admin user. When the admin user creation has been completed you will be able to create your first organizations and users and enable the enterprise features for those accounts.

Note: If you are performing an upgrade or restore for an existing installation you *do not* need to follow these steps. If your upgraded or restored installation does not function without the steps below then it was not correctly restored from backup. Please contact HashiCorp for help.

System Configuration

In all examples below, be sure to replace "<TFE HOSTNAME>" with the hostname of your Private Terraform Enterprise instance.

Navigate to <https://<TFE HOSTNAME>:8800/> in your browser. You will be presented with the installer dashboard:



Creating an Administrator

After clicking on the "Open", right below the "Stop Now" button, you will be brought to a page asking you to create the first PTFE administrator account. You will be able to create additional administrators once you log in.



Terraform
ENTERPRISE

Create an account

Congratulations! Your Terraform Enterprise install is up and running.
Please create your account. You will be given site admin access.

USERNAME

EMAIL

PASSWORD

PASSWORD CONFIRMATION

By clicking on "Create an account" below, you are agreeing to the [Terms of Use](#) and the [Privacy Policy](#).

Create an account

Creating an organization

The next step will create the first organization.



New Organization

ORGANIZATION NAME name

This will be part of your resource names used in various tools, i.e. `hashicorp/www-prod`.

EMAIL ADDRESS email

The organization email is used for any future notifications, such as billing, and the organization avatar, via [gravatar.com](#).

Create organization

After this is done, you can either continue with the creation of a new workspace, choose to configure other aspects of PTFE, or add more users.

Success!

You have successfully configured the installation and configuration steps that are specific to Private Terraform Enterprise! You can now configure SMTP (<https://<TFE HOSTNAME>/app/admin/smtp/>), Twilio (<https://<TFE HOSTNAME>/app/admin/twillio>), SAML (<https://<TFE HOSTNAME>/app/admin/saml>), or head to the Getting Started (</docs/enterprise/getting-started/index.html>) section to start using the software.

Private Terraform Enterprise - Security

Private Terraform Enterprise (PTFE) takes the security of the data it manages seriously. This table lists which parts of the PTFE app can contain sensitive data, what storage is used, and what encryption is used.

Object	Storage	Encrypted
Ingressed VCS Data	Blob Storage	Vault Transit Encryption
Terraform Plan Result	Blob Storage	Vault Transit Encryption
Terraform State	Blob Storage	Vault Transit Encryption
Terraform Logs	Blob Storage	Vault Transit Encryption
Terraform/Environment Variables	PostgreSQL	Vault Transit Encryption
Organization/Workspace/Team Settings	PostgreSQL	No
Account Password	PostgreSQL	bcrypt
2FA Recovery Codes	PostgreSQL	Vault Transit Encryption
SSH Keys	PostgreSQL	Vault Transit Encryption
User/Team/Organization Tokens	PostgreSQL	HMAC SHA512
OAuth Client ID + Secret	PostgreSQL	Vault Transit Encryption
OAuth User Tokens	PostgreSQL	Vault Transit Encryption
Twilio Account Configuration	PostgreSQL	Vault Transit Encryption
SMTP Configuration	PostgreSQL	Vault Transit Encryption
SAML Configuration	PostgreSQL	Vault Transit Encryption
Vault Unseal Key	Host	No

Vault Transit Encryption

The Vault Transit Secret Engine (<https://www.vaultproject.io/docs/secrets/transit/index.html>) handles encryption for data in-transit and is used when encrypting data from the application to the applicable storage layer (<https://www.terraform.io/docs/enterprise/private/reliability-availability.html#components>).

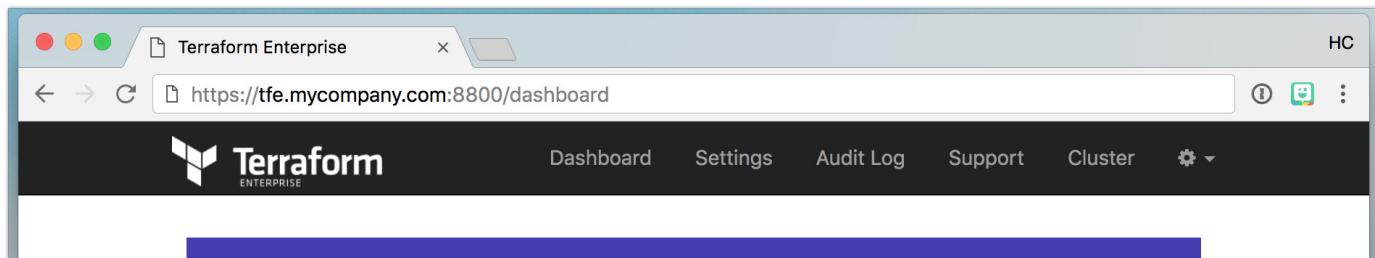
Private Terraform Enterprise Diagnostics

This document contains information on how to provide HashiCorp with diagnostic information about a Private Terraform Enterprise (PTFE) installation that requires assistance from HashiCorp support.

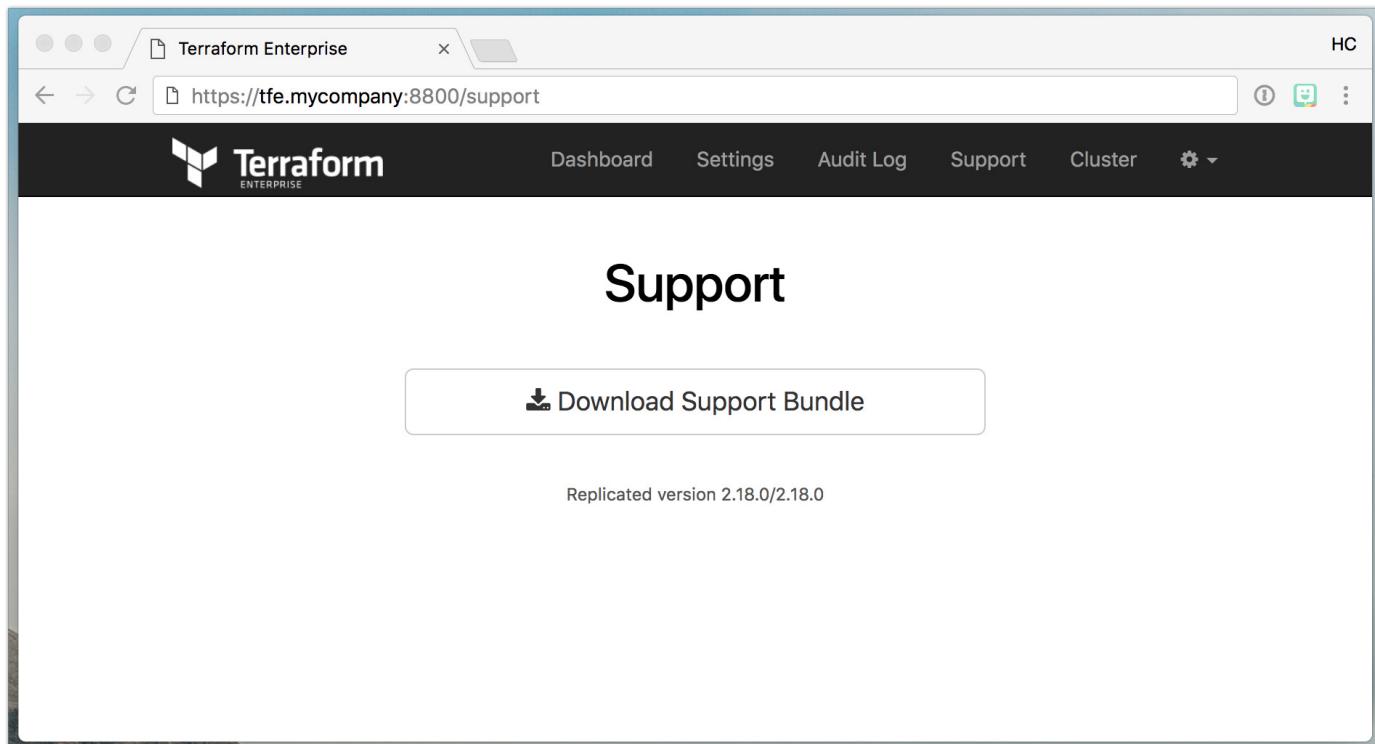
Installer-based Instances

Diagnostic information is available via in the Installer dashboard on port 8800 of your installation.

On the dashboard, click on the Support tab:



On the next page, click the *Download Support Bundle* button which will download the support bundle directly to your web browser.



Then, attach the bundle to your support ticket. If possible, use the SendSafely integration (as it allows for large file uploads).

AMI-based installs

Deprecation warning: The AMI will no longer be actively developed as of 201808-1 and will be fully decommissioned on November 30, 2018. Please see our Migration Guide ([/docs/enterprise/private/migrate.html](#)) for instructions to migrate to the new Private Terraform Enterprise Installer.

To generate a support bundle, connect to the instance via ssh and run `sudo hashicorp-support`. Below is a sample session:

For your privacy and security, the entire contents of the support bundle are encrypted with a 2048 bit RSA key to generate the `tar.gz.enc` file. Once the process is complete, copy the `hashicorp-support.tar.gz.enc` file listed above off the instance. This can be done via scp or another tool that can interface with the system over SSH/SFTP. For example, on Linux and Mac OS X:

Then, attach the bundle to your support ticket. If possible, use the SendSafely integration (as it allows for large file uploads).

Scrubbing Secrets

If you have extremely sensitive data in your Terraform build logs you may opt to omit these logs from your bundle. However, this may impede our efforts to diagnose any problems you are encountering. To create a custom support bundle, run the following commands:

```
sudo -s
hashicorp-support
cd /var/lib/hashicorp-support
tar -xzf hashicorp-support.tar.gz
rm hashicorp-support.tar.gz*
rm nomad/*build-worker*
tar -czf hashicorp-support.tar.gz *
gpg2 -e -r "Terraform Enterprise Support" \
--cipher-algo AES256 \
--compress-algo ZLIB \
-o hashicorp-support.tar.gz.enc \
hashicorp-support.tar.gz
```

You will note that we first create a support bundle using the normal procedure, extract it, remove the files we want to omit, and then create a new one.

Windows

On Microsoft Windows, tools such as PSCP (<https://www.ssh.com/ssh/putty/putty-manuals/0.68/Chapter5.html>) and WinSCP (<https://winscp.net/eng/index.php>) can be used to transfer the file.

About the Bundle

The support bundle contains logging and telemetry data from various components in Private Terraform Enterprise. It may also include log data from Terraform builds you have executed on your Private Terraform Enterprise installation.

Pre-Sales uploads

Customers in the pre-sales phase can upload support bundle files directly at <https://hashicorp.sendsafely.com/u/ptfe-support-bundles> (<https://hashicorp.sendsafely.com/u/ptfe-support-bundles>).

Private Terraform Enterprise Frequently Asked Questions

This page provides answers to many common questions about Private Terraform Enterprise.

General FAQ

1. Support
2. Managing Tool Versions
3. Migration from SaaS-based Terraform Enterprise
4. Outbound Access

AMI FAQ

Deprecation warning: The AMI will no longer be actively developed as of 201808-1 and will be fully decommissioned on November 30, 2018. Please see our Migration Guide (</docs/enterprise/private/migrate.html>) for instructions to migrate to the new Private Terraform Enterprise Installer.

1. About the AMI
2. AMI IDs
3. Additional Configuration Info
4. Upgrade Information
5. Required Network Access
6. Managing the Terraform State of the Install Process
7. Private Terraform Enterprise Architecture
8. Advanced Terraform
9. Rekeying the Vault instance used by Private Terraform Enterprise
10. Minimum Viable IAM Access Policy

Support for Private Terraform Enterprise

If some aspect of Private Terraform Enterprise is not working as expected, please reach out to support for help.

Email

You can engage HashiCorp support via the web portal at <https://support.hashicorp.com> (<https://support.hashicorp.com>) or by email at support@hashicorp.com (<mailto:support@hashicorp.com>). Please make sure to use your organization email (not your personal email) when contacting us so we can associate the support request with your organization and expedite our response.

Diagnostics

For most technical issues, HashiCorp support will ask you to include diagnostic information in your support request. To ensure the required information is included, PTFE has the ability to automatically generate a support bundle including logs and configuration details.

- AMI-based instance bundle generation (</docs/enterprise/private/diagnostics.html#ami-based-installs>)
- Installer-based instance bundle generation (</docs/enterprise/private/diagnostics.html#installer-based-instances>)

Managing Tool Versions

Terraform Enterprise has a control panel that allows admins to manage the versions of Terraform and Packer and their download locations.

In older Terraform Enterprise installations (prior to 201807-1) this control panel is available at the `/admin/tools` path or as a link in the sidebar from the general administrative interface at `/admin/manager`.

The screenshot shows the Terraform Enterprise Admin interface. On the left, there's a dark sidebar with a navigation menu:

- Admin (selected)
- Dashboard
- Search
- Users
- Organizations

Below this are sections for STATUS, MANAGER, and CONFIGURATION, each with its own set of links. The CONFIGURATION section has a 'Tools' item which is highlighted with a blue background.

The main content area has a blue header bar with the word 'tools'. Below the header, there's a green button labeled 'Add Tool Version'. The main body is a table for managing Packer versions:

Packer		
0.12.3	1 in use	Official
0.12.2	0 in use	Official
0.12.1	0 in use	Official
0.12.0	0 in use	Official
0.11.0	0 in use	Official
0.10.2	0 in use	Official
0.10.1	0 in use	Official
0.10.0	0 in use	Official

Here you'll find a list of Packer and Terraform versions as well as a link to add tool versions. If you click the `Edit` button on an individual tool version, you'll see that each version consists of:

- **Version Number** - will show up in dropdown lists for users to select
- **Download URL** - must point to a `linux-amd64` build of the tool

- **SHA256 Checksum Value** - must match the SHA256 checksum value of the download

The screenshot shows the HashiCorp Atlas Admin interface. On the left is a sidebar with sections: Admin, STATUS, MANAGER, and CONFIGURATION. Under Tools, there is a 'Tools' button. The main area is titled 'tools / terraform@0.9.4'. It shows a card for 'terraform@0.9.4 Official' with the URL https://releases.hashicorp.com/terraform/0.9.4/terraform_0.9.4_linux_amd64.zip. Below the URL is the SHA256 hash: cc1cffee3b82820b7f049bb290b841762ee920aef3cf4d95382cc7ea01135707. There are two checkboxes: 'Enabled' (checked) and 'Beta' (unchecked). A 'Save' button is at the bottom left, and a 'Delete' button is at the bottom right.

In Terraform Enterprise installations 201807-1 or later, see [Managing Tool Versions](#) (</docs/enterprise/private/admin/resources.html#managing-terraform-versions>).

Migrating from Terraform Enterprise SaaS

If you are already a user of the Terraform Enterprise SaaS (hereafter "the SaaS"), you may have Environments that you want to migrate over to your new Private Terraform Enterprise installation.

These instructions assume Terraform 0.9 or greater. See docs on legacy remote state (</docs/backends/legacy-0-8.html>) for information on upgrading usage of remote state in prior versions of Terraform.

Prerequisites

Have a Terraform Enterprise user API token ("Atlas token") handy for both Private Terraform Enterprise and the SaaS. The following examples will assume you have these stored in PTFE_ATLAS_TOKEN and SAAS_ATLAS_TOKEN, respectively.

Step 1: Connect local config to SaaS

Set up a local copy of your Terraform config that's connected to the SaaS via a backend block.

Assuming your environment is located at `my-organization/my-environment` in the SaaS, in a local copy of the Terraform config, ensure you have a backend configuration like this:

```
terraform {
  backend "atlas" {
    name = "my-organization/my-environment"
  }
}
```

Assign your SaaS API token to ATLAS_TOKEN and run `terraform init`:

```
export ATLAS_TOKEN=$SAAS_ATLAS_TOKEN
terraform init
```

Step 2: Copy state locally

This step fetches the latest copy of the state locally so it can be pushed to Private Terraform Enterprise. First, comment out the backend section of the config:

```
# Temporarily commented out to copy state locally
# terraform {
#   backend "atlas" {
#     name = "my-organization/my-environment"
#   }
# }
```

Then, re-run `terraform init`:

```
terraform init
```

This will cause Terraform to detect the change in backend and ask you if you want to copy the state.

Type yes to allow the state to be copied locally. Your state should now be present on disk as `terraform.tfstate`, ready to be uploaded to Private Terraform Enterprise.

Step 3: Update backend configuration for Private Terraform Enterprise

Change the backend config to point to the Private Terraform Enterprise installation:

```
terraform {
  backend "atlas" {
    address = "https://tfe.mycompany.example.com" # the address of your PTFE installation
    name    = "my-organization/my-environment"
  }
}
```

Assign your Private TFE API token to ATLAS_TOKEN and run `terraform init`:

```
export ATLAS_TOKEN=$PTFE_ATLAS_TOKEN
terraform init
```

You will again be asked if you want to copy the state file. Type yes and the state will be uploaded to your Private Terraform Enterprise installation.

Outbound Access

This is a list of hostnames that the product connects to and should be available for egress:

- releases.hashicorp.com
 - This egress is not required if a custom Terraform bundle (/docs/enterprise/run/index.html#custom-and-community-providers) is supplied.

Additionally these hostnames are accessed by the Installer product in online mode:

- get.replicated.com
 - api.replicated.com
 - registry.replicated.com
 - registry-data.replicated.com
 - quay.io
-

About the Private Terraform Enterprise AMI

This section contains information about the Terraform Enterprise AMI.

Operating System

The Private Terraform Enterprise AMI is based on the latest release of Ubuntu 16.04 with all security patches applied.

Network Ports

The Private Terraform Enterprise AMI requires that port :8080 be accessible. This is where all traffic from the ELB is routed. Many other internal Private Terraform services listen on the host, but they do not require external traffic. The AWS security group for the instance as well as software firewall rules within the runtime enforce this.

ulimits

The necessary limits on open file descriptors are raised within /etc/security/limits.d/nofile.conf on the machine image.

Critical Services

The Private Terraform Enterprise AMI contains dozens of services that are required for proper operation of Terraform Enterprise. These services are all configured to launch on boot. Application-level services are managed via Nomad and system-level automation is managed via systemd.

AMI IDs

For the most up-to-date list of AMI IDs, see the list maintained on our GitHub repo (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/docs/ami-ids.md>).

Additional Configuration Info

- [base-vpc](https://github.com/hashicorp/terraform-enterprise-modules/blob/master/aws-extra/base-vpc) (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/aws-extra/base-vpc>) - Configuration for creating a basic VPC and subnets that meet the documented requirements for Private Terraform Enterprise installation (/docs/enterprise/private/install-ami.html#preflight).
- [minimum-viable-iam](https://github.com/hashicorp/terraform-enterprise-modules/blob/master/aws-extra/minimum-viable-iam) (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/aws-extra/minimum-viable-iam>) - Configuration for creating an AWS user with a minimum access policy required to perform a Terraform Enterprise installation. Please note using the `AdministratorAccess` policy is recommended. This access is only required for the initial deployment.

Network Access

This section details the ingress and egress network access required by Terraform Enterprise to function properly.

Ingress Traffic

Terraform Enterprise requires certain ports to be accessible for it to function. The Terraform configuration that ships with Terraform Enterprise will by default create Security Groups (SGs) that make the appropriate ports available, but you can also specify custom SGs to be used instead.

Here are the two SGs in the system relevant for user access and the ports they require to be open:

- **Load Balancer SG:** Applied to the Elastic Load Balancer (ELB), controls incoming HTTP traffic from users
 - **Port 443** must be accessible to users for basic functionality, must also be accessible from the VPC itself, as certain internal services reach over the ELB to access cross-service APIs
 - **Port 80** is recommended to leave open for convenience - the system is set up to force SSL by redirecting users who visit Private Terraform Enterprise over HTTP to the HTTPS equivalent URL. If this port is not available, users who mistakenly visit the site over HTTP will see hanging requests in their browser
- **Instance SG:** Applied to the EC2 Instance running the application
 - **Port 8080** must be accessible to the ELB to serve traffic
 - **Port 22** must be accessible to operators to perform diagnostics and troubleshooting over SSH

There are also two internal SGs that are not currently user-configurable:

- **Database SG:** Applied to the RDS instance - allows the application to talk to PostgreSQL
- **Redis SG:** Applied to the ElastiCache instance - allows the application to talk to Redis

Egress Traffic

Terraform Enterprise makes several categories of outbound requests, detailed in the sections below.

Primary Data Stores

S3 is used for object storage, so access to the AWS S3 API and endpoints is required for basic functionality

RDS and ElastiCache instances are provisioned for application data storage. These instances are within the same VPC as the application, so communication with them does not constitute outbound traffic.

Version Control System Integrations

Private Terraform Enterprise can be configured to connect to a number of Version Control Systems (VCSs) (<https://www.terraform.io/docs/enterprise/vcs/index.html>), some supporting both SaaS and private-network installations.

In order to perform ingress of Terraform configuration from a configured VCS, Private Terraform Enterprise will need to be able to communicate with that provider's API, and webhooks from that provider will need to be able to reach Private Terraform Enterprise.

For example, an integration with GitHub.com will require Private Terraform Enterprise to have access to <https://github.com> (<https://github.com>) and for GitHub's webhooks to be able to route back to Terraform. Similarly, an integration with GitHub Enterprise will require Terraform to have access to the local GitHub instance.

Terraform Execution

As a part of its primary mode of operation, Terraform makes API calls out to infrastructure provider APIs. Since Private Terraform Enterprise runs Terraform on behalf of users, Private Terraform Enterprise will therefore need access to any Provider APIs that your colleagues want to manage with Private Terraform Enterprise.

Terraform Release Downloading

By default, Private Terraform Enterprise downloads the versions of Terraform that it executes from <https://releases.hashicorp.com/> (<https://releases.hashicorp.com/>). This behavior can be customized by specifying different download locations. See [Managing Tool Versions](#).

Terraform Latest Version Notifications

When displaying Terraform Runs, Private Terraform Enterprise has JavaScript that reaches out to <https://checkpoint-api.hashicorp.com> (<https://checkpoint-api.hashicorp.com>) to determine the latest released version of Terraform and notify users if there is a newer version available than the one they are running. This functionality is non-essential; new version notifications will not be displayed in the Web UI if checkpoint-api.hashicorp.com cannot be reached from a user's browser.

Communication Functions

- Private Terraform Enterprise uses the configured SMTP endpoint for sending emails
- Twilio can optionally be set up for SMS-based 2FA. (Virtual TOTP support is available separately, which does not

make external API calls.)

Storing Terraform Enterprise State

The Private Terraform Enterprise AMI install process uses Terraform, and therefore must store Terraform state. This presents a bootstrapping problem, because while generally you can use Terraform Enterprise to securely store versioned Terraform state, in this case Terraform Enterprise is not ready yet.

Therefore, you must choose a separate mechanism for storing the Terraform State produced by the install process.

Security Considerations for Terraform State

The Terraform state file for the Private Terraform Enterprise instance will contain the RDS Database password used by the application. While sensitive fields are separately encrypted-at-rest via Vault, this credential and network access to the database would yield access to all of the unencrypted metadata stored by Terraform Enterprise.

HashiCorp recommends storing the Terraform state for the install in an encrypted data store.

Recommended State Storage Setup

Terraform supports various remote state (<https://www.terraform.io/docs/state/remote.html>) backends that can be used to securely store the Terraform state produced by the install.

HashiCorp recommends a versioned, encrypted-at-rest S3 bucket as a good default choice.

Here are steps for setting up and using an S3 bucket for remote state storage:

```
# From the root dir of your Private Terraform Enterprise installation config
BUCKETNAME="mycompany-terraform-enterprise-state"

# Create bucket
aws s3 mb "s3://${BUCKETNAME}"

# Turn on versioning for the bucket
aws s3api put-bucket-versioning --bucket "${BUCKETNAME}" --versioning-configuration status=Enabled

# Configure terraform backend to point to the S3 bucket
cat <<EOF >backend.tf
terraform {
  backend "s3" {
    bucket  = "${BUCKETNAME}"
    key     = "terraform-enterprise.tfstate"
    encrypt = true
  }
}
EOF

# Initialize Terraform with the Remote Backend
terraform init
```

Now, if you keep the `backend.tf` file in scope when you run `terraform` operations, all state will be stored in the configured bucket.

Private Terraform Enterprise Architecture

This document describes aspects of the architecture of Private Terraform Enterprise as deployed via the AMI.

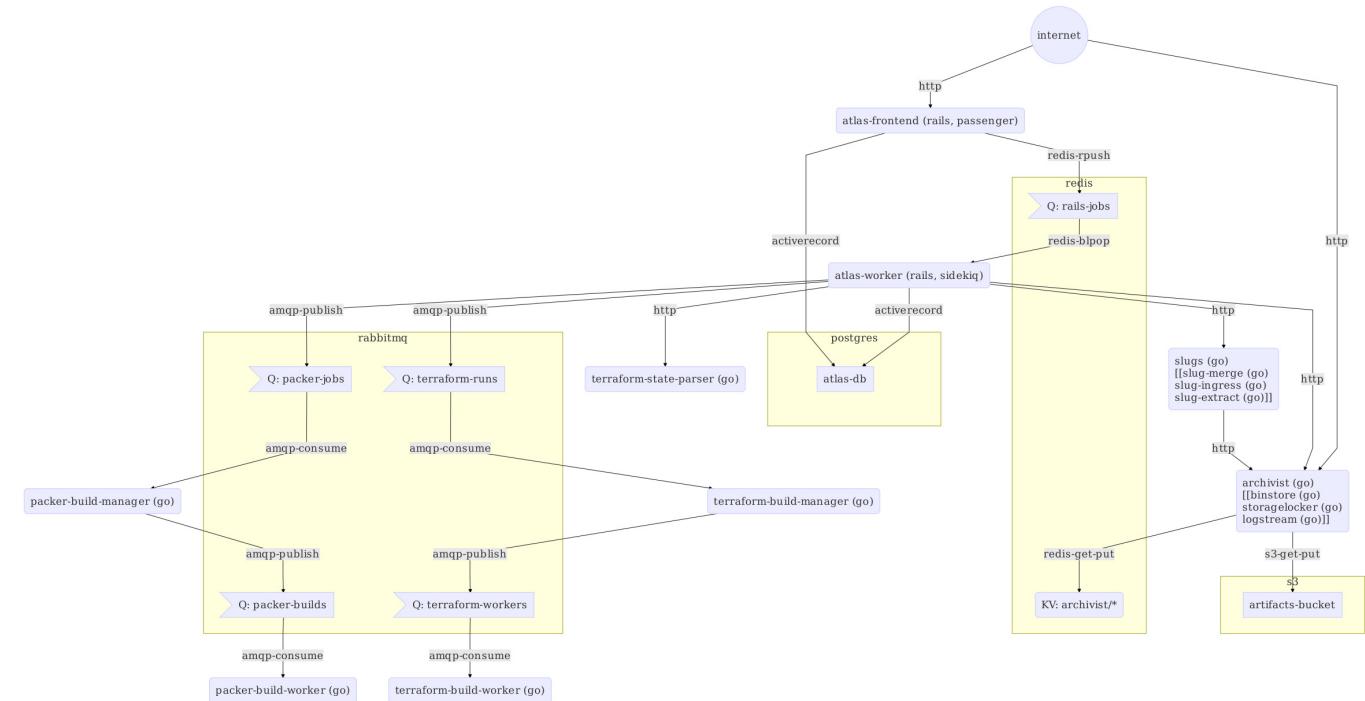
Services

These are the services used to run Private Terraform Enterprise. Each service contains a description of what actions it performs, a policy for restarts, impact of failing or degraded performance, and the service's dependencies.

- `atlas-frontend` and `atlas-worker` (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/docs/services/atlas.md>)
- `archivist`, `binstore`, `storagelocker`, and `logstream` (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/docs/services/archivist.md>)
- `terraform-build-manager`, and `terraform-build-worker` (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/docs/services/build-pipeline.md>)
- `slug-extract`, `slug-ingress`, `slug-merge` (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/docs/services/slugs.md>)

Data Flow Diagram

The following diagram shows the way data flows through the various services and data stores in Terraform Enterprise.



(Note: The services in double square brackets are soon to be replaced by the service that precedes them.)

Advanced Terraform

The aws-standard Terraform module can be used as a true Terraform module to enable some additional features to configure the cluster.

Additional IAM Role policies

The module outputs the role name used by the instance, allowing you to attach additional policies to configure access:

```
provider "aws" {
  region = "us-west-2"
}

module "standard" {
  source = "../../terraform/aws-standard"
  # Variables that would be in terraform.tfvars go here
}

data "aws_iam_policy_document" "extra-s3-perms" {
  statement {
    sid      = "AllowS3Access"
    effect   = "Allow"

    resources = [
      "arn:aws:s3:::my-private-artifacts/*",
      "arn:aws:s3:::my-private-artifacts",
    ]
  }

  actions = [
    "s3:*",
  ]
}
}

resource "aws_iam_role_policy" "extra-s3-perms" {
  role    = "${module.standard.iam_role}"
  policy  = "${data.aws_iam_policy_document.extra-s3-perms.json}"
}
```

Rekeying Vault

(Requires an AMI version 201709-1 or later)

The Vault instance used by Private Terraform Enterprise self-manages its unseal key by default. This unseal key is stored in a KMS-encrypted file on S3 and is downloaded by the instance on boot to automatically unseal Vault.

If the above configuration is insufficient for your security needs, you can choose to rekey the Vault instance after bootstrapping is completed. This allows you to change the key shares and key threshold settings, places the Vault unseal keys under your control, and deactivates the auto-unseal behavior of the Private Terraform Enterprise instance.

The Vault documentation has a guide (<https://www.vaultproject.io/guides/rekeying-and-rotating.html#rekeying-vault>) on how to perform a rekey operation and `vault rekey -help` output provides full docs on the various options available.

Walkthrough of Rekey Operation

Here is an example of rekeying the Private Terraform Enterprise vault to use 5 key shares with a key threshold of 2. These commands are executed from an SSH session on the Private Terraform Enterprise instance as the tfe-admin user.

```
vault rekey -init -key-shares=5 -key-threshold=2
```

WARNING: If you lose the keys after they are returned to you, there is no recovery. Consider using the '-pgp-keys' option to protect the returned unseal keys along with '-backup=true' to allow recovery of the encrypted keys in case of emergency. They can easily be deleted at a later time with 'vault rekey -delete'.

```
Nonce: acdd8a46-3b...
Started: true
Key Shares: 5
Key Threshold: 2
Rekey Progress: 0
Required Keys: 1
```

The rekey operation has now been started. The printed nonce and the current unseal key are required to complete it.

The current unseal key can be found under /data/vault-unseal-key

```
VAULT_UNSEAL_KEY=$(sudo cat /data/vault-unseal-key)
vault rekey -nonce=acdd8a46-3b... $VAULT_UNSEAL_KEY
```

```
Key 1: jcLit6uk...
Key 2: qi/AfO30...
Key 3: t3TezCbE...
Key 4: 506E8WFU...
Key 5: +bWaQapk...
```

```
Operation nonce: acdd8a46-3b2a-840e-0db8-e53e84fa7e64
```

Vault rekeyed with 5 keys and a key threshold of 2. Please securely distribute the above keys. When the Vault is re-sealed, restarted, or stopped, you must provide at least 2 of these keys to unseal it again.

Vault does not store the master key. Without at least 2 keys, your Vault will remain permanently sealed.

IMPORTANT: After Rekeying

Note: After performing a rekey it's important to remove the old unseal key and trigger a backup before rebooting the machine. This will ensure that Private Terraform Enterprise knows to prompt for Vault unseal keys.

```
sudo rm /data/vault-unseal-key
sudo atlas-backup
```

Minimum Viable IAM Access Policy

HashiCorp does not recommend using the minimum viable IAM access policy. You will need to edit this policy to account for any and all actions you wish Terraform to be able to perform against your AWS account. We do, however, provide a git repo

(<https://github.com/hashicorp/terraform-enterprise-modules/tree/master/aws-extra/minimum-viable-iam>) you can use as a base example to start from.

Private Terraform Enterprise GCP Reference Architecture

This document provides recommended practices and a reference architecture for HashiCorp Private Terraform Enterprise (PTFE) implementations on GCP.

Required Reading

Prior to making hardware sizing and architectural decisions, read through the installation information available for PTFE (<https://www.terraform.io/docs/enterprise/private/install-installer.html>) to familiarise yourself with the application components and architecture. Further, read the reliability and availability guidance (<https://www.terraform.io/docs/enterprise/private/reliability-availability.html>) as a primer to understanding the recommendations in this reference architecture.

Infrastructure Requirements

Note: This reference architecture focuses on the *Production - External Services* operational mode.

Depending on the chosen operational mode (<https://www.terraform.io/docs/enterprise/private/preflight-installer.html#operational-mode-decision>), the infrastructure requirements for PTFE range from a single Cloud Compute VM instance for demo installations to multiple instances connected to Cloud SQL and Cloud Storage for a stateless production installation.

The following table provides high-level server guidelines. Of particular note is the strong recommendation to avoid non-fixed performance CPUs, or "Shared-core machine types" in GCP terms, such as f1-series and g1-series instances.

PTFE Server (Compute Engine VM via Regional Managed Instance Group)

Type	CPU	Memory	Disk	GCP Machine Types
Minimum	2-4 core	8-16 GB RAM	50GB/200GB*	n1-standard-2, n1-standard-4
Recommended	4-8 core	16-32 GB RAM	50GB/200GB*	n1-standard-4, n1-standard-8

Hardware Sizing Considerations

- *PTFE requires 50GB for installation, but GCP documentation for storage performance (<https://cloud.google.com/compute/docs/disks/#performance>) recommends "to ensure consistent performance for more general use of the boot device, use either an SSD persistent disk as your boot disk or use a standard persistent disk that is at least 200 GB in size."
- The minimum size would be appropriate for most initial production deployments, or for development/testing environments.
- The recommended size is for production environments where there is a consistent high workload in the form of concurrent terraform runs.

PostgreSQL Database (Cloud SQL PostgreSQL Production)

Type	CPU	Memory	Storage	GCP Machine Types
Minimum	2 core	8 GB RAM	50GB	Custom PostgreSQL Production
Recommended	4-8 core	16-32 GB RAM	50GB	Custom PostgreSQL Production

Hardware Sizing Considerations

- The minimum size would be appropriate for most initial production deployments, or for development/testing environments.
- The recommended size is for production environments where there is a consistent high workload in the form of concurrent terraform runs.

Object Storage (Cloud Storage)

A Regional Cloud Storage (<https://cloud.google.com/storage/docs/storage-classes#regional>) bucket must be specified during the PTFE installation for application data to be stored securely and redundantly away from the Compute Engine VMs running the PTFE application. This Cloud Storage bucket must be in the same region as the Compute Engine and Cloud SQL instances. Vault is used to encrypt all application data stored in the Cloud Storage bucket. This allows for further server-side encryption (<https://cloud.google.com/storage/docs/encryption/>). by Cloud Storage.

Other Considerations

Additional GCP Resources

In order to successfully provision this reference architecture you must also be permitted to create the following GCP resources:

- Project (<https://cloud.google.com/resource-manager/docs/creating-managing-projects>)
- VPC Network (<https://cloud.google.com/vpc/docs/vpc>)
- Subnet (<https://cloud.google.com/vpc/docs/using-vpc>)
- Firewall (<https://cloud.google.com/vpc/docs/firewalls>)
- Target Pool (<https://cloud.google.com/load-balancing/docs/target-pools>)
- Forwarding Rule (<https://cloud.google.com/load-balancing/docs/forwarding-rules>)
- Compute Instance Template (<https://cloud.google.com/compute/docs/instance-templates/>)
- Regional Managed Instance Group (<https://cloud.google.com/compute/docs/instance-groups/distributing-instances-with-regional-instance-groups>)
- Cloud DNS (optional) (<https://cloud.google.com/dns/>)

Network

To deploy PTFE in GCP you will need to create new or use existing networking infrastructure. The below infrastructure diagram highlights some of the key components (network, subnets) and you will also have firewall and gateway requirements. These elements are likely to be very unique to your environment and not something this Reference Architecture can specify in detail.

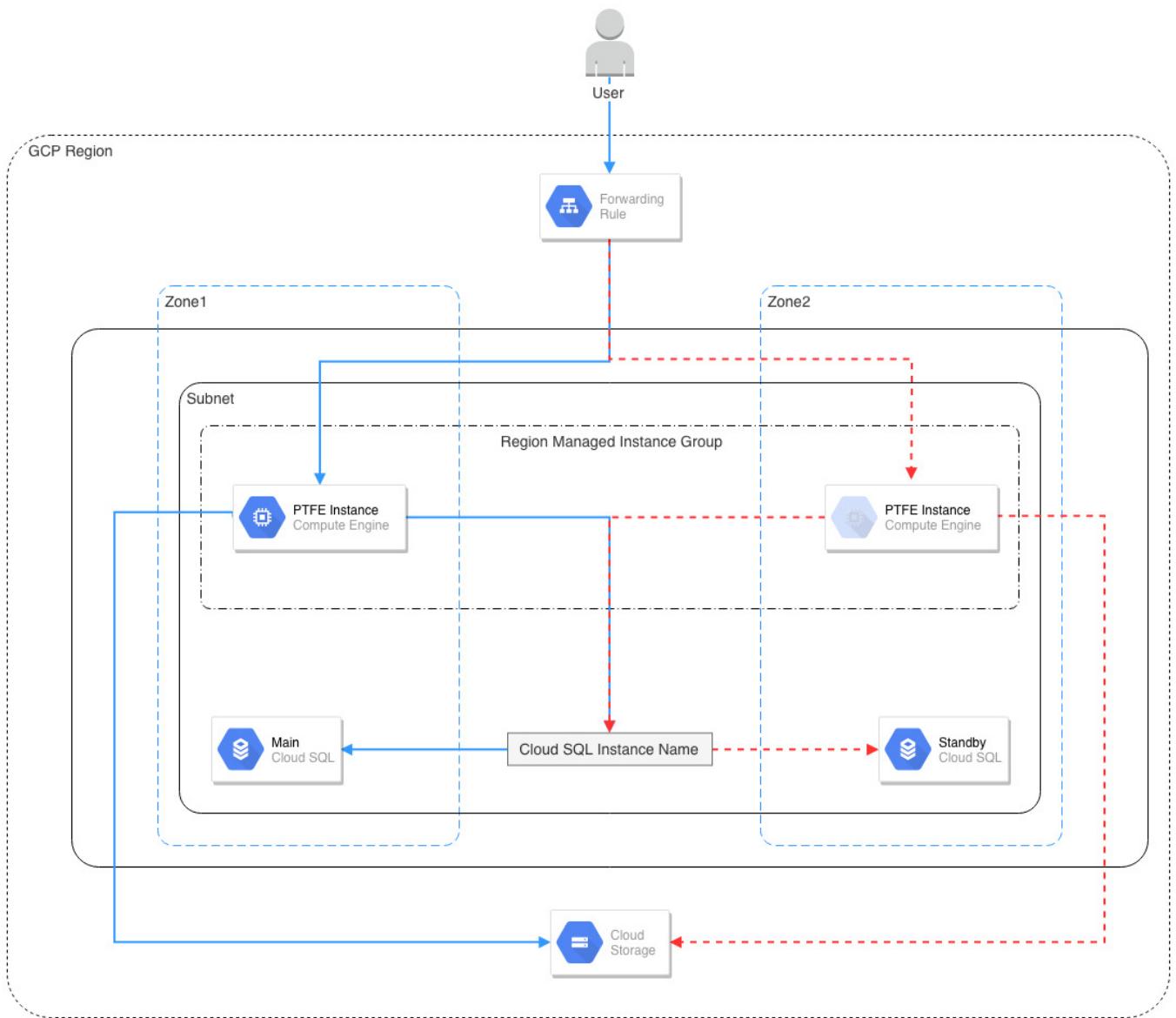
DNS

DNS can be configured external to GCP or using Cloud DNS (<https://cloud.google.com/dns/>). The fully qualified domain name should resolve to the Forwarding Rules associated with the PTFE deployment. Creating the required DNS entry is outside the scope of this guide.

SSL/TLS Certificates and Load Balancers

An SSL/TLS certificate signed by a public or private CA is required for secure communication between clients, VCS systems, and the PTFE application server. The certificate can be specified during the UI-based installation or in a configuration file used for an unattended installation.

Infrastructure Diagram



The above diagram shows the infrastructure components at a high-level.

Application Layer

The Application Layer is composed of a Regional Managed Instance Group and an Instance Template providing an auto-recovery mechanism in the event of an instance or Zone failure.

Storage Layer

The Storage Layer is composed of multiple service endpoints (Cloud SQL, Cloud Storage) all configured with or benefiting from inherent resiliency provided by GCP.

Additional Information

- Cloud SQL high-availability (<https://cloud.google.com/sql/docs/postgres/high-availability>).
- Regional Cloud Storage (<https://cloud.google.com/storage/docs/storage-classes>).

Infrastructure Provisioning

The recommended way to deploy PTFE is through use of a Terraform configuration that defines the required resources, their references to other resources, and dependencies.

Normal Operation

Component Interaction

The Forwarding Rule routes all traffic to the *PTFE* instance, which is managed by a Regional Managed Instance Group with maximum and minimum instance counts set to one.

The PTFE application is connected to the PostgreSQL database via the Cloud SQL endpoint and all database requests are routed via the Cloud SQL endpoint to the database instance.

The PTFE application is connected to object storage via the Cloud Storage endpoint for the defined bucket and all object storage requests are routed to the highly available infrastructure supporting Cloud Storage.

Upgrades

See the Upgrades section (</docs/enterprise/private/upgrades.html>) of the documentation.

High Availability

Failure Scenarios

GCP provides guidance on designing robust systems (<https://cloud.google.com/compute/docs/tutorials/robustsystems>). Working in accordance with those recommendations the PTFE Reference Architecture is designed to handle different failure scenarios with different probabilities. As the architecture evolves it may provide a higher level of service continuity.

Single Compute Engine Instance Failure

In the event of the *PTFE* instance failing in a way that GCP can observe, Live Migration (<https://cloud.google.com/compute/docs/instances/live-migration>) is used to move the instance to new physical hardware automatically. In the event that Live Migration is not possible the instance will crash and be restarted on new physical hardware automatically. During instance startup the PTFE services will be started and service will resume.

Zone Failure

In the event of the Zone hosting the main instances (Compute Engine and Cloud SQL) failing, the Regional Managed Instance Group for the Compute VMs will automatically begin booting a new one in an operational Zone.

- Cloud SQL automatically and transparently fails over to the standby zone. The GCP documentation provides more detail (<https://cloud.google.com/sql/docs/postgres/high-availability>) on the exact behaviour and expected impact.
- Cloud Storage is resilient to Zone failure based on its architecture.

See below for more detail on how each component handles Zone failure.

PTFE Server

By utilizing a Regional Managed Instance Group, the PTFE instance automatically recovers in the event of any outage except for the loss of an entire region.

With external services (PostgreSQL Database, Object Storage) in use, there is still some application configuration data present on the PTFE server such as installation type, database connection settings, hostname. This data rarely changes. If the configuration on PTFE changes you should update the Instance Template to include this updated configuration so that any newly launched Compute VM uses this new configuration.

PostgreSQL Database

Using Cloud SQL as an external database service leverages the highly available infrastructure provided by GCP. From the GCP website:

A Cloud SQL instance configured for high availability is also called a regional instance. A regional instance is located in two zones within the configured region, so if it cannot serve data from its primary zone, it fails over and continues to serve data from its secondary zone. (source (<https://cloud.google.com/sql/docs/postgres/high-availability>))

Object Storage

Using Regional Cloud Storage as an external object store leverages the highly available infrastructure provided by GCP. Regional Cloud Storage buckets are resilient to Zone failure within the region selected during bucket creation. From the GCP website:

Regional Storage is appropriate for storing data in the same regional location as Compute Engine instances or Kubernetes Engine clusters that use the data. Doing so gives you better performance for data-intensive computations, as opposed to storing your data in a multi-regional location. (source (<https://cloud.google.com/storage/docs/storage-classes>))

Disaster Recovery

Failure Scenarios

GCP provides guidance on designing robust systems (<https://cloud.google.com/compute/docs/tutorials/robustsystems>). Working in accordance with those recommendations the PTFE Reference Architecture is designed to handle different failure scenarios that have different probabilities. As the architecture evolves it may provide a higher level of service continuity.

Region Failure

PTFE is currently architected to provide high availability within a single GCP Region. Using multiple GCP Regions will give you greater control over your recovery time in the event of a hard dependency failure on a regional GCP service. In this section, we'll discuss various implementation patterns and their typical availability.

An identical infrastructure should be provisioned in a secondary GCP Region. Depending on recovery time objectives and tolerances for additional cost to support GCP Region failure, the infrastructure can be running (Warm Standby) or stopped (Cold Standby). In the event of the primary GCP Region hosting the PTFE application failing, the secondary GCP Region will require some configuration before traffic is directed to it along with some global services such as DNS.

- Cloud SQL cross-region read replicas (<https://cloud.google.com/sql/docs/postgres/replication/manage-replicas>) can be used in a warm standby architecture or Cloud SQL database backups (<https://cloud.google.com/sql/docs/postgres/backup-recovery/restoring>) can be used in a cold standby architecture.
- Multi-Regional Cloud Storage replication (<https://cloud.google.com/storage/docs/storage-classes#multi-regional>) must be configured so the object storage component of the Storage Layer is available in multiple GCP Regions.
- DNS must be redirected to the Forwarding Rule acting as the entry point for the infrastructure deployed in the secondary GCP Region.

Data Corruption

The PTFE application architecture relies on multiple service endpoints (Cloud SQL, Cloud Storage) all providing their own backup and recovery functionality to support a low MTTR in the event of data corruption.

PTFE Servers

With external services (PostgreSQL Database, Object Storage) in use, there is still some application configuration data present on the PTFE server such as installation type, database connection settings, hostname. This data rarely changes. We recommend configuring automated snapshots (<https://www.terraform.io/docs/enterprise/private/automated-recovery.html#1-configure-snapshots>) for this installation data so it can be recovered in the event of data corruption.

PostgreSQL Database

Backup and restoration of PostgreSQL is managed by GCP and configured through the GCP management console or CLI.

Automated (scheduled) and on-demand backups are available in GCP. Review the backup (<https://cloud.google.com/sql/docs/postgres/backup-recovery/backups>) and restoration (<https://cloud.google.com/sql/docs/postgres/backup-recovery/restore>) documentation for further guidance.

More details of Cloud SQL (PostgreSQL) features are available here (<https://cloud.google.com/sql/docs/postgres/>).

Object Storage

There is no automatic backup/snapshot of Cloud Storage by GCP, so it is recommended to script a bucket copy process from the bucket used by the PTFE application to a “backup bucket” in Cloud Storage that runs at regular intervals. The Nearline Storage (<https://cloud.google.com/storage/docs/storage-classes#nearline>) storage class is identified as a solution targeted more for DR backups. From the GCP website:

Nearline Storage is ideal for data you plan to read or modify on average once a month or less. For example, if you want to continuously add files to Cloud Storage and plan to access those files once a month for analysis, Nearline Storage is a great choice. Nearline Storage is also appropriate for data backup, disaster recovery, and archival storage. (source (<https://cloud.google.com/storage/docs/storage-classes#nearline>))

Private Terraform Enterprise Installation (AMI)

Deprecation warning: The AMI will no longer be actively developed as of 201808-1 and will be fully decommissioned on November 30, 2018. Please see our Migration Guide (</docs/enterprise/private/migrate.html>) for instructions to migrate to the new Private Terraform Enterprise Installer.

Delivery

The goal of this installation procedure is to set up a Terraform Enterprise cluster that is available on a DNS name that is accessed via HTTPS. This standard configuration package uses Terraform to create both the compute and data layer resources, and optionally uses Route53 to configure the DNS automatically.

Note: This document applies to the standard AMI-based deliverable of Private Terraform Enterprise. If you are using the Installer, please see the instructions for the Installer (</docs/enterprise/private/install-installer.html>).

Preflight

Dependencies

Before setup begins, a few resources need to be provisioned. We consider these out of scope for the cluster provisioning because they depend on the user's environment.

The following are **required** to complete installation:

- **AWS IAM credentials** capable of creating new IAM roles configuring various services. We strongly suggest you use the AdministratorAccess policy for this. The credentials are only used for setup; during runtime only an assumed role is used.
- **AWS VPC** containing at least 2 subnets. These will be used to launch the cluster into. Subnets do not need to be public, but they do need an internet gateway at present. If two private subnets are used they should each be in a separate Availability Zone (AZ). Also, if a third public subnet is used, and the instance resides in a private subnet, the public subnet must be in the same AZ as the instance.
- **SSH Key Pair** configured with AWS EC2. This will be used to configure support access to the cluster. This SSH key can be optionally removed from the instance once installation is complete.
 - To create a new one, see Amazon's docs for EC2 key pairs.
(<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html>)
- A **Publicly Trusted TLS certificate** registered with AWS Certificate Manager. This can be one created by ACM for a hostname or the certificate can be imported into it.
 - To create a new ACM-managed cert: <https://console.aws.amazon.com/acm/home#/wizard/>
(<https://console.aws.amazon.com/acm/home#/wizard/>)
 - To import an existing cert: <https://console.aws.amazon.com/acm/home#/importwizard/>
(<https://console.aws.amazon.com/acm/home#/importwizard/>)

- *NOTE:* Certificates are per region, so be sure to create them in the same region as you'll be deploying Terraform Enterprise into.
- *NOTE:* The certificate must allow the fully qualified hostname that Terraform Enterprise will be using. This means you need to decide on a final hostname when creating the certificates and use the same value in the configuration.

Clone the Installer Git Repo

Private Terraform Enterprise installs via Terraform. It is expected you have downloaded the Terraform Open Source binary to the machine where you will be doing the install from. To get started you will also need to clone a repo that contains the necessary modules and variable files.

- Open a terminal on a machine that has full network access to the AWS account where Private Terraform Enterprise will be deployed. Enter the following to clone the repo

```
git clone git@github.com:hashicorp/terraform-enterprise-modules.git
```

- Once the clone is complete cd into the `terraform-enterprise-modules/aws-standard/` directory.

The following details will be requested during the application bootstrapping process. It's helpful to have them prepared beforehand.

- **SMTP Credentials:** Terraform Enterprise requires SMTP information to send email. SMTP configuration can be skipped if necessary during the installation, but HashiCorp recommends configuring it during the initial bootstrapping process.
 - Hostname and port of SMTP service
 - Type of authentication (plain or login)
 - Username and password
 - Email address to use as the sender for emails originating from Terraform Enterprise
- (Optional) **Twilio Credentials:** Terraform Enterprise can use Twilio for SMS-based 2-factor authentication. If Twilio is not configured, virtual MFA device-based 2FA (e.g. Google Authenticator) will still be available.

Required Variables

The following Terraform variables are required inputs and must be populated prior to beginning installation.

The values for these variables should be placed in the `terraform.tfvars` file. Copy the `terraform.tfvars.example` in the `aws-standard` directory to `terraform.tfvars`, then edit it with the proper values. There are notes in the file regarding each variable, but please use the following list for more information:

NOTE: Use only alphanumeric characters (upper- and lower-case), as well as dashes, underscores, colons, and forward-slashes (-, _, :, /). Other characters may cause the TFE instance to be unable to boot.

- `region`: The AWS region to deploy into.
- `ami_id`: The ID of a Terraform Enterprise Base AMI. See `ami-ids` (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/docs/ami-ids.md>) to look one up.

- `fqdn`: The hostname that the cluster will be accessible at. This value needs to match the DNS setup for proper operations. Example: `tfe-eng01.mycompany.io`
- `cert_id`: An AWS certificate ARN. This is the certification that will be used by the ELB for the cluster. Example: `arn:aws:acm:us-west-2:241656615859:certificate/f32fa674-de62-4681-8035-21a4c81474c6`
- `instance_subnet_id`: Subnet ID of the subnet that the cluster's instance will be placed into. If this is a public subnet, the instance will be assigned a public IP. This is not required as the primary cluster interface is an ELB registered with the hostname. Example: `subnet-0de26b6a`
- `elb_subnet_id`: Subnet ID of the subnet that the cluster's load balancer will be placed into. If this is a public subnet, the load balancer will be accessible from the public internet. This is not required — the ELB can be marked as private via the `internal_elb` option below.
- `data_subnet_ids`: Subnet IDs that will be used to create the data services (RDS and ElastiCache) used by the cluster. There must be 2 subnet IDs given for proper redundancy. Example: `["subnet-0ce26b6b", "subnet-d0f35099"]`
- `db_password`: Password that will be used to access RDS. Example: `databaseshavesecrets`
- `bucket_name`: Name of the S3 bucket to store artifacts used by the cluster into. This bucket is automatically created. We suggest you name it `tfe-<HOSTNAME>-data`, as convention.

Optional Variables

These variables can be populated, but they have defaults that will be used if you omit them. As with the required variables, you can place these values in the `terraform.tfvars` file.

NOTE: Use only alphanumeric characters (upper- and lower-case), as well as dashes, underscores, colons, and forward-slashes (-, _, :, /). Other characters may cause the TFE instance to be unable to boot.

- `key_name`: Name of AWS SSH Key Pair that will be used (as shown in the AWS console). The pair needs to already exist, it will not be created. **If this variable is not set, no SSH access will be available to the Terraform Enterprise instance.**
- `manage_bucket` Indicate if this Terraform state should create and own the bucket. Set this to false if you are reusing an existing bucket.
- `kms_key_id` Specify the ARN for a KMS key to use rather than having one created automatically.
- `db_username` Username that will be used to access RDS. Default: `atlas`
- `db_size_gb` Disk size of the RDS instance to create. Default: `80`
- `db_instance_class` Instance type of the RDS instance to create. Default: `db.m4.large`
- `db_multi_az` Configure if the RDS cluster should use multiple AZs to improve snapshot performance. Default: `true`
- `db_snapshot_identifier` Previously made snapshot to restore when RDS is created. This is for migration of data between clusters. Default is to create the database fresh.
- `db_name` This only needs to be set if you're migrating from an RDS instance with a different database name.
- `zone_id` The ID of a Route53 zone that a record for the cluster will be installed into. Leave this blank if you need to manage DNS elsewhere. Example: `ZVEF52R7NLTW6`
- `hostname` If specifying `zone_id`, this should be set to the name that is used for the record to be registered with the

zone. This value combined with the zone information will form the full DNS name for Terraform Enterprise. Example: emp-test

- `arn_partition` Used mostly for govcloud installations. Example: `aws-us-gov`
- `internal_elb` Indicate to AWS that the created ELB is internal only. Example: `true`
- `startup_script` Shell code that should run on the first boot. This is explained in more detail below.
- `external_security_group_ids` The IDs of existing EC2 Security Groups to assign to the ELB for "external" access to the system. By default, a Security Group will be created that allows ingress to ports 80 and 443 from `0.0.0.0/0`.
- `internal_security_group_ids` The IDs of existing EC2 Security Groups to assign to the instance for "internal" access to the system. By default, a Security group will be created that allows ingress to ports 22 and 8080 from `0.0.0.0/0`.
- `proxy_url` A url (http or https, with port) to proxy all external http/https request from the cluster to. This is explained in more detail below.
- `no_proxy` Hosts to exclude from proxying, in addition to the default set. (Only applies when `proxy_url` is set.)
- `local_redis` If true, use a local Redis server on the cluster instance, eliminating the need for ElasticCache. Default: `false`
- `local_setup` If true, write the setup data to a local file called `tfe-setup-data` instead of into S3. The instance will prompt for this setup data on its first boot, after which point it will be stored in Vault. (Requires a release v201709-1 or later to be set to true.) Default: `false`
- `ebs_size` The size (in GB) to configure the EBS volumes used to store redis data. Default: 100
- `ebs_redundancy` The number of EBS volumes to mirror together for redundancy in storing redis data. Default: 2
- `archivist_sse` Setting for server-side encryption of objects in S3; if provided, *must* be set to `aws:kms`. Default: ``
- `archivist_kms_key_id` KMS key ID (full ARN) for server-side encryption of objects stored in S3.

Startup Script (Optional)

The `startup_script` variable can contain any shell code and will be executed on the first boot. This mechanism can be used to customize the AMI, adding additional software or configuration.

For example, to install a custom SSL certificate for the services to trust:

```
curl -o /usr/local/share/ca-certificates/cert.crt https://my.server.net/custom.pem  
update-ca-certificates
```

Be sure that files in `/usr/local/share/ca-certificates` end in `.crt` and that `update-ca-certificates` is run after they're placed.

Or to install additional Ubuntu packages:

```
apt-get install -y emacs
```

Because the content is likely to be multiple lines, we suggest you use the heredoc style syntax to define the variable. For example, in your `terraform.tfvars` file, you'd have:

```
startup_script = <<SHELL
apt-get install -y nano
adduser my-admin

SHELL
```

Proxy Support (Optional)

The cluster can be configured to send all outbound HTTP and HTTPS traffic through a proxy. By setting the `proxy_url` to either an `http://` or `https://` url, all systems that make HTTP and HTTPS request will connect to the proxy to perform the request.

A comma-separated list of hosts to be excluded from proxying can be specified via the `no_proxy` variable.

Note: This is only for outbound HTTP and HTTPS requests. Other traffic such as SMTP and NTP are not proxied and will attempt to connect directly.

Planning

Terraform Enterprise uses Terraform itself for deployment. Once you have filled in the `terraform.tfvars` file, simply run: `terraform init` then `terraform plan`. This will output the manifest of all the resources that will be created.

Deployment

Once you've reviewed the plan output and are ready to deploy Terraform Enterprise, run `terraform apply`. This will take anywhere from 10 to 20 minutes (mostly due to RDS creation time).

Upgrade

To upgrade your instance of Terraform Enterprise, update the repository containing the terraform configuration, run `terraform plan`, and run `terraform apply`.

Outputs

- `dns_name` - The DNS name of the load balancer for Terraform Enterprise. If you are managing DNS separately, you'll need to make a CNAME record from your indicated hostname to this value.
- `zone_id` - The Route53 Zone ID of the load balancer for Terraform Enterprise. If you are managing DNS separately but still using Route53, this value may be useful.
- `url` - The URL where Terraform Enterprise will become available when it boots.

Configuration

After completing a new install you should head to the configuration page (</docs/enterprise/private/config.html>) to create users and teams.

Private Terraform Enterprise Installation (Installer)

Delivery

This document outlines the procedure for using the Private Terraform Enterprise (PTFE) installer to set up Terraform Enterprise on a customer-controlled machine.

Note: This document is only meant for those customers using Private Terraform Enterprise via the Installer. Customers using the AMI can follow the instructions for the AMI-based install (</docs/enterprise/private/install-ami.html>).

Migrating from AMI

Deprecation warning: The AMI will no longer be actively developed as of 201808-1 and will be fully decommissioned on November 30, 2018.

If you are migrating an installation from the AMI to the installer, please use the instructions in the migration guide (</docs/enterprise/private/migrate.html>).

Preflight

Before you begin, consult Preflight (</docs/enterprise/private/preflight-installer.html>) for pre-requisites. You'll need to prepare data files and a Linux instance.

Proxy Usage

If your installation requires using a proxy server, you will be asked for the proxy server information when you first run the installer via ssh. This proxy server will be used for all outbound HTTP and HTTPS connections.

Optionally, if you're running the installer script in an automated manner, you can pass a `http-proxy` flag to set the address of the proxy. For example:

```
./install.sh http-proxy=http://internal.mycompany.com:8080
```

Proxy Exclusions (NO_PROXY)

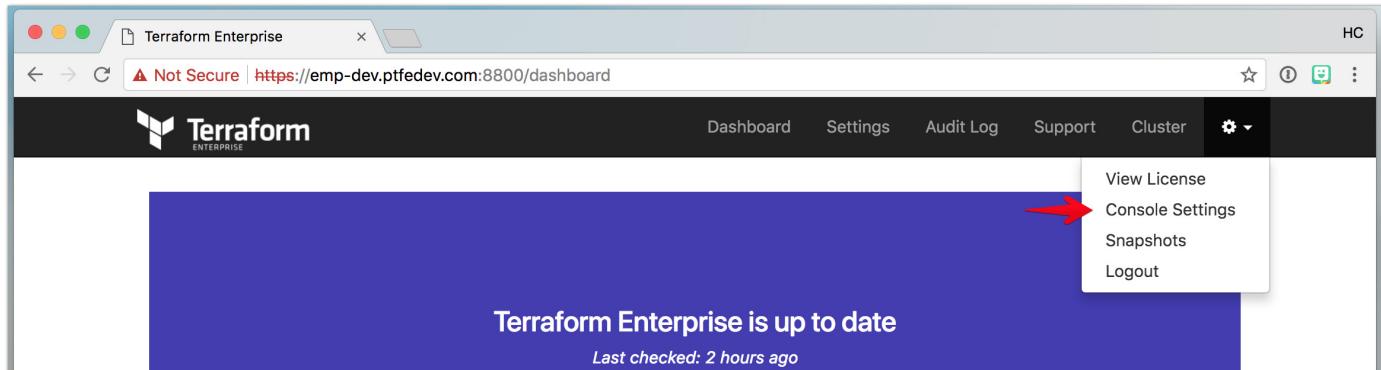
If certain hostnames should not use the proxy and the instance should connect directly to them (for instance, for S3), then you can pass an additional option to provide a list of domains:

```
./install.sh additional-no-proxy=s3.amazonaws.com,internal-vcs.mycompany.com,example.com
```

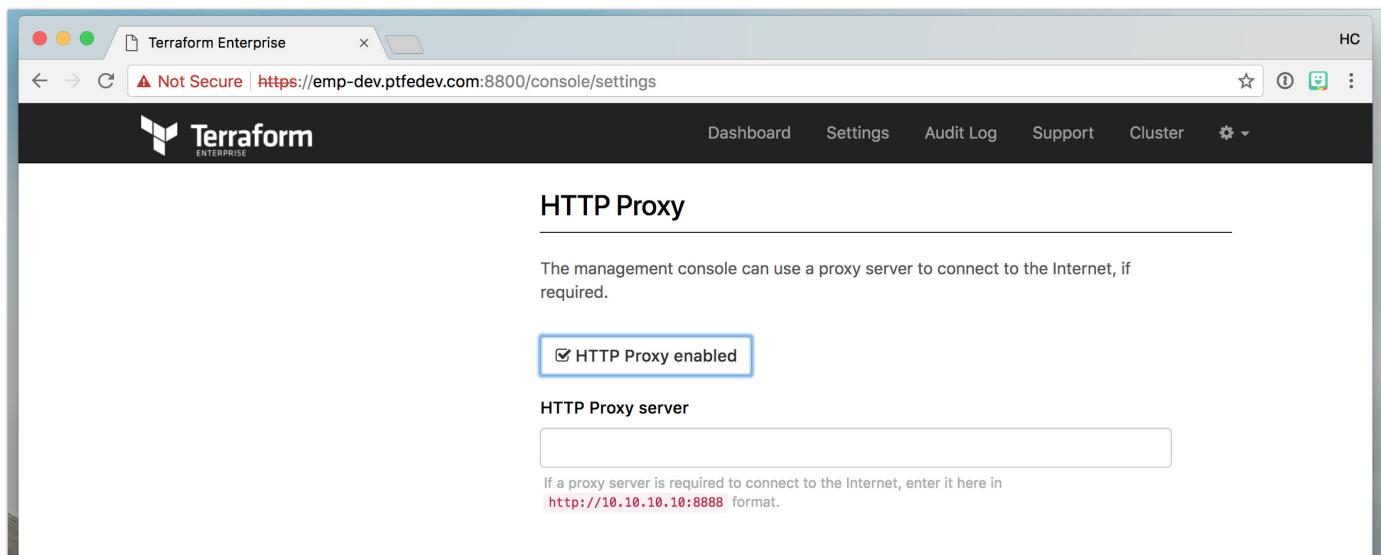
Passing this option to the installation script is particularly useful if the hostnames that should not use the proxy include services that the instance needs to be able to reach during installation, such as S3. Alternately, if the only hosts you need to add are those that are not used during installation, such as a private VCS instance, you can provide these hosts after initial installation is complete, in the installer settings (available on port 8800 under /console/settings).

Reconfiguring the Proxy

To change the proxy settings after installation, use the Console settings page, accessed from the dashboard on port 8800 under /console/settings.



On the Console Settings page, there is a section for HTTP Proxy:



Trusting SSL/TLS Certificates

There are two primary areas for SSL configuration in the installer.

TLS Key & Cert

The TLS Key & Cert field (found in the console settings after initial installation) should contain PTFE's own key and certificate, or key and certificate chain. A chain would be used in this field if the CA indicates an intermediate certificate is required as well.

Certificate Authority (CA) Bundle

PTFE needs to be able to access all services that it integrates with, such as VCS providers or database servers. Because it typically accesses them via SSL/TLS, it is critical that the certificates used by any service that PTFE integrates with are trusted by PTFE.

This section is used to allow PTFE to connect to services that use SSL/TLS certificates issued by private CAs. It allows multiple certificates to be specified as trusted, and should contain all certificates that PTFE should trust when presented with them from itself or another application.

A collection of certificates for trusted issuers is known as a Certificate Authority (CA) Bundle. All certificates in the certificate signing chain, meaning the root certificate and any intermediate certificates, must be included here. These multiple certificates are listed one after another in text format.

Note: If PTFE is configured with a SSL key and certificate issued against a private CA, the certificate chain for that CA must be included here as well. This allows the instance to query itself.

Certificates must be formatted using PEM encoding, that is, as text. For example:

```
-----BEGIN CERTIFICATE-----
MIIFtTCCA52gAwIBAgIIYY3HhjsBggUwDQYJKoZIhvcNAQEFBQAwRDEWMBQGA1UE
AwwNQUNFRElDT00gUm9vdDEMMAoGA1UECwwDUEtJMQ8wDQYDVQQKDAZFRlDT00x
CzAJBgNVBAYTAKVTMB4XDTA4MDQxODE2MjQyMloXDTI4MDQzMzE2MjQyMlowRDEW
MBQGA1UEAwNQUNFRElDT00gUm9vdDEMMAoGA1UECwwDUEtJMQ8wDQYDVQQKDAZF
....
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIB5zCCAY6gAwIBAgIUNJADaMM+URJrPMdoIeeAs9/CET4wCgYIKoZIzj0EAwIw
UjELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAKNbMRYwFAYDVQQHEw1TYW4gRnJhbmcNp
c2NvMR4wHAYDVQQDExVoYXNoaWNvcnAuZW5naW5ZXJpbmcwHhcNMrgwMjI4MDYx
....
-----END CERTIFICATE-----
```

The UI to upload these certificates looks like:

SSL/TLS Configuration

This allows you to add custom SSL/TLS data to the install. The most common usage is to add custom certificates to the system, to allow an internal certificate authority (CA) to be trusted.

This is done when you use certificates on your VCS provider and/or Terraform Enterprise installation and thus need Terraform Enterprise to trust services.

Custom Certificate Authority (CA) Bundle

Alternative Terraform worker image

TFE runs `terraform plan` and `terraform apply` operations in a disposable Docker containers. There are cases where runs may make frequent use of additional tools that are not available in the default Docker image. To allow use of these tools for

any plan or apply, users can build their own image and configure TFE to use that instead. In order for this to happen the name of the alternative docker image must be set in the config by using the `Custom image` tag field as shown below:

The screenshot shows a configuration interface for a Terraform Build Worker image. At the top, the title "Terraform Build Worker image" is displayed. Below it, a descriptive text states: "Configure which docker image will be used when running terraform plans and applies. This can either be the standard image that ships with PTFE or a custom image that includes extra tools not present in the default one." Two radio button options are presented: "Use TFE's standard image" (unchecked) and "Provide the location of a custom image" (checked). A text input field contains the value "some.docker.registry/my_custom_image". A blue "Save" button is located at the bottom right of the input field.

Requirements

- The base image must be `ubuntu:xenial`.
- The image must exist on the PTFE host. It can be added by running `docker pull` from a local registry or any other similar method.
- CA certificates must be available when terraform runs. During image creation, a file containing all necessary PEM encoded CA certificates must be placed in `/etc/ssl/certs/ca-certificates.crt`.
- Terraform must not be installed on the image. TFE will take care of that at runtime.

This is a sample Dockerfile you can use to start building your own image:

```
# This Dockerfile builds the image used for the worker containers.  
FROM ubuntu:xenial  
  
# Inject the ssl certificates  
ADD ca-certificates.crt /etc/ssl/certs/ca-certificates.crt
```

Operational Mode Decision

Terraform Enterprise can store its state in a few different ways, and you'll need to decide which works best for your installation. Each option has a different approach to recovering from failures ([/docs/enterprise/private/reliability-availability.html#recovery-from-failures-1](#)). The mode should be selected based on your organization's needs. See [Preflight: Operational Mode Decision](#) ([/docs/enterprise/private/preflight-installer.html#operational-mode-decision](#)) for more details.

Installation

The installer can run in two modes, Online or Airgapped. Each of these modes has a different way of executing the installer, but the result is the same.

Note: After running the installation script, the remainder of the installation is done through a browser using the installer dashboard on port 8800 of the TFE instance. To complete the installation, you must be able to connect to that port via HTTPS. The installer uses an internal CA to issue bootstrap certificates, so you will see a security warning when first connecting, and you'll need to proceed with the connection anyway.

Run the Installer - Online

If your instance can access the Internet, use the Online install mode.

1. From a shell on your instance:
 - To execute the installer directly, run `curl https://install.terraform.io/ptfe/stable | sudo bash`
 - To inspect the script locally before running, run `curl https://install.terraform.io/ptfe/stable > install.sh` and, once you are satisfied with the script's content, execute it with `sudo bash install.sh`.
2. The installation will take a few minutes and you'll be presented with a message about how and where to access the rest of the setup via the web. This will be `https://<TFE HOSTNAME>:8800`.
 - You will see a security warning when first connecting. This is expected and you'll need to proceed with the connection anyway.

Run the Installer - Airgapped

If the instance cannot reach the Internet, follow these steps to begin an Airgapped installation.

Prepare the Instance

1. Download the .airgap file using the information given to you in your setup email and place that file somewhere on the instance.
 - If you are using `wget` to download the file, be sure to use `wget --content-disposition "<url>"` so the downloaded file gets the correct extension.
 - The url generated for the .airgap file is only valid for a short time, so you may wish to download the file and upload it to your own artifacts repository.
2. Download the installer bootstrapper (`https://install.terraform.io/airgap/latest.tar.gz`) and put it into its own directory on the instance (e.g. `/opt/tfe-installer/`)
3. Airgap installations require Docker to be pre-installed. Double-check that your instance has a supported version of Docker (see Preflight: Software Requirements ([/docs/enterprise/private/preflight-installer.html#software-requirements](#)) for details).

Execute the Installer

From a shell on your instance, in the directory where you placed the `replicated.tar.gz` installer bootstrapper:

1. Run `tar xzf replicated.tar.gz`
2. Run `sudo ./install.sh airgap`
3. When asked, select the interface of the primary private network interface used to access the instance.
4. The software will take a few minutes and you'll be presented with a message about how and where to access the rest of the setup via the web. This will be `https://:8800`
 - You will see a security warning when first connecting. This is expected and you'll need to proceed with the connection anyway.

Continue Installation In Browser

1. Configure the hostname and the SSL certificate.
2. Upload the license file provided to you in your setup email.
3. Indicate whether you're doing an Online or Airgapped installation. Choose Online if you're not sure.
 - If you are doing an Airgapped install, provide the path on the the instance to the `.airgap` file that you downloaded using the initial instructions in your setup email.
4. Secure access to the installer dashboard. We recommend at least setting up the simple password authentication. If you're so inclined, LDAP authentication can also be configured.
5. The system will now perform a set of pre-flight checks on the instance and the configuration up to this point and indicate any failures. You can either fix the issues and re-run the checks, or ignore the warnings and proceed. If the system is running behind a proxy and is unable to connect to `releases.hashicorp.com:443`, it is likely safe to proceed; this check does not currently use the proxy. For any other issues, if you proceed despite the warnings, you are assuming the support responsibility.
6. Set an encryption password used to encrypt the sensitive information at rest. The default value is auto-generated, but we strongly suggest you create your own password. Be sure to retain the value, because you will need to use this password to restore access to the data in the event of a reinstall. See [Encryption Password](https://www.terraform.io/docs/enterprise/private/encryption-password.html) (<https://www.terraform.io/docs/enterprise/private/encryption-password.html>) for more information.
7. Configure the operational mode for this installation. See [Preflight: Operational Modes](#) ([/docs/enterprise/private/preflight-installer.html#operational-mode-decision](#)) for information on what the different values are. Ensure that you've met the relevant preflight requirements for the mode you chose.
8. *Optional:* Adjust the concurrent capacity of the instance. This should only be used if the hardware provides more resources than the baseline configuration and you wish to increase the work that the instance does concurrently. This setting should be adjusted with care as setting it too high can result in an very unresponsive instance.
9. *Optional:* Provide the text version of a certificate (or certificates) that will be added to the trusted list for the product. This is used when services the product communicates with do not use globally trusted certificates but rather a private Certificate Authority (CA). This is typically used when Private Terraform Enterprise uses a private certificate (it must trust itself) or a VCS provider uses a private CA.
10. *Optional:* Adjust the path used to store the vault files that are used to encrypt sensitive data. This is a path on the host system, which allows you to store these files outside of the product to enhance security. Additionally, you can configure the system not to store the vault files within any snapshots, giving you full custody of these files. These files

will need to be provided before any snapshot restore process is performed, and should be placed into the path configured.

11. *Optional:* Configure the product to use an externally managed Vault cluster. See [Externally Managed Vault Cluster](#) ([/docs/enterprise/private/vault.html](#)) for details on how to configure this option.

Finish Bootstrapping

Once configured, the software will finish downloading. When it's ready, the UI will indicate so and there will be an Open link to click to access the Terraform Enterprise UI.

Configuration

After completing a new install you should head to the configuration page ([/docs/enterprise/private/config.html](#)) to continue setting up Terraform Enterprise.

Private Terraform Enterprise

This section guides users through deploying Private Terraform Enterprise using the installer method. This includes required prerequisites, steps to install the software, and the basic configuration that will need to be done after install.

- Installation (</docs/enterprise/private/install-installer.html>)
- Configuration and Validation (</docs/enterprise/private/config.html>)
- Frequently Asked Questions (</docs/enterprise/private/faq.html>)

Terraform Enterprise Logs

This document contains information about interacting with Private Terraform Enterprise logs. There are two types of logs, application logs and audit logs. Application logs emit information about the services that comprise Terraform Enterprise. Audit logs emit information whenever any resource managed by Terraform Enterprise is changed.

Application Logs

Installer

Installer-based PTFE runs in a set of Docker containers. As such, any tooling that can interact with Docker logs can read the logs. This includes the command `docker logs`, as well as access via the Docker API (<https://docs.docker.com/engine/api/v1.36/#operation/ContainerLogs>).

An example of a tool that can automatically pull logs for all docker containers is `logspout` (<https://github.com/gliderlabs/logspout>). Logspout can be configured to take the Docker logs and send them to a syslog endpoint. Here's an example invocation:

```
$ docker run --name="logspout" \
--volume=/var/run/docker.sock:/var/run/docker.sock \
gliderlabs/logspout \
syslog+tls://logs.mycompany.com:55555
```

The logspout container uses the Docker API internally to find other running containers and ingress their logs, then send them to `logs.mycompany.com` on port 55555 using syslog with TCP/TLS.

NOTE: While docker has support for daemon-wide log drivers that can send all logs for all containers to various services, Private Terraform Enterprise only supports having the Docker log-driver configured to either `json-file` or `journald`. All other log drivers prevent the support bundle functionality from gathering logs, making it impossible to provide product support. **DO NOT** change the log driver of an installation to anything other than `json-file` or `journald`.

AMI based installs

Private Terraform Enterprise's application-level services all log to CloudWatch logs, with one stream per service. The stream names take the format:

```
{hostname}-{servicename}
```

Where `hostname` is the fqdn you provided when setting up Private Terraform Enterprise, and `servicename` is the name of the service whose logs can be found in the stream. More information about each service can be found in `tfe-architecture`.

For example, if your Private Terraform Enterprise installation is available at `tfe.mycompany.io`, you'll find CloudWatch Log streams like the following:

```
tfe.mycompany.io-atlas-frontend  
tfe.mycompany.io-atlas-worker  
tfe.mycompany.io-binstore  
tfe.mycompany.io-logstream  
tfe.mycompany.io-packer-build-manager  
tfe.mycompany.io-packer-build-worker  
tfe.mycompany.io-slug-extract  
tfe.mycompany.io-slug-ingress  
tfe.mycompany.io-slug-merge  
tfe.mycompany.io-storagelocker  
tfe.mycompany.io-terraform-build-manager  
tfe.mycompany.io-terraform-build-worker  
tfe.mycompany.io-terraform-state-parser
```

CloudWatch logs can be searched, filtered, and read from either from the AWS Web Console or (recommended) the command line awslogs (<https://github.com/jorgebastida/awslogs>) tool.

Note: All other system-level logs can be found in the standard locations for an Ubuntu 16.04 system.

Audit Logs

Audit log entries are written to the application logs. To distinguish audit Log entries from other log entries, the JSON is prefixed with [Audit Log].

Installer

The audit logs are emitted along with other logs by the `ptfe_atlas` container.

AMI based installs

As of Private Terraform Enterprise release v201802-1, audit logging is available in Private Terraform Enterprise. These are written out to CloudWatch logs just like all other application-level logs.

Log Contents

The audit log will be updated when any resource managed by Terraform Enterprise is changed. Read requests will be logged for resources deemed sensitive. These include:

- Authentication Tokens
- Configuration Versions
- Policy Versions
- OAuth Tokens
- SSH Keys
- State Versions

- Users
- Variables

When requests occur, these pieces of information will be logged:

1. The actor
 - Users (including IP address)
 - Version Control System users (identified in webhooks)
 - Service accounts
 - Terraform Enterprise
2. The action
 - Reading sensitive resources
 - Creation of new resources
 - Updating existing resources
 - Deletion of existing resources
 - Additional actions as defined in /actions/* namespaces
 - Webhook API calls
3. The target of the action (any resource exposed by the V2 API)
4. The time that the action occurred
5. Where the action was taken (web/API request, background job, etc.)

Log Format

Log entries are in JSON, just like other Terraform logs. Most audit log entries are formatted like this:

```
{
  "timestamp": "2017-12-19T15:23:45.148Z",
  "resource": "workspace",
  "action": "destroy",
  "resource_id": "ws-9a3hrbYffsTzg2FZ",
  "actor": "jsmith",
  "actor_ip": "94.122.17.37"
}
```

Certain entries will contain additional information in the payload, but all audit log entries will contain the above keys.

Log Location

Audit log entries are written to the application logs. To distinguish audit log entries from other log entries, the JSON is prefixed with [Audit Log]. These are written out to CloudWatch logs just like all other application-level logs. For example:

2018-03-27 21:55:29 [INFO] [Audit Log] {"resource": "oauth_client", "action": "create", "resource_id": "oc-FErAhnuHHwcad3Kx", "actor": "atlasint", "timestamp": "2018-03-27T21:55:29Z", "actor_ip": "11.22.33.44"}

Private Terraform Enterprise Installer Migration

This document outlines the procedure for migrating from the AMI-based Private Terraform Enterprise (PTFE) to the Installer-based PTFE.

Terraform State

To run this procedure, you'll need the Terraform state file used to create the AMI-based installation. Additionally, we strongly suggest you back up this state file before proceeding with this process, in the event that you need to revert back to the AMI.

Backup

Before beginning, it's best to create an additional backup of your RDS database. This will allow you to roll back the data and continue to use the AMI if necessary.

To create an RDS backup, go to the Amazon RDS Instances (<https://console.aws.amazon.com/rds/home?region=us-east-1#dbinstances:>). You may need to change the region that you are viewing in order to see your PTFE instance. Once you find it, click on the instance name. On the next page, select **Instance Actions** and then **Take snapshot**. On the **Take DB Snapshot** page, enter a name for the snapshot such as Pre-Installer Migration, and then click **Take Snapshot**.

The snapshot will take a little while to create. After it has finished, you can continue with the rest of the migration steps.

Reverting back to the AMI

To revert to the AMI after running the migration script:

- If you've already manipulated the state file to move the resources, you'll need to restore the original state file.
- With your original Terraform state file in place, return to the Amazon RDS Snapshots (<https://console.aws.amazon.com/rds/home?region=us-east-1#db-snapshots:>) and find the snapshot you created before migrating. Click on the snapshot and note its **ARN** value. Open your **.tfvars** file and add `db_snapshot = "arn-value-of-snapshot"`.
- Run `terraform apply` to make sure everything is set up. This will result in a new RDS instance being built against the snapshot.
- If the EC2 instance is still running from the migration process, run `shutdown` on it to get a new instance created.
- Return to the original hostname used for the cluster.

Simple Upgrade (Recommended)

For users who did not significantly change the reference Terraform modules used to deploy the AMI, we recommend following the migration path outlined on the Simplified Migration Page (/docs/enterprise/private/simplified-migration.html).

For users who modified the Terraform modules, we recommend following the steps below. If you prefer, you can follow the simplified steps ([/docs/enterprise/private/simplified-migration.html](#)) as well, but you'll likely need to modify some Terraform modules again to match the modifications you made in the past.

Preflight

Before you begin, consult Preflight ([/docs/enterprise/private/preflight-installer.html](#)) for pre-requisites for the installer setup. You'll need to prepare data files and a Linux instance.

Proxy Usage

If your installation requires using a proxy server, you will be asked for the proxy server information when you first run the installer via ssh. This proxy server will be used for all outbound HTTP and HTTPS connections.

Optionally, if you're running the installer script in an automated manner, you can pass a `http-proxy` flag to set the address of the proxy. For example:

```
./install.sh http-proxy=http://internal.mycompany.com:8080
```

Proxy Exclusions (NO_PROXY)

If certain hostnames should not use the proxy and the instance should connect directly to them (for instance for S3), then you can pass an additional option to provide a list of domains:

```
./install.sh additional-no-proxy=s3.amazonaws.com,internal-vcs.mycompany.com,example.com
```

Passing this option to the installation script is particularly useful if the hostnames that should not use the proxy include services that the instance needs to be able to reach during installation, such as S3. Alternately, if the only hosts you need to add are those that are not used during installation, such as a private VCS instance, you can provide these hosts after initial installation is complete, in the installer settings (available on port 8800 under `/console/settings`).

Reconfiguring the Proxy

To change the proxy settings after installation, use the Console settings page, accessed from the dashboard on port 8800 under `/console/settings`.

A screenshot of the Terraform Enterprise dashboard. At the top right, there is a user menu icon with a gear symbol. A red arrow points to the 'Console Settings' option in the dropdown menu, which is highlighted with a white background and black text. Other options in the menu include 'View License', 'Snapshots', and 'Logout'. The main content area displays a blue banner with the text 'Terraform Enterprise is up to date' and 'Last checked: 2 hours ago'.

On the Console Settings page, there is a section for HTTP Proxy:

A screenshot of the 'HTTP Proxy' settings page. The page title is 'HTTP Proxy'. Below it, a note states: 'The management console can use a proxy server to connect to the Internet, if required.' There is a checkbox labeled 'HTTP Proxy enabled' which is checked. Below the checkbox is a field labeled 'HTTP Proxy server' with a placeholder text box. A note below the text box says: 'If a proxy server is required to connect to the internet, enter it here in <http://10.10.10.10:8888> format.'

Trusting SSL/TLS Certificates

There are two primary areas for SSL configuration in the installer.

TLS Key & Cert

The TLS Key & Cert field (found in the console settings after initial installation) should contain PTFE's own key and certificate, or key and certificate chain. A chain would be used in this field if the CA indicates an intermediate certificate is required as well.

Certificate Authority (CA) Bundle

PTFE needs to be able to access all services that it integrates with, such as VCS providers. Because it typically accesses them via SSL/TLS, it is critical that the certificates used by any service that PTFE integrates with are trusted by PTFE.

This section is used to allow PTFE to connect to services that use SSL/TLS certificates issued by private CAs. It allows multiple certificates to be specified as trusted, and should contain all certificates that PTFE should trust when presented with them from itself or another application.

A collection of certificates for trusted issuers is known as a Certificate Authority (CA) Bundle. All certificates in the certificate signing chain, meaning the root certificate and any intermediate certificates, must be included here. These multiple certificates are listed one after another in text format.

Note: If PTFE is configured with a SSL key and certificate issued against a private CA, the certificate chain for that CA must be included here as well. This allows the instance to query itself.

Certificates must be formatted using PEM encoding, that is, as text. For example:

```
-----BEGIN CERTIFICATE-----
MIIFtTCCA52gAwIBAgIIYY3HhjsBggUwDQYJKoZIhvcNAQEFBQAwRDEWMBQGA1UE
AwwNQUNFRElDT00gUm9vdDEMMAoGA1UECwwDUEtJMQ8wDQYDVQQKDAZFRlDT00x
CzAJBgNVBAYTAKVTMB4XDTA4MDQxODE2MjQyMloXTI4MDQxMzE2MjQyMlowRDEW
MBQGA1UEAwNQUNFRElDT00gUm9vdDEMMAoGA1UECwwDUEtJMQ8wDQYDVQQKDAZFR
...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIB5zCCAY6gAwIBAgIUNJADaMM+URJrPMdoTeeAs9/CEt4wCgYIKoZIzj0EAwIw
UjELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAkNBMRYwFAYDVQQHEw1TYW4gRnJhbmNp
c2NvMR4wHAYDVQQDExVoYXNoaNvcnAuZW5naW5lZXJpbmcwHhcNMTgwMjI4MDYx
...
-----END CERTIFICATE-----
```

The UI to upload these certificates looks like:

SSL/TLS Configuration

This allows you to add custom SSL/TLS data to the install. The most common usage is to add custom certificates to the system, to allow an internal certificate authority (CA) to be trusted.

This is done when you use certificates on your VCS provider and/or Terraform Enterprise installation and thus need Terraform Enterprise to trust services.

Custom Certificate Authority (CA) Bundle

Operational Mode

As you are migrating from the AMI, you'll be using **Production - External Services** to access the data previously managed by the AMI-based installation. This means using RDS for PostgreSQL and S3 to store the objects.

Begin Migration

Now that the instance has been booted into the correct Linux environment, you're ready to begin the migration process.

Note: The migration process will render the AMI-based installation inoperable. Be sure to back up RDS before beginning.

Schedule downtime for the installation and do not continue until that downtime period has arrived.

Prepare for migration

SSH to the instance currently running in your PTFE AutoScaling group. Then download the migrator tool onto it to run the migration procedure:

```
$ curl -O https://s3.amazonaws.com/hashicorp-ptfe-data/migrator  
$ chmod a+x migrator
```

Migrate data out of the AMI instance

Note: The AMI instance will not function after this step. Certain services are stopped and should not be restarted after the migration. The migration moves data stored in Consul by the AMI (Vault data) into PostgreSQL to be used by the installer. If the AMI adds new data to Consul after the migration, the installer-based instance won't have access to it, destabilizing the system.

Next, create a password that will be used to protect the migration data as it is passed to the installer. The password is only used for this migration process and is not used after the migration is complete.

For this example, we'll use the password `ptfemigrate`, but you must change it to a password of your own choosing.

This process shuts down certain services on the AMI instance to verify consistency before moving data into the RDS instance to be used by the installer.

SSH into the AMI instance, then run these commands:

```
$ sudo ./migrator -password ptfemigrate
```

The command will ask you to confirm that you wish to continue. When are ready, run:

```
$ sudo ./migrator -password ptfemigrate -confirm
```

An example session of running the migrator looks like:

```
This program will prepare your database for migration as well as
output a block of text that you must provide when requested while
later running this program on the linux instance being used
to run the installer.
```

```
This program will now shutdown some primary services before
continuning to be sure that the database is consistent
Shutting down services...
```

```
2018/05/25 17:37:04 Generated new root token to transmit
2018/05/25 17:37:04 connecting to postgres at postgres://atlas:xxyyzz@tfe-aabbcc.ddeeff.us-west-2.rds.amazonaws.com:5432/atlas_production?&options=-c%20search%5Fpath%3Dvault
2018/05/25 17:37:04 setup rails schema!
2018/05/25 17:37:04 writing core/audit
2018/05/25 17:37:04 writing core/auth
```

```
2018/05/25 17:37:04 writing core/cluster/local/info
2018/05/25 17:37:04 writing core/keyring
2018/05/25 17:37:04 writing core/leader/615b50e4-206d-9170-7971-f2b6b8a508fd
2018/05/25 17:37:04 writing core/local-audit
2018/05/25 17:37:04 writing core/local-auth
2018/05/25 17:37:04 writing core/local-mounts
2018/05/25 17:37:04 writing core/lock
2018/05/25 17:37:04 writing core/master
2018/05/25 17:37:04 writing core/mounts
2018/05/25 17:37:04 writing core/seal-config
2018/05/25 17:37:04 writing core/wrapping/jwtkey
2018/05/25 17:37:04 writing logical/91a7ef51-f1c5-b0a2-8755-3a4d86f1a082/ptfe-backup
2018/05/25 17:37:04 writing logical/91a7ef51-f1c5-b0a2-8755-3a4d86f1a082/setup-data
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/archivist_encoder
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/archivist_kdf
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_events_resource_diff
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_o_auth_clients_secret
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_o_auth_clients_webhook_secret
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_o_auth_tokens_info
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_o_auth_tokens_value
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_runs_metadata
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_runs_var_snapshot
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_setting_storage_postgres_value
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_vcs_hosts_webhook_secret
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/archive/atlas_vcs_repos_webhook_secret
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/archivist_encoder
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/archivist_kdf
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_events_resource_diff
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_o_auth_clients_secret
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_o_auth_clients_webhook_secret
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_o_auth_tokens_info
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_o_auth_tokens_value
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_runs_metadata
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_runs_var_snapshot
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_setting_storage_postgres_value
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_vcs_hosts_webhook_secret
2018/05/25 17:37:04 writing logical/e0eb6eb8-56e2-d564-8fc4-78ee882229e8/policy/atlas_vcs_repos_webhook_secret
2018/05/25 17:37:04 writing sys/expire/id/auth/token/create-orphan/5c40cdecc64ba276ca810a7d43cf69a4b486ad4d
2018/05/25 17:37:04 writing sys/expire/id/auth/token/create/aaf09a178b4658fd926f9193caef9e1817aae7c4
2018/05/25 17:37:04 writing sys/policy/archivist
2018/05/25 17:37:04 writing sys/policy/atlas
2018/05/25 17:37:04 writing sys/policy/default
2018/05/25 17:37:04 writing sys/policy/response-wrapping
2018/05/25 17:37:04 writing sys/token/Accessor/105c7e4b4898cd905a915b48c7e8d3a20b7e8d25
2018/05/25 17:37:04 writing sys/token/Accessor/9ff1ff31131be14d563ce1f0fb4781610175189f
2018/05/25 17:37:04 writing sys/token/Accessor/b336f38abdbe381c8989c90f980c8267a3af999e
2018/05/25 17:37:04 writing sys/token/Accessor/c0142d331857496927a3799ea9df277e69d0a982
2018/05/25 17:37:04 writing sys/token/Id/5c40cdecc64ba276ca810a7d43cf69a4b486ad4d
2018/05/25 17:37:04 writing sys/token/Id/5ed7ddc1674fbb42fe3ddcd278802a116b7cd4ab
2018/05/25 17:37:04 writing sys/token/Id/aaf09a178b4658fd926f9193caef9e1817aae7c4
2018/05/25 17:37:04 writing sys/token/Id/c356180d070a7aaafa29f25416d5b8d2bba3dfffa8
```

```
2018/05/25 17:37:04 writing sys/token/salt
2018/05/25 17:37:04 all vault data copied
copy and paste this data onto the instance PTFE will be installed on
```

```
-----BEGIN PTFE MIGRATION DATA-----
sl+VOHlCIKTq+aX7h/ZROP+bQkuJJlw/Is8WeFB/lfETBxXPmRwA6Ll9gYxyxpmq
hYUuw5Sj6IkHYGpwyw8tHdWyDTPHbtRZJtywGx8DAV6qi6gHbjjm93h4Rx5CIOZL
N3KBsHJR2tbp+xa1AtHxzzG+dzE1GzZwIbka/v0p2vQd47sroYSppMuH5GwcwM8G
sg64byIj00oj5UhF289hEoVLAL9jz51thzy8hXKC5UQtGnesva5c2hYMKu/Hkf68
7Irg7kWuruc+6MM9F2WIZ/gGGtMqEZQM2NL7WvBlccbFZHDNGsWojiWhK+IJPK0C
vj217NebILBph0koXiXsg0PnUNhovLl9Gp/4kuHUT01pZbTzdwd/Va0uLZLY4Fc
s8qx91pxLCVThCDEFJ4mfyFk7Q0bTl760bxnqrGMSJ++Co/Wt264cUqDUoJ7rCdN
pQfT0cPk+0RLw4qBFPEL2iKOe4r878PajKaRThE4yt62noF/pfabV0yckVz2lfFY
vnll85GxgpBdyMp4EGrJyxPqCMSCiBNLzMT6MZWyndlcRwkSf8xZyw36wMoF48Nz
nITmImg+IzyV+eb3gjRqAC6zl2Dd/QZiKti1fgwKt1YFSMIHOuHWpTsxh1b/fAtQ
Gchx01Jkp2kd73jxEEJ8r9yEDZzNf5Hh4U8IIiLxFNCP/2UjEtPI5nNSnxm3/fj3v
EhdSjIQGthiXHuU4gAOc4c0=
-----END PTFE MIGRATION DATA-----
```

When the migrator command completes, the output will include some debugging information to provide support if there is an issue as well as a specially-formatting block of text that looks like:

```
-----BEGIN PTFE MIGRATION DATA-----
rbBuoUY1CZxFBH8sLHYpZzc7ndBD7on3aaro4Br9kV9Wt8Pq0MYhS9Ts/nHr2ev
cpN3EQEuK+Bur/TBYEEHjXiiv71aGj06c2z68yhvfafDbTfvW8N0rXDQM26F2rTa
1NGZNb1FUFPWFvPqbLW25xw5h+wn1Fu0lEkqVsaBSdpYWGH/7NPtRDX2dN2ZhPYN
vs5T0wZ3ToPM7e7yfwI/0uMg+2lAK0h4L/ioYgLHBSN/bGr/Plfj6+CR7zza8XC3
kSv6kzGFr7fCT4BWzSJc38s0MviBY/TdyDPRLmmM7Flg9JLYIvLVxosRfIneQn60
K5Jcf1wJEsh/ZAQpEkbp6PzQKUmAlvEHf/fSAPxf3W3ulAubrCe2QmwsPIjkIGx/
uwfKgTCFr/svIkAtccTmeQ8pxAmNEA4CW/jEGlFkkZiu74+ia8Kdf5lrtJBLA9Lj
J8sue9KYAB64fwQ+fISLfrMeahjgFip3gPxofDevM2gbIFvde2iCFFbPll3oYLRK
nNVGb9Fa2ttIlP6oFZTp7Ph0FZ+oSCtmFeDmpjQ5aM5C4WxmfMjs7IDJGup2ZyI
l7X33mQ270NGbKc/k6aCaqMZxyKewDk9bGalULh4dwSZxzNl4sJeb6DarkMyN1gp
SggAGKOsNGVfaGlm5WNTAxRJZR3dS4UV3ar9UVhblXK014cm6fKt5ByNBvGTtuy0
h3tR9gbjVjBA5S+iXP7lSb8=
-----END PTFE MIGRATION DATA-----
```

The first and last lines will be the same as above but the body will be entirely different. This is a PEM encoded block that contains the configuration as well as Vault keys to allow the installer to access your data.

Copy this block of text, including the beginning and ending lines beginning with ----- to your computer's clipboard by selecting the text and typing Control-C (Windows) or Command-C (macOS).

Import data into the installer instance

To transfer the migration data to the installer-based instance, connect to it over SSH.

Once connected, run the following commands to add the migrator tool to the instance:

```
$ curl -0 https://s3.amazonaws.com/hashicorp-ptfe-data/migrator
$ chmod a+x migrator
```

Then, to import the data, run the following command, passing in the same password used to generate the migration data.

```
$ sudo ./migrator -password ptfe migrate
```

When prompted, paste the data you previously copied to your computer's clipboard.

The migrator process places the Vault data in the default path for the installer to find, and outputs the values you'll need later in the process for the Installer web interface.

Installing PTFE

The installer can run in two modes, Online or Airgapped. Each of these modes has a different way of executing the installer, but the result is the same.

Run the Installer - Online

If your instance can access the Internet, use the Online install mode.

1. From a shell on your instance:
 - To execute the installer directly, run `curl https://install.terraform.io/ptfe/stable | sudo bash`
 - To inspect the script locally before running, run `curl https://install.terraform.io/ptfe/stable > install.sh` and, once you are satisfied with the script's content, execute it with `sudo bash install.sh`.
2. The installation will take a few minutes and you'll be presented with a message about how and where to access the rest of the setup via the web. This will be `https://<TFE HOSTNAME>:8800`
 - The installer uses an internal CA to issue bootstrap certificates, so you will see a security warning when first connecting. This is expected and you'll need to proceed with the connection anyway.

Run the Installer - Airgapped

If the instance cannot reach the Internet, follow these steps to begin an Airgapped installation.

Prepare the Instance

1. Download the `.airgap` file using the information given to you in your setup email and place that file somewhere on the the instance.
 - If you use are using `wget` to download the file, be sure to use `wget --content-disposition "<url>"` so the downloaded file gets the correct extension.
 - The url generated for the `.airgap` file is only valid for a short time, so you may wish to download the file and upload it to your own artifacts repository.
2. Download the installer bootstrapper (`https://install.terraform.io/airgap/latest.tar.gz`) and put it into its own directory on the instance (e.g. `/opt/tfe-installer/`)
3. Airgap installations require Docker to be pre-installed. Double check that your instance has a supported version of Docker (see Preflight: Software Requirements ([/docs/enterprise/private/preflight-installer.html#software-requirements](#)) above for details).

Execute the Installer

From a shell on your instance, in the directory where you placed the `replicated.tar.gz` installer bootstrapper:

1. Run `tar xzf replicated.tar.gz`
2. Run `sudo ./install.sh airgap`
3. When asked, select the interface of the primary private network interface used to access the instance.
4. The software will take a few minutes and you'll be presented with a message about how and where to access the rest of the setup via the web. This will be `https://:8800`
 - The installer uses an internal CA to issue bootstrap certificates, so you will see a security warning when first connecting. This is expected and you'll need to proceed with the connection anyway.

Continue Installation In Browser

1. Configure the hostname and the SSL certificate. **NOTE:** This does not need to be same hostname that was used by the AMI. It must be a hostname that currently resolves in DNS properly.
2. Upload your license file, provided to you in your setup email.
3. Indicate whether you're doing an Online or Airgapped installation. Choose Online if you're not sure.
 - If you are doing an Airgapped install, provide the path on the the instance to the `.airgap` file that you downloaded using the initial instructions in your setup email.
4. Secure access to the installer dashboard. We recommend at least setting up the simple password authentication. If you're so inclined, LDAP authentication can also be configured.
5. The system will now perform a set of pre-flight checks on the instance and configuration thus far and indicate any failures. You can either fix the issues and re-run the checks, or ignore the warnings and proceed. If the system is running behind a proxy and is unable to connect to `releases.hashicorp.com:443`, it is likely safe to proceed; this check does not currently use the proxy. For any other issues, if you proceed despite the warnings, you are assuming the support responsibility.
6. Set an encryption password used to encrypt the sensitive information at rest. The default value is auto-generated, but we strongly suggest you create your own password. Be sure to retain the value, because you will need to use this password to restore access to the data in the event of a reinstall.
7. Select **External Services** under Production Type.
8. For the PostgreSQL URL, copy and paste the value that was output by the `migrator` process for *PostgreSQL Database URL*.
9. Select **S3** under Object Storage.
10. Configure the Access Key ID and Secret Access Key, *OR* indicate that you want to use an instance profile. **NOTE:** An instance profile must be previously configured on the instance.
11. For the Bucket, copy and paste the value that was output by the `migrator` process for *S3 Bucket*.
12. For the Region, copy and paste the value that was output by the `migrator` process for *S3 Region*.
13. For the Server-side Encryption, copy and paste the value that was output by the `migrator` process for the server-side encryption.
14. For the KMS key, copy and paste the value that was output by the `migrator` process for *Optional KMS key*. **NOTE:** This

key is not optional for migration, as it is used to read all the data in S3.

15. *Optional:* Adjust the concurrent capacity of the instance. This should only be used if the hardware provides more resources than the baseline configuration and you wish to increase the work that the instance does concurrently. This setting should be adjusted with care as setting it too high can result in an very unresponsive instance.
16. *Optional:* Provide the text version of a certificate (or certificates) that will be added to the trusted list for the product. This is used when services the product communicates with do not use globally trusted certificates but rather a private Certificate Authority (CA). This is typically used when Private Terraform Enterprise uses a private certificate (it must trust itself) or a VCS provider uses a private CA.

Finish Installing

Once configured, the software will finish downloading. When it's ready, the UI will indicate so and there will be an Open link to click to access the Terraform Enterprise UI.

Access

You can now access your PTFE installation using the previously configured hostname.

Cleanup

Now that the installer is using a subset of resources created by the AMI cluster Terraform configuration, we need to move those resources from the Terraform state for the AMI cluster.

If these resources are not moved, it's possible to accidentally delete the data when the AMI cluster is removed!

NOTE: If you have modified the Terraform modules we have provided, you'll need to adapt these instructions to fit your modifications.

Move to new Terraform state

First, verify all the resources currently tracked in the Terraform state. If the `terraform.tfstate` file is on your local disk, change to that directory. Next, list the state:

```
$ terraform state list
aws_caller_identity.current
aws_kms_alias.key
aws_kms_key.key
aws_subnet.instance
aws_vpc.vpc
module.db.aws_db_instance.rds
module.db.aws_db_subnet_group.rds
module.db.aws_security_group.rds
module.instance.aws_autoscaling_group.ptfe
module.instance.aws_ebs_volume.data[0]
module.instance.aws_ebs_volume.data[1]
module.instance.aws_elb.ptfe
module.instance.aws_iam_instance_profile.tfe_instance
module.instance.aws_iam_policy_document.tfe-perms
module.instance.aws_iam_role.tfe_iam_role
module.instance.aws_iam_role_policy.tfe-perms
module.instance.aws_launch_configuration.ptfe
module.instance.aws_s3_bucket.tfe_bucket
module.instance.aws_s3_bucket_object.setup
module.instance.aws_security_group.ptfe
module.instance.aws_security_group.ptfe-external
module.instance.aws_subnet.subnet
module.route53.aws_route53_record.rec
random_id.installation-id
```

Of these resources, the ones to move from the state so they are can exist outside this Terraform configuration are:

```
aws_kms_alias.key
aws_kms_key.key
module.db.aws_db_instance.rds
module.db.aws_db_subnet_group.rds
module.db.aws_security_group.rds
module.instance.aws_s3_bucket.tfe_bucket
```

To move these resources from the state into a new state file for later use, run:

```
$ terraform state mv -state-out=terraform.tfstate.installer aws_kms_alias.key aws_kms_alias.key
$ terraform state mv -state-out=terraform.tfstate.installer aws_kms_key.key aws_kms_key.key
$ terraform state mv -state-out=terraform.tfstate.installer module.db.aws_db_instance.rds module.db.aws_db_instance.rds
$ terraform state mv -state-out=terraform.tfstate.installer module.db.aws_db_subnet_group.rds module.db.aws_db_subnet_group.rds
$ terraform state mv -state-out=terraform.tfstate.installer module.db.aws_security_group.rds module.db.aws_security_group.rds
$ terraform state mv -state-out=terraform.tfstate.installer module.instance.aws_s3_bucket.tfe_bucket module.instance.aws_s3_bucket.tfe_bucket
```

Destroy AMI Cluster

Now that these resources have been moved out of the AMI state, it's safe to destroy the AMI cluster without affecting the installer-based installation.

If you're ready to perform that step, run:

```
$ terraform destroy
```

You'll be asked to confirm the resources to be destroyed, and should double-check that the list doesn't contain the resources moved earlier.

Managing Installer Resources

To continue to manage the resources used by the Installer-based install using Terraform, you can move the resources in the new `terraform.tfstate.installer` file to a large state file, along with the Terraform modules for them.

Private Terraform Enterprise Installation (Installer) - Minio Setup Guide

This document provides an overview for setting up Minio (<https://minio.io>) for external object storage for HashiCorp Private Terraform Enterprise (PTFE).

Required Reading

- Ensure you are familiar with PTFE's operation and installation requirements (</docs/enterprise/private/installer.html>), and especially the Operational Mode Decision (</docs/enterprise/private/preflight-installer.html#operational-mode-decision>).
- Familiarize yourself with Minio (<https://minio.io>).

Overview

When configured to use external services, PTFE must be connected to a storage service to persist workspace state and other file-based data. Native support exists for Azure Blob Storage, Amazon S3, and services that are API-compatible with Amazon S3. If you are not using Azure or a cloud provider with an S3-compatible service, or you are running PTFE in an environment without a storage service, it may be possible to use Minio (<https://minio.io>) instead.

Installation

Note: This is not a production-ready configuration: it's intended to guide you to a working configuration that can later be automated and hardened.

This guide will walk through installing Minio in a Docker container alongside PTFE on the same host, with PTFE configured in the "Production - External Services" operational mode (</docs/enterprise/private/preflight-installer.html#operational-mode-decision>). Data will not be persisted outside of an ephemeral Docker volume, Minio will not start on system boot, etc. The guide assumes your instance will have access to the Internet and that you will be performing an online install of PTFE.

System preparation

Ensure your Linux instance meets the requirements (</docs/enterprise/private/preflight-installer.html#linux-instance>). You will need jq (<https://stedolan.github.io/jq/>) (a command-line JSON processor), and the AWS CLI (<https://aws.amazon.com/cli/>).

You also need a PostgreSQL database that meets the requirements (</docs/enterprise/private/preflight-installer.html#postgresql-requirements>), as this is part of the external services operational mode.

PTFE installation

Begin with an online installation (</docs/enterprise/private/install-installer.html#run-the-installer-online>). Once the installation script has finished and you're presented with the following text, move on to the next section:

```
To continue the installation, visit the following URL in your browser:
```

```
https://<TFE HOSTNAME>:8800
```

Start Minio

Now you'll start the Minio container, mounting a volume so that you can gain access to the generated config:

```
docker run \
-d \
--name minio \
-v /run/minio/config:/root/.minio \
minio/minio:latest \
-- \
server /data
```

Ensure that Minio has started by watching for `/var/run/minio/config/config.json` to be written:

```
while [ ! -e /var/run/minio/config/config.json ]; do
  sleep 3
done
```

You now need to collect several pieces of information about your running Minio instance:

- IP address of the running container: `docker inspect minio | jq -r .[0].NetworkSettings.IPAddress`
- Access key: `jq -r .credential.accessKey /var/run/minio/config/config.json`
- Secret key: `jq -r .credential.secretKey /var/run/minio/config/config.json`

Create a bucket

Like S3, Minio does not automatically create buckets. Use the AWS CLI to create a bucket named `ptfe` that will be used to store data:

```
export AWS_ACCESS_KEY_ID=<access key from above>
export AWS_SECRET_ACCESS_KEY=<secret key from above>

aws --region us-east-1 --endpoint-url http://<ip address from above>:9000 s3 mb s3://ptfe
```

PTFE installation

You may now continue the installation in the browser (</docs/enterprise/private/install-installer.html#continue-installation-in-browser>). When you arrive at the Operational Mode choice in the installer, follow these steps:

1. Choose the "Production" installation type

2. Choose the "External Services" production type
3. Provide the required Database URL for the PostgreSQL configuration
4. Choose "S3" for object storage
5. Enter the access key and secret access key using the information retrieved from Minio
6. Provide the endpoint URL, like: `http://<ip address from above>:9000`
7. Enter the name of the bucket you created above (ptfe in the example)
8. Enter us-east-1 for the region; this is arbitrary, but must be a valid AWS region **Note:** The "Test Authentication" button does not currently work for non-AWS endpoints
9. Click "Save"

Next Steps

- Familiarize yourself with the various storage backends provided by Minio
- Make sure you know how to back up and restore the data written to Minio

Monitoring a Private Terraform Enterprise Instance

This document outlines best practices for monitoring a Private Terraform Enterprise (PTFE) instance.

Health Check

PTFE provides a `/_health_check` endpoint on the instance. If PTFE is up, the health check will return a `200 OK`.

Metrics & Telemetry

In addition to health-check monitoring, we recommend monitoring standard server metrics on the PTFE instance:

- I/O
- CPU
- Disk

Internal Monitoring

Beginning in version 201811-1, PTFE includes internal monitoring of critical application metrics using a statsd collector on port 8125. The resulting metrics are included with the diagnostic bundles provided to HashiCorp Support ([/docs/enterprise/private/diagnostics.html](#)). If the PTFE instance is already running a collector on this port, PTFE may not start up correctly due to the conflict, and logs will indicate:

```
Error starting userland proxy: listen udp 0.0.0.0:8125: bind: address already in use
```

To prevent this conflict, disable metrics collection by PTFE:

1. Access the installer dashboard on port 8800 of your instance.
2. Locate **Advanced Configuration** on the configuration page under **Terraform Build Worker image**.
3. Uncheck "Enable metrics collection".
4. Restart the application from the dashboard if it does not restart automatically.

We recommend that internal monitoring only be disabled if it is causing issues, as it otherwise provides useful detail in diagnosing issues.

Private Terraform Enterprise Installation (Preflight Requirements)

Before installing the Private Terraform Enterprise software, you'll need to prepare an appropriate instance and relevant data files. Please make careful note of these requirements, as the installation may not be successful if these requirements are not met.

Data Files

- TLS private key and certificate
 - The installer allows for using a certificate signed by a public or private CA. If you do not use a trusted certificate, your VCS provider will likely reject that certificate when sending webhooks. The key and X.509 certificate should both be PEM (base64) encoded.
- License file (provided by HashiCorp)

Note: If you use a certificate issued by a private Certificate Authority, you must provide the certificate for that CA in the Certificate Authority (CA) Bundle section of the installation. This allows services running within PTFE to access each other properly. See [Installation: Trusting SSL/TLS Certificates \(/docs/enterprise/private/install-installer.html#trusting-ssl-tls-certificates\)](#) for more on this.

Linux Instance

Install the software on a Linux instance of your choosing. You will start and manage this instance like any other server.

The Private Terraform Enterprise Installer currently supports the following operating systems:

- Debian 7.7+
- Ubuntu 14.04 / 16.04 / 18.04
- Red Hat Enterprise Linux 7.2+
- CentOS 7+
- Amazon Linux 2016.03 / 2016.09 / 2017.03 / 2017.09 / 2018.03 / 2.0
- Oracle Linux 7.2+

Hardware Requirements

These requirements provide the instance with enough resources to run the Terraform Enterprise application as well as the Terraform plans and applies.

- At least 40GB of disk space on the root volume
- At least 8GB of system memory

- At least 2 CPU cores

Software Requirements

RedHat Enterprise Linux (RHEL) has a specific set of requirements. Please see the RHEL Install Guide ([/docs/enterprise/private/rhel-install-guide.html](#)) before continuing.

See the CentOS Install Guide ([/docs/enterprise/private/centos-install-guide.html](#)) for additional guidance if installing on CentOS.

For Linux distributions other than RHEL, check Docker compatibility:

- The instance should run a supported version of Docker engine (1.7.1 or later, minimum 17.06.2-ce, maximum 17.12.1). This also requires a 64-bit distribution with a minimum Linux Kernel version of 3.10.
 - At this time, the **18.x and higher Docker versions are not supported**
 - In Online mode, the installer will install Docker automatically
 - In Airgapped mode, Docker should be installed before you begin
- For *RedHat Enterprise* and *Oracle Linux*, you **must** pre-install Docker as these distributions are not officially supported by Docker Community Edition (<https://docs.docker.com/engine/installation/#server>).

Note: It is not recommended to run Docker under a 2.x kernel.

SELinux

Private Terraform Enterprise does not support SELinux. The host running the installer must be configured in permissive mode by running: `setenforce 0`.

Future releases may add native support for SELinux.

Network Requirements

1. Have the following ports available to the Linux instance:
 - **22** - to access the instance via SSH from your computer. SSH access to the instance is required for administration and debugging.
 - **8800** - to access the installer dashboard.
 - **443** and **80** - to access the TFE app (both ports are needed; HTTP will redirect to HTTPS).
 - **9870-9880 (inclusive)** - for internal communication on the host and its subnet; not publicly accessible.
2. If a firewall is configured on the instance, be sure that traffic can flow out of the `docker0` interface to the instance's primary address. For example, to do this with UFW run: `ufw allow in on docker0`. This rule can be added before the `docker0` interface exists, so it is best to do it now, before the Docker installation.

3. Get a domain name for the instance. Using an IP address to access the product is not supported as many systems use TLS and need to verify that the certificate is correct, which can only be done with a hostname at present.

Operational Mode Decision

Terraform Enterprise can store its state in a few different ways, and you'll need to decide which works best for your installation. Each option has a different approach to recovering from failures ([/docs/enterprise/private/reliability-availability.html#recovery-from-failures-1](#)) and should be selected based on your organization's preferences.

Note: This decision should be made before you begin installation, because some modes have additional preflight requirements that are detailed below. The operational mode is selected at install time and cannot be changed once the install is running.

1. **Production - External Services** - This mode stores the majority of the stateful data used by the instance in an external PostgreSQL database as well as an external S3-compatible endpoint or Azure blob storage. There is still critical data stored on the instance that must be managed with snapshots. Be sure to check the PostgreSQL Requirements for information that needs to be present for Terraform Enterprise to work. This option is best for users with expertise managing PostgreSQL or users that have access to managed PostgreSQL offerings like AWS RDS (<https://aws.amazon.com/rds/>).
2. **Production - Mounted Disk** - This mode stores data in a separate directory on the host, with the intention that the directory is configured to store its data on an external disk, such as EBS, iSCSI, etc. This option is best for users with experience mounting performant block storage.
3. **Demo** - This mode stores all data on the instance. The data can be backed up with the snapshot mechanism for restore later. This option is best for initial installation and testing, and is not recommended or supported for true production use.

The decision you make will be entered during setup.

Mounted Disk Guidelines

The mounted disk option provides significant flexibility in how PTFE stores its data. To help narrow down the possibilities, we've provided the following guidelines for mounted disk storage.

Supported Mounted Disk Types

The following are **supported** mounted disk types:

- AWS EBS
- GCP Zonal Persistent Disk
- Azure Disk Storage
- iSCSI
- SAN

- Physically connected disks as in non-cloud hardware

Disk Resizing

Depending on your cloud or storage application, you may need to confirm the disk has been resized to above Private Terraform Enterprise's minimum disk size of 40GB.

For example, with RedHat-flavor (RHEL, CentOS, Oracle Linux) images in Azure Cloud, the storage disk must be resized above the 30GB default after initial boot with fdisk, as documented in the Azure knowledge base article [How to: Resize Linux osDisk partition on Azure](https://blogs.msdn.microsoft.com/linuxonazure/2017/04/03/how-to-resize-linux-osdisk-partition-on-azure/) (<https://blogs.msdn.microsoft.com/linuxonazure/2017/04/03/how-to-resize-linux-osdisk-partition-on-azure/>).

Unsupported Mounted Disk Types

The following are **not** supported mounted disk types:

- NFS
- SMB/CIFS

The supported mounted disk types provide the necessary reliability and performance for data storage and retrieval in PTFE.

If the type of mounted disk you wish to use is not listed, please contact your HashiCorp representative to get clarification on whether that type is supported.

PostgreSQL Requirements

When Terraform Enterprise uses an external PostgreSQL database, the following must be present on it:

- PostgreSQL version 9.4 or greater
- User with all privileges granted on the schemas created and the ability to run "CREATE EXTENSION" on the database
- The following PostgreSQL schemas must be installed into the database: `rails`, `vault`, `registry`

To create schemas in PostgreSQL, the `CREATE SCHEMA` command is used. So to create the above required schemas, the following snippet must be run on the database:

```
CREATE SCHEMA rails;
CREATE SCHEMA vault;
CREATE SCHEMA registry;
```

When providing optional extra keyword parameters for the database connection, note an additional restriction on the `sslmode` parameter is that only the `require`, `verify-full`, `verify-ca`, and `disable` values are allowed.

External Vault Option

If you already manage your own Vault cluster, you can choose to use it in Production operational mode, rather than the default internal Vault provided by PTFE.

Note: This option is also selected at initial installation, and cannot be changed later.

If you will want to use an external Vault cluster when running Terraform Enterprise in production, select that option when initially switching to the Production operational mode. See Externally Managed Vault Cluster (</docs/enterprise/private/vault.html>) for more information on what this option requires.

Private Terraform Enterprise Reference Architecture (Installer)

HashiCorp provides Reference Architectures detailing the recommended infrastructure and resources that should be provisioned in order to support a highly-available Private Terraform Enterprise deployment.

Depending on where you choose to deploy Private Terraform Enterprise, there are different services available to maximise the resiliency of the deployment, for example, most major cloud service providers offer a resilient relational database service offering, removing the need for the customer to manage a complex database cluster/failover architecture. We have taken this into consideration and created a Reference Architecture for the most common deployment configurations, making the most appropriate use of those cloud vendor services.

Reference Architectures

- Amazon Web Services (</docs/enterprise/private/aws-setup-guide.html>)
- Microsoft Azure (</docs/enterprise/private/azure-setup-guide.html>)
- Google Cloud Platform (</docs/enterprise/private/gcp-setup-guide.html>)
- VMware (</docs/enterprise/private/vmware-setup-guide.html>)

Private Terraform Enterprise Reliability & Availability

This section covers details relating to the reliability and availability of Private Terraform Enterprise (PTFE) installations. This documentation may be useful to Customers evaluating PTFE or Operators responsible for installing and maintaining PTFE.

Components

Terraform Enterprise consists of several distinct components that each play a role when considering the reliability of the overall system:

- **Application Layer**

- *TFE Core* - A Rails application at the center of Terraform Enterprise; consists of web frontends and background workers
- *TFE Services* - A set of Go services that provide various pieces of key functionality for Terraform Enterprise
- *Terraform Workers* - A fleet of isolated execution environments that perform Terraform Runs on behalf of Terraform Enterprise users

- **Coordination Layer**

- *Redis* - Used for Rails caching and coordination between TFE Core's web and background workers
- *RabbitMQ* Used for Terraform Worker job coordination

- **Storage Layer**

- *PostgreSQL Database* - Serves as the primary store of Terraform Enterprise's application data such as workspace settings and user settings
- *Blob Storage* - Used for storage of Terraform state files, plan files, configuration, and output logs
- *HashiCorp Vault* - Used for encryption of sensitive data. There are two types of Vault data in PTFE - key material (<https://www.vaultproject.io/docs/concepts/seal.html>) and storage backend data (<https://www.vaultproject.io/docs/configuration/storage/index.html>).
- *Configuration Data* - The information provided and/or generated at install-time (e.g. database credentials, hostname, etc.)

AMI Architecture

In the AMI Architecture, both the **Application Layer** and the **Coordination Layer** execute on a single EC2 instance.

Configuration Data is provided via inputs to the Terraform modules that are published alongside the AMI (<https://github.com/hashicorp/terraform-enterprise-modules>). These inputs are interpolated into an encrypted file on S3. The EC2 instance is granted access to download this file via its IAM instance profile and configured to do so on boot via its User Data script.

This setup allows the instance to automatically reconfigure itself on each boot, making for a consistent story for upgrades and recovery. The instance is launched in an Auto Scaling Group of size one, which facilitates automatic re-launch in the case of instance loss.

The **Storage Layer** is delegated to Amazon services and inherits their respective reliability and availability properties:

- *PostgreSQL Database* - Amazon RDS is configured by default to use a Multi-AZ (<https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html>) deployment, which provides high availability and automated failover of the database instance. Nightly database snapshots are automatically configured and retained for 31 days. The Amazon RDS Documentation (<https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide>Welcome.html>) has much more information about the reliability and availability of this service.
- *Blob Storage* - Amazon S3 is used for all blob storage. The Amazon S3 Documentation (<https://aws.amazon.com/documentation/s3/>) has much more information about the reliability and availability of this service.
- *HashiCorp Vault* - Consul is run on the PTFE EC2 instance and stores all Vault data. Consul data is backed up hourly to S3 alongside of the *Configuration Data* and is set to automatically restore on boot.

Availability During Upgrades

Upgrades for the AMI Architecture are delivered as new AMIs. Switching AMI IDs within the Terraform config that manages your PTFE installation will cause the plan to include replacement of the Launch Configuration and Auto Scaling Group associate with your Terraform Enterprise instance.

Applying this plan will result in the instance being relaunched. It will automatically recover the latest backup of *Configuration Data* from S3 and resume normal operations.

In normal conditions, Terraform Enterprise will be unavailable for less than five minutes as the old instance terminates and the new instance launches and boots the application.

Recovery From Failures

The boot behavior and Auto Scaling Group configuration described above means that the system can automatically recover from any failure that results in loss of the instance. This recovery generally completes in less than five minutes.

The Multi-AZ setup used for RDS protects against failures that affect the Database Instance, and the nightly automated RDS Snapshots provide coverage against data corruption.

The redundancy guarantees of Amazon S3 serve to protect the files that PTFE stores there.

Installer Architecture

This section describes how to set up your PTFE deployment to recover from failures in the various operational modes (demo, mounted disk, external services). The operational mode is selected at install time and can not be changed once the install is running.

The below tables explain where each data type in the Storage Layer is stored and the corresponding snapshot and restore procedure. For the data types that use PTFE's built-in snapshot and restore function, follow these instructions ([/docs/enterprise/private/automated-recovery.html](#)). For the data types that do **not** use the built-in functionality, backup and restore is the responsibility of the user.

Data Location

	Configuration	Vault	PostgreSQL	Blob Storage
Demo	Stored in Docker volumes on instance	Key material is stored in Docker volumes on instance, storage backend is internal PostgreSQL	Stored in Docker volumes on instance	Stored in Docker volumes on instance
Mounted Disk	Stored in Docker volumes on instance	Key material on host in /var/lib/tfe-vault, storage backend is mounted disk PostgreSQL	Stored in mounted disks	Stored in mounted disks
External Services	Stored in Docker volumes on instance	Key material on host in /var/lib/tfe-vault, storage backend is external PostgreSQL	Stored in external service	Stored in external service
External Vault	-	Key material in external Vault with user-defined storage backend	-	-

Backup and Restore Responsibility

	Configuration	Vault	PostgreSQL	Blob Storage
Demo	PTFE	PTFE	PTFE	PTFE
Mounted Disk	PTFE	PTFE	User	User
External Services	PTFE	PTFE	User	User
External Vault	-	User	-	-

Demo

All data (Configuration, PostgreSQL, Blob Storage, Vault) is stored within Docker volumes on the instance.

Snapshot: The built-in Snapshot mechanism can be used to package up all data and store it off the instance. The frequency of automated snapshots can be configured hourly such that the worst-case data loss can be as low as 1 hour.

Restore: If the instance running Terraform Enterprise is lost, the only recovery mechanism in demo mode is to create a new instance and use the builtin Restore mechanism to recreate it from a previous snapshot.

Configure Snapshot and Restore following these instructions ([/docs/enterprise/private/automated-recovery.html](#)).

Mounted Disk

PostgreSQL Database and *Blob Storage* use mounted disks for their data. Backup and restore of those volumes is the responsibility of the user, and is not managed by PTFE's built-in systems.

Vault Data is stored in PostgreSQL and accordingly lives on the mounted disk. As long as the user has restored the mounted disk successfully, the built-in restore mechanism will restore Vault operations in the event of a failure.

Configuration Data for the installation is stored in Docker volumes on the instance. The built-in snapshot mechanism can package up the Configuration data and store it off the instance, and the built-in restore mechanism can recover the configuration data and restore operation in the event of a failure. Configure snapshot and restore by following the automated recovery instructions ([/docs/enterprise/private/automated-recovery.html](#)).

If the instance running Terraform Enterprise is lost, the use of mounted disks means no state data is lost.

External Services

In the External Services mode of the Installer Architecture, the **Application Layer** and **Coordination Layer** execute on a Linux instance, but the **Storage Layer** is configured to use external services in the form of a PostgreSQL server and an S3-compatible Blob Storage.

The maintenance of PostgreSQL and Blob Storage are handled by the user, which includes backing up and restoring if necessary.

Vault Data is stored in PostgreSQL. As long as PostgreSQL has been restored successfully by the user, the built-in restore mechanism will restore Vault operations in the event of a failure.

Configuration Data for the installation is stored in Docker volumes on the instance. The built-in snapshot mechanism can package up the data and store it off the instance, and the built-in restore mechanism can recover the data and restore operation in the event of a failure. Configure snapshot and restore by following the automated recovery instructions ([/docs/enterprise/private/automated-recovery.html](#)).

If the instance running Terraform Enterprise is lost, the use of external services means no state data is lost.

NOTE: Customers running an optional external Vault cluster ([/docs/enterprise/private/vault.html](#)) are responsible for backing up the Vault data and restoring it if necessary.

Availability During Upgrades

Upgrades for the Installer architecture use the Installer dashboard. Once an upgrade has been detected (either online or airgap), the new code is imported. Once ready, all services on the instance are restarted running the new code. The expected downtime is between 30 seconds and 5 minutes, depending on whether database updates have to be applied.

Only application services are changed during the upgrade; data is not backed up or restored. The only data changes that may occur during upgrade are the application of migrations the new version might apply to the *PostgreSQL Database*.

When an upgrade is ready to start the new code, the system waits for all Terraform runs to finish before continuing. Once the new code has started, the queue of runs is continued in the same order.

Private Terraform Enterprise Installation (Installer) - RHEL Install Guide

This install guide is specifically for users of Private Terraform Enterprise installing the product on RedHat Enterprise Linux (RHEL).

Install Recommendations

- RedHat Enterprise Linux version 7.3 or later.
- Docker 1.13.1 (available in RHEL extras), or Docker EE version 17.06 or later. The later versions are not available in the standard RHEL yum repositories.
 - For Docker EE, there are explicit RHEL instructions to follow: <https://docs.docker.com/install/linux/docker-ee/rhel/> (<https://docs.docker.com/install/linux/docker-ee/rhel/>)
 - For Docker from RHEL extras, the following should enable the RHEL extras repository:
 - `yum-config-manager --enable rhel-7-server-extras-rpms`
 - on AWS: `yum-config-manager --enable rhui-REGION-rhel-server-extras`
- A properly configured docker storage backend, either:
 - Devicemapper configured for production usage, according to the Docker documentation:
<https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production> (<https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production>). This configuration requires a second block device available to the system to be used as a thin-pool for Docker. You may need to configure this block device before the host system is booted, depending on the hosting platform.
 - A system capable of using overlay2. This requires at least kernel version 3.10.0-693 and, if XFS is being used, the flag `ftype=1`. The full documentation on this configuration is at:
<https://docs.docker.com/storage/storagedriver/overlayfs-driver/> (<https://docs.docker.com/storage/storagedriver/overlayfs-driver/>)
 - If using Docker from RHEL Extras, storage can be configured using the `docker-storage-setup` command

Note: Using `docker-1.13.1-84.git07f3374.el7.x86_64` will result in an RPC error as well as 502 errors and inability to use the application.

Pinning the Docker Version

If `docker-1.13.1-84.git07f3374.el7.x86_64` is already installed, first run:

```
sudo yum downgrade docker docker-client docker-common docker-rhel-push-plugin
```

Then, restart Docker and ensure the installed version changes to `1.13.1-72.git6f36bd4el7.x86_64`. To pin the version and prevent an inadvertent upgrade, follow this guide (<https://access.redhat.com/solutions/98873>) from RedHat.

Mandatory Configuration

If you opt to use Docker from RHEL extras, then you must make a change to its default configuration to avoid hitting an out of memory bug.

1. Open `/usr/lib/systemd/system/docker.service`
2. Remove the line that contains `--authorization-plugin=rhel-push-plugin`
3. Run `systemctl daemon-reload && systemctl restart docker`
4. Run `docker info 2> /dev/null | grep Authorization` to verify that there are no authorization plugins active. If nothing is printed, your installation is properly configured. If anything is printed, please contact support for further assistance.

FAQ:

Can I use the Docker version in rpm-extras?

Sure! Just be sure to have at least 1.13.1 and authorization plugins disabled.

When I run the installer, it allows me to download and install Docker CE on RedHat. Can I use that?

Yes, Docker CE is compatible with the current installer. However, it is not directly supported by RedHat (<https://access.redhat.com/articles/2726611>). You still need to be sure that the storage backend is configured properly as by default, Docker will be using devicemapper in loopback, an entirely unsupported mode.

Can an installation where `docker info` says that I'm using devicemapper with a loopback file work?

No. This is an installation that docker provides as sample and is not supported by Private Terraform Enterprise due to the significant instability in it. Docker themselves do not suggest using this mode:

<https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-loop-lvm-mode-for-testing>
<https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-loop-lvm-mode-for-testing>

How do I know if an installation is in devicemapper loopback mode?

Run the command `sudo docker info | grep dev/loop`. If there is any output, you're in devicemapper loopback mode. Docker may also print warning about loopback mode when you run the above command, which is another indicator.

Private Terraform Enterprise Installer - Simplified Migration

This document outlines a simplified procedure for migrating from the AMI-based Private Terraform Enterprise (PTFE) to the Installer-based PTFE that is suitable if the original Terraform modules were not modified, or the modifications were minimal.

Terraform State

To run this procedure, you'll need the Terraform state file used to create the AMI-based installation. Additionally, we strongly suggest you back up this state file before proceeding with this process, in the event that you need to revert back to the AMI.

Backup

Before beginning, it's best to create an additional backup of your RDS database. This will allow you to roll back the data and continue to use the AMI if necessary.

To create an RDS backup, go to the Amazon RDS Instances (<https://console.aws.amazon.com/rds/home?region=us-east-1#dbinstances:>). You may need to change the region that you are viewing in order to see your PTFE instance. Once you find it, click on the instance name. On the next page, select **Instance Actions** and then **Take snapshot**. On the **Take DB Snapshot** page, enter a name for the snapshot such as **Pre-Installer Migration**, and then click **Take Snapshot**.

The snapshot will take a little while to create. After it has finished, you can continue with the rest of the migration steps.

Reverting back to the AMI

To revert to the AMI after running the migration script:

- If you've already manipulated the state file to move the resources, you'll need to restore the original state file.
- With your original Terraform state file in place, return to the Amazon RDS Snapshots (<https://console.aws.amazon.com/rds/home?region=us-east-1#db-snapshots:>) and find the snapshot you created before migrating. Click on the snapshot and note its **ARN** value. Open your **.tfvars** file and add `db_snapshot = "arn-value-of-snapshot"`.
- Run `terraform apply` to make sure everything is set up. This will result in a new RDS instance being built against the snapshot.
- If the EC2 instance is still running from the migration process, run `shutdown` on it to get a new instance created.
- Return to the original hostname used for the cluster.

Preflight

Terraform Variables and State

To use this simplified procedure, you'll need the `.tfvars` file and `.tfstate` file used to deploy the AMI-based PTFE instance.

These files will be used against a new Terraform module set to change just the PTFE software and leave the state storage, such as RDS and S3, in place.

Preparing Modules

Download the new Terraform modules (<https://github.com/hashicorp/ptfe-migration-terraform>) into a local directory.

Change to that directory, and place the `.tfvars` and `.tfstate` files for the AMI-based instance in the directory.

The rest of the steps will be executed from this directory.

Data Migration

The AMI instance must be prepared before any of the new Terraform configuration is used. Use SSH to access the instance and download the migration tool by running:

```
$ curl -O https://s3.amazonaws.com/hashicorp-ptfe-data/migrator  
$ chmod a+x migrator
```

Next, run the migrator tool to move some data off the AMI and into your PostgreSQL installation.

```
$ sudo ./migrator
```

The migrator will prompt you to run again with the `-confirm` flag to confirm that you wish to run this step, because it shuts down operations on the current instance. The shutdown is required to ensure consistency of the data. To confirm the shutdown and proceed, run:

```
$ sudo ./migrator -confirm
```

Next, the migration will move the Vault data into PostgreSQL and output the encryption password value that you need to specify to use this Terraform module. The password output will be similar to:

```
Provide this value as the encryption password in the migration tfvars file:  
jIQzYXtNZNKfkyHbj2WHqTkrSQPkeX8gCrfPAkJHZ5s
```

The value of the encryption password (`jIQzYXtNZNKfkyHbj2WHqTkrSQPkeX8gCrfPAkJHZ5s` here) will be different for your run of the migrator tool. Copy the provided value and add it to your `.tfvars` file like this:

```
encryption_password = "jIQzYXtNZNKfkyHbj2WHqTkrSQPkeX8gCrfPAkJHZ5s"
```

License File

Place your Terraform Enterprise license file (ending in `.rli`) on your local disk, and specify the path to it in the `.tfvars` file like this:

```
license_file = "/path/to/license.rli"
```

where the provided path is the path on the local disk. If you put it in the current directory, you can specify just the name and omit the path.

Initialize

You're now ready to initialize the new terraform modules:

```
terraform init ./terraform/aws-standard
```

Apply the Changes

Once the initialization has succeeded, run

```
terraform plan ./terraform/aws-standard
```

to see the changes that will be made. Double-check that they are correct before continuing. There should not be any changes made to your `aws_db_instance` or `aws_s3_bucket`. If any changes to these resources are shown in the plan, pause the migration process and contact support ([/docs/enterprise/private/faq.html#support-for-private-terraform-enterprise](#)).

When you're satisfied with the plan, apply it:

```
terraform apply ./terraform/aws-standard
```

The apply will remove the Terraform Enterprise AMI instance and boot an instance running Terraform Enterprise under the installer framework in its place, configured with all the values provided by the existing Terraform state, along with the license information and encryption password you added.

When the `terraform apply` completes, the instance will be fully migrated!

Availability

This module runs with the same availability as the AMI did. It is protected against instance loss by the Auto Scaling Group and boots the new instance automatically without the need for further configuration.

Software Upgrades

This module installs the latest version of the Terraform Enterprise software automatically on boot, so an easy way to upgrade the application is to shut down the instance. When the Auto Scaling Group boots a new instance, it will install the new software and resume operation.

Alternately, you can upgrade the application from the management console, available on port 8800 of the instance, following the standard upgrade process for the installer ([/docs/enterprise/private/upgrades.html](#)).

Installer Dashboard

As part of the migration, the ELB is modified to provide access to the installer dashboard on port 8800. The dashboard is configured for access with an autogenerated password that is available in the Terraform state; it is one of the outputs generated when applying the new Terraform modules.

Upgrading

This section explains how to upgrade Private Terraform Enterprise to a new version. Learn more about availability during upgrades here (</docs/enterprise/private/reliability-availability.html#availability-during-upgrades-1>).

Online

1. From the installer dashboard (<https://<TFE HOSTNAME>:8800/dashboard>), click the "Check Now" button; the new version should be recognized.
2. Click "View Update".
3. Review the release notes and then click "Install Update".

Airgapped

1. Determine the update path where the installer will look for new .airgap packages. You can do this from the console settings of your instance (<https://<TFE HOSTNAME>:8800/console/settings>) in the field `Update Path`.
2. Download the new .airgap package onto the instance and put it into the `Update Path` location.
3. From the installer dashboard (<https://<TFE HOSTNAME>:8800/dashboard>) click the "Check Now" button; the new version should be recognized.
4. Click "View Update".
5. Review the release notes and then click "Install Update".

Private Terraform Enterprise Externally Managed Vault

For enhanced security, Private Terraform Enterprise (PTFE) can be configured to use an external Vault cluster rather than the internal Vault instance. Within PTFE, Vault is used to encrypt sensitive information such as variables and states.

Warning: This is only recommended if you currently run your own Vault cluster in production. Choosing this option means you assume full responsibility for how the Vault cluster is managed; for example, how it is sealed and unsealed, replicated, etc.

Setup

Note: The external Vault option must be selected at initial installation, and cannot be changed later. Do not attempt to migrate an existing Terraform Enterprise instance between internal and external Vault options.

Use the following as a guide to configure an external Vault instance:

1. Enable AppRole: `vault auth enable approle`
2. Enable transit: `vault secrets enable transit`
3. Install the `ptfe` policy (See below for policy): `vault policy write ptfe ptfe.hcl`
4. Create an AppRole instance: `vault write auth/approle/role/ptfe policies="ptfe"`.
5. Retrieve the AppRole `role_id`: `vault read auth/approle/role/ptfe/role-id`
6. Retrieve the AppRole `secret_id`: `vault write -f auth/approle/role/ptfe/secret-id`
7. Input the address of the vault cluster, `role_id`, and `secret_id` during the install process

role_id and secret_id

The `role_id` and `secret_id` values created during configuration will be input during the PTFE installation along with the Vault cluster URL such as `https://vault.mycompany.com:8200`. If you use SSL to access the Vault cluster, the certificate must be trusted by PTFE. That means the issuer of the certificate is either globally trusted or you have provided the certificate for the issuer in the "Certificate Authority bundle" section of the installer configuration.

Vault Policy

Vault utilizes policies to restrict portions of the data. See the official Vault docs (<https://www.vaultproject.io/docs/concepts/policies.html>) for a more extensive overview.

The following policy is required for the correct operations of PTFE:

```
path "auth/approle/login" {
    capabilities = ["create", "read"]
}

path "sys/renew/*" {
    policy = "write"
}

path "auth/token/renew/*" {
    policy = "write"
}

path "transit/encrypt/atlas_*" {
    capabilities = ["create", "update"]
}

path "transit/decrypt/atlas_*" {
    capabilities = ["update"]
}

path "transit/encrypt/archivist_*" {
    capabilities = ["create", "update"]
}

# For decrypting datakey ciphertexts.
path "transit/decrypt/archivist_*" {
    capabilities = ["update"]
}

# To upsert the transit key used for datakey generation.
path "transit/keys/archivist_*" {
    capabilities = ["create", "update"]
}

# For performing key derivation.
path "transit/datakey/plaintext/archivist_*" {
    capabilities = ["update"]
}

# For health checks to read the mount table.
path "sys/mounts" {
    capabilities = ["read"]
}
```

Private Terraform Enterprise VMware Reference Architecture

This document provides recommended practices and a reference architecture for HashiCorp Private Terraform Enterprise (PTFE) implementations on VMware.

Required Reading

Prior to making hardware sizing and architectural decisions, read through the installation information available for PTFE (<https://www.terraform.io/docs/enterprise/private/install-installer.html>) to familiarise yourself with the application components and architecture. Further, read the reliability and availability guidance (<https://www.terraform.io/docs/enterprise/private/reliability-availability.html>) as a primer to understanding the recommendations in this reference architecture.

Infrastructure Requirements

Depending on the chosen operational mode (<https://www.terraform.io/docs/enterprise/private/preflight-installer.html#operational-mode-decision>), the infrastructure requirements for PTFE range from a single virtual machine for demo or proof of concept installations, to multiple virtual machines hosting the Terraform Enterprise application, PostgreSQL, and external Vault servers for a stateless production installation.

This reference architecture focuses on the “Production - External Services” operational mode. If you require all of the pTFF infrastructure to be on-prem, you can either deploy S3-compatible storage such as ceph (<http://ceph.com/>) or select the “Production - Mounted Disk” option. This option will require you to specify the local path for the storage where PostgreSQL data will be stored and where the data typically written to S3 (blob) will be stored. The assumption is this local path is a mounted disk from either a SAN or NAS device (or some other replicated storage), allowing for rapid recovery or failover. More information about the Mounted Disk option can be found at the end of this document.

The following table provides high-level server recommendations as a guideline. Please note, thick provision, lazy zeroed storage is preferred. Thin provisioned is only recommended if you are using an external PostgreSQL database as well as an external vault server. Using thin provisioned disks when using the internal database or vault may result in serious performance issues.

PTFE Servers

Type	CPU Sockets	Total Cores*	Memory	Disk
Minimum	2	2	8 GB RAM	40GB
Recommended	2	4	16-32 GB RAM	40GB

Note: Per VMWare’s recommendation, always allocate the least amount of CPU necessary. HashiCorp recommends starting with 2 CPUs and increasing if necessary.

Hardware Sizing Considerations

- The minimum size would be appropriate for most initial production deployments, or for development/testing environments.
- The recommended size is for production environments where there is a consistent high workload in the form of concurrent terraform runs.
- Please monitor the actual CPU utilization in vCenter before making the decision to increase the CPU allocation.

PostgreSQL Database

Type	CPU Sockets	Total Cores	Memory	Storage
Minimum	2	2 core	8 GB RAM	50GB
Recommended	2	4-8 core	16-32 GB RAM	50GB

Hardware Sizing Considerations

- The minimum size would be appropriate for most initial production deployments, or for development/testing environments.
- The recommended size is for production environments where there is a consistent high workload in the form of concurrent terraform runs.
- Please monitor the actual CPU utilization in vCenter before making the decision to increase the CPU allocation.

Object Storage (S3)

An S3 Standard (<https://aws.amazon.com/s3/storage-classes/>) bucket, or compatible storage, must be specified during the PTFE installation for application data to be stored securely and redundantly away from the virtual servers running the PTFE application. This object storage must be accessible via the network to the PTFE virtual machine. Vault is used to encrypt all application data stored in this location. This allows for further server-side encryption (<https://docs.aws.amazon.com/AmazonS3/latest/dev/serv-side-encryption.html>) by S3 if required by your security policy.

Recommended object storage solutions are AWS S3, Google Cloud storage, Azure blob storage. Other options for S3-compatible storage are minio (<https://www.minio.io/>) and ceph (<https://ceph.com/>), among many others. Please feel free to reach out to support (<https://www.hashicorp.com/support>) with questions.

Vault Servers

In order to provide a fully stateless application instance, PTFE must be configured to speak with an external Vault server/cluster. This reference architecture assumes that a highly available Vault cluster is accessible at an endpoint the PTFE servers can reach.

Other Considerations

Network

To deploy PTFE on VMWare you will need to create new or use existing networking infrastructure that has access to not only the S3 bucket, the PostgreSQL instance, and the Vault server, but also any infrastructure you expect to manage with the PTFE server. If you plan to use your PTFE server to manage or deploy AWS, you will need to make sure the PTFE server has unimpeded access to AWS. The same goes for any other cloud or datacenter the server will need to connect with.

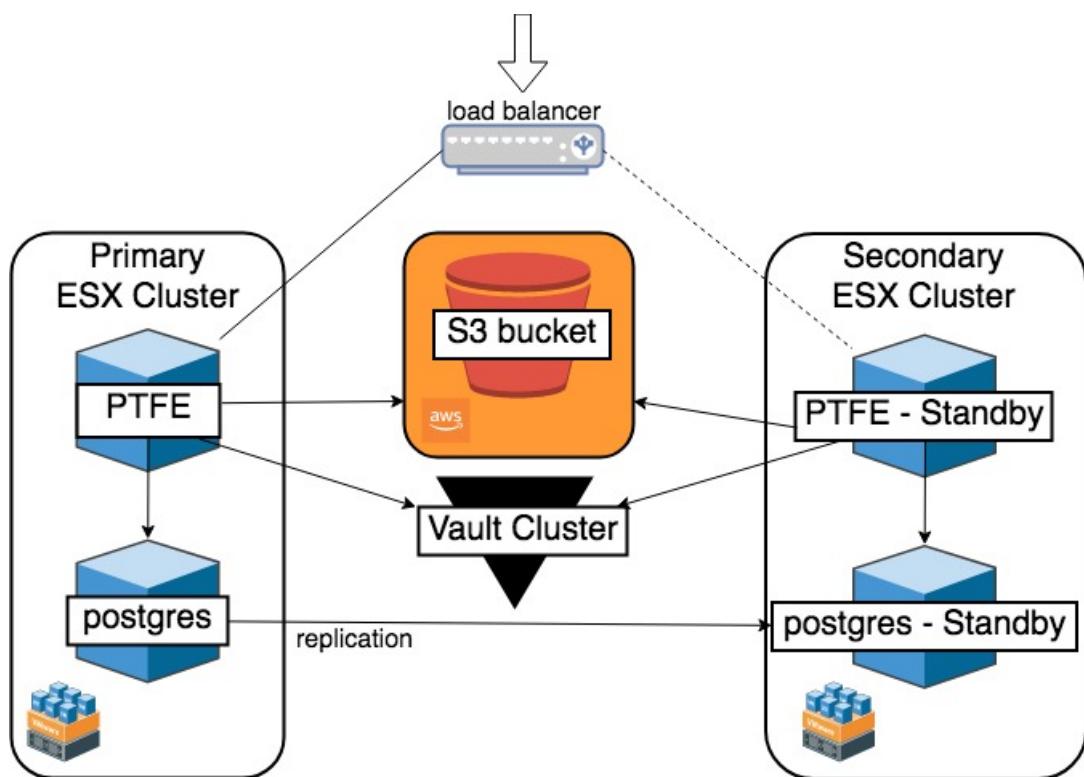
DNS

The fully qualified domain name should resolve to the IP address of the virtual machine using an A record. Creating the required DNS entry is outside the scope of this guide.

SSL/TLS

A valid, signed SSL/TLS certificate is required for secure communication between clients and the PTFE application server. Requesting a certificate is outside the scope of this guide. You will be prompted for the public and private certificates during installation.

Infrastructure Diagram



The above diagram shows the infrastructure components at a high-level.

Storage Layer

The Storage Layer is composed of multiple service endpoints (datastores, object storage, Vault) all configured with or benefitting from inherent resiliency provided by ESX and your storage provider, or assumed resiliency provided by a well-architected deployment (in the case of Vault).

- More information about S3 Standard (<https://aws.amazon.com/s3/storage-classes/>)
- More information about highly available Vault deployments (<https://www.vaultproject.io/guides/operations/vault-ha-consul.html>)

Infrastructure Provisioning

The recommended way to deploy PTFE for production is through use of a Terraform configuration that defines the required resources, their references to other resources and dependencies. An example Terraform configuration (<https://github.com/hashicorp/private-terraform-enterprise/tree/master/examples/vmware>) is provided to demonstrate how these resources can be provisioned and how they interrelate. This Terraform configuration will require you to know several pieces of information and have sufficient access to vCenter to deploy a VM from template.

Normal Operation

Component Interaction

The PTFE application is connect to the PostgreSQL database via the PostgreSQL URL.

The PTFE application is connected to object storage via the API endpoint for the defined storage location and all object storage requests are routed to the highly available infrastructure supporting the storage location.

The PTFE application is connected to the Vault cluster via the Vault cluster endpoint URL.

Monitoring

While there is not currently a monitoring guide for PTFE, information around logging (<https://www.terraform.io/docs/enterprise/private/logging.html>), diagnostics (<https://www.terraform.io/docs/enterprise/private/diagnostics.html>) as well as reliability and availability (<https://www.terraform.io/docs/enterprise/private/reliability-availability.html>) can be found on our website.

Upgrades

Please reference the upgrade instructions (<https://www.terraform.io/docs/enterprise/private/upgrades.html>) in the documentation.

High Availability

Failure Scenarios

VMWare hypervisor provides a high level of resilience in various cases of failure (at the server hardware layer through vMotion, at the storage layer through RDS, and at the network layer through virtual distributed networking.) In addition, having ESX failover to a DR datacenter provides recovery in the case of a total data center outage.

Single ESX Server Failure

In the event of a server outage ESX will vMotion the PTFE virtual machine to a functioning ESX host. This typically does not result in any visible outage to the end-user.

Failure by Application Tier

PTFE Servers

Through deployment of two virtual machines in different ESX clusters, the PTFE Reference Architecture is designed to provide improved availability and reliability. Should the *PTFE-main* server fail, it can be recovered, or traffic can be routed to the *PTFE-standby* server to resume service when the failure is limited to the PTFE server layer. The load balancer should be manually updated to point to the stand-by PTFE VM after services have been started on it in the event of a failure.

PostgreSQL Database

Using a PostgreSQL cluster will provide fault tolerance at the database layer. Documentation on how to deploy a PostgreSQL cluster can be found on the PostgreSQL documentation page (<https://www.postgresql.org/docs/9.5/static/creating-cluster.html>).

Object Storage

Using AWS S3 as an external object store leverages the highly available infrastructure provided by AWS. S3 buckets are replicated to all Availability Zones within the region selected during bucket creation. From the AWS website:

Amazon S3 runs on the world's largest global cloud infrastructure, and was built from the ground up to deliver a customer promise of 99.99999999% of durability. Data is automatically distributed across a minimum of three physical facilities that are geographically separated within an AWS Region. (source (<https://aws.amazon.com/s3/>)

Other cloud providers (Azure, GCP) also provide highly available storage. If you choose to utilize an on-premises storage solution, such as ceph, it will be your responsibility to configure HA as required by your implementation.

Vault Servers

For the purposes of this guide, the external Vault cluster is expected to be deployed and configured in line with the HashiCorp Vault Enterprise Reference Architecture (<https://www.vaultproject.io/guides/operations/reference-architecture.html>). This would provide high availability and disaster recovery support, minimising downtime in the event of an outage.

Disaster Recovery

PTFE Servers

The assumption of this architecture is that ESX replication to a DR site is configured and validated. In the event of a DR the Primary PTFE virtual machine should be recovered and powered on. Services are configured to start on boot, so the secondary virtual machine should only be powered on if there is an issue with the primary. If the DR site will be active for an extended amount of time, it's recommended to power up the standby server first, then stop the services via the console, then power up the primary server.

PostgreSQL Database

Backup and recovery of PostgreSQL will vary based on your implementation. Backup and recovery of PostgreSQL is not covered in this document. We do recommend regular database snapshots.

Object Storage

Recovery is made available via object versioning (<https://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html>) on AWS, as well as Google Cloud Storage (<https://cloud.google.com/storage/docs/object-versioning>) and Azure Storage Services (<https://docs.microsoft.com/en-us/rest/api/storageservices/versioning-for-the-azure-storage-services>). Ceph also supports bucket versioning (<http://docs.ceph.com/docs/master/radosgw/s3/bucketops/#enable-suspend-bucket-versioning>).

Vault Servers

The recommended Vault Reference Architecture uses Consul for storage. Consul provides the underlying snapshot functionality (<https://www.consul.io/docs/commands/snapshot.html>) to support Vault backup and recovery.

Production - Mounted Disk

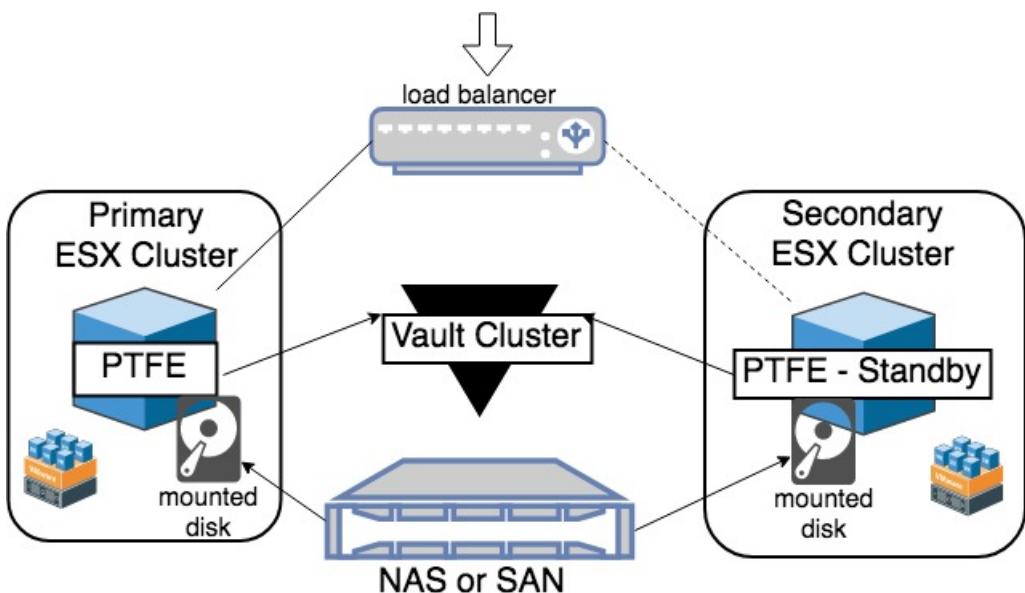
Normal Operation

The PostgreSQL database will be run in a local container and data will be written to the specified path (which should be a mounted storage device, replicated and/or backed up frequently.)

State and other data that would otherwise be written to S3 will be written to the specified local path (which should be a mounted storage device, replicated and/or backed up frequently.)

The PTFE application is connected to the Vault cluster via the Vault cluster endpoint URL.

Infrastructure Diagram



High Availability

Failure Scenarios

VMWare hypervisor provides a high level of resilience in various cases of failure (at the server hardware layer through vMotion, at the storage layer through RDS, and at the network layer through virtual distributed networking.) In addition, having ESX failover to a DR datacenter provides recovery in the case of a total data center outage.

Single ESX Server Failure

In the event of a server outage ESX will vMotion the PTFE virtual machine to a functioning ESX host. This typically does not result in any visible outage to the end-user.

Failure by Application Tier

PTFE Servers

Through deployment of two virtual machines in different ESX clusters, the PTFE Reference Architecture is designed to provide improved availability and reliability. Should the *PTFE-main* server fail, it can be recovered, or traffic can be routed to the *PTFE-standby* server to resume service when the failure is limited to the PTFE server layer. The load balancer should be manually updated to point to the stand-by PTFE VM after services have been started on it in the event of a failure.

PostgreSQL Database

When running in mounted storage mode the PostgreSQL server runs inside a Docker container. If the PostgreSQL service fails a new container should be automatically created. However, if the service is hung, or otherwise fails without triggering a new container deployment, the pTFE server should be stopped and the standby server started. All PostgreSQL data will have been written to the mounted storage and will then be accessible on the standby node.

Object Storage

The object storage will be stored on the mounted disk and the expectation is that the NAS or SAN or other highly available mounted storage is fault tolerant and replicated or has fast recovery available. The configuration of the storage device is not covered in this document. For more information about highly available storage please see your storage vendor.

Vault Servers

For the purposes of this guide, the external Vault cluster is expected to be deployed and configured in line with the HashiCorp Vault Enterprise Reference Architecture (<https://www.vaultproject.io/guides/operations/reference-architecture.html>). This would provide high availability and disaster recovery support, minimising downtime in the event of an outage.

Disaster Recovery

PTFE Servers

The assumption of this architecture is that ESX replication to a DR site is configured and validated. In the event of a DR the Primary PTFE virtual machine should be recovered and powered on. Services are configured to start on boot, so the secondary virtual machine should only be powered on if there is an issue with the primary. If the DR site will be active for an extended amount of time, it's recommended to power up the standby server first, then stop the services via the console, then power up the primary server.

PostgreSQL Database

The PostgreSQL data will be written to the mounted storage. The expectation is that the storage server is replicated or backed up offsite and will be made available to the server in the event of a DR.

Object Storage

Object storage will be written to the mounted storage. The expectation is that the storage server is replicated or backed up offsite and will be made available to the server in the event of a DR.

Vault Servers

The recommended Vault Reference Architecture uses Consul for storage. Consul provides the underlying snapshot functionality (<https://www.consul.io/docs/commands/snapshot.html>) to support Vault backup and recovery.

Private Module Registry

Terraform Enterprise (TFE)'s private module registry helps you share Terraform modules (</docs/modules/index.html>) across your organization. It includes support for module versioning, a searchable and filterable list of available modules, and a configuration designer to help you build new workspaces faster.

By design, the private module registry works much like the public Terraform Registry (</docs/registry/index.html>). If you're already used the public registry, TFE's registry will feel familiar.

Note: Currently, the private module registry works with all supported VCS providers except Bitbucket Cloud.

Using the Configuration Designer

Terraform Enterprise (TFE)’s private module registry includes a configuration designer, which can help you spend less time writing boilerplate code in a module-centric Terraform workflow.

The configuration designer is sort of like interactive documentation for your private modules, or very advanced autocomplete. It results in the same Terraform code you would have written in your text editor, but saves time by automatically discovering variables and searching module and workspace outputs for reusable values.

Workflow Summary

The configuration designer lets you outline a configuration for a new workspace by choosing any number of private modules. It then lists those modules’ variables as a fillable HTML form, with a helper interface for finding interpolatable values.

Once you finish, the designer returns the text of a `main.tf` configuration, which you must copy and paste to create a new VCS repo for workspaces. The designer does not automatically create any repos or workspaces; it’s only a shortcut for writing Terraform code faster.

Accessing the Configuration Designer

Navigate to the modules list with the top-level “Modules” button, then click the “+ Design Configuration” button in the upper right.



This brings you to the “Select Modules” page.

Adding Modules

[Modules](#) / Configuration Designer

[Select Modules](#) > Set Variables > Verify > Publish

[Next »](#)

 [Providers](#)

ADD MODULES TO WORKSPACE

app PRIVATE

Terraform module for app on provider pinode.

[Details](#) [Add Module](#)

PINODE Version 0.1.5

ecs PRIVATE

Terraform module for ecs on provider aws.

[Details](#) [Add Module](#)

AWS Version 1.0.7

SELECTED MODULES 2

app PRIVATE Version 0.1.5

[Details](#) [Remove](#)

dns PRIVATE Version 0.1.3

[Details](#) [Remove](#)

The left side of the "Select Modules" page has a filterable and searchable list of your organization's private modules. Choose any number of modules you want to add to your configuration.

When you click a module's "Add Module" button, it appears in the "Selected Modules" list on the right side of the page.

Setting Versions

By default, selecting a module adds its most recent version to the configuration. You can specify a different version by clicking the module's version number in the "Selected Modules" list on the right, which reveals a drop-down menu of available versions.

app PRIVATE

Terraform module for app on provider pinode.

[Details](#) [Add Module](#)

PINODE Version 0.1.5

ecs PRIVATE

Terraform module for ecs on provider aws.

[Details](#) [Add Module](#)

Selected Modules 2

app PRIVATE Version 0.1.5

[Details](#) [Remove](#)

dns PRIVATE Version 0.1.3

[Details](#) [Remove](#)

Version 0.1.0

Version 0.1.1

Version 0.1.2

Version 0.1.3

Version 0.1.4

Setting Variables

When you finish selecting modules, click the "Next »" button in the upper right to go to the "Set Variables" page.

SELECT MODULE TO CONFIGURE

app PRIVATE	Configured
Details	Configure

dns PRIVATE	Configured
Details	Configure

CONFIGURE VARIABLES

hostname REQUIRED
The hostname to configure
testzone.example.com Deferred

load_balancer_id REQUIRED
The ID of the loadbalancer instance to send traffic to
tertiary_bal Deferred

pinode_access_key_id REQUIRED
The PiNode IAM key id for provisioning
Deferred: variable must be set at runtime Deferred

The left side of this page lists your chosen modules, and the right side lists all variables for the currently selected module. You can freely switch between modules without losing your work; click a module's "Configure" button to switch to its variable list.

Each variable is labeled as required or optional. Once you've set a value for all of a module's required variables, its "Configure" button changes to a green "Configured" button; when all modules are configured, you can use the "Next" button in the upper right to display the finished configuration.

Interpolation Searching

Variable values can be literal strings, or can interpolate other values. When you start typing an interpolation token (`$()`), the designer displays a help message. As you continue typing, it searches the available outputs in your other selected modules, as well as workspace-level outputs from other workspaces. You can select one of these search results, or type a full name if you need to reference a value TFE doesn't know about.

SELECT MODULE TO CONFIGURE

app PRIVATE	Configure
Details	Configure

dns PRIVATE	Configure
Details	Configure

CONFIGURE VARIABLES

app_name REQUIRED
The service name for this app
\$(Deferred

Type **module.** to reference the output of another module in your configuration or type **data.** to reference the output from another Terraform Enterprise workspace you have access to.

enter value or type \$(to search variables Deferred

Deferring Variables

Some variables should be set by the author of a configuration (as static or interpolated values), but others should be set by the configuration's user. If you want to delegate a module variable to users, you can select its "Deferred" checkbox, which ties its value to a new top-level Terraform variable with no default value. Anyone creating a workspace from your configuration will have to provide a value for that variable.

The Output Configuration

When you've finished setting variables, click the "Next" button to view the completed output of the configuration designer.

The screenshot shows the Hashicorp Terraform Configuration Designer interface. At the top, there's a navigation bar with the Hashicorp logo, dropdown menus for 'hashicorp' (with 'Workspaces' and 'Modules' options), 'Documentation' and 'Status', and a user profile icon. Below the navigation is a breadcrumb trail: 'Modules / Configuration Designer'. A progress bar at the top right indicates the steps: 'Select Modules' (green), 'Set Variables' (yellow), 'Verify' (blue, currently active), and 'Publish' (grey). To the right of the progress bar is a green 'Next >' button. The main content area is titled 'Terraform Configuration' and contains a large text area with the following Terraform code:

```
1 //-----
2 // Variables
3 variable "app_pinode_access_key_id" {}
4 variable "app_pinode_secret_access_key" {}
5 variable "dns_pinode_access_key_id" {}
6 variable "dns_pinode_secret_access_key" {}
7
8 //-----
9 // Modules
10 module "app" {
11   source = "ryan.tfe.zone/hashicorp/app/pinode"
12   version = "0.1.5"
13
14   app_name = "example_app"
15   pinode_access_key_id = "${var.app_pinode_access_key_id}"
```

This page has a large textarea that contains a Terraform configuration. You must copy and paste this code into your text editor, save it as the `main.tf` file in a new directory, and commit it to version control to enable creating TFE workspaces with the configuration. You can also make arbitrary changes to the code, like adding non-module resources or public Terraform Registry modules.

After you click "Next" again and click "Done," TFE will discard the configuration you've created. Text from previous configuration designer runs is not preserved within TFE.

Publishing Modules to the Terraform Enterprise Private Module Registry

Note: Currently, the private module registry works with all supported VCS providers except Bitbucket Cloud.

Terraform Enterprise (TFE)'s private module registry lets you publish Terraform modules to be consumed by users across your organization. It works much like the public Terraform Registry (</docs/registry/index.html>), except that it uses your configured VCS integrations (</docs/enterprise/vcs/index.html>) instead of requiring public GitHub repositories.

Only members of the "owners" team can publish new modules. Once a module is published, the ability to release new versions is managed by your VCS provider.

API: See the Registry Modules API (</docs/enterprise/api/modules.html>). Note that the API also supports publishing modules without using a VCS repo as the source, which is not possible via the UI.

Workflow Summary

The private module registry is designed to be as automatic as possible, so it defers to your VCS provider for most management tasks. The only manual tasks are adding a new module and deleting versions.

After configuring at least one connection to a VCS provider (</docs/enterprise/vcs/index.html>), you can publish a new module by specifying a properly formatted VCS repository (one module per repo, with an expected name and tag format; see below for details). The registry automatically detects the rest of the information it needs, including the module's name and its available versions.

To release a new version of an existing module, push a new tag to its repo. The registry updates automatically.

Consumers of a module don't need access to its source repository, even when running Terraform from the command line; the registry handles downloads, and uses TFE's API tokens to control access.

Preparing a Module Repository

Since the registry relies on VCS repositories for most of its data, you must ensure your module repositories are in a format it can understand. A module repository must meet all of the following requirements before you can add it to the registry:

- **Available in a connected VCS provider.** The repository must be in one of your configured VCS providers (</docs/enterprise/vcs/index.html>), and TFE's VCS user account must have admin access to the repository. (Since the registry uses webhooks to import new module versions, it needs admin access to create those webhooks.)
- **One module per repository.** The registry cannot use combined repositories with multiple modules.
- **Named `terraform-<PROVIDER>-<NAME>`.** Module repositories must use this three-part name format, where `<NAME>` reflects the type of infrastructure the module manages and `<PROVIDER>` is the main provider where it creates that infrastructure. The `<NAME>` segment can contain additional hyphens. Examples: `terraform-google-vault` or `terraform-aws-ec2-instance`.
- **Standard module structure.** The module must adhere to the standard module structure

(/docs/modules/create.html#standard-module-structure). This allows the registry to inspect your module and generate documentation, track resource usage, and more.

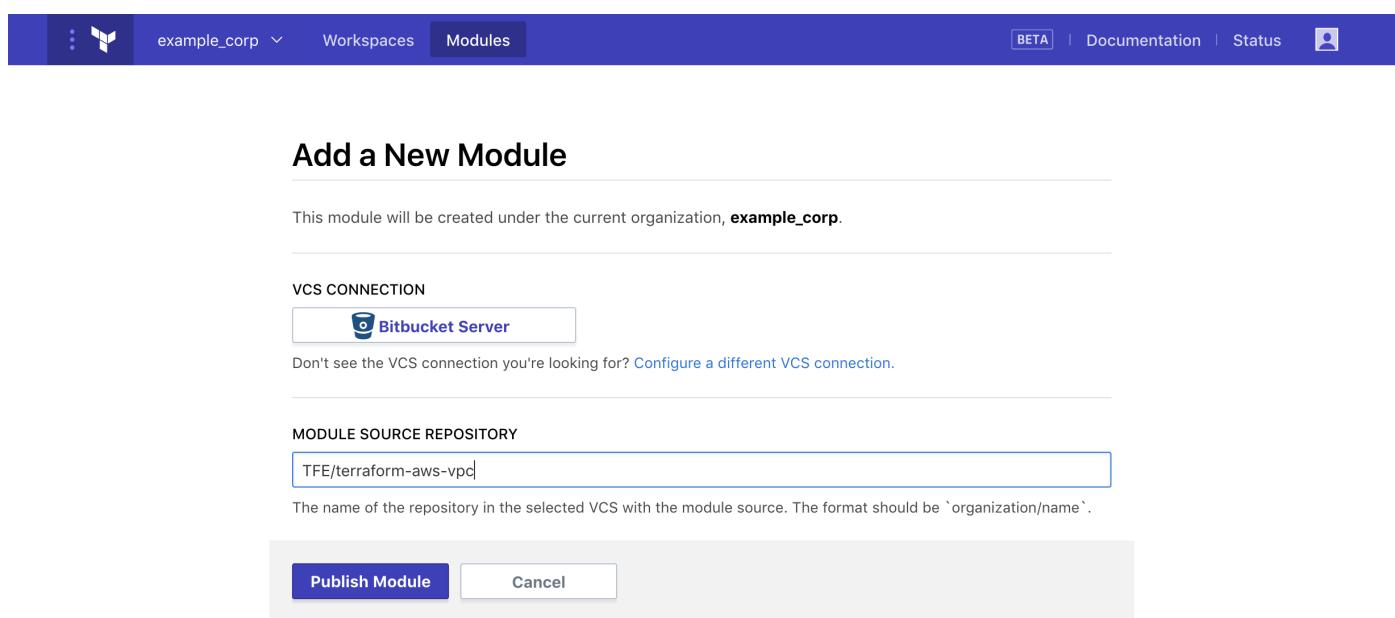
- **x.y.z tags for releases.** The registry uses tags to identify module versions. Release tag names must be a semantic version (<http://semver.org>), which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2. To publish a module initially, at least one release tag must be present. Tags that don't look like version numbers are ignored.

Publishing a New Module

To publish a module, navigate to the modules list with the "Modules" button and click the "+ Add Module" button in the upper right.



This brings you to the "Add a New Module" page, which has a text field and at least one VCS provider button.



If you have multiple VCS providers configured, use the buttons to select one. In the text field, enter the name of the repository for the module you're adding. Then click the "Publish Module" button.

Note: The name you type into the repo field will usually be something like hashicorp/terraform-aws-vpc or INFRA/terraform-azure-appserver. Module repo names use a terraform-<PROVIDER>-<NAME> format, and VCS providers use <NAMESPACE>/<REPO NAME> strings to locate repositories. (For most providers the namespace is an organization name, but Bitbucket Server uses project keys, like INFRA.)

TFE will display a loading page while it imports the module versions from version control, and will then take you to the new module's details page. On the details page you can view available versions, read documentation, and copy a usage example.



Waiting for module **vpc** to become ready...

This page will automatically refresh.

The screenshot shows the Terraform Registry interface. At the top, there's a navigation bar with a logo, the workspace name "example_corp", a "Modules" tab (which is selected), and links for "BETA", "Documentation", "Status", and a user icon. Below the navigation is a breadcrumb trail: "Modules / example_corp / vpc / 2.0.0". To the right of the breadcrumb are two buttons: "Delete Module" and "Versions".

The main content area has a "AWS" tag. It displays the following information:

- Published a few seconds ago
- Provisions: 0
- Source: <http://bitbucket-server:8080/projects/TFE/repos/terraform-aws-vpc/browse>

Below this are links for "Readme", "Inputs (32)", "Outputs (23)", "Dependencies (0)", and "Resources (27)".

A section titled "AWS VPC Terraform module" contains a brief description: "Terraform module which creates VPC resources on AWS." It also lists supported resource types:

- [VPC](#)
- [Subnet](#)
- [Route](#)
- [Route table](#)
- [Internet Gateway](#)
- [NAT Gateway](#)
- [VPN Gateway](#)
- [VPC Endpoint](#) (S3 and DynamoDB)
- [RDS DB Subnet Group](#)

On the right side, there's a "Provision Instructions" box with a "Copy" button. It contains a snippet of Terraform code:

```
module "vpc" {  
  source = "nfagerlund.tfe.zone/example-corps/vpc"  
  version = "2.0.0"  
}
```

Below this is a "Open in Configuration Designer" button.

Releasing New Versions of a Module

To release a new version of a module, push a new version tag to its VCS repository. The registry will automatically import the new version.

Version tags must be a semantic version (<http://semver.org>), which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2.

Deleting Versions and Modules

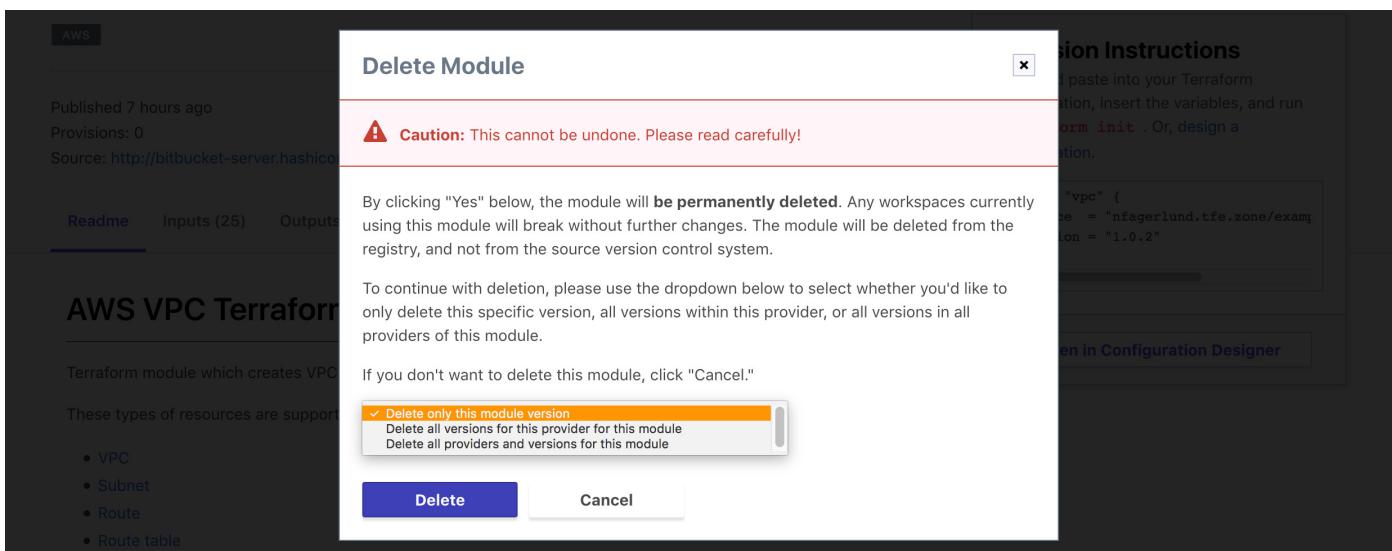
Each module's details page includes a "Delete Module" button, which can delete individual versions of a module or entire modules.

Important: Use caution; deletion can't be undone. However, you can restore a deleted module by re-adding it as a new module. You can also restore a deleted version by deleting the corresponding tag from VCS and pushing a new tag with the same name, or by deleting the whole module from the registry and re-adding it.

To delete a module or version:

1. Navigate to the module's details page.
2. If you only want to delete a single version, use the "Versions" drop-down to select it.
3. Click the "Delete Module" button.
4. Select the desired action from the drop-down and confirm with the "Delete" button.
 - "Delete only this module version" affects the version of the module you were viewing when you clicked delete.
 - The other two options are only different if you have modules with the same name but different providers. (For example, if you have module repos named `terraform-aws-appserver` and `terraform-azure-appserver`, the registry treats them as alternate providers of the same `appserver` module.) If you use multi-provider modules like this, the "Delete all providers and versions for this module" option can delete multiple modules.

Note: If a deletion would leave a module with no versions, the module will be automatically deleted.



Sharing Modules Across Organizations

In normal operation, TFE doesn't allow one organization's workspaces to use private modules from a different organization. (When TFE runs Terraform, it provides temporary credentials that are only valid for the workspace's organization, and uses those credentials to access modules.) And although it's possible to mix modules from multiple organizations when running Terraform on the command line, we strongly recommend against it.

However, you can easily share modules across organizations by sharing the underlying VCS repository. Grant each organization access to the module's repo, then add the module to each organization's registry. When you push tags to publish new module versions, both organizations will update appropriately.

Using Modules from the Terraform Enterprise Private Module Registry

Note: Currently, the private module registry works with all supported VCS providers except Bitbucket Cloud.

By design, Terraform Enterprise (TFE)'s private module registry works much like the public Terraform Registry ([/docs/registry/index.html](#)). If you're already familiar with the public registry, here are the main differences:

- Use TFE's web UI to browse and search for modules.
- Module source strings are slightly different. The public registry uses a three-part <NAMESPACE>/<MODULE NAME>/<PROVIDER> format, and private modules use a four-part <TFE HOSTNAME>/<TFE ORGANIZATION>/<MODULE NAME>/<PROVIDER> format. For example, to load a module from the `example_corp` organization on the SaaS version of TFE:

```
module "vpc" {  
  source  = "app.terraform.io/example_corp/vpc/aws"  
  version = "1.0.4"  
}
```

- TFE can automatically access your private modules during Terraform runs. However, when running Terraform on the command line, you must configure a `credentials` block in your CLI configuration file (`.terraformrc`) ([/docs/commands/cli-config.html](#)). See below for the credentials format.

Finding Modules

All users in your organization can view your private module registry.

To see which modules are available, click the "Modules" button in TFE's main navigation bar.



This brings you to the modules page, which lists all available modules.

The screenshot shows the Terraform Cloud interface. At the top, there's a navigation bar with a logo, the workspace name "example_corp", "Workspaces" and "Modules" tabs, and links for "BETA", "Documentation", "Status", and user profile. Below the navigation is a search bar with the term "consul" and a magnifying glass icon. To the right of the search bar is a "Providers" dropdown menu. The main content area displays three module cards:

- vpc** (PRIVATE) - AWS Version 2.0.0. A "Details" button is in the top right.
- ECS** (PRIVATE) - AWS Version 1.0.7. A "Details" button is in the top right.
- networking** (PRIVATE) - A "Details" button is in the top right.

You can browse the complete list, or shrink the list by searching or filtering.

- The "Providers" drop-down filters the list to show only modules for the selected provider.
- The search field searches by keyword. This only searches the titles of modules, not READMEs or resource details.

Viewing Module Details and Versions

Click a module's "Details" button to view its details page. Use the "Versions" dropdown in the upper right to switch between the available versions, and use the Readme/Inputs/Outputs/Dependencies/Resources tabs to view detailed documentation and information about a version.

AWS

Published a few seconds ago

Provisions: 0

Source: <http://bitbucket-server.████████/projects/TFF/repos/terraform-aws-vpc/browse>
[Readme](#) [Inputs \(32\)](#) [Outputs \(23\)](#) [Dependencies \(0\)](#) [Resources \(27\)](#)

AWS VPC Terraform module

Terraform module which creates VPC resources on AWS.

These types of resources are supported:

- [VPC](#)
- [Subnet](#)
- [Route](#)
- [Route table](#)
- [Internet Gateway](#)
- [NAT Gateway](#)
- [VPN Gateway](#)
- [VPC Endpoint](#) (S3 and DynamoDB)
- [RDS DB Subnet Group](#)

Provision Instructions

Copy and paste into your Terraform configuration, insert the variables, and run `terraform init`. Or, [design a configuration](#).

```
module "vpc" {
  source  = "nfagerlund.tfe.zone/example"
  version = "2.0.0"
}
```

[Open in Configuration Designer](#)

Using Private Modules in Terraform Configurations

In Terraform configurations, you can use any private module from your organization's registry. The syntax for referencing private modules in source attributes is <TFE HOSTNAME>/<TFE ORGANIZATION>/<MODULE NAME>/<PROVIDER>.

```
module "vpc" {
  source  = "app.terraform.io/example_corp/vpc/aws"
  version = "1.0.4"
}
```

If you're using the SaaS version of TFE, the hostname is `app.terraform.io`; private installs have their own hostnames. The second part of the source string (the namespace) is the name of your TFE organization.

For more details on using modules in Terraform configurations, see "Using Modules" in the Terraform docs.
[\(/docs/modules/usage.html\)](/docs/modules/usage.html)

Usage Examples and the Configuration Designer

Each registry page for a module version includes a usage example, which you can copy and paste to get started.

Alternately, you can use the configuration designer, which lets you select multiple modules and fill in their variables to build a much more useful initial configuration. See the configuration designer docs [\(/docs/enterprise/registry/design.html\)](/docs/enterprise/registry/design.html) for details.

The Generic Module Hostname (`localterrafrom.com`)

Optionally, you can use the generic hostname `localterraform.com` in module sources instead of the literal hostname of a Private Terraform Enterprise (PTFE) instance. When Terraform is executed on a PTFE instance, it automatically requests any `localterraform.com` modules from that instance.

For example:

```
module "vpc" {
  source  = "localterraform.com/example_corp/vpc/aws"
  version = "1.0.4"
}
```

Configurations that reference modules via the generic hostname can be used without modification on any PTFE instance, which is not possible when using hardcoded hostnames.

Important: `localterraform.com` only works within a PTFE instance — when run outside of PTFE, Terraform can only use private modules with a literal hostname. To test configurations on a developer workstation, you must replace the generic hostname with a literal hostname in any module sources, then change them back before committing to VCS. We are working on ways to make this smoother in the future; in the meantime, we only recommend `localterraform.com` for large organizations that use multiple PTFE instances.

Running Configurations with Private Modules

In Terraform Enterprise

TFE can use your private modules during plans and applies with no extra configuration, *as long as the workspace is configured to use Terraform 0.11 or higher.*

A given workspace can only use private modules from the organization it belongs to. If you want to use the same module in multiple organizations, you should add it to both organizations' registries. (See [Sharing Modules Across Organizations](#) (/docs/enterprise/registry/publish.html#sharing-modules-across-organizations).)

On the Command Line

If you're using Terraform 0.11 or higher, you can use private modules when applying configurations on the CLI. To do this, you must provide a valid TFE API token (/docs/enterprise/users-teams-organizations/users.html#api-tokens).

Permissions

When you authenticate with a user token, you can access modules from any organization you are a member of. (A user is a member of an organization if they belong to any team in that organization.)

Within a given Terraform configuration, you should only use modules from one organization. Mixing modules from different organizations might work on the CLI with your user token, but it will make your configuration difficult or impossible to collaborate with. If you want to use the same module in multiple organizations, you should add it to both organizations' registries. (See [Sharing Modules Across Organizations](#) (/docs/enterprise/registry/publish.html#sharing-modules-across-organizations).)

Configuration

To configure private module access, add a `credentials` block to your CLI configuration file (`.terraformrc`) ([/docs/commands/cli-config.html](#)).

```
credentials "app.terraform.io" {
  token = "xxxxxx.atlasv1.zzzzzzzzzzzz"
}
```

The block label for the `credentials` block must be TFE's hostname (`app.terraform.io` or the hostname of your private install), and the block body must contain a `token` attribute whose value is a TFE authentication token. You can generate a personal API token from your user settings page in TFE.

Make sure the hostname matches the hostname you use in module sources — if the same TFE server is available at two hostnames, Terraform doesn't have any way to know that they're the same. If you need to support multiple hostnames for module sources, you can add two `credentials` blocks with the same `token`.

Important: Make sure to protect your API token. When adding an authentication token to your CLI config file, check the file permissions and make sure other users on the same computer cannot view its contents.

About Terraform Runs in Terraform Enterprise

Terraform Enterprise (TFE) provides a central interface for running Terraform within a large collaborative organization. If you're accustomed to running Terraform from your workstation, the way TFE manages runs can be unfamiliar.

This page describes the basics of how runs work in TFE.

Runs and Workspaces

TFE always performs Terraform runs in the context of a workspace (</docs/enterprise/run/index.html>). The workspace provides the state and variables for the run, and usually specifies where the configuration should come from.

Each workspace in TFE maintains its own queue of runs, and processes those runs in order.

Whenever a new run is initiated, it's added to the end of the queue. If there's already a run in progress, the new run won't start until the current one has completely finished — TFE won't even plan the run yet, because the current run might change what a future run would do. Runs that are waiting for other runs to finish are in a *pending* state, and a workspace might have any number of pending runs.

When you initiate a run, TFE locks the run to the current Terraform code (usually associated with a specific VCS commit) and variable values. If you change variables or commit new code before the run finishes, it will only affect future runs, not ones that are already pending, planning, or awaiting apply.

Starting Runs

TFE has three main workflows for managing runs, and your chosen workflow determines when and how Terraform runs occur. For detailed information, see:

- The UI/VCS-driven run workflow (</docs/enterprise/run/ui.html>), which is TFE's primary mode of operation.
- The API-driven run workflow (</docs/enterprise/run/api.html>), which is more flexible but requires you to create some tooling.
- The CLI-driven run workflow (</docs/enterprise/run/cli.html>), which uses Terraform's standard CLI tools to execute runs in TFE.

In more abstract terms, TFE runs can be initiated by VCS webhooks, the manual "Queue Plan" button on a workspace, the standard `terraform apply` command (with the remote backend configured), and the Runs API (</docs/enterprise/api/run.html>) (or any tool that uses that API).

Plans and Applies

TFE enforces Terraform's division between *plan* and *apply* operations. It always plans first, saves the plan's output, and uses that output for the apply. In the default configuration, it waits for user approval before running an apply, but you can configure workspaces to automatically apply (</docs/enterprise/workspaces/settings.html#auto-apply-and-manual-apply>) successful plans.

Speculative Plans

In addition to normal runs, TFE can also run *speculative plans*, to test changes to a configuration during editing and code review.

Speculative plans are plan-only runs: they show a set of possible changes (and check them against Sentinel policies), but cannot apply those changes. They can begin at any time without waiting for other runs, since they don't affect real infrastructure. Speculative plans do not appear in a workspace's list of runs; viewing them requires a direct link, which is provided when the plan is initiated.

There are three ways to run speculative plans:

- In VCS-backed workspaces (</docs/enterprise/run/ui.html>), each pull request starts a speculative plan. TFE adds a link to the plan in the VCS provider's pull request interface. If multiple workspaces use the same repository, each of them will add a plan to the pull request.
- With the remote backend (</docs/backends/types/remote.html>) configured, running `terraform plan` on the command line starts a speculative plan. The plan output streams to the terminal, and a link to the plan is also included.
- The runs API creates speculative plans whenever the specified configuration version is marked as speculative. See the `configuration-versions` API (</docs/enterprise/api/configuration-versions.html#create-a-configuration-version>) for more information.

Run States

Each run passes through several stages of action (pending, plan, policy check, apply, and completion), and TFE shows the progress through those stages as run states. In some states, the run might require confirmation before continuing or ending; see [Interacting with Runs](#) below.

In the list of workspaces on TFE's main page, each workspace shows the state of the run it's currently processing. (Or, if no run is in progress, the state of the most recent completed run.)

For full details about the stages of a run, see [Run States and Stages](#) (</docs/enterprise/run/states.html>).

Network Access to VCS and Infrastructure Providers

In order to perform Terraform runs, TFE needs network access to all of the resources being managed by Terraform.

If you are using the SaaS version of TFE, this means your VCS provider and any private infrastructure providers you manage with Terraform (including VMware vSphere, OpenStack, other private clouds, and more) *must be internet accessible*.

Private installs of TFE must have network connectivity to any connected VCS providers or managed infrastructure providers.

Navigating Runs

API: See the [Runs API](#) (</docs/enterprise/api/run.html>).

Each workspace has two ways to view and navigate runs:

- A "Runs" link, which goes to the full list of runs.
- A "Current Run" link, which goes to the most recent active run. (This might not be the most recently initiated run, since runs in the "pending" state remain inactive until the current run is completed.)

The screenshot shows the 'Runs' tab selected in the navigation bar. Below it, a table lists seven runs:

Run ID	Description	Status	Triggered From	Branch	Time
03a9054	queued manually by api	NO CHANGES	#run-vMJRjrFtJLKGvGR nfagerlund triggered from Terraform Enterprise API	master	22 days ago
03a9054	Queued manually in Terraform Enterprise	APPLIED	#run-PrwomHoCLOc2k9sf nfagerlund triggered from Terraform Enterprise UI	master	a month ago
03a9054	Queued manually in Terraform Enterprise	APPLIED	#run-BK8ucYMoUa3WTUCE nfagerlund triggered from Terraform Enterprise UI	master	a month ago
03a9054	Queued manually in Terraform Enterprise	APPLIED	#run-nV2oWic432G6Jyh5 nfagerlund triggered from Terraform Enterprise UI	master	a month ago
03a9054	Queued manually in Terraform Enterprise	CANCELED	#run-zZasSvbLgtHjGeAz nfagerlund triggered from Terraform Enterprise UI	master	a month ago
03a9054	Queued manually in Terraform Enterprise	NO CHANGES	#run-CrsfsBbSsfyLwQnA nfagerlund triggered from Terraform Enterprise UI	master	a month ago
03a9054	add local-exec provisioner that sleeps for 20	NO CHANGES	#run-p6fRYfyLCGfVE13w nfagerlund triggered from GitHub	master	a month ago

From the list of runs, you can click to view or interact with an individual run.

The Run Page

An individual run page shows the progress and outcomes of each stage of the run.

minimum-prod

[Queue Plan](#)

Current Run **Runs** States Variables Settings Integrations Version Control Access

✓ APPLIED Add empty `random` provider block for resource destruction

 nfagerlund triggered a run from GitHub 9 months ago [Run Details](#)

 Plan finished 9 months ago

 Apply finished 9 months ago

 nfagerlund 9 months ago let's try this again!

Run confirmed

Comment: Leave feedback or record a decision.

Add Comment



Support Terms Privacy Security © 2018 HashiCorp, Inc.

Most importantly, it shows:

- The current status of the run.
- The code commit associated with the run.
- How the run was initiated, when, and which user initiated it (if applicable).
- A timeline of events related to the run.
- The output from both the `terraform plan` and `terraform apply` commands, if applicable. You can hide or reveal these as needed; they default to visible if the command is currently running, and hidden if the command has finished.

Interacting with Runs

API: See the Runs API (</docs/enterprise/api/run.html>).

In workspaces where you have write permissions, run pages include controls for interacting with the run at the bottom of the page. Depending on the state of the run, the following buttons might be available:

Button	Available when:
Add Comment	Always.
Confirm & Apply	A plan needs confirmation.

Button	Available when:
Override & Continue	A soft-mandatory policy failed (only available for owners team).
Discard Run	A plan needs confirmation or a soft-mandatory policy failed.
Cancel Run	A plan or apply is currently running.
Force Cancel Run	A plan or apply was canceled, but something went wrong and TFE couldn't end the run gracefully (only available with workspace admin permissions).

If a plan needs confirmation (with manual apply ([/docs/enterprise/workspaces/settings.html#auto-apply-and-manual-apply](#)) enabled) or a soft-mandatory policy failed, the run will remain paused until a user with appropriate permissions uses these buttons to continue or discard the run. For more details, see Run States and Stages ([/docs/enterprise/run/states.html](#)).

Canceling Runs

If a run is currently planning or applying (and you have write permissions to the workspace), you can cancel the run before it finishes, using the "Cancel Run" button on the run's page.

Canceling a run is roughly equivalent to hitting `ctrl+c` during a Terraform plan or apply on the CLI. The running Terraform process is sent an `INT` signal, which instructs Terraform to end its work and wrap up in the safest way possible. (This gives Terraform a chance to update state for any resources that have already been changed, among other things.)

In rare cases, a cancelled run can fail to end gracefully, and will continue to lock the workspace without accomplishing anything useful. These stuck runs can be **force-canceled**, which immediately terminates the running Terraform process and unlocks the workspace.

Since force-canceling can have dangerous side-effects (including loss of state and orphaned resources), it requires admin permissions on the workspace. Additionally, the "Force Cancel Run" button only appears after the normal cancel button has been used and a cool-off period has elapsed, to ensure TFE has a chance to terminate the run safely.

Locking Workspaces (Preventing Runs)

API: See the Lock a Workspace endpoint ([/docs/enterprise/api/workspaces.html#lock-a-workspace](#)).

If you need to temporarily stop runs from being queued, you can lock the workspace.

A lock prevents TFE from performing any plans or applies in the workspace. This includes automatic runs due to new commits in the VCS repository, manual runs queued via the UI, and runs created with the API or the TFE CLI tool. New runs remain in the "Pending" state until the workspace is unlocked.

You can find the lock button in the workspace settings page ([/docs/enterprise/workspaces/settings.html](#)). Locking a workspace requires write or admin permissions.

Important: Locking a workspace prevents runs within TFE, but it **does not** prevent state from being updated. This means a user with write access can still modify the workspace's resources by running Terraform outside TFE with the `atlas` remote backend ([/docs/backends/types/terraform-enterprise.html](#)). To prevent confusion and accidents, avoid using

the atlas backend in normal workflows; to perform runs from the command line, see TFE's CLI-driven workflow ([/docs/enterprise/run/cli.html](#)).

Workers and Run Queuing

TFE performs Terraform runs in disposable Linux environments, using multiple concurrent worker processes. These workers take jobs from a global queue of runs that are ready for processing; this includes confirmed applies, and plans that have just become the current run on their workspace.

If the global queue has more runs than the workers can handle at once, some of them must wait until a worker becomes available. When the queue is backed up, TFE gives different priorities to different kinds of runs:

- Applies that will make changes to infrastructure have the highest priority.
- Normal plans have the next highest priority.
- Speculative plans have the lowest priority.

TFE can also delay some runs in order to make performance more consistent across organizations. If an organization requests a large number of runs at once, TFE queues some of them immediately, and delays the rest until some of the initial batch have finished; this allows every organization to continue performing runs even during periods of especially heavy load.

Installing Terraform Providers

Providers Distributed by HashiCorp

TFE runs `terraform init` before every plan or apply, which automatically downloads any providers ([/docs/configuration/providers.html](#)) Terraform needs.

Private installs of TFE can automatically install providers as long as they can access `releases.hashicorp.com`. If that isn't feasible due to security requirements, you can manually install providers. Use the `terraform-bundle` tool (<https://github.com/hashicorp/terraform/tree/master/tools/terraform-bundle#installing-a-bundle-in-on-premises-terraform-enterprise>) to build a custom version of Terraform that includes the necessary providers, and configure your workspaces to use that bundled version.

Custom and Community Providers

Note: We are investigating how to improve custom provider installation, so this information might change in the near future.

Terraform only automatically installs plugins from the main list of providers ([/docs/providers/index.html](#)); to use community providers or your own custom providers, you must install them yourself.

Currently, there are two ways to use custom provider plugins with TFE.

- Add the provider binary to the VCS repo (or manually-uploaded configuration version) for any workspace that uses it. Place the compiled `linux_amd64` version of the plugin at `terraform.d/plugins/linux_amd64/<PLUGIN NAME>` (as a relative path from the root of the working directory). The plugin name should follow the naming scheme (<https://www.terraform.io/docs/configuration/providers.html#plugin-names-and-versions>). (Third-party plugins are often distributed with an appropriate filename already set in the distribution archive.)

You can add plugins directly to a configuration repo, or you can add them as Git submodules and symlink the files into `terraform.d/plugins/linux_amd64/`. Submodules are a good choice when many workspaces use the same custom provider, since they keep your repos smaller. If using submodules, enable the "Include submodules on clone" setting ([/docs/enterprise/workspaces/settings.html#include-submodules-on-clone](#)) on any affected workspace.

- **Private TFE only:** Use the `terraform-bundle` tool (<https://github.com/hashicorp/terraform/tree/master/tools/terraform-bundle#installing-a-bundle-in-on-premises-terraform-enterprise>) to add custom providers to a custom Terraform version. This keeps custom providers out of your configuration repos entirely, and is easier to update when many workspaces use the same provider.

Terraform State in TFE

API: See the State Versions API ([/docs/enterprise/api/state-versions.html](#)).

Each TFE workspace has its own separate state data. In order to read and write state for the correct workspace, TFE overrides any configured backend ([/docs/backends/index.html](#)) when running Terraform.

You can view current and historical state data for a workspace from its "States" tab. Each state in the list indicates which run and which VCS commit (if applicable) it was associated with. You can click a state in the list for more details, including a diff against the previous state and a link to the raw state file.

Cross-Workspace State Access

In your Terraform configurations, you can use a `terraform_remote_state` data source ([/docs/providers/terraform/d/remote_state.html](#)) to access outputs ([/docs/configuration/outputs.html](#)) from your other workspaces.

Important: A given workspace can only access state data from within the same organization. If you plan to use multiple TFE organizations, make sure to keep groups of workspaces that use each other's data together in the same organization.

To configure a data source for a TFE workspace, set backend to `atlas` and `config.name` to `<ORGANIZATION>/<WORKSPACE>`.

```
data "terraform_remote_state" "vpc" {
  backend = "atlas"
  config {
    name = "example_corp/vpc-prod"
  }
}

resource "aws_instance" "redis_server" {
  # ...
  subnet_id = "${data.terraform_remote_state.vpc.subnet_id}"
}
```

Backend Details

TFE uses the `atlas` backend ([/docs/backends/types/terraform-enterprise.html](#)) to exchange state data with the Terraform process during runs. The `atlas` backend requires an access token, provided via the `$ATLAS_TOKEN` environment variable. When you run Terraform from the command line, you can use a user API token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) with write permissions on the desired workspace.

When TFE performs a run, it doesn't use existing user credentials; instead it generates a unique per-run API token, and exports it to the Terraform worker's shell environment as `$ATLAS_TOKEN`. This per-run token can read and write state data for the workspace associated with the run, and can read state data from any other workspace in the same organization. It cannot make any other calls to the TFE API. Per-run tokens are not considered to be user, team, or organization tokens, and become invalid after the run is completed.

The API-driven Run Workflow

Terraform Enterprise (TFE) has three workflows for managing Terraform runs.

- The UI/VCS-driven run workflow (</docs/enterprise/run/ui.html>), which is TFE's primary mode of operation.
- The API-driven run workflow described below, which is more flexible but requires you to create some tooling.
- The CLI-driven run workflow (</docs/enterprise/run/cli.html>), which uses Terraform's standard CLI tools to execute runs in TFE.

Summary

In the API-driven workflow, workspaces are not directly associated with a VCS repo, and runs are not driven by webhooks on your VCS provider.

Instead, one of your organization's other tools is in charge of deciding when configuration has changed and a run should occur. Usually this is something like a CI system, or something else capable of monitoring changes to your Terraform code and performing actions in response.

Once your other tooling has decided a run should occur, it must make a series of calls to TFE's `runs` and `configuration-versions` APIs to upload configuration files and perform a run with them. For the exact series of API calls, see the [pushing a new configuration version](#) section.

The most significant difference in this workflow is that TFE *does not* fetch configuration files from version control. Instead, your own tooling must upload the configurations as a `.tar.gz` file. This allows you to work with configurations from unsupported version control systems, automatically generate Terraform configurations from some other source of data, or build a variety of other integrations.

Important: The script below is provided to illustrate the run process, and is not intended for production use. If you want to drive TFE runs from the command line, please see the CLI-driven run workflow (</docs/enterprise/run/cli.html>).

Pushing a New Configuration Version

Pushing a new configuration to an existing workspace is a multi-step process. This section walks through each step in detail, using an example bash script to illustrate.

1. Define Variables

To perform an upload, a few user parameters must be set:

- **path_to_content_directory** is the folder with the terraform configuration. There must be at least one `.tf` file in the root of this path.
- **organization** is the organization name (not ID) for your Terraform Enterprise organization.
- **workspace** is the workspace name (not ID) in the Terraform Enterprise organization.

- **\$TOKEN** is the API Token used for authenticating with the TFE API (/docs/enterprise/api/index.html#authentication).

This script extracts the `path_to_content_directory`, `organization`, and `workspace` from command line arguments, and expects the `$TOKEN` as an environment variable.

```
#!/bin/bash

if [ -z "$1" ] || [ -z "$2" ]; then
  echo "Usage: $0 <path_to_content_directory> <organization>/<workspace>"
  exit 0
fi

CONTENT_DIRECTORY=$1
ORG_NAME=$(cut -d'/' -f1 <<<"$2")
WORKSPACE_NAME=$(cut -d'/' -f2 <<<"$2")
```

2. Create the File for Upload

The configuration version API (/docs/enterprise/api/configuration-versions.html) requires a `tar.gz` file to be uploaded in order for the configuration version to be used for a run, so the content directory (the directory containing the Terraform configuration) must be packaged into a `tar.gz` file.

Important: The configuration directory must be the root of the tar file, with no intermediate directories. In other words, when the tar file is extracted the result must be paths like `./main.tf` rather than `./terraform-appserver/main.tf`.

```
UPLOAD_FILE_NAME="./content-$(date +%s).tar.gz"
tar cvzf $UPLOAD_FILE_NAME -C $CONTENT_DIRECTORY .
```

3. Look Up the Workspace ID

The first step identified the organization name and the workspace name; however, the configuration version API (/docs/enterprise/api/configuration-versions.html) expects the workspace ID. As such, the ID has to be looked up. If the workspace ID is already known, this step can be skipped. This step uses the `jq` tool (<https://stedolan.github.io/jq/>) to parse the JSON output and extract the ID value into the `WORKSPACE_ID` variable.

```
WORKSPACE_ID=$(curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/organizations/$ORG_NAME/workspaces/$WORKSPACE_NAME \
| jq -r '.data.id')
```

4. Create a New Configuration Version

Before uploading the configuration files, you must create a `configuration-version` to associate uploaded content with the workspace. This API call performs two tasks: it creates the new configuration version and it extracts the upload URL to be used in the next step.

```
echo '{"data":{"type":"configuration-version"}}' > ./create_config_version.json

UPLOAD_URL=$(curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @create_config_version.json \
https://app.terraform.io/api/v2/workspaces/$WORKSPACE_ID/configuration-versions \
| jq -r '.data.attributes."upload-url"'))
```

5. Upload the Configuration Content File

Next, upload the configuration version tar.gz file to the upload URL extracted from the previous step. If a file is not uploaded, the configuration version will not be usable, since it will have no Terraform configuration files.

Terraform Enterprise automatically creates a new run with a plan once the new file is uploaded. If the workspace is configured to auto-apply, it will also apply if the plan succeeds; otherwise, an apply can be triggered via the Run Apply API (/docs/enterprise/api/run.html#apply).

```
curl \
--request PUT \
-F "data=@$UPLOAD_FILE_NAME" \
$UPLOAD_URL
```

6. Delete Temporary Files

In the previous steps a few files were created; they are no longer needed, so they should be deleted.

```
rm $UPLOAD_FILE_NAME
rm ./create_config_version.json
```

Complete Script

Combine all of the code blocks into a single file, ./terraform-enterprise-push.sh and give execution permission to create a combined bash script to perform all of the operations.

```
chmod +x ./terraform-enterprise-push.sh
./terraform-enterprise-push.sh ./content my-organization/my-workspace
```

Note: This script does not have error handling, so for a more robust script consider adding error checking or using the tfe-client (<https://github.com/hashicorp/tfe-client>).

./terraform-enterprise-push.sh:

```

#!/bin/bash

if [ -z "$1" ] || [ -z "$2" ]; then
  echo "Usage: $0 <path_to_content_directory> <organization>/<workspace>"
  exit 0
fi

CONTENT_DIRECTORY=$1
ORG_NAME=$(cut -d'/' -f1 <<<"$2")
WORKSPACE_NAME=$(cut -d'/' -f2 <<<"$2")
UPLOAD_FILE_NAME="./content-$(date +%s).tar.gz"
tar -zcvf $UPLOAD_FILE_NAME $CONTENT_DIRECTORY
WORKSPACE_ID=$(curl \
  --header "Authorization: Bearer $TOKEN" \
  --header "Content-Type: application/vnd.api+json" \
  https://app.terraform.io/api/v2/organizations/$ORG_NAME/workspaces/$WORKSPACE_NAME \
  | jq -r '.data.id'))
echo '{"data":{"type":"configuration-version"}}' > ./create_config_version.json

UPLOAD_URL=$(curl \
  --header "Authorization: Bearer $TOKEN" \
  --header "Content-Type: application/vnd.api+json" \
  --request POST \
  --data @create_config_version.json \
  https://app.terraform.io/api/v2/workspaces/$WORKSPACE_ID/configuration-versions \
  | jq -r '.data.attributes."upload-url')")
curl \
  --request PUT \
  -F "data=@$UPLOAD_FILE_NAME" \
  $UPLOAD_URL
rm $UPLOAD_FILE_NAME
rm ./create_config_version.json

```

Advanced Use Cases

For advanced use cases see the TFE Automation Script (<https://github.com/hashicorp/terraform-guides/tree/master/operations/automation-script>) repository for automating interactions with Terraform Enterprise, including the creation of a workspace, uploading TFE code, setting of variables, and triggering a plan/apply.

In addition to uploading configurations and starting runs, you can use TFE's APIs to create and modify workspaces, edit variable values, and more. See the API documentation (/docs/enterprise/api/index.html) for more details.

The CLI-driven Run Workflow

Terraform Enterprise (TFE) has three workflows for managing Terraform runs.

- The UI/VCS-driven run workflow (</docs/enterprise/run/ui.html>), which is TFE's primary mode of operation.
- The API-driven run workflow (</docs/enterprise/run/api.html>), which is more flexible but requires you to create some tooling.
- The CLI-driven run workflow described below, which uses Terraform's standard CLI tools to execute runs in TFE.

Preview Release: As of Terraform 0.11.8, the remote backend is a preview release, and we do not recommend using it with production workloads. Please continue to use the existing Terraform Enterprise (atlas) backend or the TFE CLI tool (<https://github.com/hashicorp/tfe-cli/>) for production workspaces.

Summary

The Terraform remote backend (</docs/backends/types/remote.html>) brings Terraform Enterprise's collaboration features into the familiar Terraform CLI workflow. It offers the best of both worlds to developers who are already comfortable with using Terraform, and can work with existing CI/CD pipelines.

Users can start runs with the standard `terraform plan` and `terraform apply` commands, and can watch the progress of the run without leaving their terminal. These runs execute remotely in Terraform Enterprise; they use variables from the appropriate workspace, enforce any applicable Sentinel policies (</docs/enterprise/sentinel/index.html>), and can access Terraform Enterprise's private module registry (</docs/enterprise/registry/index.html>) and remote state inputs.

Terraform Enterprise offers two kinds of CLI-driven runs, to support different stages of your workflow:

- `terraform plan` starts a speculative plan (</docs/enterprise/run/index.html#speculative-plans>) in a Terraform Enterprise workspace, for fast feedback while developing Terraform configurations. Developers can quickly check the results of their edits (including compliance with Sentinel policies) without needing to copy sensitive variables to their local machine.

Speculative plans work with all workspaces, and can co-exist with the VCS-driven workflow (</docs/enterprise/run/ui.html>).
- `terraform apply` starts a normal plan and apply in a Terraform Enterprise workspace, using configuration files from a local directory.

Remote `terraform apply` is for workspaces without a linked VCS repository. It replaces the VCS-driven workflow with a more traditional CLI workflow.

To supplement these remote operations, you can also use the optional TFE CLI tool (<https://github.com/hashicorp/tfe-cli/>), which is a flexible CLI interface to Terraform Enterprise's API. This tool isn't required for initiating runs, but it can be useful for editing variables and workspace settings from your terminal. Downloads and documentation for the TFE CLI tool are available at its GitHub repository (<https://github.com/hashicorp/tfe-cli/>).

Remote Backend Configuration

To configure the remote backend, a stanza needs to be added to the Terraform configuration. It must specify the `remote` backend, the name of a Terraform Enterprise organization, and the workspace(s) to use. The example below uses one workspace; see the remote backend documentation ([/docs/backends/types/remote.html](#)) for more details.

```
terraform {
  backend "remote" {
    organization = "my-org"

    workspaces {
      name = "my-app-dev"
    }
  }
}
```

A Terraform Enterprise user API token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) is also needed. It should be set as `credentials` in the CLI config file ([/docs/commands/cli-config.html#credentials](#)). User tokens can be created in the user settings ([/docs/enterprise/users-teams-organizations/users.html#user-settings](#)).

```
credentials "app.terraform.io" {
  token = "xxxxxx.atlasv1.zzzzzzzzzzzz"
}
```

The backend can be initialized with `terraform init`. If the workspaces do not yet exist in Terraform Enterprise, they will be created at this time.

```
$ terraform init

Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Remote Speculative Plans

To run a speculative plan ([/docs/enterprise/run/index.html#speculative-plans](#)) on your configuration, use the `terraform plan` command. The plan will run in Terraform Enterprise, and the logs will stream back to the command line along with a URL to view the plan in the Terraform Enterprise UI.

Users can run speculative plans in any workspace where they have plan access ([/docs/enterprise/users-teams-organizations/permissions.html](#)).

Speculative plans use the configuration code from the local working directory, but will use variable values from the specified Terraform Enterprise workspace.

```
$ terraform plan

Running plan in the remote backend. Output will stream here. Pressing Ctrl-C
will stop streaming the logs, but will not stop the plan running remotely.
To view this plan in a browser, visit:
https://app.terraform.io/app/my-org/my-app-dev/runs/run-LU3uk79BE5Uj77io

Waiting for the plan to start...

Terraform v0.11.9

Configuring remote state backend...
Initializing Terraform configuration...
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

[...]

Plan: 1 to add, 0 to change, 0 to destroy.
```

Remote Applies

When configuration changes are ready to be applied, use the `terraform apply` command. The apply will start in Terraform Enterprise, and the command line will prompt for approval before applying the changes.

Remote applies require write access ([/docs/enterprise/users-teams-organizations/permissions.html](#)) to the workspace.

Remote applies use the configuration code from the local working directory, but will use variable values from the specified Terraform Enterprise workspace.

Important: You cannot run remote applies in workspaces that are linked to a VCS repository, since the repository serves as the workspace's source of truth. To apply changes in a VCS-linked workspace, merge your changes to the designated branch.

```
$ terraform apply

Running apply in the remote backend. Output will stream here. Pressing Ctrl-C
will cancel the remote apply if its still pending. If the apply started it
will stop streaming the logs, but will not stop the apply running remotely.
To view this run in a browser, visit:
https://app.terraform.io/app/my-org/my-app-dev/runs/run-PEekqv44Fs8NkiFx

Waiting for the plan to start...

[...]

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions in workspace "my-app-dev"?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

[...]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Sentinel Policies

If the specified workspace uses Sentinel policies, those policies will run against all speculative plans and remote applies in that workspace. The policy output will be available in the terminal. Hard mandatory checks cannot be overridden and they prevent `terraform apply` from applying changes.

```
$ terraform apply

[...]

Plan: 1 to add, 0 to change, 1 to destroy.

-----
Organization policy check:

Sentinel Result: false

Sentinel evaluated to false because one or more Sentinel policies evaluated
to false. This false was not due to an undefined value or runtime error.

1 policies evaluated.
## Policy 1: my-policy.sentinel (hard-mandatory)

Result: false

FALSE - my-policy.sentinel:1:1 - Rule "main"

Error: Organization policy check hard failed.
```

Installing Software in the Run Environment

In some cases, it may be useful to install certain software on the Terraform worker, such as a configuration management tool or cloud CLI.

Terraform's `local-exec` provisioner (<https://www.terraform.io/docs/provisioners/local-exec.html>) can be used in Terraform Enterprise (TFE). This will execute commands on the host machine running Terraform. In general, the provisioner configuration block containing the `local-exec` provisioner should be added to the resource(s) the software being installed will interact with.

```
resource "aws_instance" "example" {
  ami           = "${var.ami}"
  instance_type = "t2.micro"
  provisioner "local-exec" {
    command = <<EOH
    sudo apt-get update
    sudo apt-get install -y ansible
  EOH
  }
}
```

Using software installed via `local-exec` to create unmanaged infrastructure is not recommended. Instead, if possible, use a provider that can manage the resource using a Terraform configuration. See Terraform Runs: Custom and Community Providers (/docs/enterprise/run/index.html#custom-and-community-providers) for information on how to use custom and community providers in TFE.

Please note that the machines that run Terraform (containers, in Private Terraform Enterprise) exist in an isolated environment and are destroyed after each use. Very little is pre-installed and nothing is persisted between Terraform runs, so you will need to install any custom software on each run.

Because the run environments are disposable, using a null resource with a `local-exec` provisioner to install software, rather than a related resource, is only appropriate if the null resource can be properly configured with triggers (https://www.terraform.io/docs/provisioners/null_resource.html#example-usage). Otherwise, a null resource with the `local-exec` provisioner will only install software on the initial run where the `null_resource` is created. The `null_resource` will not be automatically recreated in subsequent runs and the related software won't be installed, which may cause runs to encounter errors.

Currently, the Terraform workers run the latest version of Ubuntu LTS, but this may change in the future.

Run States and Stages

Each run passes through several stages of action (pending, plan, policy check, apply, and completion), and TFE shows the progress through those stages as run states.

In the list of workspaces on TFE's main page, each workspace shows the state of the run it's currently processing. (Or, if no run is in progress, the state of the most recent completed run.)

1. The Pending Stage

States in this stage:

- **Pending:** TFE hasn't started action on a run yet. TFE processes each workspace's runs in the order they were queued, and a run remains pending until every run before it has completed.

Leaving this stage:

- Proceeds automatically to the plan stage (**Planning** state) when it becomes the first run in the queue.
- Can skip to completion (**Discarded** state) if a user discards it before it starts.

2. The Plan Stage

States in this stage:

- **Planning:** TFE is currently running `terraform plan`.
- **Needs Confirmation:** `terraform plan` has finished. TFE sometimes pauses in this state, depending on the workspace and organization settings.

Leaving this stage:

- If the `terraform plan` command failed, TFE skips to completion (**Plan Errored** state).
- If a user canceled the plan by pressing the "Cancel Run" button, TFE skips to completion (**Canceled** state).
- If the plan succeeded and doesn't require any changes (since it already matches the current infrastructure state), TFE skips to completion (**No Changes** state).
- If the plan succeeded and requires changes:
 - If Sentinel policies (/docs/enterprise/sentinel/index.html) are enabled, TFE proceeds automatically to the policy check stage.
 - If there are no Sentinel policies and auto-apply is enabled on the workspace, TFE proceeds automatically to the apply stage.
 - If there are no Sentinel policies and auto-apply is disabled, TFE pauses in the **Needs Confirmation** state until a user with write access to the workspace takes action. The run proceeds to the apply stage if they approve the apply, or skips to completion (**Discarded** state) if they reject the apply.

3. The Policy Check Stage

This stage only occurs if Sentinel policies (/docs/enterprise/sentinel/index.html) are enabled. After a successful `terraform` plan, TFE checks whether the plan obeys policy to determine whether it can be applied.

States in this stage:

- **Policy Check:** TFE is currently checking the plan against the organization's policies.
- **Policy Override:** The policy check finished, but a soft-mandatory policy failed, so an apply cannot proceed without approval from a member of the owners team. TFE pauses in this state.
- **Policy Checked:** The policy check succeeded, and Sentinel will allow an apply to proceed. TFE sometimes pauses in this state, depending on workspace settings.

Leaving this stage:

- If any hard-mandatory policies failed, TFE skips to completion (**Plan Errored** state).
- If any soft-mandatory policies failed, TFE pauses in the **Policy Override** state.
 - If a member of the owners team overrides the failed policy, the run proceeds to the **Policy Checked** state.
 - If an owner or a user with write access discards the run, TFE skips to completion (**Discarded** state).
- If the run reaches the **Policy Checked** state (no mandatory policies failed, or soft-mandatory policies were overridden):
 - If auto-apply is enabled on the workspace, TFE proceeds automatically to the apply stage.
 - If auto-apply is disabled, TFE pauses in the **Policy Checked** state until a user with write access takes action. The run proceeds to the apply stage if they approve the apply, or skips to completion (**Discarded** state) if they reject the apply.

4. The Apply Stage

States in this stage:

- **Applying:** TFE is currently running `terraform apply`.

Leaving this stage:

After applying, TFE proceeds automatically to completion.

- If the apply succeeded, the run ends in the **Applied** state.
- If the apply failed, the run ends in the **Apply Errored** state.
- If a user canceled the apply by pressing the "Cancel Run" button, the run ends in the **Canceled** state.

5. Completion

A run is considered completed if it finishes applying, if any part of the run fails, or if a user chooses not to continue. Once a run is completed, the next run in the queue can enter the plan stage.

States in this stage:

- **Applied:** TFE has successfully finished applying.
- **No Changes:** `terraform plan`'s output already matches the current infrastructure state, so `terraform apply` doesn't need to do anything.
- **Apply Errored:** The `terraform apply` command failed, possibly due to a missing or misconfigured provider or an illegal operation on a provider.
- **Plan Errored:** The `terraform plan` command failed (usually requiring fixes to variables or code), or a hard-mandatory Sentinel policy failed. The run cannot be applied.
- **Discarded:** A user chose not to continue this run.
- **Canceled:** A user interrupted the `terraform plan` or `terraform apply` command with the "Cancel Run" button.

The UI- and VCS-driven Run Workflow

Terraform Enterprise (TFE) has three workflows for managing Terraform runs.

- The UI/VCS-driven run workflow described below, which is TFE's primary mode of operation.
- The API-driven run workflow (</docs/enterprise/run/api.html>), which is more flexible but requires you to create some tooling.
- The CLI-driven run workflow (</docs/enterprise/run/cli.html>), which uses Terraform's standard CLI tools to execute runs in TFE.

Summary

In the UI and VCS workflow, every workspace is associated with a specific branch of a VCS repo of Terraform configurations. TFE registers webhooks with your VCS provider when you create a workspace, then automatically queues a Terraform run whenever new commits are merged to that branch of workspace's linked repository.

TFE also performs a speculative plan (</docs/enterprise/run/index.html#speculative-plans>) when a pull request is opened against that branch from another branch in the linked repository. TFE posts a link to the plan in the pull request, and re-runs the plan if the pull request is updated.

The Terraform code for a normal run always comes from version control, and is always associated with a specific commit.

Starting Runs

Most of the time, runs start automatically whenever you commit changes to version control through a merge or direct commit to the target branch.

When you initially set up the workspace and add variables, or when the code in version control hasn't changed but you've modified some variables in TFE, you can manually queue a plan from the UI. Each workspace has a "Queue Plan" button for this purpose. Manually queueing a plan requires write or admin access.

If the workspace has a plan that is still in the plan stage (</docs/enterprise/run/states.html#2-the-plan-stage>) when a new plan is queued, you can either wait for it to complete, or visit the "Current Run" page and click "Run this plan now". Be aware that this will terminate the current plan and unlock the workspace, which can lead to anomalies in behavior, but can be useful if the plans are long-running and the current plan is known not to have all the desired changes.

Confirming or Discarding Plans

By default, run plans require confirmation before TFE will apply them. Users with write access on a workspace can navigate to a run that has finished planning and click the "Confirm & Apply" or "Discard Plan" button to finish or cancel a run. If necessary, use the "View Plan" button for more details about what the run will change.

Plan finished 34 minutes ago

Resources: 1 to add, 0 to change, 1 to destroy ^

Queued 34 minutes ago > Started 34 minutes ago > Finished 34 minutes ago

[View raw log](#)

Top Bottom Expand Full Screen

```
-/+ destroy and then create replacement

Terraform will perform the following actions:

-/+ random_id.random (new resource required)
  id:      "3zkmEfTiyk8" => <computed> (forces new resource)
  b64:    "3zkmEfTiyk8" => <computed>
  b64_std: "3zkmEfTiyk8" => <computed>
  b64_url: "3zkmEfTiyk8" => <computed>
  byte_length: "8" => "8"
  dec:     "16084929395824298575" => <computed>
  hex:     "df3926105b588a4f" => <computed>
  keepers.%: "1" => "1"
  keepers.uuid: "b5bfe2b7-ca6e-c0cd-7a01-b659aac398f7" => "f87d124d-9155-90d1-c994-a14f4ffc8adf" (forces new resource)

Plan: 1 to add, 0 to change, 1 to destroy.
```

Apply pending

Needs Confirmation: Check the plan and confirm to apply it, or discard the run.

[Confirm & Apply](#) [Discard Run](#) [Add Comment](#)

Users can also leave comments if there's something unusual involved in a run.

Note that once the plan stage is completed, until you apply or discard a plan, TFE can't start another run in that workspace.

Auto apply

If you would rather automatically apply plans that don't have errors, you can enable auto apply on the workspace's settings tab.

Speculative Plans on Pull Requests

When a pull request is made to a linked repository, each environment linked to that repository runs a speculative plan ([/docs/enterprise/run/index.html#speculative-plans](#)) and posts a link to the plan in the pull request. Members of your organization can consult the results of these plans when reviewing pull requests. Speculative plans are re-run if the code in a pull request is updated.

Due to VCS providers' access controls, this feature only works for pull requests that originate *from* the linked repository — pull requests that originate from other forks of the repository do not receive speculative plans, since TFE can't reliably access or monitor the contents of those forks.

Speculative Plans During Development

You can also run speculative plans on demand before making a pull request, using the remote backend and the `terraform plan` command. For more information, see the CLI-driven run workflow ([/docs/enterprise/run/cli.html](#)).

SAML Single Sign On

Note: The SAML Single Sign On feature is only available with the Premium tier on private installs.

SAML is an XML-based standard for authentication and authorization. Terraform Enterprise can act as a service provider (SP) (or Relying Party) with your internal SAML identity provider (IdP).

Terraform Enterprise supports the SAML 2.0 standard. It has been tested with a variety of identity providers.

Attributes

The following SAML attributes correspond to properties of a Terraform Enterprise user account. When a new or existing user logs in, their account info will be updated with data from these attributes.

Username

If Username is specified, TFE will assign that username to the user instead of using an automatic name based on their email address (/docs/enterprise/saml/login.html). When the username is already taken or is invalid, login will still complete, and the existing or default value will be used instead.

```
<saml:AttributeStatement>
  <saml:Attribute Name="Username" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
    <saml:AttributeValue xsi:type="xs:string">new-username</saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

SiteAdmin

If the SiteAdmin attribute is present, the system will grant or revoke site admin access (/docs/enterprise/private/admin/index.html) for the user. Site admin access can be also be granted or revoked in the MemberOf attribute; however the SiteAdmin attribute is the recommended method of managing access and will override the other value.

```
<saml:AttributeStatement>
  <saml:Attribute Name="SiteAdmin">
    <saml:AttributeValue xsi:type="xs:boolean">false</saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

MemberOf

Team membership is specified in the MemberOf attribute. (If desired, you can configure a different name (/docs/enterprise/saml/team-membership.html) for the team membership attribute.)

Teams can be specified in separate AttributeValue items:

```
<saml:AttributeStatement>
  <saml:Attribute Name="MemberOf" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
    <saml:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">devs</saml:AttributeValue>
    <saml:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">reviewers</saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

or in one AttributeValue as a comma-separated list:

```
<saml:AttributeStatement>
  <saml:Attribute Name="MemberOf" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
    <saml:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">list,
    of,roles</saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

There is a special-case role `site-admins` that will add a user as a site admin to your private Terraform Enterprise instance.

```
<saml:AttributeStatement>
  <saml:Attribute Name="MemberOf" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
    <saml:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">site-
    admins</saml:AttributeValue>
    <saml:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">devs<
    /saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

Configuration

SAML requires the configuration of two parties:

- The Identity Provider (IdP).
- The Service Provider (SP), which is also sometimes referred to as Relying Party (RP).

Private Terraform Enterprise (PTFE) is configured as the Service Provider.

Note: For instructions for specific IdPs, see Identity Provider Configuration (</docs/enterprise/saml/identity-provider-configuration.html>).

API: See the Admin Settings API (</docs/enterprise/api/admin/settings.html>).

Terraform Enterprise (Service Provider)

Important: Only PTFE users with the site-admin permission can modify SAML settings. For more information about site admins, see Administering Private Terraform Enterprise (</docs/enterprise/private/admin/index.html>).

Prior to activating SAML, we recommend that you create a non-SSO admin account for recovery (</docs/enterprise/saml/troubleshooting.html#create-a-non-sso-admin-account-for-recovery>).

In case of any issues during SAML configuration, this ensures that there will be an admin able to log in and make necessary adjustments.

Go to the SAML section of the site admin pages. You can use the "Site Admin" link in the upper-right user icon menu, or go directly to <https://<TFE HOSTNAME>/app/admin/saml>.

Once there, enter values for TFE's SAML settings and click the "Save SAML Settings" button at the bottom of the page.

The SAML settings are separated into sections:

SAML Settings

- **Enable SAML single sign-on:** This checkbox must be enabled.

Identity Provider Settings

- **Single Sign-On URL:** The HTTP(S) endpoint on your IdP for single sign-on requests. This value is provided by your IdP configuration.
- **Single Log-Out URL:** The HTTP(s) endpoint on your IdP for single logout requests. This value is provided by your IdP configuration. Single Logout is not yet supported.
- **IdP Certificate:** The PEM encoded X.509 Certificate as provided by the IdP configuration.

Attributes

- **Username Attribute Name:** (default: Username) The name of the SAML attribute that determines the TFE username for a user logging in via SSO.
- **Site Admin Attribute Name:** (default: SiteAdmin) The name of the SAML attribute that determines whether a user has site-admin permissions. The value of this attribute in the SAML assertion must be a boolean. Site admins can manage settings and resources for the entire PTFE instance; see Administering Private Terraform Enterprise (</docs/enterprise/private/admin/index.html>) for details.
- **Team Attribute Name:** (default: MemberOf) The name of the SAML attribute that determines team membership (</docs/enterprise/saml/team-membership.html>). The value of this attribute in the SAML assertion must be a string containing a comma-separated list of team names.

Team Membership Mapping

- **Site Admin Role:** (default: site-admins; make blank to disable) An alternate way of managing site-admin permissions; if a role with this name is present in the value of the Team Attribute Name attribute, the user is an admin.

We recommend using the "site admin attribute name" setting instead. If you are using the site admin attribute, you can disable "site admin role" by deleting its value.

User Session

- **API Token Session Timeout:** (default: 1209600 seconds, or 14 days) The duration of time (in seconds) for which TFE will accept a user's API token (</docs/enterprise/users-teams-organizations/users.html#api-tokens>) before requiring the user to log in again. For more details about this behavior, see API Token Expiration (</docs/enterprise/saml/login.html#api-token-expiration>).

Identity Provider

Configure the following values in the SAML Identity Provider (IdP):

1. **Audience:** `https://<TFE HOSTNAME>/users/saml/metadata`
2. **Recipient:** `https://<TFE HOSTNAME>/users/saml/auth`
3. **ACS (Consumer) URL:** `https://<TFE HOSTNAME>/users/saml/auth`

The SAML Metadata document is available at: `https://<TFE HOSTNAME>/users/saml/metadata.xml`

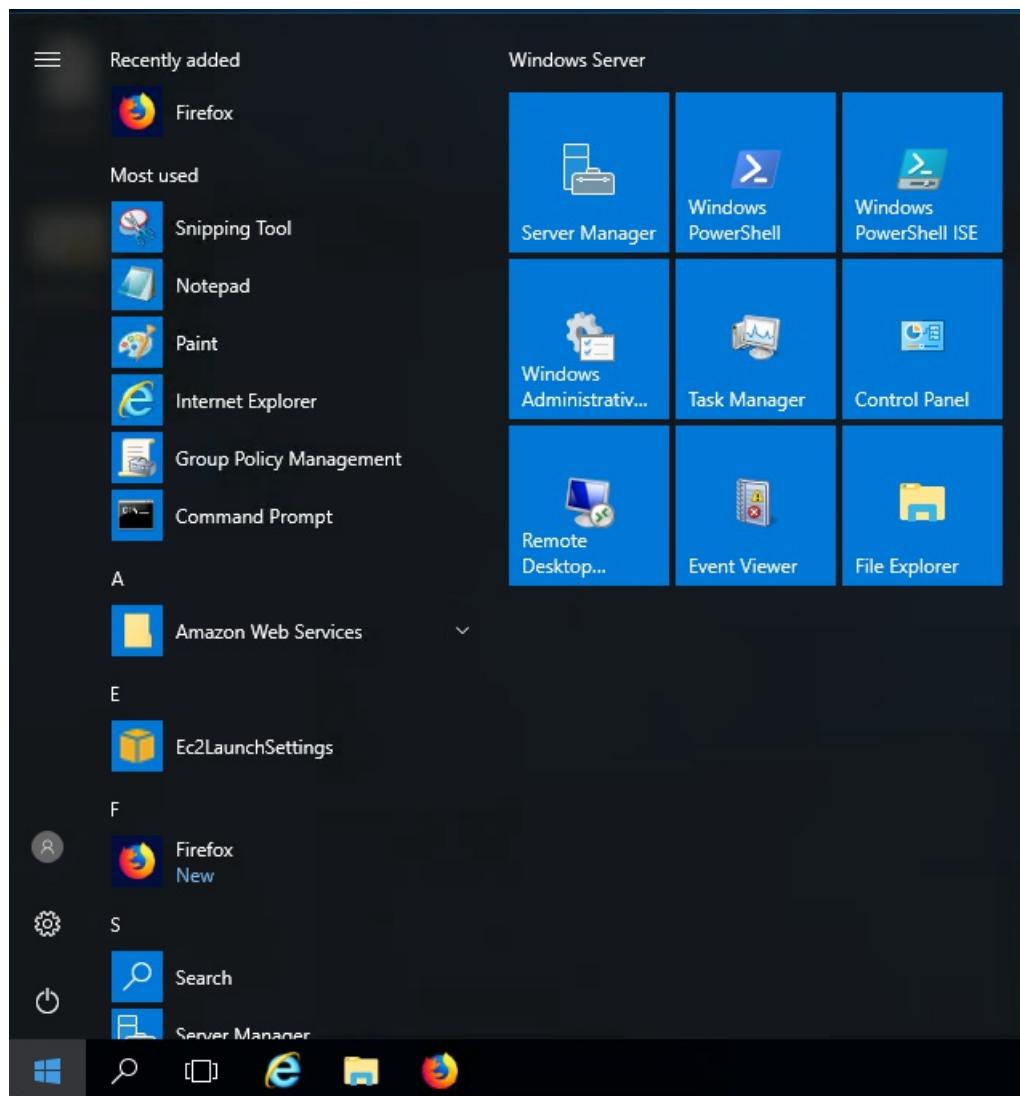
ADFS Configuration

This guide explains how to configure Active Directory Federated Services (ADFS) in order to use it as an Identity Provider (IdP) for PTFE's SAML authentication feature. The screenshots below were taken on Windows Server 2016, and the UI may not look the same on previous Windows versions.

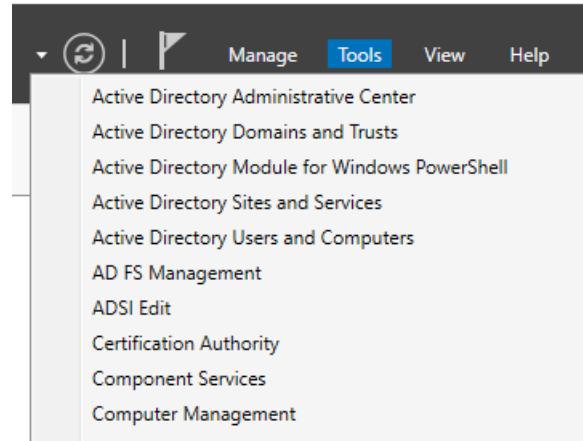
This document assumes that you have already installed and configured ADFS and that you are using PTFE version v201807-1 or later.

Gather ADFS information

1. On the ADFS server, start the Server Manager.



2. Click "Tools" -> "AD FS Management".



3. Expand the Service object and click "Endpoints".

AD FS

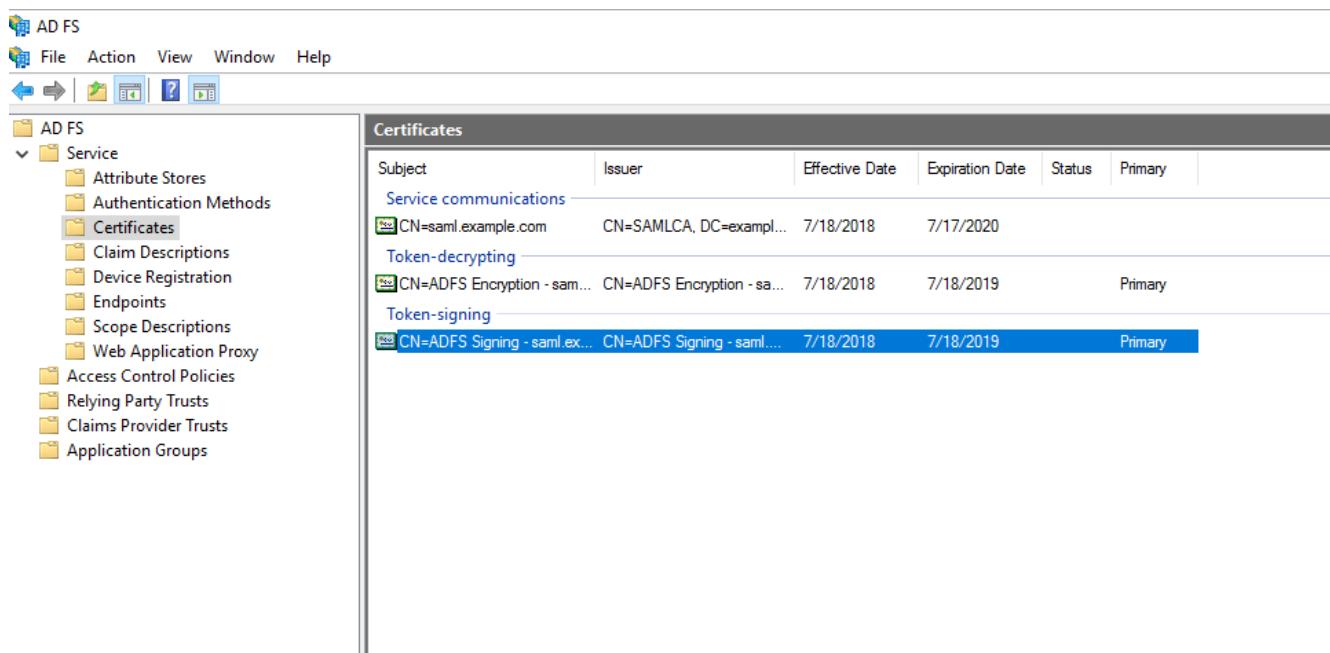
File Action View Window Help

Endpoints

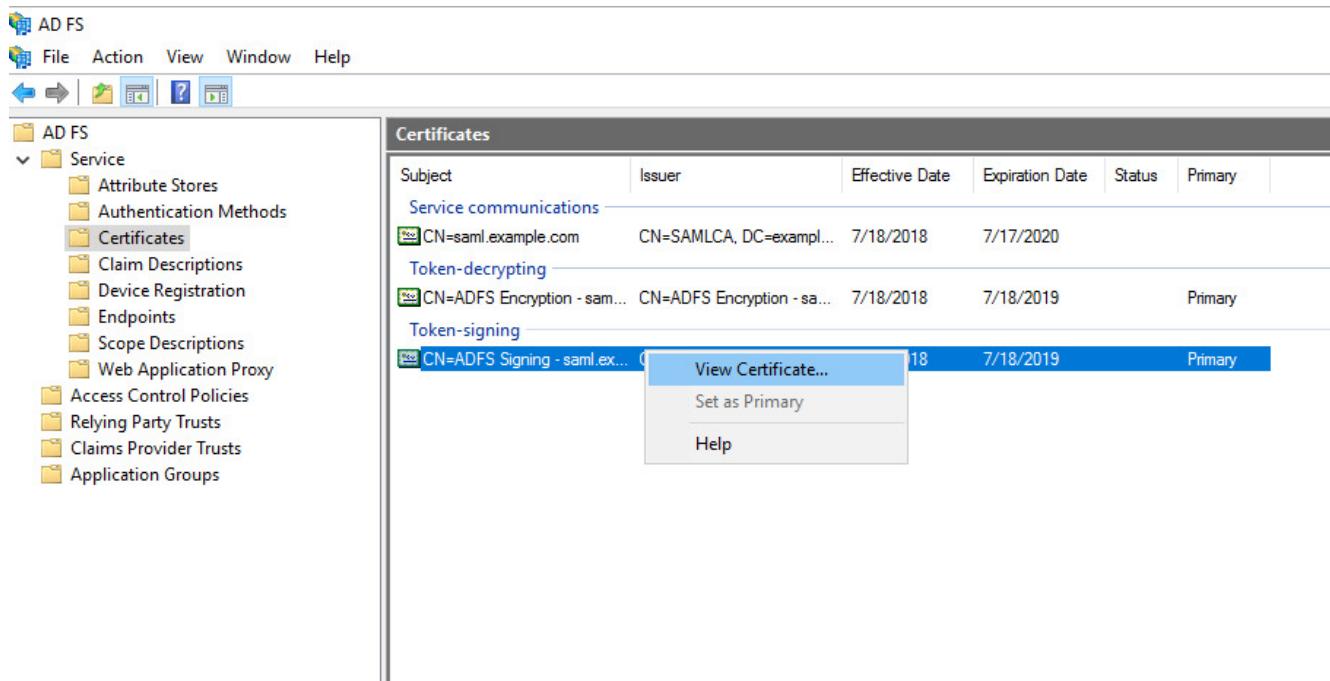
Enabled	Proxy Enabled	URL Path	Type	Authentication Type	Security Mode
Yes	Yes	/adfs/ls/	SAML 2.0/WS-Federation	Anonymous	Transport
No	No	/adfs/services/trust/2005/windows	WS-Trust 2005	Windows	Message
No	No	/adfs/services/trust/2005/windowsvmixed	WS-Trust 2005	Windows	Mixed
Yes	Yes	/adfs/services/trust/2005/windowstransport	WS-Trust 2005	Windows	Transport
No	No	/adfs/services/trust/2005/certificate	WS-Trust 2005	Certificate	Message
Yes	Yes	/adfs/services/trust/2005/certificatemixed	WS-Trust 2005	Certificate	Mixed
Yes	Yes	/adfs/services/trust/2005/username	WS-Trust 2005	Username	Transport
No	No	/adfs/services/trust/2005/usernamebasictransport	WS-Trust 2005	Username	Transport
Yes	Yes	/adfs/services/trust/2005/usernamemixed	WS-Trust 2005	Username	Mixed
Yes	No	/adfs/services/trust/2005/kerberosmixed	WS-Trust 2005	Kerberos	Mixed
No	No	/adfs/services/trust/2005/issuedtokensymmetricbasic256	WS-Trust 2005	SAML Token (Asym...)	Message
No	No	/adfs/services/trust/2005/issuedtokensymmetricbasic25...	WS-Trust 2005	SAML Token (Asym...)	Message
Yes	Yes	/adfs/services/trust/2005/issuedtokennixedasymmetricba...	WS-Trust 2005	SAML Token (Asym...)	Mixed
No	No	/adfs/services/trust/2005/issuedtokennixedasymmetricba...	WS-Trust 2005	SAML Token (Asym...)	Mixed
Yes	Yes	/adfs/services/trust/2005/issuedtokennivedsymmetricbas...	WS-Trust 2005	SAML Token (Sym...)	Mixed
No	No	/adfs/services/trust/2005/issuedtokennivedsymmetricbas...	WS-Trust 2005	SAML Token (Sym...)	Mixed
No	No	/adfs/services/trust/2005/issuedtokensymmetricbasic256	WS-Trust 2005	SAML Token (Sym...)	Message
No	No	/adfs/services/trust/2005/issuedtokensymmetricbasic256...	WS-Trust 2005	SAML Token (Sym...)	Message
No	No	/adfs/services/trust/2005/issuedtokensymmetricbasic...	WS-Trust 2005	SAML Token (Sym...)	Message
No	No	/adfs/services/trust/2005/issuedtokensymmetrictriples...	WS-Trust 2005	SAML Token (Sym...)	Message
No	No	/adfs/services/trust/2005/issuedtokennixedasymmetrictripl...	WS-Trust 2005	SAML Token (Sym...)	Mixed
No	No	/adfs/services/trust/2005/issuedtokennixedasymmetrictripl...	WS-Trust 2005	SAML Token (Sym...)	Mixed
Yes	No	/adfs/services/trust/13/kerberosmixed	WS-Trust 1.3	Kerberos	Mixed
No	No	/adfs/services/trust/13/certificate	WS-Trust 1.3	Certificate	Message
Yes	Yes	/adfs/services/trust/13/certificatemixed	WS-Trust 1.3	Certificate	Mixed
No	No	/adfs/services/trust/13/certificatetransport	WS-Trust 1.3	Certificate	Transport
No	No	/adfs/services/trust/13/username	WS-Trust 1.3	Username	Message
No	No	/adfs/services/trust/13/usernamebasictransport	WS-Trust 1.3	Username	Transport
Yes	Yes	/adfs/services/trust/13/usernamemixed	WS-Trust 1.3	Username	Mixed
No	No	/adfs/services/trust/13/issuedtokensymmetricbasic256	WS-Trust 1.3	SAML Token (Asym...)	Message
No	No	/adfs/services/trust/13/issuedtokensymmetricbasic256...	WS-Trust 1.3	SAML Token (Asym...)	Message
Yes	Yes	/adfs/services/trust/13/issuedtokennixedasymmetricbasic...	WS-Trust 1.3	SAML Token (Asym...)	Mixed
No	No	/adfs/services/trust/13/issuedtokennixedasymmetricbasic...	WS-Trust 1.3	SAML Token (Asym...)	Mixed
Yes	Yes	/adfs/services/trust/13/issuedtokennixedasymmetricbasic2...	WS-Trust 1.3	SAML Token (Sym...)	Mixed
No	No	/adfs/services/trust/13/issuedtokennixedasymmetricbasic2...	WS-Trust 1.3	SAML Token (Sym...)	Mixed
No	No	/adfs/services/trust/13/issuedtokensymmetricbasic256	WS-Trust 1.3	SAML Token (Sym...)	Message
No	No	/adfs/services/trust/13/issuedtokensymmetricbasic256h...	WS-Trust 1.3	SAML Token (Sym...)	Message
No	No	/adfs/services/trust/13/issuedtokerniedasymmetrictriples...	WS-Trust 1.3	SAML Token (Sym...)	Message
No	No	/adfs/services/trust/13/issuedtokerniedasymmetrictriplesha...	WS-Trust 1.3	SAML Token (Sym...)	Message
No	No	/adfs/services/trust/13/issuedtokennixedasymmetrictriples...	WS-Trust 1.3	SAML Token (Sym...)	Mixed
No	No	/adfs/services/trust/13/issuedtokennixedasymmetrictriples...	WS-Trust 1.3	SAML Token (Sym...)	Mixed
No	No	/adfs/services/trust/13/windows	WS-Trust 1.3	Windows	Message
No	No	/adfs/services/trust/13/windowsvmixed	WS-Trust 1.3	Windows	Mixed
No	No	/adfs/services/trust/13/windowstransport	WS-Trust 1.3	Windows	Transport
Yes	No	/adfs/services/trust/http/windows	WS-Trust 2005	Local Windows	Message
No	No	/adfs/services/trust/artifactresolution	SAML-ArtifactResolution	Anonymous	Transport
Yes	Yes	/adfs/oauth2/	OAuth	Anonymous	Transport
Metadata					
Yes	Yes	/adfs/services/trust/mex	WS-MEX	Anonymous	Transport
Yes	Yes	/FederationMetadata/2007-06/FederationMetadata.xml	Federation Metadata	Anonymous	Transport
Yes	No	/adfs/federationserverservice.asmx	ADFS 1.0 Metadata	Anonymous	Transport
OpenID Connect					
Yes	Yes	/adfs/well-known/openid-configuration	OpenID Connect Discover	Anonymous	Transport

4. Make a note of the URL Path for Type SAML 2.0/WS-Federation. (If you are using the default settings, this will be /adfs/ls/.)

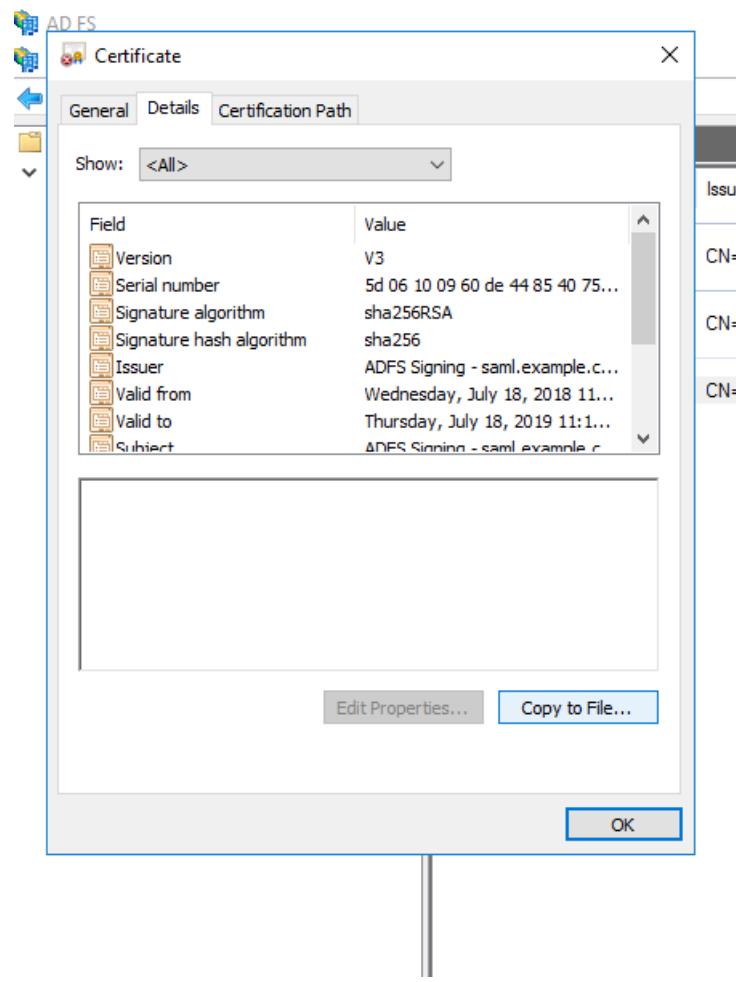
5. Switch from "Endpoints" to "Certificates" and choose the one under Token-signing.



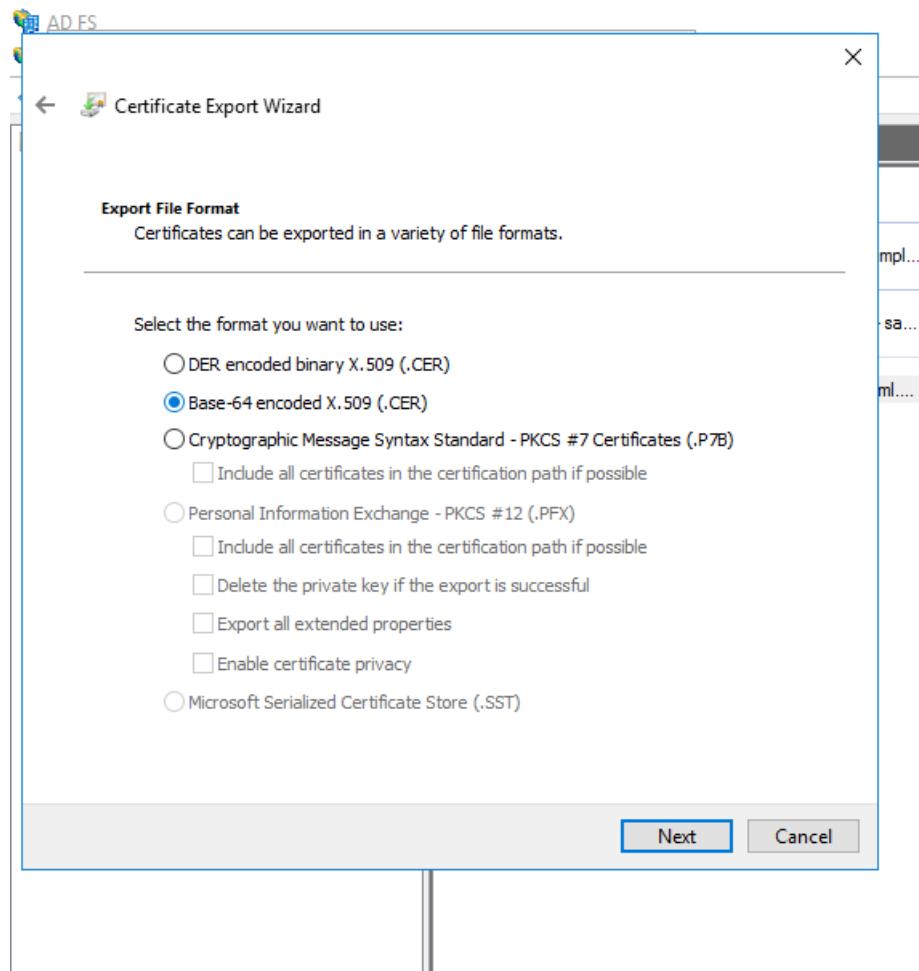
6. Right click "View Certificate".



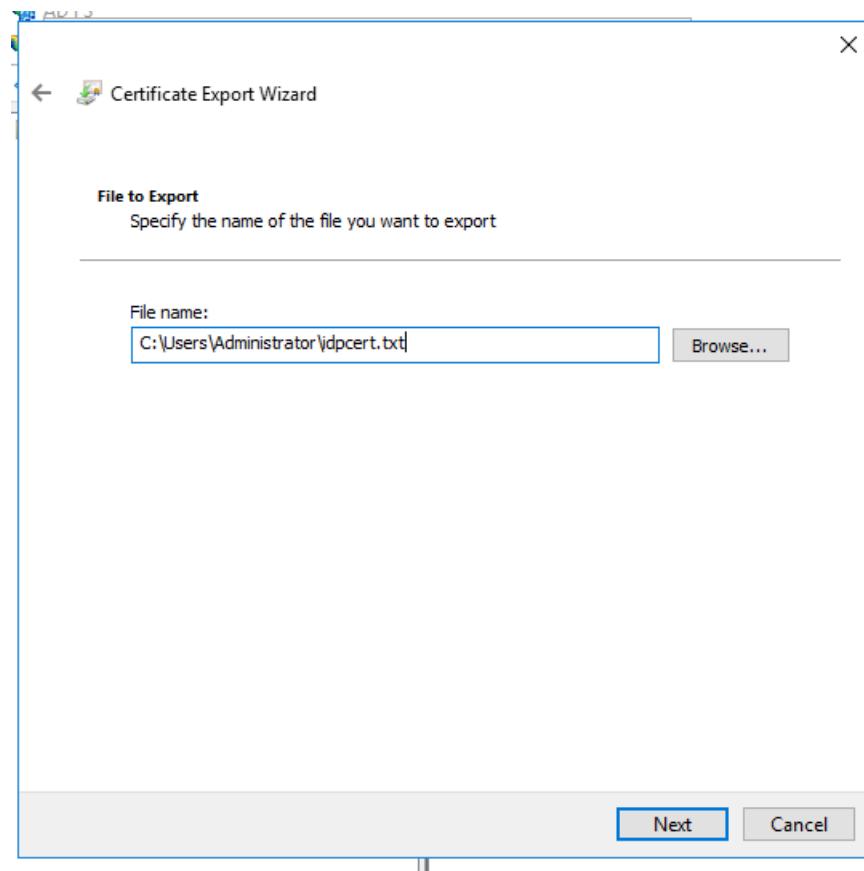
7. In the Certificate dialog, select the Details tab and click "Copy to File".



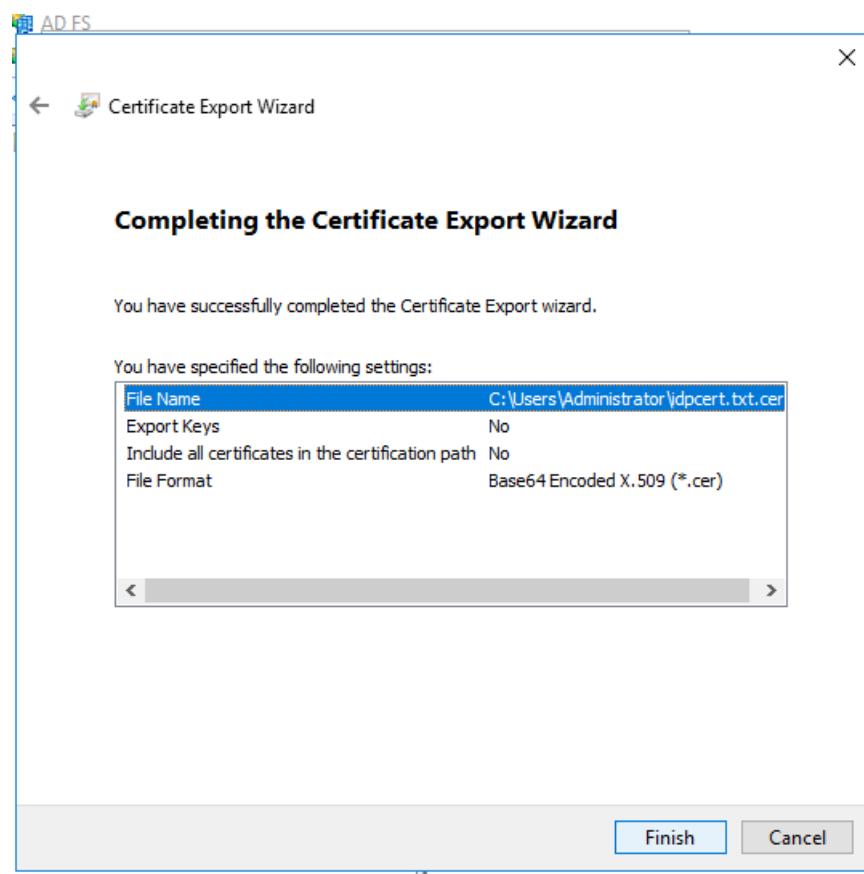
8. In the Certificate Export Wizard, click "Next", select "Base-64 encoded X.509 (.CER)" and click "Next" again.



9. Pick a location to save the file and click "Next".



10. Review the settings and click "Finish".



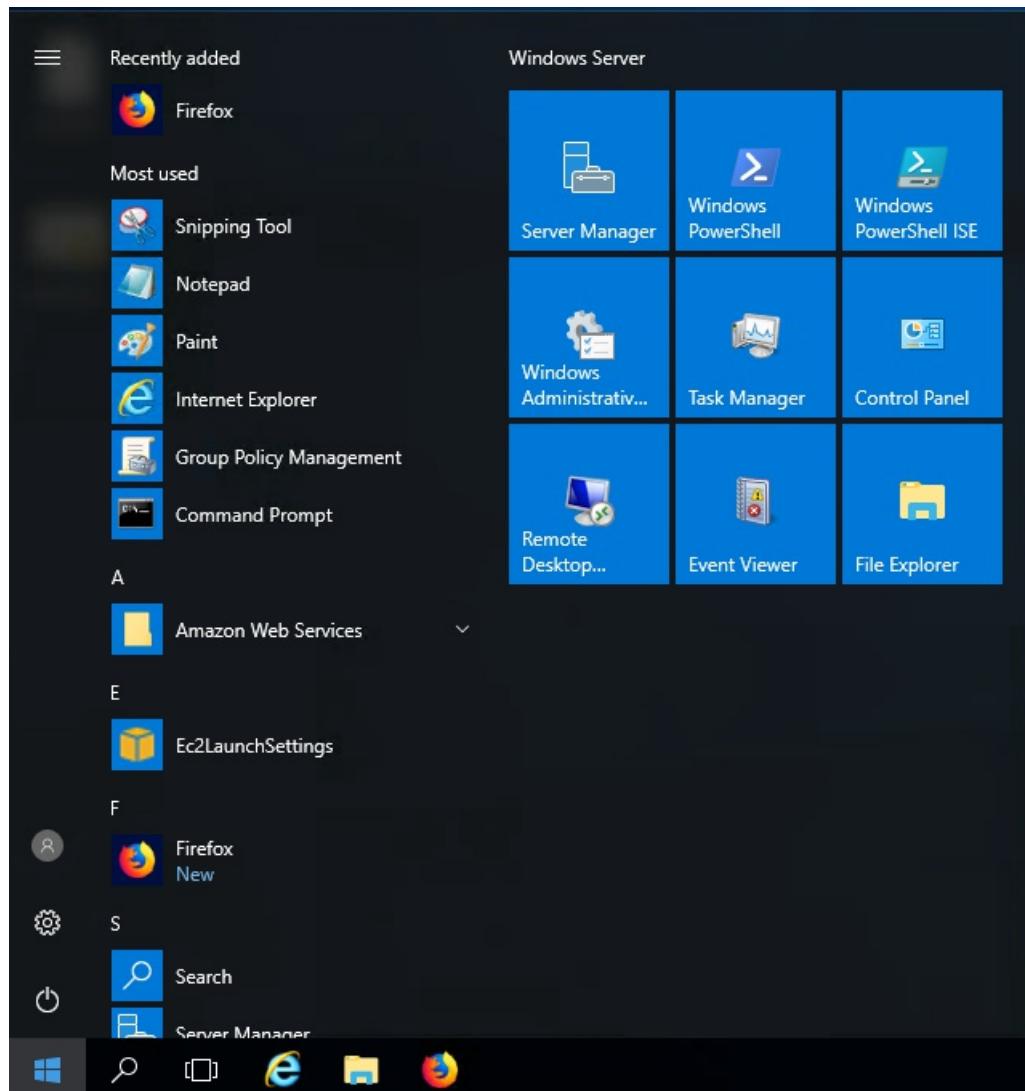
Configure PTFE

1. Visit <https://<TFE HOSTNAME>/app/admin/saml>.
2. Set "Single Sign-on URL" to <https://<ADFS hostname>/<URL Path>>, using the path you noted above in step 4.
3. Set "Single Log-out URL" to <https://<ADFS hostname>/<URL Path>?wa=wsignin1.0>. (Note that this is the same path with an additional URL parameter.)
4. Paste the contents of the saved certificate in "IDP Certificate".
5. Scroll to the bottom of the screen and click "Save SAML Settings".

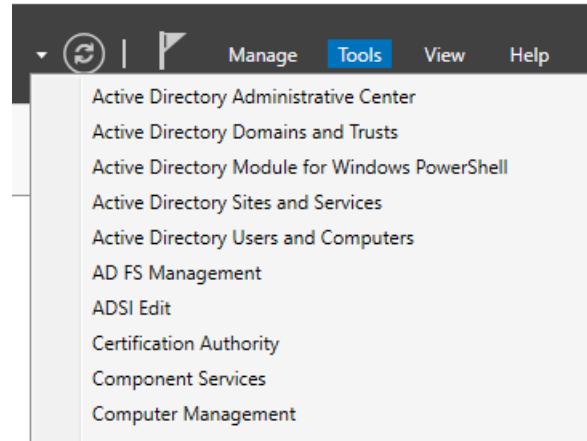
Configure ADFS

Configure the Relying Party (RP) Trust

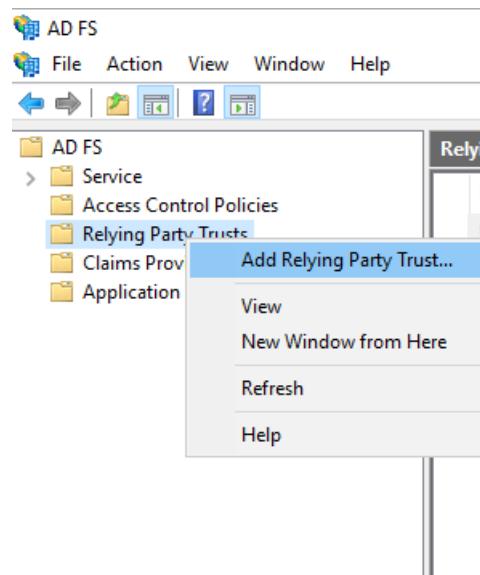
1. On the ADFS server, start the Server Manager.



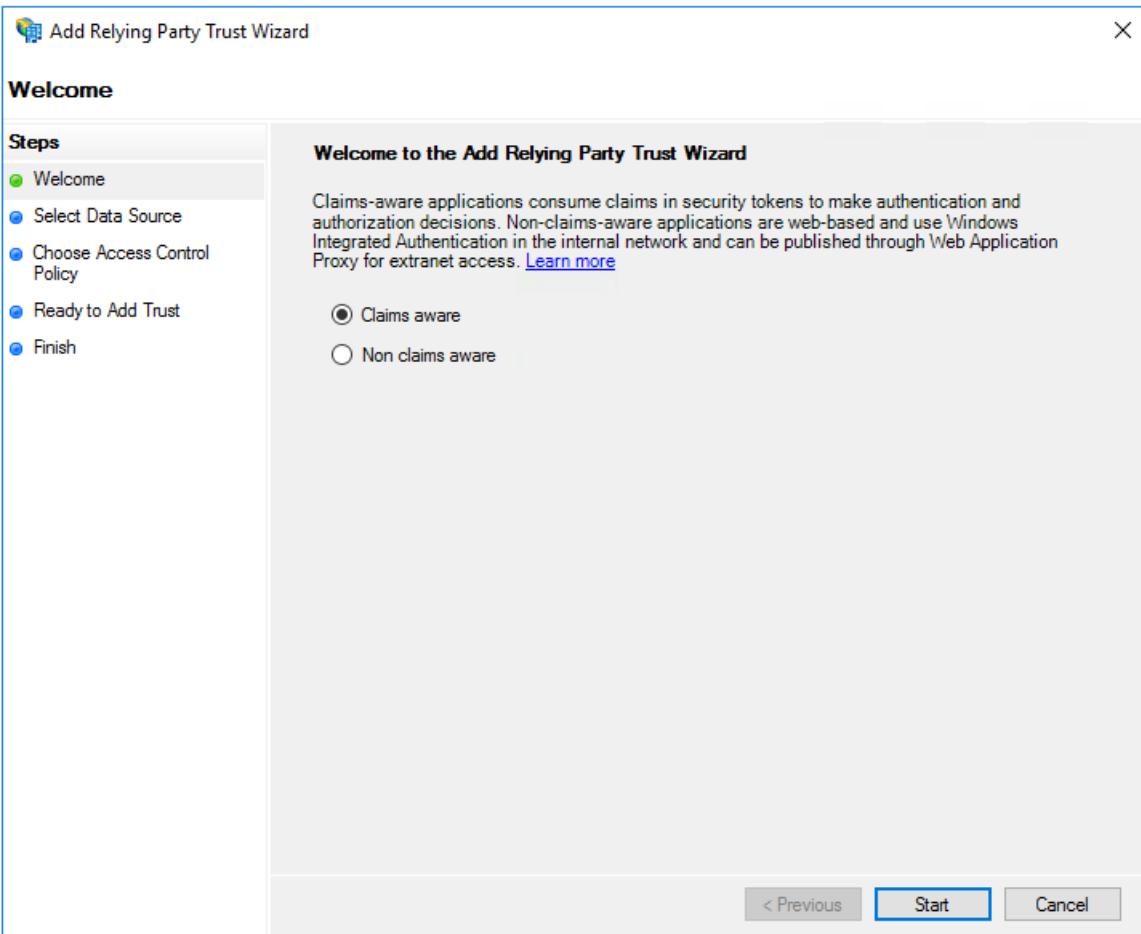
2. Click "Tools" -> "AD FS Management".



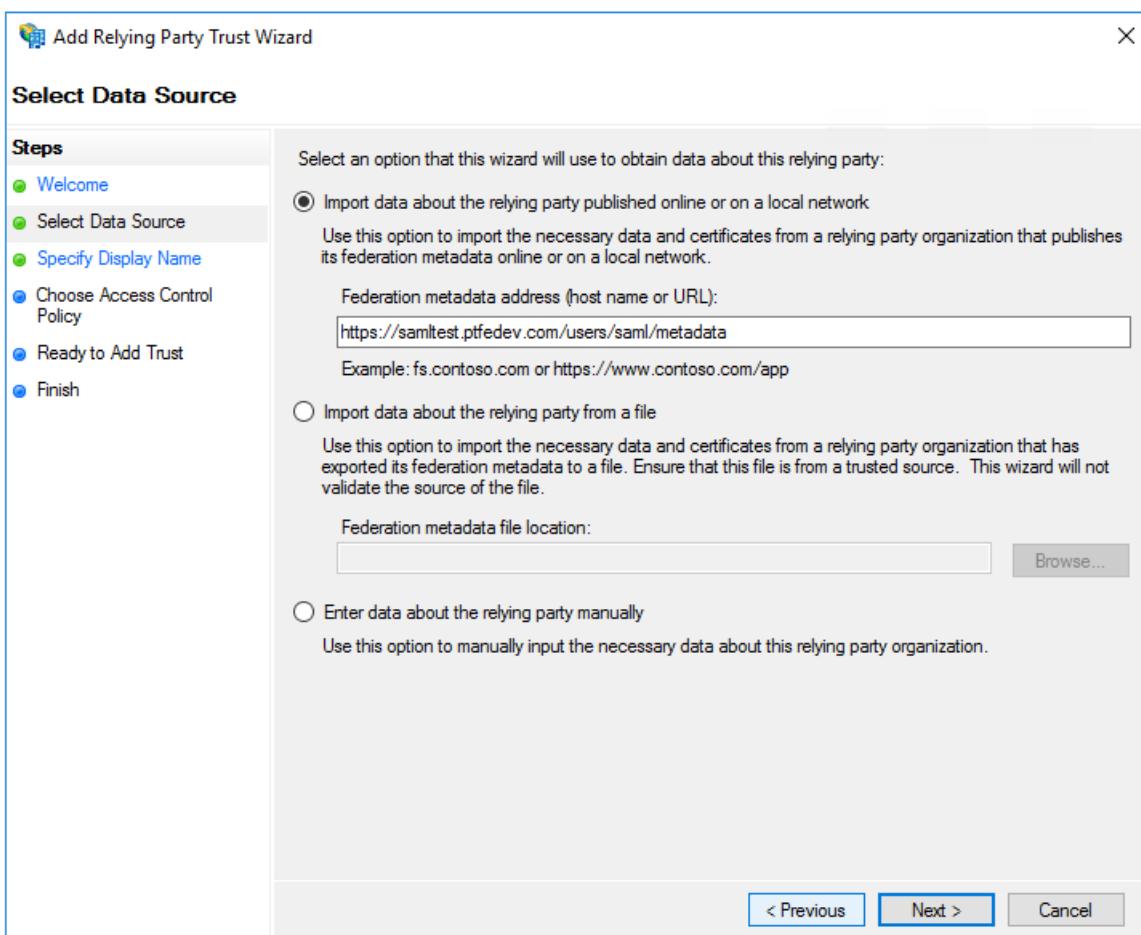
3. Right-click "Relying Party Trusts" and then click "Add Relying Party Trust".



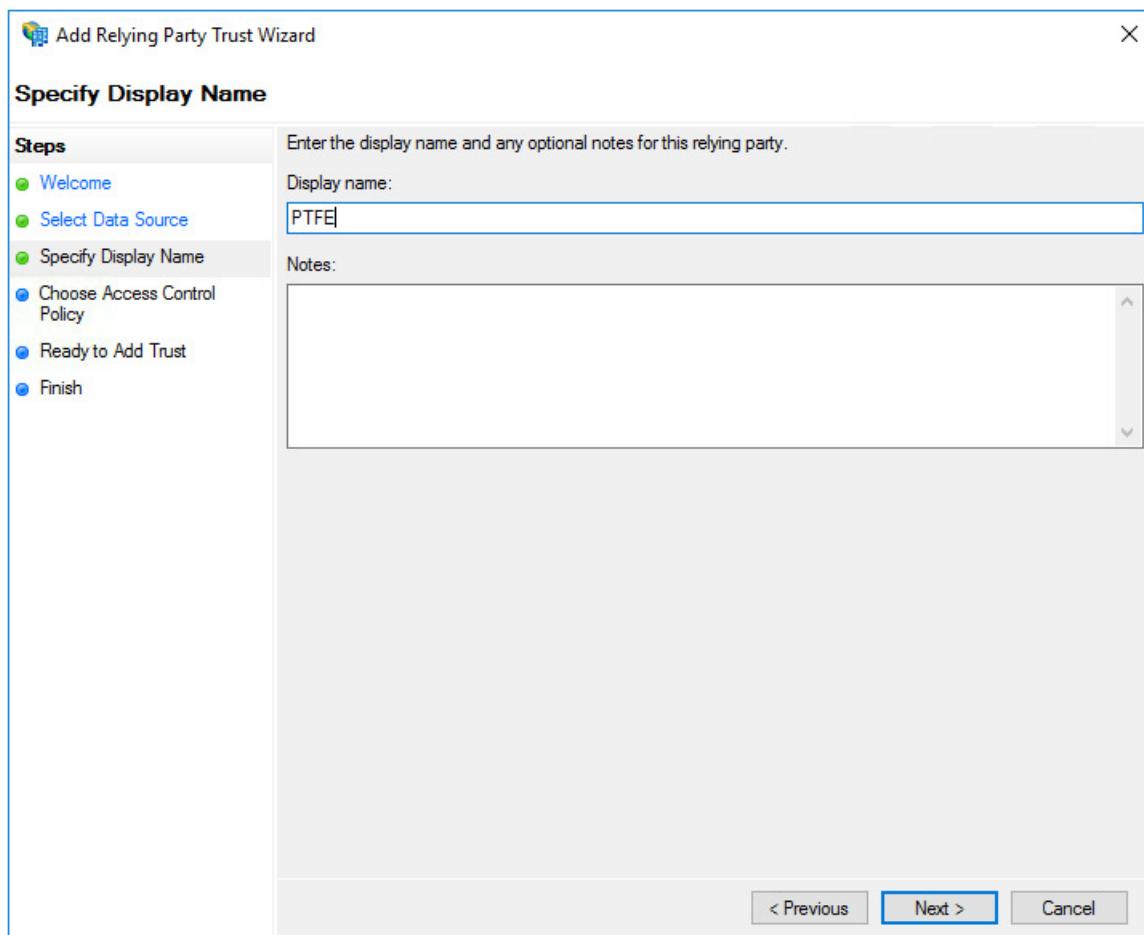
4. In the Add Relying Party Trust Wizard, select "Claims aware" and click "Start".



5. Next, select "Import data about the relying party published online or on a local network", and in the text box, enter `https://<TFE HOSTNAME>/users/saml/metadata`.



6. Click "Next", type a display name used to identify the RP trust, and click "Next" again.



7. In the "Choose Access Control Policy" screen, choose one that matches your security policy, and click "Next".

Add Relying Party Trust Wizard

Choose Access Control Policy

Steps

- Welcome
- Select Data Source
- Specify Display Name
- Choose Access Control Policy**
- Ready to Add Trust
- Finish

Choose an access control policy:

Name	Description
Permit everyone	Grant access to everyone.
Permit everyone and require MFA	Grant access to everyone and requir
Permit everyone and require MFA for specific group	Grant access to everyone and requir
Permit everyone and require MFA from extranet access	Grant access to the intranet users an
Permit everyone and require MFA from unauthenticated devices	Grant access to everyone and requir
Permit everyone and require MFA, allow automatic device registr...	Grant access to everyone and requir
Permit everyone for intranet access	Grant access to the intranet users.
Permit specific group	Grant access to users of one or more

Policy

```
Permit everyone
```

I do not want to configure access control policies at this time. No user will be permitted access for this application.

< Previous Next > Cancel

8. Review the settings and click "Next".

Add Relying Party Trust Wizard

Ready to Add Trust

Steps

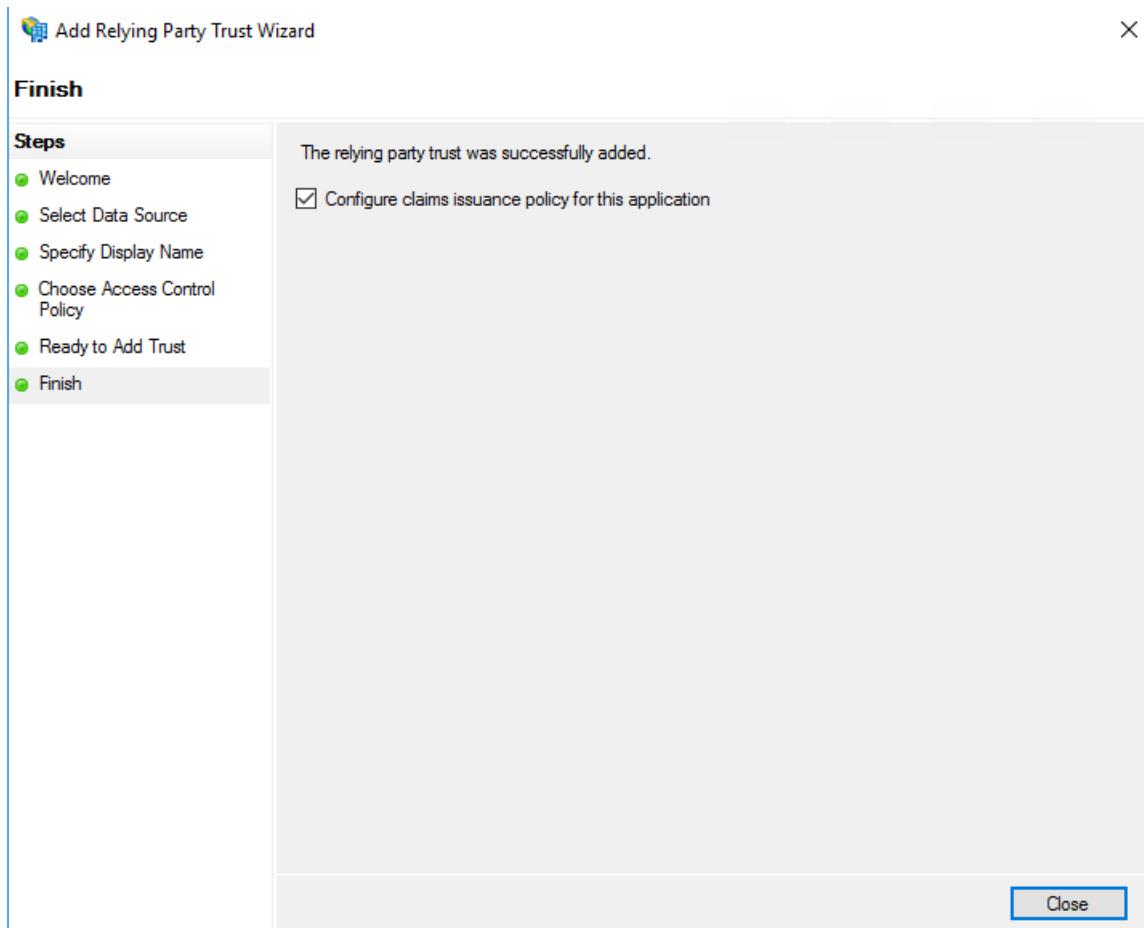
- Welcome
- Select Data Source
- Specify Display Name
- Choose Access Control Policy
- Ready to Add Trust**
- Finish

The relying party trust has been configured. Review the following settings, and then click Next to add the relying party trust to the AD FS configuration database.

Monitoring	Identifiers	Encryption	Signature	Accepted Claims	Organization	Endpoints	Notes
Specify the monitoring settings for this relying party trust.							
Relying party's federation metadata URL: <input type="text" value="https://samltest.pfedev.com/users/saml/metadata"/>							
<input checked="" type="checkbox"/> Monitor relying party <input checked="" type="checkbox"/> Automatically update relying party							
This relying party's federation metadata data was last checked on: 7/19/2018							
This relying party was last updated from federation metadata on: 7/19/2018							

< Previous Next > Cancel

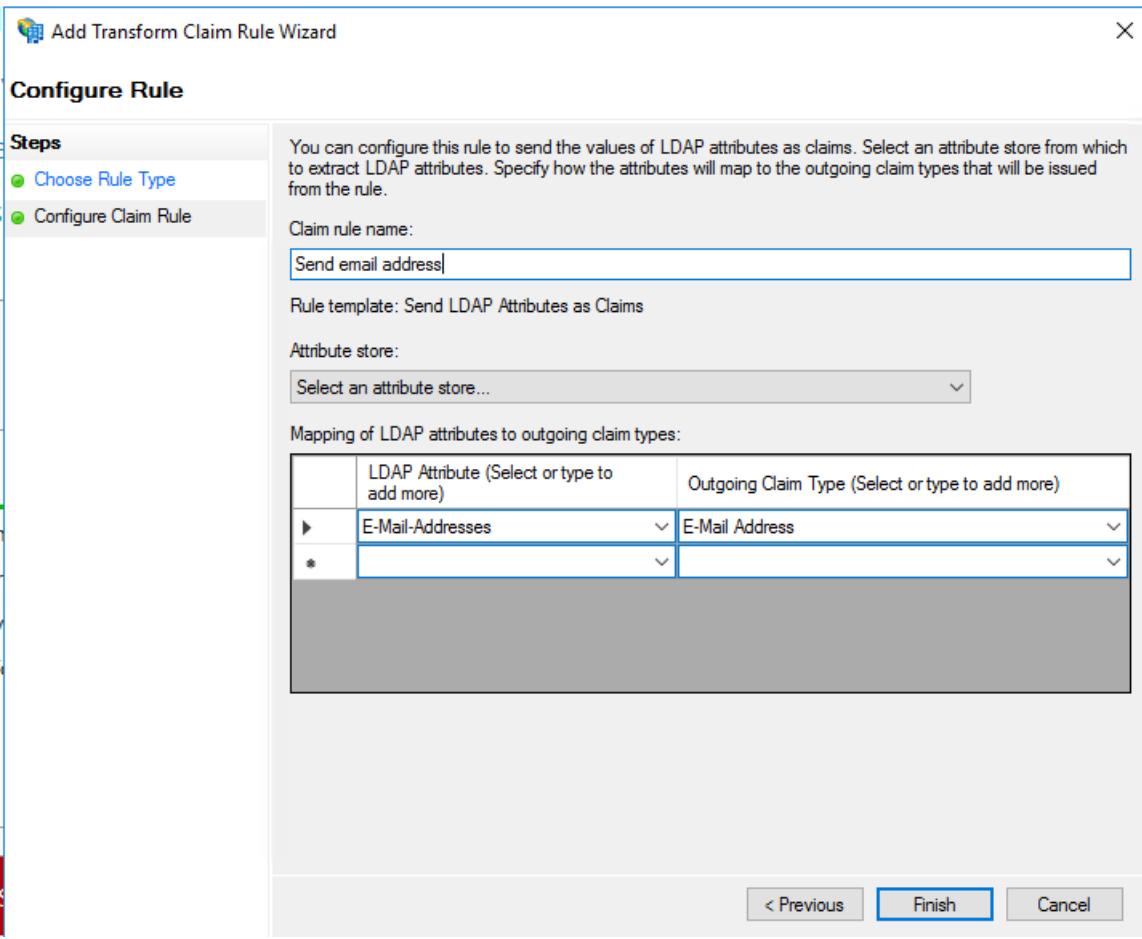
9. Finally, make sure "Configure claims issuance policy for this application" is checked and click "Close". This opens the Claim Issuance Policy editor for the RP trust just configured.



Configure Claim Issuance

LDAP Attributes as Claims

1. Click "Add Rule", and then select "Send LDAP Attributes as Claims" from the `Claim rule template` dropdown. Click "Next".
2. Set a name used to identify the claim rule.
3. Set the attribute store to "Active Directory".
 - o From the `LDAP Attribute` column, select "E-Mail Addresses".
 - o From the `Outgoing Claim Type`, select "E-Mail Address".



4. Click "Finish".

Transform Incoming Claims

1. Click "Add Rule", and then select "Transform an Incoming Claim" from the Claim rule template dropdown. Click "Next".

Select Rule Template**Steps**

- Choose Rule Type
- Configure Claim Rule

Select the template for the claim rule that you want to create from the following list. The description provides details about each claim rule template.

Claim rule template:

Transform an Incoming Claim

Claim rule template description:

Using the Transform an Incoming Claim rule template you can select an incoming claim, change its claim type, and optionally change its claim value. For example, you can use this rule template to create a rule that will send a role claim with the same claim value of an incoming group claim. You can also use this rule to send a group claim with a claim value of "Purchasers" when there is an incoming group claim with a value of "Admins". Multiple claims with the same claim type may be emitted from this rule. Sources of incoming claims vary based on the rules being edited.

< Previous

Next >

Cancel

2. Set a name used to identify the claim rule.

- o Select "E-mail Address" as the Incoming Claim Type.
- o Select "Name ID" as the Outgoing Claim Type.
- o Select "Email" for Outgoing Name ID Format.

 Add Transform Claim Rule Wizard

X

Configure Rule

Steps

- Choose Rule Type
- Configure Claim Rule

You can configure this rule to map an incoming claim type to an outgoing claim type. As an option, you can also map an incoming claim value to an outgoing claim value. Specify the incoming claim type to map to the outgoing claim type and whether the claim value should be mapped to a new claim value.

Claim rule name:

Rule template: Transform an Incoming Claim

Incoming claim type:

Incoming name ID format:

Outgoing claim type:

Outgoing name ID format:

Pass through all claim values

Replace an incoming claim value with a different outgoing claim value

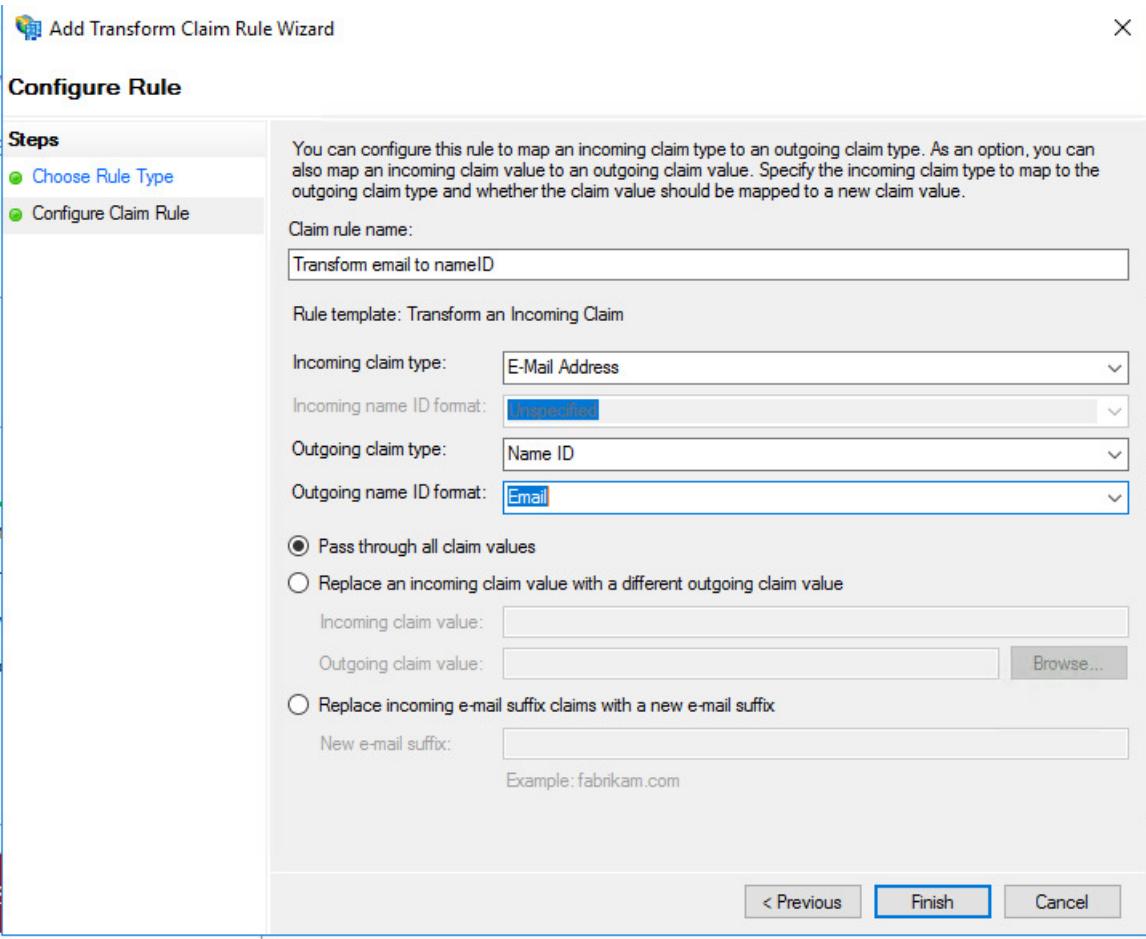
Incoming claim value:

Outgoing claim value:

Replace incoming e-mail suffix claims with a new e-mail suffix

New e-mail suffix:
Example: fabrikam.com

< Previous Cancel



3. Click "Finish".

Send Group Membership as a Claim

1. Click "Add Rule", and then select "Send Group Membership as a Claim" from the `Claim rule template` dropdown.
Click "Next".
2. Click "Browse" and locate the AD User group that contains all PTFE admins.

Add Transform Claim Rule Wizard

X

Configure Rule

Steps

- Choose Rule Type
- Configure Claim Rule

You can configure this rule to send a claim based on a user's Active Directory group membership. Specify the group that the user is a member of, and specify the outgoing claim type and value to issue.

Claim rule name:

Rule template: Send Group Membership as a Claim

User's group:

Browse...

Outgoing claim type:



Outgoing name ID format:



Outgoing claim value:

< Previous

Finish

Cancel

- Set Outgoing claim type to MemberOf.
- Set Outgoing claim value to site-admins.

Edit Rule - admin group claim

X

You can configure this rule to send a claim based on a user's Active Directory group membership. Specify the group that the user is a member of, and specify the outgoing claim type and value to issue.

Claim rule name:

Rule template: Send Group Membership as a Claim

User's group:

Browse...

Outgoing claim type:



Outgoing name ID format:



Outgoing claim value:

View Rule Language...

OK

Cancel

3. Click "Finish".

Test configured SAML login

At this point SAML is configured. Follow these instructions to log in ([/docs/enterprise/saml/login.html](#)) to Terraform Enterprise.

Okta Configuration

Follow these steps to configure Okta as the identity provider (IdP) for Private Terraform Enterprise (PTFE).

Configure a New Okta SAML Application

1. In Okta's web interface, go to the "Applications" tab and click "Create New App".

The screenshot shows the Okta Applications page. At the top, there is a search bar labeled "Search for an application" and a navigation menu with links for Help and Support, Sign out, Dashboard, Directory, Applications, Security, Reports, Settings, and My Applications. Below the search bar, there is a "Can't find an app?" section with a "Create New App" button. To the right, there is a grid of application cards. Each card includes the application logo, name, verification status (e.g., Okta Verified), and an "Add" button. The cards visible include TELADOC, &frankly, 10000ft, 101domains.com, 123RF, 15Five, and 1&1 E-mail. On the left side of the main content area, there are filters for Integration Properties (Any, Supports SAML, Supports Provisioning) and Categories (All, Application Delivery Controllers, CRM, CRM, Sales, Marketing, Collaboration, Consumer, Content Management). The "All" category filter is currently selected.

2. Select "Web" as the platform and "SAML 2.0" as the sign on method, then click "Create".

The screenshot shows the "Create a New Application Integration" dialog box overlying the Okta Applications page. The dialog has a title bar "Create a New Application Integration" with a close button. It contains two main sections: "Platform" and "Sign on method". Under "Platform", a dropdown menu is set to "Web". Under "Sign on method", there are three options: "Secure Web Authentication (SWA)", "SAML 2.0", and "OpenID Connect". The "SAML 2.0" option is selected, with a descriptive text below it stating: "Uses the SAML protocol to log users into the app. This is a better option than SWA, if the app supports it." At the bottom right of the dialog are "Create" and "Cancel" buttons. The background of the dialog shows the same application list and filters as the main Okta page, with the "15Five" application card visible.

3. In the "General Settings" page, enter an app name, then click "Next".

The screenshot shows the Okta interface for creating a new SAML integration. At the top, there's a navigation bar with links for Dashboard, Directory, Applications, Security, Reports, Settings, Help and Support, Sign out, and My Applications. Below the navigation, the title 'Create SAML Integration' is displayed, followed by a three-step wizard: 'General Settings' (selected), 'Configure SAML', and 'Feedback'. The 'General Settings' step contains fields for 'App name' (set to 'Terraform Enterprise'), 'App logo (optional)' (with a placeholder icon and a 'Browse...' button), and 'App visibility' (checkboxes for 'Do not display application icon to users' and 'Do not display application icon in the Okta Mobile app'). At the bottom of this step are 'Cancel' and 'Next' buttons.

4. In the "Configure SAML" page, configure the following settings with the specified values:

- o **Single sign on URL:** `https://<TFE HOSTNAME>/users/saml/auth` (listed as "ACS consumer (recipient) URL" in TFE's SAML settings).
- o **Use this for Recipient URL and Destination URL** (checkbox): enabled.
- o **Audience URI (SP Entity ID):** `https://<TFE HOSTNAME>/users/saml/metadata` (listed as "Metadata (audience) URL" in TFE's SAML settings).
- o **Name ID format** (drop-down): EmailAddress (the full name for this format in the SAML specification is `urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress`).
- o **Application username** (drop-down): Email

The screenshot shows the Okta interface for creating a SAML integration. At the top, there's a blue header bar with the Okta logo and navigation links: Dashboard, Directory, Applications, Security, Reports, Settings, Help and Support, and Sign out. Below the header, a title says "Create SAML Integration". A progress bar at the top indicates three steps: 1. General Settings (selected), 2. Configure SAML, and 3. Feedback.

A SAML Settings

GENERAL

- Single sign on URL: https://example.com/users/saml/auth
 - Use this for Recipient URL and Destination URL
 - Allow this app to request other SSO URLs
- Audience URI (SP Entity ID): https://example.com/users/saml/metadata
- Default RelayState: (empty field)
- Name ID format: EmailAddress
- Application username: Email

Show Advanced Settings

ATTRIBUTE STATEMENTS (OPTIONAL) [LEARN MORE](#)

What does this form do?
This form generates the XML needed for the app's SAML request.

Where do I find the info this form needs?
The app you're trying to integrate with should have its own documentation on using SAML. You'll need to find that doc, and it should outline what information you need to specify in this form.

Okta Certificate
Import the Okta certificate to your Identity Provider if required.
[Download Okta Certificate](#)

5. Still in the "Configure SAML" page, configure a group attribute statement to report which teams a user belongs to.

Under the "Group Attribute Statements (Optional)" header, configure the statement as follows:

- **Name:** MemberOf (This is the default name for TFE's group attribute (/docs/enterprise/saml/attributes.html); the name of this attribute can be changed in TFE's SAML settings (/docs/enterprise/saml/configuration.html) if necessary.)
- **Name format** (drop-down): Basic
- **Filter:** A filter type and filter value that will match all of the relevant groups that each user belongs to. The exact filter expression depends on how your Okta groups are configured, and which subset of groups you want to expose to TFE. Note that TFE ignores group names that do not correspond to existing TFE teams; see Team Membership Mapping (/docs/enterprise/saml/team-membership.html) for more details.

ATTRIBUTE STATEMENTS (OPTIONAL) [LEARN MORE](#)

Name	Name format (optional)	Value
(empty)	Unspecified	(empty)

Add Another

GROUP ATTRIBUTE STATEMENTS (OPTIONAL)

Name	Name format (optional)	Filter
MemberOf	Basic	Starts with

Add Another

Show Advanced Settings

B Preview the SAML assertion generated from the information above

[Preview the SAML Assertion](#)

This shows you the XML that will be used in the assertion - use it to verify the info you entered above

Previous Cancel Next

6. Still in the "Configure SAML" page, optionally configure a site admin permissions attribute statement. This statement determines which users can administer the entire PTFE instance (see Administering Private Terraform Enterprise ([/docs/enterprise/private/admin/index.html](#)) for more information about site admin permissions). Under the "Attribute Statements (Optional)" header, configure a statement as follows:

- **Name:** SiteAdmin (This is the default name for TFE's site admin attribute (/docs/enterprise/saml/attributes.html); the name of this attribute can be changed in TFE's SAML settings (/docs/enterprise/saml/configuration.html) if necessary.)
 - **Name Format** (drop-down): Basic
 - **Value:** An Okta expression (https://developer.okta.com/reference/okta_expression_language/) that will evaluate to a boolean: true for every user who should have site admin permissions, but false for any users who should not have site admin permissions. The exact expression depends on the user properties you use to manage admin permissions.

7. Preview the SAML response and make sure it matches your expectations.

8. Finish configuring the SAML app in Okta, then copy the provided endpoint URLs and certificate to your Terraform Enterprise SAML settings at `https://<TFE_HOSTNAME>/app/admin/saml`. TFE requires a single sign-on URL, a single log-out URL, and a PEM (base64) encoded X.509 certificate.

Example SAMLResponse


```
saml2Conditions NotBefore="2018-05-30T15:11:50.514Z" NotOnOrAfter="2018-05-30T15:52.00.000Z" xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
  <saml2:AudienceRestriction>
    <saml2:Audience>https://example.com/users/saml/metadata</saml2:Audience>
  </saml2:AudienceRestriction>
</saml2:Conditions>
<saml2:AuthnStatement AuthnInstant="2018-05-30T15:11:50.514Z" SessionIndex="_a6d4052d-4dca-4816-b811-a81834681d40" xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
  <saml2:AuthnContext>
    <saml2:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport</saml2:AuthnContextClassRef>
  </saml2:AuthnContext>
</saml2:AuthnStatement>
<saml2:AttributeStatement xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
  <saml2:Attribute Name="Username" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
    <saml2:AttributeValue xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">new_username</saml2:AttributeValue>
  </saml2:Attribute>
  <saml2:Attribute Name="MemberOf" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
    <saml2:AttributeValue xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">devs</saml2:AttributeValue>
    <saml2:AttributeValue xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">reviewers</saml2:AttributeValue>
  </saml2:Attribute>
</saml2:AttributeStatement>
</saml2:Assertion>
</saml2p:Response>
```

OneLogin Configuration

Follow these steps to configure OneLogin as the identity provider (IdP) for Terraform Enterprise (TFE).

1. Add a OneLogin app by going to Apps > Add Apps then searching for "SAML Test Connector (IdP)".
2. In the "Info" tab, enter an app name for Terraform Enterprise in the "Display Name" field.

The screenshot shows the 'Info' tab of the OneLogin app configuration. The 'Display Name' field contains 'SAML Test Connector (IdP)'. The 'Tab' dropdown is set to 'HashiCorp'. The 'Visible in portal' toggle is turned on. There are two icon options: 'Rectangular Icon' (selected) and 'Square Icon'. Below the icons, there is a note about file upload requirements. The 'Description' section has a text area for notes, which is currently empty. A 'Notes' section at the bottom right includes a 'IDEAS' button.

3. In the "Configuration" tab, configure the service provider audience and recipient URLs. These are shown in your Terraform Enterprise SAML settings at <https://<TFE HOSTNAME>/app/admin/saml>.

The screenshot shows the 'Configuration' tab of the OneLogin app configuration. Under 'Application Details', the 'RelayState' field is empty. The 'Audience' field contains 'https://example.com/users/saml/metadata'. The 'Recipient' field contains 'https://example.com/users/saml/auth'. The 'ACS (Consumer) URL Validator*' field contains a regular expression '^.*\$'. A note states: '*Required. Regular expression - Validates the ACS URL when initiated by an AuthnRequest'. The 'ACS (Consumer) URL*' field contains 'https://example.com/users/saml/auth'. A note states: '*Required'. The 'Single Logout URL' field is empty. A 'Notes' section at the bottom right includes a 'IDEAS' button.

4. In the "Parameters" tab, map the Nameld and MemberOf parameters.

onelogin

USERS APPS DEVICES ACTIVITY SETTINGS

MORE ACTIONS ▾ SAVE

← SAML Test Connector (IdP)

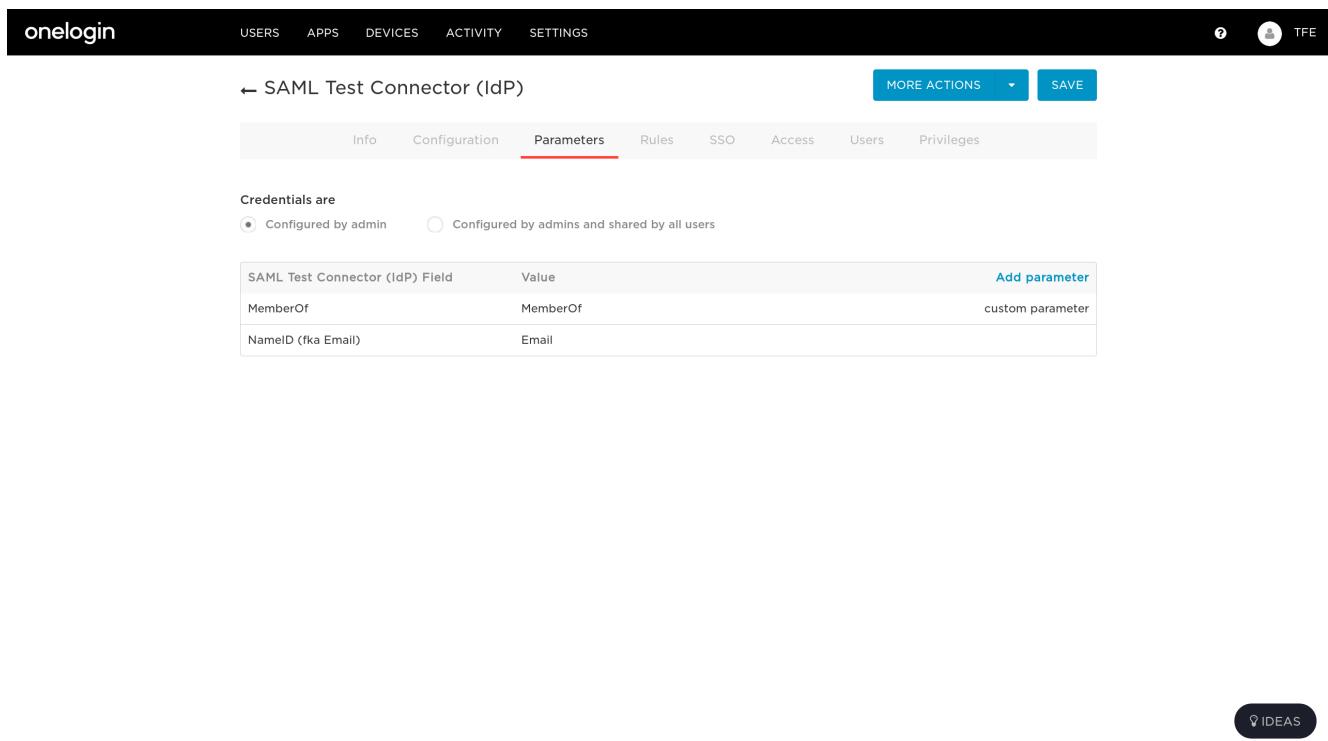
Info Configuration Parameters Rules SSO Access Users Privileges

Credentials are

Configured by admin Configured by admins and shared by all users

SAML Test Connector (IdP) Field	Value	Add parameter
MemberOf	MemberOf	custom parameter
NameID (fka Email)	Email	

IDEAS



onelogin

USERS APPS DEVICES ACTIVITY SETTINGS

MORE ACTIONS ▾ SAVE

← SAML Test Connector (IdP)

Info

Credentials are

Configured by admin

SAML Test Connector (IdP) Fields

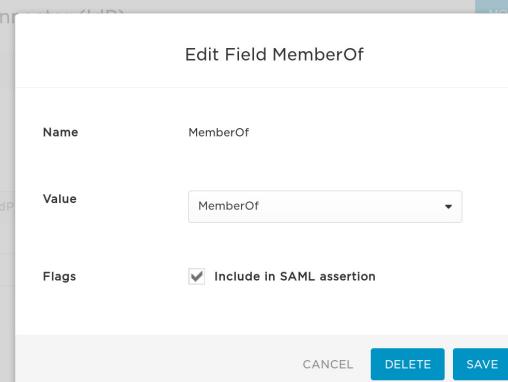
MemberOf
NameID (fka Email)

Edit Field MemberOf

Name	MemberOf
Value	MemberOf
Flags	<input checked="" type="checkbox"/> Include in SAML assertion

CANCEL DELETE SAVE

IDEAS



5. In the "SSO" tab, copy the endpoint URLs and certificate, then paste them into your Terraform Enterprise SAML settings at `https://<TFE HOSTNAME>/app/admin/saml`. (Use the certificate's "View Details" link to copy its PEM-encoded text representation.)

onelogin

USERS APPS DEVICES ACTIVITY SETTINGS

MORE ACTIONS ▾ SAVE

← SAML Test Connector (IdP)

Info Configuration Parameters Rules SSO Access Users Privileges

Enable SAML2.0

Sign on method SAML2.0

X.509 Certificate

Standard Strength Certificate (2048-bit)

Change | View Details

SAML Signature Algorithm

SHA-1

Issuer URL

https://app.onelogin.com/saml/metadata/742781

SAML 2.0 Endpoint (HTTP)

https://terraform-saml-test-dev.onelogin.com/trust/saml2/ht

SLO Endpoint (HTTP)

https://terraform-saml-test-dev.onelogin.com/trust/saml2/ht

IDEAS

onelogin

USERS APPS DEVICES ACTIVITY SETTINGS

DELETE SAVE

← Standard Strength Certificate (2048-bit)

Key length 2048-bit

SHA fingerprint Fingerprint

SHA1 00:00:00:00

X.509 Certificate

X.509 Certificate

-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----

X.509 PEM DOWNLOAD

Apps using this certificate

SAML Test Connector (...) SAML Test Connector (...) SAML Test Connector (...)

IDEAS

6. In the "Access" tab, enable access for specific roles.

onelogin

USERS APPS DEVICES ACTIVITY SETTINGS

MORE ACTIONS ▾ SAVE

← SAML Test Connector (IdP)

Info Configuration Parameters Rules SSO Access Users Privileges

Policy By default all your users will be using this policy to log into this app
-- None --

Role-Based Policy Make exception for the users in a role and select a policy for them. Make sure the roles are enabled from the ROLE ACCESS SECTION below

Select a Role will use Select a Policy Remove

Roles

admins ✓ Default devs ✓ reviewers

IDEAS

This screenshot shows the 'Access' tab of the SAML Test Connector configuration. It includes sections for 'Policy' (a dropdown set to 'None') and 'Role-Based Policy' (a section for selecting roles and policies). Below these are tabs for 'Roles' where 'admins' and 'devs' are selected, and 'reviewers' is listed. A large 'IDEAS' button is visible on the right.

7. In the "Users" tab, add users and specify their roles.

onelogin

USERS APPS DEVICES ACTIVITY SETTINGS

MORE ACTIONS ▾ SAVE

← SAML Test Connector (IdP) - Localhost Login For Bob

Info Configuration Parameters Rules SSO Access Users Privileges

Edit SAML Test Connector (IdP) - Localhost Login For Bob

Enabled Allow users to sign in

NameID (fka Email) bob@example.com

NameID Format: Email

MemberOf admins

Manually updating this field overrides any mapping CANCEL SAVE

IDEAS

This screenshot shows a modal dialog titled 'Edit SAML Test Connector (IdP) - Localhost Login For Bob'. It contains fields for 'Enabled' (checked), 'NameID (fka Email)' (bob@example.com), 'MemberOf' (admins), and a note 'Manually updating this field overrides any mapping'. At the bottom are 'CANCEL' and 'SAVE' buttons. The background shows the main SAML connector configuration page.

The screenshot shows the OneLogin interface for managing SAML connectors. At the top, there's a navigation bar with links for USERS, APPS, DEVICES, ACTIVITY, and SETTINGS. On the far right, there are icons for help, user profile, and TFE (Terraform Enterprise). Below the navigation, the title is '← SAML Test Connector (IdP)'. To the right are buttons for 'MORE ACTIONS' and 'SAVE'. A horizontal menu bar below the title includes 'Info', 'Configuration', 'Parameters', 'Rules', 'SSO', 'Access', 'Users' (which is underlined in red), and 'Privileges'. Underneath this is a search bar labeled 'Search' and dropdown menus for 'All roles' and 'All groups'. The main content area is titled 'User' and contains a single entry: 'Bob'. At the bottom of this section, it says 'Showing 1-2 of 2 users'. In the bottom right corner of the page, there's a dark button with the word 'IDEAS' and a small icon.

Terraform Enterprise SAML SSO settings

Verify the endpoint URLs, certificate, and attribute mappings are correct in the SAML SSO settings.



SITE ADMINISTRATION

Users

Organizations

Workspaces

Runs

Migrations

Terraform Versions

Settings

SAML

SMTP

Twilio

Release: b19146e

Identity Provider Configuration

Configuring SAML allows users to log into Terraform Enterprise using a Single Sign-On provider.

ACS CONSUMER (RECIPIENT) URL<https://nfagerlund.tfe.zone/users/saml/auth> ⓘ**METADATA (AUDIENCE) URL**<https://nfagerlund.tfe.zone/users/saml/metadata> ⓘ**NAMEID FORMAT**<urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress> ⓘ**Download SAML Metadata**

SAML Settings

 Enable SAML single sign-on **Enable SAML debugging**

When sign-on fails, the SAMLResponse XML will be displayed on the login page.

Identity Provider Settings**SINGLE SIGN-ON URL**<https://terraform-saml-test-dev.onelogin.com/trust/saml2/http-post/sso/746804>**SINGLE LOG-OUT URL**<https://terraform-saml-test-dev.onelogin.com/trust/saml2/http-redirect/slo/746804>**IDP CERTIFICATE**

```
-----BEGIN CERTIFICATE-----  
...  
...  
...  
-----END CERTIFICATE-----
```

Additional Resources

The following pages in the OneLogin documentation are relevant to using SAML SSO with Terraform Enterprise:

- Create Mappings to Automatically Assign Roles to Users (<https://support.onelogin.com/hc/en-us/articles/211531266-User-Provisioning-An-End-to-End-Flow#create-mappings>)
- Using Regex to Provision Members of AD/LDAP Groups to New App Groups (<https://support.onelogin.com/hc/en-us/articles/209887623-Using-Regex-to-Provision-Members-of-AD-LDAP-Groups-to-New-App-Groups>)

Example SAML Response

```
<samlp:Response xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" ID="Rcfa2c10eeafdf57496999c691655247bdf16b3549" Version="2.0" IssueInstant="2018-02-28T16:05:13Z" Destination="https://example.com/users/saml/auth" InResponseTo="_0ac000d0-ae00-0fe0-000c-0f00a000d000">  
  <saml:Issuer>https://example.com/saml/metadata/1</saml:Issuer>  
  <samlp:Status>  
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>  
  </samlp:Status>
```



```
<vsz>/saml:AttributeValue>
    <saml:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">r
eviewers</saml:AttributeValue>
</saml:Attribute>
</saml:AttributeStatement>
</saml:Assertion>
</samlp:Response>
```

Identity Provider Configuration

Single sign-on setup instructions for specific identity providers (IdP).

Identity Providers

- ADFS (/docs/enterprise/saml/identity-provider-configuration-adfs.html)
- Okta (/docs/enterprise/saml/identity-provider-configuration-okta.html)
- OneLogin (/docs/enterprise/saml/identity-provider-configuration-onelogin.html)

Request and response

To help you with the configuration of your identity provider, here are example documents for the request from Terraform Enterprise before sign-on and the response from the identity provider after sign-on.

Example AuthnRequest

```
<samlp:AuthnRequest AssertionConsumerServiceURL='https://app.terraform.io/users/saml/auth' Destination='https://example.onelogin.com/trust/saml2/http-post/sso/1' ID='_000eda0a-c0f0-00cf-b0a0-c00b000d000f' IssueInstant='2018-02-28T02:16:25Z' Version='2.0' xmlns:saml='urn:oasis:names:tc:SAML:2.0:assertion' xmlns:samlp='urn:oasis:names:tc:SAML:2.0:protocol'>
  <saml:Issuer>https://app.terraform.io/users/saml/metadata</saml:Issuer>
  <samlp:NameIDPolicy AllowCreate='true' Format='urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress' />
</samlp:AuthnRequest>
```

Example SAMLResponse

```
<samlp:Response xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" ID="Rcfa2c10eeaf57496999c691655247bdf16b3549" Version="2.0" IssueInstant="2018-02-28T16:05:13Z" Destination="https://app.terraform.io/users/saml/auth" InResponseTo="_0ac000d0-ae00-0fe0-000c-0f00a00d0000">
  <saml:Issuer>https://app.terraform.io/saml/metadata/1</saml:Issuer>
  <samlp:Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
  </samlp:Status>
  <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Version="2.0" ID="pxf0000b000-0000-0000-0000-0000bdb000ce" IssueInstant="2018-02-28T16:05:13Z">
    <saml:Issuer>https://app.onelogin.com/saml/metadata/1</saml:Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:SignedInfo>
        <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <ds:Reference URI="#pxf0000b000-0000-0000-00f0-0000bdb000ce">
          <ds:Transforms>
            <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
            <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
        </ds:Reference>
      </ds:SignedInfo>
    </ds:Signature>
  </saml:Assertion>
</samlp:Response>
```


Login with SAML

Once SAML is configured users can visit <https://<TFE HOSTNAME>/session> to login.

They can follow the link to complete the SAML login process with the identity provider. If the user is logging in for the first time, an account will be created from them in Terraform Enterprise. Their username will be autogenerated from their email address using the text before the @. The username will only contain alphanumeric characters, -, or _. All invalid characters will be converted to _.

API Token Expiration

When SAML is initially enabled, or when a user's SAML-authenticated web session expires, existing user API tokens are also temporarily disabled until they reauthenticate at <https://<TFE HOSTNAME>/session>. This is because Terraform Enterprise relies on your identity provider for team membership mapping (</docs/enterprise/saml/team-membership.html>), and a user might have been added to or removed from some teams since their session expired. This restriction only affects user tokens, not team or organization tokens (</docs/enterprise/users-teams-organizations/service-accounts.html>).

The API token session timeout is a site-wide setting that is configurable in the admin settings at <https://<TFE HOSTNAME>/app/admin/saml>.

Team Membership Mapping

Terraform Enterprise (TFE) can automatically add users to teams based on their SAML assertion, so you can manage team membership in your directory service.

Configuring Team Membership Mapping

Team membership mapping is controlled by including the Team Attribute Name ([/docs/enterprise/saml/configuration.html](#)) in your SAML assertion. You must specify the name of a SAML attribute in the Team Attribute Name ([/docs/enterprise/saml/configuration.html](#)) setting, and make sure the AttributeStatement in the SAMLResponse contains a list of AttributeValue items.

If the Team Attribute Name ([/docs/enterprise/saml/configuration.html](#)) is included in your SAML assertion, users logging in via SAML are automatically added to the teams included in their assertion, and automatically removed from any teams that *aren't* included in their assertion. Note that this overrides any manually set team memberships; whenever the user logs in, their team membership is adjusted to match their SAML assertion.

Any team names that don't match existing teams are ignored; TFE will not automatically create new teams.

To disable team membership mapping, remove the Team Attribute Name ([/docs/enterprise/saml/configuration.html](#)) from your SAML assertion. With mapping disabled, TFE won't automatically manage team membership on login, and you can manually add users to teams via the organization settings page.

Team Names

TFE expects the team names in the team membership SAML attribute to exactly match TFE's own team names. You cannot specify aliases for teams.

Note that team names are unique across an organization but not necessarily unique across a whole TFE instance. If a user is a member of multiple TFE organizations, their SAML assertion might add them to similarly-named teams. Keep this in mind when naming your teams.

Managing Membership of the Owners Team

Since the "owners" team ([/docs/enterprise/users-teams-organizations/teams.html#the-owners-team](#)) is especially important, TFE defaults to NOT managing its membership via SAML. Unless you specifically enable it, TFE won't automatically add or remove any owners, and you can manually manage membership via the teams page.

You can enable automatic membership in the owners team (on a per-organization basis) by explicitly specifying an alias (role ID) for it. On your organization settings page, click "Teams" and then click the owners team. If SAML is enabled, there will be a "SAML Role ID" field. Enter a legal team name as an ID and click "Save." The ID can be "owners," but it cannot conflict with any other team name.

Before enabling membership mapping for owners, double-check that your chosen role ID appears in the SAML assertion for users who should be owners. It's worth some extra effort to avoid accidentally removing people from the owners team.

ORGANIZATION SETTINGS

 modern-bank-0 ✓

Profile

Teams

OAuth Configuration

API Token

User Sessions

Manage SSH Keys

Sentinel Policy

Team: owners

SAML Role ID

ROLE ID

Role ID

Save

Users with the Role ID value provided in the SAML **MemberOf** attribute will automatically be added to this team.
[Read more about SAML Role ID Mapping.](#)

Add new team members

USERNAME

Username

Add user

The username of the user you wish to add to this team. Send new users the [signup link](#).

Members (10)



Dario67

Remove

Site Admins

If the "Site Admin Role" setting (in TFE's SAML settings ([/docs/enterprise/saml/configuration.html](#))) is enabled, the selected special team name (default: site-admins) will add a user as a site admin ([/docs/enterprise/private/admin/index.html](#)) for the private Terraform Enterprise instance.

Note: Instead of treating site admins like a team, we recommend using the "Site Admin Attribute Name" setting, which manages admin access via a dedicated SAML attribute. If enabled, this attribute overrides the special site admin team name.

Site admins can also always log in to Terraform Enterprise directly. If they are disabled on the identity provider but still enabled in Terraform Enterprise and bypass SSO, they will still be able to log in.

Troubleshooting Guide

Note: Verify you are on release version 201807-2 as that is the version that has the debugging functionality that is described in this guide. If you would like assistance with upgrading, please contact support (</docs/enterprise/private/faq.html#support-for-private-terraform-enterprise>).

Disable SAML Single Sign-On

Before starting, disable SAML SSO by going to <https://<TFE HOSTNAME>/app/admin/saml> and unchecking the Enable SAML Single Sign-On checkbox. It's best to start from a clean setup.

Create a non-SSO admin account for recovery

Before proceeding with troubleshooting, create a non-SSO admin account that can be used to log in if admin access gets revoked for other admins. The email address assigned to this user should not be one that will be used for SAML.

Open <https://<TFE HOSTNAME>/account/new> to create the account. Make sure to grant admin access to this user and verify they can log in at <https://<TFE HOSTNAME>/>.

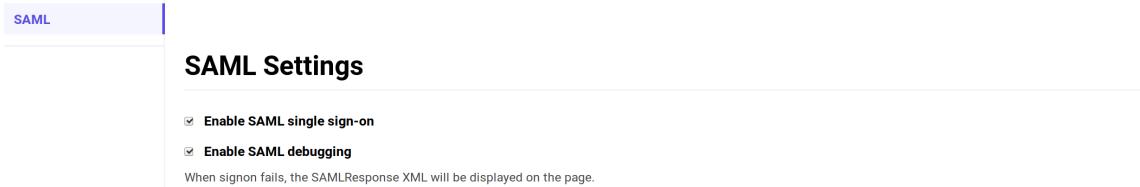
Enable SAML SSO and SAML debugging

Enable SAML SSO

Enable SAML SSO by following the configuration instructions (</docs/enterprise/saml/configuration.html>).

Enable SAML debugging

Enable SAML debugging by going to <https://<TFE HOSTNAME>/app/admin/saml>.



Test sign-on

Try signing on by going to <https://<TFE HOSTNAME>> and clicking the "Log in via SAML" button. Verify the page says 'WARNING: SAML debugging is enabled.'

If login fails, the SAMLResponse XML document sent from the identity provider is shown. The XML document may contain the user's username, list of roles, and other attributes. Checking the format of the email address and username and whether the desired list of roles is included may assist with debugging.

WARNING: SAML debugging is enabled

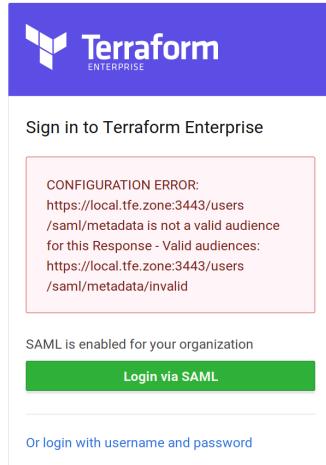
SAMLResponse

```
<samlp:Response xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" ID="R34e56f97cd0476bd09130e17e57ac83ae919a99" Version="2.0" IssueInstant="2018-07-05T16:41:40Z" Destination="https://local.tfe.zone:3443/users/saml/auth" InResponseTo="_fd2e8430-e7d8-47c4-87ac-87eba735b7c4"><saml:Issuer>https://app.onelogin.com/saml/metadata/757982</saml:Issuer><samlp>Status><samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/></samlp:Status><saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Version="2.0" ID="pfxa080b2fc-488f-0e26-ad8b-f9caab42a617" IssueInstant="2018-07-05T16:41:40Z"><saml:Issuer>https://app.onelogin.com/saml/metadata/757982</saml:Issuer><ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"><ds:SignedInfo><ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" /><ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/><ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" /></ds:SignedInfo><ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/><ds:DigestValue></ds:DigestValue></ds:Reference></ds:SignedInfo><ds:SignatureValue></ds:SignatureValue><ds:SignatureValue></ds:SignatureValue><ds:KeyInfo><ds:X509Data><ds:X509Certificate></ds:X509Certificate></ds:X509Data></ds:KeyInfo></ds:Signature><saml:Subject><saml:NameID Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">terraform-enterprise-accounts@hashicorp.com</saml:NameID><saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:beearer"><saml:SubjectConfirmationData NotOnOrAfter="2018-07-05T16:44:40Z" Recipient="https://local.tfe.zone:3443/users/saml/auth" InResponseTo="_fd2e8430-e7d8-47c4-87ac-87eba735b7c4"/></saml:SubjectConfirmation><saml:Conditions NotBefore="2018-07-05T16:38:40Z" NotOnOrAfter="2018-07-05T16:44:40Z"><saml:AudienceRestriction><saml:Audience>https://local.tfe.zone:3443/users/saml/metadata/invalid</saml:Audience></saml:AudienceRestriction></saml:Conditions><saml:AuthnStatement AuthnInstant="2018-07-05T16:41:39Z" SessionNotOnOrAfter="2018-07-06T16:41:40Z" SessionIndex=_7b07bf10-629d-0136-cef0-0a3d3846a8d6"><saml:AuthnContextClassRef></saml:AuthnContextClassRef></saml:AuthnContext></saml:AuthnStatement><saml:AttributeStatement><saml:Attribute Name="MemberOf" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic"><saml:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">tfe-test</saml:AttributeValue></saml:Attribute><saml:Attribute Name="Username" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic"><saml:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">tfe-test</saml:AttributeValue></saml:AttributeStatement></saml:Assertion></samlp:Response>
```

Processed attributes

```
Username: tfe-test
Roles: ["site-admins"]
```

If there is a configuration error, that is also shown on the login form.



Fix the configuration error and try to log in again.

Common configuration errors

Most errors will be from misconfiguration and will be shown in the red box on the Terraform Enterprise login form.

CONFIGURATION ERROR: `https://<TFE HOSTNAME>/metadata` is not a valid audience for this Response - Valid audiences: `https://<TFE HOSTNAME>/users/saml/metadata`

The audience URL was not configured correctly in the identity provider.

How to resolve: Open the Terraform Enterprise admin settings for SAML SSO, copy the ACS Consumer URL, then paste it into the identity provider settings.

CONFIGURATION ERROR: The response was received at https://<TFE HOSTNAME>/auth instead of https://<TFE HOSTNAME>/users/saml/auth

The recipient URL was not configured correctly in the identity provider.

How to resolve: Open the Terraform Enterprise admin settings for SAML SSO, copy the Metadata URL, then paste it into the identity provider settings.

ERROR: Validation failed: Email is invalid, Email is not a valid email address, Username has already been taken

NameID is invalid. It must be an email address.

How to resolve: Open the identity provider settings and configure email address as the value for NameID.

ERROR: Mail::AddressList can not parse !{onelogin:email}: Only able to parse up to "{onelogin:email}"

The NameID that was received was blank.

How to resolve: Open the identity provider settings and configure email address as the value for NameID.

ERROR: nested asn1 error

Terraform Enterprise was unable to determine the issuer of the SAML response.

How to resolve: Open the identity provider settings, copy the certificate, then paste it into the "IDP Certificate" field in Terraform Enterprise.

ERROR: Issuer of assertion not found or multiple

Terraform Enterprise was unable to determine the issuer of the SAML response.

How to resolve: The most common reason for this issue is that an F5 load balancer is not signing responses, resulting in the <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"> and related elements not being present. Follow the steps under **Configuring SAML SP Connectors** on Using APM as a SAML IdP (https://support.f5.com/kb/en-us/products/big-ip_apm/manuals/product/apm-authentication-single-sign-on-12-1-0/29.html), particularly step 9c. If you are not using an F5 as part of your SAML setup, see below to contact support.

Contacting support

If you're not able to resolve the error using the steps above, reach out to support (</docs/enterprise/private/faq.html#support-for-private-terraform-enterprise>). When contacting support, please provide:

- A screenshot of "SAML Response" and "Processed attributes" shown on the login page after failed login
- A screenshot of the error on the login page
- The SAMLResponse XML document (</docs/enterprise/saml/identity-provider-configuration.html#example-samlresponse>)
- A support bundle (</docs/enterprise/private/faq.html#diagnostics>)

Sentinel Overview

Sentinel (<https://www.hashicorp.com/sentinel>) is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products. It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

To learn how to use Sentinel and begin writing policies with the Sentinel language, see the Sentinel documentation (<https://docs.hashicorp.com/sentinel/writing/>).

You can also use the `tfe_sentinel_policy` (/docs/providers/tfe/r/sentinel_policy.html) resource from the Terraform Enterprise provider (</docs/providers/tfe/>) to upload a policy using Terraform itself.

Sentinel in Terraform Enterprise

Using Sentinel with Terraform Enterprise involves:

- Defining the Policies (</docs/enterprise/sentinel/import/index.html>) - Policies are defined using the policy language (<https://docs.hashicorp.com/sentinel/concepts/language>) with imports for parsing the Terraform plan, state and configuration.
- Managing the policies for organizations (</docs/enterprise/sentinel/manage-policies.html>) - Organization owners add policies to their organization by setting the policy name, policy file, and the enforcement level. They then group these policies into policy sets to define which workspaces the policies are checked against during runs.
- Enforcing policy checks on runs (</docs/enterprise/sentinel/enforce.html>) - Policies are checked when a run is performed, after the `terraform plan` but before it can be confirmed or the `terraform apply` is executed.

Enforce and Override Policies

Once a policy is added to an organization it is enforced on all runs.

The policy check will occur immediately after a plan is successfully executed in the run. If the plan fails, the policy check will not be performed. The policy check uses the generated tfplan file, simulated apply object ([/docs/enterprise/sentinel/import/tfplan.html#resource-applied-field](#)), state and configuration to verify the rules in each of the policies.

Enforcement level details can be found in the [Managing Policies](#) ([/docs/enterprise/sentinel/manage-policies.html](#)) documentation.

All `hard mandatory` and `soft mandatory` policies must pass in order for the run to continue to the "Confirm & Apply" state.

If a `soft mandatory` policy fails a member of the organization owners team will be presented with an "Override & Continue" button in the run. They have the ability to override the failed check and continue the execution of the run.

If an `advisory` fails, it will show the warning state in the run; however, the execution of the run will continue to the "Confirm & Apply" state. No user action is required to override or continue the run execution.

Example Policies

This page lists some example Sentinel policies. These examples are not exhaustive, but they demonstrate some of the most common use cases of Sentinel with Terraform Enterprise.

Important: These examples are a demonstration of Sentinel's features. They should not be used verbatim in your Terraform Enterprise organization. Make sure you fully understand the intent and behavior of a policy before relying on it in production.

Amazon Web Services

- Enforce owner allow list on aws_ami data source (<https://github.com/hashicorp/terraform-guides/blob/master/governance/aws/enforce-ami-owners.sentinel>)
- Enforce mandatory tags on instances (<https://github.com/hashicorp/terraform-guides/blob/master/governance/aws/enforce-mandatory-tags.sentinel>)
- Restrict availability zones (<https://github.com/hashicorp/terraform-guides/blob/master/governance/aws/restrict-aws-availability-zones.sentinel>)
- Disallow CIDR blocks (<https://github.com/hashicorp/terraform-guides/blob/master/governance/aws/restrict-aws-cidr-blocks.sentinel>)
- Restrict the type of instance to be provisioned (<https://github.com/hashicorp/terraform-guides/blob/master/governance/aws/restrict-aws-instance-type.sentinel>)
- Require VPCs to be tagged and have DNS hostnames enabled (<https://github.com/hashicorp/terraform-guides/blob/master/governance/aws/aws-vpcs-must-have-tags-and-enable-dns-hostnames.sentinel>)

Microsoft Azure

- Restrict VM images (<https://github.com/hashicorp/terraform-guides/blob/master/governance/azure/restrict-vm-image-id.sentinel>)
- Restrict the type of VM to be provisioned (<https://github.com/hashicorp/terraform-guides/blob/master/governance/azure/restrict-vm-size.sentinel>)
- Enforce limits on an ACS cluster (<https://github.com/hashicorp/terraform-guides/blob/master/governance/azure/acs-cluster-policy.sentinel>)
- Enforce limits on an AKS cluster (<https://github.com/hashicorp/terraform-guides/blob/master/governance/azure/aks-cluster-policy.sentinel>)

Google Cloud Platform

- Disallow CIDR blocks (<https://github.com/hashicorp/terraform-guides/blob/master/governance/gcp/block-allow-all-cidr.sentinel>)
- Enforce limits on a GKE cluster (<https://github.com/hashicorp/terraform-guides/blob/master/governance/gcp/gke->

cluster-policy.sentinel)

- Restrict the type of machine to be provisioned (<https://github.com/hashicorp/terraform-guides/blob/master/governance/gcp/restrict-machine-type.sentinel>)

VMware

- Require Storage DRS to be enabled (<https://github.com/hashicorp/terraform-guides/blob/master/governance/vmware/require-storage-drs.sentinel>)
- Restrict virtual disk size and type (<https://github.com/hashicorp/terraform-guides/blob/master/governance/vmware/restrict-virtual-disk-size-and-type.sentinel>)
- Restrict VM CPU count and memory (<https://github.com/hashicorp/terraform-guides/blob/master/governance/vmware/restrict-vm-cpu-and-memory.sentinel>)
- Enforce NFS 4.1 and Kerberos (https://github.com/hashicorp/terraform-guides/blob/master/governance/vmware/require_nfs41_and_kerberos.sentinel)

Defining Policies

Sentinel Policies for Terraform are defined using the Sentinel policy language (<https://docs.hashicorp.com/sentinel/language/>). A policy can include imports (<https://docs.hashicorp.com/sentinel/concepts/imports>) which enable a policy to access reusable libraries, external data and functions. Terraform Enterprise provides three imports to define policy rules for the configuration, state and plan.

- `tfplan` (/docs/enterprise/sentinel/import/tfplan.html) - This provides access to a Terraform plan, the file created as a result of `terraform plan`. The plan represents the changes that Terraform needs to make to infrastructure to reach the desired state represented by the configuration.
- `tfconfig` (/docs/enterprise/sentinel/import/tfconfig.html) - This provides access to a Terraform configuration, the set of "tf" files that are used to describe the desired infrastructure state.
- `tfstate` (/docs/enterprise/sentinel/import/tfstate.html) - This provides access to the Terraform state, the file used by Terraform to map real world resources to your configuration.

Note: Terraform Enterprise does not currently support custom imports.

Useful Idioms for Terraform Sentinel Policies

Terraform's internal data formats are complex, which means basic Sentinel policies for Terraform are more verbose than basic policies that use simpler data sources.

This will improve in future versions of Terraform and Sentinel; in the meantime, be aware of the following idioms as you start writing policies for Terraform.

To Find Resources, Iterate over Modules

The most basic Sentinel task for Terraform is to enforce a rule on all resources of a given type. Before you can do that, you need to get a collection of all the relevant resources.

The easiest way to do that is to copy a function like the following into any policies that examine every resource in a configuration:

```

# Get an array of all resources of the given type (or an empty array).
get_resources = func(type) {
    if length(tfplan.module_paths else []) > 0 { # always true in the real tfplan import
        return get_resources_all_modules(type)
    } else { # fallback for tests
        return get_resources_root_only(type)
    }
}

get_resources_root_only = func(type) {
    resources = []
    named_and_counted_resources = tfplan.resources[type] else {}
    # Get resource bodies out of nested resource maps, from:
    # {"name": {"0": {"applied": {...}, "diff": {...}}, "1": {...}}, "name": {...}}
    # to:
    # [{"applied": {...}, "diff": {...}}, {"applied": {...}, "diff": {...}}, ...]
    for named_and_counted_resources as _, instances {
        for instances as _, body {
            append(resources, body)
        }
    }
    return resources
}

get_resources_all_modules = func(type) {
    resources = []
    for tfplan.module_paths as path {
        named_and_counted_resources = tfplan.module(path).resources[type] else {}
        # Get resource bodies out of nested resource maps, from:
        # {"name": {"0": {"applied": {...}, "diff": {...}}, "1": {...}}, "name": {...}}
        # to:
        # [{"applied": {...}, "diff": {...}}, {"applied": {...}, "diff": {...}}, ...]
        for named_and_counted_resources as _, instances {
            for instances as _, body {
                append(resources, body)
            }
        }
    }
    return resources
}

```

Note: This example uses the tfplan import. You can easily substitute tfconfig or tfstate, depending on your needs.

Later, use the function to get a collection of resources:

```
aws_instances = get_resources("aws_instance")
```

This example function handles several things that are tricky about finding resources:

- It checks every module for resources (including the root module) by looping over the `module_paths` namespace. The top-level `resources` namespace is more convenient, but it only reveals resources from the root module.
- It unwraps the import's nested data structures, leaving only an array of resource bodies. The value of `tfplan.module(path).resources[type]` is a series of nested maps keyed by resource name and by `count` (/docs/configuration/resources.html#count), but the name and count are almost never relevant to a policy. Removing them early makes the rest of the policy more readable.
- It uses `else` expressions to recover from undefined values, for modules that don't have any resources of that type.

- It falls back to the `resources` namespace if the real `tfplan` import isn't available, to support testing (<https://docs.hashicorp.com/sentinel/writing/testing>). Since current versions of Sentinel don't allow you to mock `tfplan's module()` function, it isn't possible to test Sentinel code that accesses non-root modules. However, you can still test the rest of the policy by mocking resource data under the `resources` namespace.

Note: This example is checking whether it's in a test by looking for an empty `module_paths` namespace, which assumes that our organization is omitting that key in our mock data for Sentinel tests. For policies that need to test mocked `module_paths` data for other purposes, you might need to use a different method to check for the real Terraform imports.

To Test Resources, Use all/any Expressions

Once you have a collection of resources, you usually want to test some property of each resource in the collection — for example, the planned final value of a particular resource attribute.

The most concise tool for this is Sentinel's `all` and `any` expressions

(<https://docs.hashicorp.com/sentinel/language/boolexpr#any-all-expressions>). For example:

```
# Allowed Types
allowed_types = [
    "t2.small",
    "t2.medium",
    "t2.large",
]

# Rule to restrict instance types
instance_type_allowed = rule {
    all get_resources("aws_instance") as r {
        r.applied.instance_type in allowed_types
    }
}
```

Note: This example assumes that you are unwrapping the import's nested maps when finding resources, as described in the previous section. If you leave the nested maps in place, you will have to use nested `all/any` expressions to reach the resource attributes.

Mocking tfconfig Data

Below is a sample of what a mock for the `tfconfig` (/docs/enterprise/sentinel/import/tfconfig.html) import would look like. You can use this data with the mocking or testing features of the Sentinel Simulator (<https://docs.hashicorp.com/sentinel/commands/>) to create test data for rules.

For more details on how to work with mock data in Sentinel, see the section on [Mocking Imports](#) (<https://docs.hashicorp.com/sentinel/writing/imports#mocking-imports>) in the official Sentinel documentation (<https://docs.hashicorp.com/sentinel/>).

Current mock limitations

- As functions cannot be mocked in the current Sentinel testing framework, the `module()` (/docs/enterprise/sentinel/import/tfconfig.html#function-module-) function is not available. As a result, only root module data can be mocked at this time.

```
{  
  "mock": {  
    "tfconfig": {  
      "module_paths": [  
        [],  
        [  
          "foo"  
        ]  
      ],  
      "data": {  
        "null_data_source": {  
          "baz": {  
            "config": {  
              "inputs": [  
                {  
                  "bar_id": "${null_resource.bar.id}",  
                  "foo_id": "${null_resource.foo.id}"  
                }  
              ]  
            },  
            "provisioners": []  
          }  
        }  
      },  
      "modules": {  
        "foo": {  
          "config": {  
            "bar": "baz"  
          },  
          "source": "./foo"  
        }  
      },  
      "providers": {  
        "aws": {  
          "config": {},  
          "version": "~\u003e 1.34",  
          "alias": {  
            "east": {  
              "config": {  
                "region": "us-east-1"  
              },  
              "version": ""  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

```
        }
    },
    "null": {
        "config": {},
        "version": "",
        "alias": {}
    }
},
"resources": {
    "null_resource": {
        "bar": {
            "config": {
                "triggers": [
                    {
                        "foo_id": "${null_resource.foo.id}"
                    }
                ]
            },
            "provisioners": []
        },
        "foo": {
            "config": {
                "triggers": [
                    {
                        "foo": "bar"
                    }
                ]
            },
            "provisioners": [
                {
                    "config": {
                        "command": "echo hello"
                    },
                    "type": "local-exec"
                }
            ]
        }
    }
},
"variables": {
    "foo": {
        "default": "bar",
        "description": "foobar"
    },
    "map": {
        "default": {
            "foo": "bar",
            "number": 42
        },
        "description": ""
    },
    "number": {
        "default": "42",
        "description": ""
    }
}
}
```

Mocking tfplan Data

Below is a sample of what a mock for the `tfplan` (</docs/enterprise/sentinel/import/tfplan.html>) import would look like. You can use this data with the mocking or testing features of the Sentinel Simulator (<https://docs.hashicorp.com/sentinel/commands/>) to create test data for rules.

For more details on how to work with mock data in Sentinel, see the section on [Mocking Imports](#) (<https://docs.hashicorp.com/sentinel/writing/imports#mocking-imports>) in the official Sentinel documentation (<https://docs.hashicorp.com/sentinel/>).

NOTE: The top-level `config` and `state` keys are not included in this mock. Their layouts are identical to those found in the `tfconfig` mock (</docs/enterprise/sentinel/import/mock-tfconfig.html>) and the `tfstate` mock (</docs/enterprise/sentinel/import/mock-tfstate.html>).

Current mock limitations

There are currently some limitations mocking `tfplan` data in Sentinel. These issues will be fixed in future releases of the import and core runtime.

- As functions cannot be mocked in the current Sentinel testing framework, the `module()` (</docs/enterprise/sentinel/import/tfplan.html#function-module->) function is not available. As a result, only root module data can be mocked at this time.
- Resource and data source (</docs/enterprise/sentinel/import/tfplan.html#namespace-resources-data-sources>) count keys (NUMBER in TYPE.NAME[NUMBER]), which are actually represented as integers, cannot be represented accurately in JSON, and as a result, mocks that depend on count keys explicitly (example: `tfplan.resources.null_resource.foo[0]`) will not work properly with mocks.

```
{
  "mock": {
    "tfplan": {
      "terraform_version": "0.11.7",
      "variables": {
        "foo": "bar",
        "map": {
          "foo": "bar",
          "number": 42
        },
        "number": "42"
      },
      "module_paths": [
        [],
        [
          [
            "foo"
          ]
        ],
        [
          "path": [],
          "data": {
            "null_data_source": {
              "baz": {
                "0": {
                  "diff": {
                    "has_computed_default": {
                      "old": "",
                      "new": ""
                    }
                  }
                }
              }
            }
          }
        ]
      ]
    }
  }
}
```

```
        "computed": true
    },
    "id": {
        "old": "",
        "new": "",
        "computed": true
    },
    "inputs.%": {
        "old": "",
        "new": "",
        "computed": true
    },
    "outputs.%": {
        "old": "",
        "new": "",
        "computed": true
    },
    "random": {
        "old": "",
        "new": "",
        "computed": true
    }
},
"applied": {
    "has_computed_default": "74D93920-ED26-11E3-AC10-0800200C9A66",
    "id": "74D93920-ED26-11E3-AC10-0800200C9A66",
    "inputs": {},
    "outputs": {},
    "random": "74D93920-ED26-11E3-AC10-0800200C9A66"
}
}
}
}
},
"resources": {
    "null_resource": {
        "bar": {
            "0": {
                "diff": {
                    "id": {
                        "old": "",
                        "new": "",
                        "computed": true
                    },
                    "triggers.%": {
                        "old": "",
                        "new": "",
                        "computed": true
                    }
                },
                "applied": {
                    "id": "74D93920-ED26-11E3-AC10-0800200C9A66",
                    "triggers": {}
                }
            }
        },
        "foo": {
            "0": {
                "diff": {
                    "id": {
                        "old": "",
                        "new": "",
                        "computed": true
                    },
                    "triggers.%": {
                        "old": ""
                    }
                }
            }
        }
    }
}
```


Mocking tfstate Data

Below is a sample of what a mock for the `tfstate` (/docs/enterprise/sentinel/import/tfstate.html) import would look like. You can use this data with the mocking or testing features of the Sentinel Simulator (<https://docs.hashicorp.com/sentinel/commands/>) to create test data for rules.

For more details on how to work with mock data in Sentinel, see the section on [Mocking Imports](#) (<https://docs.hashicorp.com/sentinel/writing/imports#mocking-imports>) in the official Sentinel documentation (<https://docs.hashicorp.com/sentinel/>).

Current mock limitations

There are currently some limitations mocking `tfstate` data in Sentinel. These issues will be fixed in future releases of the import and core runtime.

- As functions cannot be mocked in the current Sentinel testing framework, the `module()` (/docs/enterprise/sentinel/import/tfstate.html#function-module-) function is not available. As a result, only root module data can be mocked at this time.
- Resource and data source (/docs/enterprise/sentinel/import/tfstate.html#namespace-resources-data-sources) count keys (NUMBER in `TYPE.NAME[NUMBER]`), which are actually represented as integers, cannot be represented accurately in JSON and as a result, mocks that depend on count keys explicitly (example: `tfstate.resources.null_resource.foo[0]`) will not work properly with mocks.

```
{  
  "mock": {  
    "tfstate": {  
      "module_paths": [  
        [],  
        [  
          "foo"  
        ]  
      ],  
      "terraform_version": "0.11.7",  
      "data": {  
        "null_data_source": {  
          "baz": {  
            "0": {  
              "depends_on": [  
                "null_resource.bar",  
                "null_resource.foo"  
              ],  
              "id": "static",  
              "attr": {  
                "has_computed_default": "default",  
                "id": "static",  
                "inputs": {  
                  "bar_id": "5511801804920420639",  
                  "foo_id": "3332481276939020374"  
                },  
                "outputs": {  
                  "bar_id": "5511801804920420639",  
                  "foo_id": "3332481276939020374"  
                },  
                "random": "1637775805155379683"  
              },  
              "tainted": false  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

```
        }
    },
},
"path": [
    "root"
],
"outputs": {
    "foo": {
        "sensitive": true,
        "type": "string",
        "value": "bar"
    },
    "list": {
        "sensitive": false,
        "type": "list",
        "value": [
            "foo",
            "bar"
        ]
    },
    "map": {
        "sensitive": false,
        "type": "map",
        "value": {
            "foo": "bar",
            "number": 42
        }
    },
    "string": {
        "sensitive": false,
        "type": "string",
        "value": "foo"
    }
},
"resources": {
    "null_resource": {
        "bar": {
            "0": {
                "depends_on": [
                    "null_resource.foo"
                ],
                "id": "5511801804920420639",
                "attr": {
                    "id": "5511801804920420639",
                    "triggers": {
                        "foo_id": "3332481276939020374"
                    }
                },
                "tainted": false
            }
        },
        "foo": {
            "0": {
                "depends_on": null,
                "id": "3332481276939020374",
                "attr": {
                    "id": "3332481276939020374",
                    "triggers": {
                        "foo": "bar"
                    }
                },
                "tainted": false
            }
        }
    }
}
```

}
 }
 }
 }

Import: tfconfig

The `tfconfig` import provides access to a Terraform configuration.

The Terraform configuration is the set of `*.tf` files that are used to describe the desired infrastructure state. This import alone doesn't give you access to the state of the infrastructure. To view the state of the infrastructure, see the `tfstate` (</docs/enterprise/sentinel/import/tfstate.html>) import.

Policies using the `tfconfig` import can access all aspects of the configuration: providers, resources, data sources, modules, and variables. Note that since this is the configuration and not an invocation of Terraform, you can't see values for variables, the state, or the diff for a pending plan.

The Namespace

The following is a tree view of the import namespace. For more detail on a particular part of the namespace, see below.

Note that the root-level alias keys shown here (`data`, `modules`, `providers`, `resources`, and `variables`) are shortcuts to a module namespace scoped to the root module. For more details, see the section on root namespace aliases.

You may also be interested in how to mock `tfconfig` data (</docs/enterprise/sentinel/import/mock-tfconfig.html>), which can help with further namespace visualization.

```

tfconfig
└── module() (function)
  └── (module namespace)
    ├── data
    │   └── TYPE.NAME
    │       ├── config (map of keys)
    │       └── provisioners
    │           └── NUMBER
    │               ├── config (map of keys)
    │               └── type (string)
    ├── modules
    │   └── NAME
    │       ├── config (map of keys)
    │       ├── source (string)
    │       └── version (string)
    ├── providers
    │   └── TYPE
    │       ├── alias
    │       │   └── ALIAS
    │       │       ├── config (map of keys)
    │       │       └── version (string)
    │       ├── config (map of keys)
    │       └── version (string)
    ├── resources
    │   └── TYPE.NAME
    │       ├── config (map of keys)
    │       └── provisioners
    │           └── NUMBER
    │               ├── config (map of keys)
    │               └── type (string)
    └── variables
        └── NAME
            ├── default (value)
            └── description (string)
└── module_paths ([][]string)
|
└── data (root module alias)
└── modules (root module alias)
└── providers (root module alias)
└── resources (root module alias)
└── variables (root module alias)

```

Namespace: Root

The root-level namespace consists of the values and functions documented below.

In addition to this, the root-level `data`, `modules`, `providers`, `resources`, and `variables` keys all alias to their corresponding namespaces within the module namespace.

Function: `module()`

```
module = func(ADDR)
```

- **Return Type:** A module namespace.

The `module()` function in the root namespace returns the module namespace for a particular module address.

The address must be a list and is the module address, split on the period (.), excluding the root module.

Hence, a module with an address of simply `foo` (or `root.foo`) would be `["foo"]`, and a module within that (so address `foo.bar`) would be read as `["foo", "bar"]`.

`null` (<https://docs.hashicorp.com/sentinel/language/spec#null>) is returned if a module address is invalid, or if the module is not present in the configuration.

As an example, given the following module block:

```
module "foo" {  
    # ...  
}
```

If the module contained the following content:

```
resource "null_resource" "foo" {  
    triggers = {  
        foo = "bar"  
    }  
}
```

The following policy would evaluate to true:

```
import "tfconfig"  
  
main = rule { subject.module(["foo"]).resources.null_resource.foo.config.triggers[0].foo is "bar" }
```

Value: module_paths

- **Value Type:** List of a list of strings.

The `module_paths` value within the root namespace is a list of all of the modules within the Terraform configuration.

Modules not present in the configuration will not be present here, even if they are present in the diff or state.

This data is represented as a list of a list of strings, with the inner list being the module address, split on the period (.).

The root module is included in this list, represented as an empty inner list.

As an example, if the following module block was present within a Terraform configuration:

```
module "foo" {  
    # ...  
}
```

The value of `module_paths` would be:

```
[  
  [],  
  ["foo"],  
]
```

And the following policy would evaluate to true:

```
import "tfconfig"  
  
main = rule { tfconfig.module_paths contains ["foo"] }
```

Iterating through modules

Iterating through all modules to find particular resources can be useful. This example shows how to use `module_paths` with the `module()` function to retrieve all resources of a particular type from all modules (in this case, the `azurerm_virtual_machine` ([/docs/providers/azurerm/r/virtual_machine.html](#)) resource). Note the use of `else []` in case some modules don't have any resources; this is necessary to avoid the function returning undefined.

```
import "tfconfig"  
  
get_vms = func() {  
  vms = []  
  for tfconfig.module_paths as path {  
    vms += values(tfconfig.module(path).resources.azurerm_virtual_machine) else []  
  }  
  return vms  
}
```

Namespace: Module

The **module namespace** can be loaded by calling `module()` for a particular module.

It can be used to load the following child namespaces:

- `data` - Loads the resource namespace, filtered against data sources.
- `modules` - Loads the module configuration namespace.
- `providers` - Loads the provider namespace.
- `resources` - Loads the resource namespace, filtered against resources.
- `variables` - Loads the variable namespace.

Root Namespace Aliases

The root-level `data`, `modules`, `providers`, `resources`, and `variables` keys all alias to their corresponding namespaces within the module namespace, loaded for the root module. They are the equivalent of running `module([]).KEY`.

Namespace: Resources/Data Sources

The **resource namespace** is a namespace type that applies to both resources (accessed by using the `resources` namespace key) and data sources (accessed using the `data` namespace key).

Accessing an individual resource or data source within each respective namespace can be accomplished by specifying the type and name, in the syntax `[resources|data].TYPE.NAME`.

In addition, each of these namespace levels is a map, allowing you to filter based on type and name. Some examples of multi-level access are below:

- To fetch all `aws_instance` resources within the root module, you can specify `tfconfig.resources.aws_instance`. This would give you a map of resource namespaces indexed off of the names of each resource (foo, bar, and so on).
- To fetch all resources within the root module, irrespective of type, use `tfconfig.resources`. This is indexed by type, as shown above with `tfconfig.resources.aws_instance`, with names being the next level down.

Further explanation of the namespace will be in the context of resources. As mentioned, when operating on data sources, use the same syntax, except with `data` in place of `resources`.

Value: config

- **Value Type:** A string-keyed map of values.

The `config` value within the resource namespace is a map of key-value pairs that directly map to Terraform config keys and values.

As a consequence of the mapping of this key to raw Terraform configuration, complex structures within Terraform configuration are grouped *per-instance* as they are represented within the actual configuration itself. This has implications when defining policy correctly. This applies to all complex structures - lists, sets, and maps.

As an example, consider the following resource block:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "one"
  }

  triggers = {
    bar = "two"
  }
}
```

In this example, one would need to access both `tfconfig.resources.null_resource.foo.config.triggers[0].foo` and `tfconfig.resources.null_resource.foo.config.triggers[1].bar` to reach both triggers.

This can be better represented by a loop. Given the above example, the following policy would evaluate to `true`:

```
import "tfconfig"

main = rule {
    all tfconfig.resources.null_resource.foo.config.triggers as triggers {
        all triggers as _, value {
            value in ["one", "two"]
        }
    }
}
```

Value: provisioners

- **Value Type:** List of provisioner namespaces.

The `provisioners` value within the resource namespace represents the provisioners ([/docs/provisioners/index.html](#)) within a specific resource.

Provisioners are listed in the order they were provided in the configuration file.

The data within a provisioner can be inspected via the returned provisioner namespace.

Namespace: Provisioners

The **provisioner namespace** represents the configuration for a particular provisioner ([/docs/provisioners/index.html](#)) within a specific resource or data source.

Value: config

- **Value Type:** A string-keyed map of values.

The `config` value within the provisioner namespace represents the values of the keys within the provisioner.

As an example, given the following resource block:

```
resource "null_resource" "foo" {
    # ...

    provisioner "local-exec" {
        command = "echo ${self.private_ip} > file.txt"
    }
}
```

The following policy would evaluate to true:

```
import "tfconfig"

main = rule {
    tfconfig.resources.null_resource.foo.provisioners[0].config.command is "echo ${self.private_ip} > file.txt"
}
```

Value: type

- **Value Type:** String.

The type value within the provisioner namespace represents the type of the specific provisioner.

As an example, in the following resource block:

```
resource "null_resource" "foo" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} > file.txt"
  }
}
```

The following policy would evaluate to true:

```
import "tfconfig"

main = rule { tfconfig.resources.null_resource.foo.provisioners[0].type is "local-exec" }
```

Namespace: Module Configuration

The **module configuration** namespace displays data on *module configuration* as it is given within a module block. This means that the namespace concerns itself with the contents of the declaration block (example: the source parameter and variable assignment keys), not the data within the module (example: any contained resources or data sources). For the latter, the module instance would need to be looked up with the `module()` function.

Value: source

- **Value Type:** String.

The source value within the module configuration namespace represents the module source path as supplied to the module configuration.

As an example, given the module declaration block:

```
module "foo" {
  source = "./foo"
}
```

The following policy would evaluate to true:

```
import "tfconfig"

main = rule { tfconfig.modules.foo.source is "./foo" }
```

Value: version

- **Value Type:** String.

The `version` value within the module configuration namespace represents the version constraint ([/docs/modules/usage.html#module-versions](#)) for modules that support it, such as modules within the Terraform Module Registry (<https://registry.terraform.io/>) or the Terraform Enterprise private module registry ([/docs/enterprise/registry/index.html](#)).

As an example, given the module declaration block:

```
module "foo" {  
  source  = "foo/bar"  
  version = "~> 1.2"  
}
```

The following policy would evaluate to true:

```
import "tfconfig"  
  
main = rule { tfconfig.modules.foo.version is "~> 1.2" }
```

Value: config

- **Value Type:** A string-keyed map of values.

The `config` value within the module configuration namespace represents the values of the keys within the module configuration. This is every key within a module declaration block except `source` and `version`, which have their own values.

As an example, given the module declaration block:

```
module "foo" {  
  source = "./foo"  
  
  bar = "baz"  
}
```

The following policy would evaluate to true:

```
import "tfconfig"  
  
main = rule { tfconfig.modules.foo.config.bar is "baz" }
```

Namespace: Providers

The **provider namespace** represents data on the declared providers within a namespace.

This namespace is indexed by provider type and *only* contains data about providers when actually declared. If you are using a completely implicit provider configuration, this namespace will be empty.

This namespace is populated based on the following criteria:

- The top-level namespace `config` and `version` values are populated with the configuration and version information from the default provider (the provider declaration that lacks an alias).
- Any aliased providers are added as namespaces within the `alias` value.
- If a module lacks a default provider configuration, the top-level `config` and `version` values will be empty.

Value: alias

- **Value Type:** A map of provider namespaces, indexed by alias.

The `alias` value within the provider namespace represents all declared non-default provider instances ([/docs/configuration/providers.html#multiple-provider-instances](#)) for a specific provider type, indexed by their specific alias.

The return type is a provider namespace with the data for the instance in question loaded. The `alias` key will not be available within this namespace.

As an example, given the following provider declaration block:

```
provider "aws" {
  alias  = "east"
  region = "us-east-1"
}
```

The following policy would evaluate to true:

```
import "tfconfig"

main = rule { tfconfig.providers.aws.alias.east.config.region is "us-east-1" }
```

Value: config

- **Value Type:** A string-keyed map of values.

The `config` value within the provider namespace represents the values of the keys within the provider's configuration, with the exception of the provider version, which is represented by the `version` value. `alias` is also not included when the provider is aliased.

As an example, given the following provider declaration block:

```
provider "aws" {
  region = "us-east-1"
}
```

The following policy would evaluate to true:

```
import "tfconfig"

main = rule { tfconfig.providers.aws.config.region is "us-east-1" }
```

Value: version

- **Value Type:** String.

The `version` value within the provider namespace represents the explicit expected version of the supplied provider. This includes the pessimistic operator.

As an example, given the following provider declaration block:

```
provider "aws" {
  version = "~> 1.34"
}
```

The following policy would evaluate to true:

```
import "tfconfig"

main = rule { tfconfig.providers.aws.version is "~> 1.34" }
```

Namespace: Variables

The **variable namespace** represents *declared* variable data within a configuration. As such, static data can be extracted, such as defaults, but not dynamic data, such as the current value of a variable within a plan (although this can be extracted within the `tfplan` import ([/docs/enterprise/sentinel/import/tfplan.html](#))).

This namespace is indexed by variable name.

Value: default

- **Value Type:** Any primitive type, list, map, or null.

The `default` value within the variable namespace represents the default for the variable as declared in the configuration.

The actual value will be as configured. Primitives will bear the implicit type of its declaration (string, int, float, or bool), and maps and lists will be represented as such.

If no default is present, the value will be `null` ([https://docs.hashicorp.com/sentinel/language/spec#null](#)) (not to be confused with `undefined` ([https://docs.hashicorp.com/sentinel/language/undefined](#))).

As an example, given the following variable blocks:

```
variable "foo" {
  default = "bar"
}

variable "number" {
  default = 42
}
```

The following policy would evaluate to true:

```
import "tfconfig"

default_foo = rule { tfconfig.variables.foo.default is "bar" }
default_number = rule { tfconfig.variables.number.default is 42 }

main = rule { default_foo and default_number }
```

Value: description

- **Value Type:** String.

The description value within the variable namespace represents the description of the variable, as provided in configuration.

As an example, given the following variable block:

```
variable "foo" {
  description = "foobar"
}
```

The following policy would evaluate to true:

```
import "tfconfig"

main = rule { tfconfig.variables.foo.description is "foobar" }
```

Import: tfplan

The `tfplan` import provides access to a Terraform plan. A Terraform plan is the file created as a result of `terraform plan` and is the input to `terraform apply`. The plan represents the changes that Terraform needs to make to infrastructure to reach the desired state represented by the configuration.

In addition to the diff data available in the plan, there is an applied state available that merges the plan with the state to create the planned state after apply.

Finally, this import also allows you to access the configuration files and the Terraform state at the time the plan was run. See the section on accessing a plan's state and configuration data for more information.

The Namespace

The following is a tree view of the import namespace. For more detail on a particular part of the namespace, see below.

Note that the root-level alias keys shown here (`data`, `path`, and `resources`) are shortcuts to a module namespace scoped to the root module. For more details, see the section on root namespace aliases.

You may also be interested in how to mock `tfplan` data (/docs/enterprise/sentinel/import/mock-tfplan.html), which can help with further namespace visualization.

```
tfplan
└── module() (function)
    └── (module namespace)
        ├── path ([]string)
        ├── data
        │   └── TYPE.NAME[NUMBER]
        │       ├── applied (map of keys)
        │       └── diff
        │           └── KEY
        │               ├── computed (bool)
        │               ├── new (string)
        │               └── old (string)
        └── resources
            └── TYPE.NAME[NUMBER]
                ├── applied (map of keys)
                └── diff
                    └── KEY
                        ├── computed (bool)
                        ├── new (string)
                        └── old (string)
    └── module_paths ([][]string)
    └── terraform_version (string)
    └── variables (map of keys)
    └──
    └── data (root module alias)
    └── path (root module alias)
    └── resources (root module alias)
    └──
    └── config (tfconfig namespace alias)
    └── state (tfstate import alias)
```

Namespace: Root

The root-level namespace consists of the values and functions documented below.

In addition to this, the root-level `data`, `path`, and `resources` keys alias to their corresponding namespaces or values within the module namespace.

Accessing a Plan's State and Configuration Data

The `config` and `state` keys alias to the `tfconfig` (/docs/enterprise/sentinel/import/tfconfig.html) and `tfstate` (/docs/enterprise/sentinel/import/tfstate.html) namespaces, respectively, with the data sourced from the Terraform *plan* (as opposed to actual configuration and state).

Note that these aliases are not represented as maps. While they will appear empty when viewed as maps, the specific import namespace keys will still be accessible.

Note that while current versions of Terraform Enterprise (TFE) source *all* data (including configuration and state) from the plan for the Terraform run in question, future versions of TFE may source data accessed through the `tfconfig` and `tfstate` imports (as opposed to `tfplan.config` and `tfplan.state`) from actual config bundles, or state as stored by TFE. When this happens, the distinction here will be useful - the data in the aliased namespaces will be the config and state data as the *plan* sees it, versus the actual "physical" data.

Function: module()

```
module = func(ADDR)
```

- **Return Type:** A module namespace.

The `module()` function in the root namespace returns the module namespace for a particular module address.

The address must be a list and is the module address, split on the period (`.`), excluding the root module.

Hence, a module with an address of simply `foo` (or `root.foo`) would be `["foo"]`, and a module within that (so address `foo.bar`) would be read as `["foo", "bar"]`.

`null` (<https://docs.hashicorp.com/sentinel/language/spec#null>) is returned if a module address is invalid, or if the module is not present in the diff.

As an example, given the following module block:

```
module "foo" {  
    # ...  
}
```

If the module contained the following content:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "bar"
  }
}
```

The following policy would evaluate to true:

```
import "tfplan"

main = rule { tfplan.module(["foo"]).resources.null_resource.foo[0].applied.triggers.foo is "bar" }
```

Value: module_paths

- **Value Type:** List of a list of strings.

The `module_paths` value within the root namespace is a list of all of the modules within the Terraform diff for the current plan.

Modules not present in the diff will not be present here, even if they are present in the configuration or state.

This data is represented as a list of a list of strings, with the inner list being the module address, split on the period (.).

The root module is included in this list, represented as an empty inner list, as long as there are changes.

As an example, if the following module block was present within a Terraform configuration:

```
module "foo" {
  # ...
}
```

The value of `module_paths` would be:

```
[  
  [],  
  ["foo"],  
]
```

And the following policy would evaluate to true:

```
import "tfplan"

main = rule { tfplan.module_paths contains ["foo"] }
```

Note the above example only applies if the module is present in the diff.

Iterating through modules

Iterating through all modules to find particular resources can be useful. This example shows how to use `module_paths` with the `module()` function to retrieve all resources of a particular type from all modules (in this case, the `azurerm_virtual_machine` (/docs/providers/azurerm/r/virtual_machine.html) resource). Note the use of `else []` in case some modules don't have any resources; this is necessary to avoid the function returning undefined.

Remember again that this will only locate modules (and hence resources) that have pending changes.

```
import "tfplan"

get_vms = func() {
    vms = []
    for tfplan.module_paths as path {
        vms += values(tfplan.module(path).resources.azurerm_virtual_machine) else []
    }
    return vms
}
```

Value: `terraform_version`

- **Value Type:** String.

The `terraform_version` value within the root namespace represents the version of Terraform used to create the plan. This can be used to enforce a specific version of Terraform in a policy check.

As an example, the following policy would evaluate to `true`, as long as the plan was made with a version of Terraform in the 0.11.x series, excluding any pre-release versions (example: `-beta1` or `-rc1`):

```
import "tfplan"

main = rule { tfplan.terraform_version matches "^0\\.11\\.\\d+$" }
```

Value: `variables`

- **Value Type:** A string-keyed map of values.

The `variables` value within the root namespace represents all of the variables that were set when creating the plan. This will only contain variables set for the root module.

Note that unlike the `default` (/docs/enterprise/sentinel/import/tfconfig.html#value-default) value in the `tfconfig` variables namespace (/docs/enterprise/sentinel/import/tfconfig.html#namespace-variables), primitive values here are stringified, and type conversion will need to be performed to perform comparison for int, float, or boolean values. This only applies to variables that are primitives themselves and not primitives within maps and lists, which will be their original types.

If a default was accepted for the particular variable, the default value will be populated here.

As an example, given the following variable blocks:

```

variable "foo" {
  default = "bar"
}

variable "number" {
  default = 42
}

variable "map" {
  default = {
    foo    = "bar"
    number = 42
  }
}

```

The following policy would evaluate to true, if no values were entered to change these variables:

```

import "tfplan"

default_foo = rule { tfplan.variables.foo is "bar" }
default_number = rule { tfplan.variables.number is "42" }
default_map_string = rule { tfplan.variables.map["foo"] is "bar" }
default_map_int = rule { tfplan.variables.map["number"] is 42 }

main = rule { default_foo and default_number and default_map_string and default_map_int }

```

Namespace: Module

The **module namespace** can be loaded by calling `module()` for a particular module.

It can be used to load the following child namespaces, in addition to the values documented below:

- `data` - Loads the resource namespace, filtered against data sources.
- `resources` - Loads the resource namespace, filtered against resources.

Root Namespace Aliases

The root-level `data` and `resources` keys both alias to their corresponding namespaces within the module namespace, loaded for the root module. They are the equivalent of running `module([]).KEY`.

Value: path

- **Value Type:** List of strings.

The path value within the module namespace contains the path of the module that the namespace represents. This is represented as a list of strings.

As an example, if the following module block was present within a Terraform configuration:

```
module "foo" {  
    # ...  
}
```

The following policy would evaluate to true *only* if the diff had changes for that module:

```
import "tfplan"  
  
main = rule { tfplan.module(["foo"]).path contains "foo" }
```

Namespace: Resources/Data Sources

The **resource namespace** is a namespace *type* that applies to both resources (accessed by using the `resources` namespace key) and data sources (accessed using the `data` namespace key).

Accessing an individual resource or data source within each respective namespace can be accomplished by specifying the type, name, and resource number (as if the resource or data source had a `count` value in it) in the syntax `[resources|data].TYPE.NAME[NUMBER]`. Note that `NUMBER` is always needed, even if you did not use `count` in the resource.

In addition, each of these namespace levels is a map, allowing you to filter based on type and name.

The (somewhat strange) notation here of `TYPE.NAME[NUMBER]` may imply that the inner resource index map is actually a list, but it's not - using the square bracket notation over the dotted notation (`TYPE.NAME.NUMBER`) is required here as an identifier cannot start with a number.

Some examples of multi-level access are below:

- To fetch all `aws_instance.foo` resource instances within the root module, you can specify `tfplan.resources.aws_instance.foo`. This would then be indexed by resource count index (0, 1, 2, and so on). Note that as mentioned above, these elements must be accessed using square-bracket map notation (so `[0]`, `[1]`, `[2]`, and so on) instead of dotted notation.
- To fetch all `aws_instance` resources within the root module, you can specify `tfplan.resources.aws_instance`. This would be indexed off of the names of each resource (foo, bar, and so on), with each of those maps containing instances indexed by resource count index as per above.
- To fetch all resources within the root module, irrespective of type, use `tfplan.resources`. This is indexed by type, as shown above with `tfplan.resources.aws_instance`, with names being the next level down, and so on.

Further explanation of the namespace will be in the context of resources. As mentioned, when operating on data sources, use the same syntax, except with `data` in place of `resources`.

Value: applied

- **Value Type:** A string-keyed map of values.

The applied value within the resource namespace contains a "predicted" representation of the resource's state post-apply. It's created by merging the pending resource's diff on top of the existing data from the resource's state (if any).

The map is a complex representation of these values with data going as far down as needed to represent any state values such as maps, lists, and sets.

Note that some values will not be available in the applied state because they cannot be known until the plan is actually applied. These values are represented by a placeholder (the UUID value 74D93920-ED26-11E3-AC10-0800200C9A66). This is not a stable API and should not be relied on. Instead, use the computed key within the diff namespace to determine if a value is known or not.

As an example, given the following resource:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "bar"
  }
}
```

The following policy would evaluate to true if the resource was in the diff:

```
import "tfplan"

main = rule { tfplan.resources.null_resource.foo[0].applied.triggers.foo is "bar" }
```

Value: diff

- **Value Type:** A map of diff namespaces.

The diff value within the resource namespace contains the diff for a particular resource. Each key within the map links to a diff namespace for that particular key.

Note that unlike the applied value, this map is not complex; the map is only 1 level deep with each key possibly representing a diff for a particular complex value within the resource.

See the below section for more details on the diff namespace, in addition to usage examples.

Namespace: Resource Diff

The **diff namespace** is a namespace that represents the diff for a specific attribute within a resource. For details on reading a particular attribute, see the diff value in the resource namespace.

Value: computed

- **Value Type:** Boolean.

The computed value within the diff namespace is true if the resource key in question depends on another value that isn't yet known. Typically, that means the value it depends on belongs to a resource that either doesn't exist yet, or is changing state in such a way as to affect the dependent value so that it can't be known until the apply is complete.

Keep in mind that when using `computed` with complex structures such as maps, lists, and sets, it's sometimes necessary to test the `count` attribute for the structure, versus a key within it, depending on whether or not the diff has marked the whole structure as `computed`. This is demonstrated in the example below. `Count` keys are `%` for maps, and `#` for lists and sets. If you are having trouble determining the type of specific field within a resource, contact the support team.

As an example, given the following resource:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "bar"
  }
}

resource "null_resource" "bar" {
  triggers = {
    foo_id = "${null_resource.foo.id}"
  }
}
```

The following policy would evaluate to `true`, if the `id` of `null_resource.foo` was currently not known, such as when the resource is pending creation, or is being deleted and re-created:

```
import "tfplan"

main = rule { tfplan.resources.null_resource.bar[0].diff["triggers.%"].computed }
```

Value: new

- **Value Type:** String.

The new value within the diff namespace contains the new value of a changing attribute, if the value is known at plan time.

new will be an empty string if the attribute's value is currently unknown. For more details on detecting unknown values, see `computed`.

Note that this value is *always* a string, regardless of the actual type of the value changing. Type conversion (<https://docs.hashicorp.com/sentinel/language/values#type-conversion>) within policy may be necessary to achieve the comparison needed.

As an example, given the following resource:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "bar"
  }
}
```

The following policy would evaluate to `true`, if the resource was in the diff and each of the concerned keys were changing to new values:

```
import "tfplan"

main = rule { tfplan.resources.null_resource.foo[0].diff["triggers.foo"].new is "bar" }
```

Value: old

- **Value Type:** String.

The `old` value within the `diff` namespace contains the old value of a changing attribute.

Note that this value is *always* a string, regardless of the actual type of the value changing. Type conversion (<https://docs.hashicorp.com/sentinel/language/values#type-conversion>) within policy may be necessary to achieve the comparison needed.

If the value did not exist in the previous state, `old` will always be an empty string.

As an example, given the following resource:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "baz"
  }
}
```

If that resource was previously in config as:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "bar"
  }
}
```

The following policy would evaluate to true:

```
import "tfplan"

main = rule { tfplan.resources.null_resource.foo[0].diff["triggers.foo"].old is "bar" }
```

Import: tfstate

The `tfstate` import provides access to the Terraform state.

The `state` is the data that Terraform has recorded about a workspace at a particular point in its lifecycle, usually after an `apply`. You can read more general information about how Terraform uses state here ([/docs/state/index.html](#)).

`tfstate` is also the only import that can provide introspection into the outputs in a Terraform workspace. For more information, see the output namespace documentation.

NOTE: Terraform Enterprise currently only supports policy checks at plan time, limiting the usefulness of this import. This will be resolved in future releases. Until that time, the `tfconfig` ([/docs/enterprise/sentinel/import/tfconfig.html](#)) and `tfplan` ([/docs/enterprise/sentinel/import/tfplan.html](#)) imports will probably prove to be more useful.

The Namespace

The following is a tree view of the import namespace. For more detail on a particular part of the namespace, see below.

Note that the root-level alias keys shown here (`data`, `outputs`, `path`, and `resources`) are shortcuts to a module namespace scoped to the root module. For more details, see the section on root namespace aliases.

You may also be interested in how to mock `tfstate` data ([/docs/enterprise/sentinel/import/mock-tfstate.html](#)), which can help with further namespace visualization.

```

tfstate
└── module() (function)
  └── (module namespace)
    ├── path ([]string)
    ├── data
    |   └── TYPE.NAME[NUMBER]
    |       ├── attr (map of keys)
    |       ├── depends_on ([]string)
    |       ├── id (string)
    |       └── tainted (boolean)
    ├── outputs
    |   └── NAME
    |       ├── sensitive (bool)
    |       ├── type (string)
    |       └── value (value)
    └── resources
        └── TYPE.NAME[NUMBER]
            ├── attr (map of keys)
            ├── depends_on ([]string)
            ├── id (string)
            └── tainted (boolean)

  └── module_paths ([][]string)
  └── terraform_version (string)
  └── data (root module alias)
  └── outputs (root module alias)
  └── path (root module alias)
  └── resources (root module alias)

```

Namespace: Root

The root-level namespace consists of the values and functions documented below.

In addition to this, the root-level `data`, `outputs`, `path`, and `resources` keys alias to their corresponding namespaces or values within the module namespace.

Function: `module()`

```
module = func(ADDR)
```

- **Return Type:** A module namespace.

The `module()` function in the root namespace returns the module namespace for a particular module address.

The address must be a list and is the module address, split on the period (.), excluding the root module.

Hence, a module with an address of simply `foo` (or `root.foo`) would be `["foo"]`, and a module within that (so address `foo.bar`) would be read as `["foo", "bar"]`.

`null` (<https://docs.hashicorp.com/sentinel/language/spec#null>) is returned if a module address is invalid, or if the module is not present in the state.

As an example, given the following module block:

```
module "foo" {  
    # ...  
}
```

If the module contained the following content:

```
resource "null_resource" "foo" {  
    triggers = {  
        foo = "bar"  
    }  
}
```

The following policy would evaluate to `true` if the resource was present in the state:

```
import "tfstate"  
  
main = rule { tfstate.module(["foo"]).resources.null_resource.foo[0].attr.triggers.foo is "bar" }
```

Value: module_paths

- **Value Type:** List of a list of strings.

The `module_paths` value within the root namespace is a list of all of the modules within the Terraform state at plan-time.

Modules not present in the state will not be present here, even if they are present in the configuration or the diff.

This data is represented as a list of a list of strings, with the inner list being the module address, split on the period (`.`).

The root module is included in this list, represented as an empty inner list, as long as it is present in state.

As an example, if the following module block was present within a Terraform configuration:

```
module "foo" {  
    # ...  
}
```

The value of `module_paths` would be:

```
[  
    [],  
    ["foo"],  
]
```

And the following policy would evaluate to `true`:

```
import "tfstate"  
  
main = rule { tfstate.module_paths contains ["foo"] }
```

Note the above example only applies if the module is present in the state.

Iterating through modules

Iterating through all modules to find particular resources can be useful. This example shows how to use `module_paths` with the `module()` function to retrieve all resources of a particular type from all modules (in this case, the `azurerm_virtual_machine` (/docs/providers/azurerm/r/virtual_machine.html) resource). Note the use of `else []` in case some modules don't have any resources; this is necessary to avoid the function returning undefined.

Remember again that this will only locate modules (and hence resources) that are present in state.

```
import "tfstate"

get_vms = func() {
    vms = []
    for tfstate.module_paths as path {
        vms += values(tfstate.module(path).resources.azurerm_virtual_machine) else []
    }
    return vms
}
```

Value: `terraform_version`

- **Value Type:** String.

The `terraform_version` value within the root namespace represents the version of Terraform in use when the state was saved. This can be used to enforce a specific version of Terraform in a policy check.

As an example, the following policy would evaluate to `true` as long as the state was made with a version of Terraform in the 0.11.x series, excluding any pre-release versions (example: `-beta1` or `-rc1`):

```
import "tfstate"

main = rule { tfstate.terraform_version matches "^0\\.11\\.\\d+$" }
```

NOTE: This value is also available via the `tfplan` (/docs/enterprise/sentinel/import/tfplan.html) import, which will be more current when a policy check is run against a plan. It's recommended you use the value in `tfplan` until Terraform enterprise supports policy checks in other stages of the workspace lifecycle. See the `terraform_version` (/docs/enterprise/sentinel/import/tfplan.html#value-terraform-version) reference within the `tfplan` import for more details.

Namespace: Module

The **module namespace** can be loaded by calling `module()` for a particular module.

It can be used to load the following child namespaces, in addition to the values documented below:

- `data` - Loads the resource namespace, filtered against data sources.
- `outputs` - Loads the output namespace, which supply the outputs present in this module's state.
- `resources` - Loads the resource namespace, filtered against resources.

Root Namespace Aliases

The root-level `data`, `outputs`, and `resources` keys both alias to their corresponding namespaces within the module namespace, loaded for the root module. They are the equivalent of running `module([]).KEY`.

Value: path

- **Value Type:** List of strings.

The path value within the module namespace contains the path of the module that the namespace represents. This is represented as a list of strings.

As an example, if the following module block was present within a Terraform configuration:

```
module "foo" {
  # ...
}
```

The following policy would evaluate to `true`, *only* if the module was present in the state:

```
import "tfstate"

main = rule { tfstate.module(["foo"]).path contains "foo" }
```

Namespace: Resources/Data Sources

The **resource namespace** is a namespace *type* that applies to both resources (accessed by using the `resources` namespace key) and data sources (accessed using the `data` namespace key).

Accessing an individual resource or data source within each respective namespace can be accomplished by specifying the type, name, and resource number (as if the resource or data source had a `count` value in it) in the syntax `[resources|data].TYPE.NAME[NUMBER]`. Note that `NUMBER` is always needed, even if you did not use `count` in the resource.

In addition, each of these namespace levels is a map, allowing you to filter based on type and name.

The (somewhat strange) notation here of `TYPE.NAME[NUMBER]` may imply that the inner resource index map is actually a list, but it's not - using the square bracket notation over the dotted notation (`TYPE.NAME.NUMBER`) is required here as an identifier cannot start with number.

Some examples of multi-level access are below:

- To fetch all `aws_instance.foo` resource instances within the root module, you can specify `tfstate.resources.aws_instance.foo`. This would then be indexed by resource count index (0, 1, 2, and so on). Note that as mentioned above, these elements must be accessed using square-bracket map notation (so `[0]`, `[1]`, `[2]`, and so on) instead of dotted notation.
- To fetch all `aws_instance` resources within the root module, you can specify `tfstate.resources.aws_instance`. This would be indexed off of the names of each resource (`foo`, `bar`, and so on), with each of those maps containing instances indexed by resource count index as per above.
- To fetch all resources within the root module, irrespective of type, use `tfstate.resources`. This is indexed by type, as shown above with `tfstate.resources.aws_instance`, with names being the next level down, and so on.

Further explanation of the namespace will be in the context of resources. As mentioned, when operating on data sources, use the same syntax, except with `data` in place of `resources`.

Value: attr

- **Value Type:** A string-keyed map of values.

The `attr` value within the resource namespace is a direct mapping to the state of the resource.

The map is a complex representation of these values with data going as far down as needed to represent any state values such as maps, lists, and sets.

As an example, given the following resource:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "bar"
  }
}
```

The following policy would evaluate to `true` if the resource was in the state:

```
import "tfstate"

main = rule { tfstate.resources.null_resource.foo[0].attr.triggers.foo is "bar" }
```

Value: depends_on

- **Value Type:** A list of strings.

The `depends_on` value within the resource namespace contains the dependencies for the resource.

This is a list of full resource addresses, relative to the module (example: `null_resource.foo`).

As an example, given the following resources:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "bar"
  }
}

resource "null_resource" "bar" {
  # ...
  depends_on = [
    "null_resource.foo",
  ]
}
```

The following policy would evaluate to true if the resource was in the state:

```
import "tfstate"

main = rule { tfstate.resources.null_resource.bar[0].depends_on contains "null_resource.foo" }
```

Value: id

- **Value Type:** String.

The id value within the resource namespace contains the id of the resource.

NOTE: The example below uses a *data source* here because the `null_data_source` (/docs/providers/null/data_source.html) data source gives a static ID, which makes documenting the example easier. As previously mentioned, data sources share the same namespace as resources, but need to be loaded with the `data` key. For more information, see the synopsis for the namespace itself.

As an example, given the following data source:

```
data "null_data_source" "foo" {
  # ...
}
```

The following policy would evaluate to true:

```
import "tfstate"

main = rule { tfstate.data.null_data_source.foo[0].id is "static" }
```

Value: tainted

- **Value Type:** Boolean.

The tainted value within the resource namespace is true if the resource is marked as tainted in Terraform state.

As an example, given the following resource:

```
resource "null_resource" "foo" {
  triggers = {
    foo = "bar"
  }
}
```

The following policy would evaluate to true, if the resource was marked as tainted in the state:

```
import "tfstate"

main = rule { tfstate.resources.null_resource.foo[0].tainted }
```

Namespace: Outputs

The **output namespace** represents all of the outputs present within a module. Outputs are present in a state if they were saved during a previous apply, or if they were updated with known values during the pre-plan refresh.

Note that this can be used to fetch both the outputs of the root module, and the outputs of any module in the state below the root. This makes it possible to see outputs that have not been threaded to the root module.

This namespace is indexed by output name.

Value: sensitive

- **Value Type:** Boolean.

The sensitive value within the output namespace is true when the output has been marked as sensitive ([/docs/configuration/outputs.html#sensitive-outputs](#)).

As an example, given the following output:

```
output "foo" {
  sensitive = true
  value     = "bar"
}
```

The following policy would evaluate to true:

```
import "tfstate"

main = rule { tfstate.outputs.foo.sensitive }
```

Value: type

- **Value Type:** String.

The type value within the output namespace gives the output's type. This will be one of `string`, `list`, or `map`. These are currently the only types available for outputs in Terraform.

As an example, given the following output:

```
output "string" {
  value = "foo"
}

output "list" {
  value = [
    "foo",
    "bar",
  ]
}

output "map" {
  value = {
    foo = "bar"
  }
}
```

The following policy would evaluate to `true`:

```
import "tfstate"

type_string = rule { tfstate.outputs.string.type is "string" }
type_list = rule { tfstate.outputs.list.type is "list" }
type_map = rule { tfstate.outputs.map.type is "map" }

main = rule { type_string and type_list and type_map }
```

Value: value

- **Value Type:** String, list, or map.

The `value` value within the output namespace is the value of the output in question.

Note that the only valid primitive output type in Terraform is currently a string, which means that any int, float, or boolean value will need to be converted before it can be used in comparison. This does not apply to primitives within maps and lists, which will be their original types.

As an example, given the following output blocks:

```
output "foo" {
    value = "bar"
}

output "number" {
    value = "42"
}

output "map" {
    value = {
        foo      = "bar"
        number  = 42
    }
}
```

The following policy would evaluate to true:

```
import "tfstate"

value_foo = rule { tfstate.outputs.foo.value is "bar" }
value_number = rule { int(tfstate.outputs.number.value) is 42 }
value_map_string = rule { tfstate.outputs.map.value["foo"] is "bar" }
value_map_int = rule { tfstate.outputs.map.value["number"] is 42 }

main = rule { value_foo and value_number and value_map_string and value_map_int }
```

Managing Sentinel Policies with Version Control

Terraform Enterprise (TFE)'s UI for managing Sentinel policies (</docs/enterprise/sentinel/manage-policies.html>) is designed primarily for *viewing* your organization's policies and policy sets. It also works well for demos and other simple use cases.

For complex day-to-day use, we recommend keeping Sentinel code in version control and using Terraform to automatically deploy your policies. This approach is a better fit with Sentinel's policy-as-code design, and scales better for organizations with multiple owners and administrators.

This page describes an end-to-end process for managing TFE's Sentinel policies with version control and Terraform. Use this outline and the example repository (<https://github.com/hashicorp/tfe-policies-example/>) as a starting point for managing your own organization's policies.

Summary

Managing policies with version control and Terraform requires the following steps:

- Create a VCS repository for policies.
- Write Sentinel policies and add them to the policy repo.
- Write a Terraform configuration for managing policies in TFE, and add it to the policy repo. This configuration must:
 - Configure the `tfe` provider (</docs/providers/tfe/index.html>).
 - Manage individual policies with `tfe_sentinel_policy` resources.
 - Manage policy sets with `tfe_policy_set` resources.
- Create a TFE workspace linked to the policy repo.
- Write tests for your Sentinel policies.
- Use CI to run Sentinel tests automatically.

Repository Structure

Example: We've provided a complete and working example policy repo (<https://github.com/hashicorp/tfe-policies-example/>), which you can use as a template for your own policy repo.

Create a single VCS repository for managing your organization's Sentinel policies. We recommend a short and descriptive name like "tfe-policies". Later, you will create a TFE workspace based on this repo.

Once the policy management process is fully implemented, the repo will contain the following:

- **Sentinel policies** stored as `.sentinel` files in the root of the repo.
- **A Terraform configuration** stored in the root of the repo. This can be a single `main.tf` file, or several smaller configuration files.
- **Sentinel tests** stored in a `test/` directory.

- Optionally: other information or metadata, which might include a README file, editor configurations, and CI configurations.

Important: When managing policies with Terraform, the security of your policy repo determines the security of your policies. You should strictly control merge access to this repo, prohibit direct pushes to master, and enforce a consistent process for reviewing and merging changes to policy code.

Sentinel Policies

Example: See the `.sentinel` files included in the example policy repo (<https://github.com/hashicorp/tfe-policies-example/>), or browse the list of example policies (</docs/enterprise/sentinel/examples.html>).

Write some Sentinel policies for TFE (</docs/enterprise/sentinel/import/index.html>), and commit them to your repo as `.sentinel` files. If you're already enforcing Sentinel policies in TFE, copy them into the new repo.

Note: Since your Terraform configuration for policies will be using the `tfe` provider (</docs/providers/tfe/index.html>) with very elevated permissions, you might want to use a Sentinel policy to restrict which `tfe` resources the workspace can manage. The example policy repo includes a policy (https://github.com/hashicorp/tfe-policies-example/blob/master/tfe_policies_only.sentinel) that only allows `tfe_sentinel_policy` and `tfe_policy_set` resources in the policy workspace.

Terraform Configuration

Next, write a Terraform configuration to manage your policies and policy sets in TFE using the `tfe` provider (</docs/providers/tfe/index.html>).

Example: The example policy repo includes a complete Terraform configuration for policies (<https://github.com/hashicorp/tfe-policies-example/blob/master/main.tf>), with comments for clarity. If you prefer to read the Terraform code without a guided walkthrough, you can skip to the next section.

Define Variables for Accessing TFE

The `tfe` provider needs a highly privileged Terraform Enterprise API token in order to manage policies. The best way to handle this type of secret is with a sensitive variable in a TFE workspace. To enable this, define a variable in the configuration.

```
variable "tfe_token" {}
```

If you use a private install of TFE or multiple TFE organizations (or if you might do so in the future), you can set your TFE hostname and TFE organization as variables:

```

variable "tfe_hostname" {
  description = "The domain where your TFE is hosted."
  default     = "app.terraform.io"
}

variable "tfe_organization" {
  description = "The TFE organization to apply your changes to."
  default     = "example_corp"
}

```

Optional: Define a Workspace IDs Variable

The `tfe_policy_set` resource uses workspace IDs, which can be found on a workspace's settings page ([/docs/enterprise/workspaces/settings.html#id](#)). You can use these IDs directly, but the configuration will be more readable if you provide a map of names to IDs and refer to workspaces by name throughout the configuration. Use a Terraform variable for this map, so you can update it in TFE without changing the configuration:

```

variable "tfe_workspace_ids" {
  description = "Mapping of workspace names to IDs, for easier use in policy sets."
  type       = "map"

  default = {
    "app-prod"      = "ws-LbK9gZEL4beEw9A2"
    "app-dev"        = "ws-uMM93B6XrmCwh3Bj"
  }
}

```

To quickly get a list of workspace names and IDs, you can make an API call to the [List Workspaces endpoint](#) ([/docs/enterprise/api/workspaces.html#list-workspaces](#)) and pipe the result to a `jq` command:

```

$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/organizations/my-organization/workspaces?page%5Bnumber%5D=1&page%5Bsize%5D=100 \
| jq --raw-output '.data[] | "\\"(.attributes.name)\\" = \"\\"(.id)\\\""

```

Note: If you have a very large number of workspaces, you might need to make multiple API calls to fetch subsequent pages. See API Docs: Pagination ([/docs/enterprise/api/index.html#pagination](#)) for more details.

Configure the `tfe` Provider

Configure the `tfe` provider (version 0.3 or higher) with your API token and hostname variables.

```

provider "tfe" {
  hostname = "${var.tfe_hostname}"
  token   = "${var.tfe_token}"
  version  = "~> 0.4"
}

```

Create Policy Resources

Create a `tfe_sentinel_policy` resource ([/docs/providers/tfe/r/sentinel_policy.html](#)) for each Sentinel policy in the repo.

- Set the resource name and the `name` attribute to the policy's filename, minus the `.sentinel` extension.
- Use the `file()` function to pull the policy contents into the `policy` attribute.

```
resource "tfe_sentinel_policy" "aws-block-allow-all-cidr" {  
    name      = "aws-block-allow-all-cidr"  
    description  = "Avoid nasty firewall mistakes (AWS version)"  
    organization = "${var.tfe_organization}"  
    policy      = "${file("./aws-block-allow-all-cidr.sentinel")}"  
    enforce_mode = "hard-mandatory"  
}
```

Create Policy Set Resources

Create a `tfe_policy_set` resource ([/docs/providers/tfe/r/policy_set.html](#)) for each policy set you wish to create, and specify which policies are part of the set. Each policy set must also set a value for either `global` or `workspace_external_ids`, to specify which workspaces it should be enforced on.

Note: See Managing Sentinel Policies ([/docs/enterprise/sentinel/manage-policies.html](#)) for a complete description of TFE's policy sets.

- To build the `policy_ids` list, interpolate `id` attributes from your policy resources, like `"${tfe_sentinel_policy.aws-block-allow-all-cidr.id}"`.
- To build the `workspace_external_ids` list, interpolate values from your name-to-ID map variable, like `"${var.tfe_workspace_ids["app-prod"]}"`

```

# A global policy set
resource "tfe_policy_set" "global" {
  name          = "global"
  description   = "Policies that should be enforced on ALL infrastructure."
  organization  = "${var.tfe_organization}"
  global        = true

  policy_ids = [
    "${tf_fe_sentinel_policy.aws-restrict-instance-type-default.id}",
  ]
}

# A non-global policy set
resource "tfe_policy_set" "production" {
  name          = "production"
  description   = "Policies that should be enforced on production infrastructure."
  organization  = "${var.tfe_organization}"

  policy_ids = [
    "${tf_fe_sentinel_policy.aws-restrict-instance-type-prod.id}",
  ]

  workspace_external_ids = [
    "${var.tfe_workspace_ids["app-prod"]}",
  ]
}

```

Importing Resources

If your TFE organization already has some policies or policy sets, make sure to include them when writing your Terraform configuration.

To bring the old resources under management, you can either delete them and let Terraform re-create them, or import them into the Terraform state ([/docs/import/index.html](#)).

For the specific `terraform import` commands to use, see the documentation for the `tfe_sentinel_policy` resource ([/docs/providers/tfe/r/sentinel_policy.html](#)) and the `tfe_policy_set` resource ([/docs/providers/tfe/r/policy_set.html](#)). You can find policy and policy set IDs in the URL bar when viewing them in TFE.

Be sure to import any resources **after** you have created a TFE workspace for managing policies, but **before** you have performed any runs in that workspace.

TFE Workspace

Create a new TFE workspace ([/docs/enterprise/workspaces/creating.html](#)) linked to to your policy management repo.

Note: This workspace doesn't have to belong to the organization whose policies it manages. If you use multiple TFE organizations, one of them can manage policies for the others.

Before performing any runs, go to the workspace's "Variables" page and set the following Terraform variables (using whichever names you used in the configuration):

- `tfe_token` (mark as "Sensitive") — A TFE API token that can manage your organization's Sentinel policies. This token

must be one of the following:

- The organization token (</docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts>) (recommended)
 - The team token (</docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts>) for the owners team (</docs/enterprise/users-teams-organizations/teams.html#the-owners-team>)
 - A user token (</docs/enterprise/users-teams-organizations/users.html#api-tokens>) from a member of the owners team
- `tfe_organization` — The name of the organization you want to manage policies for.
 - `tfe_hostname` — The hostname of your TFE instance.
 - `tfe_workspace_ids` (mark as "HCL") — A map of workspace names to workspace IDs. (See above.)

Once the variables are configured, you can queue a Terraform run to begin managing policies.

Policy Tests

It's easier and safer to collaborate on policy as code when your code is well-tested. Take advantage of Sentinel's built-in testing capabilities by adding tests to your policy repo.

Example: The example policy repo includes tests for every policy (<https://github.com/hashicorp/tfe-policies-example/tree/master/test/>).

See the Sentinel testing documentation (<https://docs.hashicorp.com/sentinel/writing/testing>) to learn how to write and run tests. In brief, you should:

- Create a `test/<NAME>` directory for each policy file.
- Add JSON files (one per test case) to those directories. The object in each file should include the following keys:
 - `mock` — Mock data that represents the test case. Usually you'll mock data for one or more of the Terraform imports; some policies might also require additional imports. For more details about mocking Terraform imports, see:
 - Mocking `tfplan` data (</docs/enterprise/sentinel/import/mock-tfplan.html>)
 - Mocking `tfconfig` data (</docs/enterprise/sentinel/import/mock-tfconfig.html>)
 - Mocking `tfstate` data (</docs/enterprise/sentinel/import/mock-tfstate.html>)
 - `test` — Expected results for the policy's rules in this test case. (If the only expected result is `"main": true`, you can omit the `test` key.)

For each policy, make at least two tests: one that obeys the policy, and one that violates the policy (using `"test": {"main": false}` so that the failed policy results in a passing test). Add more test cases for more complex policies.

- Run `sentinel test` (in the root of the policy repo) to see results for all of your tests.

An example Sentinel test:

```
{  
  "test": {  
    "main": false,  
    "instance_type_allowed": false  
  },  
  "mock": {  
    "tfplan": {  
      "resources": {  
        "aws_instance": {  
          "always-bad": {  
            "0": {  
              "applied": {  
                "ami": "ami-0afae182eed9d2b46",  
                "instance_type": "t3.2xlarge",  
                "tags": {  
                  "Name": "HelloWorld"  
                }  
              }  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

Continuous Integration

Once you have working Sentinel tests, use your preferred continuous integration (CI) system to automatically run those tests on pull requests to your policy repo.

Example: The example policy repo uses GitHub Actions (<https://developer.github.com/actions/>) to run `sentinel test` for every PR. You can view the repo's Actions workflow (<https://github.com/hashicorp/tfe-policies-example/blob/master/.github/main.workflow>), as well as the code for the example Sentinel test action (<https://github.com/hashicorp/sentinel-github-actions/tree/master/test>).

Managing Sentinel Policies

API: See the Policies API (</docs/enterprise/api/policies.html>) and Policy Sets API (</docs/enterprise/api/policy-sets.html>).

Terraform: See the `tfe` provider's `tfe_sentinel_policy` resource (/docs/providers/tfe/r/sentinel_policy.html) and `tfe_policy_set` resource (/docs/providers/tfe/r/policy_set.html).

Sentinel Policies are rules which are enforced on every Terraform run to validate that the plan and corresponding resources are in compliance with company policies.

Policies and Policy Sets

Policies consist of a Sentinel policy file, a name, and an enforcement level.

Policy sets are groups of policies that can be enforced on workspaces (</docs/enterprise/workspaces/index.html>). A policy set can be enforced on designated workspaces, or to all workspaces in the organization.

After the plan stage of a Terraform run, Terraform Enterprise (TFE) checks every Sentinel policy that should be enforced on the run's workspace. This includes policies from global policy sets, and from any policy sets that are explicitly assigned to the workspace.

Important: In order for a policy to take effect, it must be in one or more policy sets. Individual policies cannot be assigned to workspaces, and policies that are not in any policy sets will never be checked.

Policies and policy sets are managed at an organization level, and only organization owners (</docs/enterprise/users-teams-organizations/teams.html#the-owners-team>) can create, edit or delete them.

Note: The UI controls for managing Sentinel policies are designed for demos and other simple use cases; for complex usage with multiple policies, we recommend storing Sentinel code in version control (</docs/enterprise/sentinel/integrate-vcs.html>) and using your CI system to test and upload policies via TFE's API. In the future, Terraform Enterprise will integrate directly with VCS providers for the Sentinel workflow.

Managing Policies

To manage an organization's Sentinel policies, go to the organization settings (</docs/enterprise/users-teams-organizations/organizations.html#organization-settings>) and navigate to the "Policies" page.

To create a new policy, click the "Create a new policy" button; to edit an existing policy, click its entry in the list. Click the "Create policy" or "Update policy" button when finished.

When creating or editing a policy, the following fields are available:

- **Policy Name:** The name of the policy, which is used in the UI and Sentinel output. Accepts letters, numbers, -, and _. (Names cannot be modified after creation, and this field is disabled when editing an existing policy.)
- **Description:** A brief note about the policy's purpose, displayed in the policy list.
- **Enforcement Mode:** How the policy is enforced when performing a run. The following modes are available:

- **hard-mandatory (cannot override):** This policy is required on all Terraform runs. It cannot be overridden by any users.
 - **soft-mandatory (can override):** This policy is required, but organization owners can override the policy and allow a non-compliant run to continue.
 - **advisory (logging only):** This policy will allow the run to continue regardless of whether it passes or fails.
- **Policy Code:** The text of the Terraform-compatible Sentinel (<https://docs.hashicorp.com/sentinel/app/terraform/>) policy which defines the rules for the Terraform configurations, states and plans. Please note that custom imports are not available for use in Terraform Enterprise at this time.
 - **Policy Sets:** Which policy sets your policy will belong to. Use the drop-down menu and "Add policy set" button to add the policy to a set, and the trash can () button to remove the policy from a set.

To delete an existing policy, navigate to its edit page and use the "Delete policy" button at the bottom of the page.

Managing Policy Sets

To manage policy sets, go to the organization settings (</docs/enterprise/users-teams-organizations/organizations.html#organization-settings>) and navigate to the "Policy Sets" page. Policy sets enforced on all workspaces are marked with a "GLOBAL" tag in this list; other policy sets show how many workspaces they are enforced on.

To create a new policy set, click the "Create a new policy set" button; to edit an existing set, click its entry in the list. Click the "Create policy set" or "Update policy set" button when finished.

When creating or editing a policy set, the following fields are available:

- **Name:** The name of the policy set, which is used in the UI. Must be unique to your organization. Accepts letters, numbers, -, and _.
- **Description:** A description of the policy set's purpose. The description can be any length and supports Markdown rendering.
- **Scope of policies:** Whether the set should be enforced on all workspaces, or only on a chosen list of workspaces.
- **Policies:** Which policies to include in the set. Any policy can be in multiple policy sets. Use the drop-down menu and "Add policy" button to add policies to the set, and the trash can () button to remove policies.
- **Workspaces:** Which workspaces the policy set should be enforced on. This is only shown when the scope of policies is set to "Policies enforced on selected workspaces." Use the drop-down menu and "Add workspace" button to add workspaces, and the trash can () button to remove them.

Upgrading From Terraform Enterprise (Legacy)

If you used the legacy version of Terraform Enterprise (TFE), you probably have some older environments that aren't available in the new version. You can transfer control of that infrastructure to the new Terraform Enterprise without re-provisioning those resources.

Follow these steps to migrate your old TFE environments to new TFE workspaces.

API: See the Create a Workspace endpoint (</docs/enterprise/api/workspaces.html#create-a-workspace>); the `data.attributes.migration-environment` property enables migration from legacy environments. For end-to-end instructions for migrating many legacy environments, see Batch Migration (</docs/enterprise/upgrade/batch.html>).

Step 1: Create a New Organization

You can't use the current version of TFE with a legacy organization because the internals are too different. If you don't already have an organization in the new TFE, do the following:

1. Create an organization (</docs/enterprise/getting-started/access.html#creating-an-organization>).
2. Configure version control access (</docs/enterprise/vcs/index.html>). Use the same VCS account(s) that you used in your legacy TFE organization, so that the new organization can access the same repositories.

Note: Naming Your Organizations

Organization names are globally unique, so your old and new organizations must have different names.

If you want to re-use your existing organization's name, you can rename your legacy organization (go to your account settings, choose your legacy organization from the sidebar, and change the Username field) before creating your new organization. However, this can cause problems if you plan to continue Terraform runs in your legacy organization for a while.

Step 2: Verify Your Permissions

To migrate legacy environments to new workspaces, you must be a member of the owners team in both the new organization and the legacy organization. Make sure that your currently logged-in TFE user belongs to both owners teams.

Step 3: Migrate Environments to the New Organization

Follow these steps for each legacy environment you want to migrate to your new organization.

Step 3.a: Ensure the Legacy Environment is Ready

Navigate to your legacy TFE environment and ensure your environment is in a stable state before migrating. In particular:

- Make sure that no runs are currently in progress, and that the most recent plan has been applied.
- Make sure that the environment is either unlocked, or locked by your currently logged-in user account.
- Check with your colleagues and make sure no one needs to make changes to this infrastructure while you are migrating.

If you start migrating with a run in progress or with the workspace locked by a different user, the migration will fail.

Step 3.b: Create a New Workspace Using the "Import" Tab

Navigate to your new TFE organization, make sure you're on the main workspaces list (/docs/enterprise/workspaces/index.html), and click the "+ New Workspace" button.

On the new workspace page, there are two tabs beneath the "Create a new Workspace" header. Click the one labeled "Import from legacy (Atlas) environment".

Create a new Workspace

New workspace **Import from legacy (Atlas) environment**

This workspace will be created under the current organization, **example_corp**.

Heads up:
The current state, environment variables, workspace variables, version control settings, environment settings and teams access settings will be copied from the legacy environment to this workspace. State, run and configuration history will not be migrated; contact support about history migration. Learn more about this import in the [Upgrade guide](#).

LEGACY ENVIRONMENT
examplecorp_legacy/minimum-prod
The VCS settings will be auto-filled if available.

WORKSPACE NAME
minimum-prod
The name of your workspace is unique and used in tools, routing, and UI. Dashes, underscores, and alphanumeric characters are permitted. Learn more about [naming workspaces](#).

SOURCE
None **GitHub** +

REPOSITORY
nfagerlund/terraform-minimum
The repository identifier in the format username/repository. Only the most recently updated repositories will appear with autocomplete; however, all repositories are available for use.

Create Workspace **Cancel**

This import tab has fewer settings than the new workspace tab (/docs/enterprise/workspaces/creating.html), since it migrates most settings from the legacy environment. Fill the fields as follows:

- In the "Legacy Environment" field, enter the environment to migrate, in the form <LEGACY ORGANIZATION>/<ENVIRONMENT NAME>.
- In the "Workspace Name" field, enter the new name of the workspace, which should usually be the same as the old environment name.

- Choose the same VCS provider and repository as the old environment, or choose "None" if you plan to push configurations via the API (/docs/enterprise/run/api.html) or with the optional TFE CLI tool (/docs/enterprise/run/cli.html).

After filling the fields, click the "Create Workspace" button.

After a brief delay the migration should finish, with a "Configuration uploaded successfully" message.

This screenshot shows the HashiCorp Terraform Enterprise (TFE) interface. At the top, there's a navigation bar with a logo, the organization name 'example_corp', and tabs for 'Workspaces' and 'Modules'. On the right of the nav bar are links for 'Documentation' and 'Status'. Below the nav bar, the page title is 'minimum-prod'. Underneath the title, there are tabs for 'Current Run', 'Runs' (which is highlighted in blue), 'States', 'Variables', 'Settings', 'Integrations', and 'Access'. A message box at the top of the main content area says: 'This workspace has been manually locked by nfagerlund. It must be unlocked before runs can execute. You can manage the workspace lock in [settings](#)'. Below this, a green success toast message says 'Success: Workspace created'. The main content area contains a heading 'Configuration uploaded successfully' with a checkmark icon, followed by a paragraph of text: 'Your configuration has been uploaded. Next, you probably want to configure variables (such as access keys or configuration values). If your configuration doesn't require variables, you can queue your first plan now.' At the bottom of the main content area are two buttons: 'Configure Variables' (in blue) and 'Queue Plan' (in white). At the very bottom of the page, there's a footer with links for 'Terms', 'Privacy', 'Security', and '© 2018 HashiCorp, Inc.'

If the migration fails, the error message should explain the problem; most commonly, migration fails when a different user has locked the legacy environment, and you must unlock it to proceed. If you encounter an error you can't recover from, please contact HashiCorp support.

Step 3.c: Inspect the New Workspace's Settings

The migration should result in a new workspace with the same data and settings as the legacy environment. Compare the two to ensure everything is as expected. In particular, note that:

- TFE updates your new organization's teams and team membership to match the legacy environment. If necessary, it will create new teams and/or add users to existing teams. Carefully verify your team settings before continuing.
- Environment variables and Terraform variables are copied to the new workspace, including sensitive values. However, the new TFE does not support personal organization variables; if you used personal variables, you might need to add additional variables to the workspace.
- VCS repo settings (like working directory) should match those from the legacy environment.

Note: TFE immediately imports the settings and current state, and asynchronously imports historical states and runs. If you don't see your previous runs after migrating a workspace, this is normal; they will appear later.

Step 3.d: Set ATLAS_TOKEN (Optional)

Important: This step is **only** for configurations that access state from a legacy TFE environment using a `terraform_remote_state` data source. If this environment doesn't do that, skip to the next step.

TFE usually makes it easy to share state data between workspaces by automatically handling authentication. However, this only works *within* a single organization, and your legacy environments are in a separate organization from your new workspaces. If your Terraform configuration accesses remote state from a legacy TFE environment, you must manually provide credentials until all of the relevant environments are migrated to the new TFE.

To do this, create a new environment variable in the new workspace called `ATLAS_TOKEN` and enter a valid TFE user API token as its value. Mark the variable as sensitive to protect the token.

The user account that owns this token must be a member of both the new and legacy organizations, and must have the following permissions:

- **Read** access to any legacy environments that the configuration needs to read state from.
- **Write** access to the new workspace where the configuration will run.

If you use this workaround during your migration process, you must do two additional steps after migrating all of your legacy environments:

1. Edit this workspace's configuration so it reads state data from the new workspaces that replaced your legacy environments.
2. At the same time, delete `ATLAS_TOKEN` from this workspace's variables.

Make a note of any workspaces that access remote state as you migrate them, and update them soon after you finish migrating. Leaving a value for `ATLAS_TOKEN` over the long term can make your workspaces unnecessarily fragile.

Step 3.e: Queue Plan and Check Results

After verifying the settings and variables, queue a plan in the new workspace.

If everything matches the legacy environment, **the plan should complete with no changes** (or the expected changes if the VCS repository has new commits). If the plan would result in changes, inspect it carefully to find what has changed and update variables or settings if necessary.

At this point, your new workspace is ready for normal operation. The legacy environment is locked and will perform no more runs unless someone in your organization unlocks it.

Step 4: Delete Legacy Environments (Optional)

When you have no more need for your legacy environments, you can delete them. This is optional; if left in place, they will be automatically deleted when the administrators of your TFE instance end support for legacy environments.

Before deleting, make sure the legacy environment is no longer needed. In particular, be aware of the following:

- If any other Terraform configurations read the legacy state data with a `terraform_remote_state` data source, update them to reference the new workspace instead.

If you delete an environment before updating dependent configurations, it can cause run failures in other workspaces.

- If you want to preserve state history and run history from the legacy environment, wait until the history migration has finished before deleting.

History migration is a silent background process, and TFE does not display its status to normal users. However, site admins can view migration status in TFE's admin pages. To find out whether migration has finished, wait several days after migrating your final legacy environment, then contact the administrators of your TFE instance. For the SaaS version of TFE, email HashiCorp support; for private installs, contact your organization's admins.

If you are an administrator of your private TFE instance, you can view the current status of all history migrations on the Migrations page of the site admin section (</docs/enterprise/private/admin/resources.html#migration-of-workspace-history-from-legacy-environments>).

Batch Migration

These instructions are for migrating TFE (Atlas) legacy environments to workspaces using the API and scripts to automate large batches. Using the UI is the easiest way to migrate a legacy environment to a workspace; however, if your organization has many environments in need of migration this process may be laborious to perform in the UI.

The TFE workspace creation API (</docs/enterprise/api/workspaces.html>) can be used to perform the migration, which automatically locks the legacy environment and migrates the data to the new workspace. This API can be called using standard tools like curl, tfe-cli tool (<https://github.com/hashicorp/tfe-cli>), or community tools like the terraform-enterprise-client (<https://github.com/skierkowski/terraform-enterprise-client>). These tools can be used to iterate over a list of legacy environments to make the API calls to perform the migration.

These instructions show how to automatically migrate a batch of legacy environments using tfe-cli (<https://github.com/hashicorp/tfe-cli>), some basic scripting, and a file with legacy environments. You can follow the steps as written, or use them as an example for building your own automation.

Prerequisites

This guide requires the tfe-cli tool (<https://github.com/hashicorp/tfe-cli>).

Step 1: Identify OAuth Token ID

Note: You can skip to **Step 2** if only one VCS connection is configured or if no VCS connection is needed. The tfe tool will automatically use the VCS connection if only one exists.

To get the OAuth connection we assume you already have VCS integration set up in the new organization. You will need the OAuth ID of the connection. You can obtain this with the command line tool:

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  https://app.terraform.io/api/v2/organizations/my-organization/oauth-tokens
```

This will be the id value of the oauth-tokens.

Step 2: Set Up Files and Script

These files assume that a VCS connection is present and multiple oauth-tokens are configured. If no VCS connection is present then the legacy-envs.txt nad migrate.sh will need to be updated to remove references to the VCS repo. The oauth-id option in migrate.sh will also need to be removed if no VCS connection is used or if you'd like the one default connection to be used.

Data File: legacy-envs.txt

To prepare for migration, create a text file with all of the data you need.

```
skierkowski/training my-organization/training skierkowski/terraform-test-proj  
skierkowski/migration-source my-organization/migration-desintation skierkowski/terraform-test-proj
```

Each line identifies a legacy environment, the new workspace to migrate to, and the VCS repo to use, formatted as:

```
<legacy_org>/<legacy_environment> <new_organization>/<new_workspace> <vcs_org>/<vcs_repo>
```

If we were omitting VCS connections from the new workspaces, we would omit the repo from each line.

Script: migrate.sh

Next, create a bash script which:

- Opens the file `legacy-envs.txt`.
- Parses out the legacy workspaces, new workspaces, and VCS repo.
- Calls `tfe` command line tool to trigger the migration.

Note: You'll need to specify the path to `tfe` command line tool.

```
#!/bin/bash

regex="([a-zA-Z0-9_\-\-]+)\\/([a-zA-Z0-9_\-\-]+) ([a-zA-Z0-9_\-\-]+)\\/([a-zA-Z0-9_\-\-]+) ([a-zA-Z0-9_\-\-]+)\\/([a-zA-Z0-9_\-\-]+)"

cat legacy-envs.txt | while read line
do
  if [[ $line =~ $regex ]]
  then
    legacy_org="\${BASH_REMATCH[1]}"
    legacy_workspace="\${BASH_REMATCH[2]}"
    new_org="\${BASH_REMATCH[3]}"
    new_workspace="\${BASH_REMATCH[4]}"
    vcs_org="\${BASH_REMATCH[5]}"
    vcs_repo="\${BASH_REMATCH[6]}"
    echo MIGRATING: $line
    tfe migrate \
      -legacy-name $legacy_org/$legacy_workspace \
      -name $new_org/$new_workspace \
      -vcs-id $vcs_org/$vcs_repo \
      -oauth-id $VCS_OAUTH_TOKEN_ID
  else
    echo ERROR PARSING: $line
  fi
done
```

Step 3: Run migration

Now you can run the batch migration:

```
chmod +x migrate.sh  
./migrate.sh
```

The `migrate.sh` script does not perform any error handling. Review the logs to ensure all workspaces are migrated properly.

Differences Between Current and Legacy Terraform Enterprise

Terraform Enterprise began as part of the discontinued Atlas suite, and today's version is a redesign built for using Terraform in a team setting. Most notably, it focuses heavily on a VCS-driven run workflow (</docs/enterprise/run/ui.html>), while providing a rich API to support alternate workflows.

If you've previously used the legacy version of Terraform Enterprise, we hope you enjoy the new version's improvements. However, as part of this redesign, some features have been removed and others are not yet implemented.

Several things are significantly different in the new Terraform Enterprise. This page is a brief summary of what to expect when upgrading from the legacy version.

Renamed or Moved Features

Workspaces Replace Environments

Environments are now called workspaces (</docs/enterprise/workspaces/index.html>), to make Terraform Enterprise's terminology consistent with Terraform open source. Aside from the name change, workspaces function the same as environments did.

Terraform Push

The new Terraform Enterprise does not support the `terraform push` command, which is now deprecated.

This does not mean we're dropping support for a CLI-based workflow — we want push to be better, and the integrated push command was holding us back because we built it on an outdated understanding of how Terraform Enterprise should work. Users of `terraform push` have a few options:

- Use the add-on Terraform Enterprise command line tools (<https://github.com/hashicorp/tfe-cli>). These are designed as a one-to-one replacement for `terraform push`, with a modified syntax and some new features.
- Use the API-based run workflow (</docs/enterprise/run/api.html>) to upload a configuration tabball and queue a run. This is a more powerful option for integration with automated systems.

We're also working on a more deeply integrated CLI workflow that combines the collaborative benefits of centrally managed Terraform runs with the interactive responsiveness of running Terraform on your laptop. This doesn't have a release date yet, but we hope our current replacements for `terraform push` offer a comfortable migration path in the meantime.

Event Log is Now Audit Log (Private Installs Only)

Legacy Terraform Enterprise included an event log that logged details about run events. For the new Terraform Enterprise, we replaced it with a comprehensive audit logging system that also logs all changes to organization and workspace settings.

The new audit log is available on private installs of Terraform Enterprise. See the audit log documentation (</docs/enterprise/private/logging.html#audit-logs>) for details.

ATLAS_CONFIRM_DESTROY is now CONFIRM_DESTROY

In order to queue a destroy plan on an environment (legacy) or workspace (current), you have to confirm your intentions by setting a special environment variable. The name of that variable has changed to CONFIRM_DESTROY, as part of our move away from the Atlas brand name.

Gone But Returning Soon

Notifications

The new Terraform Enterprise doesn't yet support arbitrary notifications at the end of runs. We plan to add a webhook-based notification system in a future update.

Shared Variables

Legacy Terraform Enterprise supported organization-level variables that could be shared across multiple environments, which made it easier to re-use common variable values.

Unfortunately, this feature was entangled with personal variables, which we permanently removed. We're planning a replacement feature to let you use shared sets of variables across multiple workspaces, but we're still in the process of designing it.

Periodically Queued Plans

In legacy Terraform Enterprise, you could set an environment to periodically queue a plan regardless of whether the configuration had changed. The new Terraform Enterprise does not currently have this feature.

We haven't yet determined whether to bring periodic runs back. We think the run API (</docs/enterprise/api/run.html>) probably makes it unnecessary — your CI system can start runs in response to any kind of external trigger, and any kind of scheduler (including a cron job or a Nomad job) can start runs on a timer. If you used periodic runs heavily, please contact us and let us know whether the current feature set meets your needs.

Permanently Removed

We deliberately removed these features to streamline and improve Terraform Enterprise's core workflow. Some of these features worked against or distracted from what Terraform Enterprise does best, and some are good ideas that are handled better by external services.

We understand that removing features can be painful and disappointing for those who have built processes around them. For each of these features, we ensured only a single-digit percentage of users would be affected, and have tried to mitigate the changes with guides to other systems when appropriate. If you're badly affected by these removals and our existing documentation isn't adequate, please contact us and we'll work with you to help you migrate.

Packer Integration and Artifact Registry

Terraform Enterprise no longer includes an integrated Packer build system or an artifact registry for built images. Different organizations have such different requirements and expectations for image builds that we believe running Packer out of your own CI/CD system (<https://www.packer.io/guides/packer-on-cicd/index.html>) will usually provide a better experience.

Individual Collaborators on Environments

Legacy Terraform Enterprise could grant access permissions to individual users; the new Terraform Enterprise handles all workspace permissions via teams of users (</docs/enterprise/users-teams-organizations/teams.html>).

We removed individual access because it eventually makes workspace settings too complex. Team-based access is easier to update and govern — single-user teams offer the same granularity of permissions, but with faster updates as soon as they need to become five-user or zero-user teams.

Personal Variables

Legacy Terraform Enterprise had a personal variables feature, where each user could set values that only applied to runs that they queued.

We removed this feature because it worked against Terraform Enterprise's goal of collaborative and predictable infrastructure. Not only did it make runs inconsistent and less reliable, it also expected that users would be queueing most runs manually in the UI, when we've found that people want most runs to happen in response to changes in a VCS repo or other automatic triggers.

If you used personal variables to avoid sharing credentials, you can mark workspace variables as sensitive (</docs/enterprise/workspaces/variables.html#sensitive-values>).

Users, Teams, and Organizations

Terraform Enterprise's organizational and access control model is based on three units: users, teams, and organizations.

- Users (</docs/enterprise/users-teams-organizations/users.html>) are individual members of an organization. They belong to teams, which are granted permissions on an organization's workspaces.
- Teams (</docs/enterprise/users-teams-organizations/teams.html>) are groups of users that reflect your company's organizational structure. Organization owners can create teams and manage their membership.
- Organizations (</docs/enterprise/users-teams-organizations/organizations.html>) are shared spaces for teams to collaborate on workspaces. An organization can have many teams, and the owners of the organization set which teams have which permissions (read/write/admin) on which workspaces.

Two-factor Authentication

User accounts can be additionally protected with two-factor authentication (2FA), and an organization owner can make this a requirement for all users.

Setting up Two-factor Authentication

To reach your user security settings page, click the user icon in the upper right corner and choose "User Settings" from the menu.

The screenshot shows the 'Security' tab selected in the left sidebar under 'ACCOUNT SETTINGS'. The main section is titled 'Password' and contains fields for 'NEW PASSWORD' and 'CONFIRM NEW PASSWORD'. Below these is a 'CURRENT PASSWORD' field with a note: 'You need to enter your current password to set a new password.' A 'Change password' button is present. The 'Two Factor Authentication' section follows, with a note about adding an extra layer of security. It offers 'Application' or 'SMS Only (Text Message)' options, a 'PHONE NUMBER' input field containing '(123) 456-7890', and a note about specifying a country code. An 'Enable 2FA' button is at the bottom.

Once on this page you can set-up authentication with either a TOTP-compliant application and/or an SMS-enabled phone number. Choose your preferred authentication method and enter a phone number (optional if using an application), then follow the instructions to finish the configuration. If you are using an application, you must scan a QR code to enable it; for either method, you must enter valid authentication codes to verify a successful set-up.

After you finish, the two-factor authentication settings will change to show your currently configured authentication method. You can click the "Reveal codes" link to view backup codes, or use the "Disable 2FA" button to disable two-factor authentication.

Two Factor Authentication

ENABLED

[Disable 2FA](#)

Application is configured

Your backup codes are displayed below. Copy them and store in a safe place. Each backup code can be used once to sign in if you do not have access to your two-factor authentication device.

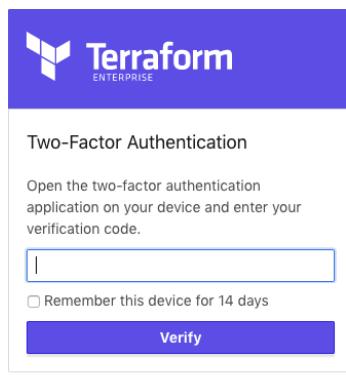
[Reveal codes !\[\]\(a648d5e1fb005c35f56e344417f21f30_img.jpg\)](#)



Logging in with Two-factor Authentication

After two-factor authentication has been successfully set-up you will need to enter the code from your TOTP-compliant application or from an SMS sent to your approved SMS-enabled phone number on login.

If necessary you can also use a backup code by clicking "Use a recovery code". Please remember that each backup code can only be used to log in once.



Requiring Two-factor Authentication for All Users

API: See the Update an Organization endpoint (</docs/enterprise/api/organizations.html#update-an-organization>) — the request body's `data.attributes.collaborator-auth-policy` property manages this setting.

If you are an organization owner you can require all users within your organization to use two-factor authentication.

To reach your organization settings page, click the name of your organization in the upper left corner and choose "Organization Settings" from the menu. On this page click "Authentication" on the left navigation menu.

The screenshot shows the 'rkthng' organization settings page. The left sidebar has 'rkthng' selected under 'ORGANIZATION'. Under 'Authentication', the 'Require two-factor' button is circled in red. The main content area is titled 'Authentication' and contains sections for 'User Sessions' and 'Two-factor Authentication'. The 'User Sessions' section includes 'TIMEOUT IN' and 'REMEMBER FOR' fields, both with '20160' and 'Use default' checked. The 'Two-factor Authentication' section states: 'As an organization owner, you can require that all members of your organization's teams enable two-factor authentication. They will be locked out until they enable two-factor authentication or leave all teams connected with your organization.' A 'Disable two-factor' button is present.

Click the button "Require two-factor". Please remember that all organization owners must have two-factor authentication on before this can be set.

This screenshot shows the same 'rkthng' organization settings page as the previous one, but the 'Require two-factor' button in the 'Two-factor Authentication' section is now circled in red. All other elements, including the sidebar and the 'User Sessions' configuration, appear identical to the first screenshot.

Organizations

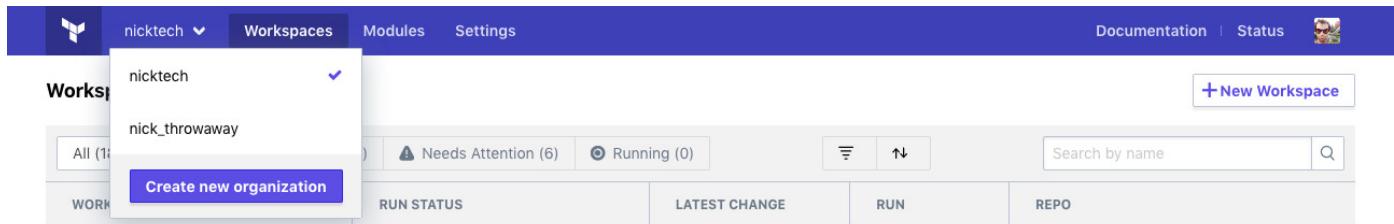
Organizations are a shared space for teams (/docs/enterprise/users-teams-organizations/teams.html) to collaborate on workspaces in Terraform Enterprise (TFE).

API: See the Organizations API (/docs/enterprise/api/organizations.html).

Terraform: See the `tfe_organization` resource (/docs/providers/tfe/r/organization.html).

Selecting Organizations

On most pages within TFE, the top navigation bar displays the name of the selected organization. Clicking the name reveals the organization switcher menu, which lists all of the organizations you belong to. You can switch to another organization by clicking its name, or you can create a new organization with the "Create new organization" button.



Adding Users to Organizations

Organization membership is automatic, and is determined by team membership. To add a user to an organization, add them to one or more of that organization's teams. See Teams (/docs/enterprise/users-teams-organizations/teams.html) for more information.

You can only add existing user accounts to teams. If a colleague has not created their TFE account yet, send them the sign-up link (<https://app.terraform.io/account/new> for SaaS, <https://<TFE HOSTNAME>/account/new> for private installs) and ask them to send you their username.

Creating Organizations

Users can create new TFE organizations by clicking the "Create new organization" button in the organization switcher menu. The new organization page also displays automatically when the currently logged-in user does not belong to any organizations, as when first logging in as a new user.



New Organization

ORGANIZATION NAME

This will be part of your resource names used in various tools, i.e. `hashicorp/www-prod`.

EMAIL ADDRESS

The organization email is used for any future notifications, such as billing, and the organization avatar, via [gravatar.com](#).

Create organization

To create a new organization, provide a unique name and a primary contact email address. Organization names can include numbers, letters, underscores (_), and hyphens (-).

Once you have created an organization, you can add other users ([/docs/enterprise/users-teams-organizations/users.html](#)) by adding them to one or more teams ([/docs/enterprise/users-teams-organizations/teams.html](#)).

Note: On the SaaS version of TFE, any user can create a new organization. On private installs of TFE, the administrators can restrict this ability, so that only site admins can create organizations. See Administration: General Settings ([/docs/enterprise/private/admin/general.html#organization-creation](#)) for more details.

Organization Settings

You can view and manage an organization's settings by clicking the "Settings" link in the top navigation bar.

The screenshot shows the "Organization Profile" page for the organization "nicktech". The left sidebar lists organization settings categories: Profile, Teams, VCS Providers, Integrations, API Tokens, Authentication, Manage SSH Keys, Policies, and Policy Sets. The "Profile" category is currently selected. The main content area displays the organization's name ("nicktech") and notification email ("nick@hashicorp.com"). A warning message states: "Deleting the nicktech organization will permanently delete all workspaces associated with it. Please be certain you know what you're doing. This action cannot be undone." A red "Delete this organization" button is located at the bottom right of this section.

Only organization owners ([/docs/enterprise/users-teams-organizations/teams.html#the-owners-team](#)) can manage an organization's settings. However, other users can use this section to view the organization's contact email, view the membership of any teams they belong to, and view the organization's authentication policy.

Most of the organization settings are documented near the specific workflows they enable. What follows is a brief summary with links to more relevant sections of the documentation.

Profile / Delete

The profile page shows the organization's name and contact email, but does not allow you to change them.

The profile page is also where you can **delete your organization**.

Teams

The teams page allows organization owners to manage the organization's teams, including creating and deleting teams, managing membership, and generating team API tokens.

Users who aren't organization owners can view the list of teams they belong to and the membership of those teams. They can't edit teams or view any teams they don't belong to.

See Teams (</docs/enterprise/users-teams-organizations/teams.html>) for more information.

VCS Provider

The VCS provider page is used for setting up VCS access for TFE. See Connecting VCS Providers (</docs/enterprise/vcs/index.html>) for more information.

API Tokens

Organizations can have a special service account API token that is not associated with a specific user or team. See Service Accounts (</docs/enterprise/users-teams-organizations/service-accounts.html>) for more information.

Authentication

The authentication page allows owners to determine when users must reauthenticate. It also allows owners to require two-factor authentication (</docs/enterprise/users-teams-organizations/2fa.html>) for all members of the organization.

Manage SSH Keys

The SSH keys page manages any keys necessary for cloning Git-based module sources during Terraform runs. It does not manage keys used for accessing a connected VCS provider. See SSH Keys for Cloning Modules (</docs/enterprise/workspaces/ssh-keys.html>) for more information.

Policies and Policy Sets

The policies page is for managing Sentinel policies, and the policy sets page is for assigning groups of policies to workspaces.

Sentinel is an embedded policy-as-code framework that can enforce rules about Terraform runs within an organization. See [Sentinel](/docs/enterprise/sentinel/index.html) (</docs/enterprise/sentinel/index.html>) for more information about Sentinel, or [Managing Sentinel Policies](#) (</docs/enterprise/sentinel/manage-policies.html>) for details about these two settings pages.

Inactive Organizations

TFE is a paid product offered to organizations on a subscription basis; for more information about billing, please speak to a HashiCorp sales representative. TFE is also available as a free trial to organizations evaluating its features.

When a free trial has expired, or when paid billing has been canceled, an organization becomes inactive. Inactive organizations display a banner reading "TRIAL EXPIRED — Upgrade Required" in the top navigation bar:



Members of an inactive organization can still log into TFE, view data about past runs, manage team memberships, and manage access permissions on workspaces. However, inactive organizations cannot initiate new Terraform runs (</docs/enterprise/run/index.html>).

Permissions

Teams can have **read**, **write**, or **admin** permissions on workspaces.

Read

Can read any information on the workspace, including:

- StateVersions
- Runs
- ConfigurationVersions
- Variables

Cannot do anything which alters state of the above.

Plan

Can do everything the read access level can do plus:

- Create runs

Write

Can do everything the plan access level can do plus:

- Execute functions which alter state of the above models.
- Approve runs.
- Edit variables on the workspace.
- Lock and unlock the workspace.

Admin

Can do everything the write access level can do, plus:

- Delete the workspace.
- Add and remove teams from the workspace at any access level.
- Read and write workspace settings (VCS config, etc).

Service Accounts

Terraform Enterprise (TFE) provides two types of service accounts: team, and organization. These accounts can access Terraform Enterprise APIs, but cannot be used interactively. The service accounts are designed to support server-to-server API calls using the service identity as opposed to individual user identities.

Team and organization service accounts don't have API tokens by default; you must generate a token before you can use them.

Like with user tokens ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)), service account tokens are displayed only once when they are created, and are obfuscated thereafter. If the token is lost, it must be regenerated. When a token is regenerated, the previous token immediately becomes invalid.

Important: Unlike user tokens ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)), team and organization tokens cannot authenticate with Terraform's `atlas` backend. This means they can't be used to run Terraform locally against state stored in TFE, access `terraform_remote_state` data sources during a run, or migrate local state into TFE ([/docs/enterprise/migrate/index.html](#)). If you need to use the `atlas` backend for any reason, use a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)).

Team Service Accounts

API: See the Team Token API ([/docs/enterprise/api/team-tokens.html](#)).

To manage the API token for a team service account, go to **Organization settings > Teams > (desired team)** and use the controls under the "Team API Token" header.

Each team can have **one** valid API token at a time, and any member of a team can generate or revoke that team's token. When a token is regenerated, the previous token immediately becomes invalid.

Team service accounts are designed for performing API operations on workspaces. They have access to the same workspaces as their team, at the same access level; for example, if a team has write access to a workspace, the team's token can create runs and configuration versions via the API.

Note that the individual members of a team can usually perform actions the team itself cannot, since users can belong to multiple teams, can belong to multiple organizations, and can authenticate with Terraform's `atlas` backend for running Terraform locally.

Organization Service Accounts

API: See the Organization Token API ([/docs/enterprise/api/organization-tokens.html](#)).

To manage the API token for an organization service account, go to **Organization settings > API Token** and use the controls under the "Organization Tokens" header.

Each organization can have **one** valid API token at a time. Only organization owners ([/docs/enterprise/users-teams-organizations/teams.html#the-owners-team](#)) can generate or revoke an organization's token.

Organization service accounts are designed for creating and configuring workspaces and teams. We don't recommend using them as an all-purpose interface to TFE; their purpose is to do some initial setup before delegating a workspace to a team. For more routine interactions with workspaces, use team service accounts.

Organization service accounts have permissions across the entire organization. They can perform all CRUD operations on most resources, but have some limitations; most importantly, they cannot start runs or create configuration versions. Any API endpoints that can't be used with an organization account include a note like the following:

Note: This endpoint cannot be accessed with organization tokens ([/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts](#)). You must access it with a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) or team token ([/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts](#)).

Teams

Teams are groups of Terraform Enterprise (TFE) users (</docs/enterprise/users-teams-organizations/users.html>) within an organization (</docs/enterprise/users-teams-organizations/organizations.html>). To delegate provisioning work, the organization's owners can grant workspace permissions to specific teams.

Teams can only have permissions on workspaces within their organization, although any user in a team can belong to teams in other organizations.

If a user belongs to at least one team in an organization, they are considered a member of that organization.

The Owners Team

Every organization has a team named `owners`, whose members have special permissions. In TFE's documentation and UI, members of the `owners` team are sometimes called organization owners.

An organization's creator is the first member of its `owners` team; other members can be added or removed in the same way as other teams. Unlike other teams, the `owners` team can't be deleted and can't be empty; if there is only one member, you must add another before removing the current member.

Members of the `owners` team have full access to every workspace in the organization. Additionally, the following tasks can only be performed by organization owners:

- Creating new workspaces
- Managing Sentinel policies
- Overriding soft mandatory Sentinel policies
- Creating and deleting teams
- Managing team membership
- Viewing the full list of teams
- Managing organization settings (</docs/enterprise/users-teams-organizations/organizations.html#organization-settings>)

Managing Teams

Teams are managed in the organization settings (</docs/enterprise/users-teams-organizations/organizations.html#organization-settings>). Click the "Settings" link in the top navigation bar, then click the "Teams" link in the left sidebar.

ORGANIZATION SETTINGS

nicktech

Profile

Teams

VCS Providers

Integrations

API Tokens

Authentication

Manage SSH Keys

Sentinel Policies

Team Management

Teams let you group users into specific categories to allow for finer grained access control policies. For example, your developers could be on a dev team that only has access to applications.

In order to allow a team access to a resource, go to the Access settings for the specific resource and enter the team name. At this point you can control the access level for that team.

The **owners** team is a special team that has implied access for all of your resources, but also has the ability to manage your organization.

Create a New Team

NAME

Create team

Teams

owners

5 members

test_team

3 members

The teams page includes a list of the organization's teams. Clicking a team in the list loads its team settings page, which manages its membership and other settings:

ORGANIZATION SETTINGS
nicktech

Profile

Teams

VCS Providers

Integrations

API Tokens

Authentication

Manage SSH Keys

Sentinel Policies

Team: test_team

Add a New Team Member

USERNAME
 Add member

The username of the user you wish to add to this team. Share the signup link with new users:

<https://app.terraform.io/account/new>

Members (3)



kfishner

2FA



nfagerlund

2FA



skierkowski

2FA



Team API Token

Treat this token like a password, as it can be used to access your account without a username, password, or two-factor authentication.

Last used **4 months ago**

Created **4 months ago** by user **nfagerlund**

Delete this Team

Warning! This will permanently delete this team and any permissions associated with it.

The team settings page lists the team's current members, with badges to indicate which users have two-factor authentication ([/docs/enterprise/users-teams-organizations/2fa.html](#)) enabled.

Only organization owners can manage teams or view the full list of teams. Other users can view the teams page in read-only mode and view any teams they are members of.

Creating and Deleting Teams

API: See the Teams API ([/docs/enterprise/api/teams.html](#)).

Terraform: See the `tfe` provider's `tfe_team` resource ([/docs/providers/tfe/r/team.html](#)).

Organization owners can create new teams from the teams page, using the controls under the "Create a New Team" header.

To create a new team, provide a name (unique within the organization) and click the "Create team" button. Team names can include numbers, letters, underscores (_), and hyphens (-).

To delete a team, go to the target team's settings page and click the "Delete TEAM NAME" button.

Managing Team Membership

API: See the Team Members API (</docs/enterprise/api/team-members.html>).

Terraform: See the `tfe_team_member` resource (/docs/providers/tfe/r/team_member.html) or `tfe_team_members` resource (/docs/providers/tfe/r/team_members.html).

Organization owners can use a team's settings page to add and remove users from the team.

To add a user, enter their username in the "Username" text field (located under the "Add a New Team Member" header) and click the "Add member" button. (You must know the user's TFE username; users cannot be added using email addresses or other personal information.)

To remove a user, click the "—" (trash can) button by their entry in the member list.

Typically, your team structure will mirror your company's group structure. The Terraform Recommended Practices guide (</docs/enterprise/guides/recommended-practices/index.html>) offers more in-depth discussion of how team structure interacts with the structure of your Terraform configurations and the IT infrastructure they manage.

API Tokens

API: See the Team Tokens API (</docs/enterprise/api/team-tokens.html>).

Each team can have a special service account API token that is not associated with a specific user. You can manage this API token from the team's settings page. See Service Accounts (</docs/enterprise/users-teams-organizations/service-accounts.html>) for more information.

Managing Workspace Access

API: See the Team Access API (</docs/enterprise/api/team-access.html>).

Terraform: See the `tfe_team_access` resource (/docs/providers/tfe/r/team_access.html).

A team can be given read, write, or admin permissions on one or more workspaces.

- Use any workspace's "Access" tab to manage team permissions on that workspace. For full instructions, see Managing Access to Workspaces (</docs/enterprise/workspaces/access.html>).
- For detailed information about the available permissions levels, see Permissions (</docs/enterprise/users-teams-organizations/permissions.html>).

When determining whether a user can take an action on a resource, TFE uses the highest permission level from that user's teams. (For example, if a user belongs to a team with read permissions on a workspace and another team with admin permissions on that workspace, that user has admin permissions.)

Users

Users are the individual members of an organization (</docs/enterprise/users-teams-organizations/organizations.html>).

Creating an Account

Users must create an account in Terraform Enterprise (TFE) before they can be added to an organization. Creating an account requires a username, an email address, and a password.

The sign-up page is not linked from TFE's sign-in page, so you should provide the sign-up URL to any colleagues you want to invite to TFE:

- For the SaaS version of TFE, create a new account at <https://app.terraform.io/account/new> (<https://app.terraform.io/account/new>).
- For private installs of TFE, go to <https://<TFE HOSTNAME>/account/new>.

New users do not belong to any organizations.

After you create a new user account, TFE immediately takes you to a page where you can create a new organization.

The screenshot shows the 'New Organization' creation interface. At the top, there is a blue header bar with the HashiCorp logo, 'Documentation', 'Status', and a user profile icon. Below the header, the title 'New Organization' is centered. There are two input fields: 'ORGANIZATION NAME' with a placeholder 'name' and a note explaining it will be part of resource names; and 'EMAIL ADDRESS' with a placeholder 'email' and a note about using it for notifications via Gravatar. A large blue button labeled 'Create organization' is at the bottom of the form.

- If you are the first TFE user in your organization, use this page to create your TFE organization. See Organizations (</docs/enterprise/users-teams-organizations/organizations.html>) for more information.
- If you intend to join an existing organization, do not create a new one; instead, send your username to one of your TFE organization's owners and ask them to add you to a team. Once they have brought you into the organization, you can reload the page to begin using TFE.

Team and Organization Membership

To add a user to an organization, a member of that organization's owners team must add them to one or more teams. See Teams (</docs/enterprise/users-teams-organizations/teams.html>) for more information.

Adding a user to a team requires only their username.

Site Admin Permissions

On private instances of TFE, some user accounts have a special "site admin" permission that allows administration of the entire instance.

Admin permissions are distinct from normal organization-level permissions, and they apply to a different set of UI controls and API endpoints. Although admin users can administer any resource across the instance when using the site admin pages or the admin API (/docs/enterprise/api/admin/index.html), they have a normal user's permissions (with access determined by the teams they belong to) when using an organization's standard UI controls and API endpoints.

For more information, see Administering Private Terraform Enterprise (/docs/enterprise/private/admin/index.html).

User Settings

API: See the Account API (/docs/enterprise/api/account.html).

TFE users can manage many of their own account details, including email address, password, API tokens, and two-factor authentication.

To reach your user settings page, click the user icon in the upper right corner and choose "User Settings" from the menu.

Once on this page, can use the navigation on the left to choose which settings to manage.

The screenshot shows the 'Profile' section of the User Settings page. On the left, a sidebar lists 'USER SETTINGS' with options: 'Profile' (selected), 'Password', 'Two Factor Authentication', and 'Tokens'. The main content area has a title 'Profile'. It contains fields for 'USERNAME' (nfagerlund) with a note about changing it, 'EMAIL ADDRESS' (redacted) with a note about confirming a new email, and an 'AVATAR' section featuring a placeholder image of a man wearing sunglasses. A blue 'Update profile' button is at the bottom.

Profile

TFE user profiles are very small, consisting only of a username and an email address. You can change either of these from the "Profile" page of the user settings.

Important: Changing your username can cause important operations to fail. This is because it is used in URL paths to various resources. If external systems make requests to these resources, you'll need to update them prior to making a change.

Additionally, TFE uses Gravatar (<http://en.gravatar.com>) to display a user icon if you have associated one with your email address. For details about changing your user icon, see Gravatar's documentation (<http://en.gravatar.com/support/>).

Password Management

Users manage their own passwords. To change your password, click the "Password" page of the user settings. You'll need to confirm your current password, and enter your new password twice.

Password management isn't available if your TFE installation uses SAML single sign on (</docs/enterprise/saml/index.html>).

Two-Factor Authentication

For additional security, you can enable two-factor authentication, using a TOTP-compliant application or an SMS-capable phone number. Depending on your organization's policies, you might be required to enable two-factor authentication.

For more details, see Two-Factor Authentication (</docs/enterprise/users-teams-organizations/2fa.html>).

API Tokens

Users can create any number of API tokens, and can revoke existing tokens at any time. To manage API tokens, click the "Tokens" page of the user settings.

API tokens are necessary for:

- Authenticating with the Terraform Enterprise API (</docs/enterprise/api/index.html>). API calls require an `Authorization: Bearer <TOKEN>` HTTP header.
- Authenticating with the Terraform `atlas` backend (</docs/backends/types/terraform-enterprise.html>). The backend looks for a token in the `ATLAS_TOKEN` environment variable.
- Using private modules (</docs/enterprise/registry/using.html>) in command-line Terraform runs on local machines. This requires a `credentials` block (</docs/enterprise/registry/using.html#configuration>) in your `~/.terraformrc` file.

Terraform Enterprise has three kinds of API tokens: user, team, and organization. For more information about team and organization tokens, see Service Accounts (</docs/enterprise/users-teams-organizations/service-accounts.html>).

Protect your tokens carefully, because they can do anything your user account can. For example, if you belong to a team with write access to a workspace, your API token can edit variables in that workspace. (See Permissions (</docs/enterprise/users-teams-organizations/permissions.html>) for details about workspace permissions.)

Since users can be members of multiple organizations, user tokens work with any organization their user belongs to.



nicktech ▾

Workspaces

Modules

Settings

Documentation | Status



USER SETTINGS

Profile

Password

Two Factor Authentication

Tokens

Tokens

Generate new token

DESCRIPTION

Generate token

Use this description to help identify the use of the token in the future.

Tokens (1)

For authenticating backends on CLI

Created 7 months ago



- To create a new token, enter a comment to identify it and click the "Generate token" button.

A token is only displayed once, at the time of creation; if you lose it, you will need to revoke the old token and create a new one. Make sure your description includes enough information so you know which token to revoke later.

- To revoke a token, click the " " (trash can) icon button next to the token's description. That token will no longer be able to authenticate as your user account.

Note: When SAML SSO is enabled there is a session timeout for user API tokens, forcing users to periodically reauthenticate through the web UI in order to keep their tokens active. See the API Token Expiration ([/docs/enterprise/saml/login.html#api-token-expiration](#)) section in the SAML SSO documentation for more details.

Connecting VCS Providers to Terraform Enterprise

Terraform Enterprise (TFE) is more powerful when you integrate it with your version control system (VCS) provider. Although you can use almost all of TFE's features without one, a VCS connection provides major workflow benefits. In particular:

- When workspaces are linked to a VCS repository, TFE can automatically initiate Terraform runs (</docs/enterprise/run/ui.html>) when changes are committed to the specified branch.
- TFE makes code review easier by automatically predicting (</docs/enterprise/run/ui.html#speculative-plans-on-pull-requests>) how pull requests will affect infrastructure.
- Publishing new versions of a private Terraform module (</docs/enterprise/registry/publish.html>) is as easy as pushing a tag to the module's repository.

We recommend configuring VCS access when first setting up a TFE organization, and you might need to add additional VCS providers later depending on how your organization grows.

Supported VCS Providers

TFE supports the following VCS providers:

- GitHub (</docs/enterprise/vcs/github.html>)
- GitHub Enterprise (</docs/enterprise/vcs/github-enterprise.html>)
- GitLab.com (</docs/enterprise/vcs/gitlab-com.html>)
- GitLab EE and CE (</docs/enterprise/vcs/gitlab-eece.html>)
- Bitbucket Cloud (</docs/enterprise/vcs/bitbucket-cloud.html>)
- Bitbucket Server (</docs/enterprise/vcs/bitbucket-server.html>)

Use the links above to see details on configuring VCS access for each supported provider. If you use another VCS that is not supported, you can build an integration via the API-driven run workflow (</docs/enterprise/run/api.html>).

How TFE Uses VCS Access

Most workspaces in TFE are associated with a VCS repository, which provides Terraform configurations for that workspace. To find out which repos are available, access their contents, and create webhooks, TFE needs access to your VCS provider.

Although TFE's API lets you create workspaces and push configurations to them without a VCS connection, the primary workflow expects every workspace to be backed by a repository.

To use configurations from VCS, TFE needs to do several things:

- Access a list of repositories, to let you search for repos when creating new workspaces.
- Register webhooks with your VCS provider, to get notified of new commits to a chosen branch.
- Download the contents of a repository at a specific commit in order to run Terraform with that code.

Webhooks

TFE uses webhooks to monitor new commits and pull requests.

- When someone adds new commits to a branch in a repository linked to TFE, any workspaces based on that branch will begin a Terraform run. Usually a user must inspect the plan output and approve an apply, but you can also enable automatic applies on a per-workspace basis. You can prevent automatic runs by locking a workspace.
- When someone submits a pull request/merge request to a branch from another branch in the same repository, TFE performs a speculative plan ([/docs/enterprise/run/index.html#speculative-plans](#)) with the contents of the request and links to the results on the PR's page. This helps you avoid merging PRs that cause plan failures.

SSH Keys

For most supported VCS providers, TFE does not need an SSH key — it can do everything it needs with the provider's API and an OAuth token. The exception is Bitbucket Server, which requires an SSH key for downloading repository contents. The setup instructions for Bitbucket Server ([/docs/enterprise/vcs/bitbucket-server.html](#)) include this step.

For other VCS providers, most organizations will not need to add an SSH private key. However, if the organization repositories include Git submodules that can only be accessed via SSH, an SSH key can be added along with the OAuth credentials.

If submodules will be cloned via SSH from a private VCS instance, SSH must be running on the standard port 22 on the VCS server.

To add an SSH key to a VCS connection, finish configuring OAuth in the organization settings, and then use the "add a private SSH key" link on the VCS Provider settings page to add a private key that has access to the submodule repositories. When setting up a workspace, if submodules are required, select "Include submodules on clone". More at [Workspace settings](#) ([/docs/enterprise/workspaces/settings.html](#)).

Multiple VCS Connections

If your infrastructure code is spread across multiple VCS providers, you can configure multiple VCS connections. TFE will ask which VCS connection to use whenever you create a new workspace.

Configuring VCS Access

TFE uses the OAuth protocol to authenticate with VCS providers.

Important: Even if you've used OAuth before, read the instructions carefully. Since TFE's security model treats each *organization* as a separate OAuth application, we authenticate with OAuth's developer workflow, which is more complex than the standard user workflow.

The exact steps to authenticate are different for each VCS provider, but they follow this general order:

On your VCS

Register your TFE organization as a new app. Get ID and key.

On TFE

Tell TFE how to reach VCS, and provide ID and key. Get callback URL.

Provide callback URL.

Request VCS access.

Approve access request.

For complete details, click the link for your VCS provider:

- GitHub (</docs/enterprise/vcs/github.html>)
- GitHub Enterprise (</docs/enterprise/vcs/github-enterprise.html>)
- GitLab.com (</docs/enterprise/vcs/gitlab-com.html>)
- GitLab EE and CE (</docs/enterprise/vcs/gitlab-eece.html>)
- Bitbucket Cloud (</docs/enterprise/vcs/bitbucket-cloud.html>)
- Bitbucket Server (</docs/enterprise/vcs/bitbucket-server.html>)

Note: Alternately, you can skip the OAuth configuration process and authenticate with a personal access token. This requires using TFE's API. For details, see the OAuth Clients API page (</docs/enterprise/api/oauth-clients.html>).

Configuring Bitbucket Cloud Access

These instructions are for using Bitbucket Cloud for Terraform Enterprise (TFE)'s VCS features. Bitbucket Cloud is the cloud-hosted version of Bitbucket; self-hosted Bitbucket Server instances have separate instructions, (/docs/enterprise/vcs/bitbucket-server.html) as do the other supported VCS providers. (/docs/enterprise/vcs/index.html)

Connecting TFE to your VCS involves five steps:

On your VCS	On TFE
Register your TFE organization as a new app. Get ID and key.	Tell TFE how to reach VCS, and provide ID and key. Get callback URL.
Provide callback URL.	Request VCS access.
Approve access request.	

The rest of this page explains the Bitbucket Cloud-specific versions of these steps.

Note: Alternately, you can skip the OAuth configuration process and authenticate with an app password. This requires using TFE's API. For details, see the OAuth Clients API page (/docs/enterprise/api/oauth-clients.html).

Step 1: On Bitbucket Cloud, Create a New OAuth Consumer

1. Open Bitbucket Cloud (<https://bitbucket.org>) in your browser and log in as whichever account you want TFE to act as. For most organizations this should be a dedicated service user, but a personal account will also work.

Important: The account you use for connecting TFE **must have admin access** to any shared repositories of Terraform configurations, since creating webhooks requires admin permissions.

2. Navigate to Bitbucket's "Add OAuth Consumer" page.

This page is located at https://bitbucket.org/account/user/<YOUR_USERNAME>/oauth-consumers/new. You can also reach it through Bitbucket's menus:

- In the lower left corner, click your profile picture and choose "Bitbucket settings."
- In the settings navigation, click "OAuth," which is in the "Access Management" section.
- On the OAuth settings page, click the "Add consumer" button.

3. This page has a form with four text fields and many checkboxes.

The screenshot shows the Terraform Enterprise Settings page with the 'OAuth' section selected. On the left, there's a sidebar with links like Snippets, Teams, and Settings. The main area is titled 'Add OAuth consumer' and contains fields for 'Details', 'Permissions', and a 'Save' button.

Details

- Name: Terraform Enterprise (example_corp)
- Description: BitBucket access for Terraform Enterprise.
- Callback URL: https://example.com/replace-later
- URL: https://app.terraform.io
- Privacy policy URL: (empty)
- End user license agreement URL: (empty)
- This is a private consumer
- Installable applications that ship their OAuth consumer credentials as part of the application should not be marked as private.

Permissions

- Account:** Email, Read, Write
- Team membership:** Read, Write
- Projects:** Read, Write
- Repositories:** Read, Write, Admin, Delete
- Pull requests:** Read, Write
- Issues:** Read, Write
- Wikis:** Read and write
- Snippets:** Read, Write
- Webhooks:** Read and write
- Pipelines:** Read, Write, Edit variables

Buttons: Save, Cancel

Fill out the text fields as follows:

Field	Value
Name	Terraform Enterprise (<YOUR ORGANIZATION NAME>)
Description	Any description of your choice.
Callback URL	https://example.com/replace-this-later (or any placeholder; the correct URI doesn't exist until the next step.)
URL	https://app.terraform.io (or the URL of your private TFE install)

Ensure that the "This is a private consumer" option is checked. Then, activate the following permissions checkboxes:

Permission type	Permission level
Account	Write
Repositories	Admin
Pull requests	Write
Webhooks	Read and write

4. Click the "Save" button, which returns you to the OAuth settings page.
5. Find your new OAuth consumer under the "OAuth Consumers" heading, and click its name to reveal its details.

Leave this page open in a browser tab. In the next step, you will copy and paste the unique **Key** and **Secret**.

The screenshot shows the Bitbucket OAuth consumers page. On the left, there's a sidebar with 'Access controls PREMIUM' at the top, followed by 'SECURITY', 'SSH keys', 'Two-step verification', 'Connected accounts', 'Sessions', and 'Audit log'. Below that is a 'INTEGRATIONS AND FEATURES' section with 'Labs'. The main area is titled 'OAuth consumers' with the sub-instruction 'Generate your own OAuth consumer key and secret to build your own custom integration with Bitbucket.' A blue 'Add consumer' button is visible. A table lists the consumer details:

Name	Description	...
Terraform En...	BitBucket access for Terraform Enterprise.	...
URL	https://app.terraform.io	
Key	Up7ss...	
Secret	HFhLUUpUB...	

Step 2: On TFE, Add a VCS Provider

1. Open TFE in your browser and navigate to the "VCS Provider" settings for your organization. Click the "Add VCS Provider" button.

If you just created your organization, you might already be on this page. Otherwise:

1. Click the upper-left organization menu, making sure it currently shows your organization.
 2. Click the "Settings" link at the top of the page (or within the \equiv menu)
 3. On the next page, click "VCS Provider" in the left sidebar.
 4. Click the "Add VCS Provider" button.
2. The next page has a drop-down and four text fields. Select "Bitbucket Cloud" from the drop-down, and enter the **Key** and **Secret** from the previous step. (Ignore the two disabled URL fields, which are used for on-premise VCSs.)

The screenshot shows the 'Add VCS Provider' interface. On the left, a sidebar lists organization settings like Profile, Teams, and VCS Provider (which is selected). The main form is titled 'Add VCS Provider' and contains fields for Bitbucket Cloud configuration: HTTP URL (https://bitbucket.org), API URL (https://api.bitbucket.org/2.0), Key (containing '<"KEY" FROM BITBUCKET>'), and Secret (containing '<"SECRET" FROM BITBUCKET>'). Buttons at the bottom are 'Create VCS Provider' and 'Cancel'.

3. Click "Create VCS Provider." This will take you back to the VCS Provider page, which now includes your new Bitbucket client.
4. Locate the new client's "Callback URL," and copy it to your clipboard; you'll paste it in the next step. Leave this page open in a browser tab.

The screenshot shows the VCS Provider list. A single entry for 'Bitbucket Cloud' is visible, displaying its details: Callback URL (https://app.terraform.io/auth/032370c8-[REDACTED]/callback), HTTP URL (https://bitbucket.org), API URL (https://api.bitbucket.org/2.0), and Created (Nov 13, 2017 16:50:39PM).

Step 3: On Bitbucket Cloud, Update the Callback URL

1. Go back to your Bitbucket Cloud browser tab. (If you accidentally closed it, you can reach your OAuth settings page through the menus: use the lower left menu > Bitbucket settings > OAuth.)
2. Locate your TFE OAuth consumer. Click the ellipsis ("...") button on the far right, and choose "Edit" from the menu.

The screenshot shows the Bitbucket Cloud OAuth consumers page. It lists a consumer named 'Terraform En...' with the description 'BitBucket access for Terraform Enterprise.' The consumer details are shown in a table:

Name	Description
URL	https://app.terraform.io
Key	Up7ss-[REDACTED]
Secret	HFhLUpU-[REDACTED]

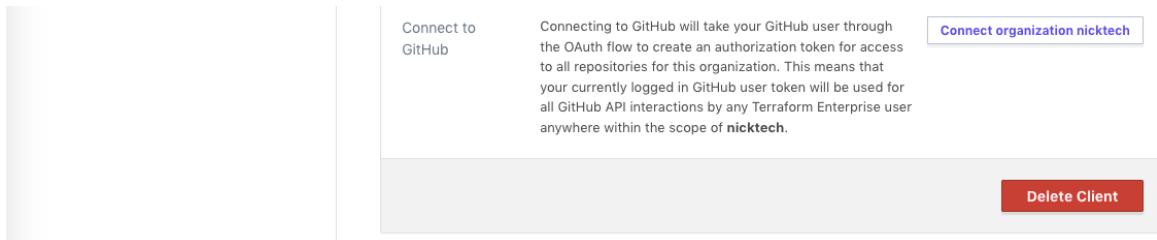
Buttons for 'Edit' and 'Delete' are visible on the right side of the consumer row.

3. In the "Callback URL" field, paste the callback URL from TFE's OAuth Configuration page, replacing the "example.com" placeholder you entered earlier.

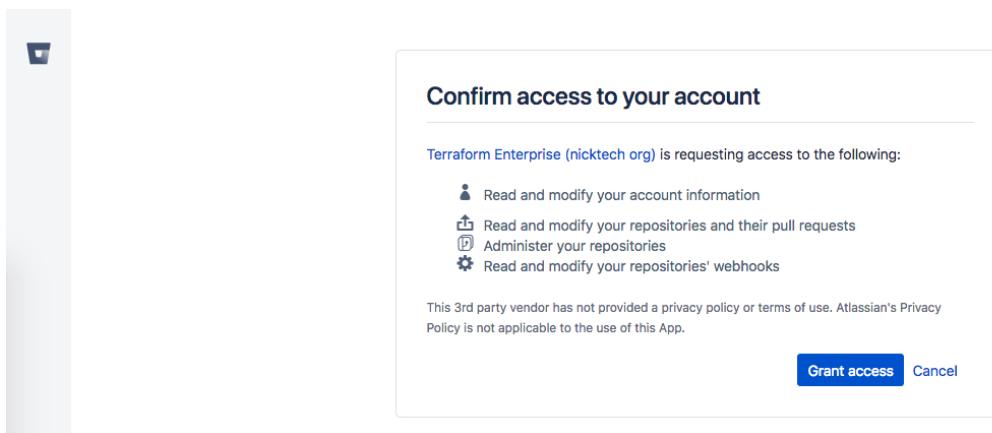
4. Click the "Save" button. A banner saying the update succeeded should appear.

Step 4: On TFE, Request Access

1. Go back to your TFE browser tab and click the "Connect organization <NAME>" button on the VCS Provider page.



This takes you to a page on Bitbucket Cloud, asking whether you want to authorize the app.



2. Click the green "Authorize" button at the bottom of the authorization page. This returns you to TFE's VCS Provider page, where the Bitbucket Cloud client's information has been updated.

Finished

At this point, Bitbucket Cloud access for TFE is fully configured, and you can create Terraform workspaces based on your organization's shared repositories.

Configuring Bitbucket Server Access

These instructions are for using Bitbucket Server for Terraform Enterprise (TFE)'s VCS features. Bitbucket Server is the on-premise version of Bitbucket; Bitbucket Cloud has separate instructions, (</docs/enterprise/vcs/bitbucket-cloud.html>) as do the other supported VCS providers. (</docs/enterprise/vcs/index.html>)

Note that Bitbucket Server requires both OAuth authentication and an SSH key. The instructions below include SSH key configuration.

Version note: TFE supports Bitbucket Server versions 4.9.1 and newer. HashiCorp does not test older versions of Bitbucket Server with TFE, and they might not work as expected. Also note that, although we do not deliberately remove support for versions that have reached end of life (per the Atlassian Support End of Life Policy (<https://confluence.atlassian.com/support/atlassian-support-end-of-life-policy-201851003.html>))), our ability to resolve customer issues with end of life versions might be limited.

Important: TFE needs to contact your Bitbucket Server instance during setup and during normal operation. For the SaaS version of TFE, this means Bitbucket Server must be internet-accessible; for private installs of TFE, you must have network connectivity between your TFE and Bitbucket Server instances over HTTP or HTTPS and SSH. Bitbucket Server repository clone operations are performed over SSH on the port the Bitbucket Server instance uses.

Note: Alternately, you can skip the OAuth configuration process and authenticate with a personal access token. This requires using TFE's API. For details, see the OAuth Clients API page (</docs/enterprise/api/oauth-clients.html>).

Step 1: On Bitbucket Server, Ensure the Webhooks Plugin is Installed

TFE uses webhooks to get new configurations. To support this, Bitbucket Server needs Atlassian's webhooks plugin.

1. Open your Bitbucket server instance in your browser and log in as an admin user.
2. Go to the "Manage add-ons" page. You can click the gear icon in the upper right corner and then use the "Manage add-ons" link in the sidebar, or go directly to `https://<BITBUCKET INSTANCE HOSTNAME>/plugins/servlet/upm`.
3. Look for an add-on named "Bitbucket Server Web Post Hooks Plugin," and make sure it is installed and enabled. The plugin is disabled by default. Clicking Enabled will toggle the plugin on.
4. If the plugin isn't present, click "Find new add-ons" in the sidebar navigation. Search for the plugin by name and install it.

Make sure to install the correct plugin. TFE is designed to work with the Bitbucket Server Web Post Hooks Plugin published by Atlassian Labs. (<https://marketplace.atlassian.com/plugins/com.atlassian.stash.plugin.stash-web-post-receive-hooks-plugin/server/overview>)



Bitbucket Server Web Post Hooks Plugin

Atlassian Labs • Unsupported

REPOSITORY HOOKS

★★★☆☆ (13)

3,095 installations

Free

Notify other systems or services of commits to your Bitbucket Server repository by adding a Post-Receive Webhook.

5. Visit the repository's settings, click on Hooks and check that the plugin is *enabled* there as well.

There is an option to configure a webhook URL on the plugin. Leave this optional field blank. Terraform Enterprise will dynamically update the webhook URL after the VCS connection is established.

Leave the page open in a browser tab, and remain logged in as an admin user.

Step 2: On TFE, Add a VCS Provider

1. Open TFE in your browser and navigate to the "VCS Provider" settings for your organization. Click the "Add VCS Provider" button.

If you just created your organization, you might already be on this page. Otherwise:

1. Click the upper-left organization menu, making sure it currently shows your organization.
 2. Click the "Settings" link at the top of the page (or within the \equiv menu)
 3. On the next page, click "VCS Provider" in the left sidebar.
 4. Click the "Add VCS Provider" button.
2. The next page has a drop-down and several text fields. Select "Bitbucket Server" from the drop-down. Several text fields will vanish, leaving only two. Enter the URL of your Bitbucket Server instance in both fields. The API URL should be the same as the main URL.

Note: If Bitbucket Server isn't accessible on the standard ports (for example, if it's using its default ports of 7990 or 8443 and is not behind a reverse proxy), make sure to specify the port in the URL. If you omit the port in the URL, TFE uses the standard port for the protocol (80 for HTTP, 443 for HTTPS).

The screenshot shows the 'Add VCS Provider' page for Bitbucket Server. On the left, there's a sidebar with 'ORGANIZATION SETTINGS' and a selected 'example_corp' organization. The 'VCS Provider' option is highlighted. The main form has a 'VERSION CONTROL SYSTEM (VCS) PROVIDER' dropdown set to 'Bitbucket Server'. Below it, the 'HTTP URL' is set to 'https://bitbucket-server.yourcompany.com' and the 'API URL' is also set to 'https://bitbucket-server.yourcompany.com'. A note says 'Add a new OAuth consumer on Bitbucket Server installation to complete the process.' At the bottom are 'Create VCS Provider' and 'Cancel' buttons.

3. Click "Create VCS Provider." This will take you back to the VCS Provider page, which now includes your new Bitbucket Server client.
4. Leave this page open in a browser tab. In the next step, you will copy and paste the unique **Consumer Key** and **Public Key**.

The screenshot shows the 'VCS Provider' page with the newly added Bitbucket Server client. The client details are:

- Consumer Key:** eflad1 [REDACTED] [Copy](#)
- Consumer Name:** Terraform Enterprise [Copy](#)
- Public Key:** -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEAAQCAQ8AMIIIBCgKCAQEA5dOD2hUD8+nMDix/qWAd
72dn0YogB6tlN4u/WJwn0kp5jjCann9ad2bk27qWa5Pf0SB+RPf1tbucYkWBEGd
[REDACTED]
-----END PUBLIC KEY----- [Copy](#)
- HTTP URL:** http://bitbucket-server.[REDACTED]
- API URL:** http://bitbucket-server.[REDACTED]
- Created:** May 28, 2018 22:41:18PM
- Connect to Bitbucket Server:** Connecting to Bitbucket Server will take your Bitbucket Server user through the OAuth flow to create an authorization token for access to all repositories for this organization. This means that your currently logged in Bitbucket Server user token will be used for all Bitbucket Server API interactions by any Terraform Enterprise user anywhere within the scope of `example_corp`. [Connect organization example_corp](#)

At the bottom right is a 'Delete Client' button.

Step 3: On Bitbucket Server, Create a New Application Link

1. While logged in as an admin user, go to Bitbucket Server's "Application Links" administration page. You can use the sidebar navigation in the admin pages, or go directly to `https://<BITBUCKET INSTANCE HOSTNAME>/plugins/servlet/applinks/listApplicationLinks`.

This page has a text field for creating a new application link, followed by a list of existing application links.

The screenshot shows the Bitbucket Administration interface under the 'Application Links' section. On the left, a sidebar lists various administration categories like Accounts, Settings, and Application Navigator. The main area is titled 'Configure Application Links'. A text input field contains 'https://app.terraform.io/app/nicktech'. To its right is a blue 'Create new link' button. Below this, a table lists seven application links:

Application	Version	Status	Actions
Jessica's TFE Zone (PRIMARY)	Generic Application	NON-ATLASSIAN	
Terraform Enterprise	Generic Application	NON-ATLASSIAN	
Jessica's Dev Env	Generic Application	NON-ATLASSIAN	
atlas.hashicorp.com	Generic Application	NON-ATLASSIAN	
atlas.hashicorp.com/app/jessica-v2-bitbucket-server	Generic Application	NON-ATLASSIAN	
Terraform Enterprise (nicktech org)	Generic Application	NON-ATLASSIAN	
Phinze's TFE Zone	Generic Application	NON-ATLASSIAN	

2. Enter TFE's URL in the text field (`https://app.terraform.io`, or the hostname of your private TFE instance) and click the "Create new link" button.

Note: If you're connecting multiple TFE organizations to the same Bitbucket Server instance, you can only use TFE's main URL once. For subsequent organizations, you can enter the organization URL instead. Organization URLs look like `https://app.terraform.io/app/<ORG NAME>` or `https://<TFE HOSTNAME>/app/<ORG NAME>` — it's the page TFE's "Workspaces" button takes you to.

3. In the "Configure application URL" dialog, confirm that you wish to use the URL exactly as you entered it. If you used TFE's main URL, click "Continue;" if you used an organization URL, click the "Use this URL" checkbox and then click "Continue."

The screenshot shows the 'Configure Application URL' dialog box overlaid on the main configuration page. The dialog contains a warning message: 'No response was received from the URL you entered - it may not be valid. Please fix the URL below, if needed, and click Continue.' Below the message are two input fields: 'Entered URL' containing 'https://app.terraform.io' and 'New URL*' containing 'https://app.terraform.io'. At the bottom of the dialog are 'Continue' and 'Cancel' buttons. The background table of application links is partially visible.

4. In the "Link applications" dialog, fill out the form fields as follows:

Field	Value
Application Name (text)	Terraform Enterprise (<ORG NAME>)
Application Type (drop-down)	Generic Application
Create incoming link (checkbox)	(enabled)

Leave all the other fields blank, and click "Continue."

5. This takes you to another dialog, also titled "Link applications," with three text fields. In the "Consumer Key" and "Public Key" fields, copy and paste the values from step 2. In the "Consumer Name" field, enter "Terraform Enterprise (<ORG NAME>)". Click "Continue."

The screenshot shows the Bitbucket Administration interface. On the left, there's a sidebar with various settings like Accounts, Users, Groups, and Application Links. The 'Application Links' section is currently selected. A modal window titled 'Configure Application Links' is open, specifically the 'Link applications' tab. It shows a list of applications on the left and configuration fields on the right. The application being linked is 'Bitbucket' (Name: Bitbucket, Application: Bitbucket Server). The target application is 'Terraform Enterprise (nicktech org)' (Consumer Name: Terraform Enterprise (nicktech org)). The consumer key is '0265341c1a314' and the public key is '-----BEGIN PUBLIC KEY-----'. There are 'Continue' and 'Cancel' buttons at the bottom of the modal.

Step 4: On Workstation: Create an SSH Key for TFE

On a secure workstation, create an SSH keypair that TFE can use to connect to Bitbucket Server. The exact command depends on your OS, but is usually something like `ssh-keygen -t rsa -f "/Users/<NAME>/.ssh/service_tfe" -C "service_terraform_enterprise"`. This creates a `service_tfe` file with the private key, and a `service_tfe.pub` file with the public key.

This SSH key **must have an empty passphrase**. TFE cannot use SSH keys that require a passphrase.

Important Notes

- Do not use your personal SSH key to connect TFE and Bitbucket Server; generate a new one or use an existing key reserved for service access.
- In the following steps, you must provide TFE with the private key. Although TFE does not display the text of the key to users after it is entered, it retains it and will use it for authenticating to Bitbucket Server.
- **Protect this private key carefully.** It can push code to the repositories you use to manage your infrastructure. Take note of your organization's policies for protecting important credentials and be sure to follow them.

Step 5: On Bitbucket Server, Switch Users and Add an SSH Key

1. If you are still logged in to Bitbucket Server as an administrator, log out now.
2. Log in as whichever account you want TFE to act as. For most organizations this should be a dedicated service user, but a personal account will also work.

Important: The account you use for connecting TFE **must have admin access** to any shared repositories of Terraform configurations, since creating webhooks requires admin permissions.

3. Go to the "SSH keys" page. You can click the profile icon in the upper right corner, choose "Manage account," then click "SSH keys" in the sidebar navigation, or you can go directly to <https://<BITBUCKET INSTANCE HOSTNAME>/plugins/servlet/ssh/account/keys>.

The screenshot shows the Bitbucket account settings page. On the left, there's a sidebar with links: Account settings, Change password, SSH keys (which is selected and highlighted in blue), and Authorized applications. The main content area is titled "SSH keys" and has a sub-header "Add key". It says "Use SSH keys to connect simply and safely to repositories". Below this is a table with two rows. The first row has a "Label" column containing "nick@MacBook-Pro" and a "Key" column containing "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDdY0VD785S8...". The second row has a "Label" column containing "service_terraform_enterprise..." and a "Key" column containing "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDUkvri2ZEM...". At the bottom of the table, it says "Bitbucket's RSA fingerprint: 09:14:cc:81:a9:68:84:b6:30:2d:5e:1b:c0:d6:81:ce".

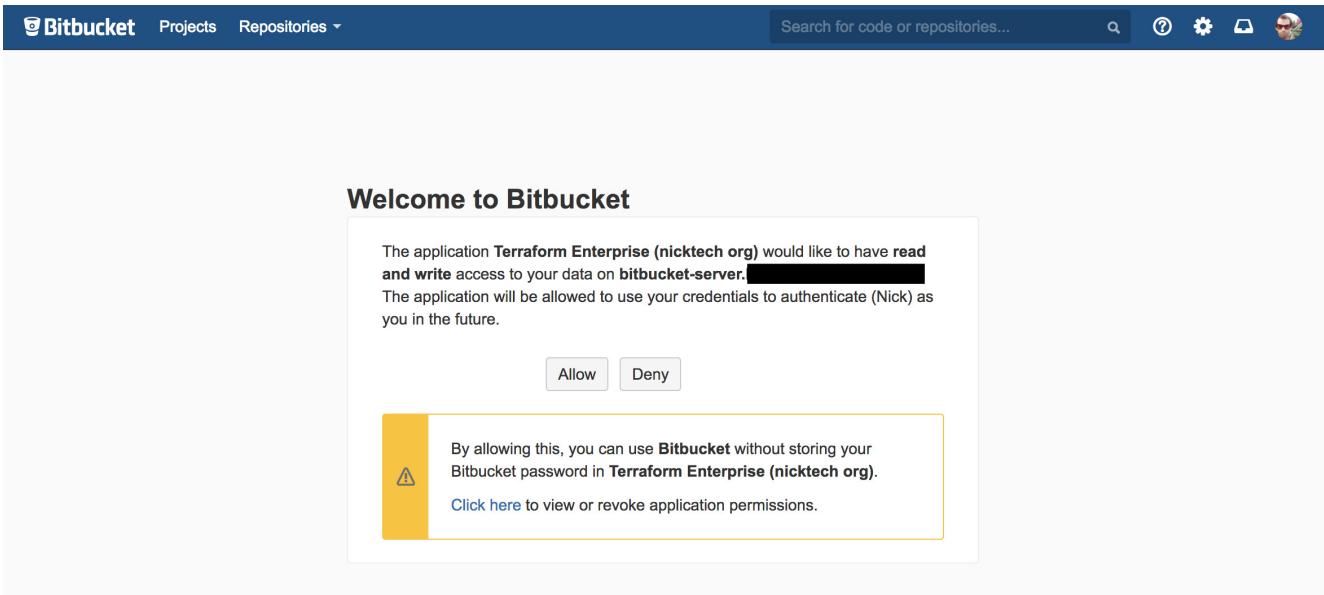
4. Click the "Add key" button. Paste the text of the **SSH public key** you created in step 4 (from the .pub file) into the text field, then click the "Add key" button to confirm.

Step 6: On TFE, Request Access and Add an SSH Private Key

1. Go back to your TFE browser tab and click the "Connect organization <NAME>" button on the VCS Provider page.

The screenshot shows the "VCS Providers" section of the Terraform Enterprise configuration interface. It lists GitHub as a provider. To the right of GitHub, there are three buttons: "Connect organization nicktech" (highlighted in blue), "Delete Client" (in red), and "Edit Client" (disabled).

This takes you to a page on your Bitbucket Server instance, asking if you want to authorize TFE. Double-check that you're logged in as the user account TFE will be using, and not as a Bitbucket administrator.



2. Click the "Allow" button. This returns you to TFE's VCS Provider page, where the Bitbucket Server client's information has been updated.

If this results in a 500 error, it usually means TFE was unable to reach your Bitbucket Server instance.

3. Click the "Add a private SSH key" link. A large text field will appear. Paste the text of the **SSH private key** you created in step 4, and click the "Add SSH Key" button.

A screenshot of the Terraform Enterprise (TFE) VCS Provider page. It shows a connection to 'Bitbucket Server' with the following details:

- Consumer Key: eflad13 [REDACTED] [Edit](#)
- Consumer Name: Terraform Enterprise [Edit](#)
- HTTP URL: http://bitbucket-server. [REDACTED]
- API URL: http://bitbucket-server. [REDACTED]
- Created: May 28, 2018 22:41:18PM

The 'Connection' section notes a recent connection made on May 28, 2018, and includes a 'Revoke Connection' button. A callout box provides instructions: "Heads up: This VCS connection requires a private SSH key to enable git clone operations. For more information, please see the Terraform Enterprise documentation on [Adding a private SSH key to Bitbucket Server OAuth connection](#)".
Below this, there is a 'PRIVATE SSH KEY' input field containing the text "-----BEGIN RSA PRIVATE KEY-----" followed by a large blacked-out area. An 'Add SSH Key' button is located below the input field. In the bottom right corner of the main panel, there is a 'Delete Client' button.

Finished

At this point, Bitbucket Server access for TFE is fully configured, and you can create Terraform workspaces based on your organization's shared repositories.

Configuring GitHub Enterprise Access

These instructions are for using an on-premise installation of GitHub Enterprise for TFE's VCS features. GitHub.com has separate instructions, ([/docs/enterprise/vcs/github-enterprise.html](#)) as do the other supported VCS providers. ([/docs/enterprise/vcs/index.html](#))

Connecting TFE to your VCS involves five steps:

On your VCS	On TFE
Register your TFE organization as a new app. Get ID and key.	Tell TFE how to reach VCS, and provide ID and key. Get callback URL.
Provide callback URL.	Request VCS access.
Approve access request.	

The rest of this page explains the GitHub Enterprise versions of these steps.

Important: TFE needs to contact your GitHub Enterprise instance during setup and during normal operation. For the SaaS version of TFE, this means GitHub Enterprise must be internet-accessible; for private installs of TFE, you must have network connectivity between your TFE and GitHub Enterprise instances.

Note: Alternately, you can skip the OAuth configuration process and authenticate with a personal access token. This requires using TFE's API. For details, see the OAuth Clients API page ([/docs/enterprise/api/oauth-clients.html](#)).

Step 1: On GitHub, Create a New OAuth Application

1. Open your GitHub Enterprise instance in your browser and log in as whichever account you want TFE to act as. For most organizations this should be a dedicated service user, but a personal account will also work.

Important: The account you use for connecting TFE **must have admin access** to any shared repositories or Terraform configurations, since creating webhooks requires admin permissions.

2. Navigate to GitHub's Register a New OAuth Application page.

This page is located at `https://<GITHUB INSTANCE HOSTNAME>/settings/applications/new`. You can also reach it through GitHub's menus:

- In the upper right corner, click your profile picture and choose "Settings."
- In the navigation sidebar, click "OAuth Apps" (under the "Developer settings" section).
- In the upper right corner, click the "Register a new application" button.

3. This page has a form with four text fields.

The screenshot shows the GitHub OAuth application registration interface. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the navigation, the title 'Register a new OAuth application' is displayed. The form contains several input fields: 'Application name' (with placeholder 'Something users will recognize and trust'), 'Homepage URL' (with placeholder 'The full URL to your application homepage'), 'Application description' (with placeholder 'Application description is optional' and note 'This is displayed to all users of your application'), and 'Authorization callback URL' (with placeholder 'Your application's callback URL. Read our [OAuth documentation](#) for more information.'). At the bottom right of the form are two buttons: 'Register application' (in green) and 'Cancel'.

Fill them in as follows:

Field name	Value
Application Name	Terraform Enterprise (<YOUR ORGANIZATION NAME>)
Homepage URL	https://app.terraform.io (or the URL of your private TFE install)
Application Description	Any description of your choice.
Authorization callback URL	https://example.com/replace-this-later (or any placeholder; the correct URI doesn't exist until the next step.)

4. Click the "Register application" button, which creates the application and takes you to its page.
5. Download this image of the Terraform logo (`./images/tfe_logo.png`), upload it with the "Upload new logo" button or the drag-and-drop target, and set the badge background color to `#5C4EE5`. This optional step helps you identify TFE's pull request checks at a glance.
6. Leave this page open in a browser tab. In the next step, you will copy and paste the unique **Client ID** and **Client Secret**.

The screenshot shows the GitHub Developer settings page for the organization 'nicktech org'. On the left, there's a sidebar with options: OAuth Apps (selected), GitHub Apps, and Personal access tokens. The main content area is titled 'Terraform Enterprise (nicktech org)' and shows that 'nfagerlund' owns the application. It includes a button to 'Transfer ownership' and a link to 'List this application in the Marketplace'. Below this, there's a section for users, showing '1 user' with a client ID (642fee7ac...) and a client secret (237bf7c8c...). Buttons for 'Revoke all user tokens' and 'Reset client secret' are present. Further down, there's a section for the 'Application logo' with a placeholder image and a 'Upload new logo' button. A note says you can also drag and drop a picture from your computer. Finally, there's a 'Badge background color' section with a hex value '#5C4EE5' and a preview circle containing the Terraform logo.

Step 2: On TFE, Add a VCS Provider

1. Open TFE in your browser and navigate to the "VCS Provider" settings for your organization. Click the "Add VCS Provider" button.

If you just created your organization, you might already be on this page. Otherwise:

1. Click the upper-left organization menu, making sure it currently shows your organization.
 2. Click the "Settings" link at the top of the page (or within the \equiv menu)
 3. On the next page, click "VCS Provider" in the left sidebar.
 4. Click the "Add VCS Provider" button.
2. The next page has a drop-down and four text fields. Select "GitHub Enterprise" from the drop-down, and fill in all four text fields as follows:

Field	Value
HTTP URL	<code>https://<GITHUB INSTANCE HOSTNAME></code>
API URL	<code>https://<GITHUB INSTANCE HOSTNAME>/api/v3</code>
Client ID	(paste value from previous step)
Client Secret	(paste value from previous step)

The screenshot shows the 'Add VCS Provider' page for GitHub Enterprise. On the left, there's a sidebar with 'Organization Settings' for 'nicktech'. The 'VCS Provider' option is selected. The main area has a heading 'Add VCS Provider' and instructions about connecting workspaces to git repositories. It includes fields for 'HTTP URL' (set to 'https://github.example.com') and 'API URL' (set to 'https://github.example.com/api/v3'). Below these are fields for 'CLIENT ID' (placeholder '<CLIENT ID FROM GITHUB ENTERPRISE>') and 'CLIENT SECRET' (placeholder '<CLIENT SECRET FROM GITHUB ENTERPRISE>'), each with a note about generating them via OAuth application registration. At the bottom are 'Create VCS Provider' and 'Cancel' buttons.

3. Click "Create VCS Provider." This will take you back to the VCS Provider page, which now includes your new GitHub Enterprise client.
4. Locate the new client's "Callback URL," and copy it to your clipboard; you'll paste it in the next step. Leave this page open in a browser tab.

The screenshot shows the 'VCS Providers' page. A new client for GitHub is listed, showing its details: 'Callback URL' (https://app.terraform.io/auth/ed0aa76c-a.../callback), 'HTTP URL' (https://github.com), 'API URL' (https://api.github.com), and 'Created' (Sep 15, 2017 14:51:20PM). The 'Connection' section notes a recent connection made by user 'nfagerlund'. At the bottom are 'Revoke Connection' and 'Delete Client' buttons.

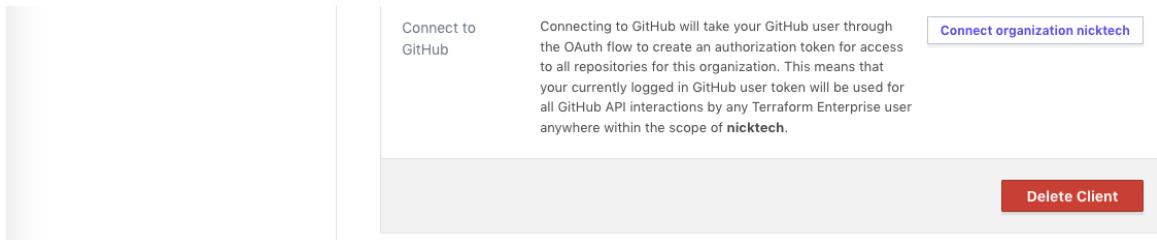
Step 3: On GitHub, Update the Callback URL

1. Go back to your GitHub browser tab. (If you accidentally closed it, you can reach your OAuth app page through the menus: use the upper right menu > Settings > Developer settings > OAuth Apps > "Terraform Enterprise (<YOUR ORG NAME>)".)
2. In the "Authorization Callback URL" field, near the bottom of the page, paste the callback URL from TFE's OAuth Configuration page, replacing the "example.com" placeholder you entered earlier.

3. Click the "Update application" button. A banner saying the update succeeded should appear at the top of the page.

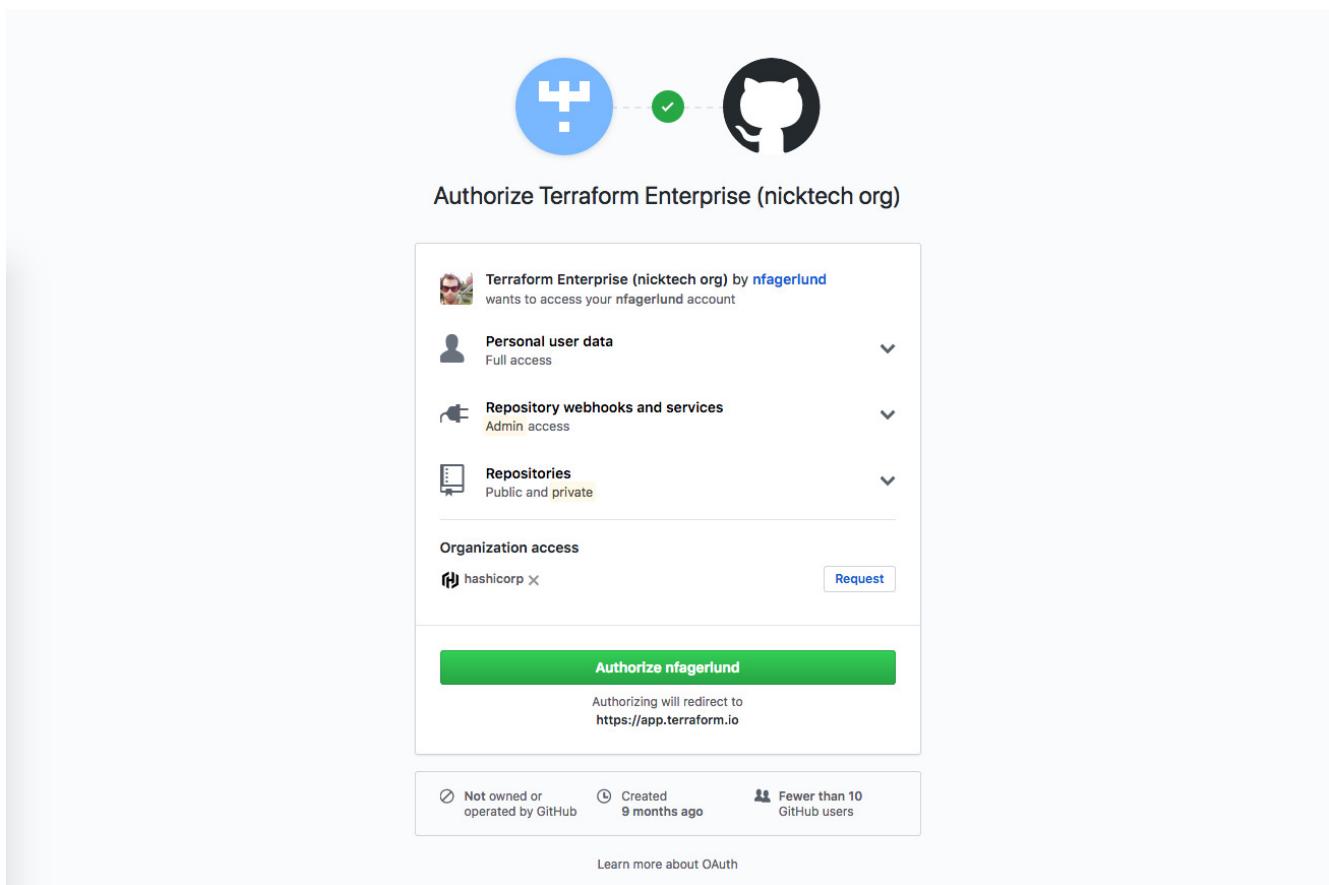
Step 4: On TFE, Request Access

1. Go back to your TFE browser tab and click the "Connect organization <NAME>" button on the VCS Provider page.



This takes you to a page on github.com, asking whether you want to authorize the app.

2. The authorization page lists any GitHub organizations this account belongs to. If there is a "Request" button next to the organization that owns your Terraform code repositories, click it now. Note that you need to do this even if you are only connecting workspaces to private forks of repositories in those organizations since those forks are subject to the organization's access restrictions. See About OAuth App access restrictions (<https://help.github.com/articles/about-oauth-app-access-restrictions>).



3. Click the green "Authorize <GITHUB USER>" button at the bottom of the authorization page. GitHub might request your password to confirm the operation.

This returns you to TFE's OAuth Configuration page. If it results in a 500 error, it usually means TFE was unable to reach your GitHub Enterprise instance.

Step 5: Contact Your GitHub Organization Admins

If your organization uses OAuth app access restrictions, you had to click a "Request" button when authorizing TFE, which sent an automated email to the administrators of your GitHub organization. An administrator must approve the request before TFE can access your organization's shared repositories.

If you're a GitHub administrator, check your email now and respond to the request; otherwise, contact whoever is responsible for GitHub accounts in your organization, and wait for confirmation that they've approved your request.

Finished

At this point, GitHub access for TFE is fully configured, and you can create Terraform workspaces based on your organization's shared GitHub Enterprise repositories.

Configuring GitHub Access

These instructions are for using GitHub.com for TFE's VCS features. GitHub Enterprise has separate instructions, (/docs/enterprise/vcs/github-enterprise.html) as do the other supported VCS providers. (/docs/enterprise/vcs/index.html)

Connecting TFE to your VCS involves five steps:

On your VCS	On TFE
Register your TFE organization as a new app. Get ID and key.	Tell TFE how to reach VCS, and provide ID and key. Get callback URL.
Provide callback URL.	Request VCS access.
Approve access request.	

The rest of this page explains the GitHub versions of these steps.

Note: Alternately, you can skip the OAuth configuration process and authenticate with a personal access token. This requires using TFE's API. For details, see the OAuth Clients API page (/docs/enterprise/api/oauth-clients.html).

Step 1: On GitHub, Create a New OAuth Application

1. Open github.com (<https://github.com>) in your browser and log in as whichever account you want TFE to act as. For most organizations this should be a dedicated service user, but a personal account will also work.

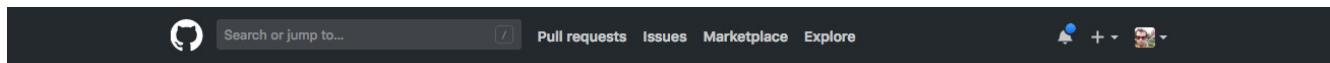
Important: The account you use for connecting TFE **must have admin access** to any shared repositories or Terraform configurations, since creating webhooks requires admin permissions.

2. Navigate to GitHub's Register a New OAuth Application (<https://github.com/settings/applications/new>) page.

This page is located at <https://github.com/settings/applications/new> (<https://github.com/settings/applications/new>). You can also reach it through GitHub's menus:

- In the upper right corner, click your profile picture and choose "Settings."
- In the navigation sidebar, click "Developer settings," then make sure you're on the "OAuth Apps" page (not "GitHub Apps").
- In the upper right corner, click the "New OAuth App" button.

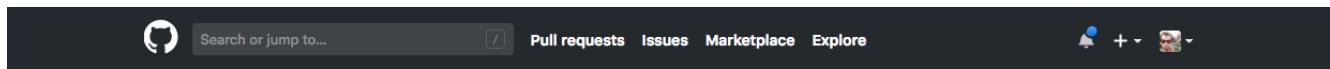
3. This page has a form with four text fields.



Fill them in as follows:

Field name	Value
Application Name	Terraform Enterprise (<YOUR ORGANIZATION NAME>)
Homepage URL	https://app.terraform.io (or the URL of your private TFE install)
Application Description	Any description of your choice.
Authorization callback URL	https://example.com/replace-this-later (or any placeholder; the correct URI doesn't exist until the next step.)

4. Click the "Register application" button, which creates the application and takes you to its page.
5. Download this image of the Terraform logo (`./images/tfe_logo.png`), upload it with the "Upload new logo" button or the drag-and-drop target, and set the badge background color to `#5C4EE5`. This optional step helps you identify TFE's pull request checks at a glance.
6. Leave this page open in a browser tab. In the next step, you will copy and paste the unique **Client ID** and **Client Secret**.



Settings / Developer settings

OAuth Apps
GitHub Apps
Personal access tokens

Terraform Enterprise (nicktech org)

 nfagerlund owns this application.

[Transfer ownership](#)

You can list your application in the [GitHub Marketplace](#) so that other users can discover it.

[List this application in the Marketplace](#)

1 user

Client ID
642fee7ac [REDACTED]

Client Secret
237bf7c8c [REDACTED]

[Revoke all user tokens](#) [Reset client secret](#)

Application logo



[Upload new logo](#)

You can also drag and drop a picture from your computer.

Badge background color

#5C4EE5

The hex value of the badge background color.



Step 2: On TFE, Add a VCS Provider

1. Open TFE in your browser and navigate to the "VCS Provider" settings for your organization. Click the "Add VCS Provider" button.

If you just created your organization, you might already be on this page. Otherwise:

1. Click the upper-left organization menu, making sure it currently shows your organization.
 2. Click the "Settings" link at the top of the page (or within the \equiv menu)
 3. On the next page, click "VCS Provider" in the left sidebar.
 4. Click the "Add VCS Provider" button.
2. The next page has a drop-down and four text fields. Select "GitHub.com" from the drop-down, and enter the **Client ID** and **Client Secret** from the previous step. (Ignore the two disabled URL fields, which are used for on-premise VCSs.)

The screenshot shows the Terraform Enterprise interface. At the top, there's a navigation bar with a logo, the user name 'nicktech', and links for 'Workspaces' and 'Modules'. On the right, there are 'Documentation' and 'Status' links. Below the navigation, on the left, is a sidebar titled 'ORGANIZATION SETTINGS' under the 'nicktech' organization. The 'VCS Provider' option is selected and highlighted in blue. The main content area is titled 'Add VCS Provider' and is for GitHub. It includes fields for 'HTTP URL' (set to 'https://github.com'), 'API URL' (set to 'https://api.github.com'), 'CLIENT ID' (placeholder: '<"CLIENT ID" FROM GITHUB>'), and 'CLIENT SECRET' (placeholder: '<"CLIENT SECRET" FROM GITHUB>'). Below these fields are explanatory notes: 'Register a new OAuth application on GitHub.com to generate the Client ID and Client Secret.' and 'Register a new OAuth application on GitHub.com to generate the Client ID and Client Secret.'. At the bottom are 'Create VCS Provider' and 'Cancel' buttons.

3. Click "Create VCS Provider." This will take you back to the VCS Provider page, which now includes your new GitHub client.
4. Locate the new client's "Callback URL," and copy it to your clipboard; you'll paste it in the next step. Leave this page open in a browser tab.

The screenshot shows the GitHub OAuth client configuration page. It lists the following details for the client:

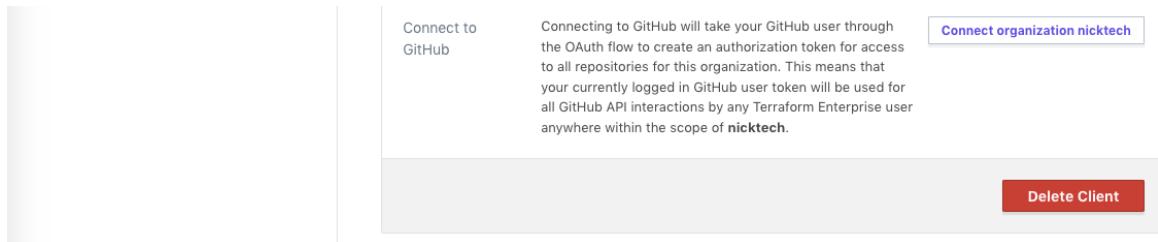
- Callback URL: <https://app.terraform.io/auth/ed0aa76c-a.../callback>
- HTTP URL: https://github.com
- API URL: https://api.github.com
- Created: Sep 15, 2017 14:51:20PM
- Connection: A connection was made on Sep 15, 2017 16:26:55PM by authenticating via OAuth as GitHub user **nfagerlund**, which assigned an OAuth token for use by all Terraform Enterprise users in the **nicktech** organization. This section includes a 'Revoke Connection' button.
- At the bottom right is a 'Delete Client' button.

Step 3: On GitHub, Update the Callback URL

1. Go back to your GitHub browser tab. (If you accidentally closed it, you can reach your OAuth app page through the menus: use the upper right menu > Settings > Developer settings > OAuth Apps > "Terraform Enterprise (<YOUR ORG NAME>)".)
2. In the "Authorization Callback URL" field, near the bottom of the page, paste the callback URL from TFE's OAuth Configuration page, replacing the "example.com" placeholder you entered earlier.
3. Click the "Update application" button. A banner saying the update succeeded should appear at the top of the page.

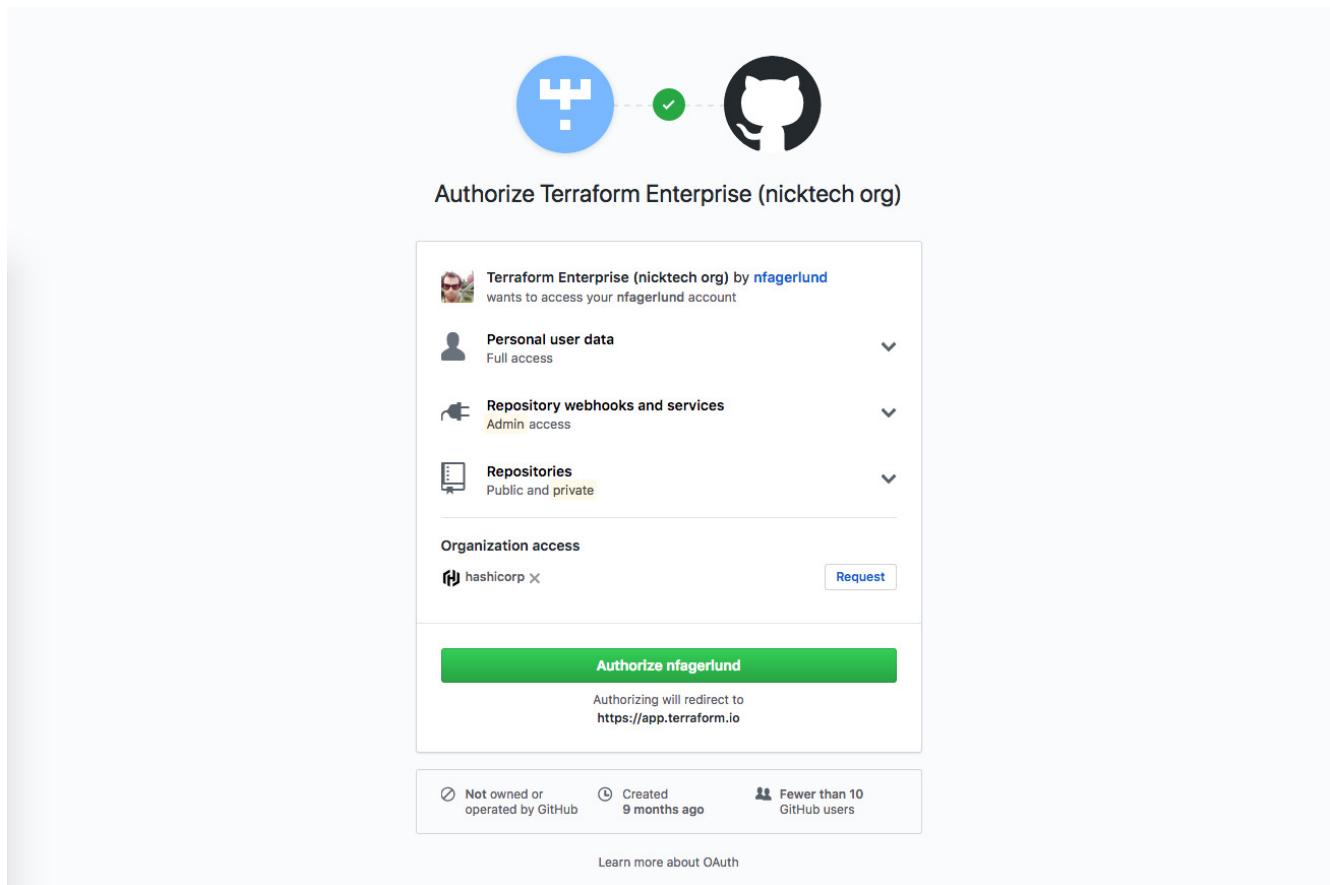
Step 4: On TFE, Request Access

1. Go back to your TFE browser tab and click the "Connect organization <NAME>" button on the VCS Provider page.



This takes you to a page on github.com, asking whether you want to authorize the app.

2. The authorization page lists any GitHub organizations this account belongs to. If there is a "Request" button next to the organization that owns your Terraform code repositories, click it now. Note that you need to do this even if you are only connecting workspaces to private forks of repositories in those organizations since those forks are subject to the organization's access restrictions. See About OAuth App access restrictions (<https://help.github.com/articles/about-oauth-app-access-restrictions>).



3. Click the green "Authorize <GITHUB_USER>" button at the bottom of the authorization page. GitHub might request your password to confirm the operation.

Step 5: Contact Your GitHub Organization Admins

If your organization uses OAuth app access restrictions, you had to click a "Request" button when authorizing TFE, which sent an automated email to the administrators of your GitHub organization. An administrator must approve the request before TFE can access your organization's shared repositories.

If you're a GitHub administrator, check your email now and respond to the request; otherwise, contact whoever is responsible for GitHub accounts in your organization, and wait for confirmation that they've approved your request.

Finished

At this point, GitHub access for TFE is fully configured, and you can create Terraform workspaces based on your organization's shared GitHub repositories.

Configuring GitLab.com Access

These instructions are for using GitLab.com for Terraform Enterprise (TFE)'s VCS features. GitLab CE and GitLab EE have separate instructions, (/docs/enterprise/vcs/gitlab-eece.html) as do the other supported VCS providers. (/docs/enterprise/vcs/index.html)

Connecting TFE to your VCS involves five steps:

On your VCS	On TFE
Register your TFE organization as a new app. Get ID and key.	Tell TFE how to reach VCS, and provide ID and key. Get callback URL.
Provide callback URL.	Request VCS access.
Approve access request.	

The rest of this page explains the GitLab.com versions of these steps.

Note: Alternately, you can skip the OAuth configuration process and authenticate with a personal access token. This requires using TFE's API. For details, see the OAuth Clients API page (/docs/enterprise/api/oauth-clients.html).

Step 1: On GitLab, Create a New Application

1. Open gitlab.com (<https://gitlab.com>) in your browser and log in as whichever account you want TFE to act as. For most organizations this should be a dedicated service user, but a personal account will also work.

Important: The account you use for connecting TFE **must have admin (master) access** to any shared repositories of Terraform configurations, since creating webhooks requires admin permissions.

2. Navigate to GitLab's User Settings > Applications (<https://gitlab.com/profile/applications>) page.

This page is located at <https://gitlab.com/profile/applications> (<https://gitlab.com/profile/applications>). You can also reach it through GitLab's menus:

- In the upper right corner, click your profile picture and choose "Settings."
- In the navigation sidebar, click "Applications."

3. This page has a list of applications and a form for adding new ones. The form has two text fields and some checkboxes.

The screenshot shows the 'User Settings' section of the GitLab interface, specifically the 'Applications' tab. On the left sidebar, under 'Applications', the 'Add new application' option is selected. In the main area, the 'Name' field contains 'Terraform Enterprise (nicktech org)'. The 'Redirect URI' field contains 'https://example.com/replace-later'. Under 'Scopes', several checkboxes are listed: 'api', 'read_user', 'sudo', 'read_repository', 'read_registry', and 'openid'. A note below 'openid' states: 'The ability to authenticate using GitLab, and read-only access to the user's profile information and group memberships'. At the bottom right of the form is a green 'Save application' button.

Fill out the form as follows:

Field	Value
(all checkboxes)	(empty)
Name	Terraform Enterprise (<YOUR ORGANIZATION NAME>)
Redirect URI	https://example.com/replace-this-later (or any placeholder; the correct URI doesn't exist until the next step.)

- Click the "Save application" button, which creates the application and takes you to its page.
- Leave this page open in a browser tab. In the next step, you will copy and paste the unique **Application ID** and **Secret**.

The screenshot shows the 'User Settings > Applications' page for the 'Terraform Enterprise (nicktech org)' application. A blue banner at the top states 'The application was created successfully.' Below it, the application details are listed: 'Application: Terraform Enterprise (nicktech org)', 'Application Id: 79195593e40b3', 'Secret: 6c38811d69a36', and 'Callback url: https://example.com/replace-later'. At the bottom of the page are two buttons: a blue 'Edit' button and a red 'Destroy' button.

Step 2: On TFE, Add a VCS Provider

- Open TFE in your browser and navigate to the "VCS Provider" settings for your organization. Click the "Add VCS

Provider" button.

If you just created your organization, you might already be on this page. Otherwise:

1. Click the upper-left organization menu, making sure it currently shows your organization.
 2. Click the "Settings" link at the top of the page (or within the \equiv menu)
 3. On the next page, click "VCS Provider" in the left sidebar.
 4. Click the "Add VCS Provider" button.
2. The next page has a drop-down and four text fields. Select "GitLab.com" from the drop-down, and enter the **Application ID** and **Secret** from the previous step. (Ignore the two disabled URL fields, which are used for on-premise VCSs.)

ORGANIZATION SETTINGS

nicktech

Profile

Teams

VCS Provider

API Token

Authentication

Manage SSH Keys

Sentinel Policy

Add VCS Provider

To connect workspaces to git repositories containing Terraform configurations, Terraform Enterprise needs access to your version control system (VCS) provider. Use this page to configure OAuth authentication with your VCS provider. For more information, please see the Terraform Enterprise documentation on [Configuring Version Control Access](#).

VERSION CONTROL SYSTEM (VCS) PROVIDER

GitLab.com

HTTP URL

https://gitlab.com

API URL

https://gitlab.com/api/v3

APPLICATION ID

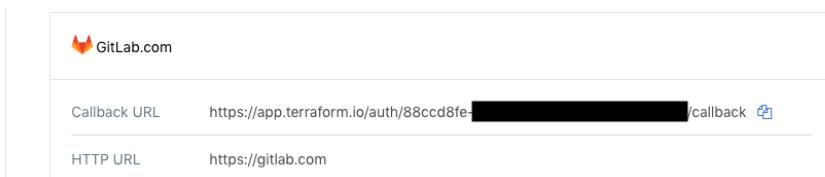
Add a new application on GitLab.com to generate the Application Id and Secret.

SECRET

Add a new application on GitLab.com to generate the Application Id and Secret.

Create VCS Provider Cancel

3. Click "Create connection." This will take you back to the VCS Provider page, which now includes your new GitLab client.
4. Locate the new client's "Callback URL," and copy it to your clipboard; you'll paste it in the next step. Leave this page open in a browser tab.



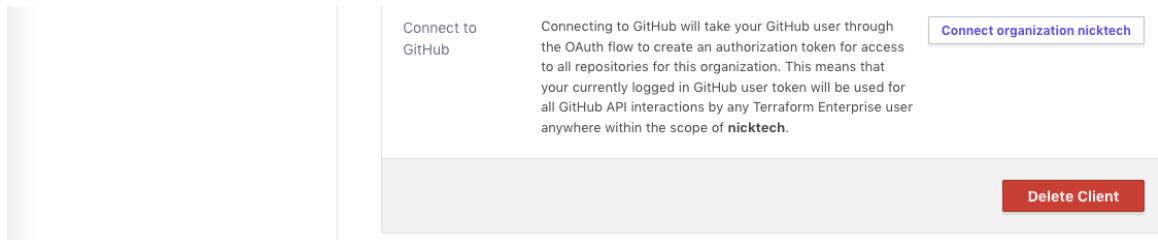
Step 3: On GitLab, Update the Callback URL

1. Go back to your GitLab browser tab. (If you accidentally closed it, you can reach your OAuth app page through the menus: use the upper right menu > Settings > Applications > "Terraform Enterprise (<YOUR ORG NAME>)".)
2. Click the "Edit" button.
3. In the "Redirect URI" field, paste the callback URL from TFE's VCS Provider page, replacing the "example.com" placeholder you entered earlier.

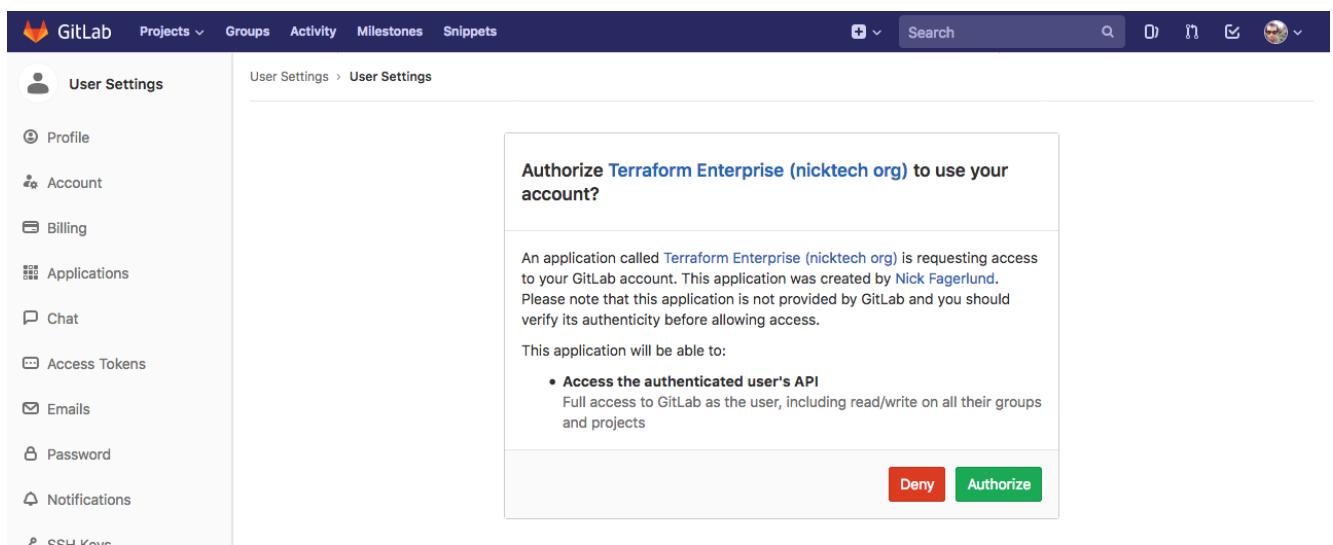
4. Click the "Save application" button. A banner saying the update succeeded should appear at the top of the page.

Step 4: On TFE, Request Access

1. Go back to your TFE browser tab and click the "Connect organization <NAME>" button on the VCS Provider page.



This takes you to a page on GitLab.com, asking whether you want to authorize the app.



2. Click the green "Authorize" button at the bottom of the authorization page. This returns you to TFE's VCS Provider page, where the GitLab.com client's information has been updated.

Finished

At this point, GitLab.com access for TFE is fully configured, and you can create Terraform workspaces based on your organization's shared repositories.

Configuring GitLab EE and CE Access

These instructions are for using an on-premise installation of GitLab Enterprise Edition (EE) or GitLab Community Edition (CE) for Terraform Enterprise (TFE)'s VCS features. GitLab.com has separate instructions, ([/docs/enterprise/vcs/gitlab-com.html](#)) as do the other supported VCS providers. ([/docs/enterprise/vcs/index.html](#))

Connecting TFE to your VCS involves five steps:

On your VCS	On TFE
Register your TFE organization as a new app. Get ID and key.	Tell TFE how to reach VCS, and provide ID and key. Get callback URL.
Provide callback URL.	Request VCS access.
Approve access request.	

The rest of this page explains the on-premise GitLab versions of these steps.

Important: TFE needs to contact your GitLab instance during setup and during normal operation. For the SaaS version of TFE, this means GitLab must be internet-accessible; for private installs of TFE, you must have network connectivity between your TFE and GitLab instances.

Note: Alternately, you can skip the OAuth configuration process and authenticate with a personal access token. This requires using TFE's API. For details, see the OAuth Clients API page ([/docs/enterprise/api/oauth-clients.html](#)).

Step 1: On GitLab, Create a New Application

1. Open your GitLab instance in your browser and log in as whichever account you want TFE to act as. For most organizations this should be a dedicated service user, but a personal account will also work.

Important: The account you use for connecting TFE **must have admin (master) access** to any shared repositories of Terraform configurations, since creating webhooks requires admin permissions. Do not create the application as an administrative application not owned by a user; TFE needs user access to repositories to create webhooks and ingress configurations.

Important: In GitLab EE 10.6 and up, you may also need to enable **Allow requests to the local network from hooks and services** on the "Outbound requests" section inside the Admin area under Settings ([/admin/application_settings](#)). Refer to the GitLab EE documentation (<https://docs.gitlab.com/ee/security/webhooks.html>) for details.

2. Navigate to GitLab's "User Settings > Applications" page.

This page is located at `https://<GITLAB INSTANCE HOSTNAME>/profile/applications`. You can also reach it

through GitLab's menus:

- In the upper right corner, click your profile picture and choose "Settings."
- In the navigation sidebar, click "Applications."

3. This page has a list of applications and a form for adding new ones. The form has two text fields and some checkboxes.

The screenshot shows the 'User Settings' page in GitLab, specifically the 'Applications' section. On the left, there is a sidebar with various settings options like Profile, Account, Billing, and Applications (which is selected). The main content area shows the 'Applications' section with a brief description: 'Manage applications that can use GitLab as an OAuth provider, and applications that you've authorized to use your account.' Below this, there is a form to add a new application. It includes fields for 'Name' (set to 'Terraform Enterprise (nicktech.org)'), 'Redirect URI' (set to 'https://example.com/replace-later'), and a 'Scopes' section with several checkboxes. The 'Scopes' section includes options like 'api', 'read_user', 'sudo', 'read_repository', 'read_registry', and 'openid'. A green 'Save application' button is at the bottom of the form.

Fill out the form as follows:

Field	Value
(all checkboxes)	(empty)
Name	Terraform Enterprise (<YOUR ORGANIZATION NAME>)
Redirect URI	https://example.com/replace-this-later (or any placeholder; the correct URI doesn't exist until the next step.)

4. Click the "Save application" button, which creates the application and takes you to its page.
5. Leave this page open in a browser tab. In the next step, you will copy and paste the unique **Application ID** and **Secret**.

The screenshot shows the GitLab User Settings Applications page. On the left sidebar, under the Applications section, there is a link to 'Access Tokens'. The main content area displays a success message: 'The application was created successfully.' Below this, the application details are shown: Application ID (79195593e40b3 [REDACTED]), Secret (6c38811d69a36 [REDACTED]), and Callback url (https://example.com/replace-later). At the bottom of the page are two buttons: 'Edit' and 'Destroy'.

Step 2: On TFE, Add a VCS Provider

1. Open TFE in your browser and navigate to the "VCS Provider" settings for your organization. Click the "Add VCS Provider" button.

If you just created your organization, you might already be on this page. Otherwise:

1. Click the upper-left organization menu, making sure it currently shows your organization.
 2. Click the "Settings" link at the top of the page (or within the \equiv menu)
 3. On the next page, click "VCS Provider" in the left sidebar.
 4. Click the "Add VCS Provider" button.
2. The next page has a drop-down and four text fields. Select "GitLab Enterprise Edition" or "GitLab Community Edition" from the drop-down, and fill in all four text fields as follows:

Field	Value
HTTP URL	<code>https://<GITLAB INSTANCE HOSTNAME></code>
API URL	<code>https://<GITLAB INSTANCE HOSTNAME>/api/v4</code>
Application ID	(paste value from previous step)
Secret	(paste value from previous step)

ORGANIZATION SETTINGS

- nicktech ✓
- Profile
- Teams
- VCS Provider
- API Token
- Authentication
- Manage SSH Keys
- Sentinel Policy

Add VCS Provider

To connect workspaces to git repositories containing Terraform configurations, Terraform Enterprise needs access to your version control system (VCS) provider. Use this page to configure OAuth authentication with your VCS provider. For more information, please see the Terraform Enterprise documentation on [Configuring Version Control Access](#).

VERSION CONTROL SYSTEM (VCS) PROVIDER

GitLab Enterprise Edition

HTTP URL

`https://gitlab.yourcompany.com`

The location of your GitLab Enterprise Edition installation.

API URL

`https://gitlab.yourcompany.com/api/v3`

The location of your GitLab Enterprise Edition installation's API.

APPLICATION ID

[Add a new application on your GitLab Enterprise installation](#) to generate the Application Id and Secret.

SECRET

[Add a new application on your GitLab Enterprise installation](#) to generate the Application Id and Secret.

Create VCS Provider

Cancel

Note that TFE uses GitLab's v3 API.

3. Click "Create connection." This will take you back to the VCS Provider page, which now includes your new GitLab client.
4. Locate the new client's "Callback URL," and copy it to your clipboard; you'll paste it in the next step. Leave this page open in a browser tab.

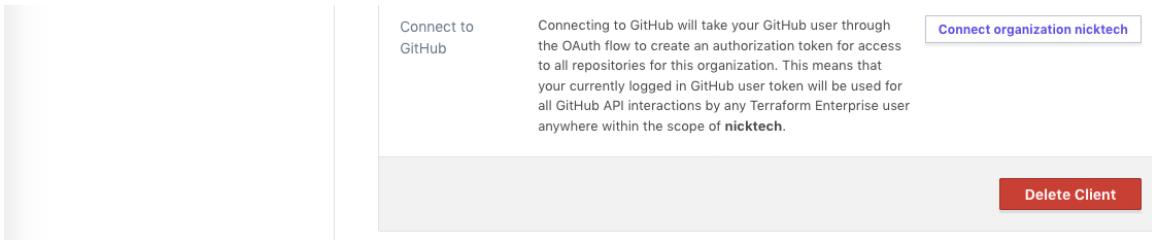
GitLab.com	
Callback URL	<code>https://app.terraform.io/auth/88cccd8fe-.../callback</code>
HTTP URL	<code>https://gitlab.com</code>

Step 3: On GitLab, Update the Callback URL

1. Go back to your GitLab browser tab. (If you accidentally closed it, you can reach your OAuth app page through the menus: use the upper right menu > Settings > Applications > "Terraform Enterprise (<YOUR ORG NAME>)".)
2. Click the "Edit" button.
3. In the "Redirect URI" field, paste the callback URL from TFE's VCS Provider page, replacing the "example.com" placeholder you entered earlier.
4. Click the "Save application" button. A banner saying the update succeeded should appear at the top of the page.

Step 4: On TFE, Request Access

1. Go back to your TFE browser tab and click the "Connect organization <NAME>" button on the VCS Provider page.



This takes you to a page on GitLab, asking whether you want to authorize the app.

User Settings > User Settings

Authorize [Terraform Enterprise \(nicktech org\)](#) to use your account?

An application called [Terraform Enterprise \(nicktech org\)](#) is requesting access to your GitLab account. This application was created by [Nick Fagerlund](#). Please note that this application is not provided by GitLab and you should verify its authenticity before allowing access.

This application will be able to:

- Access the authenticated user's API

Full access to GitLab as the user, including read/write on all their groups and projects

[Deny](#) [Authorize](#)

2. Click the green "Authorize" button at the bottom of the authorization page. This returns you to TFE's VCS Provider page, where the GitLab client's information has been updated.

If this results in a 500 error, it usually means TFE was unable to reach your GitLab instance.

Finished

At this point, GitLab access for TFE is fully configured, and you can create Terraform workspaces based on your organization's shared repositories.

Troubleshooting VCS Integration in Terraform Enterprise

This page collects solutions to the most common problems our users encounter with VCS integration in Terraform Enterprise (TFE).

Bitbucket Server

Clicking "Connect organization <X>" with Bitbucket Server raises an error message in Terraform Enterprise

TFE uses OAuth 1 to authenticate the user to Bitbucket Server. The first step in the authentication process is for TFE to call Bitbucket Server to obtain a "request token". After the call completes TFE will redirect you to Bitbucket Server with the request token.

An error occurs when TFE calls to Bitbucket Server to obtain the request token but the request is rejected. Some common reasons for the request to be rejected are:

- The API endpoint is unreachable; this can happen if the address or port is incorrect or the domain name doesn't resolve.
- The certificate used on Bitbucket Server is rejected by the TFE HTTP client because the SSL verification fails. This is often the case with self-signed certificates or when the server with TFE is not configured to trust the signing chain of the Bitbucket Server SSL certificate.

To fix this issue, do the following:

- Verify that the instance running Terraform Enterprise can resolve the domain name and can reach Bitbucket Server.
- Verify that the TFE client accepts the HTTPS connection to Bitbucket Server. This can be done by performing a `curl` from the TFE instance to Bitbucket Server; it should not return any SSL errors.
- Verify that the Consumer Key, Consumer Name, and the Public Key are configured properly in Bitbucket Server.
- Verify that the HTTP URL and API URL in TFE are correct for your Bitbucket Server instance. This includes the proper scheme (HTTP vs HTTPS), as well as the port.

Creating a workspace from a repository hangs indefinitely, displaying a spinner on the confirm button

If you were able to connect TFE to Bitbucket Server but cannot create workspaces, it often means TFE isn't able to automatically add webhook URLs for that repository.

To fix this issue:

- Make sure you haven't manually entered any webhook URLs for the affected repository or project. Although the Bitbucket Web Post Hooks Plugin documentation describes how to manually enter a hook URL, TFE handles this automatically. Manually entered URLs can interfere with TFE's operation.

To check the hook URLs for a repository, go to the repository's settings, then go to the "Hooks" page (in the "Workflow" section) and click on the "Post-Receive WebHooks" link.

Also note that some Bitbucket Server versions might allow you to set per-project or server-wide hook URLs in addition to per-repository hooks. These should all be empty; if you set a hook URL that might affect more than one repo when installing the plugin, go back and delete it.

- Make sure you aren't trying to connect too many workspaces to a single repository. Bitbucket Server's webhooks plugin can only attach five hooks to a given repo. You might need to create additional repositories if you need to make more than five workspaces from a single configuration repo.

Bitbucket Cloud

Terraform Enterprise fails to obtain the list of repositories from Bitbucket Cloud.

This typically happens when the TFE application in Bitbucket Cloud wasn't configured to have the full set of permissions. Go to the OAuth section of the Bitbucket settings, find your TFE OAuth consumer, click the edit link in the "..." menu, and ensure it has the required permissions enabled:

Permission type	Permission level
Account	Write
Repositories	Admin
Pull requests	Write
Webhooks	Read and write

GitHub

"Host key verification failed" error in `terraform init` when attempting to ingress Terraform modules via Git over SSH

This is most common when running Terraform 0.10.3 or 0.10.4, which had a bug in handling SSH submodule ingress. Try upgrading affected TFE workspaces to the latest Terraform version or 0.10.8 (the latest in the 0.10 series).

TFE can't ingress Git submodules, with auth errors during `init`

This usually happens when an SSH key isn't associated with the VCS provider's OAuth client.

- Go to your organization's "VCS Provider" settings page and check your GitHub client. If it still says "You can add a private SSH key to this connection to be used for git clone operations" (instead of "A private SSH key has been added..."), you need to click the "add a private SSH key" link and add a key.
- Check the settings page for affected workspaces and ensure that "Include submodules on clone" is enabled.

Note that the "SSH Key" section in a workspace's settings is only used for mid-run operations like cloning Terraform modules. It isn't used when cloning the linked repository before a run.

General

Terraform Enterprise returns 500 after authenticating with the VCS provider (other than Bitbucket Server)

The Callback URL in the OAuth application configuration in the VCS provider probably wasn't updated in the last step of the instructions and still points to the default "/" path (or an example.com link) instead of the full callback url.

The fix is to update the callback URL in your VCS provider's application settings. You can look up the real callback URL in TFE's settings.

Can't delete a workspace or module, resulting in 500 errors

This often happens when the VCS connection has been somehow broken: it might have had permissions revoked, been reconfigured, or had the repository removed. Check for these possibilities and contact HashiCorp support for further assistance, including any information you collected in your support ticket.

On TFE's SaaS version: redirect_uri_mismatch error on "Connect"

The domain name for Terraform Enterprise changed on 02/22 at 9AM from atlas.hashicorp.com to app.terraform.io. If the OAuth client was originally configured on the old domain, using it for a new VCS connection can result in this error.

The fix is to update the OAuth Callback URL in your VCS provider to use app.terraform.io instead of atlas.hashicorp.com.

Certificate Errors on Private Terraform Enterprise

When debugging failures of VCS connections due to certificate errors, running additional diagnostics using the OpenSSL command may provide more information about the failure.

First, attach a bash session to the application container:

```
docker exec -it ptfe_atlas sh -c "stty rows 50 && stty cols 150 && bash"
```

Then run the `openssl s_client` command, using the certificate at `/tmp/cust-ca-certificates.crt` in the container:

```
openssl s_client -showcerts -CAfile /tmp/cust-ca-certificates.crt -connect git-server-hostname:443
```

For example, a Gitlab server that uses a self-signed certificate might result in an error like `verify error:num=18:self signed certificate`, as shown in the output below:

```
bash-4.3# openssl s_client -showcerts -CAfile /tmp/cust-ca-certificates.crt -connect gitlab.local:443
```

```
CONNECTED(00000003)
depth=0 CN = gitlab.local
verify error:num=18:self signed certificate
verify return:1
depth=0 CN = gitlab.local
verify return:1
---
Certificate chain
0 s:/CN=gitlab.local
 i:/CN=gitlab.local
-----BEGIN CERTIFICATE-----
MIIC/DCCAeSgAwIBAgIJAihG2GWtcj7lMA0GCSqGSIB3DQEBCwUAMCAxHjAcBgNV
BAMMFwdpdGxhYi1sb2NhbC5oYXNoaS5jbzAeFw0xODA2MDQyMjAwMDhaFw0xOTA2
MDQyMjAwMDhaMCAXhjAcBgNVBAMMFwdpdGxhYi1sb2NhbC5oYXNoaS5jbzCCASiw
DQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMMgrp03zsoy2BP/AoGIgrYwEMnj
PwSOFGNHbclmiVBCW9jvrZrtva8Qh+twU7CSQdkeSP34ZgLrRp1msmLvUuVMgPts
i7isri5hug/IHLLOG05xMvx0crHknvySYJRmvYFRIEBPNRPYJGJ901ZUVUYeNwW/
l9eegBDpJrdsjGmFKCoZEdUA3zu7PfNgf788uIi4UKVXZNa/OFhsZi630Yyf0c2
Zm0/vRKOn17dew0OesHhw77yBh80FsEic10JCe5y3MD9yrhV1h9Z4niK8rHPXz
XEh3JfV+BBArodmbv4UtT+IGdDueUllXv7kbwqvQ670Fmmek0GZOY7ZvMCawEA
AaM5MDcwIAYDVR0RBBkwF4IVZ2l0bGFILWxvY2FsLmhhc2hpLmNvMBMGA1udJQQM
MAoGCCsGAQUFBwMBMA0GCSqGSIB3DQEBCwUAA4IBAQcfkukNV/9vCA/8qoEbPt1M
mvf2FHyUD69p/Gq/04IHGty3sno4eVcwWEc5EvfNt8vv1FykFQ6zMJuWA0jL9x2s
LbC8yuRDnsAlukSBvyazC9pt3qseG0LskaVCe0qG3b+hJqikZihFUD95IvWNFQs
RpvGvnA/AH2Lqqeyk2ITtLYj1AcSB1hBSnG/0fdtao9zs0JQsrS59CD1lbbTPPRN
orbKtVTWF2JlJxl2watfCNTw6nTCPi+51CYd687T3MuRN7LsTgglzP4xazuNbWB
QGAiQRd6aKj+xAJnqjzXt9wl6a493m8aNkyWrxZGHfIA1W70RtMqIC/554flZ4ia
-----END CERTIFICATE-----
---
Server certificate
subject=/CN=gitlab.local
issuer=/CN=gitlab.local
---
No client certificate CA names sent
Peer signing digest: SHA512
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read 1443 bytes and written 433 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol : TLSv1.2
    Cipher   : ECDHE-RSA-AES256-GCM-SHA384
    Session-ID: AF5286FB7C7725D377B4A5F556DEB6DDC38B302153DDAE90C552ACB5DC4D86B8
    Session-ID-ctx:
    Master-Key: DB75AEC12C6E7B62246C653C8CB8FC3B90DE86886D68CB09898A6A6F5D539007F7760BC25EC4563A893D34ABC
FAAC28A
    Key-Ag      : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 300 (seconds)
    TLS session ticket:
        0000 - 03 c1 35 c4 ff 6d 24 a8-6c 70 61 fb 2c dc 2e b8 ..5..m$.lpa.,...
        0010 - de 4c 6d b0 2c 13 8e b6-63 95 18 ee 4d 33 a6 dc .Lm.,...c...M3..
        0020 - 0d 64 24 f0 8d 3f 9c aa-b8 a4 e2 4f d3 c3 4d 88 .d$..?.....0..M.
        0030 - 58 99 10 73 83 93 70 4a-2c 61 e7 2d 41 74 d3 e9 X..s..pJ,a.-At..
        0040 - 83 8c 4a 7f ae 7b e8 56-5c 51 fc 6f fe e3 a0 ec ..J..{.V\Q.o.....
        0050 - 3c 2b 6b 13 fc a0 e5 15-a8 31 16 19 11 98 56 43 <+k.....1....VC
        0060 - 16 86 c4 cd 53 e6 c3 61-e2 6c 1b 99 86 f5 a8 bd ....S..a.l.....
```

```
0070 - 3c 49 c0 0a ce 81 a9 33-9b 95 2c e1 f4 6d 05 1e <I.....3...,m..  
0080 - 18 fa bf 2e f2 27 cc 0b-df 08 13 7e 4d 5a c8 41 .....'.....~MZ.A  
0090 - 93 26 23 90 f1 bb ba 3a-15 17 1b 09 6a 14 a8 47 .#.....:....j..G  
00a0 - 61 eb d9 91 0a 5c 4d e0-4a 8f 4d 50 ab 4b 81 aa a....\M.J.MP.K..
```

Start Time: 1528152434

Timeout : 300 (sec)

Verify return code: 18 (self signed certificate)

closed

Workspaces

Workspaces are how Terraform Enterprise (TFE) organizes infrastructure. If you've used the legacy version of TFE, workspaces used to be called environments.

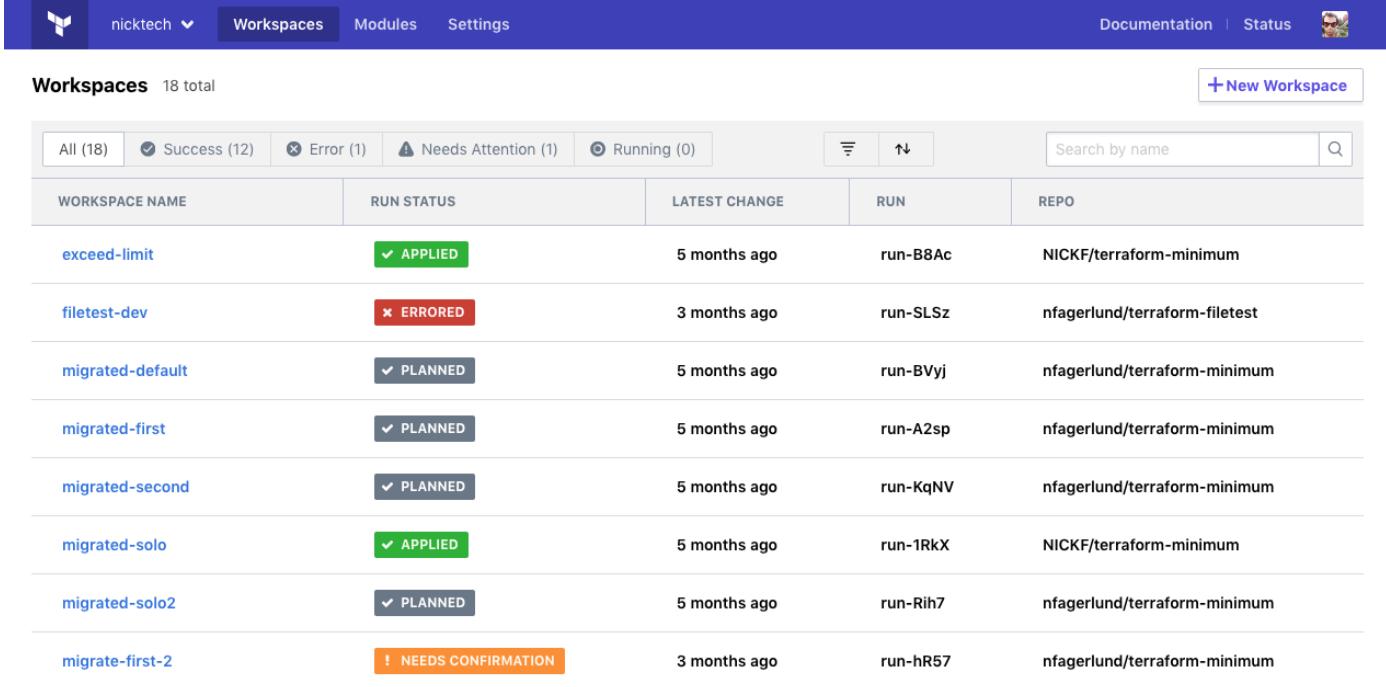
A workspace consists of:

- A Terraform configuration (usually retrieved from a VCS repo, but sometimes uploaded directly).
- Values for variables used by the configuration.
- Persistent stored state for the resources the configuration manages.
- Historical state and run logs.

Listing and Filtering Workspaces

API: See the Workspaces API (/docs/enterprise/api/workspaces.html).

TFE's top navigation bar includes a "Workspaces" link, which takes you to the list of workspaces in the current organization.



The screenshot shows the TFE Workspaces page with a blue header bar. The header includes the TFE logo, a dropdown for 'nicktech', and navigation links for 'Workspaces', 'Modules', and 'Settings'. On the right side of the header are 'Documentation' and 'Status' links, along with a user profile icon. Below the header is a search bar with a placeholder 'Search by name' and a magnifying glass icon. To the right of the search bar is a button labeled '+ New Workspace' with a plus sign. The main content area displays a table titled 'Workspaces' with 18 total entries. The table has columns: 'WORKSPACE NAME', 'RUN STATUS', 'LATEST CHANGE', 'RUN', and 'REPO'. Each row contains a workspace name, its status (e.g., 'APPLIED', 'ERRORED', 'PLANNED'), the date of the latest change, the run ID, and the repository it belongs to. For example, the first workspace is 'exceed-limit' with an 'APPLIED' status, last changed 5 months ago, run ID 'run-B8Ac', and repository 'NICKF/terraform-minimum'. The last workspace listed is 'migrate-first-2' with a 'NEEDS CONFIRMATION' status, last changed 3 months ago, run ID 'run-hR57', and repository 'nfagerlund/terraform-minimum'.

WORKSPACE NAME	RUN STATUS	LATEST CHANGE	RUN	REPO
exceed-limit	✓ APPLIED	5 months ago	run-B8Ac	NICKF/terraform-minimum
filetest-dev	✗ ERRORED	3 months ago	run-SLSz	nfagerlund/terraform-filetest
migrated-default	✓ PLANNED	5 months ago	run-BVjy	nfagerlund/terraform-minimum
migrated-first	✓ PLANNED	5 months ago	run-A2sp	nfagerlund/terraform-minimum
migrated-second	✓ PLANNED	5 months ago	run-KqNV	nfagerlund/terraform-minimum
migrated-solo	✓ APPLIED	5 months ago	run-1RkX	NICKF/terraform-minimum
migrated-solo2	✓ PLANNED	5 months ago	run-Rih7	nfagerlund/terraform-minimum
migrate-first-2	! NEEDS CONFIRMATION	3 months ago	run-hR57	nfagerlund/terraform-minimum

This list only includes workspaces where your user account has at least read permissions (/docs/enterprise/users-teams-organizations/permissions.html).

If the list is large, you can use the filter tools at the top of the list to find the workspaces you're interested in.

The following filters are available:

- **Status filters:** These filters sort workspaces by the status of their current run. There are four quick filter buttons that collect the most commonly used groups of statuses (success, error, needs attention, and running), and a custom filter button (with a funnel icon) where you can select any number of statuses from a menu.

When you choose a status filter, the list will only include workspaces whose current runs match the selected statuses. You can remove the status filter by clicking the "All" button, or by unchecking everything in the custom filter menu.

- **List order:** The list order button is marked with two arrows, pointing up and down. You can choose to order the list by time or by name, in forward or reverse order.
- **Name filter:** The search field at the far right of the filter bar lets you filter workspaces by name. If you enter a string in this field and press enter, only workspaces whose names contain that string will be shown.

The name filter can combine with a status filter, to narrow the list down further.

Planning and Organizing Workspaces

We recommend that organizations break down large monolithic Terraform configurations into smaller ones, then assign each one to its own workspace and delegate permissions and responsibilities for them. TFE can manage monolithic configurations just fine, but managing smaller infrastructure components like this is the best way to take full advantage of TFE's governance and delegation features.

For example, the code that manages your production environment's infrastructure could be split into a networking configuration, the main application's configuration, and a monitoring configuration. After splitting the code, you would create "networking-prod", "app1-prod", "monitoring-prod" workspaces, and assign separate teams to manage them.

Much like splitting monolithic applications into smaller microservices, this enables teams to make changes in parallel. In addition, it makes it easier to re-use configurations to manage other environments of infrastructure ("app1-dev," etc.).

Managing Access to Workspaces

Terraform Enterprise (TFE) workspaces can only be accessed by users with the correct permissions. You can manage permissions for a workspace on a per-team basis.

Workspace access should be managed by organization owners ([/docs/enterprise/users-teams-organizations/teams.html#the-owners-team](#)). (Users with admin privileges ([/docs/enterprise/users-teams-organizations/permissions.html](#)) on a workspace can make minor adjustments to its permissions, but only owners can manage permissions for any team in the organization.)

Background

TFE manages workspace permissions with teams, and uses three levels of permissions (read, write, and admin).

For more information see:

- [Users, Teams and Organizations](#) ([/docs/enterprise/users-teams-organizations/index.html](#))
- [Permissions](#) ([/docs/enterprise/users-teams-organizations/permissions.html](#))

Managing Workspace Access Permissions

API: See the Team Access APIs ([/docs/enterprise/api/team-access.html](#)).

Terraform: See the `tfe_team_access` resource ([/docs/providers/tfe/r/team_access.html](#)).

When a workspace is created, the only team able to access it is the owners team, with full admin permissions. The owners team can never be removed from a workspace.

To manage other teams' access, go to the workspace's page and click the "Access" tab.

minimum-prod

 Queue PlanCurrent Run Runs States Variables Settings Version Control **Access**

Access

Add a team

TEAM NAME

The name of the team you wish to add.

PERMISSIONS

The level of access the team will have.

[Add team](#)

Teams

Owners of nicktech

DEFAULT

test_team (admin)

[Remove](#)

This page has a pair of drop-downs for adding new teams, and a list of teams that already have access.

To add a team, select it from the first dropdown and choose which permissions it should have (read, write, or admin) with the second dropdown, then click the "Add team" button.

To remove a team's permissions on the workspace, click the "Remove" button next to that team's entry in the teams list.

To change a team's permissions on the workspace, you must first remove the team, then re-add it.

Creating Workspaces

Important: Only organization owners (</docs/enterprise/users-teams-organizations/teams.html#the-owners-team>) can create new workspaces.

API: See the Create a Workspace endpoint (</docs/enterprise/api/workspaces.html#create-a-workspace>) (POST `/organizations/:organization/workspaces`).

Terraform: See the `tfe` provider's `tfe_workspace` resource (</docs/providers/tfe/r/workspace.html>).

Create new Terraform Enterprise (TFE) workspaces with the "+ New Workspace" button, which appears on the list of workspaces. If you're not already viewing the workspace list, you can get there with the "Workspaces" button in the top navigation bar.

Configuring a New Workspace

The screenshot shows the 'Create a new Workspace' dialog. At the top, there are tabs for 'New workspace' (selected) and 'Import from legacy (Atlas) environment'. Below this, a note says 'This workspace will be created under the current organization, nicktech.' The 'WORKSPACE NAME' field contains 'e.g. workspace-name'. A note below it says 'The name of your workspace is unique and used in tools, routing, and UI. Dashes, underscores, and alphanumeric characters are permitted. Learn more about [naming workspaces](#).' Under 'SOURCE', there are buttons for 'None', 'Bitbucket Server' (selected), 'Bitbucket Cloud', 'GitLab.com', 'GitHub' (disabled), and a '+' button. The 'REPOSITORY' field contains 'e.g. organization/repository-name'. A note below it says 'The repository identifier in the format username/repository. Only the most recently updated repositories will appear with autocomplete; however, all repositories are available for use.' At the bottom, there is a 'More options (working directory, VCS branch, ingress submodules)' link, a 'Create Workspace' button (highlighted in blue), and a 'Cancel' button.

You must fill out several fields to configure your new workspace:

- **Workspace name** (required) — A name for the workspace, which must be unique in the organization. Names can include letters, numbers, _, and -. See more advice about workspace names here (</docs/enterprise/workspaces/naming.html>).

- **Source** (required; list of buttons) — Which connected VCS provider (/docs/enterprise/vcs/index.html) the workspace should pull configurations from. If you've configured multiple VCS providers, there is a button for each of them.

If you select "None," the workspace cannot pull configurations automatically, but you can upload configurations with the remote backend (/docs/enterprise/run/cli.html), the run API (/docs/enterprise/run/api.html), or the optional TFE CLI client (<https://github.com/hashicorp/tfe-cli/>).

- **Repository** — The VCS repository that contains the Terraform configuration for this workspace. This field is hidden when creating a workspace without a VCS source.

Repository identifiers are determined by your VCS provider, and use a format like <ORGANIZATION>/<REPO NAME> or <PROJECT KEY>/<REPO NAME>.

This field supports autocomplete of your most recently used repositories. If you need to specify a repository that isn't included in the autocomplete list, you can enter the full name manually.

If necessary, you can change a workspace's VCS repository after creating it.

The screenshot shows the configuration interface for a VCS-backed workspace. At the top, under the 'SOURCE' section, there are four buttons: 'None', 'Bitbucket Server', 'Bitbucket Cloud', and 'GitLab.com'. Below these is a button for 'GitHub' which is highlighted in blue, indicating it is selected. To the right of the GitHub button is a '+' sign for adding more sources. A horizontal line separates this from the 'REPOSITORY' section. In the 'REPOSITORY' section, there is a search input field containing 'terrafor'. Below the input field, a list of repositories is shown, with the first item, 'nfagerlund/terraform-website', highlighted in purple. Other items in the list include 'nfagerlund/terraform', 'nfagerlund/terraform-minimum', and a link 'Use "terrafor"'. The entire interface has a clean, modern design with a light gray background and white text.

VCS-backed workspaces support several optional fields, which you can reveal by clicking the "More options" link. These fields are hidden when creating a workspace without a VCS source.

REPOSITORY

The repository identifier in the format username/repository. Only the most recently updated repositories will appear with autocomplete; however, all repositories are available for use.

[^ Hide additional options](#)

TERRAFORM WORKING DIRECTORY

The directory that Terraform will execute within. This defaults to the root of your repository and is typically set to a subdirectory matching the environment when multiple environments exist within the same repository.

VCS BRANCH

The branch from which to import new versions. This defaults to the value your version control provides as the default branch for this repository.

Include submodules on clone

Checking this box will perform a recursive clone of your repository's submodules, making them available in the resulting slug containing your Terraform configuration.

Create Workspace
Cancel

- **Terraform working directory** — The directory where Terraform will execute, specified as a relative path from the root of the repo. This is useful when working with VCS repos that contain multiple Terraform configurations. Defaults to the root of the repo.

Note: If you specify a working directory, TFE will still queue a plan for changes to the repository outside that working directory. This is because local modules are often outside the working directory, and changes to those modules should result in a new run. If you have a repo that manages multiple infrastructure components with different lifecycles and are experiencing too many runs, we recommend splitting the components out into independent repos. See Repository Structure (/docs/enterprise/workspaces/repo-structure.html) for more detailed explanations.

- **VCS branch** — Which branch of the repository to use. If left blank, TFE will use the repository's default branch.
- **Include submodules on clone** (checkbox) — Whether to recursively clone all of the repository's Git submodules when fetching a configuration.

Note: The SSH key for cloning Git submodules (/docs/enterprise/vcs/index.html#ssh-keys) is set in the VCS provider settings for the organization, and is not necessarily related to the SSH key set in the workspace's settings.

Importing Legacy Environments

The new workspace page includes an "Import from legacy (Atlas) environment" tab, located above all of the workspace settings. If you previously used the legacy version of Terraform Enterprise, you can use this tab to migrate legacy environments to your new organization, preserving their existing state and settings.

The process of migrating a legacy environment to a new Terraform Enterprise workspace is covered in detail at Upgrading from Legacy Terraform Enterprise (/docs/enterprise/upgrade/index.html).

After Creating a Workspace

When you create a new workspace, a few things happen:

- TFE *doesn't* immediately queue a plan for the workspace. Instead, it presents a dialog with shortcut links to either queue a plan or edit variables. If you don't need to edit variables, manually queuing a plan confirms to TFE that the workspace is ready to run.
- If you selected a VCS provider and repository, TFE automatically registers a webhook. The next time new commits appear in the selected branch of that repo or a PR is opened to that branch, TFE will automatically queue a Terraform plan for the workspace. More at [VCS Connection webhooks](#) ([/docs/enterprise/vcs/index.html#webhooks](#)).

A workspace with no runs will not accept new runs via VCS webhook; at least one run must be manually queued to confirm that the workspace is ready for further runs.

Most of the time, you'll want to do one or more of the following after creating a workspace:

- Edit variables ([/docs/enterprise/workspaces/variables.html](#))
- Edit workspace settings ([/docs/enterprise/workspaces/settings.html](#))
- Work with runs ([/docs/enterprise/run/index.html](#))

Workspace Naming

Terraform Enterprise organizes workspaces by name, so it's important to use a consistent and informative naming strategy. And although future releases of TFE will add more organizational tools, the name will always be the most important piece of information about a workspace.

Note that workspace names can only include letters, numbers, -, and _.

The best way to make names that are both unique and useful is to combine the workspace's most distinguishing *attributes* in a consistent order. Attributes can be any defining characteristic of a workspace — such as the component being managed, the environment it runs in, and the region it is provisioned into.

A good strategy to start with is <COMPONENT>-<ENVIRONMENT>-<REGION>. For example:

- networking-prod-us-east
- networking-staging-us-east
- networking-prod-us-eu-central
- networking-staging-eu-central
- monitoring-prod-us-east
- monitoring-staging-us-east
- monitoring-prod-us-eu-central
- monitoring-staging-eu-central

If those three attributes can't uniquely distinguish all of your workspaces, you might need to add another attribute; for example, the infrastructure provider (AWS, GCP, Azure), datacenter, or line of business.

Repository Structure

Terraform Enterprise integrates with version control repositories to obtain configurations and trigger Terraform runs. Structuring these repos properly is important because it determines which files Terraform has access to when Terraform is executed within Terraform Enterprise, and when Terraform plans will run.

Manageable Repos

As a best practice for repository structure, each repository containing Terraform code should be a manageable chunk of infrastructure ([/docs/enterprise/guides/recommended-practices/part1.html#the-recommended-terraform-workspace-structure](#)), such as an application, service, or specific type of infrastructure (like common networking infrastructure).

When repositories are interrelated, we recommend using remote state ([/docs/enterprise/guides/recommended-practices/part3.3.html#3-design-your-organization-s-workspace-structure](#)) to transfer information between workspaces. Small configurations connected by remote state are more efficient for collaboration than monolithic repos, because they let you update infrastructure without running unnecessary plans in unrelated workspaces.

Structuring Repos for Multiple Environments

When each repository represents a manageable chunk of Terraform code, it's often still useful to attach a single repository to multiple workspaces in order to handle multiple environments or other cases where similar infrastructure is used in a different context. There are three primary ways to structure the Terraform code in your repository to manage multiple environments (such as dev, stage, prod).

Depending on your organization's use of version control, one method for multi-environment management may be better than another.

Multiple Workspaces per Repo (Recommended)

Using a single repo attached to multiple workspaces is the simplest best-practice approach, as it enables the creation of a pipeline to promote changes through environments, without additional overhead in version control. When using this model, one repo, such as `terraform-networking`, is connected to multiple workspaces — `networking-prod`, `networking-stage`, `networking-dev`. While the repo connection is the same in each case, each workspace can have a unique set of variables to configure the differences per environment.

To make an infrastructure change, a user opens a pull request on the `terraform-networking` repo, which will trigger a speculative plan ([/docs/enterprise/run/index.html#speculative-plans](#)) in all three connected workspaces. The user can then merge the PR and apply it in one workspace at a time, first with `networking-dev`, then `networking-stage`, and finally `networking-prod`. Eventually, Terraform Enterprise will have functionality to enforce the stages in this pipeline.

This model will not work for a given repo if there are major environmental differences. For example, if the `networking-prod` workspace has 10 more unique resources than the `networking-stage` workspace, they likely cannot share the same Terraform configuration and thus cannot share the same repo. If there are major structural differences between environments, one of the below approaches may be better.

Branches

For organizations that prefer long-running branches, we recommend creating a branch for each environment. When using this model, one repo, such as `terraform-networking`, would have three long-running, branches — `prod`, `stage`, and `dev`.

Using the branch strategy reduces the number of files needed in the repo. In the example repo structure below, there is only one `main.tf` configuration and one `variables.tf` file. When connecting the repo to a workspace in TFE, you can set different variables for each workspace — one set of variables for `prod`, one set for `stage`, and one set for `dev`.

```
└── README.md
└── variables.tf
└── main.tf
└── outputs.tf
└── modules
    └── compute
        ├── README.md
        ├── variables.tf
        ├── main.tf
        └── outputs.tf
    └── networking
        ├── README.md
        ├── variables.tf
        ├── main.tf
        └── outputs.tf
```

Each workspace listens to a specific branch for changes, as configured by the VCS branch setting ([/docs/enterprise/workspaces/settings.html#vcs-branch](#)). This means that plans will not occur in a given workspace until a PR is opened or a push event occurs on the designated branch. The `networking-prod` workspace would be configured to listen to the `prod` branch, `networking-stage` to `stage`, and `networking-dev` to `dev`. To promote a change to `stage`, open a PR against the `stage` branch. To promote to `prod`, open a PR from `stage` against `prod`.

The upside of this approach is that it requires fewer files and runs fewer plans, but the potential downside is that the branches can drift out of sync. Thus, in this model, it's very important to enforce consistent branch merges for promoting changes.

Directories

For organizations that have significant differences between environments, or prefer short-lived branches that are frequently merged into the master branch, we recommend creating a separate directory for each environment.

Important: Since workspaces will queue a plan whenever their VCS branch changes or receives a pull request, you should avoid connecting too many workspaces to the same repository. Connecting more than about ten workspaces to one branch of a repo can cause slow performance and unnecessary noise; if you need to support more workspaces than that, we recommend splitting components into multiple repos.

In the example repo structure below, the `prod`, `stage`, and `dev` environments have separate `main.tf` configurations and `variables.tf` files. These environments can still refer to the same modules (like `compute` and `networking`).

```
└── environments
    ├── prod
    │   ├── README.md
    │   ├── variables.tf
    │   ├── main.tf
    │   └── outputs.tf
    ├── stage
    │   ├── README.md
    │   ├── variables.tf
    │   ├── main.tf
    │   └── outputs.tf
    └── dev
        ├── README.md
        ├── variables.tf
        ├── main.tf
        └── outputs.tf
└── modules
    ├── compute
    │   ├── README.md
    │   ├── variables.tf
    │   ├── main.tf
    │   └── outputs.tf
    └── networking
        ├── README.md
        ├── variables.tf
        ├── main.tf
        └── outputs.tf
```

When using this model, each workspace is configured with a different Terraform Working Directory ([/docs/enterprise/workspaces/settings.html#terraform-working-directory](#)). This setting tells TFE which directory to execute Terraform in. The `networking-prod` workspace is configured with `prod` as its working directory, the `networking-stage` workspace is configured with `stage` as its working directory, and likewise for `networking-dev`. Unlike in the previous example, every workspace listens for changes to the master branch. Thus, every workspace will run a plan when a change is made to master, because (for example) changes to the modules could affect any environment's behavior.

The potential downside to this approach is that changes have to be manually promoted between stages, and the directory contents can drift out of sync. This model also results in more plans than the long-lived branch model, since every workspace plans on each PR or change to master.

Workspace Settings

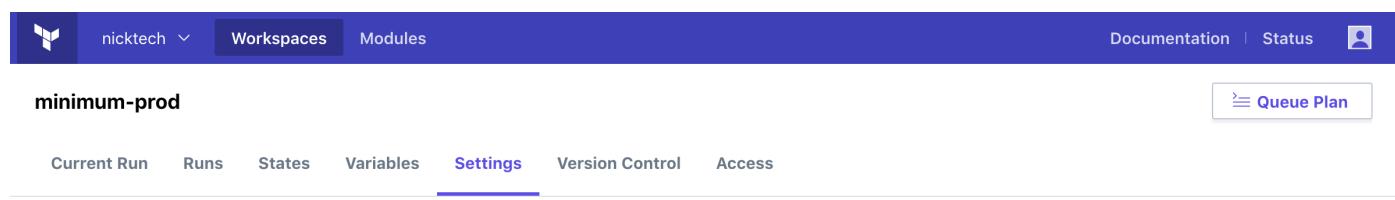
API: See the Update a Workspace endpoint ([/docs/enterprise/api/workspaces.html#update-a-workspace](#)) (PATCH `/organizations/:organization_name/workspaces/:name`).

Terraform Enterprise (TFE) workspaces can be reconfigured after creation.

Each workspace's page has two tabs of settings:

- "Settings," for general configuration.
- "Version Control," for managing the workspace's VCS integration.

Changing settings requires admin privileges ([/docs/enterprise/users-teams-organizations/permissions.html](#)) on the affected workspace.



General Settings

The following settings are available in the "Settings" tab.

ID

The permanent unique ID of the workspace, which cannot be changed. Workspace IDs are sometimes necessary when working with TFE's API ([/docs/enterprise/api/index.html](#)).

Click the icon beside the ID to copy it to the clipboard.

Name

The display name of the workspace.

Important: Since some API calls refer to a workspace by its name, changing the name can sometimes break existing integrations.

After changing this setting you must click the "Save settings" button below the "Terraform Working Directory" field.

Auto Apply and Manual Apply

Whether or not TFE should automatically apply a successful Terraform plan. If you choose manual apply, an operator must confirm a successful plan and choose to apply it.

After changing this setting you must click the "Save settings" button below the "Terraform Working Directory" field.

Terraform Version

Which version of Terraform to use for all operations in the workspace. You can choose "latest" to automatically update to new versions, or you can lock a workspace to any specific version.

By default, new workspaces are locked to the current version of Terraform at the time of their creation. (You can specify a Terraform version when creating a workspace via the API.)

After changing this setting you must click the "Save settings" button below the "Terraform Working Directory" field.

Terraform Working Directory

The directory where Terraform will execute, specified as a relative path from the root of the configuration directory. This is useful when working with VCS repos that contain multiple Terraform configurations. Defaults to the root of the configuration directory.

Note: If you specify a working directory, TFE will still queue a plan for changes to the repository outside that working directory. This is because local modules are often outside the working directory, and changes to those modules should result in a new run. If you have a repo that manages multiple infrastructure components with different lifecycles and are experiencing too many runs, we recommend splitting the components out into independent repos. See Repository Structure (/docs/enterprise/workspaces/repo-structure.html) for more detailed explanations.

After changing this setting you must click the "Save settings" button below it.

SSH Key

If a workspace's configuration uses Git-based module sources (/docs/modules/sources.html) to reference Terraform modules in private Git repositories, Terraform needs an SSH key to clone those repositories.

This feature is documented in more detail in Using SSH Keys for Cloning Modules (/docs/enterprise/workspaces/ssh-keys.html).

After changing this setting, you must click the "Update SSH key" button below it.

Workspace Lock

Important: Unlike the rest of the settings on this page, locks can be managed with either write or admin privileges.

If you need to prevent Terraform runs for any reason, you can lock a workspace. This prevents users with write access from manually queueing runs, prevents automatic runs due to changes to the backing VCS repo, and prevents the creation of runs via the API. To enable runs again, a user must unlock the workspace.

Locking a workspace also restricts state uploads. In order to upload state, the workspace must be locked by the user who is uploading state.

Important: The `atlas` backend (</docs/backends/types/terraform-enterprise.html>) ignores this restriction, and allows users with write access to modify state when the workspace is locked. To prevent confusion and accidents, avoid using the `atlas` backend in normal workflows and use the `remote` backend instead; see TFE's CLI-driven workflow (</docs/enterprise/run/cli.html>) for details.

Users with write access can lock and unlock a workspace, but can't unlock a workspace which was locked by another user. Users with admin privileges can force unlock a workspace even if another user has locked it.

Locks are managed with a single "Lock/Unlock/Force unlock <WORKSPACE NAME>" button. TFE asks for confirmation when unlocking.

Workspace Delete

This section includes two buttons:

- "Queue destroy Plan"
- "Delete from Terraform Enterprise"

In almost all cases, you should perform both actions in that order when destroying a workspace.

Queueing a destroy plan destroys the infrastructure managed by a workspace. If you don't do this, the infrastructure resources will continue to exist but will become unmanaged; you'll need to go into your infrastructure providers to delete the resources manually.

Before queueing a destroy plan, you must set a `CONFIRM_DESTROY` environment variable in the workspace with a value of 1.

Version Control Settings

The following settings are available in the "Version Control" tab.

Important: After changing any of these settings you must click the "Update VCS settings" button at the bottom of the page.

VCS Connection and Repository

You can use the "Select a VCS connection" buttons and "Repository" field to change which VCS repository the workspace gets configurations from. See [Creating Workspaces](#) (</docs/enterprise/workspaces/creating.html>) for more details about selecting a VCS repository, and see [Connecting VCS Providers to Terraform Enterprise](#) (</docs/enterprise/vcs/index.html>) for more details about configuring VCS integrations.

VCS Branch

Which branch of the repository to use. If left blank, TFE will use the repository's default branch.

Include submodules on clone

Whether to recursively clone all of the repository's Git submodules when fetching a configuration.

Note: The SSH key for cloning Git submodules ([/docs/enterprise/vcs/index.html#ssh-keys](#)) is set in the VCS provider settings for the organization, and is not necessarily related to the SSH key set in the workspace's settings.

Using SSH Keys for Cloning Modules

Terraform configurations can pull in Terraform modules from a variety of different sources (/docs/modules/sources.html), and private Git repositories are the most common source for private modules.

To access a private Git repository, Terraform either needs login credentials (for HTTPS access) or an SSH key. Terraform Enterprise (TFE) can store private SSH keys centrally, and you can easily use them in any workspace that clones modules from a Git server.

Note: SSH keys for cloning Terraform modules from Git repos are only used during Terraform runs. They are managed separately from any keys used for bringing VCS content into TFE (/docs/enterprise/vcs/index.html#ssh-keys).

TFE manages SSH keys used to clone Terraform modules at the organization level, and allows multiple keys to be added for the organization. You can add or delete keys via the organization's settings. Once a key is uploaded, the text of the key is not displayed to users.

To assign a key to a workspace, go to its settings and choose a previously added key from the drop-down menu on Integrations under "SSH Key". Each workspace can only use one SSH key.

Adding and Deleting Keys

API: See the SSH Keys API (/docs/enterprise/api/ssh-keys.html).

Terraform: See the `tfe_ssh_key` provider's resource (/docs/providers/tfe/r/ssh_key.html).

To add or delete an SSH private key, use the main menu to go to your organization's settings and choose "Manage SSH Keys" from the navigation sidebar. This page has a form for adding new keys and a list of existing keys.

The screenshot shows the 'Manage SSH Keys' page in the Terraform Enterprise web interface. The left sidebar has a dark blue header with the organization name 'nicktech'. Below it, there are several navigation items: Profile, Teams, OAuth Configuration, API Token, User Sessions, and 'Manage SSH Keys', which is highlighted with a blue background. The main content area has a white background. At the top, it says 'Manage SSH Keys'. Below that is a section for 'Add a private key' with fields for 'NAME' (containing 'Key name') and 'PRIVATE SSH KEY' (containing 'Key text'). At the bottom of this section is a blue 'Add Private SSH Key' button. Below this is a section titled 'Private Keys' containing a table with one row: 'Name: service_tfe_rsa' and a red 'Destroy' button. The overall layout is clean and modern, using a combination of blues, whites, and greys.

To add a key:

1. Obtain an SSH keypair that TFE can use to download modules during a Terraform run. You might already have an appropriate key; if not, create one on a secure workstation and distribute the public key to your VCS provider(s). Do not use or generate a key that has a passphrase; Git is running non-interactively and won't be able to prompt for it.

The exact command to create a keypair depends on your OS, but is usually something like `ssh-keygen -t rsa -f "/Users/<NAME>/.ssh/service_tfe" -C "service_terraform_enterprise"`. This creates a `service_tfe` file with the private key, and a `service_tfe.pub` file with the public key.

2. Enter a name for the key in the "Name" field. Choose something identifiable, since the name is the only way to tell two SSH keys apart once the key text is hidden.
3. Paste the text of the **private key** in the "Private SSH Key" field.
4. Click the "Add Private SSH Key" button.

After the key is saved, it will appear below in the list of keys. Keys are only listed by name; TFE retains the text of the private key, but will never again display it for any purpose.

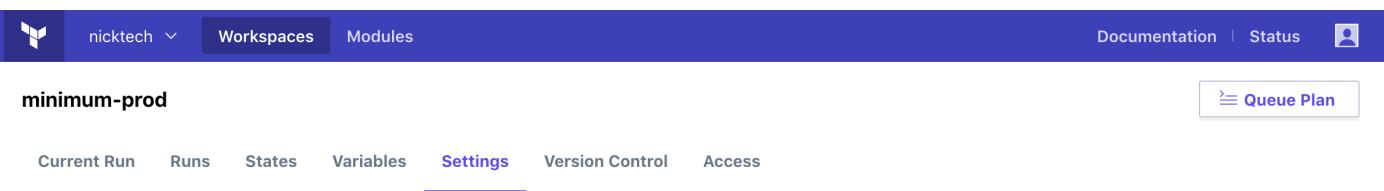
To delete a key, find it in the list of keys and click its "Delete" button. Before deleting a key, you should assign a new key to any workspaces that are using it.

Important: If any workspaces are still using a key when you delete it, they will be unable to clone modules from private repos until you assign them a new key. This might cause Terraform runs to fail.

Assigning Keys to Workspaces

API: See the Assign an SSH Key to a Workspace endpoint (</docs/enterprise/api/workspaces.html#assign-an-ssh-key-to-a-workspace>).

To assign a key to a workspace, navigate to that workspace's page and click the "Settings" link.



The screenshot shows the workspace settings page for 'minimum-prod'. At the top, there is a navigation bar with a user icon, the username 'nicktech', and dropdown menus for 'Workspaces' and 'Modules'. To the right of the navigation bar are links for 'Documentation' and 'Status'. Below the navigation bar, the workspace name 'minimum-prod' is displayed. Underneath the workspace name, there is a horizontal menu with tabs: 'Current Run', 'Runs', 'States', 'Variables', 'Settings' (which is highlighted in blue), 'Version Control', and 'Access'. On the far right of this menu, there is a button labeled 'Queue Plan'. The main content area contains a section titled 'SSH Key' with a dropdown menu open, showing a list of keys. Below the dropdown, there is a button labeled 'Update SSH key'.

Scroll down and locate the "SSH Key" dropdown menu. Select a named key from the list in this dropdown, then click the "Update SSH key" button directly below the menu.

SSH Key

Optionally choose a private SSH key to be used during Terraform operations. An SSH Key is required if a [provisioner connection](#) in the terraform configuration requires an SSH connection. This key is not used while cloning the workspace VCS repository.

[Manage SSH keys for the nicktech organization.](#)

SSH KEY

▼

[Update SSH key](#)

In subsequent runs, TFE will use the selected SSH key in this workspace when cloning modules from Git.

Variables

Terraform Enterprise (TFE) workspaces can set values for two kinds of variables:

- Terraform input variables (</docs/configuration/variables.html>), which define the parameters of a Terraform configuration.
- Shell environment variables, which many providers can use for credentials and other data. You can also set environment variables that affect Terraform's behavior (</docs/configuration/environment-variables.html>), like `TF_LOG`.

You can edit a workspace's variables via the UI or the API. All runs in a workspace use its variables.

API: See the Variables API (</docs/enterprise/api/variables.html>).

Terraform: See the `tfe` provider's `tfe_variable` resource (</docs/providers/tfe/r/variable.html>).

Loading Variables from Files

If a workspace is configured to use Terraform 0.10.0 or later, you can commit any number of `*.auto.tfvars` files (</docs/configuration/variables.html#variable-files>) to provide default variable values. Terraform will automatically load variables from those files.

If any automatically loaded variables have the same names as variables specified in the TFE workspace, TFE's values will override the automatic values (except for map values, which are merged (</docs/configuration/variables.html#variable-merging>)).

You can also use the optional TFE CLI tool (<https://github.com/hashicorp/tfe-cli/>)'s `tfe pushvars` command to update a workspace's variables using a local variables file. This has the same effect as managing workspace variables manually or via the API, but can be more convenient for large numbers of complex variables.

Managing Variables in the UI

To view and manage a workspace's variables, navigate to that workspace and click the "Variables" navigation link at the top.

The variables page has separate lists of Terraform variables and environment variables:

minimum-prod

 Queue PlanCurrent Run Runs States **Variables** Settings Integrations Version Control Access

Variables

These variables are used for all plans and applies in this workspace. Workspaces using Terraform 0.10.0 or later can also load default values from any `*.auto.tfvars` files in the configuration.

Sensitive variables are hidden from view in the UI and API, and can't be edited. (To change a sensitive variable, delete and replace it.) Sensitive variables can still appear in Terraform logs if your configuration is designed to output them.

When setting many variables at once, the [TFE CLI tool's](#) `pushvars` command or the [variables API](#) can often save time.

Terraform Variables

These [Terraform variables](#) are set using a `terraform.tfvars` file. To use interpolation or set a non-string value for a variable, click its HCL checkbox.

username	service-deploy
prior_workspace	nicktech/minimum-dev

[+ Add Variable](#)

Environment Variables

These variables are set in Terraform's shell environment using `export`.

EXAMPLE_PROVIDER_ACCESS_TOKEN	sensitive - write only
-------------------------------	------------------------

[+ Add Variable](#)

To edit a variable, click one of its text fields or its pencil (edit) icon to reveal the editing controls. Make any desired changes to the variable's name, value, and settings, then click the "Save Variable" button.

To add a variable, click the "+ Add Variable" button, enter a name and value, and save.

To delete a variable, click its " " (trash can) icon, then confirm your decision in the dialog box that appears.

You can edit one variable at a time, and must save or cancel your current edits before editing other variables in the list.

Terraform Variables

These [Terraform variables](#) are set using a `terraform.tfvars` file. To use interpolation or set a non-string value for a variable, click its HCL checkbox.

The screenshot shows a modal dialog for managing Terraform variables. At the top, there's a note about setting variables via `terraform.tfvars`. Below that, a variable is defined with the name "username" and the value "service-deploy". There are checkboxes for "HCL" and "Sensitive". A "Save Variable" button is prominent, and a "Cancel" button is also present. At the bottom, there are two input fields: "prior_workspace" and "nicktech/minimum-dev", each with edit and delete icons. A "+ Add Variable" button is located at the bottom left.

Environment Variables

These variables are set in Terraform's shell environment using `export`.

The screenshot shows a modal dialog for managing environment variables. It states that variables are set in Terraform's shell environment using `export`. A variable is listed with the name "EXAMPLE_PROVIDER_ACCESS_TOKEN" and the value "sensitive - write only". There are edit and delete icons next to the value. A "+ Add Variable" button is at the bottom left.

Multi-line Values

The text fields for variable values can handle multi-line text (typed or pasted) without any special effort.

HCL Values

Variable values are strings by default. To enter list or map values, click the variable's "HCL" checkbox (visible when editing) and enter the value with the same HCL syntax you would use when writing Terraform code. For example:

```
{  
  us-east-1 = "image-1234"  
  us-west-2 = "image-4567"  
}
```

HCL can be used for Terraform variables, but not for environment variables. The HCL code you enter for values is interpreted by the same Terraform version that performs runs in the workspace. (See [How TFE Uses Variables](#) below.)

Sensitive Values

Terraform often needs cloud provider credentials and other sensitive information that shouldn't be widely available within your organization.

To protect these secrets, you can mark any Terraform or environment variable as sensitive data by clicking its "Sensitive" checkbox (visible when editing).

Marking a variable as sensitive prevents anybody (including you) from viewing its value in TFE's UI or API.

Users with edit permissions can set new values for sensitive variables. No other attribute of a sensitive variable can be modified. To update other attributes, delete the variable and create a new variable to replace it.

Important: Terraform runs will receive the full text of sensitive variables, and might print the value in logs if the configuration pipes the value through to an output or a resource parameter. Take care when writing your configurations to avoid unnecessary credential disclosure.

Looking Up Variable Names

Terraform Enterprise can't automatically discover variable names from a workspace's Terraform code. You must discover the necessary variable names by reading code or documentation, then enter them manually.

If a required input variable is missing, Terraform plans in the workspace will fail and print an explanation in the log.

How TFE Uses Variables

Terraform Variables

TFE passes variables to Terraform by writing a `terraform.tfvars` file and passing the `-var-file=terraform.tfvars` option to the Terraform command.

Do not commit a file named `terraform.tfvars` to version control, since TFE will overwrite it. (Note that you shouldn't check in `terraform.tfvars` even when running Terraform solely on the command line.)

Environment Variables

TFE performs Terraform runs on disposable Linux worker VMs using a POSIX-compatible shell. Before running Terraform, TFE populates the shell with environment variables using the `export` command.

Special Environment Variables

TFE uses some special environment variables to control dangerous or rarely used run behaviors.

- `CONFIRM_DESTROY` — If this environment variable is set to 1 in a workspace, an admin user can destroy all of the infrastructure managed by the workspace using the "Queue destroy plan" button in the settings page. The UI text for the destroy plan button includes a reminder about this safety measure.
- `TFE_PARALLELISM` — If present, TFE uses this to set `terraform apply`'s `-parallelism=<N>` flag (more info [\(/docs/internals/graph.html#walking-the-graph\)](#)). Valid values are between 1 and 256, inclusive; the default is 10. This is rarely necessary, but can fix problems with infrastructure providers that error on concurrent operations or use non-standard rate limiting. We recommend talking to HashiCorp support before using this.

Secure Storage of Variables

TFE encrypts all variable values securely using Vault's transit backend (<https://www.vaultproject.io/docs/secrets/transit/index.html>) prior to saving them. This ensures that no out-of-band party can read these values without proper authorization.

Extending Terraform

Terraform can be extended to allow users to manage more infrastructure providers with Providers (<https://www.terraform.io/docs/providers/index.html>) (containing Resources (<https://www.terraform.io/docs/configuration/resources.html>) and/or Data Sources (<https://www.terraform.io/docs/configuration/data-sources.html>)), more options to store Terraform state with Backends (<https://www.terraform.io/docs/backends>) and more options to provision instance with Provisioners (<https://www.terraform.io/docs/provisioners/index.html>). **Providers** and **Provisioners** are collectively categorized as "Plugins".

This is an advanced section! If you are looking for information on using Terraform with any of the existing Plugins, please refer to the Docs (/docs/index.html) section of this website.

The Extending Terraform section contains content for users who wish to extend Terraform. The intended audience is anyone wanting to add or edit source code ("developers") for either Terraform itself or a Terraform Plugin. The content assumes you have basic operating knowledge or experience using Terraform.

Below is a brief description of each section. The content is organized from simplest to most complex — developers new to writing code for Terraform should start at the top.

How Terraform Works (</docs/extend/how-terraform-works.html>)

High level overview of how the Terraform tool works. Learn about the logical components of Terraform (Core vs. Plugins) and the basics of how they interact.

Plugin Types (</docs/extend/plugin-types.html>)

Learn about the different types of Terraform plugins — Providers and Provisioners.

Writing Custom Providers (</docs/extend/writing-custom-providers.html>)

A step by step guide for writing, compiling, and executing an example Terraform Provider.

Schemas (</docs/extend/schemas/index.html>)

The Schema package is a high-level framework for easily writing Plugins for Terraform. Providers (with Resources and/or Data Sources), and Provisioners are all defined in terms of the Schema package, which includes builtin types and methods for developers to use when writing plugins.

Resources (</docs/extend/resources.html>)

The Resource package provides several utilities and conveniences for handling tasks such as state migrations and customized difference behavior, these tasks often come up during provider development as schemas must change and evolve over time.

Best Practices (</docs/extend/best-practices/index.html>)

The Best Practices section offers guides on techniques that range from the steps required to deprecate schema attributes, to testing patterns, and how to version and manage a provider changelog. These techniques were learned through years of Terraform development from both HashiCorp employees and Community members.

Testing (</docs/extend/testing/index.html>)

Terraform provides a testing framework for validating resource implementations. The Testing section provides a breakdown of how to compose these tests using Test Cases and Test Steps, as well as covering unit test conventions and how they apply to Terraform plugin development.

Community (</docs/extend/community/index.html>)

Terraform is a mature project with a growing community. There are active, dedicated people willing to help you through various mediums.

Terraform Plugin Best Practices

A key feature of Terraform is its plugin system, which separates the details of specific vendor APIs from the shared logic for managing state, managing configuration, and providing a safe plan and apply lifecycle. Plugins are responsible for the implementation of functionality for provisioning resources for a specific cloud provider, allowing each provider to fully support its unique resources and lifecycles and not settling for the lowest common denominator across all provider resources of that type (virtual machines, networks, configuration management systems, et. al). While each provider is unique, over the years we've accumulated some patterns that should be adhered to, to ensure a consistent user experience when using Terraform for any given provider. Listed below are a few best practices we've found that generally apply to most Providers, with a brief description of each, and link to read more. Each practice is also linked in navigation on the left.

This section is a work in progress, with more sections to come.

Naming

Naming ([/docs/extend/best-practices/naming.html](#)) resources, data sources, and attributes in plugins is how plugin authors expose their functionality to operators and using patterns common to other plugins lays the foundation for a good user experience.

Deprecations, Removals, and Renames

Over time, remote services evolve and better workflows are designed. Terraform's plugin system has functionality to aid in iterative improvements. In Deprecations, Removals, and Renames ([/docs/extend/best-practices/deprecations.html](#)), we cover procedures for backwards compatible code and documentation updates to ensure that operators are well informed of changes ahead of functionality being removed or renamed.

Detecting Drift

Terraform is a declarative tool designed to be the source of truth for infrastructure. In order to safely and predictably change and iterate infrastructure, Terraform needs to be able to detect changes made outside of its configuration and provide means of reconciliation. In Detecting Drift ([/docs/extend/best-practices/detecting-drift.html](#)), we cover some best practices to ensure Terraform's statefile is an accurate reflection of reality, to provide accurate plan and apply functionality.

Testing Patterns

Terraform developers are encouraged to write acceptance tests that create real resource to verify the behavior of plugins, ensuring a reliable and safe way to manage infrastructure. In Testing Patterns ([/docs/extend/best-practices/testing.html](#)) we cover some basic acceptance tests that almost all resources should have to validate not only the functionality of the resource, but that the resource behaves as Terraform would expect.

Versioning and Changelog

Terraform development serves two distinct audiences: those writing plugin code and those implementing them. By clearly

and consistently allowing operators to easily understand changes in plugin implementation via version numbering and documenting those changes, a trust is formed between the two audiences. In Versioning and Changelog (/docs/extend/best-practices/versioning.html) we cover some guidelines when deciding release versions and how to relay changes through documentation.

Deprecations, Removals, and Renames

Terraform is trusted for managing many facets of infrastructure across many organizations. Part of that trust is due to consistent versioning guidelines and setting expectations for various levels of upgrades. Ensuring backwards compatibility for all patch and minor releases, potentially in concert with any upcoming major changes, is recommended and supported by the Terraform development framework. This allows operators to iteratively update their Terraform configurations rather than require massive refactoring.

This guide is designed to walk through various scenarios where existing Terraform functionality requires future removal, while maintaining backwards compatibility. Further information about the versioning terminology (e.g. MAJOR.MINOR.PATCH) in this guide can be found in the versioning guidelines documentation ([/docs/extend/best-practices/versioning.html](#)).

NOTE: Removals should only ever occur in MAJOR version upgrades.

Table of Contents

- Provider Attribute Removal
- Provider Attribute Rename
 - Renaming a Required Attribute
 - Renaming an Optional Attribute
 - Renaming a Computed Attribute
- Provider Data Source or Resource Removal
- Provider Data Source or Resource Rename

Provider Attribute Removal

The recommended process for removing an attribute from a data source or resource in a provider is as follows:

1. Add `Deprecated` in the attribute schema definition. After an operator upgrades to this version, they will be shown a warning with the message provided when using the attribute, but the Terraform run will still complete.
2. Ensure the changelog has an entry noting the deprecation.
3. Release a `MINOR` version with the deprecation.
4. In the next `MAJOR` version, remove all code associated with the attribute except for its schema definition.
5. Replace `Deprecated` with `Removed` in the attribute schema definition. After an operator upgrades to this version, they will be shown an error with the message provided when using the attribute.
6. Ensure the changelog has an entry noting the removal.
7. Release the `MAJOR` version.
8. In the next `MAJOR` version afterwards, completely remove the attribute schema definition. No changelog is necessary.

Provider Attribute Rename

When renaming an attribute from one name to another, it is important to keep backwards compatibility with both existing Terraform configurations and the Terraform state (/docs/state/index.html) while operators migrate. To accomplish this, there will be some duplicated logic to support both attributes until the next MAJOR release. Once both attributes are appropriately handled, the process for deprecating and removing the old attribute is the same as noted in the Provider Attribute Removal section.

The procedure for renaming an attribute depends on what type of attribute it is:

- Renaming a Required Attribute
- Renaming an Optional Attribute
- Renaming a Computed Attribute

Renaming a Required Attribute

NOTE: If the schema definition does not contain `Optional` or `Required`, see the Renaming a Computed Attribute section instead. If the schema definition contains `Optional` instead of `Required`, see the Renaming an Optional Attribute section.

Required attributes (/docs/extend/schemas/schema-behaviors.html#required) are also referred to as required "arguments" throughout the Terraform documentation.

In general, the procedure here does two things:

- Prevents the operator from needing to define two attributes with the same value.
- Allows the operator to migrate the configuration to the new attribute at the same time requiring that any other references only work with the new attribute. This is to prevent a situation with Terraform showing a difference when the existing attribute is configured, but the new attribute is saved into the Terraform state. For example, in `terraform plan` output format:

```
existing_attribute: "" => "value"
new_attribute:      "value" => ""
```

The recommended process is as follows:

1. Replace `Required: true` with `Optional: true` in the existing attribute schema definition.
2. Replace `Required` with `Optional` in the existing attribute documentation.
3. Duplicate the schema definition of the existing attribute, renaming one of them with the new attribute name.
4. Duplicate the documentation of the existing attribute, renaming one of them with the new attribute name.
5. Add `Deprecated` to the schema definition of the existing (now the "old") attribute, noting to use the new attribute in the message.
6. Add `**Deprecated**` to the documentation of the existing (now the "old") attribute, noting to use the new attribute.

7. Add a note to the documentation that either the existing (now the "old") attribute or new attribute must be configured.
8. Add `ConflictsWith` to the schema definitions of both the old and new attributes so they will present an error to the operator if both are configured at the same time.
9. Add conditional logic in the Create, Read, and Update functions of the data source or resource to handle both attributes. Generally, this involves using `ResourceData.GetOk()` (<https://godoc.org/github.com/hashicorp/terraform/helper/schema#ResourceData.GetOk>) (commonly `d.GetOk()` in HashiCorp maintained providers).
10. Add conditional logic in the Create and Update function that returns an error if both the old and new attributes are not defined.
11. Follow the rest of the procedures in the Provider Attribute Removal section. When the old attribute is removed, update the schema definition and documentation of the new attribute back to Required.

Example Renaming of a Required Attribute

Given this sample resource:

```

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Create: resourceExampleWidgetCreate,
        Read:   resourceExampleWidgetRead,
        Update: resourceExampleWidgetUpdate,

        Schema: map[string]*schema.Schema{
            // ... other attributes ...

            "existing_attribute": {
                Type:     schema.TypeString,
                Required: true,
            },
        },
    }
}

func resourceExampleWidgetCreate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    existingAttribute := d.Get("existing_attribute").(string)
    // add attribute to provider create API call

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    d.Set("existing_attribute", /* ... */)

    // ... other logic ...
    return nil
}

func resourceExampleWidgetUpdate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    existingAttribute := d.Get("existing_attribute").(string)
    // add attribute to provider update API call

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

```

In order to support renaming `existing_attribute` to `new_attribute`, this sample can be written as the following to support both attributes simultaneously until the `existing_attribute` is removed:

```

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Create: resourceExampleWidgetCreate,
        Read:   resourceExampleWidgetRead,
        Update: resourceExampleWidgetUpdate,

        Schema: map[string]*schema.Schema{
            // ... other attributes ...

```

```

        "existing_attribute": {
            Type:           schema.TypeString,
            Optional:       true,
            ConflictsWith: []string{"new_attribute"},
            Deprecated:    "use new_attribute instead",
        },
        "new_attribute": {
            Type:           schema.TypeString,
            Optional:       true,
            ConflictsWith: []string{"existing_attribute"},
        },
    },
},
}

func resourceExampleWidgetCreate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    existingAttribute, existingAttributeOk := d.GetOk("existing_attribute")
    newAttribute, newAttributeOk := d.GetOk("new_attribute")
    if !existingAttributeOk && !newAttributeOk {
        return errors.New("one of existing_attribute or new_attribute must be configured")
    }
    if existingAttributeOk {
        // add existingAttribute to provider create API call
    } else {
        // add newAttribute to provider create API call
    }

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    if _, ok := d.GetOk("existing_attribute"); ok {
        d.Set("existing_attribute", /* ... */)
    } else {
        d.Set("new_attribute", /* ... */)
    }

    // ... other logic ...
    return nil
}

func resourceExampleWidgetUpdate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    existingAttribute, existingAttributeOk := d.GetOk("existing_attribute")
    newAttribute, newAttributeOk := d.GetOk("new_attribute")
    if !existingAttributeOk && !newAttributeOk {
        return errors.New("one of existing_attribute or new_attribute must be configured")
    }
    if existingAttributeOk {
        // add existingAttribute to provider update API call
    } else {
        // add newAttribute to provider update API call
    }

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

```

When the `existing_attribute` is ready for removal, then this can be written as:

```
func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Create: resourceExampleWidgetCreate,
        Read:   resourceExampleWidgetRead,
        Update: resourceExampleWidgetUpdate,

        Schema: map[string]*schema.Schema{
            // ... other attributes ...

            "new_attribute": {
                Type:     schema.TypeString,
                Required: true,
            },
        },
    }
}

func resourceExampleWidgetCreate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    newAttribute := d.Get("new_attribute").(string)
    // add attribute to provider create API call

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    d.Set("new_attribute", /* ... */)

    // ... other logic ...
    return nil
}

func resourceExampleWidgetUpdate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    newAttribute := d.Get("new_attribute").(string)
    // add attribute to provider update API call

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}
```

Renaming an Optional Attribute

NOTE: If the schema definition does not contain `Optional` or `Required`, see the Renaming a Computed Attribute section instead. If the schema definition contains `Required` instead of `Optional`, see the Renaming a Required Attribute section.

Optional attributes (/docs/extend/schemas/schema-behaviors.html#optional) are also referred to as optional "arguments" throughout the Terraform documentation.

In general, the procedure here allows the operator to migrate the configuration to the new attribute at the same time requiring that any other references only work with the new attribute. This is to prevent a situation with Terraform showing a difference when the existing attribute is configured, but the new attribute is saved into the Terraform state. For example, in `terraform plan` output format:

```
existing_attribute: "" => "value"
new_attribute:      "value" => ""
```

The recommended process is as follows:

1. Duplicate the schema definition of the existing attribute, renaming one of them with the new attribute name.
2. Duplicate the documentation of the existing attribute, renaming one of them with the new attribute name.
3. Add `Deprecated` to the schema definition of the existing (now the "old") attribute, noting to use the new attribute in the message.
4. Add `**Deprecated**` to the documentation of the existing (now the "old") attribute, noting to use the new attribute.
5. Add `ConflictsWith` to the schema definitions of both the old and new attributes so they will present an error to the operator if both are configured at the same time.
6. Add conditional logic in the Create, Read, and Update functions of the data source or resource to handle both attributes. Generally, this involves using `ResourceData.GetOk()` (<https://godoc.org/github.com/hashicorp/terraform/helper/schema#ResourceData.GetOk>) (commonly `d.GetOk()` in HashiCorp maintained providers).
7. Follow the rest of the procedures in the Provider Attribute Removal section.

Example Renaming of an Optional Attribute

Given this sample resource:

```

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Create: resourceExampleWidgetCreate,
        Read:   resourceExampleWidgetRead,
        Update: resourceExampleWidgetUpdate,

        Schema: map[string]*schema.Schema{
            // ... other attributes ...

            "existing_attribute": {
                Type:      schema.TypeString,
                Optional: true,
            },
        },
    }
}

func resourceExampleWidgetCreate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    if v, ok := d.GetOk("existing_attribute"); ok {
        // add attribute to provider create API call
    }

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    d.Set("existing_attribute", /* ... */)

    // ... other logic ...
    return nil
}

func resourceExampleWidgetUpdate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    if v, ok := d.GetOk("existing_attribute"); ok {
        // add attribute to provider update API call
    }

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

```

In order to support renaming `existing_attribute` to `new_attribute`, this sample can be written as the following to support both attributes simultaneously until the `existing_attribute` is removed:

```

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Create: resourceExampleWidgetCreate,
        Read:   resourceExampleWidgetRead,
        Update: resourceExampleWidgetUpdate,

        Schema: map[string]*schema.Schema{

```

```

// ... other attributes ...

"existing_attribute": {
    Type:          schema.TypeString,
    Optional:      true,
    ConflictsWith: []string{"new_attribute"},
    Deprecated:   "use new_attribute instead",
},
"new_attribute": {
    Type:          schema.TypeString,
    Optional:      true,
    ConflictsWith: []string{"existing_attribute"},
},
},
},
}

func resourceExampleWidgetCreate(d *schema.ResourceData, meta interface{}) error {
// ... other logic ...

if v, ok := d.GetOk("existing_attribute"); ok {
    // add attribute to provider create API call
} else if v, ok := d.GetOk("new_attribute"); ok {
    // add attribute to provider create API call
}

// ... other logic ...
return resourceExampleWidgetRead(d, meta)
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
// ... other logic ...

if v, ok := d.GetOk("existing_attribute"); ok {
    d.Set("existing_attribute", /* ... */)
} else {
    d.Set("new_attribute", /* ... */)
}

// ... other logic ...
return nil
}

func resourceExampleWidgetUpdate(d *schema.ResourceData, meta interface{}) error {
// ... other logic ...

if v, ok := d.GetOk("existing_attribute"); ok {
    // add attribute to provider update API call
} else if v, ok := d.GetOk("new_attribute"); ok {
    // add attribute to provider update API call
}

// ... other logic ...
return resourceExampleWidgetRead(d, meta)
}

```

When the `existing_attribute` is ready for removal, then this can be written as:

```

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Create: resourceExampleWidgetCreate,
        Read:   resourceExampleWidgetRead,
        Update: resourceExampleWidgetUpdate,

        Schema: map[string]*schema.Schema{
            // ... other attributes ...

            "new_attribute": {
                Type:      schema.TypeString,
                Optional: true,
            },
        },
    }
}

func resourceExampleWidgetCreate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    if v, ok := d.GetOk("new_attribute"); ok {
        // add attribute to provider create API call
    }

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    d.Set("new_attribute", /* ... */)

    // ... other logic ...
    return nil
}

func resourceExampleWidgetUpdate(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    if v, ok := d.GetOk("new_attribute"); ok {
        // add attribute to provider update API call
    }

    // ... other logic ...
    return resourceExampleWidgetRead(d, meta)
}

```

Renaming a Computed Attribute

NOTE: If the schema definition contains `Optional` see the Renaming an Optional Attribute section instead. If the schema definition contains `Required` see the Renaming a Required Attribute section instead.

The recommended process is as follows:

1. Duplicate the schema definition of the existing attribute, renaming one of them with the new attribute name.

2. Duplicate the documentation of the existing attribute, renaming one of them with the new attribute name.
3. Add `Deprecated` to the schema definition of the existing (now the "old") attribute, noting to use the new attribute in the message.
4. Add `**Deprecated**` to the documentation of the existing (now the "old") attribute, noting to use the new attribute.
5. Set both attributes in the Terraform state in the `Read` functions of the data source or resource.
6. Follow the rest of the procedures in the Provider Attribute Removal section.

Example Renaming of a Computed Attribute

Given this sample resource:

```
func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Read: resourceExampleWidgetRead,
        Schema: map[string]*schema.Schema{
            // ... other attributes ...

            "existing_attribute": {
                Type:      schema.TypeString,
                Computed: true,
            },
        },
    }
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    d.Set("existing_attribute", /* ... */)

    // ... other logic ...
    return nil
}
```

In order to support renaming `existing_attribute` to `new_attribute`, this sample can be written as the following to support both attributes simultaneously until the `existing_attribute` is removed:

```

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Read: resourceExampleWidgetRead,

        Schema: map[string]*schema.Schema{
            // ... other attributes ...

            "existing_attribute": {
                Type:      schema.TypeString,
                Computed:  true,
                Deprecated: "use new_attribute instead",
            },
            "new_attribute": {
                Type:      schema.TypeString,
                Computed: true,
            },
        },
    }
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    d.Set("existing_attribute", /* ... */)
    d.Set("new_attribute", /* ... */)

    // ... other logic ...
    return nil
}

```

When the `existing_attribute` is ready for removal, then this can be written as:

```

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        Read: resourceExampleWidgetRead,

        Schema: map[string]*schema.Schema{
            // ... other attributes ...

            "new_attribute": {
                Type:      schema.TypeString,
                Computed: true,
            },
        },
    }
}

func resourceExampleWidgetRead(d *schema.ResourceData, meta interface{}) error {
    // ... other logic ...

    d.Set("new_attribute", /* ... */)

    // ... other logic ...
    return nil
}

```

Provider Data Source or Resource Removal

The recommended process for removing a data source or resource from a provider is as follows:

1. Add `DeprecationMessage` in the data source or resource schema definition. After an operator upgrades to this version, they will be shown a warning with the message provided when using the deprecated data source or resource, but the Terraform run will still complete.
2. Ensure the changelog has an entry noting the deprecation.
3. Release a `MINOR` version with the deprecation.
4. In the next `MAJOR` version, remove all code associated with the deprecated data source or resource. After an operator upgrades to this version, they will be shown an error about the missing data source or resource.
5. Ensure the changelog has an entry noting the removal.
6. Release the `MAJOR` version.

Example Resource Removal

Given this sample provider and resource:

```
func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        // ... other configuration ...

        ResourcesMap: map[string]*schema.Resource{
            // ... other resources ...
            "example_widget": resourceExampleWidget(),
        },
    }
}

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...
    }
}
```

In order to deprecate `example_widget`, this sample can be written as:

```

func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        // ... other configuration ...

        ResourcesMap: map[string]*schema.Resource{
            // ... other resources ...
            "example_widget": resourceExampleWidget(),
        },
    }
}

func resourceExampleWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        DeprecationMessage: "use example_thing resource instead"
    }
}

```

To remove example_widget:

```

func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        // ... other configuration ...

        ResourcesMap: map[string]*schema.Resource{
            // ... other resources ...
        },
    }
}

```

Provider Data Source or Resource Rename

When renaming a resource from one name to another, it is important to keep backwards compatibility with both existing Terraform configurations and the Terraform state while operators migrate. To accomplish this, there will be some duplicated logic to support both resources until the next MAJOR release. Once both resources are appropriately handled, the process for deprecating and removing the old resource is the same as noted in the Provider Data Source or Resource Removal section.

The recommended process is as follows:

1. Duplicate the code of the existing resource, renaming (and potentially modifying) functions as necessary.
2. Duplicate the documentation of the existing resource, renaming (and potentially modifying) as necessary.
3. Add `DeprecatedMessage` to the schema definition of the existing (now the "old") resource, noting to use the new resource in the message.
4. Add `!> **WARNING:** This resource is deprecated and will be removed in the next major version` to the documentation of the existing (now the "old") resource, noting to use the new resource.
5. Add the new resource to the provider `ResourcesMap`
6. Follow the rest of the procedures in the Provider Attribute Removal section.

Example Resource Renaming

Given this sample provider and resource:

```
func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        // ... other configuration ...

        ResourcesMap: map[string]*schema.Resource{
            // ... other resources ...

            "example_existing_widget": resourceExampleExistingWidget(),
        },
    }
}

func resourceExampleExistingWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...
    }
}
```

In order to support renaming `example_existing_widget` to `example_new_widget`, this sample can be written as the following to support both attributes simultaneously until the `existing_attribute` is removed:

```
func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        // ... other configuration ...

        ResourcesMap: map[string]*schema.Resource{
            // ... other resources ...

            "example_existing_widget": resourceExampleExistingWidget(),
            "example_new_widget":     resourceExampleNewWidget(),
        },
    }
}

func resourceExampleExistingWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...

        DeprecationMessage: "use example_new_widget resource instead"
    }
}

func resourceExampleNewWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...
    }
}
```

To remove `example_existing_widget`:

```
func Provider() terraform.ResourceProvider {
    return &schema.Provider{
        // ... other configuration ...

        ResourcesMap: map[string]*schema.Resource{
            // ... other resources ...

            "example_new_widget": resourceExampleNewWidget(),
        },
    }
}

func resourceExampleNewWidget() *schema.Resource {
    return &schema.Resource{
        // ... other configuration ...
    }
}
```

Detecting Drift

One of the core challenges of infrastructure as code is keeping an up-to-date record of all deployed infrastructure and their properties. Terraform manages this by maintaining state information in a single file, called the state file.

Terraform uses declarative configuration files to define the infrastructure resources to provision. This configuration serves as the target source of truth for what exists on the backend API. Changes to Infrastructure outside of Terraform will be detected as deviation by Terraform and shown as a diff in future runs of `terraform plan`. This type of change is referred to as "drift", and its detection is an important responsibility of Terraform in order to inform users of changes in their infrastructure. Here are a few techniques for developers to ensure drift is detected.

Capture all state in READ

A provider's READ method is where state is synchronized from the remote API to Terraform state. It's essential that all attributes defined in the schema are recorded and kept up-to-date in state. Consider this provider code:

```
// resource_example_simple.go
package example

func resourceExampleSimple() *schema.Resource {
    return &schema.Resource{
        Read:   resourceExampleSimpleRead,
        Create: resourceExampleSimpleCreate,
        Schema: map[string]*schema.Schema{
            "name": {
                Type:     schema.TypeString,
                Required: true,
                ForceNew: true,
            },
            "type": {
                Type:     schema.TypeString,
                Optional: true,
            },
        },
    }
}

func resourceExampleSimpleRead(d *schema.ResourceData, meta interface{}) error {
    client := meta.(*ProviderApi).client
    resource, _ := client.GetResource(d.Id())
    d.Set("name", resource.Name)
    d.Set("type", resource.Type)
    return nil
}
```

As defined in the schema, the `type` attribute is optional, now consider this config:

```
# config.tf
resource "simple" "ex" {
    name = "example"
}
```

Even though type is omitted from the config, it is vital that we record it into state in the READ function, as the backend API could set it to a default value. To illustrate the importance of capturing all state consider a configuration that interpolates the optional value into another resource:

```
resource "simple" "ex" {
  name = "example"
}

resource "another" "ex" {
  name = "${simple.ex.type}"
}
```

Update state after modification

A provider's CREATE and UPDATE functions will create or modify resources on the remote API. APIs might perform things like provide default values for unspecified attributes (as described in the above example config/provider code), or normalize inputs (lower or upper casing all characters in a string). The end result is a backend API containing modified versions of values that Terraform has in its state locally. Immediately after creation or updating of a resource, Terraform will have a stale state, which will result in a detected deviation on subsequent plan or applys, as Terraform refreshes its state and wants to reconcile the diff. Because of this, it is standard practice to call READ at the end of all modifications to synchronize immediately and avoid that diff.

```
func resourceExampleSimpleRead(d *schema.ResourceData, meta interface{}) error {
  client := meta.(*ProviderApi).client
  resource, _ := client.GetResource(d.Id())
  d.Set("name", resource.Name)
  d.Set("type", resource.Type)
  return nil
}

func resourceExampleSimpleCreate(d *schema.ResourceData, meta interface{}) error {
  client := meta.(*ProviderApi).client
  name := d.Get("name").(string)
  client.CreateResource(name)
  d.SetId(name)
  return resourceExampleSimpleRead(d, meta)
}
```

Error checking aggregate types

Terraform schema is defined using primitive types (<https://www.terraform.io/docs/extend/schemas/schema-types.html#primitive-types>) and aggregate types (<https://www.terraform.io/docs/extend/schemas/schema-types.html#aggregate-types>). The preceding examples featured primitive types which don't require error checking. Aggregate types on the other hand, schema.TypeList, schema.TypeSet, and schema.TypeMap, are converted to key/value pairs when set into state. As a result the Set method must be error checked, otherwise Terraform will think it's operation was successful despite having broken state. The same can be said for error checking API responses.

```
# config.tf
resource "simple" "ex" {
  name = "example"
  type = "simple"
  tags = {
    name = "example"
  }
}
```

```
// resource_example_simple.go
package example

func resourceExampleSimple() *schema.Resource {
  return &schema.Resource{
    Read:   resourceExampleSimpleRead,
    Create: resourceExampleSimpleCreate,
    Schema: map[string]*schema.Schema{
      "name": {
        Type:     schema.TypeString,
        Required: true,
        ForceNew: true,
      },
      "type": {
        Type:     schema.TypeString,
        Optional: true,
      },
      "tags": {
        Type:     schema.TypeMap,
        Optional: true,
      },
    },
  }
}

func resourceExampleSimpleRead(d *schema.ResourceData, meta interface{}) error {
  client := meta.(*ProviderApi).client
  resource, err := client.GetResource(d.Id())
  if err != nil {
    return fmt.Errorf("error getting resource %s: %s", d.Id(), err)
  }
  d.Set("name", resource.Name)
  d.Set("type", resource.Type)
  if err := d.Set("tags", resource.TagMap); err != nil {
    return fmt.Errorf("error setting tags for resource %s: %s", d.Id(), err)
  }
  return nil
}
```

Use Schema Helper methods

As mentioned, remote APIs can often perform mutations to the attributes of a resource outside of Terraform's control. Common examples include data containing uppercase letters and being normalized to lowercase, or complex defaults being set for unset attributes. These situations expectedly result in drift, but can be reconciled by using Terraform's schema functions (<https://www.terraform.io/docs/extend/schemas/schema-behaviors.html#function-behaviors>), such as `DiffSuppressFunc` or `DefaultFunc`.

Naming

Most names in a Terraform provider will be drawn from the upstream API/SDK that the provider is using. The upstream API names will likely need to be modified for casing or changing between plural and singular to make the provider more consistent with the common Terraform practices below.

Resource Names

Resource names are nouns, since resource blocks each represent a single object Terraform is managing. Resource names must always start with their containing provider's name followed by an underscore, so a resource from the provider `postgresql` might be named `postgresql_database`.

It is preferable to use resource names that will be familiar to those with prior experience using the service in question, e.g. via a web UI it provides.

Data Source Names

Similar to resource names, data source names should be nouns. The main difference is that in some cases data sources are used to return a list and can in those cases be plural. For example the data source `aws_availability_zones` (https://www.terraform.io/docs/providers/aws/d/availability_zones.html) in the AWS provider returns a list of availability zones.

Attribute Names

Below is an example of a resource configuration block which illustrates some general design patterns that can apply across all plugin object types:

```
resource "aws_instance" "example" {
  ami                  = "ami-408c7f28"
  instance_type        = "t1.micro"
  monitoring           = true
  vpc_security_group_ids = [
    "sg-1436abcf",
  ]
  tags                = {
    Name      = "Application Server"
    Environment = "production"
  }
  root_block_device {
    delete_on_termination = false
  }
}
```

Attribute names within Terraform configuration blocks are conventionally named as all-lowercase with underscores separating words, as shown above.

Simple single-value attributes, like `ami` and `instance_type` in the above example, are given names that are singular nouns, to reflect that only one value is required and allowed.

Boolean attributes like `monitoring` are usually written also as nouns describing what is being enabled. However, they can sometimes be named as verbs if the attribute is specifying whether to take some action, as with the `delete_on_termination` flag within the `root_block_device` block.

Boolean attributes are ideally oriented so that `true` means to do something and `false` means not to do it; it can be confusing to have "negative" flags that prevent something from happening, since they require the user to follow a double-negative in order to reason about what value should be provided.

Some attributes expect list, set or map values. In the above example, `vpc_security_group_ids` is a set of strings, while `tags` is a map from strings to strings. Such attributes should be named with *plural* nouns, to reflect that multiple values may be provided.

List and set attributes use the same bracket syntax, and differ only in how they are described to and used by the user. In lists, the ordering is significant and duplicate values are often accepted. In sets, the ordering is *not* significant and duplicated values are usually *not* accepted, since presence or absence is what is important.

Map blocks use the same syntax as other configuration blocks, but the keys in maps are arbitrary and not explicitly named by the plugin, so in some cases (as in this `tags` example) they will not conform to the usual "lowercase with underscores" naming convention.

Configuration blocks may contain other sub-blocks, such as `root_block_device` in the above example. The patterns described above can also apply to such sub-blocks. Sub-blocks are usually introduced by a singular noun, even if multiple instances of the same-named block are accepted, since each distinct instance represents a single object.

Testing Patterns

In Testing Terraform Plugins (/docs/extend/testing/index.html) we introduce Terraform's Testing Framework, providing reference for its functionality and introducing the basic parts of writing acceptance tests. In this section we'll cover some test patterns that are common and considered a best practice to have when developing and verifying your Terraform plugins. At time of writing these guides are particular to Terraform Resources, but other testing best practices may be added later.

Table of Contents

- Built-in Patterns
- Basic test to verify attributes
- Update test verify configuration changes
- Expecting errors or non-empty plans
- Regression tests

Built-in Patterns

Acceptance tests use TestCases (/docs/extend/testing/acceptance-tests/testcase.html) to construct scenarios that can be evaluated with Terraform's lifecycle of plan, apply, refresh, and destroy. The test framework has some behaviors built in that provide very basic workflow assurance tests, such as verifying configurations apply with no diff generated by the next plan.

Each TestCase will run any PreCheck (/docs/extend/testing/acceptance-tests/testcase.html#precheck) function provided before running the test, and then any CheckDestroy (/docs/extend/testing/acceptance-tests/testcase.html#checkdestroy) functions after the test concludes. These functions allow developers to verify the state of the resource and test before and after it runs.

When a test is ran, Terraform runs plan, apply, refresh, and then final plan for each TestStep (/docs/extend/testing/acceptance-tests/teststep.html) in the TestCase. If the last plan results in a non-empty plan, Terraform will exit with an error. This enables developers to ensure that configurations apply cleanly. In the case of introducing regression tests or otherwise testing specific error behavior, TestStep offers a boolean field ExpectNonEmptyPlan

(<https://github.com/hashicorp/terraform/blob/9441e78fb9c35037da71fd0284b97e546dd6a53b/helper/resource/testing.go#L299-L301>) as well ExpectError

(<https://github.com/hashicorp/terraform/blob/9441e78fb9c35037da71fd0284b97e546dd6a53b/helper/resource/testing.go#L303-L306>) regex field to specify ways the test framework can handle expected failures. If these properties are omitted and either a non-empty plan occurs or an error encountered, Terraform will fail the test.

After all TestSteps have been ran, Terraform then runs destroy, and ends by running any CheckDestroy function provided.

[Back to top](#)

Basic test to verify attributes

The most basic resource acceptance test should use what is likely to be a common configuration for the resource under test, and verify that Terraform can correctly create the resource, and that resources attributes are what Terraform expects them to be. At a high level, the first basic test for a resource should establish the following:

- Terraform can plan and apply a common resource configuration without error.

- Verify the expected attributes are saved to state, and contain the values expected.
- Verify the values in the remote API/Service for the resource match what is stored in state.
- Verify that a subsequent terraform plan does not produce a diff/change.

The first and last item are provided by the test framework as described above in **Built-in Patterns**. The middle items are implemented by composing a series of Check Functions as described in Acceptance Tests: TestSteps ([/docs/extend/testing/acceptance-tests/teststep.html#check-functions](#)).

To verify attributes are saved to the state file correctly, use a combination of the built-in check functions provided by the testing framework. See Built-in Check Functions ([/docs/extend/testing/acceptance-tests/teststep.html#builtin-check-functions](#)) to see available functions.

Checking the values in a remote API generally consists of two parts: a function to verify the corresponding object exists remotely, and a separate function to verify the values of the object. By separating the check used to verify the object exists into its own function, developers are free to re-use it for all TestCases as a means of retrieving its values, and can provide custom check functions per TestCase to verify different attributes or scenarios specific to that TestCase.

Here's an example test, with in-line comments to demonstrate the key parts of a basic test.

```
package example

// example.Widget represents a concrete Go type that represents an API resource
func TestAccExampleWidget_basic(t *testing.T) {
    var widget example.Widget

    // generate a random name for each widget test run, to avoid
    // collisions from multiple concurrent tests.
    // the acctest package includes many helpers such as RandStringFromCharSet
    // See https://godoc.org/github.com/hashicorp/terraform/helper/acctest
    rName := acctest.RandStringFromCharSet(10, acctest.CharSetAlphaNum)

    resource.Test(t, resource.TestCase{
        PreCheck: func() { testAccPreCheck(t) },
        Providers: testAccProviders,
        CheckDestroy: testAccCheckExampleResourceDestroy,
        Steps: []resource.TestStep{
            {
                // use a dynamic configuration with the random name from above
                Config: testAccExampleResource(rName),
                // compose a basic test, checking both remote and local values
                Check: resource.ComposeTestCheckFunc(
                    // query the API to retrieve the widget object
                    testAccCheckExampleResourceExists("example_widget.foo", &widget),
                    // verify remote values
                    testAccCheckExampleWidgetValues(widget, rName),
                    // verify local values
                    resource.TestCheckResourceAttr("example_widget.foo", "active", "true"),
                    resource.TestCheckResourceAttr("example_widget.foo", "name", rName),
                ),
            },
        },
    })
}

func testAccCheckExampleWidgetValues(widget *example.Widget, name string) resource.TestCheckFunc {
    return func(s *terraform.State) error {
        if *widget.Active != true {
            return fmt.Errorf("bad active state, expected \"true\", got: %#v", *widget.Active)
        }
        if *widget.Name != name {

```

```

        return fmt.Errorf("bad name, expected \"%s\", got: %#v", name, *widget.Name)
    }
    return nil
}
}

// testAccCheckExampleResourceExists queries the API and retrieves the matching Widget.
func testAccCheckExampleResourceExists(n string, widget *example.Widget) resource.TestCheckFunc {
    return func(s *terraform.State) error {
        // find the corresponding state object
        rs, ok := s.RootModule().Resources[n]
        if !ok {
            return fmt.Errorf("Not found: %s", n)
        }

        // retrieve the configured client from the test setup
        conn := testAccProvider.Meta().(*ExampleClient)
        resp, err := conn.DescribeWidget(&example.DescribeWidgetsInput{
            WidgetIdentifier: rs.Primary.ID,
        })

        if err != nil {
            return err
        }

        // If no error, assign the response Widget attribute to the widget pointer
        *widget = *resp.Widget

        return fmt.Errorf("Widget (%s) not found", rs.Primary.ID)
    }
}

// testAccExampleResource returns an configuration for an Example Widget with the provided name
func testAccExampleResource(name string) string {
    return fmt.Sprintf(`resource "example_widget" "foo" {
    active = true
    name = "%s"
}`, name)
}

```

This example covers all the items needed for a basic test, and will be referenced or added to in the other test cases to come.

[Back to top](#)

Update test verify configuration changes

A basic test covers a simple configuration that should apply successfully and with no follow up differences in state. To verify a resource correctly applies updates, the second most common test found is an extension of the basic test, that simply applies another `TestStep` with a modified version of the original configuration.

Below is an example test, copied and modified from the basic test. Here we preserve the `TestStep` from the basic test, but we add an additional `TestStep`, changing the configuration and rechecking the values, with a different configuration function `testAccExampleResourceUpdated` and check function `testAccCheckExampleWidgetValuesUpdated` for verifying the values.

```

func TestAccExampleWidget_update(t *testing.T) {
    var widget example.Widget
    rName := acctest.RandomStringFromCharSet(10, acctest.CharSetAlphaNum)

    resource.Test(t, resource.TestCase{
        PreCheck: func() { testAccPreCheck(t) },
        Providers: testAccProviders,
        CheckDestroy: testAccCheckExampleResourceDestroy,
        Steps: []resource.TestStep{
            {
                // use a dynamic configuration with the random name from above
                Config: testAccExampleResource(rName),
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckExampleResourceExists("example_widget.foo", &widget),
                    testAccCheckExampleWidgetValues(widget, rName),
                    resource.TestCheckResourceAttr("example_widget.foo", "active", "true"),
                    resource.TestCheckResourceAttr("example_widget.foo", "name", rName),
                ),
            },
            {
                // use a dynamic configuration with the random name from above
                Config: testAccExampleResourceUpdated(rName),
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckExampleResourceExists("example_widget.foo", &widget),
                    testAccCheckExampleWidgetValuesUpdated(widget, rName),
                    resource.TestCheckResourceAttr("example_widget.foo", "active", "false"),
                    resource.TestCheckResourceAttr("example_widget.foo", "name", rName),
                ),
            },
        },
    })
}

func testAccCheckExampleWidgetValuesUpdated(widget *example.Widget, name string) resource.TestCheckFunc {
    return func(s *terraform.State) error {
        if *widget.Active != false {
            return fmt.Errorf("bad active state, expected \"false\", got: %#v", *widget.Active)
        }
        if *widget.Name != name {
            return fmt.Errorf("bad name, expected \"%s\", got: %#v", name, *widget.Name)
        }
        return nil
    }
}

// testAccExampleResource returns an configuration for an Example Widget with the provided name
func testAccExampleResourceUpdated(name string) string {
    return fmt.Sprintf(`resource "example_widget" "foo" {
    active = false
    name = "%s"
}`, name)
}

```

It's common for resources to just have the above update test, as it is a superset of the basic test. So long as the basics are covered, combining the two tests is sufficient as opposed to having two separate tests.

[Back to top](#)

Expecting errors or non-empty plans

The number of acceptance tests for a given resource typically start small with the basic and update scenarios covered. Other tests should be added to demonstrate common expected configurations or behavior scenarios for a given resource, such as typical updates or changes to configuration, or exercising logic that uses polling for updates such as an autoscaling group adding or draining instances.

It is possible for scenarios to exist where a valid configuration (no errors during plan) would result in a non-empty plan after successfully running `terraform apply`. This is typically due to a valid but otherwise misconfiguration of the resource, and is generally undesirable. Occasionally it is useful to intentionally create this scenario in an early `TestStep` in order to demonstrate correcting the state with proper configuration in a follow-up `TestStep`. Normally a `TestStep` that results in a non-empty plan would fail the test after `apply`, however developers can use the `ExpectNonEmptyPlan` attribute to prevent failure and allow the `TestCase` to continue:

```
func TestAccExampleWidget_expectPlan(t *testing.T) {
    var widget example.Widget
    rName := acctest.RandStringFromCharSet(10, acctest.CharSetAlphaNum)

    resource.Test(t, resource.TestCase{
        PreCheck:     func() { testAccPreCheck(t) },
        Providers:   testAccProviders,
        CheckDestroy: testAccCheckExampleResourceDestroy,
        Steps: []resource.TestStep{
            {
                // use an incomplete configuration that we expect
                // to result in a non-empty plan after apply
                Config: testAccExampleResourceIncomplete(rName),
                Check: resource.ComposeTestCheckFunc(
                    resource.TestCheckResourceAttr("example_widget.foo", "name", rName),
                ),
                ExpectNonEmptyPlan: true,
            },
            {
                // apply the complete configuration
                Config: testAccExampleResourceComplete(rName),
                Check: resource.ComposeTestCheckFunc(
                    resource.TestCheckResourceAttr("example_widget.foo", "name", rName),
                ),
            },
        },
    })
}
```

In addition to `ExpectNonEmptyPlan`, `TestStep` also exposes an `ExpectError` hook, allowing developers to test configuration that they expect to produce an error, such as configuration that fails schema validators:

```

func TestAccExampleWidget_expectError(t *testing.T) {
    var widget example.Widget
    rName := acctest.RandStringFromCharSet(10, acctest.CharSetAlphaNum)

    resource.Test(t, resource.TestCase{
        PreCheck: func() { testAccPreCheck(t) },
        Providers: testAccProviders,
        CheckDestroy: testAccCheckExampleResourceDestroy,
        Steps: []resource.TestStep{
            {
                // use a configuration that we expect to fail a validator
                // on the resource Name attribute, which only allows alphanumeric
                // characters
                Config: testAccExampleResourceError(rName + "*$%^"),
                // No check function is given because we expect this configuration
                // to fail before any infrastructure is created
                ExpectError: regexp.MustCompile("Widget names may only contain alphanumeric characters"),
            },
        },
    })
}

```

`ExpectError` expects a valid regular expression, and the error message must match in order to consider the error as expected and allow the test to pass. If the regular expression does not match, the `TestStep` fails explaining that the configuration did not produce the error expected.

[Back to top](#)

Regression tests

As resources are put into use, issues can arise as bugs that need to be fixed and released in a new version. Developers are encouraged to introduce regression tests that demonstrate not only any bugs reported, but that code modified to address any bug is verified as fixing the issues. These regression tests should be named and documented appropriately to identify the issue(s) they demonstrate fixes for. When possible the documentation for a regression test should include a link to the original bug report.

An ideal bug fix would include at least 2 commits to source control:

A single commit introducing the regression test, verifying the issue(s)
1 or more commits that modify code to fix the issue(s)

This allows other developers to independently verify that a regression test indeed reproduces the issue by checking out the source at that commit first, and then advancing the revisions to evaluate the fix.

[Back to top](#)

Conclusion

Terraform's Testing Framework (</docs/extend/testing/index.html>) allows for powerful, iterative acceptance tests that enable developers to fully test the behavior of Terraform plugins. By following the above best practices, developers can ensure their plugin behaviors correctly across the most common use cases and everyday operations users will have using their plugins, and ensure that Terraform remains a world-class tool for safely managing infrastructure.

Versioning and Changelog

Given the breadth of available Terraform plugins, ensuring a consistent experience across them requires a standard guideline for compatibility promises. These guidelines are enforced for plugins released by HashiCorp and are recommended for all community plugins.

Versioning Specification

Observing that Terraform plugins are in many ways analogous to shared libraries in a programming language, we adopted a version numbering scheme that follows the guidelines of Semantic Versioning (<http://semver.org/>). In summary, this means that with a version number of the form MAJOR.MINOR.PATCH, the following meanings apply:

- Increasing only the patch number suggests that the release includes only bug fixes, and is intended to be functionally equivalent.
- Increasing the minor number suggests that new features have been added but that existing functionality remains broadly compatible.
- Increasing the major number indicates that significant breaking changes have been made, and thus extra care or attention is required during an upgrade.

Version numbers above 1.0.0 signify stronger compatibility guarantees, based on the rules above. Each increasing level can also contain changes of the lower level (e.g. MINOR can contain PATCH changes).

Example Major Number Increments

Increasing the MAJOR number is intended to signify potentially breaking changes.

Within Terraform provider development, some examples include:

- Removing a resource or data source
- Removing an attribute
- Renaming a resource or data source
- Renaming an attribute
- Changing fundamental provider behaviors (e.g. authentication or configuration precedence)
- Changing resource import ID format
- Changing resource ID format
- Changing attribute type where the new type is functionally incompatible (e.g. TypeSet to TypeList)
- Changing attribute format (e.g. changing a timestamp from epoch time to a string)

Example Minor Number Increments

MINOR increments are intended to signify the availability of new functionality or deprecations of existing functionality without breaking changes to the previous version.

Within Terraform provider development, some examples include:

- Marking a resource or data source as deprecated
- Marking an attribute as deprecated
- Adding a new resource or data source
- Aliasing an existing resource or data source
- Implementing new attributes within the provider configuration or an existing resource or data source
- Implementing new validation within an existing resource or data source
- Changing attribute type where the new type is functionally compatible (e.g. `TypeList` to `TypeSet`)

Example Patch Number Increments

Increasing the PATCH number is intended to signify mainly bug fixes and to be functionally equivalent with the previous version.

Within Terraform provider development, some examples include:

- Fixing an interaction with the remote API or Terraform state drift detection (e.g. broken create, read, update, or delete functionality)
- Fixing attributes to match behavior with resource code (e.g. removing `Optional` when an attribute can not be configured in the remote API)
- Fixing attributes to match behavior with the remote API (e.g. changing `Required` to `Optional`, fixing validation)

Changelog Specification

For better operator experience, we provide a standardized format so development information is available across all providers consistently. The changelog should live in a top level file in the project, named `CHANGELOG` or `CHANGELOG.md`. We generally recommend that the changelog is updated outside of pull requests unless a clear process is setup for handling merge conflicts.

Version Headers

The upcoming release version number is always at the top of the file and is marked specifically as `(Unreleased)`, with other previously released versions below.

NOTE: For HashiCorp released providers, the release process will replace the "Unreleased" header with the current date. This line must be present with the target release version to successfully release that version.

```
## X.Y.Z (Unreleased)
```

```
...
```

```
## A.B.C (Month Day, Year)
```

```
...
```

Categorization

Information in the changelog should be broken down as follows:

- **BACKWARDS INCOMPATIBILITIES** or **BREAKING CHANGES**: This section documents in brief any incompatible changes and how to handle them. This should only be present in major version upgrades.
- **NOTES**: Additional information for potentially unexpected upgrade behavior, upcoming deprecations, or to highlight very important crash fixes (e.g. due to upstream API changes)
- **FEATURES**: These are major new improvements that deserve a special highlight, such as a new resource or data source.
- **IMPROVEMENTS** or **ENHANCEMENTS**: Smaller features added to the project such as a new attribute for a resource.
- **BUG FIXES**: Any bugs that were fixed.

These should be displayed as left aligned text with new lines above and below:

```
CATEGORY:
```

Entry Format

Each entry under a category should use the following format:

```
* subsystem: Descriptive message [GH-1234]
```

For provider development typically the "subsystem" is the resource or data source affected e.g. `resource/load_balancer`, or `provider` if the change affects whole provider (e.g. authentication logic). Each bullet also references the corresponding pull request number that contained the code changes, in the format of `[GH-####]` (for HashiCorp released plugins, this will be automatically updated on release).

Entry Ordering

To order entries, these basic rules should be followed:

1. If large cross-cutting changes are present, list them first (e.g. provider)

2. Order other entries lexicographically based on subsystem (e.g. resource/load_balancer then resource/subnet)

Example Changelog

```
## 1.0.0 (Unreleased)

BREAKING CHANGES:
* Resource `network_port` has been removed [GH-1]

FEATURES:
* **New Resource:** `cluster` [GH-43]

IMPROVEMENTS:
* resource/load_balancer: Add `ATTRIBUTE` argument (support X new functionality) [GH-12]
* resource/subnet: Now better [GH-22, GH-32]

## 0.2.0 (Month Day, Year)

FEATURES:
...
...
```

Terraform Community

Terraform is a mature project with a growing community. There are active, dedicated people willing to help you through various mediums.

Events: HashiCorp sponsors various community events to bring together Terraform ecosystem developers. See the events (</docs/extend/community/events/index.html>) page for more information.

Stack Exchange: Terraform questions often get asked and answered on Server Fault (<https://serverfault.com/>) and Stack Overflow (<https://stackoverflow.com/>). Use the tag "terraform" to help your question be found by Terraform experts, and please be respectful of the "How to Ask" guidelines in each community.

Mailing list: Terraform Google Group (<https://groups.google.com/group/terraform-tool>)

Gitter: Terraform Gitter Room (<https://gitter.im/hashicorp-terraform/Lobby>)

IRC: Use the Gitter IRC bridge (<https://irc.gitter.im>)

Bug Trackers:

- Terraform Core Issue tracker on GitHub (<https://github.com/hashicorp/terraform/issues>). Please only use this for reporting bugs. Do not ask for general help here; use a Stack Exchange community, Gitter chat, or the mailing list for that.
- Terraform Providers distributed by HashiCorp are part of the `terraform-providers` (<https://github.com/terraform-providers>) GitHub Organization. Each Provider is a separate GitHub project with their own issue trackers.

Training: Paid HashiCorp training courses (<https://www.hashicorp.com/training.html>) are also available in a city near you. Private training courses are also available.

Terraform Developers Code of Conduct

HashiCorp Community Guidelines apply to you when interacting with the community while working on Terraform Providers.

Please read the full text at <https://www.hashicorp.com/community-guidelines> (<https://www.hashicorp.com/community-guidelines>)

Contributing to Terraform

Terraform and its diverse collection of plugins are a collaborative work between HashiCorp employees, independent cloud vendors, and a large open source community. HashiCorp and the Terraform team truly value every issue, pull request, and feature request received on any of its many GitHub repositories. The only prerequisite to contributing to Terraform is an interest to improve the project!

While you do not need to be an expert in any of the following, these are things that are helpful to have or know if you're wanting to contribute more:

- Basic knowledge of Terraform. If you're new to Terraform, please see our introduction documentation (</intro/index.html>).
- Basic programming knowledge. Terraform and Terraform Plugins are written in the Go programming language (<https://golang.org>), but even if you've never written a line of Go before, you're still welcome to take a dive into the code and submit patches. The community is happy to assist with code reviews and offer guidance specific to Go.
- Infrastructure as Code. If this is a new term for you, check out Infrastructure as Code (https://en.wikipedia.org/wiki/Infrastructure_as_Code) on Wikipedia for a brief introduction. Our Getting Started guide (<https://learn.hashicorp.com/terraform/getting-started/install>) is a great way to get started as well.

HashiCorp vs. Community Providers

We separate providers out into what we call "HashiCorp Providers" and "Community Providers".

HashiCorp providers are providers that we'll dedicate full time resources to improving, supporting the latest features, and fixing bugs. These are providers we understand deeply and are confident we have the resources to manage ourselves.

Community providers are providers where we depend on the community to contribute fixes and enhancements to improve. HashiCorp will run automated tests and ensure these providers continue to work, but will not dedicate full time resources to add new features to these providers. These providers are available in official Terraform releases, but the functionality is primarily contributed.

The current list of HashiCorp Providers is as follows:

- Amazon Web Services (<https://github.com/terraform-providers/terraform-provider-aws>)
- Azure RM (<https://github.com/terraform-providers/terraform-provider-azurerm>)
- Google Cloud Platform (<https://github.com/terraform-providers/terraform-provider-google>)
- Oracle Public Cloud (<https://github.com/terraform-providers/terraform-provider-opc>)

Our testing standards are the same for both HashiCorp and Community providers, and HashiCorp runs full acceptance test suites for every provider nightly to ensure Terraform remains stable.

We make the distinction between these two types of providers to help highlight the vast amounts of community effort that goes in to making Terraform great, and to help contributors better understand the role HashiCorp employees play in the various areas of the code base.

Issues

Issue Reporting Checklists

We welcome issues of all kinds including feature requests, bug reports, and general questions. The code and issue tracker for Terraform is at <https://github.com/hashicorp/terraform> (<https://github.com/hashicorp/terraform>). Each officially supported Terraform Provider has its own GitHub repository in the Terraform Providers (<https://github.com/terraform-providers>) GitHub Organization.

Below you'll find checklists with guidelines for well-formed issues of each type.

Bug Reports

- [] **Test against latest release:** Make sure you test against the latest released version. It is possible we already fixed the bug you're experiencing.
- [] **Search for possible duplicate reports:** It's helpful to keep bug reports consolidated to one thread, so do a quick search on existing bug reports to check if anybody else has reported the same thing. You can scope searches by the label "bug" to help narrow things down.
- [] **Include steps to reproduce:** Provide steps to reproduce the issue, along with your .tf files, with secrets removed, so we can try to reproduce it. Without this, it makes it much harder to fix the issue.
- [] **For panics, include crash.log:** If you experienced a panic, please create a gist (<https://gist.github.com>) of the *entire* generated crash log for us to look at. Double check no sensitive items were in the log.

Feature Requests

- [] **Search for possible duplicate requests:** It's helpful to keep requests consolidated to one thread, so do a quick search on existing requests to check if anybody else has reported the same thing. You can scope searches by the label "enhancement" to help narrow things down.
- [] **Include a use case description:** In addition to describing the behavior of the feature you'd like to see added, it's helpful to also lay out the reason why the feature would be important and how it would benefit Terraform users.

Questions

- [] **Search for answers in Terraform documentation:** We're happy to answer questions in GitHub Issues, but it helps reduce issue churn and maintainer workload if you work to find answers to common questions in the documentation. Often times Question issues result in documentation updates to help future users, so if you don't find an answer, you can give us pointers for where you'd expect to see it in the docs.

Issue Lifecycle

1. The issue is reported.
2. The issue is verified and categorized by a Terraform collaborator. Categorization is done via GitHub labels. We generally use a two-label system of (1) issue/PR type, and (2) section of the codebase. Type is usually "bug", "enhancement", "documentation", or "question", and section can be any of the providers or provisioners or "core".

3. Unless it is critical, the issue is left for a period of time (sometimes many weeks), giving outside contributors a chance to address the issue.
4. The issue is addressed in a pull request or commit. The issue will be referenced in the commit message so that the code that fixes it is clearly linked.
5. The issue is closed. Sometimes, valid issues will be closed to keep the issue tracker clean. The issue is still indexed and available for future viewers, or can be re-opened if necessary.

Pull Requests

Thank you for contributing! Here you'll find information on what to include in your Pull Request to ensure it is accepted quickly.

- For pull requests that follow the guidelines, we expect to be able to review and merge very quickly.
- Pull requests that don't follow the guidelines will be annotated with what they're missing. A community or core team member may be able to swing around and help finish up the work, but these PRs will generally hang out much longer until they can be completed and merged.

Pull Request Lifecycle

1. You are welcome to submit your pull request for commentary or review before it is fully completed. Please prefix the title of your pull request with "[WIP]" to indicate this. It's also a good idea to include specific questions or items you'd like feedback on.
2. Once you believe your pull request is ready to be merged, you can remove any "[WIP]" prefix from the title and a core team member will review. Follow the checklists below to help ensure that your contribution will be merged quickly.
3. One of Terraform's core team members will look over your contribution and either provide comments letting you know if there is anything left to do. We do our best to provide feedback in a timely manner, but it may take some time for us to respond.
4. Once all outstanding comments and checklist items have been addressed, your contribution will be merged! Merged PRs will be included in the next Terraform release. The core team takes care of updating the CHANGELOG as they merge.
5. In rare cases, we might decide that a PR should be closed. We'll make sure to provide clear reasoning when this happens.

Checklists for Contribution

There are several different kinds of contribution, each of which has its own standards for a speedy review. The following sections describe guidelines for each type of contribution.

Documentation Update

Because Terraform's website (<https://github.com/hashicorp/terraform/tree/master/website>) is in the same repo as the code, it's easy for anybody to help us improve our docs.

- [] **Reasoning for docs update:** Including a quick explanation for why the update needed is helpful for reviewers.
- [] **Relevant Terraform version:** Is this update worth deploying to the site immediately, or is it referencing an upcoming version of Terraform and should get pushed out with the next release?

As mentioned above, each Terraform Provider has its own code repository in the Terraform Providers (<https://github.com/terraform-providers>). Each repository has its own folder named `website` that contains that Providers documentation. Updates for documentation for a specific provider should be reported or posted there.

Enhancement/Bugfix to a Resource

Working on existing resources is a great way to get started as a Terraform contributor because you can work within existing code and tests to get a feel for what to do.

- [] **Acceptance test coverage of new behavior:** Existing resources each have a set of acceptance tests (<https://github.com/hashicorp/terraform#acceptance-tests>) covering their functionality. These tests should exercise all the behavior of the resource. Whether you are adding something or fixing a bug, the idea is to have an acceptance test that fails if your code were to be removed. Sometimes it is sufficient to "enhance" an existing test by adding an assertion or tweaking the config that is used, but often a new test is better to add. You can copy/paste an existing test and follow the conventions you see there, modifying the test to exercise the behavior of your code.
- [] **Documentation updates:** If your code makes any changes that need to be documented, you should include those doc updates in the same PR. The Terraform website (<https://github.com/hashicorp/terraform/tree/master/website>) source is in this repo and includes instructions for getting a local copy of the site up and running if you'd like to preview your changes.
- [] **Well-formed Code:** Do your best to follow existing conventions you see in the codebase, and ensure your code is formatted with `go fmt`. (The Travis CI build will fail if `go fmt` has not been run on incoming code.) The PR reviewers can help out on this front, and may provide comments with suggestions on how to improve the code.

New Resource

Implementing a new resource is a good way to learn more about how Terraform interacts with upstream APIs. There are plenty of examples to draw from in the existing resources, but you still get to implement something completely new.

- [] **Minimal LOC:** It can be inefficient for both the reviewer and author to go through long feedback cycles on a big PR with many resources. We therefore encourage you to only submit **1 resource at a time**.
- [] **Acceptance tests:** New resources should include acceptance tests covering their behavior. See Writing Acceptance Tests below for a detailed guide on how to approach these.
- [] **Documentation:** Each resource gets a page in the Terraform documentation. The Terraform website (<https://github.com/hashicorp/terraform/tree/master/website>) source is in this repo and includes instructions for getting a local copy of the site up and running if you'd like to preview your changes. For a resource, you'll want to add a new file in the appropriate place and add a link to the sidebar for that page.
- [] **Well-formed Code:** Do your best to follow existing conventions you see in the codebase, and ensure your code is formatted with `go fmt`. (The Travis CI build will fail if `go fmt` has not been run on incoming code.) The PR reviewers can help out on this front, and may provide comments with suggestions on how to improve the code.

New Provider

Implementing a new provider gives Terraform the ability to manage resources in a whole new API. It's a larger undertaking, but brings major new functionality into Terraform.

- [] **Minimal initial LOC:** Some providers may be big, and it can be inefficient for both reviewer & author to go through long feedback cycles on a big PR with many resources. We encourage you to only submit the necessary minimum in a single PR, ideally **just the first resource** of the provider.
- [] **Acceptance tests:** Each provider should include an acceptance test suite with tests for each resource should include acceptance tests covering its behavior. See Writing Acceptance Tests below for a detailed guide on how to approach these.
- [] **Documentation:** Each provider has a section in the Terraform documentation. The Terraform website (<https://github.com/hashicorp/terraform/tree/master/website>) source is in this repo and includes instructions for getting a local copy of the site up and running if you'd like to preview your changes. For a provider, you'll want to add new index file and individual pages for each resource.
- [] **Well-formed Code:** Do your best to follow existing conventions you see in the codebase, and ensure your code is formatted with `go fmt`. (The Travis CI build will fail if `go fmt` has not been run on incoming code.) The PR reviewers can help out on this front, and may provide comments with suggestions on how to improve the code.

Core Bugfix/Enhancement

We are always happy when any developer is interested in diving into Terraform's core to help out! Here's what we look for in smaller Core PRs.

- [] **Unit tests:** Terraform's core is covered by hundreds of unit tests at several different layers of abstraction. Generally the best place to start is with a "Context Test". These are higher level test that interact end-to-end with most of Terraform's core. They are divided into test files for each major action (plan, apply, etc.). Getting a failing test is a great way to prove out a bug report or a new enhancement. With a context test in place, you can work on implementation and lower level unit tests. Lower level tests are largely context dependent, but the Context Tests are almost always part of core work.
- [] **Documentation updates:** If the core change involves anything that needs to be reflected in our documentation, you can make those changes in the same PR. The Terraform website (<https://github.com/hashicorp/terraform/tree/master/website>) source is in this repo and includes instructions for getting a local copy of the site up and running if you'd like to preview your changes.
- [] **Well-formed Code:** Do your best to follow existing conventions you see in the codebase, and ensure your code is formatted with `go fmt`. (The Travis CI build will fail if `go fmt` has not been run on incoming code.) The PR reviewers can help out on this front, and may provide comments with suggestions on how to improve the code.

Core Feature

If you're interested in taking on a larger core feature, it's a good idea to get feedback early and often on the effort.

- [] **Early validation of idea and implementation plan:** Terraform's core is complicated enough that there are often several ways to implement something, each of which has different implications and tradeoffs. Working through a plan of attack with the team before you dive into implementation will help ensure that you're working in the right direction.
- [] **Unit tests:** Terraform's core is covered by hundreds of unit tests at several different layers of abstraction. Generally

the best place to start is with a "Context Test". These are higher level test that interact end-to-end with most of Terraform's core. They are divided into test files for each major action (plan, apply, etc.). Getting a failing test is a great way to prove out a bug report or a new enhancement. With a context test in place, you can work on implementation and lower level unit tests. Lower level tests are largely context dependent, but the Context Tests are almost always part of core work.

- [] **Documentation updates:** If the core change involves anything that needs to be reflected in our documentation, you can make those changes in the same PR. The Terraform website (<https://github.com/hashicorp/terraform/tree/master/website>) source is in this repo and includes instructions for getting a local copy of the site up and running if you'd like to preview your changes.
- [] **Well-formed Code:** Do your best to follow existing conventions you see in the codebase, and ensure your code is formatted with `go fmt`. (The Travis CI build will fail if `go fmt` has not been run on incoming code.) The PR reviewers can help out on this front, and may provide comments with suggestions on how to improve the code.

Writing Acceptance Tests

Terraform includes an acceptance test harness that does most of the repetitive work involved in testing a resource.

Acceptance Tests Often Cost Money to Run

Because acceptance tests create real resources, they often cost money to run. Because the resources only exist for a short period of time, the total amount of money required is usually relatively small. Nevertheless, we don't want financial limitations to be a barrier to contribution, so if you are unable to pay to run acceptance tests for your contribution, simply mention this in your pull request. We will happily accept "best effort" implementations of acceptance tests and run them for you on our side. This might mean that your PR takes a bit longer to merge, but it most definitely is not a blocker for contributions.

Running an Acceptance Test

Acceptance tests can be run using the `testacc` target in the Terraform Makefile. The individual tests to run can be controlled using a regular expression. Prior to running the tests provider configuration details such as access keys must be made available as environment variables.

For example, to run an acceptance test against the Azure Resource Manager provider, the following environment variables must be set:

```
export ARM_SUBSCRIPTION_ID=...
export ARM_CLIENT_ID=...
export ARM_CLIENT_SECRET=...
export ARM_TENANT_ID=...
```

Tests can then be run by specifying the target provider and a regular expression defining the tests to run:

```
$ make testacc TEST=./builtin/providers/azurerm TESTARGS=' -run=TestAccAzureRMPublicIpStatic_update'
==> Checking that code complies with gofmt requirements...
go generate ./...
TF_ACC=1 go test ./builtin/providers/azurerm -v -run=TestAccAzureRMPublicIpStatic_update -timeout 120m
--- RUN TestAccAzureRMPublicIpStatic_update
--- PASS: TestAccAzureRMPublicIpStatic_update (177.48s)
PASS
ok    github.com/hashicorp/terraform/builtin/providers/azurerm    177.504s
```

Entire resource test suites can be targeted by using the naming convention to write the regular expression. For example, to run all tests of the azurerm_public_ip resource rather than just the update test, you can start testing like this:

```
$ make testacc TEST=./builtin/providers/azurerm TESTARGS=' -run=TestAccAzureRMPublicIpStatic'
==> Checking that code complies with gofmt requirements...
go generate ./...
TF_ACC=1 go test ./builtin/providers/azurerm -v -run=TestAccAzureRMPublicIpStatic -timeout 120m
--- RUN TestAccAzureRMPublicIpStatic_basic
--- PASS: TestAccAzureRMPublicIpStatic_basic (137.74s)
--- RUN TestAccAzureRMPublicIpStatic_update
--- PASS: TestAccAzureRMPublicIpStatic_update (180.63s)
PASS
ok    github.com/hashicorp/terraform/builtin/providers/azurerm    318.392s
```

Writing an Acceptance Test

Terraform has a framework for writing acceptance tests which minimizes the amount of boilerplate code necessary to use common testing patterns. The entry point to the framework is the `resource.Test()` function.

Tests are divided into TestSteps. Each TestStep proceeds by applying some Terraform configuration using the provider under test, and then verifying that results are as expected by making assertions using the provider API. It is common for a single test function to exercise both the creation of and updates to a single resource. Most tests follow a similar structure.

1. Pre-flight checks are made to ensure that sufficient provider configuration is available to be able to proceed - for example in an acceptance test targeting AWS, `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` must be set prior to running acceptance tests. This is common to all tests exercising a single provider.

Each TestStep is defined in the call to `resource.Test()`. Most assertion functions are defined out of band with the tests. This keeps the tests readable, and allows reuse of assertion functions across different tests of the same type of resource. The definition of a complete test looks like this:

```

func TestAccAzureRMPublicIpStatic_update(t *testing.T) {
    resource.Test(t, resource.TestCase{
        PreCheck:     func() { testAccPreCheck(t) },
        Providers:    testAccProviders,
        CheckDestroy: testCheckAzureRMPublicIpDestroy,
        Steps: []resource.TestStep{
            resource.TestStep{
                Config: testAccAzureRMVPublicIpStatic_basic,
                Check: resource.ComposeTestCheckFunc(
                    testCheckAzureRMPublicIpExists("azurerm_public_ip.test"),
                ),
            },
        },
    })
}

```

When executing the test, the following steps are taken for each TestStep:

1. The Terraform configuration required for the test is applied. This is responsible for configuring the resource under test, and any dependencies it may have. For example, to test the `azurerm_public_ip` resource, an `azurerm_resource_group` is required. This results in configuration which looks like this:

```

resource "azurerm_resource_group" "test" {
    name = "acceptanceTestResourceGroup1"
    location = "West US"
}

resource "azurerm_public_ip" "test" {
    name = "acceptanceTestPublicIp1"
    location = "West US"
    resource_group_name = "${azurerm_resource_group.test.name}"
    public_ip_address_allocation = "static"
}

```

2. Assertions are run using the provider API. These use the provider API directly rather than asserting against the resource state. For example, to verify that the `azurerm_public_ip` described above was created successfully, a test function like this is used:

```

func testCheckAzureRMPublicIpExists(name string) resource.TestCheckFunc {
    return func(s *terraform.State) error {
        // Ensure we have enough information in state to look up in API
        rs, ok := s.RootModule().Resources[name]
        if !ok {
            return fmt.Errorf("Not found: %s", name)
        }

        publicIPName := rs.Primary.Attributes["name"]
        resourceGroup, hasResourceGroup := rs.Primary.Attributes["resource_group_name"]
        if !hasResourceGroup {
            return fmt.Errorf("Bad: no resource group found in state for public ip: %s", availSetName)
        }

        conn := testAccProvider.Meta().(*ArmClient).publicIPClient

        resp, err := conn.Get(resourceGroup, publicIPName, "")
        if err != nil {
            return fmt.Errorf("Bad: Get on publicIPClient: %s", err)
        }

        if resp.StatusCode == http.StatusNotFound {
            return fmt.Errorf("Bad: Public IP %q (resource group: %q) does not exist", name, resourceGroup)
        }

        return nil
    }
}

```

Notice that the only information used from the Terraform state is the ID of the resource - though in this case it is necessary to split the ID into constituent parts in order to use the provider API. For computed properties, we instead assert that the value saved in the Terraform state was the expected value if possible. The testing framework provides helper functions for several common types of check - for example:

```

```go
resource.TestCheckResourceAttr("azurerm_public_ip.test", "domain_name_label", "mylabel01"),
```

```

1. The resources created by the test are destroyed. This step happens automatically, and is the equivalent of calling `terraform destroy`.
2. Assertions are made against the provider API to verify that the resources have indeed been removed. If these checks fail, the test fails and reports "dangling resources". The code to ensure that the `azurerm_public_ip` shown above looks like this:

```
func testCheckAzureRMPublicIpDestroy(s *terraform.State) error {
    conn := testAccProvider.Meta().(*ArmClient).publicIPClient

    for _, rs := range s.RootModule().Resources {
        if rs.Type != "azurerm_public_ip" {
            continue
        }

        name := rs.Primary.Attributes["name"]
        resourceGroup := rs.Primary.Attributes["resource_group_name"]

        resp, err := conn.Get(resourceGroup, name, "")

        if err != nil {
            return nil
        }

        if resp.StatusCode != http.StatusNotFound {
            return fmt.Errorf("Public IP still exists:\n%#v", resp.Properties)
        }
    }

    return nil
}
```

These functions usually test only for the resource directly under test: we skip the check that the azurerm_resource_group has been destroyed when testing azurerm_resource_group, under the assumption that azurerm_resource_group is tested independently in its own acceptance tests.

Terraform Ecosystem Community Events

The Terraform development ecosystem has amazing contributors and community members. HashiCorp sponsors various community events to bring together Terraform developers of all experience for realtime collaboration and growth.

Upcoming Events

Additional upcoming events including webinars and HashiCorp User Groups can be found on the HashiCorp Events (<https://www.hashicorp.com/events>) page.

- Community Gardening - Fall 2018 (</docs/extend/community/events/2018/fall-gardening.html>) (September 12, 2018) - Terraform Azure provider gardening event.
- HashiConf 2018 (<https://www.hashiconf.com/>) (October 22-24, 2018) - Annual HashiCorp user and technology conference. Three days of hands-on product training, keynotes, technical talks, and one-on-one time with the HashiCorp developers.

Past Events

- Community Gardening - Summer 2018 (</docs/extend/community/events/2018/summer-gardening.html>) (June 21, 2018) - Terraform AWS provider gardening event.

Terraform Ecosystem Community Events - 2018 Fall Gardening

HashiCorp does quarterly community gardening sessions for Terraform providers. Our second is September 12, 2018 for Azure. It is an all virtual event across time zones. As a user, you will be able to ask HashiCorp employees Azure Terraform provider questions in our Slack room (<https://terraform-azure.slack.com/>) (request invite (https://terraform-azure.slack.com/join/shared_invite/enQtNDMzNjQ5NzcxMDc3LTJkZTjhNTg3NTE5ZTdjZjFhMThmMTVmOTg5YWjkMDU1YTMzN2YyOWJmZGM3MGl4OTQ0ODQxNTEyNjdjMDAxMjM)). The concept is based on the Go Gardening (<https://github.com/golang/go/wiki/Gardening>) idea.

What are we planning to do?

- Weeding
- New resources/data sources
- Enhancements to existing resources/data sources
- Bug fixes
- Documentation
- Verifying and adding tests to existing resources/data sources

Why we love hosting these?

- We want to get our community involved and introduced to each other.
- This allows people with all levels to participate from the small, medium to larger infrastructures. As well as the beginners having an opportunity to get positive support from more advanced users.

The Slack room is open and available for conversations among participants, drop ins, all while someone from HashiCorp will be available throughout the time zones.

Anything marked HashiCorp in Progress please reach out first via Slack please.

Please send all good, bad, and could be better feedback to beth@hashicorp.com (<mailto:beth@hashicorp.com>) so that we can make improvements on a continuous basis.

We really appreciate the communities feedback, help and participation. We also want to make sure we are bring you along with our roadmap and listening to your feedback about what business problems you are running into so that we can adjust quicker.

Terraform Ecosystem Community Events - 2018 Summer Gardening

On June 21, 2018 the Terraform AWS team is going to put together a virtual, all time zones (best we can), community gardening session for the Terraform AWS provider, inspired by Go's gardening concept (<https://github.com/golang/go/wiki/Gardening>). We plan to do this for other Terraform providers and OSS projects eventually.

We have some amazing contributors to AWS and want to get them more involved. This allows people with all levels of AWS knowledge to participate from small, medium, and larger infrastructures. Beginners will have an opportunity to get positive support from more advanced users and HashiCorp will provide participants for quick knowledge.

What to Expect

What do we plan to do first?

- Weeding

Why are we starting here?

- We currently have 1600 issues for AWS.
- There are certainly duplicates
- Bugs that have been fixed
- Bugs that need to be fixed that we do not have time for due to volume

What we are giving away for participating?

- Sweatshirts for the top 5 people who close the most issues via bug closure or finding duplicates and closing issues
- T-shirts for the next top 10 people who close the most issues via bug closure or finding duplicates and closing issues
- HashiCorp mug for the next top 10 who close the most issues via bug closure or finding duplicates and closing issues
- Stickers for everyone.

If you are going to be at HashiDays or HashiConf, let us know and we will set the goodies aside for you. Otherwise please feel free to reach out to Beth Fuller with your address and she'll send that to you.

Event Information

Realtime Communication

We will have a Gitter (<https://gitter.im/>) room available at `hashicorp-terraform/gardening` (<https://gitter.im/hashicorp-terraform/gardening>) and a someone from HashiCorp on Zoom conferencing (<https://zoom.us/>) through out the time zones.

Community Issue Selection

Anything marked HashiCorp in Progress please reach out first via the Gitter room or Zoom channel.

Feedback

AWS is our first step, as contributors to this project, we want your feedback on how the event went. Please send all good, bad, could be better feedback to beth@hashicorp.com (<mailto:beth@hashicorp.com>) so that we can make improvements on a continuous basis.

We really appreciate the communities feedback, help and participation. We also want to make sure we are bring you along with our roadmap and listening to your feedback about what business problems you are running into so that we can adjust quicker.

Provider Maintainers

HashiCorp tools are developed in open source with contributions from a wide variety of community members. A high value is placed on these individual contributions and different members can be grouped by the following definitions:

- Community member: Anyone using Terraform as a practitioner, or extending it as a developer.
- Contributor: Anyone who has added to the code repo, things like issues, votes, reactions, comments, and pull requests. If they have commits merged into the project, they can be recognized with a "contributor" badge on GitHub.
- Collaborator: Someone who has demonstrated the skills, abilities, and care for working on a particular project, such that we have asked and they have accepted responsibilities for oversight/work on a project. For providers maintained by the cloud vendor, these users are selected by the vendor and might even share most responsibilities of a "maintainer". On GitHub they can be recognized with a "collaborator" badge.
- Maintainer: Includes all previously described responsibilities but also plays a governance role on the project. On GitHub and in the context of Terraform Providers they can be recognized with an "owner" badge.

The overlap between "collaborator" and "maintainer" can fluctuate project to project; in fact, GitHub maintains no such distinction. Some providers have a clear HashiCorp maintainer role with a community of impactful collaborators donating their time to the project. Other providers can be a joint engineering effort between HashiCorp maintainers and outside collaborators the provider vendor has chosen. HashiCorp will generally avoid vetting external maintainers, but will help ensure existing and prospective maintainers have a shared understanding and acceptance of roles and responsibilities.

Community Guidelines

All community members are responsible for their interactions with the rest of the community, to keep Terraform and provider development successful it's important to follow our community guidelines (<https://www.hashicorp.com/community-guidelines>).

Provider Development Program

Providers in the development program get vetted and hosted in the terraform-providers GitHub organization. Please see the website (<https://www.terraform.io/guides/terraform-provider-development-program.html#checklist>) for instructions on following that process.

Maintainer Responsibilities

Being a maintainer of a provider involves several responsibilities from triaging issues and pull requests to editing the changelog and requesting for releases to be cut.

Pull Requests

Pull requests to a provider should feature relevant tests with instructions on running them, or output showing they have passed. For bug fixes, inclusion of repro instructions is helpful.

Maintainers are welcome to assess, reply, and label open issues/pull requests. We typically use a 2-label system of:

1. Type: such as "bug", "enhancement", "documentation", or "question"
2. Service/Section: in the case of AWS something like "service/ec2" or "service/rds"

Checklist

Pull requests should cover the following:

- Acceptance test coverage of new behavior: these tests should exercise all the behavior of the resource. Sometimes it is sufficient to "enhance" an existing test by adding an assertion or tweaking the config that is used, but often a new test is better. Tests should also pass with the TF_SCHEMA_PANIC_ON_ERROR environment variable set, ensuring the schema is properly configured and not silently failing.
- Documentation updates: All relevant changes (schema, unique behaviors) should be documented and be covered in the same pull request.
- Well-formed code: Ensure code passes linting (go fmt, go vet). Ensure code follows provider development best practices (<https://www.terraform.io/docs/extend/index.html>).
- Vendor updates: Updating the vendor folder should be done in a separate pull request. This is to avoid conflicts as the versions of dependencies tend to be fast moving targets.

Merging

If a pull request has been approved by a maintainer and the submitter has push privileges (recognizable via Collaborator or Member badge), the submitter should merge their own pull request. Any pull request that significantly changes or breaks user experience should always get at least one vote of approval from a HashiCorp employee prior to merge. We prefer to merge via the GitHub web interface using the green merge button. Squash when the commit history is irrelevant.

Versioning and Changelog

All providers should follow a consistent versioning scheme and changes should be documented in a CHANGELOG file. The specific changelog formatting required for HashiCorp released providers (due to the release process) and recommended for all community providers is outlined in our versioning best practices documentation (/docs/extend/best-practices/versioning.html).

To keep the CHANGELOG up to date, we recommend updating the CHANGELOG after every set of commits that change the project. Never include CHANGELOG updates in a pull request. If you do, it will very likely cause a merge conflict with other pull requests. Instead, make the pull request without CHANGELOG updates, and add to the CHANGELOG only after merge.

Release process

Release cadence will vary project to project. There is no explicit timeline for provider releases, we tend to evaluate each provider release based on a few factors such as the collection of small bug fixes and features or conversely one critical fix or highly demanded feature. Releases can be requested in our slack channel. Smaller providers typically have no consistent cadence, larger ones such as AWS have leaned towards weekly releases. We feel this is probably the most frequent a project should cut releases.

Our release process is automated. That process tags, cross compiles, signs, and uploads the binaries to our release site (<https://releases.hashicorp.com/>), the list of our official providers can be found on [terraform.io](https://www.terraform.io/docs/providers/index.html) (<https://www.terraform.io/docs/providers/index.html>). HashiCorp handles all releases internally, and requests for provider releases can be made in the #committers-terraform Slack channel by mentioning @provider-releases. The employee cutting the release will notify maintainers via Slack before and after release so pull request merging can be avoided during that interval. We compile binaries for all platforms & architectures supported by Terraform core (<https://www.terraform.io/downloads.html>) (currently MacOS, FreeBSD, Linux, OpenBSD, Solaris, Windows). Exceptions can be made if there's a strong reason a provider cannot be compiled for any of these. Such cases have to be discussed with a HashiCorp employee.

How Terraform Works

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform is built on a plugin-based architecture, enabling developers to extend Terraform by writing new plugins or compiling modified versions of existing plugins.

Terraform is logically split into two main parts: **Terraform Core** and **Terraform Plugins**. Terraform Core uses remote procedure calls (RPC) to communicate with Terraform Plugins, and offers multiple ways to discover and load plugins to use. Terraform Plugins expose an implementation for a specific service, such as AWS, or provisioner, such as bash.

Terraform Core

Terraform Core is a statically-compiled binary (https://en.wikipedia.org/wiki/Static_build#Static_building) written in the Go programming language (<https://golang.org/>). The compiled binary is the command line tool (CLI) `terraform`, the entrypoint for anyone using Terraform. The code is open source and hosted at github.com/hashicorp/terraform.

The primary responsibilities of Terraform Core are:

- Infrastructure as code: reading and interpolating configuration files and modules
- Resource state management
- Construction of the Resource Graph (/docs/internals/graph.html)
- Plan execution
- Communication with plugins over RPC

Terraform Plugins

Terraform Plugins are written in Go and are executable binaries invoked by Terraform Core over RPC. Each plugin exposes an implementation for a specific service, such as AWS, or provisioner, such as bash. All Providers and Provisioners used in Terraform configurations are plugins. They are executed as a separate process and communicate with the main Terraform binary over an RPC interface. Terraform has several Provisioners built-in, while Providers are discovered dynamically as needed (See **Discovery** below). Terraform Core provides a high-level framework that abstracts away the details of plugin discovery and RPC communication so developers do not need to manage either.

Terraform Plugins are responsible for the domain specific implementation of their type.

The primary responsibilities of Provider Plugins are:

- Initialization of any included libraries used to make API calls
- Authentication with the Infrastructure Provider
- Define Resources that map to specific Services

The primary responsibilities of Provisioner Plugins are:

- Executing commands or scripts on the designated Resource after creation, or on destruction.

Discovery

Advanced topic: This section describes Terraform's plugin discovery behavior at the level of detail a plugin developer might need. For instructions suited to normal Terraform use, see Configuring Providers (/docs/configuration/providers.html).

When `terraform init` is run, Terraform reads configuration files in the working directory to determine which plugins are necessary, searches for installed plugins in several locations, sometimes downloads additional plugins, decides which plugin versions to use, and writes a lock file to ensure Terraform will use the same plugin versions in this directory until `terraform init` runs again.

For purposes of discovery, Terraform plugins behave in one of three ways:

| Kind | Behavior |
|--|--|
| Built-in provisioners | Always available; included in Terraform binary. |
| Providers distributed by HashiCorp | Automatically downloaded if not already installed. |
| Third-party providers and provisioners | Must be manually installed. |

Plugin Locations

Note: Third-party plugins should usually be installed in the user plugins directory, which is located at `~/.terraform.d/plugins` on most operating systems and `%APPDATA%\terraform.d\plugins` on Windows.

By default, `terraform init` searches the following directories for plugins. Some of these directories are static, and some are relative to the current working directory.

| Directory | Purpose |
|--|---|
| . | For convenience during plugin development. |
| Location of the <code>terraform</code> binary
(<code>/usr/local/bin</code> , for example.) | For airgapped installations; see <code>terraform bundle</code> (https://github.com/hashicorp/terraform/tree/master/tools/terraform-bundle). |
| <code>terraform.d/plugins/<OS>_<ARCH></code> | For checking custom providers into a configuration's VCS repository. Not usually desirable, but sometimes necessary in Terraform Enterprise. |
| <code>.terraform/plugins/<OS>_<ARCH></code> | Automatically downloaded providers. |
| <code>~/.terraform.d/plugins</code> or
<code>%APPDATA%\terraform.d\plugins</code> | The user plugins directory. |

| Directory | Purpose |
|--|--|
| <code>~/.terraform.d/plugins/<OS>_<ARCH></code> or
<code>%APPDATA%\terraform.d\plugins\<OS>_<ARCH></code> | The user plugins directory, with explicit OS and architecture. |

Note: <OS> and <ARCH> use the Go language's standard OS and architecture names; for example, `darwin_amd64`.

If `terraform init` is run with the `-plugin-dir=<PATH>` option (with a non-empty <PATH>), it overrides the default plugin locations and searches only the specified path.

Provider and provisioner plugins can be installed in the same directories. Provider plugin binaries are named with the scheme `terraform-provider-<NAME>_vx.Y.Z`, while provisioner plugins use the scheme `terraform-provisioner-<NAME>_vx.Y.Z`. Terraform relies on filenames to determine plugin types, names, and versions.

Selecting Plugins

After locating any installed plugins, `terraform init` compares them to the configuration's version constraints ([/docs/configuration/providers.html#provider-versions](#)) and chooses a version for each plugin as follows:

- If any acceptable versions are installed, Terraform uses the newest *installed* version that meets the constraint (even if [releases.hashicorp.com](#) has a newer acceptable version).
- If no acceptable versions are installed and the plugin is one of the providers distributed by HashiCorp ([/docs/providers/index.html](#)), Terraform downloads the newest acceptable version from [releases.hashicorp.com](#) and saves it in `.terraform/plugins/<OS>_<ARCH>`.
 - This step is skipped if `terraform init` is run with the `-plugin-dir=<PATH>` or `-get-plugins=false` options.
- If no acceptable versions are installed and the plugin is not distributed by HashiCorp, initialization fails and the user must manually install an appropriate version.

Upgrading Plugins

When `terraform init` is run with the `-upgrade` option, it re-checks [releases.hashicorp.com](#) for newer acceptable provider versions and downloads them if available.

This behavior only applies to providers whose *only* acceptable versions are in `.terraform/plugins/<OS>_<ARCH>` (the automatic downloads directory); if any acceptable version of a given provider is installed elsewhere, `terraform init -upgrade` will not download a newer version of it.

Terraform Plugin Types

Terraform is logically split into two main parts: Terraform Core and Terraform Plugins. Each plugin exposes an implementation for a specific service, such as the AWS provider or bash provisioner. Terraform Plugins are written in Go and are executable binaries executed as a separate process and communicate with the main Terraform binary over an RPC interface. The network communication and RPC is handled automatically by higher-level Terraform libraries, so developers need only worry about the implementation of their specific Plugin behavior.

There are two types of plugins supported by Terraform:

Providers

Providers are the most common type of Plugin, which expose the features that a specific service offers via its application programming interface (https://en.wikipedia.org/wiki/Application_programming_interface) (API). Providers define **Resources** and are responsible for managing their life cycles. Examples of providers are the Amazon Web Service Provider (/docs/providers/aws/index.html) or the Google Cloud Provider (/docs/providers/google/index.html). Example resources are `aws_instance` and `google_compute_instance`.

Terraform Providers contain all the code needed to authenticate and connect to a service on behalf of the user. Each Resource implements CREATE, READ, UPDATE, and DELETE (CRUD) methods to manage itself, while Terraform Core manages a Resource Graph (/docs/internals/graph.html) of all the resources declared in the configuration as well as their current state. Resources remain ignorant of the current state, only responding to method calls from Terraform Core and performing the matching CRUD action.

Terraform determines the Providers needed by reading and interpolating configuration files. Terraform will dynamically discover and fetch the needed providers from releases.hashicorp.com (<https://releases.hashicorp.com>), or from specific locations on disk (/docs/extend/how-terraform-works.html#discovery). At time of writing, the source code for all Providers distributed by HashiCorp for automatic discovery are hosted on in the `terraform-providers` GitHub Organization (<https://github.com/terraform-providers>).

Visit the Provider index (/docs/providers/index.html) in our documentation section to learn more about our existing Providers.

Provisioners

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction. Provisioners can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc.

The official Terraform Provisioners are included in the Terraform Core codebase and are compiled into the `terraform` binary. While they are built in, Provisioners are still executed in a separate process over RPC, and benefit from the same discovery process as Terraform Providers, making custom Provisioners just as easy to create and use.

Visit the Provisioners index (/docs/provisioners/index.html) in our documentation section to learn more about our existing Provisioners.

Resources

A key component to Provider development is defining a resource. Aspects of this component have already been covered in Writing Custom Providers (/docs/extend/writing-custom-providers.html) and Schemas (/docs/extend/schemas/), this document will cover the other features of `schema.Resource`.

Table of Contents

- State Migrations
- Retry Helpers
 - Retry
 - `StateChangeConf`
- Importers
- Customizing Differences

State Migrations

Resources define the data types and API interactions required to create, update, and destroy infrastructure with a cloud vendor while the Terraform state (/docs/state/index.html) stores mapping and metadata information for those remote objects. There are several reasons why a resource implementation needs to change: backend APIs Terraform interacts with will change overtime, or the current implementation might be incorrect or unmaintainable. Some of these changes may not be backward compatible and a migration is needed for resources provisioned in the wild with old schema configurations.

For this task provider developers should use a resource's `SchemaVersion` and `MigrateState` function. Resources do not have these options set on first implementation, the `SchemaVersion` defaults to 0.

```
package example

import "github.com/hashicorp/terraform/helper/schema"

func resourceExampleInstance() *schema.Resource {
    return &schema.Resource{
        Create: resourceExampleInstanceCreate,
        Read:   resourceExampleInstanceRead,
        Update: resourceExampleInstanceUpdate,
        Delete: resourceExampleInstanceDelete,

        Schema: map[string]*schema.Schema{
            "name": {
                Type:     schema.TypeString,
                Required: true,
            },
        },
    }
}
```

Say the instance resource API now requires the `name` attribute to end with a period ".."

```

package example

import (
    "fmt"
    "strings"

    "github.com/hashicorp/terraform/helper/schema"
)

func resourceExampleInstance() *schema.Resource {
    return &schema.Resource{
        Create: resourceExampleInstanceCreate,
        Read:   resourceExampleInstanceRead,
        Update: resourceExampleInstanceUpdate,
        Delete: resourceExampleInstanceDelete,

        Schema: map[string]*schema.Schema{
            "name": {
                Type:     schema.TypeString,
                Required: true,
                ValidateFunc: func(v interface{}, k string) (warnings []string, errors []error) {
                    if !strings.HasSuffix(v.(string), ".") {
                        errors = append(errors, fmt.Errorf("%q must end with a period '.'", k))
                    }
                    return
                },
            },
            },
        SchemaVersion: 1,
        MigrateState:  resourceExampleInstanceMigrateState,
    }
}

```

To trigger the migration we set the `SchemaVersion` to 1. When Terraform saves state it also sets the `SchemaVersion` at the time, that way when differences are calculated, if the saved `SchemaVersion` is less than what the `Resource` is currently set to, the state is run through the `MigrateState` function.

```

func resourceExampleInstanceStateMigrateState(v int, inst *terraform.InstanceState, meta interface{}) (*terraform.InstanceState, error) {
    switch v {
    case 0:
        log.Println("[INFO] Found Example Instance State v0; migrating to v1")
        return migrateExampleInstanceStateV0toV1(inst)
    default:
        return inst, fmt.Errorf("Unexpected schema version: %d", v)
    }
}

func migrateExampleInstanceStateV0toV1(inst *terraform.InstanceState, error) (*terraform.InstanceState, error) {
    if inst.Empty() {
        log.Println("[DEBUG] Empty InstanceState; nothing to migrate.")
        return inst, nil
    }

    if !strings.HasSuffix(inst.Attributes["name"], ".") {
        log.Printf("[DEBUG] Attributes before migration: %#v", inst.Attributes)
        inst.Attributes["name"] = inst.Attributes["name"] + "."
        log.Printf("[DEBUG] Attributes after migration: %#v", inst.Attributes)
    }

    return inst, nil
}

```

Although not required, it's a good idea to break the migration function up into version jumps. As the provider developer you will have to account for migrations that are larger than one version upgrade, using the switch/case pattern above will allow you to create codepaths for states coming from all the versions of state in the wild. Please be careful to allow all legacy versions to migrate to the latest schema. Consider the code now where the name attribute has moved to an attribute called fqdn.

```

func resourceExampleInstanceStateMigrateState(v int, inst *terraform.InstanceState, meta interface{}) (*terraform.InstanceState, error) {
    var err error
    switch v {
    case 0:
        log.Println("[INFO] Found Example Instance State v0; migrating to v1")
        inst, err = migrateExampleInstanceStateV0toV1(inst)
        if err != nil {
            return inst, err
        }
        fallthrough
    case 1:
        log.Println("[INFO] Found Example Instance State v1; migrating to v2")
        return migrateExampleInstanceStateV1toV2(inst)
    default:
        return inst, fmt.Errorf("Unexpected schema version: %d", v)
    }
}

func migrateExampleInstanceStateV1toV2(inst *terraform.InstanceState) (*terraform.InstanceState, error) {
    if inst.Empty() {
        log.Println("[DEBUG] Empty InstanceState; nothing to migrate.")
        return inst, nil
    }

    if inst.Attributes["name"] != "" {
        inst.Attributes["fqdn"] = inst.Attributes["name"]
        delete(inst.Attributes, "name")
    }
    return inst, nil
}

```

The fallthrough allows a very old state to move from 0 to 1 and now to 2. Sometimes state migrations are more complicated, and requires making API calls, to allow this the configured `meta interface{}` is also passed to the `MigrateState` function.

Retry Helpers

The reality of cloud infrastructure is that it typically takes time to do things like boot operating systems, discover services, and replicate state across network edges. As the provider developer you should take known delays in resource APIs into account in the CRUD functions of the resource. Terraform supports configurable timeouts to assist in these situations.

```

package example

import (
    "fmt"

    "github.com/hashicorp/terraform/helper/resource"
    "github.com/hashicorp/terraform/helper/schema"
)

func resourceExampleInstance() *schema.Resource {
    return &schema.Resource{
        Create: resourceExampleInstanceCreate,
        Read:   resourceExampleInstanceRead,
        Update: resourceExampleInstanceUpdate,
        Delete: resourceExampleInstanceDelete,

        Schema: map[string]*schema.Schema{
            "name": {
                Type:     schema.TypeString,
                Required: true,
            },
        },
        Timeouts: &schema.ResourceTimeout{
            Create: schema.DefaultTimeout(45 * time.Minute),
        },
    }
}

```

In the above example we see the usage of the timeouts in the schema being configured for what is deemed the appropriate amount of time for the Create function. Read, Update, and Delete are also configurable as well as a Default. These configured timeouts can be fetched later in the CRUD functions from the passed in `*schema.ResourceData`.

Retry

A common case for requiring retries or polling is when the backend infrastructure being provisioned is designed to be asynchronous, requiring the developer to repeatedly check the status of the resource. The retry helper takes a timeout and a function that is retried repeatedly. The timeout can be retrieved from the `*schema.ResourceData` struct, using the `Timeout` method, passing in the appropriate timeout key (`scheme.TimeoutCreate`). The retry function provided should return either a `resource.NonRetryableError` for unexpected errors or states, otherwise continue to retry with a `resource.RetryableError`. In the context of a CREATE function, once the backend responds with the desired state, finish the function with a `resource.NonRetryableError` wrapping the READ function (anything that goes wrong in there is considered unexpected).

```

func resourceExampleInstanceCreate(d *schema.ResourceData, meta interface{}) error {
    name := d.Get("name").(string)
    client := meta.(*ExampleClient)
    resp, err := client.CreateInstance(name)

    if err != nil {
        return fmt.Errorf("Error creating instance: %s", err)
    }

    return resource.Retry(d.Timeout(schema.TimeoutCreate), func() *resource.RetryError {
        resp, err := client.DescribeInstance(name)

        if err != nil {
            return resource.NonRetryableError(fmt.Errorf("Error describing instance: %s", err))
        }

        if resp.Status != "CREATED" {
            return resource.RetryableError(fmt.Errorf("Expected instance to be created but was in state %s", resp.Status))
        }

        return resource.NonRetryableError(resourceExampleInstanceRead(d, meta))
    })
}

```

StateChangeConf

`resource.Retry` is useful for simple scenarios, particularly when the API response is either success or failure, but sometimes handling an APIs latency or eventual consistency requires more fine tuning. `resource.Retry` is in fact a wrapper for another helper: `resource.StateChangeConf`.

Use `resource.StateChangeConf` when your resource has multiple states to progress through, you require fine grained control of retry and delay timing, or you want to ensure a minimum number of occurrences of a target state is reached (this is very common when dealing with eventually consistent APIs, where a response can reply back with an old state between calls before becoming consistent).

```

func resourceExampleInstanceCreate(d *schema.ResourceData, meta interface{}) error {
    name := d.Get("name").(string)
    client := meta.(*ExampleClient)
    resp, err := client.CreateInstance(name)

    createStateConf := &resource.StateChangeConf{
        Pending: []string{
            client.ExampleInstaceStateRequesting,
            client.ExampleInstaceStatePending,
            client.ExampleInstaceStateCreating,
            client.ExampleInstaceStateVerifying,
        },
        Target: []string{
            client.ExampleInstaceStateCreateComplete,
        },
        Refresh: func() (interface{}, string, error) {
            resp, err := client.DescribeInstance(name)
            if err != nil {
                0, "", err
            }
            return resp, resp.Status, nil
        },
        Timeout:    d.Timeout(schema.TimeoutCreate),
        Delay:      10 * time.Second,
        MinTimeout: 5 * time.Second,
        ContinuousTargetOccurence: 5,
    }
    _, err = createStateConf.WaitForState()
    if err != nil {
        return fmt.Errorf("Error waiting for example instance (%s) to be created: %s", d.Id(), err)
    }

    return resourceExampleInstanceRead(d, meta)
}

```

Importers

Many users migrating to Terraform often have manually managed infrastructure they want to bring under the management of Terraform. Terraform provides a mechanism known as an importer to consolidate those resources into state. As of writing the user will still have to write configuration that will be associated to the import.

When importing the user will specify the configuration address and id of the resource

```
terraform import example_instance.foo 000-0000
```

A resources READ function will perform a lookup based on the configured id. To support this Terraform provides a convenience that allows this passthrough.

```

package example

import (
    "fmt"

    "github.com/hashicorp/terraform/helper/schema"
)

func resourceExampleInstance() *schema.Resource {
    return &schema.Resource{
        Create: resourceExampleInstanceCreate,
        Read:   resourceExampleInstanceRead,
        Update: resourceExampleInstanceUpdate,
        Delete: resourceExampleInstanceDelete,

        Schema: map[string]*schema.Schema{
            "name": {
                Type:     schema.TypeString,
                Required: true,
            },
        },
        Importer: &schema.ResourceImporter{
            State: schema.ImportStatePassthrough,
        },
    }
}

func resourceExampleInstanceRead(d *schema.ResourceData, meta interface{}) error {
    client := m.(*MyClient)

    obj, err := client.Get(d.Id())

    if err != nil {
        d.SetId("")
        return fmt.Errorf("Error retrieving example instance: %s: %s", d.Id(), err)
    }

    d.Set("name", obj.Name)
    return nil
}

```

There are other cases where the READ function uses configuration parameters as the identifier or support for importing multiple resources, as seen in the AWS Provider (https://github.com/terraform-providers/terraform-provider-aws/blob/d3fe7e9907263b1aa41ddc0736a34b42899d1536/aws/import_aws_dx_gateway.go#L12) is needed. In general it is advised to stick to the passthrough importer when possible.

Customizing Differences

Terraform tracks the state of provisioned resources in its state file. The user passed configuration is compared against what is in the state file. When there is a detected discrepancy the user is presented with the difference of what is configured versus what is in state. Sometimes these scenarios require special handling, this is where the `CustomizeDiff` function is used. It is passed a `*schema.ResourceDiff`, a structure similar to `schema.ResourceData` but lacking most write functions like `Set`, while introducing new functions that work with the difference such as `SetNew`, `SetNewComputed`, and `ForceNew`.

While any function can be provided for difference customization, it is recommended to try and compose the behavior using the `customdiff` (<https://godoc.org/github.com/hashicorp/terraform/helper/customdiff>) helper package. This will allow for a more declarative configuration; however, it should not be overused, so for highly custom requirements, opt for a tailor-

made function.

```
package example

import (
    "fmt"

    "github.com/hashicorp/terraform/helper/customdiff"
    "github.com/hashicorp/terraform/helper/schema"
)

func resourceExampleInstance() *schema.Resource {
    return &schema.Resource{
        Create: resourceExampleInstanceCreate,
        Read:   resourceExampleInstanceRead,
        Update: resourceExampleInstanceUpdate,
        Delete: resourceExampleInstanceDelete,

        Schema: map[string]*schema.Schema{
            "size": {
                Type:     schema.TypeInt,
                Required: true,
            },
        },
        CustomizeDiff: customdiff.All(
            customdiff.ValidateChange("size", func (old, new, meta interface{}) error {
                // If we are increasing "size" then the new value must be
                // a multiple of the old value.
                if new.(int) <= old.(int) {
                    return nil
                }
                if (new.(int) % old.(int)) != 0 {
                    return fmt.Errorf("new size value must be an integer multiple of old value %d", old.(int))
                }
                return nil
            }),
            customdiff.ForceNewIfChange("size", func (old, new, meta interface{}) bool {
                // "size" can only increase in-place, so we must create a new resource
                // if it is decreased.
                return new.(int) < old.(int)
            }),
        ),
    }
}
```

In this example we use the helpers to ensure the size can only be increased to multiples of the original size, and that if it is ever decreased it forces a new resource. The `customdiff.All` helper will run all the customization functions, collecting any errors as a `multieerror`. To have the functions short-circuit on error, please use `customdiff.Sequence`.

Terraform Schemas

Terraform Plugins are expressed using schemas to define attributes and their behaviors, using a high level package exposed by Terraform Core named `schema` (<https://github.com/hashicorp/terraform/tree/master/helper/schema>). Providers, Resources, and Provisioners all contains schemas, and Terraform Core uses them to produce plan and apply executions based on the behaviors described.

Below is an example `provider.go` file, detailing a hypothetical `ExampleProvider` implementation:

```
package exampleprovider

import (
    "github.com/hashicorp/terraform/helper/schema"
    "github.com/hashicorp/terraform/terraform"
)

// Provider returns a terraform.ResourceProvider.
func Provider() terraform.ResourceProvider {
    // Example Provider requires an API Token.
    // The Email is optional
    return &schema.Provider{
        Schema: map[string]*schema.Schema{
            "api_token": {
                Type:     schema.TypeString,
                Required: true,
            },
            "email": {
                Type:     schema.TypeString,
                Optional: true,
                Default:  "",
            },
        },
    }
}
```

In this example we're creating a `Provider` and setting it's schema. This schema is a collection of key value pairs of schema elements the attributes a user can specify in their configuration. The keys are strings, and the values are `schema.Schema` (<https://github.com/hashicorp/terraform/blob/5727d3335247e5940af2bef35c88657753f6d260/helper/schema/schema.go#L37>) structs that define the behavior.

Schemas can be thought of as a type paired one or more properties that describe it's behavior.

Schema Types

Schema items must be defined using one of the builtin types, such as `TypeString`, `TypeBool`, `TypeInt`, et. al. The type defines what is considered valid input for a given schema item in a users configuration.

See Schema Types (</docs/extend/schemas/schema-types.html>) for more information on the types available to schemas.

Schema Behaviors

Schema items can have various properties that can be combined to match their behaviors represented by their API. Some items are **Required**, others **Optional**, while others may be **Computed** such that they are useful to be tracked in state, but cannot be configured by users.

See Schema Behaviors ([/docs/extend/schemas/schema-behaviors.html](#)) for more information on the properties a schema can have.

Terraform Schemas

Terraform Plugins are expressed using schemas to define attributes and their behaviors, using a high level package exposed by Terraform Core named `schema` (<https://github.com/hashicorp/terraform/tree/master/helper/schema>). Providers, Resources, and Provisioners all contains schemas, and Terraform Core uses them to produce plan and apply executions based on the behaviors described.

Below is an example `provider.go` file, detailing a hypothetical `ExampleProvider` implementation:

```
package exampleprovider

import (
    "github.com/hashicorp/terraform/helper/schema"
    "github.com/hashicorp/terraform/terraform"
)

// Provider returns a terraform.ResourceProvider.
func Provider() terraform.ResourceProvider {
    // Example Provider requires an API Token.
    // The Email is optional
    return &schema.Provider{
        Schema: map[string]*schema.Schema{
            "api_token": {
                Type:     schema.TypeString,
                Required: true,
            },
            "email": {
                Type:     schema.TypeString,
                Optional: true,
                Default:  "",
            },
        },
    }
}
```

In this example we're creating a `Provider` and setting it's schema. This schema is a collection of key value pairs of schema elements the attributes a user can specify in their configuration. The keys are strings, and the values are `schema.Schema` (<https://github.com/hashicorp/terraform/blob/5727d3335247e5940af2bef35c88657753f6d260/helper/schema/schema.go#L37>) structs that define the behavior.

Schemas can be thought of as a type paired one or more properties that describe it's behavior.

Schema Types

Schema items must be defined using one of the builtin types, such as `TypeString`, `TypeBool`, `TypeInt`, et. al. The type defines what is considered valid input for a given schema item in a users configuration.

See Schema Types (</docs/extend/schemas/schema-types.html>) for more information on the types available to schemas.

Schema Behaviors

Schema items can have various properties that can be combined to match their behaviors represented by their API. Some items are **Required**, others **Optional**, while others may be **Computed** such that they are useful to be tracked in state, but cannot be configured by users.

See Schema Behaviors ([/docs/extend/schemas/schema-behaviors.html](#)) for more information on the properties a schema can have.

Schema Behaviors

Schema fields that can have an effect at plan or apply time are collectively referred to as "Behavioral fields", or an elements *behaviors*. These fields are often combined in several ways to create different behaviors, depending on the need of the element in question, typically customized to match the behavior of a cloud service API. For example, at time of writing AWS Launch Configurations cannot be updated through the AWS API. As a result all of the schema elements in the corresponding Terraform Provider resource `aws_launch_configuration` are marked as `ForceNew: true`. This behavior instructs Terraform to first destroy and then recreate the resource if any of the attributes change in the configuration, as opposed to trying to update the existing resource.

Primitive Behaviors

Optional

Data structure: `bool` (<https://golang.org/pkg/builtin/#bool>)

Values: `true` or `false`

Restrictions:

- Cannot be used if `Required` is `true`
- Must be set if `Required` is omitted **and** element is not `Computed`

Indicates that this element is optional to include in the configuration. Note that `Optional` does not itself establish a default value. See `Default` below.

Schema example:

```
"encrypted": {  
    Type:      schema.TypeBool,  
    Optional:  true,  
},
```

Configuration example:

```
resource "example_volume" "ex" {  
    encrypted = true  
}
```

Required

Data structure: `bool` (<https://golang.org/pkg/builtin/#bool>)

Values: `true` or `false`

Restrictions:

- Cannot be used if `Optional` is `true`
- Cannot be used if `Computed` is `true`
- Must be set if `Optional` is omitted **and** element is not `Computed`

Indicates that this element must be provided in the configuration. Omitting this attribute from configuration, or later removing it, will result in a plan-time error.

Schema example:

```
"name": {  
    Type:     schema.TypeString,  
    Required: true,  
},
```

Configuration example:

```
resource "example_volume" "ex" {  
    name = "swap volume"  
}
```

Default

Data structure: interface (https://golang.org/doc/effective_go.html#interfaces)

Value: any value of an elements Type for primitive types, or the type defined by Elem for complex types.

Restrictions:

- Cannot be used if Required is true
- Cannot be used with DefaultFunc

If Default is specified, that value that is used when this item is not set in the configuration.

Schema example:

```
"encrypted": {  
    Type:     schema.TypeBool,  
    Optional: true,  
    Default: false,  
},
```

Configuration example (specified):

```
resource "example_volume" "ex" {  
    name = "swap volume"  
    encrypted = true  
}
```

Configuration example (omitted):

```
resource "example_volume" "ex" {  
    name = "swap volume"  
    # encrypted receives its default value, false  
}
```

Computed

Data structure: bool (<https://golang.org/pkg/builtin/#bool>)

Value: true or false

Restrictions:

- Cannot be used when Required is true
- Cannot be used when Default is specified
- Cannot be used with DefaultFunc

Computed is often used to represent values that are not user configurable or can not be known at time of terraform plan or apply, such as date of creation or a service specific UUID. Computed can be combined with other attributes to achieve specific behaviors, and can be used as output for interpolation into other resources

Schema example:

```
"uuid": {  
  Type:    schema.TypeString,  
  Computed: true,  
},
```

Configuration example:

```
resource "example_volume" "ex" {  
  name = "swap volume"  
  encrypted = true  
}  
  
output "volume_uuid" {  
  value = "${example_volume.ex.uuid}"  
}
```

ForceNew

Data structure: bool (<https://golang.org/pkg/builtin/#bool>)

Value: true or false

ForceNew indicates that any change in this field requires the resource to be destroyed and recreated.

Schema example:

```
"base_image": {  
  Type:    schema.TypeString,  
  Required: true,  
  ForceNew: true,  
},
```

Configuration example:

```
resource "example_instance" "ex" {
  name = "bastion host"
  base_image = "ubuntu_17.10"
}
```

Function Behaviors

DiffSuppressFunc

Data structure: SchemaDiffSuppressFunc

(<https://github.com/hashicorp/terraform/blob/ead558261d5e322f1f1e90c8e74834ba9215f24e/helper/schema/schema.go#L202>)

When provided DiffSuppressFunc will be used by Terraform to calculate the diff of this field. Common use cases are capitalization differences in string names, or logical equivalences in JSON values.

Schema example:

```
"base_image": {
  Type:      schema.TypeString,
  Required:  true,
  ForceNew:  true,
  // Suppress the diff shown if the base_image name are equal when both compared in lower case.
  DiffSuppressFunc: func(k, old, new string, d *schema.ResourceData) bool {
    if strings.ToLower(old) == strings.ToLower(new) {
      return true
    }
    return false
  },
},
```

Configuration example:

Here we assume the service API accepts capitalizations of the base_image name and converts it to a lowercase string. The API then returns the lower case value in its responses.

```
resource "example_instance" "ex" {
  name = "bastion host"
  base_image = "UBunTu_17.10"
}
```

DefaultFunc

Data structure: SchemaDefaultFunc

(<https://github.com/hashicorp/terraform/blob/ead558261d5e322f1f1e90c8e74834ba9215f24e/helper/schema/schema.go#L209>)

Restrictions:

- Cannot be used if Default is specified

When `DefaultFunc` will be used to compute a dynamic default for this element. The return value of this function should be "stable", such that it is uncommon to return different values in subsequent plans without any other changes being made, to avoid unnecessary diffs in `terraform` plan.

`DefaultFunc` is most commonly used in Provider schemas, allows elements to have a default read from the environment.

Schema example:

In this example, Terraform will attempt to read `region` from the environment if it is omitted from configuration. If it's not found in the environment, a default value of `us-west` is given.

```
"region": {
  Type:      schema.TypeString,
  Required: true,
  DefaultFunc: func() (interface{}, error) {
    if v := os.Getenv("PROVIDER_REGION"); v != "" {
      return v, nil
    }

    return "us-west", nil
  },
},
```

Configuration example (provided):

```
provider "example" {
  api_key = "somesecretkey"
  region  = "us-east"
}
```

Configuration example (default func with `PROVIDER_REGION` set to `us-west` in the environment):

```
provider "example" {
  api_key = "somesecretkey"
  # region is "us-west"
}
```

Configuration example (default func with `PROVIDER_REGION` unset in the environment):

```
provider "example" {
  api_key = "somesecretkey"
  # region is "us-east"
}
```

StateFunc

Data structure: SchemaStateFunc

(<https://github.com/hashicorp/terraform/blob/a20dbb43782ade816baaeffa8033da0027ee6b26/helper/schema/schema.go#L245>)

`SchemaStateFunc` is a function used to convert the value of this element to a string to be stored in the state.

Schema example:

In this example, the `StateFunc` converts a string value to all lower case.

```

"name": &schema.Schema{
  Type:    schema.TypeString,
  ForceNew: true,
  Required: true,
  StateFunc: func(val interface{}) string {
    return strings.ToLower(val.(string))
  },
},

```

Configuration example (provided):

```

resource "example" "ex_instance" {
  name = "SomeValueCASEinsensitive"
}

```

Value in statefile:

```
"name": "somevaluecaseinsensitive"
```

ValidateFunc

Data structure: SchemaValidateFunc

(<https://github.com/hashicorp/terraform/blob/a20dbb43782ade816baaeffa8033da0027ee6b26/helper/schema/schema.go#L249>)

Restrictions:

- Only works with primitive types

SchemaValidateFunc is a function used to validate the value of a primitive type. Common use cases include ensuring an integer falls within a range or a string value is present in a list of valid options. The function returns two slices, the first for warnings, the second is errors which can be used to catch multiple invalid cases. Terraform will only halt execution if an error is returned. Returning warnings will warn the user but the data provided is considered valid.

Terraform includes a number of validators for use in plugins in the validation package. A full list can be found here:

<https://godoc.org/github.com/hashicorp/terraform/helper/validation>

(<https://godoc.org/github.com/hashicorp/terraform/helper/validation>)

Schema example:

In this example, the ValidateFunc ensures the integer provider is a value between 0 and 10.

```

"amount": &schema.Schema{
  Type:    schema.TypeInt,
  Required: true,
  ValidateFunc: func(val interface{}, key string) (warns []string, errs []error) {
    v := val.(int)
    if v < 0 || v > 10 {
      errs = append(errs, fmt.Errorf("%q must be between 0 and 10 inclusive, got: %d", key, v))
    }
    return
  },
},

```

Configuration example:

```
resource "example" "ex_instance" {  
    amount = "-1"  
}
```

Schema Attributes and Types

Almost every Terraform Plugin offers user configurable parameters, examples such as a Provisioner `secret_key`, a Provider's `region`, or a Resources name. Each parameter is defined in the items schema, which is a map of string names to schema structs.

In the below example implementation of a Resource you see parameters `uuid` and `name` defined:

```
func resourceExampleResource() *schema.Resource {
    return &schema.Resource{
        // ...
        Schema: map[string]*schema.Schema{
            "uuid": {
                Type:     schema.TypeString,
                Computed: true,
            },
            "name": {
                Type:     schema.TypeString,
                Required: true,
                ForceNew: true,
                ValidateFunc: validatName,
            },
            // ...
        },
    }
}
```

The Schema attribute `Type` defines what kind of values users can provide in their configuration for this element. Here we define the available schema types supported. See Schema Behaviors ([/docs/extend/schemas/schema-behaviors.html](#)) for more information on configuring element behaviors.

Types

The schema attribute `Type` determines what data is valid in configuring the element, as well as the type of data returned when used in interpolation ([/docs/configuration/interpolation.html](#)). Schemas attributes must be one of the types defined below, and can be loosely categorized as either **Primitive** or **Aggregate** types:

Primitive types

Primitive types are simple values such as integers, booleans, and strings. Primitives are stored in the state file (<https://www.terraform.io/docs/state/index.html>) as "key": "value" string pairs, where both key and value are string representations.

Aggregate types

Aggregate types form more complicated data types by combining primitive types. Aggregate types may define the types of elements they contain by using the `Elem` property. If the `Elem` property is omitted, the default element data type is a `string`.

Aggregate types are stored in state as a `key.index` and `value` pair for each element of the property, with a unique `index` appended to the key based on the type. There is an additional `key.index` item included in the state that tracks the number of items the property contains.

Primitive Types

TypeBool

Data structure: bool (<https://golang.org/pkg/builtin/#bool>)

Example: true or false

Schema example:

```
"encrypted": {  
    Type:      schema.TypeBool,  
},
```

Configuration example:

```
resource "example_volume" "ex" {  
    encrypted = true  
}
```

State representation:

```
"encrypted": "true",
```

TypeInt

Data structure: int (<https://golang.org/pkg/builtin/#int>)

Example: -9, 0, 1, 2, 9

Schema example:

```
"cores": {  
    Type:      schema.TypeInt,  
},
```

Configuration example:

```
resource "example_compute_instance" "ex" {  
    cores = 16  
}
```

State representation:

```
"cores": "16",
```

TypeFloat

Data structure: float64 (<https://golang.org/pkg/builtin/#float64>)

Example: 1.0, 7.19009

Schema example:

```
"price": {  
  Type: schema.TypeFloat,  
},
```

Configuration example:

```
resource "example_spot_request" "ex" {  
  price = 0.37  
}
```

State representation:

```
"price": "0.37",
```

TypeString

Data structure: string (<https://golang.org/pkg/builtin/#string>)

Example: "Hello, world!"

Schema example:

```
"name": {  
  Type: schema.TypeString,  
},
```

Configuration example:

```
resource "example_spot_request" "ex" {  
  description = "Managed by Terraform"  
}
```

State representation:

```
"description": "Managed by Terraform",
```

Date & Time Data

TypeString is also used for date/time data, the preferred format is RFC 3339 (you can use the provided validation function (<https://godoc.org/github.com/hashicorp/terraform/helper/validation#ValidateRFC3339TimeString>)).

Example: 2006-01-02T15:04:05Z07:00

Schema example:

```
"expiration": {  
    Type:      schema.TypeString,  
    ValidateFunc: validation.ValidateRFC3339TimeString,  
},
```

Configuration example:

```
resource "example_resource" "ex" {  
    expiration = "2006-01-02T15:04:05Z07:00"  
}
```

State representation:

```
"expiration": "2006-01-02T15:04:05Z07:00",
```

Aggregate Types

TypeMap

Data structure: map (https://golang.org/doc/effective_go.html#maps): map[string]interface{}

Example: key = value

A key based map (also known as a dictionary) with string keys and values defined by the `Elem` property.

Schema example:

```
"tags": {  
    Type:      schema.TypeMap,  
},
```

Configuration example:

```
resource "example_compute_instance" "ex" {  
    tags {  
        env = "development"  
        name = "example tag"  
    }  
}
```

State representation:

TypeMap items are stored in state with the key as the index. The count of items in a map is denoted by the `%` index:

```
"tags.%": "2",
"tags.env": "development",
"tags.name": "example tag",
```

TypeList

Data structure: Slice (https://golang.org/doc/effective_go.html#slices): []interface{}

Example: []interface{"2", "3", "4"}

Used to represent an **ordered** collection of items, where the order the items are presented can impact the behavior of the resource being modeled. An example of ordered items would be network routing rules, where rules are examined in the order they are given until a match is found. The items are all of the same type defined by the `Elem` property.

Schema example:

```
"termination_policies": {
  Type:    schema.TypeList,
  Elem: &schema.Schema{
    Type: schema.TypeString,
  },
},
```

Configuration example:

```
resource "example_compute_instance" "ex" {
  termination_policies = ["OldestInstance", "ClosestToNextInstanceHour"]
}
```

State representation:

TypeList items are stored in state in a zero based index data structure.

```
"name_servers.#": "4",
"name_servers.0": "ns-1508.awsdns-60.org",
"name_servers.1": "ns-1956.awsdns-52.co.uk",
"name_servers.2": "ns-469.awsdns-58.com",
"name_servers.3": "ns-564.awsdns-06.net",
```

TypeSet

Data structure: *schema.Set

(<https://github.com/hashicorp/terraform/blob/504d5ef233230984ccd378a6847fa8b9e32f6bee/helper/schema/set.go#L44>)

Example: []string{"one", "two", "three"}

TypeSet implements set behavior and is used to represent an **unordered** collection of items, meaning that their ordering specified does not need to be consistent, and the ordering itself has no impact on the behavior of the resource.

The elements of a set can be any of the other types allowed by Terraform, including another schema. Set items cannot be repeated.

Schema example:

```
"ingress": {
  Type:      schema.TypeSet,
  Elem: &schema.Resource{
    Schema: map[string]*schema.Schema{
      "from_port": {
        Type:      schema.TypeInt,
        Required: true,
      },
      "to_port": {
        Type:      schema.TypeInt,
        Required: true,
      },
      "protocol": {
        Type:      schema.TypeString,
        Required: true,
        StateFunc: protocolStateFunc,
      },
      "cidr_blocks": {
        Type:      schema.TypeList,
        Optional: true,
        Elem: &schema.Schema{
          Type:      schema.TypeString,
        },
      },
    },
  },
}
```

Configuration example:

```
resource "example_security_group" "ex" {
  name        = "sg_test"
  description = "managed by Terraform"

  ingress {
    protocol    = "tcp"
    from_port   = 80
    to_port     = 9000
    cidr_blocks = ["10.0.0.0/8"]
  }

  ingress {
    protocol    = "tcp"
    from_port   = 80
    to_port     = 8000
    cidr_blocks = ["0.0.0.0/0", "10.0.0.0/8"]
  }
}
```

State representation:

TypeSet items are stored in state with an index value calculated by the hash of the attributes of the set.

```
"ingress.#": "2",
"ingress.1061987227.cidr_blocks.#": "1",
"ingress.1061987227.cidr_blocks.0": "10.0.0.0/8",
"ingress.1061987227.description": "",
"ingress.1061987227.from_port": "80",
"ingress.1061987227.ipv6_cidr_blocks.#": "0",
"ingress.1061987227.protocol": "tcp",
"ingress.1061987227.security_groups.#": "0",
"ingress.1061987227.self": "false",
"ingress.1061987227.to_port": "9000",
"ingress.493694946.cidr_blocks.#": "2",
"ingress.493694946.cidr_blocks.0": "0.0.0.0/0",
"ingress.493694946.cidr_blocks.1": "10.0.0.0/8",
"ingress.493694946.description": "",
"ingress.493694946.from_port": "80",
"ingress.493694946.ipv6_cidr_blocks.#": "0",
"ingress.493694946.protocol": "tcp",
"ingress.493694946.security_groups.#": "0",
"ingress.493694946.self": "false",
"ingress.493694946.to_port": "8000",
```

Next Steps

Checkout Schema Behaviors ([/docs/extend/schemas/schema-behaviors.html](#)) to learn how to customize each schema elements behavior.

Testing Terraform Plugins

Here we cover information needed to write successful tests for Terraform Plugins. Tests are a vital part of the Terraform ecosystem, verifying we can deliver on our mission to safely and predictably create, change, and improve infrastructure. Documentation for Terraform tests are broken into categories briefly described below. Each category has more detailed information by clicking on the matching item in the left navigation.

Acceptance Tests

In order to deliver on our promise to be safe and predictable, we need to be able to easily and routinely verify that Terraform Plugins produce the expected outcome. The most common usage of an acceptance test is in Terraform Providers, where each Resource is tested with configuration files and the resulting infrastructure is verified. Terraform includes a framework for constructing acceptance tests that imitate the execution of one or more steps of applying one or more configuration files, allowing multiple scenarios to be tested.

It's important to reiterate that acceptance tests in resources *create actual cloud infrastructure*, with possible expenses incurred, and are the responsibility of the user running the tests. Creating real infrastructure in tests verifies the described behavior of Terraform Plugins in real world use cases against the actual APIs, and verifies both local state and remote values match. Acceptance tests require a network connection and often require credentials to access an account for the given API. When writing and testing plugins, **it is highly recommended to use an account dedicated to testing, to ensure no infrastructure is created in error in any environment that cannot be completely and safely destroyed.**

HashiCorp runs nightly acceptance tests of providers found in the Terraform Providers GitHub Organization (<https://github.com/terraform-providers>) to ensure each Provider is working correctly.

For a given plugin, Acceptance Tests can be run from the root of the project by using a common make task:

```
$ make testacc
```

See Acceptance Testing (/docs/extend/testing/acceptance-tests/index.html) to learn more.

Unit Tests

Testing plugin code in small, isolated units is distinct from Acceptance Tests, and does not require network connections. Unit tests are commonly used for testing helper methods that expand or flatten API response data into data structures for storage into state by Terraform. This section covers the specifics of writing Unit Tests for Terraform Plugin code.

For a given plugin, Unit Tests can be run from the root of the project by using a common make task:

```
$ make test
```

See Unit Testing (/docs/extend/testing/unit-testing.html) to learn more.

Next Steps

See the navigation on the left of this page for documentation and guides on writing tests for Terraform Plugins.

Acceptance Tests

In order to deliver on our promise to be safe and predictable, we need to be able to easily and routinely verify that Terraform Plugins produce the expected outcome. The most common usage of an acceptance test is in Terraform Providers, where each Resource is tested with configuration files and the resulting infrastructure is verified. Terraform includes a framework for constructing acceptance tests that imitate the execution of one or more steps of applying one or more configuration files, allowing multiple scenarios to be tested.

Terraform acceptance tests use real Terraform configurations to exercise the code in real plan, apply, refresh, and destroy life cycles. When run from the root of a Terraform Provider codebase, Terraform's testing framework compiles the current provider in-memory and executes the provided configuration in developer defined steps, creating infrastructure along the way. At the conclusion of all the steps, Terraform automatically destroys the infrastructure. It's important to note that during development, it's possible for Terraform to leave orphaned or "dangling" resources behind, depending on the correctness of the code in development. The testing framework provides means to validate all resources are destroyed, alerting developers if any fail to destroy. It is the developer's responsibility to clean up any dangling resources left over from testing and development.

Test files

Terraform follows many of the Go programming language conventions with regards to testing, with both acceptance tests and unit tests being placed in a file that matches the file under test, with an added `_test.go` suffix. Here's an example file structure:

```
terraform-plugin-example/
├── provider.go
├── provider_test.go
└── example/
    ├── resource_example_compute.go
    └── resource_example_compute_test.go
```

To create an acceptance test in the example `resource_example_compute_test.go` file, the function name must begin with `TestAccXxx`, and have the following signature:

```
func TestAccXxx(*testing.T)
```

Running Acceptance Tests

Terraform requires an environment variable `TF_ACC` be set in order to run acceptance tests. This is by design, and intended to prevent developers from incurring unintended charges when running tests. The easiest way to run acceptance tests is to use the built in `make` step `testacc`, which explicitly sets the `TF_ACC` value for you. Example:

```
$ make testacc
```

It's important to reiterate that acceptance tests create actual cloud resources, possibly incurring expenses which are the responsibility of the user running the tests. Creating real infrastructure in tests verifies the described behavior of Terraform Plugins in real world use cases against the actual APIs, and verifies both local state and remote values match. Acceptance

tests require a network connection and often require credentials to access an account for the given API.

Note: When developing or testing Terraform plugins, we highly recommend running acceptance tests with an account dedicated to testing. This ensures no infrastructure is created or destroyed in error during development or validation of any Provider Resources in any environment that cannot be completely and safely destroyed.

Next Steps

Terraform relies heavily on acceptance tests to ensure we keep our promise of helping users safely and predictably create, change, and improve infrastructure. In our next section we detail how to create "Test Cases", individual acceptance tests using Terraform's testing framework, in order to build and verify real infrastructure. Proceed to Test Cases ([/docs/extend/testing/acceptance-tests/testcase.html](#))

Sweepers

Acceptance tests in Terraform provision and verify real infrastructure using Terraform's testing framework ([/docs/extend/testing/acceptance-tests/index.html](#)). Ideally all infrastructure created is then destroyed within the lifecycle of a test, however the reality is that there are several situations that can arise where resources created during a test are "leaked". Leaked test resources are resources created by Terraform during a test, but Terraform either failed to destroy them as part of the test, or the test falsely reported all resources were destroyed after completing the test. Common causes are intermittent errors or failures in vendor APIs, or developer error in the resource code or test.

To address the possibility of leaked resources, Terraform provides a mechanism called sweepers to cleanup leftover infrastructure. We will add a file to our folder structure that will invoke the sweeper helper.

```
terraform-plugin-example/
├── provider.go
├── provider_test.go
└── example/
    ├── example_sweeper_test.go
    ├── resource_example_compute.go
    └── resource_example_compute_test.go
```

`example_sweeper_test.go`

```
package example

import (
    "testing"

    "github.com/hashicorp/terraform/helper/resource"
)

func TestMain(m *testing.M) {
    resource.TestMain(m)
}

// sharedClientForRegion returns a common provider client configured for the specified region
func sharedClientForRegion(region string) (interface{}, error) {
    ...
    return client, nil
}
```

`resource.TestMain` is responsible for parsing the special test flags and invoking the sweepers. Sweepers should be added within the acceptance test file of a resource.

`resource_example_compute_test.go`

```

package example

import (
    "log"
    "strings"
    "testing"

    "github.com/hashicorp/terraform/helper/resource"
)

func init() {
    resource.AddTestSweepers("example_compute", &resource.Sweeper{
        Name:    "example_compute",
        F: func (region string) error {
            client, err := sharedClientForRegion(region)
            if err != nil {
                return fmt.Errorf("Error getting client: %s", err)
            }
            conn := client.(*ExampleClient)

            instances, err := conn.DescribeComputeInstances()
            if err != nil {
                return fmt.Errorf("Error getting instances: %s", err)
            }
            for _, instance := range instances {
                if strings.HasPrefix(instance.Name, "test-acc") {
                    err := conn.DestroyInstance(instance.ID)
                }
                if err != nil {
                    log.Printf("Error destroying %s during sweep: %s", instance.Name, err)
                }
            }
        },
    })
}

```

This example demonstrates adding a sweeper, it is important to note that the string passed to `resource.AddTestSweepers` is added to a map, this name must therefore be unique. Also note there needs to be a way of identifying resources created by Terraform during acceptance tests, a common practice is to prefix all resource names created during acceptance tests with "test-acc" or something similar.

For more complex leaks, sweepers can also specify a list of sweepers that need to be run prior to the one being defined.

`resource_example_compute_disk_test.go`

```

package example

import (
    "testing"

    "github.com/hashicorp/terraform/helper/resource"
)

func init() {
    resource.AddTestSweepers("example_compute_disk", &resource.Sweeper{
        Name: "example_compute_disk",
        Dependencies: []string{"example_compute"},
        ...
    })
}

```

The sweepers can be invoked with the common make target `sweep`:

```
$ make sweep
WARNING: This will destroy infrastructure. Use only in development accounts.
go test ...
...
```

Acceptance Tests: TestCases

Acceptance tests are expressed in terms of **Test Cases**, each using one or more Terraform configurations designed to create a set of resources under test, and then verify the actual infrastructure created. Terraform's resource package offers a method `Test()`, accepting two parameters and acting as the entry point to Terraform's acceptance test framework. The first parameter is the standard `*testing.T` struct from Golang's Testing package (<https://golang.org/pkg/testing/#T>), and the second is `TestCase` (<https://github.com/hashicorp/terraform/blob/0cc9e050ecd4a46ba6448758c2edc0b29bef5695/helper/resource/testing.go#L195-L247>), a Go struct that developers use to setup the acceptance tests.

Here's an example acceptance test. Here the Provider is named `Example`, and the Resource under test is `Widget`. The parts of this test are explained below the example.

```
package example

// example.Widget represents a concrete Go type that represents an API resource
func TestAccExampleWidget_basic(t *testing.T) {
    var widgetBefore, widgetAfter example.Widget
    rName := acctest.RandStringFromCharSet(10, acctest.CharSetAlphaNum)

    resource.Test(t, resource.TestCase{
        PreCheck: func() { testAccPreCheck(t) },
        Providers: testAccProviders,
        CheckDestroy: testAccCheckExampleResourceDestroy,
        Steps: []resource.TestStep{
            {
                Config: testAccExampleResource(rName),
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckExampleResourceExists("example_widget.foo", &widgetBefore),
                ),
            },
            {
                Config: testAccExampleResource_removedPolicy(rName),
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckExampleResourceExists("example_widget.foo", &widgetAfter),
                ),
            },
        },
    })
}
```

Creating Acceptance Tests Functions

Terraform acceptance tests are declared with the naming pattern `TestAccXxx` with the standard Go test function signature of `func TestAccXxx(*testing.T)`. Using the above test as an example:

```
// File: example/widget_test.go
package example

func TestAccExampleWidget_basic(t *testing.T) {
    // ...
}
```

Inside this function we invoke `resource.Test()` with the `*testing.T` input and a new `testCase` object:

```
// File: example/widget_test.go
package example

func TestAccExampleWidget_basic(t *testing.T) {
    resource.Test(t, resource.TestCase{
        // ...
    })
}
```

The majority of acceptance tests will only invoke `resource.Test()` and exit. If at any point this method encounters an error, either in executing the provided Terraform configurations or subsequent developer defined checks, `Test()` will invoke the `t.Error` method of Go's standard testing framework and the test will fail. A failed test will not halt or otherwise interrupt any other tests currently running.

TestCase Reference API

TestCase offers several fields for developers to add to customize and validate each test, defined below. The source for TestCase can be viewed here on godoc.org (<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCase>)

IsUnitTest

Type: bool (<https://golang.org/pkg/builtin/#bool>)

Default: false

Required: no

IsUnitTest allows a test to run regardless of the TF_ACC environment variable. This should be used with care - only for fast tests on local resources (e.g. remote state with a local backend) but can be used to increase confidence in correct operation of Terraform without waiting for a full acceptance test run.

PreCheck

Type: function

Default: nil

Required: no

PreCheck if non-nil, will be called before any test steps are executed. It is commonly used to verify that required values exist for testing, such as environment variables containing test keys that are used to configure the Provider or Resource under test.

Example usage:

```

// File: example/widget_test.go
package example

func TestAccExampleWidget_basic(t *testing.T) {
    resource.Test(t, resource.TestCase{
        PreCheck: func() { testAccPreCheck(t) },
        // ...
    })
}

// testAccPreCheck validates the necessary test API keys exist
// in the testing environment
func testAccPreCheck(t *testing.T) {
    if v := os.Getenv("EXAMPLE_KEY"); v == "" {
        t.Fatal("EXAMPLE_KEY must be set for acceptance tests")
    }
    if v := os.Getenv("EXAMPLE_SECRET"); v == "" {
        t.Fatal("EXAMPLE_SECRET must be set for acceptance tests")
    }
}

```

Providers

Type: map[string]terraform.ResourceProvider

(https://github.com/hashicorp/terraform/blob/0cc9e050ecd4a46ba6448758c2edc0b29bef5695/terraform/resource_provider.go#L10-L171)

Required: Yes

Providers is a map of `terraform.ResourceProvider` values with `string` keys, representing the Providers that will be under test. Only the Providers included in this map will be loaded during the test, so any Provider included in a configuration file for testing must be represented in this map or the test will fail during initialization.

This map is most commonly constructed once in a common `init()` method of the Provider's main test file, and includes an object of the current Provider type.

Example usage: (note the different files `widget_test.go` and `provider_test.go`)

```

// File: example/widget_test.go
package example

func TestAccExampleWidget_basic(t *testing.T) {
    resource.Test(t, resource.TestCase{
        PreCheck: func() { testAccPreCheck(t) },
        Providers: testAccProviders,
        // ...
    })
}

// File: example/provider_test.go
package example

var testAccProviders map[string]terraform.ResourceProvider
var testAccProvider *schema.Provider

func init() {
    testAccProvider = Provider().(*schema.Provider)
    testAccProviders = map[string]terraform.ResourceProvider{
        "example": testAccProvider,
    }
}

```

CheckDestroy

Type: TestCheckFunc

(<https://github.com/hashicorp/terraform/blob/0cc9e050ecd4a46ba6448758c2edc0b29bef5695/helper/resource/testing.go#L182-L186>)

Default: nil

Required: no

CheckDestroy is called after all test steps have been ran, and Terraform has ran `destroy` on the remaining state. This allows developers to ensure any resource created is truly destroyed. This method receives the last known Terraform state as input, and commonly uses infrastructure SDKs to query APIs directly to verify the expected objects are no longer found, and should return an error if any resources remain.

Example usage:

```

// File: example/widget_test.go
package example

func TestAccExampleWidget_basic(t *testing.T) {
    resource.Test(t, resource.TestCase{
        PreCheck:    func() { testAccPreCheck(t) },
        Providers:   testAccProviders,
        CheckDestroy: testAccCheckExampleResourceDestroy,
        // ...
    })
}

// testAccCheckExampleResourceDestroy verifies the Widget
// has been destroyed
func testAccCheckExampleResourceDestroy(s *terraform.State) error {
    // retrieve the connection established in Provider configuration
    conn := testAccProvider.Meta().(*ExampleClient)

    // loop through the resources in state, verifying each widget
    // is destroyed
    for _, rs := range s.RootModule().Resources {
        if rs.Type != "example_widget" {
            continue
        }

        // Retrieve our widget by referencing it's state ID for API lookup
        request := &example.DescribeWidgets{
            IDs: []string{rs.Primary.ID},
        }

        response, err := conn.DescribeWidgets(request)
        if err == nil {
            if len(response.Widgets) > 0 && response.Widgets[0].ID == rs.Primary.ID {
                return fmt.Errorf("Widget (%s) still exists.", rs.Primary.ID)
            }
        }

        return nil
    }

    // If the error is equivalent to 404 not found, the widget is destroyed.
    // Otherwise return the error
    if !strings.Contains(err.Error(), "Widget not found") {
        return err
    }
}

return nil
}

```

Steps

Type: []TestStep

(<https://github.com/hashicorp/terraform/blob/0cc9e050ecd4a46ba6448758c2edc0b29bef5695/helper/resource/testing.go#L249-L367>)

Required: yes

TestStep is a single apply sequence of a test, done within the context of a state. Multiple TestSteps can be sequenced in a Test to allow testing potentially complex update logic and usage. Basic tests typically contain one to two steps, to verify the resource can be created and subsequently updated, depending on the properties of the resource. In general, simply create/destroy tests will only need one step.

TestSteps are covered in detail in the next section, TestSteps (</docs/extend/testing/acceptance-tests/teststep.html>).

Example usage:

```
// File: example/widget_test.go
package example

func TestAccExampleWidget_basic(t *testing.T) {
    resource.Test(t, resource.TestCase{
        PreCheck:    func() { testAccPreCheck(t) },
        Providers:   testAccProviders,
        CheckDestroy: testAccCheckExampleResourceDestroy,
        Steps: []resource.TestStep{
            {
                Config: testAccExampleResource(rName),
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckExampleResourceExists("example_widget.foo", &widgetBefore),
                ),
            },
            {
                Config: testAccExampleResource_removedPolicy(rName),
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckExampleResourceExists("example_widget.foo", &widgetAfter),
                ),
            },
        },
    })
}
```

Next Steps

TestCases are used to verify the features of a given part of a plugin. Each case should represent a scenario of normal usage of the plugin, from simple creation to creating, adding, and removing specific properties. In the next Section [TestSteps](#) ([/docs/extend/testing/acceptance-tests/teststep.html](#)), we'll detail Steps portion of TestCase and see how to create these scenarios by iterating on Terraform configurations.

Acceptance Tests: TestSteps

TestSteps represent the application of an actual Terraform configuration file to a given state. Each step requires a configuration as input and provides developers several means of validating the behavior of the specific resource under test.

Test Modes

Terraform's test framework facilitates two distinct modes of acceptance tests, *Lifecycle* and *Import*.

Lifecycle mode is the most common mode, and is used for testing plugins by providing one or more configuration files with the same logic as would be used when running `terraform apply`.

Import mode is used for testing resource functionality to import existing infrastructure into a Terraform statefile, using the same logic as would be used when running `terraform import`.

An acceptance test's mode is implicitly determined by the fields provided in the `TestStep` definition. The applicable fields are defined below in the [\[TestStep Reference API\]](#)[#teststep-reference-api].

Steps

`Steps` is slice property of `TestCase` (/docs/extend/testing/acceptance-tests/testcase.html), the object used to construct acceptance tests. Each step represents a full `terraform apply` of a given configuration language, followed by zero or more checks (defined later) to verify the application. Each `Step` is applied in order, and require its own configuration and optional check functions.

Below is a code example of a lifecycle test that provides two `TestStep` objects:

```

package example

// example.Widget represents a concrete Go type that represents an API resource
func TestAccExampleWidget_basic(t *testing.T) {
    var widgetBefore, widgetAfter example.Widget
    rName := acctest.RandStringFromCharSet(10, acctest.CharSetAlphaNum)

    resource.Test(t, resource.TestCase{
        PreCheck: func() { testAccPreCheck(t) },
        Providers: testAccProviders,
        CheckDestroy: testAccCheckExampleResourceDestroy,
        Steps: []resource.TestStep{
            {
                Config: testAccExampleResource(rName),
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckExampleResourceExists("example_widget.foo", &widgetBefore),
                ),
            },
            {
                Config: testAccExampleResource_removedPolicy(rName),
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckExampleResourceExists("example_widget.foo", &widgetAfter),
                ),
            },
        },
    })
}

```

In the above example each TestCase invokes a function to retrieve it's desired configuration, based on a randomized name provided, however an in-line string or constant string would work as well, so long as they contain valid Terraform configuration for the plugin or resource under test. This pattern of first applying and checking a basic configuration, followed by applying a modified configuration with updated or additional checks is a common pattern used to test update functionality.

Check Functions

After the configuration for a TestStep is applied, Terraform's testing framework provides developers an opportunity to check the results by providing a "Check" function. While possible to only supply a single function, it is recommended you use multiple functions to validate specific information about the results of the `terraform apply` ran in each TestStep. The Check attribute is of TestStep is singular, so in order to include multiple checks developers should use either `ComposeTestCheckFunc` or `ComposeAggregateTestCheckFunc` (defined below) to group multiple check functions, defined below:

ComposeTestCheckFunc

`ComposeTestCheckFunc` lets you compose multiple `TestCheckFunc` functions into a single check. As a user testing their provider, this lets you decompose your checks into smaller pieces more easily, with individual methods for checking specific attributes. Each check is ran in the order provided, and on failure the entire TestCase is stopped, and Terraform attempts to destroy any resources created.

Example:

```

Steps: []resource.TestStep{
{
  Config: testAccExampleResource(rName),
  Check: resource.ComposeTestCheckFunc(
    // if testAccCheckExampleResourceExists fails to find the resource,
    // the parent TestStep and TestCase fail
    testAccCheckExampleResourceExists("example_widget.foo", &widgetBefore),
    resource.TestCheckResourceAttr("example_widget.foo", "size", "expected size"),
  ),
},
},
},

```

ComposeAggregateTestCheckFunc

ComposeAggregateTestCheckFunc lets you compose multiple TestCheckFunc functions into a single check. Its purpose and usage is identical to ComposeTestCheckFunc, however each check is ran in order even if a previous check failed, collecting the errors returned from any checks and returning a single aggregate error. The entire TestCase is still stopped, and Terraform attempts to destroy any resources created.

Example:

```

Steps: []resource.TestStep{
{
  Config: testAccExampleResource(rName),
  Check: resource.ComposeAggregateTestCheckFunc(
    testAccCheckExampleResourceExists("example_widget.foo", &widgetBefore), // if testAccCheckExampleResourceExists fails to find the resource, the following TestCheckResourceAttr is still ran, with any errors aggregated
    resource.TestCheckResourceAttr("example_widget.foo", "active", "true"),
  ),
},
},
},

```

Builtin check functions

Terraform has several TestCheckFunc functions built in for developers to use for common checks, such as verifying the status and value of a specific attribute in the resulting state. Developers are encouraged to use as many as reasonable to verify the behavior of the plugin/resource, and should combine them with the above mentioned ComposeTestCheckFunc or ComposeAggregateTestCheckFunc functions.

Most builtin functions accept name, key, and/or value fields, derived from the typical Terraform configuration stanzas:

```

resource "example_widget" "foo" {
  active = true
}

```

Here the name represents the resource name in state (`example_widget.foo`), the key represents the attribute to check (`active`), and value represents the desired value to check against (`true`). Not all functions accept all three inputs.

Below is a list of builtin check functions, with links to their corresponding documentation on godoc.org:

- `TestCheckResourceAttrSet(name, key string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckResourceAttrSet>)
- `TestCheckModuleResourceAttrSet(mp []string, name string, key string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckModuleResourceAttrSet>)
- `TestCheckResourceAttr(name, key, value string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckResourceAttr>)
- `TestCheckModuleResourceAttr(mp []string, name string, key string, value string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckModuleResourceAttr>)
- `TestCheckNoResourceAttr(name, key string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckNoResourceAttr>)
- `TestCheckModuleNoResourceAttr(mp []string, name string, key string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckModuleNoResourceAttr>)
- `TestCheckResourceAttrPtr(name string, key string, value *string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckResourceAttrPtr>)
- `TestCheckModuleResourceAttrPtr(mp []string, name string, key string, value *string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckModuleResourceAttrPtr>)
- `TestCheckResourceAttrPair(nameFirst, keyFirst, nameSecond, keySecond string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckResourceAttrPair>)
- `TestCheckModuleResourceAttrPair(mpFirst []string, nameFirst string, keyFirst string, mpSecond []string, nameSecond string, keySecond string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckModuleResourceAttrPair>)
- `TestCheckOutput(name, value string)`
(<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckOutput>)

Custom check functions

The `Check` field of `TestStep` accepts any function of type `TestCheckFunc` (<https://godoc.org/github.com/hashicorp/terraform/helper/resource#TestCheckFunc>). Developers are free to write their own check functions to create customized validation functions for their plugin. Any function that matches the `TestCheckFunc` function signature of `func(*terraform.State) error` can be used individually, or with other `TestCheckFunc` functions with one of the above Aggregate functions.

It's common to write custom `TestCheckFunc` functions to validate resources were created correctly by using SDKs directly to verify identity and properties of resources. These functions can retrieve information by SDKs and provide the results to other `TestCheckFunc` methods. The below example uses `ComposeTestCheckFunc` to group a set of `TestCheckFunc` functions together. The first function `testAccCheckExampleWidgetExists` uses the Example service SDK directly, and queries it for the ID of the widget we have in state. Once found, the result is stored into the `widget` struct declared at the begining of the test function. The next check function `testAccCheckExampleWidgetAttributes` receives the updated `widget` and checks its attributes. The final check `TestCheckResourceAttr` verifies that the same value is stored in state.

```
func TestAccExampleWidget_basic(t *testing.T) {
    var widget example.WidgetDescription
```

```

resource.TestCheck(t, resource.TestCase{
    PreCheck: func() { testAccPreCheck(t) },
    Providers: testAccProviders,
    CheckDestroy: testAccCheckExampleWidgetDestroy,
    Steps: []resource.TestStep{
        {
            Config: testAccExampleWidgetConfig,
            Check: resource.ComposeTestCheckFunc(
                testAccCheckExampleWidgetExists("example_widget.bar", &widget),
                testAccCheckExampleWidgetAttributes(&widget),
                resource.TestCheckResourceAttr("example_widget.bar", "active", "true"),
            ),
        },
    },
})
}

// testAccCheckExampleWidgetAttributes verifies attributes are set correctly by
// Terraform
func testAccCheckExampleWidgetAttributes(widget *example.WidgetDescription) resource.TestCheckFunc {
    return func(s *terraform.State) error {
        if *widget.active != true {
            return fmt.Errorf("widget is not active")
        }

        return nil
    }
}

// testAccCheckExampleWidgetExists uses the Example SDK directly to retrieve
// the Widget description, and stores it in the provided
// *example.WidgetDescription
func testAccCheckExampleWidgetExists(resourceName string, widget *example.WidgetDescription) resource.TestCheckFunc {
    return func(s *terraform.State) error {
        // retrieve the resource by name from state
        rs, ok := s.RootModule().Resources[resourceName]
        if !ok {
            return fmt.Errorf("Not found: %s", resourceName)
        }

        if rs.Primary.ID == "" {
            return fmt.Errorf("Widget ID is not set")
        }

        // retrieve the client from the test provider
        client := testAccProvider.Meta().(*ExampleClient)

        response, err := client.DescribeWidgets(&example.DescribeWidgetsInput{
            WidgetIDs: []string{rs.Primary.ID},
        })

        if err != nil {
            return err
        }

        // we expect only a single widget by this ID. If we find zero, or many,
// then we consider this an error
        if len(response.WidgetDescriptions) != 1 ||
            *response.WidgetDescriptions[0].WidgetID != rs.Primary.ID {
            return fmt.Errorf("Widget not found")
        }

        // store the resulting widget in the *example.WidgetDescription pointer
        *widget = *response.WidgetDescriptions[0]
    }
}

```

```
        return nil
    }
}
```

Next Steps

Acceptance Testing is an essential approach to validating the implementation of a Terraform Provider. Using actual APIs to provision resources for testing can leave behind real infrastructure that costs money between tests. The reasons for these leaks can vary, regardless Terraform provides a mechanism known as Sweepers ([/docs/extend/testing/acceptance-tests/sweepers.html](#)) to help keep the testing account clean.

Unit Testing

Testing plugin code in small, isolated units is distinct from Acceptance Tests, and does not require network connections. Unit tests are commonly used for testing helper methods that expand or flatten API responses into data structures for storage into state by Terraform. This section covers the specifics of writing Unit Tests for Terraform Plugin code.

The procedure for writing unit tests for Terraform follows the same setup and conventions of writing any Go unit tests. We recommend naming tests to follow the same convention as our acceptance tests, `Test<Provider>_<Test Name>`. For more information on Go tests, see the official Golang docs on testing (<https://golang.org/pkg/testing/>).

Below is an example unit test used in flattening AWS security group rules, demonstrating a typical `flattener` type method that's commonly used to convert structures returned from APIs into data structures used by Terraform in saving to state. This example is truncated for brevity, but you can see the full test in the `aws/structure_test.go` in the Terraform AWS Provider repository on GitHub (https://github.com/terraform-providers/terraform-provider-aws/blob/f22ae122d8407672bd38951f80a2813b8b9af683/aws/structure_test.go#L930-L1027)

```
func TestFlattenSecurityGroups(t *testing.T) {
    cases := []struct {
        ownerId  *string
        pairs    []*ec2.UserIdGroupPair
        expected []*GroupIdentifier
    }{
        // simple, no user id included
        {
            ownerId: aws.String("user1234"),
            pairs: []*ec2.UserIdGroupPair{
                &ec2.UserIdGroupPair{
                    GroupId: aws.String("sg-12345"),
                },
            },
            expected: []*GroupIdentifier{
                &GroupIdentifier{
                    GroupId: aws.String("sg-12345"),
                },
            },
        },
        // include the owner id, but keep it consistent with the same account. Tests
        // EC2 classic situation
        {
            ownerId: aws.String("user1234"),
            pairs: []*ec2.UserIdGroupPair{
                &ec2.UserIdGroupPair{
                    GroupId: aws.String("sg-12345"),
                    UserId:  aws.String("user1234"),
                },
            },
            expected: []*GroupIdentifier{
                &GroupIdentifier{
                    GroupId: aws.String("sg-12345"),
                },
            },
        },
        // include the owner id, but from a different account. This is reflects
        // EC2 Classic when referring to groups by name
        {
            ownerId: aws.String("user1234"),
            pairs: []*ec2.UserIdGroupPair{
                &ec2.UserIdGroupPair{
                    GroupId: aws.String("sg-12345"),
                },
            },
        },
    },
}
```

```
        GroupName: aws.String("somegroup"), // GroupName is only included in Classic
        UserId:    aws.String("user4321"),
    },
},
expected: []*GroupIdentifier{
    &GroupIdentifier{
        GroupId:  aws.String("sg-12345"),
        GroupName: aws.String("user4321/somegroup"),
    },
},
}

for _, c := range cases {
    out := flattenSecurityGroups(c.pairs, c.ownerId)
    if !reflect.DeepEqual(out, c.expected) {
        t.Fatalf("Error matching output and expected: %#v vs %#v", out, c.expected)
    }
}
}
```

Writing Custom Providers

In Terraform, a Provider is the logical abstraction of an upstream API. This guide details how to build a custom provider for Terraform.

NOTE: This guide details steps to author code and compile a working Provider. It omits many implementation details in order to get developers going with coding an example Provider and executing it with Terraform. Please refer to the rest of the Extending Terraform (/docs/extend/index.html) for a more complete reference on authoring Providers and Resources.

Why?

There are a few possible reasons for authoring a custom Terraform provider, such as:

- An internal private cloud whose functionality is either proprietary or would not benefit the open source community.
- A "work in progress" provider being tested locally before contributing back.
- Extensions of an existing provider

Local Setup

Terraform supports a plugin model, and all providers are actually plugins. Plugins are distributed as Go binaries. Although technically possible to write a plugin in another language, almost all Terraform plugins are written in Go (<https://golang.org>). For more information on installing and configuring Go, please visit the Golang installation guide (<https://golang.org/doc/install>).

This post assumes familiarity with Golang and basic programming concepts.

As a reminder, all of Terraform's core providers are open source. When stuck or looking for examples, please feel free to reference the open source providers (<https://github.com/terraform-providers>) for help.

The Provider Schema

To start, create a file named `provider.go`. This is the root of the provider and should include the following boilerplate code:

```
package main

import (
    "github.com/hashicorp/terraform/helper/schema"
)

func Provider() *schema.Provider {
    return &schema.Provider{
        ResourcesMap: map[string]*schema.Resource{},
    }
}
```

The `helper/schema` (<https://godoc.org/github.com/hashicorp/terraform/helper/schema>) library is part of Terraform Core (/docs/extend/how-terraform-works.html#terraform-core). It abstracts many of the complexities and ensures consistency between providers. The example above defines an empty provider (there are no *resources*).

The `*schema.Provider` type describes the provider's properties including:

- the configuration keys it accepts
- the resources it supports
- any callbacks to configure

Building the Plugin

Go requires a `main.go` file, which is the default executable when the binary is built. Since Terraform plugins are distributed as Go binaries, it is important to define this entry-point with the following code:

```
package main

import (
    "github.com/hashicorp/terraform/plugin"
    "github.com/hashicorp/terraform/terraform"
)

func main() {
    plugin.Serve(&plugin.ServeOpts{
        ProviderFunc: func() terraform.ResourceProvider {
            return Provider()
        },
    })
}
```

This establishes the main function to produce a valid, executable Go binary. The contents of the main function consume Terraform's plugin library. This library deals with all the communication between Terraform core and the plugin.

Next, build the plugin using the Go toolchain:

```
$ go build -o terraform-provider-example
```

The output name (`-o`) is **very important**. Terraform searches for plugins in the format of:

```
terraform-<TYPE>-<NAME>
```

In the case above, the plugin is of type "provider" and of name "example".

To verify things are working correctly, execute the binary just created:

```
$ ./terraform-provider-example
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically
```

Custom built providers can be sideloaded ([/docs/configuration/providers.html#third-party-plugins](#)) for Terraform to use.

This is the basic project structure and scaffolding for a Terraform plugin. To recap, the file structure is:

```
.  
└── main.go  
└── provider.go
```

Defining Resources

Terraform providers manage resources. A provider is an abstraction of an upstream API, and a resource is a component of that provider. As an example, the AWS provider supports `aws_instance` and `aws_elastic_ip`. DNSimple supports `dnsimple_record`. Fastly supports `fastly_service`. Let's add a resource to our fictitious provider.

As a general convention, Terraform providers put each resource in their own file, named after the resource, prefixed with `resource_`. To create an `example_server`, this would be `resource_server.go` by convention:

```
package main

import (
    "github.com/hashicorp/terraform/helper/schema"
)

func resourceServer() *schema.Resource {
    return &schema.Resource{
        Create: resourceServerCreate,
        Read:   resourceServerRead,
        Update: resourceServerUpdate,
        Delete: resourceServerDelete,

        Schema: map[string]*schema.Schema{
            "address": &schema.Schema{
                Type:     schema.TypeString,
                Required: true,
            },
        },
    }
}
```

This uses the `schema.Resource` type (<https://godoc.org/github.com/hashicorp/terraform/helper/schema#Resource>). This structure defines the data schema and CRUD operations for the resource. Defining these properties are the only required thing to create a resource.

The schema above defines one element, `"address"`, which is a required string. Terraform's schema automatically enforces validation and type casting.

Next there are four "fields" defined - Create, Read, Update, and Delete. The Create, Read, and Delete functions are required for a resource to be functional. There are other functions, but these are the only required ones. Terraform itself handles which function to call and with what data. Based on the schema and current state of the resource, Terraform can determine whether it needs to create a new resource, update an existing one, or destroy. The create and update function should always return the read function to ensure the state is reflected in the `terraform.state` file.

Each of the four struct fields point to a function. While it is technically possible to inline all functions in the resource schema, best practice dictates pulling each function into its own method. This optimizes for both testing and readability. Fill in those stubs now, paying close attention to method signatures.

```
func resourceServerCreate(d *schema.ResourceData, m interface{}) error {
    return resourceServerRead(d, m)
}

func resourceServerRead(d *schema.ResourceData, m interface{}) error {
    return nil
}

func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
    return resourceServerRead(d, m)
}

func resourceServerDelete(d *schema.ResourceData, m interface{}) error {
    return nil
}
```

Lastly, update the provider schema in `provider.go` to register this new resource.

```
func Provider() *schema.Provider {
    return &schema.Provider{
        ResourcesMap: map[string]*schema.Resource{
            "example_server": resourceServer(),
        },
    }
}
```

Build and test the plugin. Everything should compile as-is, although all operations are a no-op.

```
$ go build -o terraform-provider-example
$ ./terraform-provider-example
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically
```

The layout now looks like this:

```
.
├── main.go
└── provider.go
├── resource_server.go
└── terraform-provider-example
```

Invoking the Provider

Previous sections showed running the provider directly via the shell, which outputs a warning message like:

```
This binary is a plugin. These are not meant to be executed directly.  
Please execute the program that consumes these plugins, which will  
load any plugins automatically
```

Terraform plugins should be executed by Terraform directly. To test this, create a `main.tf` in the working directory (the same place where the plugin exists).

```
resource "example_server" "my-server" {}
```

When `terraform init` is run, Terraform parses configuration files and searches for providers in several locations. For the convenience of plugin developers, this search includes the current working directory. (For full details, see [How Terraform Works: Plugin Discovery](#) (`/docs/extend/how-terraform-works.html#discovery`)).

Run `terraform init` to discover our newly compiled provider:

```
$ terraform init  
  
Initializing provider plugins...  
  
Terraform has been successfully initialized!  
  
You may now begin working with Terraform. Try running "terraform plan" to see  
any changes that are required for your infrastructure. All Terraform commands  
should now work.  
  
If you ever set or change modules or backend configuration for Terraform,  
rerun this command to reinitialize your working directory. If you forget, other  
commands will detect it and remind you to do so if necessary.
```

Now execute `terraform plan`:

```
$ terraform plan  
  
1 error(s) occurred:  
  
* example_server.my-server: "address": required field is not set
```

This validates Terraform is correctly delegating work to our plugin and that our validation is working as intended. Fix the validation error by adding an address field to the resource:

```
resource "example_server" "my-server" {  
    address = "1.2.3.4"  
}
```

Execute `terraform plan` to verify the validation is passing:

```
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
```

```
-----  
An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:
```

```
+ create
```

```
Terraform will perform the following actions:
```

```
+ example_server.my-server
  id:      <computed>
  address: "1.2.3.4"
```

```
Plan: 1 to add, 0 to change, 0 to destroy.
```

```
-----  
It is possible to run terraform apply, but it will be a no-op because all of the resource options currently take no action.
```

Implement Create

Back in `resource_server.go`, implement the create functionality:

```
func resourceServerCreate(d *schema.ResourceData, m interface{}) error {
    address := d.Get("address").(string)
    d.SetId(address)
    return resourceServerRead(d, m)
}
```

This uses the `schema.ResourceData` API

(<https://godoc.org/github.com/hashicorp/terraform/helper/schema#ResourceData>) to get the value of "address" provided by the user in the Terraform configuration. Due to the way Go works, we have to typecast it to string. This is a safe operation, however, since our schema guarantees it will be a string type.

Next, it uses `SetId`, a built-in function, to set the ID of the resource to the address. The existence of a non-blank ID is what tells Terraform that a resource was created. This ID can be any string value, but should be a value that can be used to read the resource again.

Finally, we must recompile the binary and instruct Terraform to reinitialize it by rerunning `terraform init`. This is only necessary because we have modified the code and recompiled the binary, and it no longer matches an internal hash Terraform uses to ensure the same binaries are used for each operation.

Run `terraform init`, and then run `terraform plan`.

```
$ go build -o terraform-provider-example
$ terraform init
# ...
```

```
$ terraform plan

+ example_server.my-server
  address: "1.2.3.4"

Plan: 1 to add, 0 to change, 0 to destroy.
```

Terraform will ask for confirmation when you run `terraform apply`. Enter yes to create your example server and commit it to state:

```
$ terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

+ example_server.my-server
  id:      <computed>
  address: "1.2.3.4"

Plan: 1 to add, 0 to change, 0 to destroy.
```

```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value: yes
```

```
example_server.my-server: Creating...
  address: "" => "1.2.3.4"
example_server.my-server: Creation complete after 0s (ID: 1.2.3.4)
```

Since the Create operation used `SetId`, Terraform believes the resource created successfully. Verify this by running `terraform plan`.

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

example_server.my-server: Refreshing state... (ID: 1.2.3.4)

-----
No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
```

Again, because of the call to `SetId`, Terraform believes the resource was created. When running `plan`, Terraform properly determines there are no changes to apply.

To verify this behavior, change the value of the `address` field and run `terraform plan` again. You should see output like

this:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

example_server.my-server: Refreshing state... (ID: 1.2.3.4)

-----
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
~ update in-place

Terraform will perform the following actions:

~ example_server.my-server
  address: "1.2.3.4" => "5.6.7.8"

Plan: 0 to add, 1 to change, 0 to destroy.

-----
Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.
```

Terraform detects the change and displays a diff with a ~ prefix, noting the resource will be modified in place, rather than created new.

Run `terraform apply` to apply the changes. Terraform will again prompt for confirmation:

```
$ terraform apply
example_server.my-server: Refreshing state... (ID: 1.2.3.4)

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
~ update in-place

Terraform will perform the following actions:

~ example_server.my-server
  address: "1.2.3.4" => "5.6.7.8"

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value: yes

example_server.my-server: Modifying... (ID: 1.2.3.4)
  address: "1.2.3.4" => "5.6.7.8"
example_server.my-server: Modifications complete after 0s (ID: 1.2.3.4)

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Since we did not implement the `Update` function, you would expect the `terraform plan` operation to report changes, but it does not! How were our changes persisted without the `Update` implementation?

Error Handling & Partial State

Previously our `Update` operation succeeded and persisted the new state with an empty function definition. Recall the current `update` function:

```
func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
    return resourceServerRead(d, m)
}
```

The `return nil` tells Terraform that the update operation succeeded without error. Terraform assumes this means any changes requested applied without error. Because of this, our state updated and Terraform believes there are no further changes.

To say it another way: if a callback returns no error, Terraform automatically assumes the entire diff successfully applied, merges the diff into the final state, and persists it.

Functions should *never* intentionally panic or call `os.Exit` - always return an error.

In reality, it is a bit more complicated than this. Imagine the scenario where our update function has to update two separate fields which require two separate API calls. What do we do if the first API call succeeds but the second fails? How do we properly tell Terraform to only persist half the diff? This is known as a *partial state* scenario, and implementing these properly is critical to a well-behaving provider.

Here are the rules for state updating in Terraform. Note that this mentions callbacks we have not discussed, for the sake of completeness.

- If the `Create` callback returns with or without an error without an ID set using `SetId`, the resource is assumed to not be created, and no state is saved.
- If the `Create` callback returns with or without an error and an ID has been set, the resource is assumed created and all state is saved with it. Repeating because it is important: if there is an error, but the ID is set, the state is fully saved.
- If the `Update` callback returns with or without an error, the full state is saved. If the ID becomes blank, the resource is destroyed (even within an update, though this shouldn't happen except in error scenarios).
- If the `Destroy` callback returns without an error, the resource is assumed to be destroyed, and all state is removed.
- If the `Destroy` callback returns with an error, the resource is assumed to still exist, and all prior state is preserved.
- If partial mode (covered next) is enabled when a create or update returns, only the explicitly enabled configuration keys are persisted, resulting in a partial state.

Partial mode is a mode that can be enabled by a callback that tells Terraform that it is possible for partial state to occur. When this mode is enabled, the provider must explicitly tell Terraform what is safe to persist and what is not.

Here is an example of a partial mode with an `update` function:

```

func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
    // Enable partial state mode
    d.Partial(true)

    if d.HasChange("address") {
        // Try updating the address
        if err := updateAddress(d, m); err != nil {
            return err
        }

        d.SetPartial("address")
    }

    // If we were to return here, before disabling partial mode below,
    // then only the "address" field would be saved.

    // We succeeded, disable partial mode. This causes Terraform to save
    // all fields again.
    d.Partial(false)

    return resourceServerRead(d, m)
}

```

Note - this code will not compile since there is no updateAddress function. You can implement a dummy version of this function to play around with partial state. For this example, partial state does not mean much in this documentation example. If updateAddress were to fail, then the address field would not be updated.

Implementing Destroy

The Destroy callback is exactly what it sounds like - it is called to destroy the resource. This operation should never update any state on the resource. It is not necessary to call d.SetId(""), since any non-error return value assumes the resource was deleted successfully.

```

func resourceServerDelete(d *schema.ResourceData, m interface{}) error {
    // d.SetId("") is automatically called assuming delete returns no errors, but
    // it is added here for explicitness.
    d.SetId("")
    return nil
}

```

The destroy function should always handle the case where the resource might already be destroyed (manually, for example). If the resource is already destroyed, this should not return an error. This allows Terraform users to manually delete resources without breaking Terraform. Recompile and reinitialize the Provider:

```

$ go build -o terraform-provider-example
$ terraform init
#...

```

Run `terraform destroy` to destroy the resource.

```

$ terraform destroy
example_server.my-server: Refreshing state... (ID: 5.6.7.8)

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
- destroy

Terraform will perform the following actions:

- example_server.my-server

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy?
Terraform will destroy all your managed infrastructure, as shown above.
There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

example_server.my-server: Destroying... (ID: 5.6.7.8)
example_server.my-server: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.

```

Implementing Read

The Read callback is used to sync the local state with the actual state (upstream). This is called at various points by Terraform and should be a read-only operation. This callback should never modify the real resource.

If the ID is updated to blank, this tells Terraform the resource no longer exists (maybe it was destroyed out of band). Just like the destroy callback, the Read function should gracefully handle this case.

```

func resourceServerRead(d *schema.ResourceData, m interface{}) error {
    client := m.(*MyClient)

    // Attempt to read from an upstream API
    obj, ok := client.Get(d.Id())

    // If the resource does not exist, inform Terraform. We want to immediately
    // return here to prevent further processing.
    if !ok {
        d.SetId("")
        return nil
    }

    d.Set("address", obj.Address)
    return nil
}

```

Implementing a more complex Read

Often the resulting data structure from the API is more complicated and contains nested structures. The following example illustrates this fact. The goal is that the `terraform.state` maps the resulting data structure as close as possible. This mapping is called **flattening** whereas mapping the terraform configuration to an API call, e.g. on create is called **expanding**.

This example illustrates the flattening of a nested structure which contains a `TypeSet` and `TypeMap`.

Considering the following structure as the response from the API:

```
{  
  "ID": "ozfsuj7dblzwjo8zoguosr1l5",  
  "Spec": {  
    "Name": "tf-test-service-basic",  
    "Labels": {},  
    "Address": "tf-test-address",  
    "TaskTemplate": {  
      "ContainerSpec": {  
        "Mounts": [  
          {  
            "Type": "volume",  
            "Source": "tf-test-volume",  
            "Target": "/mount/test",  
            "VolumeOptions": {  
              "NoCopy": true,  
              "DriverConfig": {}  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

The nested structures are `Spec` -> `TaskTemplate` -> `ContainerSpec` -> `Mounts`. There can be multiple `Mounts` but they have to be unique, so `TypeSet` is the appropriate type.

Due to the limitation of tf-11115 (<https://github.com/hashicorp/terraform/issues/11115>) it is not possible to nest maps. So the workaround is to let only the innermost data structure be of the type `TypeMap`: in this case `driver_options`. The outer data structures are of `TypeList` which can only have one item.

```
/// ...  
"task_spec": &schema.Schema{  
  Type:     schema.TypeList,  
  MaxItems: 1,  
  Required: true,  
  Elems: &schema.Resource{  
    Schema: map[string]*schema.Schema{  
      "container_spec": &schema.Schema{  
        Type:     schema.TypeList,  
        Required: true,  
        MaxItems: 1,  
        Elems: &schema.Resource{  
          Schema: map[string]*schema.Schema{  
            "mounts": &schema.Schema{  
              Type:     schema.TypeSet,  
              Optional: true,  
              Elems: &schema.Resource{  
                Schema: map[string]*schema.Schema{  
                  "target": &schema.Schema{  
                    Type:     schema.TypeString,  
                  }  
                }  
              }  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

```

        Required:    true,
    },
    "source": &schema.Schema{
        Type:          schema.TypeString,
        Required:    true,
    },
    "type": &schema.Schema{
        Type:          schema.TypeString,
        Required:    true,
    },
    "volume_options": &schema.Schema{
        Type:          schema.TypeList,
        Optional:    true,
        MaxItems:    1,
        Elem: &schema.Resource{
            Schema: map[string]*schema.Schema{
                "no_copy": &schema.Schema{
                    Type:          schema.TypeBool,
                    Optional:    true,
                },
                "labels": &schema.Schema{
                    Type:          schema.TypeMap,
                    Optional:    true,
                    Elem: &schema.Schema{Type: schema.TypeString},
                },
                "driver_name": &schema.Schema{
                    Type:          schema.TypeString,
                    Optional:    true,
                },
                "driver_options": &schema.Schema{
                    Type:          schema.TypeMap,
                    Optional:    true,
                    Elem: &schema.Schema{Type: schema.TypeString},
                },
            },
        },
    },
    },
    },
    },
    },
    },
    },
    },
    },
    }
}

```

The resourceServerRead function now also sets/flattens the nested data structure from the API:

```
func resourceServerRead(d *schema.ResourceData, m interface{}) error {
    client := m.(*MyClient)
    server, ok := client.Get(d.Id())

    if !ok {
        log.Printf("[WARN] No Server found: %s", d.Id())
        d.SetId("")
        return nil
    }

    d.Set("address", server.Address)

    if server.Spec != nil && server.Spec.TaskTemplate != nil {
        if err = d.Set("task_spec", flattenTaskSpec(server.Spec.TaskTemplate)); err != nil {
            return err
        }
    }

    return nil
}
```

The so-called flatteners are in a separate file `structures_server.go`. The outermost data structure is a `map[string]interface{}` and each item a `[]interface{}`:

```

func flattenTaskSpec(in *server.TaskSpec) []interface{} {
    // NOTE: the top level structure to set is a map
    m := make(map[string]interface{})
    if in.ContainerSpec != nil {
        m["container_spec"] = flattenContainerSpec(in.ContainerSpec)
    }
    /// ...

    return []interface{}{m}
}

func flattenContainerSpec(in *server.ContainerSpec) []interface{} {
    // NOTE: all nested structures are lists of interface{}
    var out = make([]interface{}, 0, 0)
    m := make(map[string]interface{})
    /// ...
    if len(in.Mounts) > 0 {
        m["mounts"] = flattenServiceMounts(in.Mounts)
    }
    /// ...
    out = append(out, m)
    return out
}

func flattenServiceMounts(in []mount.Mount) []map[string]interface{} {
    var out = make([]map[string]interface{}, len(in), len(in))
    for i, v := range in {
        m := make(map[string]interface{})
        m["target"] = v.Target
        m["source"] = v.Source
        m["type"] = v.Type

        if v.VolumeOptions != nil {
            volumeOptions := make(map[string]interface{})

            volumeOptions["no_copy"] = v.VolumeOptions.NoCopy
            // NOTE: this is an internally written map from map[string]string => map[string]interface{}
            // because terraform can only store map with interface{} as the type of the value
            volumeOptions["labels"] = mapStringStringToMapStringInterface(v.VolumeOptions.Labels)
            if v.VolumeOptions.DriverConfig != nil {
                volumeOptions["driver_name"] = v.VolumeOptions.DriverConfig.Name
                volumeOptionsItem["driver_options"] = mapStringStringToMapStringInterface(v.VolumeOptions.DriverConfig.Options)
            }

            m["volume_options"] = []interface{}{volumeOptions}
        }

        out[i] = m
    }
    return out
}

```

Next Steps

This guide covers the schema and structure for implementing a Terraform provider using the provider framework. As next steps, reference the internal providers for examples. Terraform also includes a full framework for testing providers.

General Rules

Dedicated Upstream Libraries

One of the biggest mistakes new users make is trying to conflate a client library with the Terraform implementation. Terraform should always consume an independent client library which implements the core logic for communicating with the upstream. Do not try to implement this type of logic in the provider itself.

Data Sources

While not explicitly discussed here, *data sources* are a special subset of resources which are read-only. They are resolved earlier than regular resources and can be used as part of Terraform's interpolation.

Terraform GitHub Actions

This is the documentation for Terraform's GitHub Actions.

GitHub Actions (<https://developer.github.com/actions>) allow you to run commands in reaction to GitHub events. Terraform's GitHub Actions are designed to run on new and updated Pull Requests to help you review and validate Terraform changes.

If you are new to Terraform's GitHub Actions, begin with the Getting Started Guide (</docs/github-actions/getting-started/index.html>).

Here are some example outputs from Terraform's GitHub Actions:

Terraform Fmt

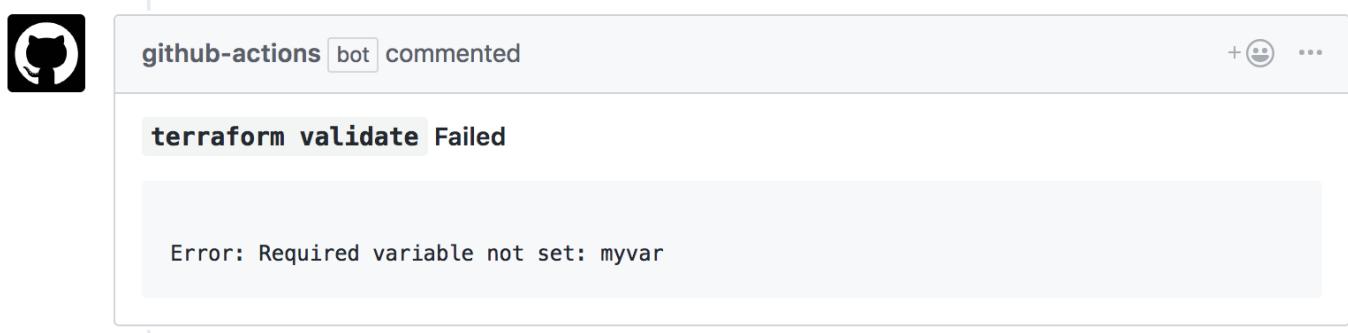


A screenshot of a GitHub pull request comment. The comment is from the bot "github-actions" and has the label "commented". The message says "terraform fmt Failed". It shows a diff of a file named "main.tf". The diff highlights a change in the "count" value of a resource block. The original line "- count = "\${var.myvar}" is shown in red, and the new line "+ count = "\${var.myvar}" is shown in green. The file also contains a "variable" block for "myvar".

```
resource "null_resource" "test" {
- count = "${var.myvar}"
+ count = "${var.myvar}"
+}

variable "myvar" {}
```

Terraform Validate



A screenshot of a GitHub pull request comment. The comment is from the bot "github-actions" and has the label "commented". The message says "terraform validate Failed". It shows an error message: "Error: Required variable not set: myvar".

```
Error: Required variable not set: myvar
```

Terraform Plan



github-actions [bot] commented

+ 😊 ...

Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be persisted to local or remote state storage.

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

```
+ null_resource.test  
      id: <computed>  
Plan: 1 to add, 0 to change, 0 to destroy.
```

Actions

Terraform's GitHub Actions can be used individually or together.

See the links below for full documentation of each Action.

- [fmt \(/docs/github-actions/actions/fmt.html\)](#)
- [init \(/docs/github-actions/actions/init.html\)](#)
- [validate \(/docs/github-actions/actions/validate.html\)](#)
- [plan \(/docs/github-actions/actions/plan.html\)](#)

Terraform Fmt Action

Runs `terraform fmt` to validate all Terraform files in a directory are in the canonical format. If any files differ, this action will comment back on the pull request with the diffs of each file.

Success Criteria

This action succeeds if `terraform fmt` runs without error.

Usage

To use the `fmt` action, add it to your workflow file.

```
action "terraform fmt" {
  # Replace <latest tag> with the latest tag from
  # https://github.com/hashicorp/terraform-github-actions/releases.
  uses = "hashicorp/terraform-github-actions/fmt@<latest tag>

  # See Environment Variables below for details.
  env = {
    TF_ACTION_WORKING_DIR = "."
  }

  # We need the GitHub token to be able to comment back on the pull request.
  secrets = ["GITHUB_TOKEN"]
}
```

Environment Variables

| Name | Default | Description |
|------------------------------------|---------------------|---|
| <code>TF_ACTION_WORKING_DIR</code> | <code>."</code> | Which directory <code>fmt</code> runs in. Relative to the root of the repo. |
| <code>TF_ACTION_COMMENT</code> | <code>"true"</code> | Set to <code>"false"</code> to disable commenting back on pull request with the diffs of unformatted files. |

Secrets

The `GITHUB_TOKEN` secret is required for posting a comment back to the pull request if `fmt` fails.

If you have set `TF_ACTION_COMMENT = "false"`, then `GITHUB_TOKEN` is not required.

Arguments

Any arguments will be appended to the `terraform fmt` command; however, we do not anticipate that this will be needed.

Terraform Init Action

Runs `terraform init` to initialize a Terraform working directory and to confirm that any backends, modules, and providers are configured correctly. This action will comment back on the pull request on failure.

Success Criteria

This action succeeds if `terraform init` runs without error.

Usage

To use the `init` action, add it to your workflow file.

```
action "terraform init" {
  # Replace <latest tag> with the latest tag from
  # https://github.com/hashicorp/terraform-github-actions/releases.
  uses = "hashicorp/terraform-github-actions/init@<latest tag>

  # See Environment Variables below for details.
  env = {
    TF_ACTION_WORKING_DIR = "."
  }

  # We need the GitHub token to be able to comment back on the pull request.
  secrets = ["GITHUB_TOKEN"]
}
```

Environment Variables

| Name | Default | Description |
|-----------------------|---------|--|
| TF_ACTION_WORKING_DIR | "." | Which directory <code>init</code> runs in. Relative to the root of the repo. |
| TF_ACTION_COMMENT | "true" | Set to "false" to disable commenting back on pull on error. |

Secrets

The `GITHUB_TOKEN` secret is required for posting a comment back to the pull request if `init` fails.

If you have set `TF_ACTION_COMMENT = "false"`, then `GITHUB_TOKEN` is not required.

Arguments

Arguments to `init` will be appended to the `terraform init` command:

```
action "terraform init" {
  ...
  args = ["-lock=false"]
}
```

Terraform Plan Action

Runs `terraform plan` and comments back on the pull request with the plan output.

Success Criteria

This action succeeds if `terraform plan` runs without error.

Usage

To use the plan action, add it to your workflow file.

```
action "terraform plan" {
  # Replace <latest tag> with the latest tag from
  # https://github.com/hashicorp/terraform-github-actions/releases.
  uses = "hashicorp/terraform-github-actions/plan@<latest tag>

  # `terraform plan` will always fail unless `terraform init` is run first.
  needs = "terraform init"

  # See Environment Variables below for details.
  env = {
    TF_ACTION_WORKING_DIR = "."
    TF_ACTION_WORKSPACE   = "default"
  }

  # If you need to specify additional arguments to terraform plan, add them here.
  # Otherwise, delete this line or leave the array empty.
  args = ["-var", "foo=bar"]

  # We need the GitHub token to be able to comment back on the pull request.
  secrets = ["GITHUB_TOKEN"]
}

action "terraform init" {
  uses = "hashicorp/terraform-github-actions/init@<latest tag>"
  secrets = ["GITHUB_TOKEN"]
}
```

Environment Variables

| Name | Default | Description |
|------------------------------------|------------------------|--|
| <code>TF_ACTION_WORKING_DIR</code> | <code>".."</code> | Which directory <code>plan</code> runs in. Relative to the root of the repo. |
| <code>TF_ACTION_COMMENT</code> | <code>"true"</code> | Set to <code>"false"</code> to disable commenting back on pull request. |
| <code>TF_ACTION_WORKSPACE</code> | <code>"default"</code> | Which Terraform workspace (/docs/state/workspaces.html) to run in. |

Workspaces

The `plan` action only supports running in a single Terraform workspace (<https://www.terraform.io/docs/state/workspaces.html>). If you need to run `plan` in multiple workspaces, see Workspaces (</docs/github-actions/workspaces.html>).

Secrets

- The `GITHUB_TOKEN` secret is required for posting a comment back to the pull request if `validate` fails. If you have set `TF_ACTION_COMMENT = "false"`, then `GITHUB_TOKEN` is not required.
- You'll also likely need to add secrets for your providers, like `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` or `GOOGLE_CREDENTIALS`.

WARNING These secrets could be exposed if the `plan` action is run on a malicious Terraform file. To avoid this, we recommend you do not use this action on public repos or repos where untrusted users can submit pull requests.

Arguments

Arguments to `plan` will be appended to the `terraform plan` command:

```
action "terraform plan" {
  ...
  args = ["-var", "foo=bar", "-var-file=foo"]
}
```

Terraform Validate Action

Runs `terraform validate` to validate the Terraform files in a directory. Validation includes a basic check of syntax as well as checking that all variables declared in the configuration are specified in one of the possible ways:

- `-var foo=...`
- `-var-file=foo.vars`
- `TF_VAR_foo` environment variable
- `terraform.tfvars`
- default value

Success Criteria

This action succeeds if `terraform validate` runs without error.

Usage

To use the validate action, add it to your workflow file.

```
action "terraform validate" {  
    # Replace <latest tag> with the latest tag from  
    # https://github.com/hashicorp/terraform-github-actions/releases.  
    uses = "hashicorp/terraform-github-actions/validate@<latest tag>"  
  
    # `terraform validate` will always fail unless `terraform init` is run first.  
    needs = "terraform init"  
  
    # See Environment Variables below for details.  
    env = {  
        TF_ACTION_WORKING_DIR = ".."  
    }  
  
    # If you need to specify additional arguments to terraform validate, add them here.  
    # Otherwise, delete this line or leave the array empty.  
    args = ["-var", "foo=bar"]  
  
    # We need the GitHub token to be able to comment back on the pull request.  
    secrets = ["GITHUB_TOKEN"]  
}  
  
action "terraform init" {  
    uses = "hashicorp/terraform-github-actions/init@<latest tag>"  
    secrets = ["GITHUB_TOKEN"]  
}
```

Environment Variables

| Name | Default | Description |
|-----------------------|---------|--|
| TF_ACTION_WORKING_DIR | ". " | Which directory validate runs in. Relative to the root of the repo. |
| TF_ACTION_COMMENT | "true" | Set to "false" to disable commenting back on pull request if validate fails. |

Secrets

The GITHUB_TOKEN secret is required for posting a comment back to the pull request if validate fails.

If you have set TF_ACTION_COMMENT = "false", then GITHUB_TOKEN is not required.

Arguments

Arguments to validate will be appended to the `terraform validate` command:

```
action "terraform validate" {
  ...
  args = [ "-var", "foo=bar", "-var-file=foo" ]
}
```

Directories

Currently, each Terraform GitHub Action only supports running in a single directory. The directory is set by the `TF_ACTION_WORKING_DIR` environment variable.

If you need to run the Terraform Actions in multiple directories, you have to create separate workflows for each directory. For example, here is a set of workflows for running in two directories, `dir1` and `dir2`:

```
workflow "terraform-dir1" {
  resolves = "terraform-plan-dir1"
  on       = "pull_request"
}

action "filter-to-pr-open-synced" {
  uses = "actions/bin/filter@master"
  args = "action 'opened|synchronize'"
}

action "terraform-fmt-dir1" {
  uses      = "hashicorp/terraform-github-actions/fmt@v0.1.1"
  needs    = "filter-to-pr-open-synced"
  secrets  = ["GITHUB_TOKEN"]

  env = {
    TF_ACTION_WORKING_DIR = "dir1"
  }
}

action "terraform-init-dir1" {
  uses      = "hashicorp/terraform-github-actions/init@v0.1.1"
  secrets  = ["GITHUB_TOKEN"]
  needs    = "terraform-fmt-dir1"

  env = {
    TF_ACTION_WORKING_DIR = "dir1"
  }
}

action "terraform-validate-dir1" {
  uses      = "hashicorp/terraform-github-actions/validate@v0.1.1"
  secrets  = ["GITHUB_TOKEN"]
  needs    = "terraform-init-dir1"

  env = {
    TF_ACTION_WORKING_DIR = "dir1"
  }
}

action "terraform-plan-dir1" {
  uses      = "hashicorp/terraform-github-actions/plan@v0.1.1"
  needs    = "terraform-validate-dir1"
  secrets  = ["GITHUB_TOKEN"]

  env = {
    TF_ACTION_WORKING_DIR = "dir1"
  }
}

workflow "terraform-dir2" {
  resolves = "terraform-plan-dir2"
  on       = "pull_request"
}
```

```
action "terraform-fmt-dir2" {
  uses      = "hashicorp/terraform-github-actions/fmt@v0.1.1"
  needs    = "filter-to-pr-open-synced"
  secrets  = ["GITHUB_TOKEN"]

  env = {
    TF_ACTION_WORKING_DIR = "dir2"
  }
}

action "terraform-init-dir2" {
  uses      = "hashicorp/terraform-github-actions/init@v0.1.1"
  secrets  = ["GITHUB_TOKEN"]
  needs    = "terraform-fmt-dir2"

  env = {
    TF_ACTION_WORKING_DIR = "dir2"
  }
}

action "terraform-validate-dir2" {
  uses      = "hashicorp/terraform-github-actions/validate@v0.1.1"
  secrets  = ["GITHUB_TOKEN"]
  needs    = "terraform-init-dir2"

  env = {
    TF_ACTION_WORKING_DIR = "dir2"
  }
}

action "terraform-plan-dir2" {
  uses      = "hashicorp/terraform-github-actions/plan@v0.1.1"
  needs    = "terraform-validate-dir2"
  secrets  = ["GITHUB_TOKEN"]

  env = {
    TF_ACTION_WORKING_DIR = "dir2"
  }
}
```

Getting Started

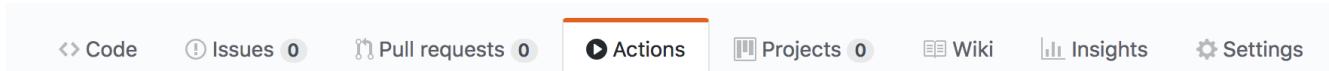
GitHub Actions allow you to trigger commands in reaction to GitHub events. Terraform's GitHub Actions are designed to run on new and updated pull requests to help you review and validate Terraform changes.

Recommended Workflow

The easiest way to get started is to copy our recommended workflow, which runs all of Terraform's GitHub Actions on new and updated pull requests.

Note: If you'd like to write your own custom workflow using our Actions, check out the [Actions Reference](#) (/docs/github-actions/actions/index.html).

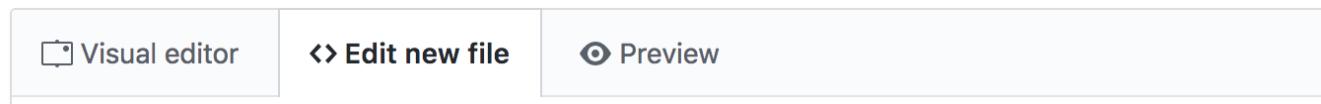
1. Open up your repository in GitHub and click on the **Actions** tab.



2. Click the **Create a new workflow** button.



3. Click the **Edit new file** tab.



4. Replace the default workflow with the following:

```

workflow "Terraform" {
  resolves = "terraform-plan"
  on = "pull_request"
}

action "filter-to-pr-open-synced" {
  uses = "actions/bin/filter@master"
  args = "action 'opened|synchronize'"
}

action "terraform-fmt" {
  uses = "hashicorp/terraform-github-actions/fmt@v0.1.1"
  needs = "filter-to-pr-open-synced"
  secrets = ["GITHUB_TOKEN"]
  env = {
    TF_ACTION_WORKING_DIR = "."
  }
}

action "terraform-init" {
  uses = "hashicorp/terraform-github-actions/init@v0.1.1"
  needs = "terraform-fmt"
  secrets = ["GITHUB_TOKEN"]
  env = {
    TF_ACTION_WORKING_DIR = "."
  }
}

action "terraform-validate" {
  uses = "hashicorp/terraform-github-actions/validate@v0.1.1"
  needs = "terraform-init"
  secrets = ["GITHUB_TOKEN"]
  env = {
    TF_ACTION_WORKING_DIR = "."
  }
}

action "terraform-plan" {
  uses = "hashicorp/terraform-github-actions/plan@v0.1.1"
  needs = "terraform-validate"
  secrets = ["GITHUB_TOKEN"]
  env = {
    TF_ACTION_WORKING_DIR = "."
    # If you're using Terraform workspaces, set this to the workspace name.
    TF_ACTION_WORKSPACE = "default"
  }
}

```

5. **Directories** — If your Terraform configuration is not in the root of your repo, replace all instances of:

```
TF_ACTION_WORKING_DIR = ","
```

...with your directory, relative to the root of the repo. For example:

```
TF_ACTION_WORKING_DIR = "./terraform"
```

If you have multiple directories of Terraform code, see Directories (/docs/github-actions/directories.html).

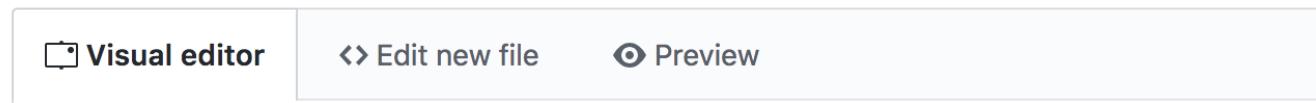
6. **Workspaces** — If your Terraform runs in a different Terraform workspace (/docs/state/workspaces.html) than

default, change the `TF_ACTION_WORKSPACE` environment variable in the `terraform-plan` action.

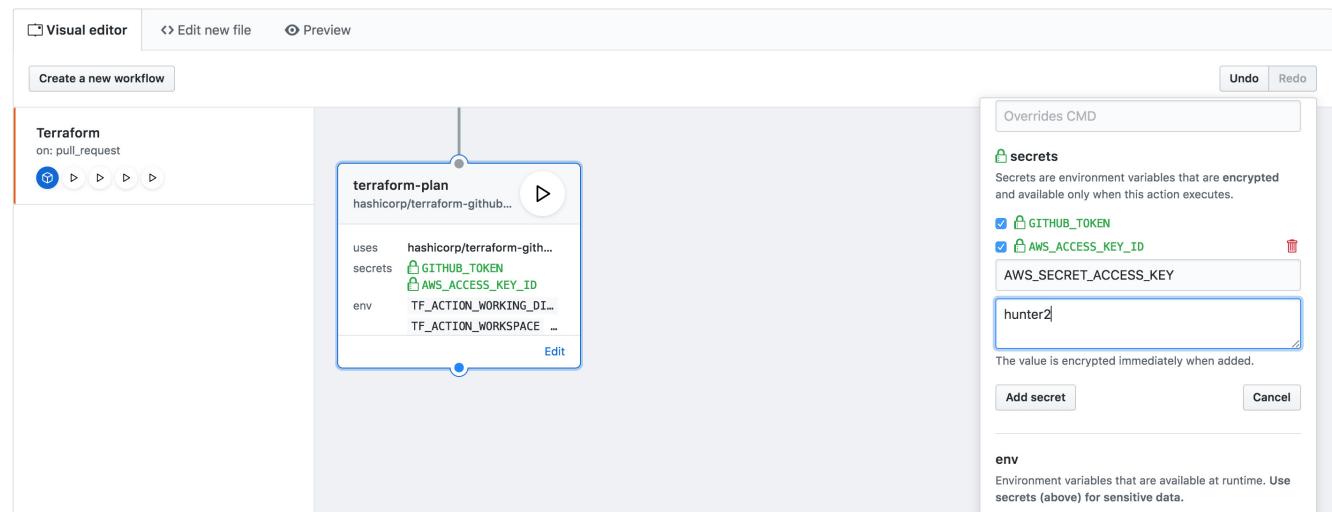
```
TF_ACTION_WORKSPACE = "your-workspace"
```

If you have multiple workspaces, see [Workspaces \(/docs/github-actions/workspaces.html\)](#).

7. **Credentials** — If you're using a Terraform provider that requires credentials to run `terraform plan` (like AWS or Google Cloud Platform) then you need to add those credentials as secrets to the `terraform-plan` action. Secrets can only be added from the **Visual Editor**, so switch to that tab.



Scroll down to the `terraform-plan` action and click **Edit**. This will open the action editor on the right side, where you'll be able to add your secrets as environment variables, like `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. See your provider documentation (<https://www.terraform.io/docs/providers/>) for the specific environment variables your provider needs.



WARNING These secrets could be exposed if the plan action is run on a malicious Terraform file. To avoid this, we recommend you do not use the plan action on public repos or repos where untrusted users can submit pull requests.

8. Click **Start commit** to commit the Workflow.
9. On your next pull request, you should see the Actions running.

Workspaces

Currently, the Terraform Plan Action (/docs/github-actions/actions/plan.html) only supports running in a single Terraform workspace (<https://www.terraform.io/docs/state/workspaces.html>). The workspace is defined by the TF_ACTION_WORKSPACE environment variable.

If you need to run the Terraform Actions in multiple workspaces, you have to create separate workflows for each workspace. For example, here is a set of workflows for running in two workspaces, `workspace1` and `workspace2`:

```
workflow "terraform-workspace1" {
  resolves = "terraform-plan-workspace1"
  on       = "pull_request"
}

workflow "terraform-workspace2" {
  resolves = "terraform-plan-workspace2"
  on       = "pull_request"
}

action "filter-to-pr-open-synced" {
  uses = "actions/bin/filter@master"
  args = "action 'opened|synchronize'"
}

action "terraform-fmt" {
  uses      = "hashicorp/terraform-github-actions/fmt@v0.1.1"
  needs    = "filter-to-pr-open-synced"
  secrets  = ["GITHUB_TOKEN"]
}

action "terraform-init" {
  uses      = "hashicorp/terraform-github-actions/init@v0.1.1"
  secrets  = ["GITHUB_TOKEN"]
  needs    = "terraform-fmt"
}

action "terraform-validate" {
  uses      = "hashicorp/terraform-github-actions/validate@v0.1.1"
  secrets  = ["GITHUB_TOKEN"]
  needs    = "terraform-init"
}

action "terraform-plan-workspace1" {
  uses      = "hashicorp/terraform-github-actions/plan@v0.1.1"
  needs    = "terraform-validate"
  secrets  = ["GITHUB_TOKEN"]

  env = {
    TF_ACTION_WORKSPACE = "workspace1"
  }
}

action "terraform-plan-workspace2" {
  uses      = "hashicorp/terraform-github-actions/plan@v0.1.1"
  needs    = "terraform-validate"
  secrets  = ["GITHUB_TOKEN"]

  env = {
    TF_ACTION_WORKSPACE = "workspace2"
  }
}
```


Import

Terraform is able to import existing infrastructure. This allows you take resources you've created by some other means and bring it under Terraform management.

This is a great way to slowly transition infrastructure to Terraform, or to be able to be confident that you can use Terraform in the future if it potentially doesn't support every feature you need today.

Currently State Only

The current implementation of Terraform import can only import resources into the state (/docs/state). It does not generate configuration. A future version of Terraform will also generate configuration.

Because of this, prior to running `terraform import` it is necessary to write manually a resource configuration block for the resource, to which the imported object will be attached.

While this may seem tedious, it still gives Terraform users an avenue for importing existing resources. A future version of Terraform will fully generate configuration, significantly simplifying this process.

Resource Importability

Each resource in Terraform must implement some basic logic to become importable. As a result, not all Terraform resources are currently importable. For those resources that support import, they are documented at the bottom of each resource documentation page, under the `Import` heading. If you find a resource that you want to import and Terraform reports that it is not importable, please report an issue in the relevant provider repository.

Converting a resource to be importable is also relatively simple, so if you're interested in contributing that functionality, the Terraform team would be grateful.

To make a resource importable, please see the plugin documentation on writing a resource ([/docs/plugins/provider.html](#)).

Import Usage

The `terraform import` command is used to import existing infrastructure.

The command currently can only import one resource at a time. This means you can't yet point Terraform import to an entire collection of resources such as an AWS VPC and import all of it. This workflow will be improved in a future version of Terraform.

To import a resource, first write a resource block for it in your configuration, establishing the name by which it will be known to Terraform:

```
resource "aws_instance" "example" {
  # ...instance configuration...
}
```

The name "example" here is local to the module where it is declared and is chosen by the configuration author. This is distinct from any ID issued by the remote system, which may change over time while the resource name remains constant.

If desired, you can leave the body of the resource block blank for now and return to fill it in once the instance is imported.

Now `terraform import` can be run to attach an existing instance to this resource configuration:

```
$ terraform import aws_instance.example i-abcd1234
```

This command locates the AWS instance with ID `i-abcd1234` and attaches its existing settings, as described by the EC2 API, to the name `aws_instance.example` in the Terraform state.

It is also possible to import to resources in child modules and to single instances of a resource with `count` set. See *Resource Addressing* ([/docs/internals/resource-addressing.html](#)) for more details on how to specify a target resource.

The syntax of the given ID is dependent on the resource type being imported. For example, AWS instances use an opaque ID issued by the EC2 API, but AWS Route53 Zones use the domain name itself. Consult the documentation for each importable resource for details on what form of ID is required.

As a result of the above command, the resource is recorded in the state file. You can now run `terraform plan` to see how the configuration compares to the imported resource, and make any adjustments to the configuration to align with the current (or desired) state of the imported object.

Complex Imports

The above import is considered a "simple import": one resource is imported into the state file. An import may also result in a "complex import" where multiple resources are imported. For example, an AWS security group imports an `aws_security_group` but also one `aws_security_group_rule` for each rule.

In this scenario, the secondary resources will not already exist in configuration, so it is necessary to consult the import output and create a resource block in configuration for each secondary resource. If this is not done, Terraform will plan to destroy the imported objects on the next run.

If you want to rename or otherwise move the imported resources, the state management commands ([/docs/commands/state/index.html](#)) can be used.

Terraform Internals

This section covers the internals of Terraform and explains how plans are generated, the lifecycle of a provider, etc. The goal of this section is to remove any notion of "magic" from Terraform. We want you to be able to trust and understand what Terraform is doing to function.

Note: Knowledge of Terraform internals is not required to use Terraform. If you aren't interested in the internals of Terraform, you may safely skip this section.

Debugging Terraform

Terraform has detailed logs which can be enabled by setting the `TF_LOG` environment variable to any value. This will cause detailed logs to appear on `stderr`.

You can set `TF_LOG` to one of the log levels `TRACE`, `DEBUG`, `INFO`, `WARN` or `ERROR` to change the verbosity of the logs. `TRACE` is the most verbose and it is the default if `TF_LOG` is set to something other than a log level name.

To persist logged output you can set `TF_LOG_PATH` in order to force the log to always be appended to a specific file when logging is enabled. Note that even when `TF_LOG_PATH` is set, `TF_LOG` must be set in order for any logging to be enabled.

If you find a bug with Terraform, please include the detailed log by using a service such as [gist](#).

Interpreting a Crash Log

If Terraform ever crashes (a "panic" in the Go runtime), it saves a log file with the debug logs from the session as well as the panic message and backtrace to `crash.log`. Generally speaking, this log file is meant to be passed along to the developers via a GitHub Issue. As a user, you're not required to dig into this file.

However, if you are interested in figuring out what might have gone wrong before filing an issue, here are the basic details of how to read a crash log.

The most interesting part of a crash log is the panic message itself and the backtrace immediately following. So the first thing to do is to search the file for `panic:`, which should jump you right to this message. It will look something like this:

```
panic: runtime error: invalid memory address or nil pointer dereference

goroutine 123 [running]:
panic(0xabc100, 0xd93000a0a0)
    /opt/go/src/runtime/panic.go:464 +0x3e6
github.com/hashicorp/terraform/builtin/providers/aws.resourceAwsSomeResourceCreate(...)
    /opt/gopath/src/github.com/hashicorp/terraform/builtin/providers/aws/resource_aws_some_resource.go:12
3 +0x123
github.com/hashicorp/terraform/helper/schema.(*Resource).Refresh(...)
    /opt/gopath/src/github.com/hashicorp/terraform/helper/schema/resource.go:209 +0x123
github.com/hashicorp/terraform/helper/schema.(*Provider).Refresh(...)
    /opt/gopath/src/github.com/hashicorp/terraform/helper/schema/provider.go:187 +0x123
github.com/hashicorp/terraform/rpc.(*ResourceProviderServer).Refresh(...)
    /opt/gopath/src/github.com/hashicorp/terraform/rpc/resource_provider.go:345 +0x6a
reflect.Value.call(...)
    /opt/go/src/reflect/value.go:435 +0x120d
reflect.Value.Call(...)
    /opt/go/src/reflect/value.go:303 +0xb1
net/rpc.(*service).call(...)
    /opt/go/src/net/rpc/server.go:383 +0x1c2
created by net/rpc.(*Server).ServeCodec
    /opt/go/src/net/rpc/server.go:477 +0x49d
```

The key part of this message is the first two lines that involve `hashicorp/terraform`. In this example:

```
github.com/hashicorp/terraform/builtin/providers/aws.resourceAwsSomeResourceCreate(...)
    /opt/gopath/src/github.com/hashicorp/terraform/builtin/providers/aws/resource_aws_some_resource.go:12
3 +0x123
```

The first line tells us that the method that failed is `resourceAwsSomeResourceCreate`, which we can deduce that involves the creation of a (fictional) `aws_some_resource`.

The second line points to the exact line of code that caused the panic, which--combined with the panic message itself--is normally enough for a developer to quickly figure out the cause of the issue.

As a user, this information can help work around the problem in a pinch, since it should hopefully point to the area of the code base in which the crash is happening.

Resource Graph

Terraform builds a dependency graph (https://en.wikipedia.org/wiki/Dependency_graph) from the Terraform configurations, and walks this graph to generate plans, refresh state, and more. This page documents the details of what are contained in this graph, what types of nodes there are, and how the edges of the graph are determined.

Advanced Topic! This page covers technical details of Terraform. You don't need to understand these details to effectively use Terraform. The details are documented here for those who wish to learn about them without having to go spelunking through the source code.

For some background on graph theory, and a summary of how Terraform applies it, see the HashiCorp 2016 presentation *Applying Graph Theory to Infrastructure as Code* (<https://www.youtube.com/watch?v=Ce3RNfRbdZ0>). This presentation also covers some similar ideas to the following guide.

Graph Nodes

There are only a handful of node types that can exist within the graph. We'll cover these first before explaining how they're determined and built:

- **Resource Node** - Represents a single resource. If you have the `count` metaparameter set, then there will be one resource node for each count. The configuration, diff, state, etc. of the resource under change is attached to this node.
- **Provider Configuration Node** - Represents the time to fully configure a provider. This is when the provider configuration block is given to a provider, such as AWS security credentials.
- **Resource Meta-Node** - Represents a group of resources, but does not represent any action on its own. This is done for convenience on dependencies and making a prettier graph. This node is only present for resources that have a `count` parameter greater than 1.

When visualizing a configuration with `terraform graph`, you can see all of these nodes present.

Building the Graph

Building the graph is done in a series of sequential steps:

1. Resources nodes are added based on the configuration. If a diff (plan) or state is present, that meta-data is attached to each resource node.
2. Resources are mapped to provisioners if they have any defined. This must be done after all resource nodes are created so resources with the same provisioner type can share the provisioner implementation.
3. Explicit dependencies from the `depends_on` meta-parameter are used to create edges between resources.
4. If a state is present, any "orphan" resources are added to the graph. Orphan resources are any resources that are no longer present in the configuration but are present in the state file. Orphans never have any configuration associated with them, since the state file does not store configuration.
5. Resources are mapped to providers. Provider configuration nodes are created for these providers, and edges are created such that the resources depend on their respective provider being configured.

6. Interpolations are parsed in resource and provider configurations to determine dependencies. References to resource attributes are turned into dependencies from the resource with the interpolation to the resource being referenced.
7. Create a root node. The root node points to all resources and is created so there is a single root to the dependency graph. When traversing the graph, the root node is ignored.
8. If a diff is present, traverse all resource nodes and find resources that are being destroyed. These resource nodes are split into two: one node that destroys the resource and another that creates the resource (if it is being recreated). The reason the nodes must be split is because the destroy order is often different from the create order, and so they can't be represented by a single graph node.
9. Validate the graph has no cycles and has a single root.

Walking the Graph

To walk the graph, a standard depth-first traversal is done. Graph walking is done in parallel: a node is walked as soon as all of its dependencies are walked.

The amount of parallelism is limited using a semaphore to prevent too many concurrent operations from overwhelming the resources of the machine running Terraform. By default, up to 10 nodes in the graph will be processed concurrently. This number can be set using the `-parallelism` flag on the plan ([/docs/commands/plan.html](#)), apply ([/docs/commands/apply.html](#)), and destroy ([/docs/commands/destroy.html](#)) commands.

Setting `-parallelism` is considered an advanced operation and should not be necessary for normal usage of Terraform. It may be helpful in certain special use cases or to help debug Terraform issues.

Note that some providers (AWS, for example), handle API rate limiting issues at a lower level by implementing graceful backoff/retry in their respective API clients. For this reason, Terraform does not use this `parallelism` feature to address API rate limits directly.

Internal Plugins

Terraform providers and provisioners are provided via plugins. Each plugin provides an implementation for a specific service, such as AWS, or provisioner, such as bash. Plugins are executed as a separate process and communicate with the main Terraform binary over an RPC interface.

Upgrading From Versions Earlier Than 0.7

In versions of Terraform prior to 0.7, each plugin shipped as a separate binary. In versions of Terraform ≥ 0.7 , all of the official plugins are shipped as a single binary. This saves a lot of disk space and makes downloads faster for you!

However, when you upgrade you will need to manually delete old plugins from disk. You can do this via something like this, depending on where you installed `terraform`:

```
rm /usr/local/bin/terraform-*
```

If you don't do this you will see an error message like the following:

```
[WARN] /usr/local/bin/terraform-provisioner-file overrides an internal plugin for file-provisioner.  
If you did not expect to see this message you will need to remove the old plugin.  
See https://www.terraform.io/docs/internals/plugins.html  
Error configuring: 2 error(s) occurred:  
  
* Unrecognized remote plugin message: 2|unix|/var/folders/pj/66q7ztvd17v_vgfg8c99gm1m0000gn/T/tf-plugin60  
4337945  
  
This usually means that the plugin is either invalid or simply  
needs to be recompiled to support the latest protocol.  
* Unrecognized remote plugin message: 2|unix|/var/folders/pj/66q7ztvd17v_vgfg8c99gm1m0000gn/T/tf-plugin64  
7987867  
  
This usually means that the plugin is either invalid or simply  
needs to be recompiled to support the latest protocol.
```

Why Does This Happen?

In previous versions of Terraform all of the plugins were included in a zip file. For example, when you upgraded from 0.6.12 to 0.6.15, the newer version of each plugin file would have replaced the older one on disk, and you would have ended up with the latest version of each plugin.

Going forward there is only one file in the distribution so you will need to perform a one-time cleanup when upgrading from Terraform < 0.7 to Terraform 0.7 or higher.

If you're curious about the low-level details, keep reading!

Go Plugin Architecture

Terraform is written in the Go programming language. One of Go's interesting properties is that it produces statically-compiled binaries. This means that it does not need to find libraries on your computer to run, and in general only needs to be compatible with your operating system (to make system calls) and with your CPU architecture (so the assembly instructions match the CPU you're running on).

Another property of Go is that it does not support dynamic libraries. It *only* supports static binaries. This is part of Go's overall design and is the reason why it produces statically-compiled binaries in the first place -- once you have a Go binary for your platform it should *Just Work*.

In other languages, plugins are built using dynamic libraries. Since this is not an option for us in Go we use a network RPC interface instead. This means that each plugin is an independent program, and instead of communicating via shared memory, the main process communicates with the plugin process over HTTP. When you start Terraform, it identifies the plugin you want to use, finds it on disk, runs the other binary, and does some handshaking to make sure they can talk to each other (the error you may see after upgrading is a handshake failure in the RPC code).

Downsides

There is a significant downside to this approach. Statically compiled binaries are much larger than dynamically-linked binaries because they include everything they need to run. And because Terraform shares a lot of code with its plugins, there is a lot of binary data duplicated between each of these programs.

In Terraform 0.6.15 there were 42 programs in total, using around 750MB on disk. And it turns out that about 600MB of this is duplicate data! This uses up a lot of space on your hard drive and a lot of bandwidth on our CDN. Fortunately, there is a way to resolve this problem.

Our Solution

In Terraform 0.7 we merged all of the programs into the same binary. We do this by using a special command `terraform internal-plugin` which allows us to invoke a plugin just by calling the same Terraform binary with extra arguments. In essence, Terraform now just calls itself in order to activate the special behavior in each plugin.

Supporting our Community

Why would you do this? Why not just eliminate the network RPC interface and simplify everything?

Terraform is an open source project with a large community, and while we maintain a wide range of plugins as part of the core distribution, we also want to make it easy for people anywhere to write and use their own plugins.

By using the network RPC interface, you can build and distribute a plugin for Terraform without having to rebuild Terraform itself. This makes it easy for you to build a Terraform plugin for your organization's internal use, for a proprietary API that you don't want to open source, or to prototype something before contributing it back to the main project.

In theory, because the plugin interface is HTTP, you could even develop a plugin using a completely different programming language! (Disclaimer, you would also have to re-implement the plugin API which is not a trivial amount of work.)

So to conclude, with the RPC interface *and* internal plugins, we get the best of all of these features: Binaries that *Just Work*, savings from shared code, and extensibility through plugins. We hope you enjoy using these features in Terraform.

Resource Lifecycle

Resources have a strict lifecycle, and can be thought of as basic state machines. Understanding this lifecycle can help better understand how Terraform generates an execution plan, how it safely executes that plan, and what the resource provider is doing throughout all of this.

Advanced Topic! This page covers technical details of Terraform. You don't need to understand these details to effectively use Terraform. The details are documented here for those who wish to learn about them without having to go spelunking through the source code.

Lifecycle

A resource roughly follows the steps below:

1. `ValidateResource` is called to do a high-level structural validation of a resource's configuration. The configuration at this point is raw and the interpolations have not been processed. The value of any key is not guaranteed and is just meant to be a quick structural check.
2. `Diff` is called with the current state and the configuration. The resource provider inspects this and returns a diff, outlining all the changes that need to occur to the resource. The diff includes details such as whether or not the resource is being destroyed, what attribute necessitates the destroy, old values and new values, whether a value is computed, etc. It is up to the resource provider to have this knowledge.
3. `Apply` is called with the current state and the diff. `Apply` does not have access to the configuration. This is a safety mechanism that limits the possibility that a provider changes a diff on the fly. `Apply` must apply a diff as prescribed and do nothing else to remain true to the Terraform execution plan. `Apply` returns the new state of the resource (or nil if the resource was destroyed).
4. If a resource was just created and did not exist before, and the apply succeeded without error, then the provisioners are executed in sequence. If any provisioner errors, the resource is marked as *tainted*, so that it will be destroyed on the next apply.

Partial State and Error Handling

If an error happens at any stage in the lifecycle of a resource, Terraform stores a partial state of the resource. This behavior is critical for Terraform to ensure that you don't end up with any *zombie* resources: resources that were created by Terraform but no longer managed by Terraform due to a loss of state.

Remote Service Discovery

Terraform implements much of its functionality in terms of remote services. While in many cases these are generic third-party services that are useful to many applications, some of these services are tailored specifically to Terraform's needs. We call these *Terraform-native services*, and Terraform interacts with them via the remote service discovery protocol described below.

User-facing Hostname

Terraform-native services are provided, from a user's perspective, at a user-facing "friendly hostname" which serves as the key for configuration and for any authentication credentials required.

The discovery protocol's purpose is to map from a user-provided hostname to the base URL of a particular service. Each host can provide different combinations of services -- or no services at all! -- and so the discovery protocol has a secondary purpose of allowing Terraform to identify *which* services are valid for a given hostname.

For example, module source strings can include a module registry hostname as their first segment, like `example.com/namespace/name/provider`, and Terraform uses service discovery to determine whether `example.com` *has* a module registry, and if so where its API is available.

A user-facing hostname is a fully-specified internationalized domain name (https://en.wikipedia.org/wiki/Internationalized_domain_name) expressed in its Unicode form (the corresponding "punycode" form is not allowed) which must be resolvable in DNS to an address that has an HTTPS server running on port 443.

User-facing hostnames are normalized for internal comparison using the standard Unicode Nameprep (<https://en.wikipedia.org/wiki/Nameprep>) algorithm, which includes converting all letters to lowercase, normalizing combining diacritics to precomposed form where possible, and various other normalization steps.

Discovery Process

Given a hostname, discovery begins by forming an initial discovery URL using that hostname with the `https:` scheme and the fixed path `/.well-known/terraform.json`.

For example, given the hostname `example.com` the initial discovery URL would be `https://example.com/.well-known/terraform.json`.

Terraform then sends a GET request to this discovery URL and expects a JSON response. If the response does not have status 200, does not have a media type of `application/json` or, if the body cannot be parsed as a JSON object, then discovery fails and Terraform considers the host to not support *any* Terraform-native services.

If the response is an HTTP redirect then Terraform repeats this step with the new location as its discovery URL. Terraform is guaranteed to follow at least one redirect, but nested redirects are not guaranteed nor recommended.

If the response is a valid JSON object then its keys are Terraform native service identifiers, consisting of a service type name and a version string separated by a period. For example, the service identifier for version 1 of the module registry protocol is `modules.v1`.

The value of each object element is the base URL for the service in question. This URL may be either absolute or relative, and if relative it is resolved against the final discovery URL (*after* following redirects).

The following is an example discovery document declaring support for version 1 of the module registry protocol:

```
{  
  "modules.v1": "https://modules.example.com/v1/"  
}
```

Supported Services

At present, only one service identifier is in use:

- `modules.v1`: module registry API version 1 ([/docs/registry/api.html](#))

Authentication

If credentials for the given hostname are available in the CLI config ([/docs/commands/cli-config.html](#)) then they will be included in the request for the discovery document.

The credentials may also be provided to endpoints declared in the discovery document, depending on the requirements of the service in question.

Non-standard Ports in User-facing Hostnames

It is strongly recommended to provide the discovery document for a hostname on the standard HTTPS port 443. However, in development environments this is not always possible or convenient, so Terraform allows a hostname to end with a port specification consisting of a colon followed by one or more decimal digits.

When a custom port number is present, the service on that port is expected to implement HTTPS and respond to the same fixed discovery path.

For day-to-day use it is strongly recommended *not* to rely on this mechanism and to instead provide the discovery document on the standard port, since this allows use of the most user-friendly hostname form.

Resource Addressing

A **Resource Address** is a string that references a specific resource in a larger infrastructure. An address is made up of two parts:

```
[module path][resource spec]
```

Module path:

A module path addresses a module within the tree of modules. It takes the form:

```
module.A.module.B.module.C...
```

Multiple modules in a path indicate nesting. If a module path is specified without a resource spec, the address applies to every resource within the module. If the module path is omitted, this addresses the root module.

Resource spec:

A resource spec addresses a specific resource in the config. It takes the form:

```
resource_type.resource_name[N]
```

- `resource_type` - Type of the resource being addressed.
- `resource_name` - User-defined name of the resource.
- `[N]` - where `N` is a 0-based index into a resource with multiple instances specified by the `count` meta-parameter.
Omitting an index when addressing a resource where `count > 1` means that the address references all instances.

Examples

Given a Terraform config that includes:

```
resource "aws_instance" "web" {  
    # ...  
    count = 4  
}
```

An address like this:

```
aws_instance.web[3]
```

Refers to only the last instance in the config, and an address like this:

```
aws_instance.web
```

Refers to all four "web" instances.

Modules

Modules in Terraform are self-contained packages of Terraform configurations that are managed as a group. Modules are used to create reusable components in Terraform as well as for basic code organization.

Modules are very easy to both use and create. Depending on what you're looking to do first, use the navigation on the left to dive into how modules work.

Definitions

Root module That is the current working directory when you run `terraform apply` ([/docs/commands/apply.html](#)) or `get` ([/docs/commands/get.html](#)), holding the Terraform configuration files ([/docs/configuration/index.html](#)). It is itself a valid module.

Modules

Modules in Terraform are self-contained packages of Terraform configurations that are managed as a group. Modules are used to create reusable components in Terraform as well as for basic code organization.

Modules are very easy to both use and create. Depending on what you're looking to do first, use the navigation on the left to dive into how modules work.

Definitions

Root module That is the current working directory when you run `terraform apply` ([/docs/commands/apply.html](#)) or `get` ([/docs/commands/get.html](#)), holding the Terraform configuration files ([/docs/configuration/index.html](#)). It is itself a valid module.

Creating Modules

Creating modules in Terraform is easy. You may want to do this to better organize your code, to make a reusable component, or just to learn more about Terraform. For any reason, if you already know the basics of Terraform, then creating a module is a piece of cake.

Modules in Terraform are folders with Terraform files. In fact, when you run `terraform apply`, the current working directory holding the Terraform files you're applying comprise what is called the *root module*. This itself is a valid module.

Therefore, you can enter the source of any module, satisfy any required variables, run `terraform apply`, and expect it to work.

Modules that are created for reuse should follow the standard structure. This structure enables tooling such as the Terraform Registry ([/docs/registry/index.html](#)) to inspect and generate documentation, read examples, and more.

An Example Module

Within a folder containing Terraform configurations, create a subfolder called `child`. In this subfolder, make one empty `main.tf` file. Then, back in the root folder containing the `child` folder, add this to one of your Terraform configuration files:

```
module "child" {
  source = "./child"
}
```

You've now created your first module! You can now add resources to the `child` module.

Note: Prior to running the above, you'll have to run the `get` command ([/docs/commands/get.html](#)) for Terraform to sync your modules. This should be instant since the module is a local path.

Inputs/Outputs

To make modules more useful than simple isolated containers of Terraform configurations, modules can be configured and also have outputs that can be consumed by your Terraform configuration.

Inputs of a module are variables ([/docs/configuration/variables.html](#)) and outputs are outputs ([/docs/configuration/outputs.html](#)). There is no special syntax to define these, they're defined just like any other variables or outputs. You can think about these variables and outputs as the API interface to your module.

Let's add a variable and an output to our `child` module.

```
variable "memory" {}

output "received" {
  value = "${var.memory}"
}
```

This will create a required variable, `memory`, and then an output, `received`, that will be the value of the `memory` variable.

You can then configure the module and use the output like so:

```

module "child" {
  source = "./child"

  memory = "1G"
}

output "child_memory" {
  value = "${module.child.received}"
}

```

If you now run `terraform apply`, you see how this works.

Paths and Embedded Files

It is sometimes useful to embed files within the module that aren't Terraform configuration files, such as a script to provision a resource or a file to upload.

In these cases, you can't use a relative path, since paths in Terraform are generally relative to the working directory from which Terraform was executed. Instead, you want to use a module-relative path. To do this, you should use the path interpolated variables ([/docs/configuration/interpolation.html](#)).

```

resource "aws_instance" "server" {
  # ...

  provisioner "remote-exec" {
    script = "${path.module}/script.sh"
  }
}

```

Here we use `${path.module}` to get a module-relative path.

Nested Modules

You can nest a module within another module. This module will be hidden from your root configuration, so you'll have to re-expose any variables and outputs you require.

The `get` command ([/docs/commands/get.html](#)) will automatically get all nested modules.

You don't have to worry about conflicting versions of modules, since Terraform builds isolated subtrees of all dependencies. For example, one module might use version 1.0 of module `foo` and another module might use version 2.0, and this will all work fine within Terraform since the modules are created separately.

Standard Module Structure

The standard module structure is a file and folder layout we recommend for reusable modules. Terraform tooling is built to understand the standard module structure and use that structure to generate documentation, index modules for the registry, and more.

The standard module expects the structure documented below. The list may appear long, but everything is optional except for the root module. All items are documented in detail. Most modules don't need to do any work to follow the standard structure.

- **Root module.** This is the **only required element** for the standard module structure. Terraform files must exist in the root directory of the module. This should be the primary entrypoint for the module and is expected to be opinionated. For the Consul module (<https://registry.terraform.io/modules/hashicorp/consul>) the root module sets up a complete Consul cluster. A lot of assumptions are made, however, and it is fully expected that advanced users will use specific nested modules to more carefully control what they want.
- **README.** The root module and any nested modules should have README files. This file should be named `README` or `README.md`. The latter will be treated as markdown. There should be a description of the module and what it should be used for. If you want to include an example for how this module can be used in combination with other resources, put it in an examples directory like this (<https://github.com/hashicorp/terraform-aws-consul/tree/master/examples>). Consider including a visual diagram depicting the infrastructure resources the module may create and their relationship. The README doesn't need to document inputs or outputs of the module because tooling will automatically generate this. If you are linking to a file or embedding an image contained in the repository itself, use a commit-specific absolute URL so the link won't point to the wrong version of a resource in the future.
- **LICENSE.** The license under which this module is available. If you are publishing a module publicly, many organizations will not adopt a module unless a clear license is present. We recommend always having a license file, even if the license is non-public.
- **main.tf, variables.tf, outputs.tf.** These are the recommended filenames for a minimal module, even if they're empty. `main.tf` should be the primary entrypoint. For a simple module, this may be where all the resources are created. For a complex module, resource creation may be split into multiple files but all nested module usage should be in the main file. `variables.tf` and `outputs.tf` should contain the declarations for variables and outputs, respectively.
- **Variables and outputs should have descriptions.** All variables and outputs should have one or two sentence descriptions that explain their purpose. This is used for documentation. See the documentation for variable configuration (/docs/configuration/variables.html) and output configuration (/docs/configuration/outputs.html) for more details.
- **Nested modules.** Nested modules should exist under the `modules/` subdirectory. Any nested module with a `README.md` is considered usable by an external user. If a `README` doesn't exist, it is considered for internal use only. These are purely advisory; Terraform will not actively deny usage of internal modules. Nested modules should be used to split complex behavior into multiple small modules that advanced users can carefully pick and choose. For example, the Consul module (<https://registry.terraform.io/modules/hashicorp/consul>) has a nested module for creating the Cluster that is separate from the module to setup necessary IAM policies. This allows a user to bring in their own IAM policy choices.
- **Examples.** Examples of using the module should exist under the `examples/` subdirectory at the root of the repository. Each example may have a `README` to explain the goal and usage of the example. Examples for submodules should also be placed in the root `examples/` directory.

A minimal recommended module following the standard structure is shown below. While the root module is the only required element, we recommend the structure below as the minimum:

```
$ tree minimal-module/
.
├── README.md
├── main.tf
├── variables.tf
└── outputs.tf
```

A complete example of a module following the standard structure is shown below. This example includes all optional elements and is therefore the most complex a module can become:

```
$ tree complete-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
└── modules/
    ├── nestedA/
    │   ├── README.md
    │   ├── variables.tf
    │   ├── main.tf
    │   └── outputs.tf
    ├── nestedB/
    ├── ...
    └── examples/
        ├── exampleA/
        │   ├── main.tf
        ├── exampleB/
        └── .../
```

Publishing Modules

If you've built a module that you intend to be reused, we recommend publishing the module ([/docs/registry/modules/publish.html](#)) on the Terraform Registry (<https://registry.terraform.io>). This will version your module, generate documentation, and more.

Published modules can be easily consumed by Terraform, and (from Terraform 0.11) you can also constrain module versions ([/docs/modules/usage.html#module-versions](#)) for safe and predictable updates. The following example shows how easy it is to consume a module from the registry:

```
module "consul" {
  source = "hashicorp/consul/aws"
}
```

You can also gain all the benefits of the registry for private modules by signing up for a private registry ([/docs/registry/private.html](#)).

Module Sources

As introduced in the [Usage](#) section (/docs/modules/usage.html), the `source` argument in a `module` block tells Terraform where to find the source code for the desired child module.

Terraform uses this during the module installation step of `terraform init` to download the source code to a directory on local disk so that it can be used by other Terraform commands.

The module installer supports installation from a number of different source types, as listed below.

- Local paths
- Terraform Registry
- GitHub
- Bitbucket
- Generic Git, Mercurial repositories
- HTTP URLs
- S3 buckets

Each of these is described in the following sections. Module source addresses use a *URL-like* syntax, but with extensions to support unambiguous selection of sources and additional features.

We recommend using local file paths for closely-related modules used primarily for the purpose of factoring out repeated code elements, and using a native Terraform module registry for modules intended to be shared by multiple calling configurations. We support other sources so that you can potentially distribute Terraform modules internally with existing infrastructure.

Many of the source types will make use of "ambient" credentials available when Terraform is run, such as from environment variables or credentials files in your home directory. This is covered in more detail in each of the following sections.

We recommend placing each module that is intended to be re-usable in the root of its own repository or archive file, but it is also possible to reference modules from subdirectories.

Local Paths

Local path references allow for factoring out portions of a configuration within a single source repository.

```
module "consul" {
  source = "./consul"
}
```

A local path must begin with either `./` or `../` to indicate that a local path is intended, to distinguish from a module registry address.

Local paths are special in that they are not "installed" in the same sense that other sources are: the files are already present on local disk (possibly as a result of installing a parent module) and so can just be used directly. Their source code is automatically updated if the parent module is upgraded.

Terraform Registry

A module registry is the native way of distributing Terraform modules for use across multiple configurations, using a Terraform-specific protocol that has full support for module versioning.

Terraform Registry (<https://registry.terraform.io/>) is an index of modules shared publicly using this protocol. This public registry is the easiest way to get started with Terraform and find modules created by others in the community.

You can also use a private registry (/docs/registry/private.html), either via the built-in feature from Terraform Enterprise, or by running a custom service that implements the module registry protocol (/docs/registry/api.html).

Modules on the public Terraform Registry can be referenced using a registry source address of the form <NAMESPACE>/<NAME>/<PROVIDER>, with each module's information page on the registry site including the exact address to use.

```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

The above example will use the Consul module for AWS (<https://registry.terraform.io/modules/hashicorp/consul/aws>) from the public registry.

For modules hosted in other registries, prefix the source address with an additional <HOSTNAME>/ portion, giving the hostname of the private registry:

```
module "consul" {
  source = "app.terraform.io/example-corp/k8s-cluster/azurerm"
  version = "1.1.0"
}
```

If you are using the managed version of Terraform Enterprise, its private registry hostname is `app.terraform.io`. If you are using Terraform Enterprise on-premises, its private registry hostname is the same hostname you use to access your Terraform Enterprise instance.

Registry modules support versioning. You can provide a specific version as shown in the above examples, or use flexible version constraints (/docs/modules/usage.html#module-versions).

You can learn more about the registry at the Terraform Registry documentation (/docs/registry/modules/use.html#using-modules).

To access modules from a private registry, you may need to configure an access token in the CLI config (/docs/commands/cli-config.html#credentials). Use the same hostname as used in the module source string. For a private registry within Terraform Enterprise, use the same authentication token as you would use with the Enterprise API or command-line clients.

GitHub

Terraform will recognize unprefixed `github.com` URLs and interpret them automatically as Git repository sources.

```
module "consul" {
  source = "github.com/hashicorp/example"
}
```

The above address scheme will clone over HTTPS. To clone over SSH, use the following form:

```
module "consul" {
  source = "git@github.com:hashicorp/example.git"
}
```

These GitHub schemes are treated as convenient aliases for the general Git repository address scheme, and so they obtain credentials in the same way and support the `ref` argument for selecting a specific revision. You will need to configure credentials in particular to access private repositories.

Bitbucket

Terraform will recognize unprefixed `bitbucket.org` URLs and interpret them automatically as BitBucket repositories:

```
module "consul" {
  source = "bitbucket.org/hashicorp/terraform-consul-aws"
}
```

This shorthand works only for public repositories, because Terraform must access the BitBucket API to learn if the given repository uses Git or Mercurial.

Terraform treats the result either as a Git source or a Mercurial source depending on the repository type. See the sections on each version control type for information on how to configure credentials for private repositories and how to specify a specific revision to install.

Generic Git Repository

Arbitrary Git repositories can be used by prefixing the address with the special `git::` prefix. After this prefix, any valid Git URL (https://git-scm.com/docs/git-clone#_git_urls_a_id_urls_a) can be specified to select one of the protocols supported by Git.

For example, to use HTTPS or SSH:

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}

module "storage" {
  source = "git::ssh://username@example.com/storage.git"
}
```

Terraform installs modules from Git repositories by running `git clone`, and so it will respect any local Git configuration set on your system, including credentials. To access a non-public Git repository, configure Git with suitable credentials for that repository.

If you use the SSH protocol then any configured SSH keys will be used automatically. This is the most common way to access non-public Git repositories from automated systems because it is easy to configure and allows access to private repositories without interactive prompts.

If using the HTTP/HTTPS protocol, or any other protocol that uses username/password credentials, configure Git Credentials Storage (<https://git-scm.com/book/en/v2/Git-Tools-Credential-Storage>) to select a suitable source of credentials for your environment.

If your Terraform configuration will be used within Terraform Enterprise (<https://www.hashicorp.com/products/terraform>), only SSH key authentication is supported, and keys can be configured on a per-workspace basis ([/docs/enterprise/workspaces/ssh-keys.html](#)).

Selecting a Revision

By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repository. You can override this using the `ref` argument:

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

The value of the `ref` argument can be any reference that would be accepted by the `git checkout` command, including branch and tag names.

Generic Mercurial Repository

You can use arbitrary Mercurial repositories by prefixing the address with the special `hg::` prefix. After this prefix, any valid Mercurial URL (<https://www.mercurial-scm.org/repo/hg/help/urls>) can be specified to select one of the protocols supported by Mercurial.

```
module "vpc" {
  source = "hg::http://example.com/vpc.hg"
}
```

Terraform installs modules from Mercurial repositories by running `hg clone`, and so it will respect any local Mercurial configuration set on your system, including credentials. To access a non-public repository, configure Mercurial with suitable credentials for that repository.

If you use the SSH protocol then any configured SSH keys will be used automatically. This is the most common way to access non-public Mercurial repositories from automated systems because it is easy to configure and allows access to private repositories without interactive prompts.

If your Terraform configuration will be used within Terraform Enterprise (<https://www.hashicorp.com/products/terraform>), only SSH key authentication is supported, and keys can be configured on a per-workspace basis ([/docs/enterprise/workspaces/ssh-keys.html](#)).

Selecting a Revision

You can select a non-default branch or tag using the optional `ref` argument:

```
module "vpc" {
  source = "hg::http://example.com/vpc.hg?ref=v1.2.0"
}
```

HTTP URLs

When you use an HTTP or HTTPS URL, Terraform will make a `GET` request to the given URL, which can return *another* source address. This indirection allows using HTTP URLs as a sort of "vanity redirect" over a more complicated module source address.

Terraform will append an additional query string argument `terraform-get=1` to the given URL before sending the `GET` request, allowing the server to optionally return a different result when Terraform is requesting it.

If the response is successful (200-range status code), Terraform looks in the following locations in order for the next address to access:

- The value of a response header field named `X-Terraform-Get`.
- If the response is an HTML page, a `meta` element with the name `terraform-get`:

```
<meta name="terraform-get"
      content="github.com/hashicorp/example" />
```

In either case, the result is interpreted as another module source address using one of the forms documented elsewhere on this page.

If an HTTP/HTTPS URL requires authentication credentials, use a `.netrc` file in your home directory to configure these. For information on this format, see the documentation for using it in `curl` (<https://ec.haxx.se/usingcurl-netrc.html>).

Fetching archives over HTTP

As a special case, if Terraform detects that the URL has a common file extension associated with an archive file format then it will bypass the special `terraform-get=1` redirection described above and instead just use the contents of the referenced archive as the module source code:

```
module "vpc" {
  source = "https://example.com/vpc-module.zip"
}
```

The extensions that Terraform recognizes for this special behavior are:

- `zip`
- `tar.bz2` and `tbz2`
- `tar.gz` and `tgz`
- `tar.xz` and `txz`

If your URL *doesn't* have one of these extensions but refers to an archive anyway, use the `archive` argument to force this interpretation:

```
module "vpc" {
  source = "https://example.com/vpc-module?archive=zip"
}
```

S3 Bucket

You can use archives stored in S3 as module sources using the special `s3::` prefix, followed by an S3 bucket object URL (<http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingBucket.html#access-bucket-intro>).

```
module "consul" {
  source = "s3::https://s3-eu-west-1.amazonaws.com/examplecorp-terraform-modules/vpc.zip"
}
```

The `s3::` prefix causes Terraform to use AWS-style authentication when accessing the given URL. As a result, this scheme may also work for other services that mimic the S3 API, as long as they handle authentication in the same way as AWS.

The resulting object must be an archive with one of the same file extensions as for archives over standard HTTP. Terraform will extract the archive to obtain the module source tree.

The module installer looks for AWS credentials in the following locations, preferring those earlier in the list when multiple are available:

- The `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables.
- The default profile in the `.aws/credentials` file in your home directory.
- If running on an EC2 instance, temporary credentials associated with the instance's IAM Instance Profile.

Modules in Package Sub-directories

When the source of a module is a version control repository or archive file (generically, a "package"), the module itself may be in a sub-directory relative to the root of the package.

A special double-slash syntax is interpreted by Terraform to indicate that the remaining path after that point is a sub-directory within the package. For example:

- `hashicorp/consul/aws//modules/consul-cluster`
- `git::https://example.com/network.git//modules/vpc`
- `https://example.com/network-module.zip//modules/vpc`
- `s3::https://s3-eu-west-1.amazonaws.com/examplecorp-terraform-modules/network.zip//modules/vpc`

If the source address has arguments, such as the `ref` argument supported for the version control sources, the sub-directory portion must be *before* those arguments:

- `git::https://example.com/network.git//modules/vpc?ref=v1.2.0`

Terraform will still extract the entire package to local disk, but will read the module from the subdirectory. As a result, it is safe for a module in a sub-directory of a package to use a local path to another module as long as it is in the *same* package.

Module Usage

Using child modules in Terraform is very similar to defining resources:

```
module "consul" {
  source  = "hashicorp/consul/aws"
  servers = 3
}
```

You can view the full documentation for configuring modules in the [Module Configuration](#) ([/docs/configuration/modules.html](#)) section.

In modules we only specify a name, rather than a name and a type as for resources. This name is used elsewhere in the configuration to reference the module and its outputs.

The source tells Terraform what to create. In this example, we instantiate the Consul module for AWS (<https://registry.terraform.io/modules/hashicorp/consul/aws>) from the Terraform Registry (<https://registry.terraform.io>). Other source types are supported, as described in the following section.

Just like a resource, a module's configuration can be deleted to destroy the resources belonging to the module.

Source

The only required configuration key for a module is the `source` parameter. The value of this tells Terraform where to download the module's source code. Terraform comes with support for a variety of module sources.

We recommend using modules from the public Terraform Registry ([/docs/registry/index.html](#)) or from Terraform Enterprise's private module registry ([/docs/enterprise/registry/index.html](#)). These sources support version constraints for a more reliable experience, and provide a searchable marketplace for finding the modules you need.

Registry modules are specified using a simple slash-separated path like the `hashicorp/consul/aws` path used in the above example. The full source string for each registry module can be found from the registry website.

Terraform also supports modules in local directories, identified by a relative path starting with either `./` or `../`. Such local modules are useful to organize code in more complex repositories, and are described in more detail in [Creating Modules](#) ([/docs/modules/create.html](#)).

Finally, Terraform can download modules directly from various storage providers and version control systems. These sources do not support versioning and other registry benefits, but can be convenient for getting started when already available within an organization. The full list of available sources are documented in the module sources documentation ([/docs/modules/sources.html](#)).

When a configuration uses modules, they must first be installed by running `terraform init` ([/docs/commands/init.html](#)):

```
$ terraform init
```

This command will download any modules that haven't been updated already, as well as performing other Terraform working directory initialization such as installing providers.

By default the command will not check for available updates to already-installed modules, but you can use the `-upgrade` option to check for available upgrades. When version constraints are specified (as described in the following section) a

newer version will be used only if it is within the given constraint.

Module Versions

We recommend explicitly constraining the acceptable version numbers for each external module to avoid unexpected or unwanted changes.

Use the `version` attribute in the `module` block to specify versions:

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"

  servers = 3
}
```

The `version` attribute value may either be a single explicit version or a version constraint expression. Constraint expressions use the following syntax to specify a *range* of versions that are acceptable:

- `>= 1.2.0`: version 1.2.0 or newer
- `<= 1.2.0`: version 1.2.0 or older
- `~> 1.2.0`: any non-beta version $\geq 1.2.0$ and $< 1.3.0$, e.g. 1.2.X
- `~> 1.2`: any non-beta version $\geq 1.2.0$ and $< 2.0.0$, e.g. 1.X.Y
- `>= 1.0.0, <= 2.0.0`: any version between 1.0.0 and 2.0.0 inclusive

When depending on third-party modules, references to specific versions are recommended since this ensures that updates only happen when convenient to you.

For modules maintained within your organization, a version range strategy may be appropriate if a semantic versioning methodology is used consistently or if there is a well-defined release process that avoids unwanted updates.

Version constraints are supported only for modules installed from a module registry, such as the Terraform Registry (<https://registry.terraform.io/>) or Terraform Enterprise's private module registry (</docs/enterprise/registry/index.html>). Other module sources can provide their own versioning mechanisms within the source string itself, or might not support versions at all. In particular, modules sourced from local file paths do not support `version`; since they're loaded from the same source repository, they always share the same version as their caller.

Configuration

The arguments used in a `module` block, such as the `servers` parameter above, correspond to variables (</docs/configuration/variables.html>) within the module itself. You can therefore discover all the available variables for a module by inspecting the source of it.

The special arguments `source`, `version` and `providers` are exceptions. These are used for special purposes by Terraform and should therefore not be used as variable names within a module.

Outputs

Modules encapsulate their resources. A resource in one module cannot directly depend on resources or attributes in other modules, unless those are exported through outputs ([/docs/configuration/outputs.html](#)). These outputs can be referenced in other places in your configuration, for example:

```
resource "aws_instance" "client" {
  ami           = "ami-408c7f28"
  instance_type = "t1.micro"
  availability_zone = "${module.consul.server_availability_zone}"
}
```

This is deliberately very similar to accessing resource attributes. Instead of referencing a resource attribute, however, the expression in this case references an output of the module.

Just like with resources, interpolation expressions can create implicit dependencies on resources and other modules. Since modules encapsulate other resources, however, the dependency is not on the module as a whole but rather on the `server_availability_zone` output specifically, which allows Terraform to work on resources in different modules concurrently rather than waiting for the entire module to be complete before proceeding.

Providers within Modules

In a configuration with multiple modules, there are some special considerations for how resources are associated with provider configurations.

While in principle provider blocks can appear in any module, it is recommended that they be placed only in the *root* module of a configuration, since this approach allows users to configure providers just once and re-use them across all descendent modules.

Each resource in the configuration must be associated with one provider configuration, which may either be within the same module as the resource or be passed from the parent module. Providers can be passed down to descendent modules in two ways: either *implicitly* through inheritance, or *explicitly* via the `providers` argument within a `module` block. These two options are discussed in more detail in the following sections.

In all cases it is recommended to keep explicit provider configurations only in the root module and pass them (whether implicitly or explicitly) down to descendent modules. This avoids the provider configurations from being "lost" when descendent modules are removed from the configuration. It also allows the user of a configuration to determine which providers require credentials by inspecting only the root module.

Provider configurations are used for all operations on associated resources, including destroying remote objects and refreshing state. Terraform retains, as part of its state, a reference to the provider configuration that was most recently used to apply changes to each resource. When a resource block is removed from the configuration, this record in the state is used to locate the appropriate configuration because the resource's `provider` argument (if any) is no longer present in the configuration.

As a consequence, it is required that all resources created for a particular provider configuration must be destroyed before that provider configuration is removed, unless the related resources are re-configured to use a different provider configuration first.

Implicit Provider Inheritance

For convenience in simple configurations, a child module automatically inherits default (un-aliased) provider configurations from its parent. This means that explicit provider blocks appear only in the root module, and downstream modules can simply declare resources for that provider and have them automatically associated with the root provider configurations.

For example, the root module might contain only a provider block and a module block to instantiate a child module:

```
provider "aws" {
  region = "us-west-1"
}

module "child" {
  source = "./child"
}
```

The child module can then use any resource from this provider with no further provider configuration required:

```
resource "aws_s3_bucket" "example" {
  bucket = "provider-inherit-example"
}
```

This approach is recommended in the common case where only a single configuration is needed for each provider across the entire configuration.

In more complex situations there may be multiple provider instances ([/docs/configuration/providers.html#multiple-provider-instances](#)), or a child module may need to use different provider settings than its parent. For such situations, it's necessary to pass providers explicitly as we will see in the next section.

Passing Providers Explicitly

When child modules each need a different configuration of a particular provider, or where the child module requires a different provider configuration than its parent, the `providers` argument within a `module` block can be used to define explicitly which provider configs are made available to the child module. For example:

```

# The default "aws" configuration is used for AWS resources in the root
# module where no explicit provider instance is selected.
provider "aws" {
  region = "us-west-1"
}

# A non-default, or "aliased" configuration is also defined for a different
# region.
provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

# An example child module is instantiated with the _aliased_ configuration,
# so any AWS resources it defines will use the us-west-2 region.
module "example" {
  source    = "./example"
  providers = {
    aws = "aws.usw2"
  }
}

```

The `providers` argument within a `module` block is similar to the `provider` argument within a resource as described for multiple provider instances ([/docs/configuration/providers.html#multiple-provider-instances](#)), but is a map rather than a single string because a module may contain resources from many different providers.

Once the `providers` argument is used in a `module` block, it overrides all of the default inheritance behavior, so it is necessary to enumerate mappings for *all* of the required providers. This is to avoid confusion and surprises that may result when mixing both implicit and explicit provider passing.

Additional provider configurations (those with the `alias` argument set) are *never* inherited automatically by child modules, and so must always be passed explicitly using the `providers` map. For example, a module that configures connectivity between networks in two AWS regions is likely to need both a source and a destination region. In that case, the root module may look something like this:

```

provider "aws" {
  alias  = "usw1"
  region = "us-west-1"
}

provider "aws" {
  alias  = "usw2"
  region = "us-west-2"
}

module "tunnel" {
  source    = "./tunnel"
  providers = {
    aws.src = "aws.usw1"
    aws.dst = "aws.usw2"
  }
}

```

In the `providers` map, the keys are provider names as expected by the child module, while the values are the names of corresponding configurations in the *current* module. The subdirectory `./tunnel` must then contain *proxy configuration blocks* like the following, to declare that it requires configurations to be passed with these from the `providers` block in the parent's `module` block:

```
provider "aws" {
  alias = "src"
}

provider "aws" {
  alias = "dst"
}
```

Each resource should then have its own provider attribute set to either "aws.src" or "aws.dst" to choose which of the two provider instances to use.

At this time it is required to write an explicit proxy configuration block even for default (un-aliased) provider configurations when they will be passed via an explicit providers block:

```
provider "aws" {
```

If such a block is not present, the child module will behave as if it has no configurations of this type at all, which may cause input prompts to supply any required provider configuration arguments. This limitation will be addressed in a future version of Terraform.

Multiple Instances of a Module

A particular module source can be instantiated multiple times:

```
# my_buckets.tf

module "assets_bucket" {
  source = "./publish_bucket"
  name   = "assets"
}

module "media_bucket" {
  source = "./publish_bucket"
  name   = "media"
}
```

```
# publish_bucket/bucket-and-cloudfront.tf

variable "name" {} # this is the input parameter of the module

resource "aws_s3_bucket" "example" {
  # ...
}

resource "aws_iam_user" "deploy_user" {
  # ...
}
```

This example defines a local child module in the ./publish_bucket subdirectory. That module has configuration to create an S3 bucket. The module wraps the bucket and all the other implementation details required to configure a bucket.

We can then instantiate the module multiple times in our configuration by giving each instance a unique name -- here `module "assets_bucket"` and `module "media_bucket"` -- whilst specifying the same source value.

Resources from child modules are prefixed with `module.<module-instance-name>` when displayed in plan output and elsewhere in the UI. For example, the `./publish_bucket` module contains `aws_s3_bucket.example`, and so the two instances of this module produce S3 bucket resources with *resource addresses* ([/docs/internals/resource-addressing.html](#)) `module.assets_bucket.aws_s3_bucket.example` and `module.media_bucket.aws_s3_bucket.example` respectively. These full addresses are used within the UI and on the command line, but are not valid within interpolation expressions due to the encapsulation behavior described above.

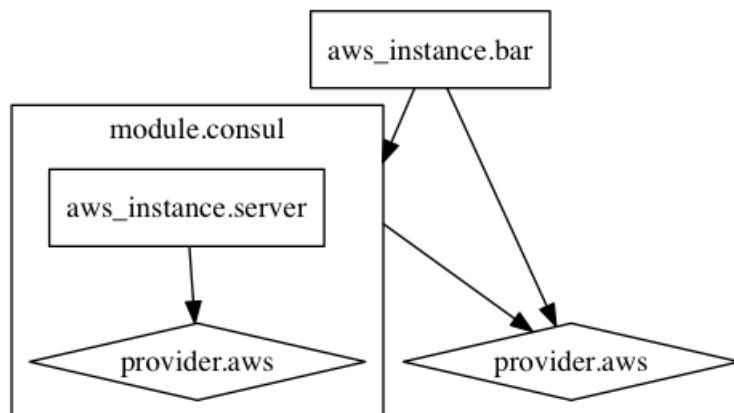
When refactoring an existing configuration to introduce modules, moving resource blocks between modules causes Terraform to see the new location as an entirely separate resource to the old. Always check the execution plan after performing such actions to ensure that no resources are surprisingly deleted.

Each instance of a module may optionally have different providers passed to it using the `providers` argument described above. This can be useful in situations where, for example, a duplicated set of resources must be created across several regions or datacenters.

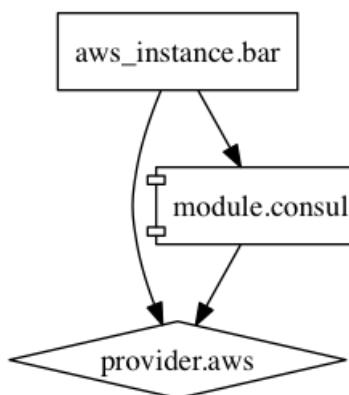
Summarizing Modules in the UI

By default, commands such as the `plan` command ([/docs/commands/plan.html](#)) and `graph` command ([/docs/commands/graph.html](#)) will show each resource in a nested module to represent the full scope of the configuration. For more complex configurations, the `-module-depth` option may be useful to summarize some or all of the modules as single objects.

For example, with a configuration similar to what we've built above, the default graph output looks like the following:



If we instead set `-module-depth=0`, the graph will look like this:



Other commands work similarly with modules. Note that `-module-depth` only affects how modules are presented in the UI; it does not affect how modules and their contained resources are processed by Terraform operations.

Tainting resources within a module

The `taint` command ([/docs/commands/taint.html](#)) can be used to *taint* specific resources within a module:

```
$ terraform taint -module=salt_master aws_instance.salt_master
```

It is not possible to taint an entire module. Instead, each resource within the module must be tainted separately.

Plugins

Terraform is built on a plugin-based architecture. All providers and provisioners that are used in Terraform configurations are plugins, even the core types such as AWS and Heroku. Users of Terraform are able to write new plugins in order to support new functionality in Terraform.

This section of the documentation gives a high-level overview of how to write plugins for Terraform. It does not hold your hand through the process, however, and expects a relatively high level of understanding of Go, provider semantics, Unix, etc.

Advanced topic! Plugin development is a highly advanced topic in Terraform, and is not required knowledge for day-to-day usage. If you don't plan on writing any plugins, we recommend not reading this section of the documentation.

Plugin Basics

Advanced topic! Plugin development is a highly advanced topic in Terraform, and is not required knowledge for day-to-day usage. If you don't plan on writing any plugins, this section of the documentation is not necessary to read. For general use of Terraform, please see our Intro to Terraform (/intro/index.html) and Getting Started (/intro/getting-started/install.html) guides.

This page documents the basics of how the plugin system in Terraform works, and how to setup a basic development environment for plugin development if you're writing a Terraform plugin.

How it Works

Terraform providers and provisioners are provided via plugins. Each plugin exposes an implementation for a specific service, such as AWS, or provisioner, such as bash. Plugins are executed as a separate process and communicate with the main Terraform binary over an RPC interface.

More details are available in *Plugin Internals* (/docs/internals/internal-plugins.html).

The code within the binaries must adhere to certain interfaces. The network communication and RPC is handled automatically by higher-level Terraform libraries. The exact interface to implement is documented in its respective documentation section.

Installing Plugins

The provider plugins distributed by HashiCorp (/docs/providers/index.html) are automatically installed by `terraform init`. Third-party plugins (both providers and provisioners) can be manually installed into the user plugins directory, located at `%APPDATA%\terraform.d\plugins` on Windows and `~/.terraform.d/plugins` on other systems.

For more information, see:

- Configuring Providers (/docs/configuration/providers.html)
- Configuring Providers: Third-party Plugins (/docs/configuration/providers.html#third-party-plugins)

For developer-centric documentation, see:

- How Terraform Works: Plugin Discovery (/docs/extend/how-terraform-works.html#discovery)

Developing a Plugin

Developing a plugin is simple. The only knowledge necessary to write a plugin is basic command-line skills and basic knowledge of the Go programming language (<http://golang.org>).

Note: A common pitfall is not properly setting up a \$GOPATH. This can lead to strange errors. You can read more about this here (<https://golang.org/doc/code.html>) to familiarize yourself.

Create a new Go project somewhere in your \$GOPATH. If you're a GitHub user, we recommend creating the project in the directory \$GOPATH/src/github.com/USERNAME/terraform-NAME, where USERNAME is your GitHub username and NAME is the name of the plugin you're developing. This structure is what Go expects and simplifies things down the road.

The NAME should either begin with provider- or provisioner-, depending on what kind of plugin it will be. The repository name will, by default, be the name of the binary produced by go install for your plugin package.

With the package directory made, create a main.go file. This project will be a binary so the package is "main":

```
package main

import (
    "github.com/hashicorp/terraform/plugin"
)

func main() {
    plugin.Serve(new(MyPlugin))
}
```

The name MyPlugin is a placeholder for the struct type that represents your plugin's implementation. This must implement either `terraform.ResourceProvider` or `terraform.ResourceProvisioner`, depending on the plugin type.

To test your plugin, the easiest method is to copy your `terraform` binary to \$GOPATH/bin and ensure that this copy is the one being used for testing. `terraform init` will search for plugins within the same directory as the `terraform` binary, and \$GOPATH/bin is the directory into which `go install` will place the plugin executable.

Provider Plugins

Advanced topic! Plugin development is a highly advanced topic in Terraform, and is not required knowledge for day-to-day usage. If you don't plan on writing any plugins, this section of the documentation is not necessary to read. For general use of Terraform, please see our Intro to Terraform (/intro/index.html) and Getting Started (/intro/getting-started/install.html) guides.

A provider in Terraform is responsible for the lifecycle of a resource: create, read, update, delete. An example of a provider is AWS, which can manage resources of type `aws_instance`, `aws_eip`, `aws_elb`, etc.

The primary reasons to care about provider plugins are:

- You want to add a new resource type to an existing provider.
- You want to write a completely new provider for managing resource types in a system not yet supported.
- You want to write a completely new provider for custom, internal systems such as a private inventory management system.

If you're interested in provider development, then read on. The remainder of this page will assume you're familiar with plugin basics (/docs/plugins/basics.html) and that you already have a basic development environment setup.

Provider Plugin Codebases

Provider plugins live outside of the Terraform core codebase in their own source code repositories. The official set of provider plugins released by HashiCorp (developed by both HashiCorp staff and community contributors) all live in repositories in the `terraform-providers` organization (<https://github.com/terraform-providers>) on GitHub, but third-party plugins can be maintained in any source code repository.

When developing a provider plugin, it is recommended to use a common GOPATH that includes both the core Terraform repository and the repositories of any providers being used. This makes it easier to use a locally-built `terraform` executable and a set of locally-built provider plugins together without further configuration.

For example, to download both Terraform and the template provider into GOPATH:

```
$ go get github.com/hashicorp/terraform
$ go get github.com/terraform-providers/terraform-provider-template
```

These two packages are both "main" packages that can be built into separate executables with `go install`:

```
$ go install github.com/hashicorp/terraform
$ go install github.com/terraform-providers/terraform-provider-template
```

After running the above commands, both Terraform core and the template provider will both be installed in the current GOPATH and \$GOPATH/bin will contain both `terraform` and `terraform-provider-template` executables. This `terraform` executable will find and use the template provider plugin alongside it in the bin directory in preference to downloading and installing an official release.

When constructing a new provider from scratch, it's recommended to follow a similar repository structure as for the existing providers, with the main package in the repository root and a library package in a subdirectory named after the provider. For more information, see the custom providers guide (</guides/writing-custom-terraform-providers.html>).

When making changes only to files within the provider repository, it is *not* necessary to re-build the main Terraform executable. Note that some packages from the Terraform repository are used as library dependencies by providers, such as `github.com/hashicorp/terraform/helper/schema`; it is recommended to use `govendor` to create a local vendor copy of the relevant packages in the provider repository, as can be seen in the repositories within the `terraform-providers` GitHub organization.

Low-Level Interface

The interface you must implement for providers is `ResourceProvider` (https://github.com/hashicorp/terraform/blob/master/terraform/resource_provider.go).

This interface is extremely low level, however, and we don't recommend you implement it directly. Implementing the interface directly is error prone, complicated, and difficult.

Instead, we've developed some higher level libraries to help you out with developing providers. These are the same libraries we use in our own core providers.

helper/schema

The `helper/schema` library is a framework we've built to make creating providers extremely easy. This is the same library we use to build most of the core providers.

To give you an idea of how productive you can become with this framework: we implemented the Google Cloud provider in about 6 hours of coding work. This isn't a simple provider, and we did have knowledge of the framework beforehand, but it goes to show how expressive the framework can be.

The GoDoc for `helper/schema` can be found here (<https://godoc.org/github.com/hashicorp/terraform/helper/schema>). This is API-level documentation but will be extremely important for you going forward.

Provider

The first thing to do in your plugin is to create the `schema.Provider` (<https://godoc.org/github.com/hashicorp/terraform/helper/schema#Provider>) structure. This structure implements the `ResourceProvider` interface. We recommend creating this structure in a function to make testing easier later. Example:

```
func Provider() *schema.Provider {
    return &schema.Provider{
        ...
    }
}
```

Within the `schema.Provider`, you should initialize all the fields. They are documented within the godoc, but a brief overview is here as well:

- Schema - This is the configuration schema for the provider itself. You should define any API keys, etc. here. Schemas are covered below.
- ResourcesMap - The map of resources that this provider supports. All keys are resource names and the values are the schema.Resource (<https://godoc.org/github.com/hashicorp/terraform/helper/schema#Resource>) structures implementing this resource.
- ConfigureFunc - This function callback is used to configure the provider. This function should do things such as initialize any API clients, validate API keys, etc. The interface{} return value of this function is the meta parameter that will be passed into all resource CRUD (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) functions. In general, the returned value is a configuration structure or a client.

As part of the unit tests, you should call `InternalValidate`. This is used to verify the structure of the provider and all of the resources, and reports an error if it is invalid. An example test is shown below:

```
func TestProvider(t *testing.T) {
    if err := Provider().(*schema.Provider).InternalValidate(); err != nil {
        t.Fatalf("err: %s", err)
    }
}
```

Having this unit test will catch a lot of beginner mistakes as you build your provider.

Resources

Next, you'll want to create the resources that the provider can manage. These resources are put into the `ResourcesMap` field of the provider structure. Again, we recommend creating functions to instantiate these. An example is shown below.

```
func resourceComputeAddress() *schema.Resource {
    return &schema.Resource {
        ...
    }
}
```

Resources are described using the `schema.Resource`

(<https://godoc.org/github.com/hashicorp/terraform/helper/schema#Resource>) structure. This structure has the following fields:

- Schema - The configuration schema for this resource. Schemas are covered in more detail below.
- Create, Read, Update, and Delete - These are the callback functions that implement CRUD operations for the resource. The only optional field is `Update`. If your resource doesn't support update, then you may keep that field `nil`.
- Importer - If this is non-nil, then this resource is importable (/docs/import/importability.html). It is recommended to implement this.

The CRUD operations in more detail, along with their contracts:

- Create - This is called to create a new instance of the resource. Terraform guarantees that an existing ID is not set on the resource data. That is, you're working with a new resource. Therefore, you are responsible for calling `SetId` on your `schema.ResourceData` using a value suitable for your resource. This ensures whatever resource state you set on `schema.ResourceData` will be persisted in local state. If you neglect to `SetId`, no resource state will be persisted.

- Read - This is called to resync the local state with the remote state. Terraform guarantees that an existing ID will be set. This ID should be used to look up the resource. Any remote data should be updated into the local data. **No changes to the remote resource are to be made.** If the resource is no longer present, calling SetId with an empty string will signal its removal.
- Update - This is called to update properties of an existing resource. Terraform guarantees that an existing ID will be set. Additionally, the only changed attributes are guaranteed to be those that support update, as specified by the schema. Be careful to read about partial states below.
- Delete - This is called to delete the resource. Terraform guarantees an existing ID will be set.
- Exists - This is called to verify a resource still exists. It is called prior to Read, and lowers the burden of Read to be able to assume the resource exists. false should be returned if the resources is no longer present, which has the same effect as calling SetId("") from Read (i.e. removal of the resource data from state).

Schemas

Both providers and resources require a schema to be specified. The schema is used to define the structure of the configuration, the types, etc. It is very important to get correct.

In both provider and resource, the schema is a `map[string]*schema.Schema`. The key of this map is the configuration key, and the value is a schema for the value of that key.

Schemas are incredibly powerful, so this documentation page won't attempt to cover the full power of them. Instead, the API docs should be referenced which cover all available settings.

We recommend viewing schemas of existing or similar providers to learn best practices. A good starting place is the core Terraform providers (<https://github.com/terraform-providers>).

Resource Data

The parameter to provider configuration as well as all the CRUD operations on a resource is a `schema.ResourceData` (<https://godoc.org/github.com/hashicorp/terraform/helper/schema#ResourceData>). This structure is used to query configurations as well as to set information about the resource such as its ID, connection information, and computed attributes.

The API documentation covers `ResourceData` well, as well as the core providers in Terraform.

Partial state deserves a special mention. Occasionally in Terraform, create or update operations are not atomic; they can fail halfway through. As an example, when creating an AWS security group, creating the group may succeed, but creating all the initial rules may fail. In this case, it is incredibly important that Terraform record the correct *partial state* so that a subsequent `terraform apply` fixes this resource.

Most of the time, partial state is not required. When it is, it must be specifically enabled. An example is shown below:

```

func resourceUpdate(d *schema.ResourceData, meta interface{}) error {
    // Enable partial state mode
    d.Partial(true)

    if d.HasChange("tags") {
        // If an error occurs, return with an error,
        // we didn't finish updating
        if err := updateTags(d, meta); err != nil {
            return err
        }

        d.SetPartial("tags")
    }

    if d.HasChange("name") {
        if err := updateName(d, meta); err != nil {
            return err
        }

        d.SetPartial("name")
    }

    // We succeeded, disable partial mode
    d.Partial(false)

    return nil
}

```

In the example above, it is possible that setting the tags succeeds, but setting the name fails. In this scenario, we want to make sure that only the state of the tags is updated. To do this the `Partial` and `SetPartial` functions are used.

`Partial` toggles partial-state mode. When disabled, all changes are merged into the state upon result of the operation. When enabled, only changes enabled with `SetPartial` are merged in.

`SetPartial` tells Terraform what state changes to adopt upon completion of an operation. You should call `SetPartial` with every key that is safe to merge into the state. The parameter to `SetPartial` is a prefix, so if you have a nested structure and want to accept the whole thing, you can just specify the prefix.

Providers

Terraform is used to create, manage, and update infrastructure resources such as physical machines, VMs, network switches, containers, and more. Almost any infrastructure type can be represented as a resource in Terraform.

A provider is responsible for understanding API interactions and exposing resources. Providers generally are an IaaS (e.g. AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Enterprise, DNSimple, CloudFlare).

Use the navigation to the left to find available providers by type or scroll down to see all providers.

| | | |
|---|---|---|
| ACME (/docs/providers/acme/index.html) | Alicloud
(/docs/providers/alicloud/index.html) | Archive
(/docs/providers/archive/index.html) |
| Arukas (/docs/providers/arukas/index.html) | AWS (/docs/providers/aws/index.html) | Azure
(/docs/providers/azurerm/index.html) |
| Azure Stack
(/docs/providers/azurestack/index.html) | Bitbucket
(/docs/providers/bitbucket/index.html) | Brightbox
(/docs/providers/brightbox/index.html) |
| CenturyLinkCloud
(/docs/providers/clc/index.html) | Chef (/docs/providers/chef/index.html) | Circonus
(/docs/providers/circonus/index.html) |
| Cloudflare
(/docs/providers/cloudflare/index.html) | CloudScale.ch
(/docs/providers/cloudscale/index.html) | CloudStack
(/docs/providers/cloudstack/index.html) |
| Cobbler (/docs/providers/cobbler/index.html) | Consul
(/docs/providers/consul/index.html) | Datadog
(/docs/providers/datadog/index.html) |
| DigitalOcean (/docs/providers/do/index.html) | DNS (/docs/providers/dns/index.html) | DNSMadeEasy
(/docs/providers/dme/index.html) |
| DNSimple (/docs/providers/dnsimple/index.html) | Docker
(/docs/providers/docker/index.html) | Dyn (/docs/providers/dyn/index.html) |
| External (/docs/providers/external/index.html) | F5 BIG-IP
(/docs/providers/bigip/index.html) | Fastly (/docs/providers/fastly/index.html) |
| FlexibleEngine
(/docs/providers/flexibleengine/index.html) | GitHub
(/docs/providers/github/index.html) | Gitlab (/docs/providers/gitlab/index.html) |
| Google Cloud
(/docs/providers/google/index.html) | Grafana
(/docs/providers/grafana/index.html) | Hedvig
(/docs/providers/hedvig/index.html) |
| Helm (/docs/providers/helm/index.html) | Heroku
(/docs/providers/heroku/index.html) | Hetzner Cloud
(/docs/providers/hcloud/index.html) |
| HTTP (/docs/providers/http/index.html) | HuaweiCloud
(/docs/providers/huaweicloud/index.html) | Icinga2
(/docs/providers/icinga2/index.html) |
| Ignition (/docs/providers/ignition/index.html) | InfluxDB
(/docs/providers/influxdb/index.html) | Kubernetes
(/docs/providers/kubernetes/index.html) |
| Librato (/docs/providers/librato/index.html) | Linode
(/docs/providers/linode/index.html) | Local (/docs/providers/local/index.html) |
| Logentries
(/docs/providers/logentries/index.html) | LogicMonitor
(/docs/providers/logicmonitor/index.html) | Mailgun
(/docs/providers/mailgun/index.html) |
| MySQL (/docs/providers/mysql/index.html) | Netlify
(/docs/providers/netlify/index.html) | New Relic
(/docs/providers/newrelic/index.html) |
| Nomad (/docs/providers/nomad/index.html) | NS1 (/docs/providers/ns1/index.html) | Null (/docs/providers/null/index.html) |
| Nutanix (/docs/providers/nutanix/index.html) | 1&1
(/docs/providers/oneandone/index.html) | OpenStack
(/docs/providers/openstack/index.html) |
| OpenTelekomCloud
(/docs/providers/opentelekomcloud/index.html) | OpsGenie
(/docs/providers/opsgenie/index.html) | Oracle Cloud Infrastructure
(/docs/providers/oci/index.html) |

| | | |
|---|--|---|
| Oracle Cloud Platform
(/docs/providers/oraclepaas/index.html) | Oracle Public Cloud
(/docs/providers/opc/index.html) | OVH (/docs/providers/ovh/index.html) |
| Packet (/docs/providers/packet/index.html) | PagerDuty
(/docs/providers/pagerduty/index.html) | Palo Alto Networks
(/docs/providers/panos/index.html) |
| PostgreSQL
(/docs/providers/postgresql/index.html) | PowerDNS
(/docs/providers/powerdns/index.html) | ProfitBricks
(/docs/providers/profitbricks/index.html) |
| RabbitMQ (/docs/providers/rabbitmq/index.html) | Rancher
(/docs/providers/rancher/index.html) | Random
(/docs/providers/random/index.html) |
| RightScale
(/docs/providers/rightscale/index.html) | Rundeck
(/docs/providers/rundeck/index.html) | RunScope
(/docs/providers/runscope/index.html) |
| Scaleway (/docs/providers/scaleway/index.html) | Selectel
(/docs/providers/selvpc/index.html) | Skytap
(/docs/providers/skytap/index.html) |
| SoftLayer (/docs/providers/softlayer/index.html) | StatusCake
(/docs/providers/statuscake/index.html) | Spotinst
(/docs/providers/spotinst/index.html) |
| TelefonicaOpenCloud
(/docs/providers/telefonicaopencloud/index.html) | Template
(/docs/providers/template/index.html) | TencentCloud
(/docs/providers/tencentcloud/index.html) |
| Terraform
(/docs/providers/terraform/index.html) | Terraform Enterprise
(/docs/providers/tfe/index.html) | TLS (/docs/providers/tls/index.html) |
| Triton (/docs/providers/triton/index.html) | UCloud
(/docs/providers/ucloud/index.html) | UltraDNS
(/docs/providers/ultradns/index.html) |
| Vault (/docs/providers/vault/index.html) | VMware vCloud Director
(/docs/providers/vcd/index.html) | VMware NSX-T
(/docs/providers/nsxt/index.html) |
| VMware vSphere
(/docs/providers/vsphere/index.html) | <> | <> |

More providers can be found on our Community Providers (/docs/providers/type/community-index.html) page.

Terraform Commands (CLI)

Terraform is controlled via a very easy to use command-line interface (CLI). Terraform is only a single command-line application: `terraform`. This application then takes a subcommand such as "apply" or "plan". The complete list of subcommands is in the navigation to the left.

The `terraform` CLI is a well-behaved command line application. In erroneous cases, a non-zero exit status will be returned. It also responds to `-h` and `--help` as you'd most likely expect.

To view a list of the available commands at any time, just run `terraform` with no arguments:

```
$ terraform
Usage: terraform [--version] [--help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
  apply      Builds or changes infrastructure
  console    Interactive console for Terraform interpolations
  destroy    Destroy Terraform-managed infrastructure
  fmt        Rewrites config files to canonical format
  get        Download and install modules for the configuration
  graph     Create a visual graph of Terraform resources
  import    Import existing infrastructure into Terraform
  init      Initialize a new or existing Terraform configuration
  output    Read an output from a state file
  plan      Generate and show an execution plan
  providers Prints a tree of the providers used in the configuration
  push      Upload this Terraform module to Terraform Enterprise to run
  refresh   Update local state file against real resources
  show      Inspect Terraform state or plan
  taint     Manually mark a resource for recreation
  untaint  Manually unmark a resource as tainted
  validate  Validates the Terraform files
  version   Prints the Terraform version
  workspace Workspace management

All other commands:
  debug      Debug output management (experimental)
  force-unlock  Manually unlock the terraform state
  state      Advanced state management
```

To get help for any specific command, pass the `-h` flag to the relevant subcommand. For example, to see help about the `graph` subcommand:

```
$ terraform graph -h
Usage: terraform graph [options] PATH

Outputs the visual graph of Terraform resources. If the path given is
the path to a configuration, the dependency graph of the resources are
shown. If the path is a plan file, then the dependency graph of the
plan itself is shown.
```

The graph is outputted in DOT format. The typical program that can
read this format is GraphViz, but many web services are also available
to read this format.

Shell Tab-completion

If you use either bash or zsh as your command shell, Terraform can provide tab-completion support for all command names and (at this time) *some* command arguments.

To add the necessary commands to your shell profile, run the following command:

```
terraform -install-autocomplete
```

After installation, it is necessary to restart your shell or to re-read its profile script before completion will be activated.

To uninstall the completion hook, assuming that it has not been modified manually in the shell profile, run the following command:

```
terraform -uninstall-autocomplete
```

Currently not all of Terraform's subcommands have full tab-completion support for all arguments. We plan to improve tab-completion coverage over time.

Upgrade and Security Bulletin Checks

The Terraform CLI commands interact with the HashiCorp service Checkpoint (<https://checkpoint.hashicorp.com/>) to check for the availability of new versions and for critical security bulletins about the current version.

One place where the effect of this can be seen is in `terraform version`, where it is used by default to indicate in the output when a newer version is available.

Only anonymous information, which cannot be used to identify the user or host, is sent to Checkpoint. An anonymous ID is sent which helps de-duplicate warning messages. Both the anonymous id and the use of checkpoint itself are completely optional and can be disabled.

Checkpoint itself can be entirely disabled for all HashiCorp products by setting the environment variable `CHECKPOINT_DISABLE` to any non-empty value.

Alternatively, settings in the CLI configuration file (</docs/commands/cli-config.html>) can be used to disable checkpoint features. The following checkpoint-related settings are supported in this file:

- `disable_checkpoint` - set to `true` to disable checkpoint calls entirely. This is similar to the `CHECKPOINT_DISABLE` environment variable described above.
- `disable_checkpoint_signature` - set to `true` to disable the use of an anonymous signature in checkpoint requests. This allows Terraform to check for security bulletins but does not send the anonymous signature in these requests.

The Checkpoint client code (<https://github.com/hashicorp/go-checkpoint>) used by Terraform is available for review by any interested party.

Command: apply

The `terraform apply` command is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a `terraform plan` execution plan.

Usage

Usage: `terraform apply [options] [dir-or-plan]`

By default, `apply` scans the current directory for the configuration and applies the changes appropriately. However, a path to another configuration or an execution plan can be provided. Explicit execution plans files can be used to split plan and apply into separate steps within automation systems ([/guides/running-terraform-in-automation.html](#)).

The command-line flags are all optional. The list of available flags are:

- `-backup=path` - Path to the backup file. Defaults to `-state-out` with the `".backup"` extension. Disabled by setting to `".`.
- `-lock=true` - Lock the state file when locking is supported.
- `-lock-timeout=0s` - Duration to retry a state lock.
- `-input=true` - Ask for input for variables if not directly set.
- `-auto-approve` - Skip interactive approval of plan before applying.
- `-no-color` - Disables output with coloring.
- `-parallelism=n` - Limit the number of concurrent operation as Terraform walks the graph ([/docs/internals/graph.html#walking-the-graph](#)).
- `-refresh=true` - Update the state for each resource prior to planning and applying. This has no effect if a plan file is given directly to apply.
- `-state=path` - Path to the state file. Defaults to `"terraform.tfstate"`. Ignored when remote state ([/docs/state/remote.html](#)) is used.
- `-state-out=path` - Path to write updated state file. By default, the `-state` path will be used. Ignored when remote state ([/docs/state/remote.html](#)) is used.
- `-target=resource` - A Resource Address ([/docs/internals/resource-addressing.html](#)) to target. For more information, see the targeting docs from `terraform plan` ([/docs/commands/plan.html#resource-targeting](#)).
- `-var 'foo=bar'` - Set a variable in the Terraform configuration. This flag can be set multiple times. Variable values are interpreted as HCL ([/docs/configuration/syntax.html#HCL](#)), so list and map values can be specified via this flag.
- `-var-file=foo` - Set variables in the Terraform configuration from a variable file ([/docs/configuration/variables.html#variable-files](#)). If a `terraform.tfvars` or any `.auto.tfvars` files are present in the current directory, they will be automatically loaded. `terraform.tfvars` is loaded first and the `.auto.tfvars` files after in alphabetical order. Any files specified by `-var-file` override any values set automatically from files in the working directory. This flag can be used multiple times.

CLI Configuration File (.terraformrc/terraform.rc)

The CLI configuration file configures per-user settings for CLI behaviors, which apply across all Terraform working directories. This is separate from your infrastructure configuration (/docs/configuration/index.html).

Location

The configuration is placed in a single file whose location depends on the host operating system:

- On Windows, the file must be named named `terraform.rc` and placed in the relevant user's `%APPDATA%` directory. The physical location of this directory depends on your Windows version and system configuration; use `$env:APPDATA` in PowerShell to find its location on your system.
- On all other systems, the file must be named `.terraformrc` (note the leading period) and placed directly in the home directory of the relevant user.

On Windows, beware of Windows Explorer's default behavior of hiding filename extensions. Terraform will not recognize a file named `terraform.rc.txt` as a CLI configuration file, even though Windows Explorer may *display* its name as just `terraform.rc`. Use `dir` from PowerShell or Command Prompt to confirm the filename.

Configuration File Syntax

The configuration file uses the same *HCL* syntax as `.tf` files, but with different attributes and blocks. The following example illustrates the general syntax; see the following section for information on the meaning of each of these settings:

```
plugin_cache_dir  = "$HOME/.terraform.d/plugin-cache"
disable_checkpoint = true
```

Available Settings

The following settings can be set in the CLI configuration file:

- `disable_checkpoint` — when set to `true`, disables upgrade and security bulletin checks (/docs/commands/index.html#upgrade-and-security-bulletin-checks) that require reaching out to HashiCorp-provided network services.
- `disable_checkpoint_signature` — when set to `true`, allows the upgrade and security bulletin checks described above but disables the use of an anonymous id used to de-duplicate warning messages.
- `plugin_cache_dir` — enables plugin caching (/docs/configuration/providers.html#provider-plugin-cache) and specifies, as a string, the location of the plugin cache directory.
- `credentials` — provides credentials for use with Terraform Enterprise. Terraform uses this when performing remote operations or state access with the remote backend (/docs/backends/types/remote.html) and when accessing Terraform Enterprise's private module registry. (/docs/enterprise/registry/index.html)

This setting is a repeatable block, where the block label is a hostname (either `app.terraform.io` or the hostname of your private install) and the block body contains a `token` attribute. Whenever Terraform accesses state, modules, or remote operations from that hostname, it will authenticate with that API token.

```
credentials "app.terraform.io" {
  token = "xxxxxx.atlasv1.zzzzzzzzzzzz"
}
```

Important: The token provided here must be a user API token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)), and not a team or organization token.

Note: The credentials hostname must match the hostname in your module sources and/or backend configuration. If your Terraform Enterprise instance is available at multiple hostnames, use one of them consistently. (The SaaS version of Terraform Enterprise responds to API calls at both its newer hostname, `app.terraform.io`, and its historical hostname, `atlas.hashicorp.com`.)

Deprecated Settings

The following settings are supported for backward compatibility but are no longer recommended for use:

- `providers` - a configuration block that allows specifying the locations of specific plugins for each named provider. This mechanism is deprecated because it is unable to specify a version number for each plugin, and thus it does not co-operate with the plugin versioning mechanism. Instead, place the plugin executable files in the third-party plugins directory ([/docs/configuration/providers.html#third-party-plugins](#)).

Command: console

The `terraform console` command creates an interactive console for using interpolations ([/docs/configuration/interpolation.html](#)).

Usage

Usage: `terraform console [options] [dir]`

This opens an interactive console for experimenting with interpolations. This is useful for testing interpolations before using them in configurations as well as interacting with an existing state ([/docs/state/index.html](#)).

If a state file doesn't exist, the console still works and can be used to experiment with supported interpolation functions. Try entering some basic math such as `1 + 5` to see.

The `dir` argument can be used to open a console for a specific Terraform configuration directory. This will load any state from that directory as well as the configuration. This defaults to the current working directory. The `console` command does not require Terraform state or configuration to function.

The command-line flags are all optional. The list of available flags are:

- `-state=path` - Path to the state file. Defaults to `terraform.tfstate`. A state file doesn't need to exist.

You can close the console with the `exit` command or by using Control-C or Control-D.

Scripting

The `terraform console` command can be used in non-interactive scripts by piping newline-separated commands to it. Only the output from the final command is outputted unless an error occurs earlier.

An example is shown below:

```
$ echo "1 + 5" | terraform console
6
```

Remote State

The `terraform console` command will read configured state even if it is remote ([/docs/state/remote.html](#)). This is great for scripting state reading in CI environments or other remote scenarios.

After configuring remote state, run a `terraform remote pull` command to sync state locally. The `terraform console` command will use this state for operations.

Because the console currently isn't able to modify state in any way, this is a one way operation and you don't need to worry about remote state conflicts in any way.

Command: destroy

The `terraform destroy` command is used to destroy the Terraform-managed infrastructure.

Usage

Usage: `terraform destroy [options] [dir]`

Infrastructure managed by Terraform will be destroyed. This will ask for confirmation before destroying.

This command accepts all the arguments and flags that the `apply` command ([/docs/commands/apply.html](#)) accepts, with the exception of a plan file argument.

If `-auto-approve` is set, then the destroy confirmation will not be shown.

The `-target` flag, instead of affecting "dependencies" will instead also destroy any resources that *depend on* the target(s) specified.

The behavior of any `terraform destroy` command can be previewed at any time with an equivalent `terraform plan -destroy` command.

Command: env

The `terraform env` command is deprecated. The `terraform workspace` command (</docs/commands/workspace/>) should be used instead.

Command: fmt

The `terraform fmt` command is used to rewrite Terraform configuration files to a canonical format and style.

Usage

Usage: `terraform fmt [options] [DIR]`

By default, `fmt` scans the current directory for configuration files. If the `dir` argument is provided then it will scan that given directory instead. If `dir` is a single dash (-) then `fmt` will read from standard input (STDIN).

The command-line flags are all optional. The list of available flags are:

- `-list=true` - List files whose formatting differs (disabled if using STDIN)
- `-write=true` - Write result to source file instead of STDOUT (disabled if using STDIN or `-check`)
- `-diff=false` - Display diffs of formatting changes
- `-check=false` - Check if the input is formatted. Exit status will be 0 if all input is properly formatted and non-zero otherwise.

Command: force-unlock

Manually unlock the state for the defined configuration.

This will not modify your infrastructure. This command removes the lock on the state for the current configuration. The behavior of this lock is dependent on the backend being used. Local state files cannot be unlocked by another process.

Usage

Usage: `terraform force-unlock LOCK_ID [DIR]`

Manually unlock the state for the defined configuration.

This will not modify your infrastructure. This command removes the lock on the state for the current configuration. The behavior of this lock is dependent on the backend being used. Local state files cannot be unlocked by another process.

Options:

- `-force` - Don't ask for input for unlock confirmation.

Command: get

The `terraform get` command is used to download and update modules (`/docs/modules/index.html`) mentioned in the root module.

Usage

Usage: `terraform get [options] [dir]`

The modules are downloaded into a local `.terraform` folder. This folder should not be committed to version control. The `.terraform` folder is created relative to your current working directory regardless of the `dir` argument given to this command.

If a module is already downloaded and the `-update` flag is *not* set, Terraform will do nothing. As a result, it is safe (and fast) to run this command multiple times.

The command-line flags are all optional. The list of available flags are:

- `-update` - If specified, modules that are already downloaded will be checked for updates and the updates will be downloaded if present.
- `dir` - Sets the path of the root module (`/docs/modules/index.html#definitions`).

Command: graph

The `terraform graph` command is used to generate a visual representation of either a configuration or execution plan. The output is in the DOT format, which can be used by GraphViz (<http://www.graphviz.org>) to generate charts.

Usage

Usage: `terraform graph [options] [DIR]`

Outputs the visual dependency graph of Terraform resources according to configuration files in DIR (or the current directory if omitted).

The graph is outputted in DOT format. The typical program that can read this format is GraphViz, but many web services are also available to read this format.

The `-type` flag can be used to control the type of graph shown. Terraform creates different graphs for different operations. See the options below for the list of types supported. The default type is "plan" if a configuration is given, and "apply" if a plan file is passed as an argument.

Options:

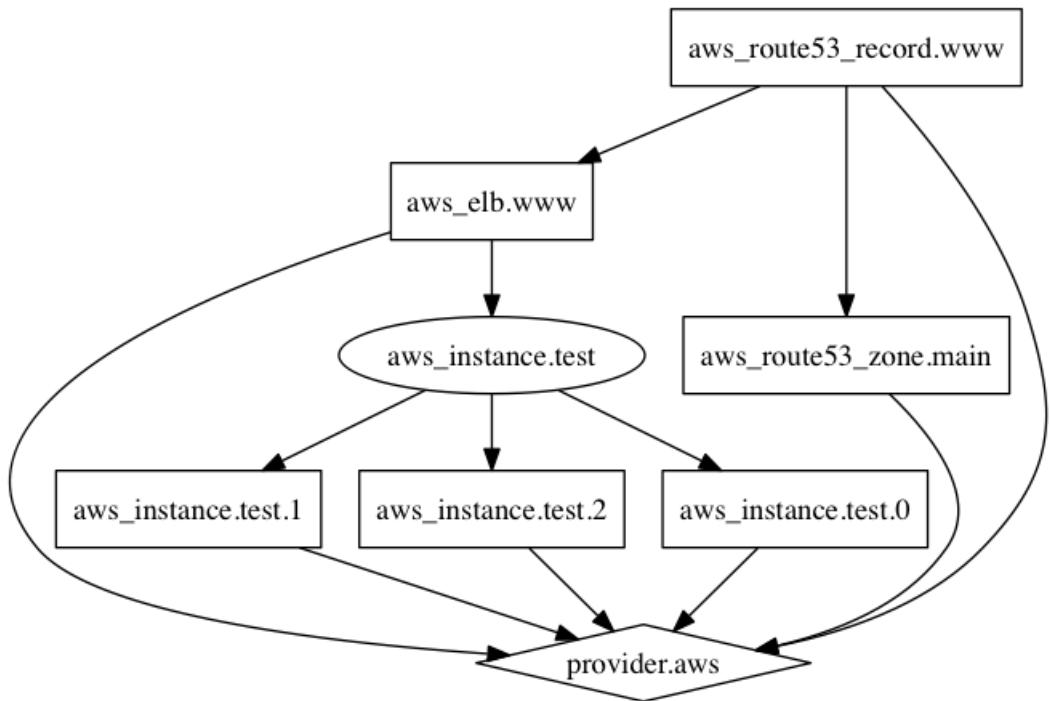
- `-draw-cycles` - Highlight any cycles in the graph with colored edges. This helps when diagnosing cycle errors.
- `-no-color` - If specified, output won't contain any color.
- `-type=plan` - Type of graph to output. Can be: plan, plan-destroy, apply, legacy.

Generating Images

The output of `terraform graph` is in the DOT format, which can easily be converted to an image by making use of dot provided by GraphViz:

```
$ terraform graph | dot -Tsvg > graph.svg
```

Here is an example graph output:



Command: import

The `terraform import` command is used to import existing resources ([/docs/import/index.html](#)) into Terraform.

Usage

Usage: `terraform import [options] ADDRESS ID`

Import will find the existing resource from ID and import it into your Terraform state at the given ADDRESS.

ADDRESS must be a valid resource address ([/docs/internals/resource-addressing.html](#)). Because any resource address is valid, the import command can import resources into modules as well directly into the root of your state.

ID is dependent on the resource type being imported. For example, for AWS instances it is the instance ID (`i-abcd1234`) but for AWS Route53 zones it is the zone ID (`Z12ABC4UGMOZ2N`). Please reference the provider documentation for details on the ID format. If you're unsure, feel free to just try an ID. If the ID is invalid, you'll just receive an error message.

The command-line flags are all optional. The list of available flags are:

- `-backup=path` - Path to backup the existing state file. Defaults to the `-state-out` path with the ".backup" extension. Set to "-" to disable backups.
- `-config=path` - Path to directory of Terraform configuration files that configure the provider for import. This defaults to your working directory. If this directory contains no Terraform configuration files, the provider must be configured via manual input or environmental variables.
- `-input=true` - Whether to ask for input for provider configuration.
- `-lock=true` - Lock the state file when locking is supported.
- `-lock-timeout=0s` - Duration to retry a state lock.
- `-no-color` - If specified, output won't contain any color.
- `-provider=provider` - Specified provider to use for import. The value should be a provider alias in the form `TYPE.ALIAS`, such as "aws.eu". This defaults to the normal provider based on the prefix of the resource being imported. You usually don't need to specify this.
- `-state=path` - Path to the source state file to read from. Defaults to the configured backend, or "terraform.tfstate".
- `-state-out=path` - Path to the destination state file to write to. If this isn't specified the source state file will be used. This can be a new or existing path.
- `-var 'foo=bar'` - Set a variable in the Terraform configuration. This flag can be set multiple times. Variable values are interpreted as HCL ([/docs/configuration/syntax.html#HCL](#)), so list and map values can be specified via this flag. This is only useful with the `-config` flag.
- `-var-file=foo` - Set variables in the Terraform configuration from a variable file ([/docs/configuration/variables.html#variable-files](#)). If a `terraform.tfvars` or any `.auto.tfvars` files are present in the current directory, they will be automatically loaded. `terraform.tfvars` is loaded first and the `.auto.tfvars` files after in alphabetical order. Any files specified by `-var-file` override any values set automatically from files in the working directory. This flag can be used multiple times. This is only useful with the `-config` flag.

Provider Configuration

Terraform will attempt to load configuration files that configure the provider being used for import. If no configuration files are present or no configuration for that specific provider is present, Terraform will prompt you for access credentials. You may also specify environmental variables to configure the provider.

The only limitation Terraform has when reading the configuration files is that the import provider configurations must not depend on non-variable inputs. For example, a provider configuration cannot depend on a data source.

As a working example, if you're importing AWS resources and you have a configuration file with the contents below, then Terraform will configure the AWS provider with this file.

```
variable "access_key" {}
variable "secret_key" {}

provider "aws" {
  access_key = "${var.access_key}"
  secret_key = "${var.secret_key}"
}
```

You can force Terraform to explicitly not load your configuration by specifying `-config=""` (empty string). This is useful in situations where you want to manually configure the provider because your configuration may not be valid.

Example: AWS Instance

This example will import an AWS instance:

```
$ terraform import aws_instance.foo i-abcd1234
```

Example: Import to Module

The example below will import an AWS instance into a module:

```
$ terraform import module.foo.aws_instance.bar i-abcd1234
```

Command: init

The `terraform init` command is used to initialize a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.

Usage

Usage: `terraform init [options] [DIR]`

This command performs several different initialization steps in order to prepare a working directory for use. More details on these are in the sections below, but in most cases it is not necessary to worry about these individual steps.

This command is always safe to run multiple times, to bring the working directory up to date with changes in the configuration. Though subsequent runs may give errors, this command will never delete your existing configuration or state.

If no arguments are given, the configuration in the current working directory is initialized. It is recommended to run Terraform with the current working directory set to the root directory of the configuration, and omit the `DIR` argument.

General Options

The following options apply to all of (or several of) the initialization steps:

- `-input=true` Ask for input if necessary. If false, will error if input was required.
- `-lock=false` Disable locking of state files during state-related operations.
- `-lock-timeout=<duration>` Override the time Terraform will wait to acquire a state lock. The default is 0s (zero seconds), which causes immediate failure if the lock is already held by another process.
- `-no-color` Disable color codes in the command output.
- `-upgrade` Opt to upgrade modules and plugins as part of their respective installation steps. See the sections below for more details.

Copy a Source Module

By default, `terraform init` assumes that the working directory already contains a configuration and will attempt to initialize that configuration.

Optionally, `init` can be run against an empty directory with the `-from-module=MODULE-SOURCE` option, in which case the given module will be copied into the target directory before any other initialization steps are run.

This special mode of operation supports two use-cases:

- Given a version control source, it can serve as a shorthand for checking out a configuration from version control and then initializing the work directory for it.
- If the source refers to an *example* configuration, it can be copied into a local directory to be used as a basis for a new configuration.

For routine use it is recommended to check out configuration from version control separately, using the version control system's own commands. This way it is possible to pass extra flags to the version control system when necessary, and to perform other preparation steps (such as configuration generation, or activating credentials) before running `terraform init`.

Backend Initialization

During `init`, the root configuration directory is consulted for backend configuration ([/docs/backends/config.html](#)) and the chosen backend is initialized using the given configuration settings.

Re-running `init` with an already-initialized backend will update the working directory to use the new backend settings. Depending on what changed, this may result in interactive prompts to confirm migration of workspace states. The `-force-copy` option suppresses these prompts and answers "yes" to the migration questions. The `-reconfigure` option disregards any existing configuration, preventing migration of any existing state.

To skip backend configuration, use `-backend=false`. Note that some other `init` steps require an initialized backend, so it is recommended to use this flag only when the working directory was already previously initialized for a particular backend.

The `-backend-config=...` option can be used for partial backend configuration ([/docs/backends/config.html#partial-configuration](#)), in situations where the backend settings are dynamic or sensitive and so cannot be statically specified in the configuration file.

Child Module Installation

During `init`, the configuration is searched for module blocks, and the source code for referenced modules ([/docs/modules/](#)) is retrieved from the locations given in their `source` arguments.

Re-running `init` with modules already installed will install the sources for any modules that were added to configuration since the last `init`, but will not change any already-installed modules. Use `-upgrade` to override this behavior, updating all modules to the latest available source code.

To skip child module installation, use `-get=false`. Note that some other `init` steps can complete only when the module tree is complete, so it's recommended to use this flag only when the working directory was already previously initialized with its child modules.

Plugin Installation

During `init`, Terraform searches the configuration for both direct and indirect references to providers and attempts to load the required plugins.

For providers distributed by HashiCorp ([/docs/providers/index.html](#)), `init` will automatically download and install plugins if necessary. Plugins can also be manually installed in the user `plugins` directory, located at `~/.terraform.d/plugins` on most operating systems and `%APPDATA%\terraform.d\plugins` on Windows.

For more information about configuring and installing providers, see [Configuration: Providers](#) ([/docs/configuration/providers.html](#)).

On subsequent runs, `init` only installs providers without acceptable versions installed. (This includes newly added providers, and providers whose installed versions can't meet the current version constraints.) Use `-upgrade` if you want to update *all* providers to the newest acceptable version.

You can modify `terraform init`'s plugin behavior with the following options:

- `-upgrade` — Update all previously installed plugins to the newest version that complies with the configuration's version constraints. This option does not apply to manually installed plugins.
- `-get-plugins=false` — Skips plugin installation. Terraform will use plugins installed in the user plugins directory, and any plugins already installed for the current working directory. If the installed plugins aren't sufficient for the configuration, `init` fails.
- `-plugin-dir=PATH` — Skips plugin installation and loads plugins *only* from the specified directory. This ignores the user plugins directory and any plugins already installed in the current working directory. To restore the default behavior after using this option, run `init` again and pass an empty string to `-plugin-dir`.
- `-verify-plugins=false` — Skips release signature validation when installing downloaded plugins (not recommended). Official plugin releases are digitally signed by HashiCorp, and Terraform verifies these signatures when automatically downloading plugins. This option disables that verification. (Terraform does not check signatures for manually installed plugins.)

Running `terraform init` in automation

For teams that use Terraform as a key part of a change management and deployment pipeline, it can be desirable to orchestrate Terraform runs in some sort of automation in order to ensure consistency between runs, and provide other interesting features such as integration with version control hooks.

There are some special concerns when running `init` in such an environment, including optionally making plugins available locally to avoid repeated re-installation. For more information, see [Running Terraform in Automation](#) (/guides/running-terraform-in-automation.html).

Command: output

The `terraform output` command is used to extract the value of an output variable from the state file.

Usage

Usage: `terraform output [options] [NAME]`

With no additional arguments, `output` will display all the outputs for the root module. If an output `NAME` is specified, only the value of that output is printed.

The command-line flags are all optional. The list of available flags are:

- `-json` - If specified, the outputs are formatted as a JSON object, with a key per output. If `NAME` is specified, only the output specified will be returned. This can be piped into tools such as `jq` for further processing.
- `-state=path` - Path to the state file. Defaults to "terraform.tfstate". Ignored when remote state (/docs/state/remote.html) is used.
- `-module=module_name` - The module path which has needed output. By default this is the root path. Other modules can be specified by a period-separated list. Example: "foo" would reference the module "foo" but "foo.bar" would reference the "bar" module in the "foo" module.

Examples

These examples assume the following Terraform output snippet.

```
output "lb_address" {
  value = "${aws_alb.web.public_dns}"
}

output "instance_ips" {
  value = ["${aws_instance.web.*.public_ip}"]
}
```

To list all outputs:

```
$ terraform output
```

To query for the DNS address of the load balancer:

```
$ terraform output lb_address
my-app-alb-1657023003.us-east-1.elb.amazonaws.com
```

To query for all instance IP addresses:

```
$ terraform output instance_ips
test = [
  54.43.114.12,
  52.122.13.4,
  52.4.116.53
]
```

To query for a particular value in a list, use `-json` and a JSON command-line parser such as `jq` (<https://stedolan.github.io/jq/>).

For example, to query for the first instance's IP address:

```
$ terraform output -json instance_ips | jq '.value[0]'
```

Command: plan

The `terraform plan` command is used to create an execution plan. Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.

This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state. For example, `terraform plan` might be run before committing a change to version control, to create confidence that it will behave as expected.

The optional `-out` argument can be used to save the generated plan to a file for later execution with `terraform apply`, which can be useful when running Terraform in automation ([/guides/running-terraform-in-automation.html](#)).

Usage

Usage: `terraform plan [options] [dir-or-plan]`

By default, `plan` requires no flags and looks in the current directory for the configuration and state file to refresh.

If the command is given an existing saved plan as an argument, the command will output the contents of the saved plan. In this scenario, the `plan` command will not modify the given plan. This can be used to inspect a planfile.

The command-line flags are all optional. The list of available flags are:

- `-destroy` - If set, generates a plan to destroy all the known resources.
- `-detailed-exitcode` - Return a detailed exit code when the command exits. When provided, this argument changes the exit codes and their meanings to provide more granular information about what the resulting plan contains:
 - 0 = Succeeded with empty diff (no changes)
 - 1 = Error
 - 2 = Succeeded with non-empty diff (changes present)
- `-input=true` - Ask for input for variables if not directly set.
- `-lock=true` - Lock the state file when locking is supported.
- `-lock-timeout=0s` - Duration to retry a state lock.
- `-module-depth=n` - Specifies the depth of modules to show in the output. This does not affect the plan itself, only the output shown. By default, this is `-1`, which will expand all.
- `-no-color` - Disables output with coloring.
- `-out=path` - The path to save the generated execution plan. This plan can then be used with `terraform apply` to be certain that only the changes shown in this plan are applied. Read the warning on saved plans below.
- `-parallelism=n` - Limit the number of concurrent operation as Terraform walks the graph ([/docs/internals/graph.html#walking-the-graph](#)).
- `-refresh=true` - Update the state prior to checking for differences.
- `-state=path` - Path to the state file. Defaults to "terrafrom.tfstate". Ignored when remote state ([/docs/state/remote.html](#)) is used.

- `-target=resource` - A Resource Address ([/docs/internals/resource-addressing.html](#)) to target. This flag can be used multiple times. See below for more information.
- `-var 'foo=bar'` - Set a variable in the Terraform configuration. This flag can be set multiple times. Variable values are interpreted as HCL ([/docs/configuration/syntax.html#HCL](#)), so list and map values can be specified via this flag.
- `-var-file=foo` - Set variables in the Terraform configuration from a variable file ([/docs/configuration/variables.html#variable-files](#)). If a `terraform.tfvars` or any `.auto.tfvars` files are present in the current directory, they will be automatically loaded. `terraform.tfvars` is loaded first and the `.auto.tfvars` files after in alphabetical order. Any files specified by `-var-file` override any values set automatically from files in the working directory. This flag can be used multiple times.

Resource Targeting

The `-target` option can be used to focus Terraform's attention on only a subset of resources. Resource Address ([/docs/internals/resource-addressing.html](#)) syntax is used to specify the constraint. The resource address is interpreted as follows:

- If the given address has a *resource spec*, only the specified resource is targeted. If the named resource uses `count` and no explicit index is specified in the address, all of the instances sharing the given resource name are targeted.
- If the given address *does not* have a resource spec, and instead just specifies a module path, the target applies to all resources in the specified module *and* all of the descendent modules of the specified module.

This targeting capability is provided for exceptional circumstances, such as recovering from mistakes or working around Terraform limitations. It is *not recommended* to use `-target` for routine operations, since this can lead to undetected configuration drift and confusion about how the true state of resources relates to configuration.

Instead of using `-target` as a means to operate on isolated portions of very large configurations, prefer instead to break large configurations into several smaller configurations that can each be independently applied. Data sources ([/docs/configuration/data-sources.html](#)) can be used to access information about resources created in other configurations, allowing a complex system architecture to be broken down into more manageable parts that can be updated independently.

Security Warning

Saved plan files (with the `-out` flag) encode the configuration, state, diff, and *variables*. Variables are often used to store secrets. Therefore, the plan file can potentially store secrets.

Terraform itself does not encrypt the plan file. It is highly recommended to encrypt the plan file if you intend to transfer it or keep it at rest for an extended period of time.

Future versions of Terraform will make plan files more secure.

Command: providers

The `terraform providers` command prints information about the providers used in the current configuration.

Provider dependencies are created in several different ways:

- Explicit use of a provider block in configuration, optionally including a version constraint.
- Use of any resource belonging to a particular provider in a `resource` or `data` block in configuration.
- Existence of any resource instance belonging to a particular provider in the current *state*. For example, if a particular resource is removed from configuration, it continues to create a dependency on its provider until its instances have been destroyed.

This command gives an overview of all of the current dependencies, as an aid to understanding why a particular provider is needed.

Usage

Usage: `terraform providers [config-path]`

Pass an explicit configuration path to override the default of using the current working directory.

Command: push

Important: The `terraform push` command is deprecated, and only works with the legacy version of Terraform Enterprise (</docs/enterprise-legacy/index.html>). In the current version of Terraform Enterprise, you can upload configurations using the API. See the docs about API-driven runs (</docs/enterprise/run/api.html>) for more details.

The `terraform push` command uploads your Terraform configuration to be managed by HashiCorp's Terraform Enterprise (<https://www.hashicorp.com/products/terraform/>). Terraform Enterprise can automatically run Terraform for you, save all state transitions, save plans, and keep a history of all Terraform runs.

This makes it significantly easier to use Terraform as a team: team members modify the Terraform configurations locally and continue to use normal version control. When the Terraform configurations are ready to be run, they are pushed to Terraform Enterprise, and any member of your team can run Terraform with the push of a button.

Terraform Enterprise can also be used to set ACLs on who can run Terraform, and a future update of Terraform Enterprise will allow parallel Terraform runs and automatically perform infrastructure locking so only one run is modifying the same infrastructure at a time.

When using this command, it is important to match your local Terraform version with the version selected for the target workspace in Terraform Enterprise, since otherwise the uploaded configuration archive may not be compatible with the remote Terraform process.

Usage

Usage: `terraform push [options] [path]`

The path argument is the same as for the `apply` (</docs/commands/apply.html>) command.

The command-line flags are all optional. The list of available flags are:

- `-atlas-address=<url>` - An alternate address to an instance. Defaults to <https://atlas.hashicorp.com>.
- `-upload-modules=true` - If true (default), then the modules (</docs/modules/index.html>) being used are all locked at their current checkout and uploaded completely. This prevents Terraform Enterprise from running `terraform get` for you.
- `-name=<name>` - Name of the infrastructure configuration in Terraform Enterprise. The format of this is: "username/name" so that you can upload configurations not just to your account but to other accounts and organizations. This setting can also be set in the configuration in the Terraform Enterprise section (</docs/configuration/terraform-enterprise.html>).
- `-no-color` - Disables output with coloring
- `-overwrite=foo` - Marks a specific variable to be updated. Normally, if a variable is already set Terraform will not send the local value (even if it is different). This forces it to send the local value to Terraform Enterprise. This flag can be repeated multiple times.
- `-token=<token>` - Terraform Enterprise API token to use to authorize the upload. If blank or unspecified, the `ATLAS_TOKEN` environment variable will be used.

- `-var='foo=bar'` - Set the value of a variable for the Terraform configuration.
- `-var-file=foo` - Set the value of variables using a variable file. This flag can be used multiple times.
- `-vcs=true` - If true (default), then Terraform will detect if a VCS is in use, such as Git, and will only upload files that are committed to version control. If no version control system is detected, Terraform will upload all files in path (parameter to the command).

Packaged Files

The files that are uploaded and packaged with a push are all the files in the path given as the parameter to the command, recursively. By default (unless `-vcs=false` is specified), Terraform will automatically detect when a VCS such as Git is being used, and in that case will only upload the files that are committed. Because of this built-in intelligence, you don't have to worry about excluding folders such as `".git"` or `".hg"` usually.

If Terraform doesn't detect a VCS, it will upload all files.

The reason Terraform uploads all of these files is because Terraform cannot know what is and isn't being used for provisioning, so it uploads all the files to be safe. To exclude certain files, specify the `-exclude` flag when pushing, or specify the `exclude` parameter in the Terraform Enterprise configuration section ([/docs/configuration/terraform-enterprise.html](#)).

Terraform also includes in the package all of the modules that were installed during the most recent `terraform init` or `terraform get` command. Since the details of how modules are cached in the filesystem vary between Terraform versions, it is important to use the same version of Terraform both locally (when running `terraform init` and then `terraform push`) and in your remote Terraform Enterprise workspace.

Terraform Variables

When you push, Terraform will automatically set the local values of your Terraform variables on Terraform Enterprise. The values are only set if they don't already exist. If you want to force push a certain variable value to update it, use the `-overwrite` flag.

All the variable values stored are encrypted and secured using Vault (<https://www.vaultproject.io>). We blogged about the architecture of our secure storage system (<https://www.hashicorp.com/blog/how-atlas-uses-vault-for-managing-secrets.html>) if you want more detail.

The variable values can be updated using the `-overwrite` flag or via the Terraform Enterprise website (<https://www.hashicorp.com/products/terraform/>). An example of updating just a single variable `foo` is shown below:

```
$ terraform push -var 'foo=bar' -overwrite foo
```

Both the `-var` and `-overwrite` flag are required. The `-var` flag sets the value locally (the exact same process as commands such as `apply` or `plan`), and the `-overwrite` flag tells the `push` command to update Terraform Enterprise.

Remote State Requirement

Important: This section only refers to the legacy version of Terraform Enterprise. The current version of Terraform

Enterprise always manages its own state, and does not support arbitrary remote state backends.

`terraform push` requires that remote state ([/docs/state/remote.html](#)) is enabled. The reasoning for this is simple: `terraform push` sends your configuration to be managed remotely. For it to keep the state in sync and for you to be able to easily access that state, remote state must be enabled instead of juggling local files.

While `terraform push` sends your configuration to be managed by Terraform Enterprise, the remote state backend *does not* have to be Terraform Enterprise. It can be anything as long as it is accessible by the public internet, since Terraform Enterprise will need to be able to communicate to it.

Warning: The credentials for accessing the remote state will be sent up to Terraform Enterprise as well. Therefore, we recommend you use access keys that are restricted if possible.

Command: refresh

The `terraform refresh` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file.

This does not modify infrastructure, but does modify the state file. If the state is changed, this may cause changes to occur during the next plan or apply.

Usage

Usage: `terraform refresh [options] [dir]`

By default, `refresh` requires no flags and looks in the current directory for the configuration and state file to refresh.

The command-line flags are all optional. The list of available flags are:

- `-backup=path` - Path to the backup file. Defaults to `-state-out` with the `".backup"` extension. Disabled by setting to `"."`.
- `-input=true` - Ask for input for variables if not directly set.
- `-lock=true` - Lock the state file when locking is supported.
- `-lock-timeout=0s` - Duration to retry a state lock.
- `-no-color` - If specified, output won't contain any color.
- `-state=path` - Path to read and write the state file to. Defaults to `"terraform.tfstate"`. Ignored when remote state (`/docs/state/remote.html`) is used.
- `-state-out=path` - Path to write updated state file. By default, the `-state` path will be used. Ignored when remote state (`/docs/state/remote.html`) is used.
- `-target=resource` - A Resource Address (`/docs/internals/resource-addressing.html`) to target. Operation will be limited to this resource and its dependencies. This flag can be used multiple times.
- `-var 'foo=bar'` - Set a variable in the Terraform configuration. This flag can be set multiple times. Variable values are interpreted as HCL (`/docs/configuration/syntax.html#HCL`), so list and map values can be specified via this flag.
- `-var-file=foo` - Set variables in the Terraform configuration from a variable file (`/docs/configuration/variables.html#variable-files`). If a `terraform.tfvars` or any `.auto.tfvars` files are present in the current directory, they will be automatically loaded. `terraform.tfvars` is loaded first and the `.auto.tfvars` files after in alphabetical order. Any files specified by `-var-file` override any values set automatically from files in the working directory. This flag can be used multiple times.

Command: show

The `terraform show` command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Usage

Usage: `terraform show [options] [path]`

You may use `show` with a path to either a Terraform state file or plan file. If no path is specified, the current state will be shown.

The command-line flags are all optional. The list of available flags are:

- `-module-depth=n` - Specifies the depth of modules to show in the output. By default this is `-1`, which will expand all.
- `-no-color` - Disables output with coloring

State Command

The `terraform state` command is used for advanced state management. As your Terraform usage becomes more advanced, there are some cases where you may need to modify the Terraform state (</docs/state/index.html>). Rather than modify the state directly, the `terraform state` commands can be used in many cases instead.

This command is a nested subcommand, meaning that it has further subcommands. These subcommands are listed to the left.

Usage

Usage: `terraform state <subcommand> [options] [args]`

Please click a subcommand to the left for more information.

Remote State

The Terraform state subcommands all work with remote state just as if it was local state. Reads and writes may take longer than normal as each read and each write do a full network roundtrip. Otherwise, backups are still written to disk and the CLI usage is the same as if it were local state.

Backups

All `terraform state` subcommands that modify the state write backup files. The path of these backup file can be controlled with `-backup`.

Subcommands that are read-only (such as `list` (</docs/commands/state/list.html>)) do not write any backup files since they aren't modifying the state.

Note that backups for state modification *can not be disabled*. Due to the sensitivity of the state file, Terraform forces every state modification command to write a backup file. You'll have to remove these files manually if you don't want to keep them around.

Command-Line Friendly

The output and command-line structure of the state subcommands is designed to be easy to use with Unix command-line tools such as `grep`, `awk`, etc. Consequently, the output is also friendly to the equivalent PowerShell commands within Windows.

For advanced filtering and modification, we recommend piping Terraform state subcommands together with other command line tools.

Resource Addressing

The `terraform state` subcommands make heavy use of resource addressing for targeting and filtering specific resources and modules within the state.

Resource addressing is a common feature of Terraform that is used in multiple locations. For example, resource addressing syntax is also used for the `-target` flag for `apply` and `plan` commands.

Because resource addressing is unified across Terraform, it is documented in a single place rather than duplicating it in multiple locations. You can find the resource addressing documentation here ([/docs/internals/resource-addressing.html](#)).

Provisioners

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction. Provisioners can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc.

Provisioners are added directly to any resource:

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} > file.txt"
  }
}
```

For provisioners other than local execution, you must specify connection settings ([/docs/provisioners/connection.html](#)) so Terraform knows how to communicate with the resource.

Creation-Time Provisioners

Provisioners by default run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system.

If a creation-time provisioner fails, the resource is marked as **tainted**. A tainted resource will be planned for destruction and recreation upon the next `terraform apply`. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is tainting.

You can change this behavior by setting the `on_failure` attribute, which is covered in detail below.

Destroy-Time Provisioners

If `when = "destroy"` is specified, the provisioner will run when the resource it is defined within is *destroyed*.

Destroy provisioners are run before the resource is destroyed. If they fail, Terraform will error and rerun the provisioners again on the next `terraform apply`. Due to this behavior, care should be taken for destroy provisioners to be safe to run multiple times.

Destroy-time provisioners can only run if they remain in the configuration at the time a resource is destroyed. If a resource block with a destroy-time provisioner is removed entirely from the configuration, its provisioner configurations are removed along with it and thus the destroy provisioner won't run. To work around this, a multi-step process can be used to safely remove a resource with a destroy-time provisioner:

- Update the resource configuration to include `count = 0`.
- Apply the configuration to destroy any existing instances of the resource, including running the destroy provisioner.
- Remove the resource block entirely from configuration, along with its provisioner blocks.
- Apply again, at which point no further action should be taken since the resources were already destroyed.

This limitation may be addressed in future versions of Terraform. For now, destroy-time provisioners must be used sparingly and with care.

Multiple Provisioners

Multiple provisioners can be specified within a resource. Multiple provisioners are executed in the order they're defined in the configuration file.

You may also mix and match creation and destruction provisioners. Only the provisioners that are valid for a given operation will be run. Those valid provisioners will be run in the order they're defined in the configuration file.

Example of multiple provisioners:

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo first"
  }

  provisioner "local-exec" {
    command = "echo second"
  }
}
```

Failure Behavior

By default, provisioners that fail will also cause the Terraform apply itself to error. The `on_failure` setting can be used to change this. The allowed values are:

- "continue" - Ignore the error and continue with creation or destruction.
- "fail" - Error (the default behavior). If this is a creation provisioner, taint the resource.

Example:

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command      = "echo ${self.private_ip} > file.txt"
    on_failure   = "continue"
  }
}
```

Chef Provisioner

The chef provisioner installs, configures and runs the Chef Client on a remote resource. The chef provisioner supports both ssh and winrm type connections ([/docs/provisioners/connection.html](#)).

Requirements

The chef provisioner has some prerequisites for specific connection types:

- For ssh type connections, curl must be available on the remote host.
- For winrm connections, PowerShell 2.0 must be available on the remote host.

Without these prerequisites, your provisioning execution will fail.

Example usage

```
resource "aws_instance" "web" {
  # ...

  provisioner "chef" {
    attributes_json = <<-EOF
    {
      "key": "value",
      "app": {
        "cluster1": {
          "nodes": [
            "webserver1",
            "webserver2"
          ]
        }
      }
    }
  EOF

  environment      = "_default"
  run_list        = ["cookbook::recipe"]
  node_name       = "webserver1"
  secret_key      = "${file("../encrypted_data_bag_secret")}"
  server_url      = "https://chef.company.com/organizations/org1"
  recreate_client = true
  user_name       = "bork"
  user_key        = "${file("../bork.pem")}"
  version         = "12.4.1"
  # If you have a self signed cert on your chef server change this to :verify_none
  ssl_verify_mode = ":verify_peer"
}
}
```

Argument Reference

The following arguments are supported:

- `attributes_json` (string) - (Optional) A raw JSON string with initial node attributes for the new node. These can also be loaded from a file on disk using the `file()` interpolation function ([/docs/configuration/interpolation.html#file_path](#)).
- `channel` (string) - (Optional) The Chef Client release channel to install from. If not set, the `stable` channel will be used.
- `client_options` (array) - (Optional) A list of optional Chef Client configuration options. See the Chef Client ([https://docs.chef.io/config_rb_client.html](#)) documentation for all available options.
- `disable_reporting` (boolean) - (Optional) If `true` the Chef Client will not try to send reporting data (used by Chef Reporting) to the Chef Server (defaults to `false`).
- `environment` (string) - (Optional) The Chef environment the new node will be joining (defaults to `_default`).
- `fetch_chef_certificates` (boolean) (Optional) If `true` the SSL certificates configured on your Chef Server will be fetched and trusted. See the `knife ssl_fetch` ([https://docs.chef.io/knife_ssl_fetch.html](#)) documentation for more details.
- `log_to_file` (boolean) - (Optional) If `true`, the output of the initial Chef Client run will be logged to a local file instead of the console. The file will be created in a subdirectory called `logfiles` created in your current directory. The filename will be the `node_name` of the new node.
- `use_policyfile` (boolean) - (Optional) If `true`, use the policy files to bootstrap the node. Setting `policy_group` and `policy_name` are required if this is `true`. (defaults to `false`).
- `policy_group` (string) - (Optional) The name of a policy group that exists on the Chef server. Required if `use_policyfile` is set; `policy_name` must also be specified.
- `policy_name` (string) - (Optional) The name of a policy, as identified by the `name` setting in a Policyfile.rb file. Required if `use_policyfile` is set; `policy_group` must also be specified.
- `http_proxy` (string) - (Optional) The proxy server for Chef Client HTTP connections.
- `https_proxy` (string) - (Optional) The proxy server for Chef Client HTTPS connections.
- `named_run_list` (string) - (Optional) The name of an alternate run-list to invoke during the initial Chef Client run. The run-list must already exist in the Policyfile that defines `policy_name`. Only applies when `use_policyfile` is `true`.
- `no_proxy` (array) - (Optional) A list of URLs that should bypass the proxy.
- `node_name` (string) - (Required) The name of the node to register with the Chef Server.
- `ohai_hints` (array) - (Optional) A list with Ohai hints ([https://docs.chef.io/ohai.html#hints](#)) to upload to the node.
- `os_type` (string) - (Optional) The OS type of the node. Valid options are: `linux` and `windows`. If not supplied, the connection type will be used to determine the OS type (`ssh` will assume `linux` and `winrm` will assume `windows`).
- `prevent_sudo` (boolean) - (Optional) Prevent the use of the `sudo` command while installing, configuring and running the initial Chef Client run. This option is only used with `ssh` type connections ([/docs/provisioners/connection.html](#)).
- `recreate_client` (boolean) - (Optional) If `true`, first delete any existing Chef Node and Client before registering the new Chef Client.
- `run_list` (array) - (Optional) A list with recipes that will be invoked during the initial Chef Client run. The run-list will also be saved to the Chef Server after a successful initial run. Required if `use_policyfile` is `false`; ignored when `use_policyfile` is `true` (see `named_run_list` to specify a run-list defined in a Policyfile).

- `secret_key` (`string`) - (Optional) The contents of the secret key that is used by the Chef Client to decrypt data bags on the Chef Server. The key will be uploaded to the remote machine. This can also be loaded from a file on disk using the `file()` interpolation function ([/docs/configuration/interpolation.html#file_path_](#)).
- `server_url` (`string`) - (Required) The URL to the Chef server. This includes the path to the organization. See the example.
- `skip_install` (`boolean`) - (Optional) Skip the installation of Chef Client on the remote machine. This assumes Chef Client is already installed when you run the `chef` provisioner.
- `skip_register` (`boolean`) - (Optional) Skip the registration of Chef Client on the remote machine. This assumes Chef Client is already registered and the private key (`client.pem`) is available in the default Chef configuration directory when you run the `chef` provisioner.
- `ssl_verify_mode` (`string`) - (Optional) Used to set the verify mode for Chef Client HTTPS requests. The options are `:verify_none`, or `:verify_peer` which is default.
- `user_name` (`string`) - (Required) The name of an existing Chef user to register the new Chef Client and optionally configure Chef Vaults.
- `user_key` (`string`) - (Required) The contents of the user key that will be used to authenticate with the Chef Server. This can also be loaded from a file on disk using the `file()` interpolation function ([/docs/configuration/interpolation.html#file_path_](#)).
- `vault_json` (`string`) - (Optional) A raw JSON string with Chef Vaults and Items to which the new node should have access. These can also be loaded from a file on disk using the `file()` interpolation function ([/docs/configuration/interpolation.html#file_path_](#)).
- `version` (`string`) - (Optional) The Chef Client version to install on the remote machine. If not set, the latest available version will be installed.

Provisioner Connections

Many provisioners require access to the remote resource. For example, a provisioner may need to use SSH or WinRM to connect to the resource.

Terraform uses a number of defaults when connecting to a resource, but these can be overridden using a connection block in either a resource or provisioner. Any connection information provided in a resource will apply to all the provisioners, but it can be scoped to a single provisioner as well. One use case is to have an initial provisioner connect as the `root` user to setup user accounts, and have subsequent provisioners connect as a user with more limited permissions.

Example usage

```
# Copies the file as the root user using SSH
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "/etc/myapp.conf"

  connection {
    type      = "ssh"
    user      = "root"
    password = "${var.root_password}"
  }
}

# Copies the file as the Administrator user using WinRM
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "C:/App/myapp.conf"

  connection {
    type      = "winrm"
    user      = "Administrator"
    password = "${var.admin_password}"
  }
}
```

Argument Reference

The following arguments are supported by all connection types:

- `type` - The connection type that should be used. Valid types are `ssh` and `winrm`. Defaults to `ssh`.
- `user` - The user that we should use for the connection. Defaults to `root` when using type `ssh` and defaults to `Administrator` when using type `winrm`.
- `password` - The password we should use for the connection. In some cases this is specified by the provider.
- `host` - The address of the resource to connect to. This is usually specified by the provider.
- `port` - The port to connect to. Defaults to 22 when using type `ssh` and defaults to 5985 when using type `winrm`.
- `timeout` - The timeout to wait for the connection to become available. This defaults to 5 minutes. Should be provided as a string like `30s` or `5m`.

- `script_path` - The path used to copy scripts meant for remote execution.

Additional arguments only supported by the ssh connection type:

- `private_key` - The contents of an SSH key to use for the connection. These can be loaded from a file on disk using the `file()` interpolation function ([/docs/configuration/interpolation.html#file_path_](#)). This takes preference over the password if provided.
- `agent` - Set to `false` to disable using `ssh-agent` to authenticate. On Windows the only supported SSH authentication agent is Pageant (<http://the.earth.li/~sgtatham/putty/0.66/htmldoc/Chapter9.html#pageant>).
- `agent_identity` - The preferred identity from the ssh agent for authentication.
- `host_key` - The public key from the remote host or the signing CA, used to verify the connection.

Additional arguments only supported by the winrm connection type:

- `https` - Set to `true` to connect using HTTPS instead of HTTP.
- `insecure` - Set to `true` to not validate the HTTPS certificate chain.
- `use_ntlm` - Set to `true` to use NTLM authentication, rather than default (basic authentication), removing the requirement for basic authentication to be enabled within the target guest. Further reading for remote connection authentication can be found here ([https://msdn.microsoft.com/en-us/library/aa384295\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa384295(v=vs.85).aspx)).
- `cacert` - The CA certificate to validate against.

Connecting through a Bastion Host with SSH

The ssh connection also supports the following fields to facilitate connections via a bastion host (https://en.wikipedia.org/wiki/Bastion_host).

- `bastion_host` - Setting this enables the bastion Host connection. This host will be connected to first, and then the host connection will be made from there.
- `bastion_host_key` - The public key from the remote host or the signing CA, used to verify the host connection.
- `bastion_port` - The port to use connect to the bastion host. Defaults to the value of the `port` field.
- `bastion_user` - The user for the connection to the bastion host. Defaults to the value of the `user` field.
- `bastion_password` - The password we should use for the bastion host. Defaults to the value of the `password` field.
- `bastion_private_key` - The contents of an SSH key file to use for the bastion host. These can be loaded from a file on disk using the `file()` interpolation function ([/docs/configuration/interpolation.html#file_path_](#)). Defaults to the value of the `private_key` field.

File Provisioner

The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource. The file provisioner supports both ssh and winrm type connections ([/docs/provisioners/connection.html](#)).

Example usage

```
resource "aws_instance" "web" {
  # ...

  # Copies the myapp.conf file to /etc/myapp.conf
  provisioner "file" {
    source      = "conf/myapp.conf"
    destination = "/etc/myapp.conf"
  }

  # Copies the string in content into /tmp/file.log
  provisioner "file" {
    content      = "ami used: ${self.ami}"
    destination = "/tmp/file.log"
  }

  # Copies the configs.d folder to /etc/configs.d
  provisioner "file" {
    source      = "conf/configs.d"
    destination = "/etc"
  }

  # Copies all files and folders in apps/app1 to D:/IIS/webapp1
  provisioner "file" {
    source      = "apps/app1/"
    destination = "D:/IIS/webapp1"
  }
}
```

Argument Reference

The following arguments are supported:

- **source** - This is the source file or folder. It can be specified as relative to the current working directory or as an absolute path. This attribute cannot be specified with **content**.
- **content** - This is the content to copy on the destination. If destination is a file, the content will be written on that file, in case of a directory a file named `tf-file-content` is created. It's recommended to use a file as the destination. A `template_file` ([/docs/providers/template/d/file.html](#)) might be referenced in here, or any interpolation syntax. This attribute cannot be specified with **source**.
- **destination** - (Required) This is the destination path. It must be specified as an absolute path.

Directory Uploads

The file provisioner is also able to upload a complete directory to the remote machine. When uploading a directory, there are a few important things you should know.

First, when using the `ssh` connection type the destination directory must already exist. If you need to create it, use a `remote-exec` provisioner just prior to the file provisioner in order to create the directory. When using the `winrm` connection type the destination directory will be created for you if it doesn't already exist.

Next, the existence of a trailing slash on the source path will determine whether the directory name will be embedded within the destination, or whether the destination will be created. An example explains this best:

If the source is `/foo` (no trailing slash), and the destination is `/tmp`, then the contents of `/foo` on the local machine will be uploaded to `/tmp/foo` on the remote machine. The `foo` directory on the remote machine will be created by Terraform.

If the source, however, is `/foo/` (a trailing slash is present), and the destination is `/tmp`, then the contents of `/foo` will be uploaded directly into `/tmp`.

This behavior was adopted from the standard behavior of `rsync` (<https://linux.die.net/man/1/rsync>).

Note: Under the covers, `rsync` may or may not be used.

Habitat Provisioner

The habitat provisioner installs the Habitat (<https://habitat.sh>) supervisor and loads configured services. This provisioner only supports Linux targets using the ssh connection type at this time.

Requirements

The habitat provisioner has some prerequisites for specific connection types:

- For ssh type connections, we assume a few tools to be available on the remote host:
 - curl
 - tee
 - setsid - Only if using the unmanaged service type.

Without these prerequisites, your provisioning execution will fail.

Example usage

```
resource "aws_instance" "redis" {
  count = 3

  provisioner "habitat" {
    peer = "${aws_instance.redis.0.private_ip}"
    use_sudo = true
    service_type = "systemd"

    service {
      name = "core/redis"
      topology = "leader"
      user_toml = "${file("conf/redis.toml")}"
    }
  }
}
```

Argument Reference

There are 2 configuration levels, supervisor and service. Configuration placed directly within the provisioner block are supervisor configurations, and a provisioner can define zero or more services to run, and each service will have a service block within the provisioner. A service block can also contain zero or more bind blocks to create service group bindings.

Supervisor Arguments

- `version` (string) - (Optional) The Habitat version to install on the remote machine. If not specified, the latest available version is used.
- `use_sudo` (bool) - (Optional) Use sudo when executing remote commands. Required when the user specified in the

connection block is not root. (Defaults to true)

- `service_type` (string) - (Optional) Method used to run the Habitat supervisor. Valid options are unmanaged and systemd. (Defaults to systemd)
- `service_name` (string) - (Optional) The name of the Habitat supervisor service, if using an init system such as systemd. (Defaults to hab-supervisor)
- `peer` (string) - (Optional) IP or FQDN of a supervisor instance to peer with. (Defaults to none)
- `permanent_peer` (bool) - (Optional) Marks this supervisor as a permanent peer. (Defaults to false)
- `listen_gossip` (string) - (Optional) The listen address for the gossip system (Defaults to 0.0.0.0:9638)
- `listen_http` (string) - (Optional) The listen address for the HTTP gateway (Defaults to 0.0.0.0:9631)
- `ring_key` (string) - (Optional) The name of the ring key for encrypting gossip ring communication (Defaults to no encryption)
- `ring_key_content` (string) - (Optional) The key content. Only needed if using ring encryption and want the provisioner to take care of uploading and importing it. Easiest to source from a file (eg `ring_key_content = "${file("conf/foo-123456789.sym.key")}"`) (Defaults to none)
- `url` (string) - (Optional) The URL of a Builder service to download packages and receive updates from. (Defaults to <https://bldr.habitat.sh> (<https://bldr.habitat.sh>))
- `channel` (string) - (Optional) The release channel in the Builder service to use. (Defaults to stable)
- `events` (string) - (Optional) Name of the service group running a Habitat EventSrv to forward Supervisor and service event data to. (Defaults to none)
- `override_name` (string) - (Optional) The name of the Supervisor (Defaults to default)
- `organization` (string) - (Optional) The organization that the Supervisor and its subsequent services are part of. (Defaults to default)
- `builder_auth_token` (string) - (Optional) The builder authorization token when using a private origin. (Defaults to none)

Service Arguments

- `name` (string) - (Required) The Habitat package identifier of the service to run. (ie core/haproxy or core/redis/3.2.4/20171002182640)
- `binds` (array) - (Optional) An array of bind specifications. (ie `binds = ["backend:nginx.default"]`)
- `bind` - (Optional) An alternative way of declaring binds. This method can be easier to deal with when populating values from other values or variable inputs without having to do string interpolation. The following example is equivalent to `binds = ["backend:nginx.default"]`:

```
bind {  
    alias = "backend"  
    service = "nginx"  
    group = "default"  
}
```

- **topology** (string) - (Optional) Topology to start service in. Possible values `standalone` or `leader`. (Defaults to `standalone`)
- **strategy** (string) - (Optional) Update strategy to use. Possible values `at-once`, `rolling` or `none`. (Defaults to `none`)
- **user_toml** (string) - (Optional) TOML formatted user configuration for the service. Easiest to source from a file (eg `user_toml = "${file("conf/redis.toml")}"`). (Defaults to `none`)
- **channel** (string) - (Optional) The release channel in the Builder service to use. (Defaults to `stable`)
- **group** (string) - (Optional) The service group to join. (Defaults to `default`)
- **url** (string) - (Optional) The URL of a Builder service to download packages and receive updates from. (Defaults to `https://blldr.habitat.sh` (`https://blldr.habitat.sh`))
- **application** (string) - (Optional) The application name. (Defaults to `none`)
- **environment** (string) - (Optional) The environment name. (Defaults to `none`)
- **override_name** (string) - (Optional) The name for the state directory if there is more than one Supervisor running. (Defaults to `default`)
- **service_key** (string) - (Optional) The key content of a service private key, if using service group encryption. Easiest to source from a file (eg `service_key = "${file("conf/redis.default@org-123456789.box.key")}"`) (Defaults to `none`)

local-exec Provisioner

The `local-exec` provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource. See the `remote-exec` provisioner ([/docs/provisioners/remote-exec.html](#)) to run commands on the resource.

Note that even though the resource will be fully created when the provisioner is run, there is no guarantee that it will be in an operable state - for example system services such as `sshd` may not be started yet on compute resources.

Example usage

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
  }
}
```

Argument Reference

The following arguments are supported:

- `command` - (Required) This is the command to execute. It can be provided as a relative path to the current working directory or as an absolute path. It is evaluated in a shell, and can use environment variables or Terraform variables.
- `working_dir` - (Optional) If provided, specifies the working directory where `command` will be executed. It can be provided as a relative path to the current working directory or as an absolute path. The directory must exist.
- `interpreter` - (Optional) If provided, this is a list of interpreter arguments used to execute the command. The first argument is the interpreter itself. It can be provided as a relative path to the current working directory or as an absolute path. The remaining arguments are appended prior to the command. This allows building command lines of the form `"/bin/bash", "-c", "echo foo"`. If `interpreter` is unspecified, sensible defaults will be chosen based on the system OS.
- `environment` - (Optional) block of key value pairs representing the environment of the executed command. inherits the current process environment.

Interpreter Examples

```
resource "null_resource" "example1" {
  provisioner "local-exec" {
    command = "open WFH, '>completed.txt' and print WFH scalar localtime"
    interpreter = ["perl", "-e"]
  }
}
```

```
resource "null_resource" "example2" {
  provisioner "local-exec" {
    command = "Get-Date > completed.txt"
    interpreter = ["PowerShell", "-Command"]
  }
}
```

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo $FOO $BAR $BAZ >> env_vars.txt"

    environment {
      FOO = "bar"
      BAR = 1
      BAZ = "true"
    }
  }
}
```

null_resource

The `null_resource` is a resource that allows you to configure provisioners that are not directly associated with a single existing resource.

A `null_resource` behaves exactly like any other resource, so you configure provisioners (/docs/provisioners/index.html), connection details (/docs/provisioners/connection.html), and other meta-parameters in the same way you would on any other resource.

This allows fine-grained control over when provisioners run in the dependency graph.

Example usage

```
resource "aws_instance" "cluster" {
  count = 3

  # ...
}

resource "null_resource" "cluster" {
  # Changes to any instance of the cluster requires re-provisioning
  triggers {
    cluster_instance_ids = "${join(", ", aws_instance.cluster.*.id)}"
  }

  # Bootstrap script can run on any instance of the cluster
  # So we just choose the first in this case
  connection {
    host = "${element(aws_instance.cluster.*.public_ip, 0)}"
  }

  provisioner "remote-exec" {
    # Bootstrap script called with private_ip of each node in the cluster
    inline = [
      "bootstrap-cluster.sh ${join(" ", aws_instance.cluster.*.private_ip)}",
    ]
  }
}
```

Argument Reference

In addition to all the resource configuration available, `null_resource` supports the following specific configuration options:

- `triggers` - A mapping of values which should trigger a rerun of this set of provisioners. Values are meant to be interpolated references to variables or attributes of other resources.

remote-exec Provisioner

The `remote-exec` provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc. To invoke a local process, see the `local-exec` provisioner ([/docs/provisioners/local-exec.html](#)) instead. The `remote-exec` provisioner supports both `ssh` and `winrm` type connections ([/docs/provisioners/connection.html](#)).

Example usage

```
resource "aws_instance" "web" {
  # ...

  provisioner "remote-exec" {
    inline = [
      "puppet apply",
      "consul join ${aws_instance.web.private_ip}",
    ]
  }
}
```

Argument Reference

The following arguments are supported:

- `inline` - This is a list of command strings. They are executed in the order they are provided. This cannot be provided with `script` or `scripts`.
- `script` - This is a path (relative or absolute) to a local script that will be copied to the remote resource and then executed. This cannot be provided with `inline` or `scripts`.
- `scripts` - This is a list of paths (relative or absolute) to local scripts that will be copied to the remote resource and then executed. They are executed in the order they are provided. This cannot be provided with `inline` or `script`.

Script Arguments

You cannot pass any arguments to scripts using the `script` or `scripts` arguments to this provisioner. If you want to specify arguments, upload the script with the file provisioner ([/docs/provisioners/file.html](#)) and then use `inline` to call it. Example:

```
resource "aws_instance" "web" {
  # ...

  provisioner "file" {
    source      = "script.sh"
    destination = "/tmp/script.sh"
  }

  provisioner "remote-exec" {
    inline = [
      "chmod +x /tmp/script.sh",
      "/tmp/script.sh args",
    ]
  }
}
```

Salt Masterless Provisioner

Type: salt-masterless

The salt-masterless Terraform provisioner provisions machines built by Terraform using Salt (<http://saltstack.com/>) states, without connecting to a Salt master. The salt-masterless provisioner supports ssh connections ([/docs/provisioners/connection.html](#)).

Requirements

The salt-masterless provisioner has some prerequisites. curl must be available on the remote host.

Example usage

The example below is fully functional.

```
provisioner "salt-masterless" {
  "local_state_tree" = "/srv/salt"
}
```

Argument Reference

The reference of available configuration options is listed below. The only required argument is the path to your local salt state tree.

Optional:

- `bootstrap_args` (string) - Arguments to send to the bootstrap script. Usage is somewhat documented on github (<https://github.com/saltstack/salt-bootstrap>), but the script itself (<https://github.com/saltstack/salt-bootstrap/blob/develop/bootstrap-salt.sh>) has more detailed usage instructions. By default, no arguments are sent to the script.
- `disable_sudo` (boolean) - By default, the bootstrap install command is prefixed with sudo. When using a Docker builder, you will likely want to pass true since sudo is often not pre-installed.
- `remote_pillar_roots` (string) - The path to your remote pillar roots (<http://docs.saltstack.com/ref/configuration/master.html#pillar-configuration>). default: /srv/pillar. This option cannot be used with `minion_config`.
- `remote_state_tree` (string) - The path to your remote state tree (<http://docs.saltstack.com/ref/states/highstate.html#the-salt-state-tree>). default: /srv/salt. This option cannot be used with `minion_config`.
- `local_pillar_roots` (string) - The path to your local pillar roots (<http://docs.saltstack.com/ref/configuration/master.html#pillar-configuration>). This will be uploaded to the `remote_pillar_roots` on the remote.

- `local_state_tree` (string) - The path to your local state tree (<http://docs.saltstack.com/ref/states/highstate.html#the-salt-state-tree>). This will be uploaded to the `remote_state_tree` on the remote.
- `custom_state` (string) - A state to be run instead of `state.highstate`. Defaults to `state.highstate` if unspecified.
- `minion_config_file` (string) - The path to your local minion config file (<http://docs.saltstack.com/ref/configuration/minion.html>). This will be uploaded to the `/etc/salt` on the remote. This option overrides the `remote_state_tree` or `remote_pillar_roots` options.
- `skip_bootstrap` (boolean) - By default the salt provisioner runs salt bootstrap (<https://github.com/saltstack/salt-bootstrap>) to install salt. Set this to true to skip this step.
- `temp_config_dir` (string) - Where your local state tree will be copied before moving to the `/srv/salt` directory. Default is `/tmp/salt`.
- `no_exit_on_failure` (boolean) - Terraform will exit if the `salt-call` command fails. Set this option to true to ignore Salt failures.
- `log_level` (string) - Set the logging level for the `salt-call` run.
- `salt_call_args` (string) - Additional arguments to pass directly to `salt-call`. See `salt-call` (<https://docs.saltstack.com/ref/cli/salt-call.html>) documentation for more information. By default no additional arguments (besides the ones Terraform generates) are passed to `salt-call`.
- `salt_bin_dir` (string) - Path to the `salt-call` executable. Useful if it is not on the PATH.

Terraform Registry

The Terraform Registry (<https://registry.terraform.io>) is a repository of modules written by the Terraform community. The registry can help you get started with Terraform more quickly, see examples of how Terraform is written, and find pre-made modules for infrastructure components you require.

The Terraform Registry is integrated directly into Terraform to make consuming modules easy. See the usage information (</docs/registry/modules/use.html#using-modules>).

You can also publish your own modules on the Terraform Registry. You may use the public registry (<https://registry.terraform.io>) for public modules. For private modules, you can use a Private Registry (</docs/registry/private.html>), or reference repositories and other sources directly (</docs/modules/sources.html>). Some features are available only for registry modules, such as versioning and documentation generation.

Use the navigation to the left to learn more about using the registry.

HTTP API

When downloading modules from registry sources such as the public Terraform Registry (<https://registry.terraform.io>), Terraform expects the given hostname to support the following module registry protocol.

A registry module source is of the form `hostname/namespace/name/provider`, where the initial hostname portion is implied to be `registry.terraform.io/` if not specified. The public Terraform Registry is therefore the default module source.

Terraform Registry (<https://registry.terraform.io>) implements a superset of this API to allow for importing new modules, etc, but any endpoints not documented on this page are subject to change over time.

Service Discovery

The hostname portion of a module source string is first passed to the service discovery protocol ([/docs/internals/remote-service-discovery.html](#)) to determine if the given host has a module registry and, if so, the base URL for its module registry endpoints.

The service identifier for this protocol is `modules.v1`, and the declared URL should always end with a slash such that the paths shown in the following sections can be appended to it.

For example, if discovery produces the URL `https://modules.example.com/v1/` then this API would use full endpoint URLs like `https://modules.example.com/v1/{namespace}/{name}/{provider}/versions`.

Base URL

The example request URLs shown in this document are for the public Terraform Registry (<https://registry.terraform.io>), and use its API `<base_url>` of `https://registry.terraform.io/v1/modules/`. Note that although the base URL in the discovery document *may include* a trailing slash, we include a slash after the placeholder in the Paths below for clarity.

List Modules

These endpoints list modules according to some criteria.

| Method | Path | Produces |
|--------|--|-------------------------------|
| GET | <code><base_url></code> | <code>application/json</code> |
| GET | <code><base_url>/:namespace</code> | <code>application/json</code> |

Parameters

- `namespace` (string: `<optional>`) - Restricts listing to modules published by this user or organization. This is optionally specified as part of the URL path.

Query Parameters

- `offset, limit (int: <optional>)` - See Pagination for details.
- `provider (string: <optional>)` - Limits modules to a specific provider.
- `verified (bool: <optional>)` - If true, limits results to only verified modules. Any other value including none returns all modules *including* verified ones.

Sample Request

```
$ curl 'https://registry.terraform.io/v1/modules?limit=2&verified=true'
```

Sample Response

```
{
  "meta": {
    "limit": 2,
    "current_offset": 0,
    "next_offset": 2,
    "next_url": "/v1/modules?limit=2&offset=2&verified=true"
  },
  "modules": [
    {
      "id": "GoogleCloudPlatform/lb-http/google/1.0.4",
      "owner": "",
      "namespace": "GoogleCloudPlatform",
      "name": "lb-http",
      "version": "1.0.4",
      "provider": "google",
      "description": "Modular Global HTTP Load Balancer for GCE using forwarding rules.",
      "source": "https://github.com/GoogleCloudPlatform/terraform-google-lb-http",
      "published_at": "2017-10-17T01:22:17.792066Z",
      "downloads": 213,
      "verified": true
    },
    {
      "id": "terraform-aws-modules/vpc/aws/1.5.1",
      "owner": "",
      "namespace": "terraform-aws-modules",
      "name": "vpc",
      "version": "1.5.1",
      "provider": "aws",
      "description": "Terraform module which creates VPC resources on AWS",
      "source": "https://github.com/terraform-aws-modules/terraform-aws-vpc",
      "published_at": "2017-11-23T10:48:09.400166Z",
      "downloads": 29714,
      "verified": true
    }
  ]
}
```

Search Modules

This endpoint allows searching modules.

| Method | Path | Produces |
|--------|-------------------|------------------|
| GET | <base_url>/search | application/json |

Query Parameters

- `q` (string: <required>) - The search string. Search syntax understood depends on registry implementation. The public registry supports basic keyword or phrase searches.
- `offset, limit` (int: <optional>) - See Pagination for details.
- `provider` (string: <optional>) - Limits results to a specific provider.
- `namespace` (string: <optional>) - Limits results to a specific namespace.
- `verified` (bool: <optional>) - If true, limits results to only verified modules. Any other value including none returns all modules *including* verified ones.

Sample Request

```
$ curl 'https://registry.terraform.io/v1/modules/search?q=network&limit=2'
```

Sample Response

```
{
  "meta": {
    "limit": 2,
    "current_offset": 0,
    "next_offset": 2,
    "next_url": "/v1/modules/search?limit=2&offset=2&q=network"
  },
  "modules": [
    {
      "id": "zoitech/network/aws/0.0.3",
      "owner": "",
      "namespace": "zoitech",
      "name": "network",
      "version": "0.0.3",
      "provider": "aws",
      "description": "This module is intended to be used for configuring an AWS network.",
      "source": "https://github.com/zoitech/terraform-aws-network",
      "published_at": "2017-11-23T15:12:06.620059Z",
      "downloads": 39,
      "verified": false
    },
    {
      "id": "Azure/network/azurerm/1.1.1",
      "owner": "",
      "namespace": "Azure",
      "name": "network",
      "version": "1.1.1",
      "provider": "azurerm",
      "description": "Terraform Azure RM Module for Network",
      "source": "https://github.com/Azure/terraform-azurerm-network",
      "published_at": "2017-11-22T17:15:34.325436Z",
      "downloads": 1033,
      "verified": true
    }
  ]
}
```

List Available Versions for a Specific Module

This is the primary endpoint for resolving module sources, returning the available versions for a given fully-qualified module.

| Method | Path | Produces |
|--------|--|------------------|
| GET | <base_url>/:namespace/:name/:provider/versions | application/json |

Parameters

- **namespace** (string: <required>) - The user or organization the module is owned by. This is required and is specified as part of the URL path.
- **name** (string: <required>) - The name of the module. This is required and is specified as part of the URL path.
- **provider** (string: <required>) - The name of the provider. This is required and is specified as part of the URL path.

Sample Request

```
$ curl https://registry.terraform.io/v1/modules/hashicorp/consul/aws/versions
```

Sample Response

The `modules` array in the response always includes the requested module as the first element. Other elements of this list, if present, are dependencies of the requested module that are provided to potentially avoid additional requests to resolve these modules.

Additional modules are not required to be provided but, when present, can be used by Terraform to optimize the module installation process.

Each returned module has an array of available versions, which Terraform matches against any version constraints given in configuration.

```
{  
  "modules": [  
    {  
      "source": "hashicorp/consul/aws",  
      "versions": [  
        {  
          "version": "0.0.1",  
          "submodules" : [  
            {  
              "path": "modules/consul-cluster",  
              "providers": [  
                {  
                  "name": "aws",  
                  "version": ""  
                }  
              ],  
              "dependencies": []  
            },  
            {  
              "path": "modules/consul-security-group-rules",  
              "providers": [  
                {  
                  "name": "aws",  
                  "version": ""  
                }  
              ],  
              "dependencies": []  
            },  
            {  
              "providers": [  
                {  
                  "name": "aws",  
                  "version": ""  
                }  
              ],  
              "dependencies": [],  
              "path": "modules/consul-iam-policies"  
            }  
          ],  
          "root": {  
            "dependencies": [],  
            "providers": [  
              {  
                "name": "template",  
                "version": ""  
              },  
              {  
                "name": "aws",  
                "version": ""  
              }  
            ]  
          }  
        ]  
      ]  
    }  
  ]  
}
```

Download Source Code for a Specific Module Version

This endpoint downloads the specified version of a module for a single provider.

A successful response has no body, and includes the location from which the module version's source can be downloaded in the X-Terraform-Get header. Note that this string may contain special syntax interpreted by Terraform via go-getter (<https://github.com/hashicorp/go-getter>). See the go-getter documentation (<https://github.com/hashicorp/go-getter#url-format>) for details.

The value of X-Terraform-Get may instead be a relative URL, indicated by beginning with /, ./ or ../, in which case it is resolved relative to the full URL of the download endpoint.

| Method | Path | Produces |
|--------|---|------------------|
| GET | <base_url>/:namespace/:name/:provider/:version/download | application/json |

Parameters

- `namespace` (string: <required>) - The user the module is owned by. This is required and is specified as part of the URL path.
- `name` (string: <required>) - The name of the module. This is required and is specified as part of the URL path.
- `provider` (string: <required>) - The name of the provider. This is required and is specified as part of the URL path.
- `version` (string: <required>) - The version of the module. This is required and is specified as part of the URL path.

Sample Request

```
$ curl -i \
  https://registry.terraform.io/v1/modules/hashicorp/consul/aws/0.0.1/download
```

Sample Response

```
HTTP/1.1 204 No Content
Content-Length: 0
X-Terraform-Get: https://api.github.com/repos/hashicorp/terraform-aws-consul/tarball/v0.0.1//?archive=ta
r.gz
```

List Latest Version of Module for All Providers

This endpoint returns the latest version of each provider for a module.

| Method | Path | Produces |
|--------|-----------------------------|------------------|
| GET | <base_url>/:namespace/:name | application/json |

Parameters

- `namespace` (`string: <required>`) - The user or organization the module is owned by. This is required and is specified as part of the URL path.
- `name` (`string: <required>`) - The name of the module. This is required and is specified as part of the URL path.

Query Parameters

- `offset, limit` (`int: <optional>`) - See Pagination for details.

Sample Request

```
$ curl \
  https://registry.terraform.io/v1/modules/hashicorp/consul
```

Sample Response

```
{
  "meta": {
    "limit": 15,
    "current_offset": 0
  },
  "modules": [
    {
      "id": "hashicorp/consul/azurerm/0.0.1",
      "owner": "gruntwork-team",
      "namespace": "hashicorp",
      "name": "consul",
      "version": "0.0.1",
      "provider": "azurerm",
      "description": "A Terraform Module for how to run Consul on AzureRM using Terraform and Packer",
      "source": "https://github.com/hashicorp/terraform-azurerm-consul",
      "published_at": "2017-09-14T23:22:59.923047Z",
      "downloads": 100,
      "verified": false
    },
    {
      "id": "hashicorp/consul/aws/0.0.1",
      "owner": "gruntwork-team",
      "namespace": "hashicorp",
      "name": "consul",
      "version": "0.0.1",
      "provider": "aws",
      "description": "A Terraform Module for how to run Consul on AWS using Terraform and Packer",
      "source": "https://github.com/hashicorp/terraform-aws-consul",
      "published_at": "2017-09-14T23:22:44.793647Z",
      "downloads": 113,
      "verified": false
    }
  ]
}
```

Latest Version for a Specific Module Provider

This endpoint returns the latest version of a module for a single provider.

| Method | Path | Produces |
|--------|---------------------------------------|------------------|
| GET | <base_url>/:namespace/:name/:provider | application/json |

Parameters

- `namespace` (string: <required>) - The user the module is owned by. This is required and is specified as part of the URL path.
- `name` (string: <required>) - The name of the module. This is required and is specified as part of the URL path.
- `provider` (string: <required>) - The name of the provider. This is required and is specified as part of the URL path.

Sample Request

```
$ curl \
  https://registry.terraform.io/v1/modules/hashicorp/consul/aws
```

Sample Response

Note this response has some fields trimmed for clarity.

```
{
  "id": "hashicorp/consul/aws/0.0.1",
  "owner": "gruntwork-team",
  "namespace": "hashicorp",
  "name": "consul",
  "version": "0.0.1",
  "provider": "aws",
  "description": "A Terraform Module for how to run Consul on AWS using Terraform and Packer",
  "source": "https://github.com/hashicorp/terraform-aws-consul",
  "published_at": "2017-09-14T23:22:44.793647Z",
  "downloads": 113,
  "verified": false,
  "root": {
    "path": "",
    "readme": "# Consul AWS Module\n\nThis repo contains a Module for how to deploy a [Consul]...",
    "empty": false,
    "inputs": [
      {
        "name": "ami_id",
        "description": "The ID of the AMI to run in the cluster. ...",
        "default": "\"\""
      },
      {
        "name": "aws_region",
        "description": "The AWS region to deploy into (e.g. us-east-1).",
        "default": "\"us-east-1\""
      }
    ]
  }
}
```

```

        }
    ],
    "outputs": [
    {
        "name": "num_servers",
        "description": ""
    },
    {
        "name": "asg_name_servers",
        "description": ""
    }
],
"dependencies": [],
"resources": []
},
"submodules": [
{
    "path": "modules/consul-cluster",
    "readme": "# Consul Cluster\n\nThis folder contains a [Terraform](https://www.terraform.io/) ...",
    "empty": false,
    "inputs": [
    {
        "name": "cluster_name",
        "description": "The name of the Consul cluster (e.g. consul-stage). This variable is used to namespace all resources created by this module.",
        "default": ""
    },
    {
        "name": "ami_id",
        "description": "The ID of the AMI to run in this cluster. Should be an AMI that had Consul installed and configured by the install-consul module.",
        "default": ""
    }
],
"outputs": [
{
        "name": "asg_name",
        "description": ""
    },
{
        "name": "cluster_size",
        "description": ""
    }
],
"dependencies": [],
"resources": [
{
        "name": "autoscaling_group",
        "type": "aws_autoscaling_group"
    },
{
        "name": "launch_configuration",
        "type": "aws_launch_configuration"
    }
]
}
],
"providers": [
    "aws",
    "azurerm"
],
"versions": [
    "0.0.1"
]
}
}

```

Get a Specific Module

This endpoint returns the specified version of a module for a single provider.

| Method | Path | Produces |
|--------|--|------------------|
| GET | <base_url>/:namespace/:name/:provider/:version | application/json |

Parameters

- `namespace` (string: <required>) - The user the module is owned by. This is required and is specified as part of the URL path.
- `name` (string: <required>) - The name of the module. This is required and is specified as part of the URL path.
- `provider` (string: <required>) - The name of the provider. This is required and is specified as part of the URL path.
- `version` (string: <required>) - The version of the module. This is required and is specified as part of the URL path.

Sample Request

```
$ curl \
https://registry.terraform.io/v1/modules/hashicorp/consul/aws/0.0.1
```

Sample Response

Note this response has some fields trimmed for clarity.

```
{
  "id": "hashicorp/consul/aws/0.0.1",
  "owner": "gruntwork-team",
  "namespace": "hashicorp",
  "name": "consul",
  "version": "0.0.1",
  "provider": "aws",
  "description": "A Terraform Module for how to run Consul on AWS using Terraform and Packer",
  "source": "https://github.com/hashicorp/terraform-aws-consul",
  "published_at": "2017-09-14T23:22:44.793647Z",
  "downloads": 113,
  "verified": false,
  "root": {
    "path": "",
    "readme": "# Consul AWS Module\n\nThis repo contains a Module for how to deploy a [Consul]...",
    "empty": false,
    "inputs": [
      {
        "name": "ami_id",
        "description": "The ID of the AMI to run in the cluster. ...",
        "default": "\\"\\\""
      }
    ]
  }
}
```

```

    "default": "\\"},
  },
  {
    "name": "aws_region",
    "description": "The AWS region to deploy into (e.g. us-east-1).",
    "default": "\"us-east-1\""
  }
],
"outputs": [
  {
    "name": "num_servers",
    "description": ""
  },
  {
    "name": "asg_name_servers",
    "description": ""
  }
],
"dependencies": [],
"resources": []
},
"submodules": [
  {
    "path": "modules/consul-cluster",
    "readme": "# Consul Cluster\n\nThis folder contains a [Terraform](https://www.terraform.io/) ...",
    "empty": false,
    "inputs": [
      {
        "name": "cluster_name",
        "description": "The name of the Consul cluster (e.g. consul-stage). This variable is used to namespace all resources created by this module.",
        "default": ""
      },
      {
        "name": "ami_id",
        "description": "The ID of the AMI to run in this cluster. Should be an AMI that had Consul installed and configured by the install-consul module.",
        "default": ""
      }
    ],
    "outputs": [
      {
        "name": "asg_name",
        "description": ""
      },
      {
        "name": "cluster_size",
        "description": ""
      }
    ],
    "dependencies": [],
    "resources": [
      {
        "name": "autoscaling_group",
        "type": "aws_autoscaling_group"
      },
      {
        "name": "launch_configuration",
        "type": "aws_launch_configuration"
      }
    ]
  }
],
"providers": [
  "aws",
  "azurerm"
]
}

```

```
  ],
  "versions": [
    "0.0.1"
  ]
}
```

Download the Latest Version of a Module

This endpoint downloads the latest version of a module for a single provider.

It returns a 302 redirect whose Location header redirects the client to the download endpoint (above) for the latest version.

| Method | Path | Produces |
|--------|--|------------------|
| GET | <base_url>/:namespace/:name/:provider/download | application/json |

Parameters

- `namespace` (string: <required>) - The user the module is owned by. This is required and is specified as part of the URL path.
- `name` (string: <required>) - The name of the module. This is required and is specified as part of the URL path.
- `provider` (string: <required>) - The name of the provider. This is required and is specified as part of the URL path.

Sample Request

```
$ curl -i \
  https://registry.terraform.io/v1/modules/hashicorp/consul/aws/download
```

Sample Response

```
HTTP/1.1 302 Found
Location: /v1/modules/hashicorp/consul/aws/0.0.1/download
Content-Length: 70
Content-Type: text/html; charset=utf-8

<a href="/v1/modules/hashicorp/consul/aws/0.0.1/download">Found</a>.
```

HTTP Status Codes

The API follows regular HTTP status semantics. To make implementing a complete client easier, some details on our policy and potential future status codes are listed below. A robust client should consider how to handle all of the following.

- **Success:** Return status is 200 on success with a body or 204 if there is no body data to return.
- **Redirects:** Moved or aliased endpoints redirect with a 301. Endpoints redirecting to the latest version of a module may redirect with 302 or 307 to indicate that they logically point to different resources over time.
- **Client Errors:** Invalid requests will receive the relevant 4xx status. Except where noted below, the request should not be retried.
- **Rate Limiting:** Clients placing excessive load on the service might be rate-limited and receive a 429 code. This should be interpreted as a sign to slow down, and wait some time before retrying the request.
- **Service Errors:** The usual 5xx errors will be returned for service failures. In all cases it is safe to retry the request after receiving a 5xx response.
- **Load Shedding:** A 503 response indicates that the service is under load and can't process your request immediately. As with other 5xx errors you may retry after some delay, although clients should consider being more lenient with retry schedule in this case.

Error Responses

When a 4xx or 5xx status code is returned. The response payload will look like the following example:

```
{  
  "errors": [  
    "something bad happened"  
  ]  
}
```

The `errors` key is a list containing one or more strings where each string describes an error that has occurred.

Note that it is possible that some 5xx errors might result in a response that is not in JSON format above due to being returned by an intermediate proxy.

Pagination

Endpoints that return lists of results use a common pagination format.

They accept positive integer query variables `offset` and `limit` which have the usual SQL-like semantics. Each endpoint will have a sane default limit and a default offset of 0. Each endpoint will also apply a sane maximum limit, requesting more results will just result in the maximum limit being used.

The response for a paginated result set will look like:

```
{  
  "meta": {  
    "limit": 15,  
    "current_offset": 15,  
    "next_offset": 30,  
    "prev_offset": 0,  
  },  
  "<object name>": []  
}
```

Note that: - `next_offset` will only be present if there are more results available. - `prev_offset` will only be present if not at `offset = 0`. - `limit` is the actual limit that was applied, it may be lower than the requested limit param. - The key for the result array varies based on the endpoint and will be the type of result pluralized, for example `modules`.

Publishing Modules

Anyone can publish and share modules on the Terraform Registry (<https://registry.terraform.io>).

Published modules support versioning, automatically generate documentation, allow browsing version histories, show examples and READMEs, and more. We recommend publishing reusable modules to a registry.

Public modules are managed via Git and GitHub. Publishing a module takes only a few minutes. Once a module is published, you can release a new version of a module by simply pushing a properly formed Git tag.

The registry extracts information about the module from the module's source. The module name, provider, documentation, inputs/outputs, and dependencies are all parsed and available via the UI or API, as well as the same information for any submodules or examples in the module's source repository.

Requirements

The list below contains all the requirements for publishing a module. Meeting the requirements for publishing a module is extremely easy. The list may appear long only to ensure we're detailed, but adhering to the requirements should happen naturally.

- **GitHub.** The module must be on GitHub and must be a public repo. This is only a requirement for the public registry (<https://registry.terraform.io>). If you're using a private registry, you may ignore this requirement.
- **Named** `terraform-<PROVIDER>-<NAME>`. Module repositories must use this three-part name format, where `<NAME>` reflects the type of infrastructure the module manages and `<PROVIDER>` is the main provider where it creates that infrastructure. The `<NAME>` segment can contain additional hyphens. Examples: `terraform-google-vault` or `terraform-aws-ec2-instance`.
- **Repository description.** The GitHub repository description is used to populate the short description of the module. This should be a simple one sentence description of the module.
- **Standard module structure.** The module must adhere to the standard module structure ([/docs/modules/create.html#standard-module-structure](#)). This allows the registry to inspect your module and generate documentation, track resource usage, parse submodules and examples, and more.
- **x.y.z tags for releases.** The registry uses tags to identify module versions. Release tag names must be a semantic version (<http://semver.org>), which can optionally be prefixed with a v. For example, `v1.0.4` and `0.9.2`. To publish a module initially, at least one release tag must be present. Tags that don't look like version numbers are ignored.

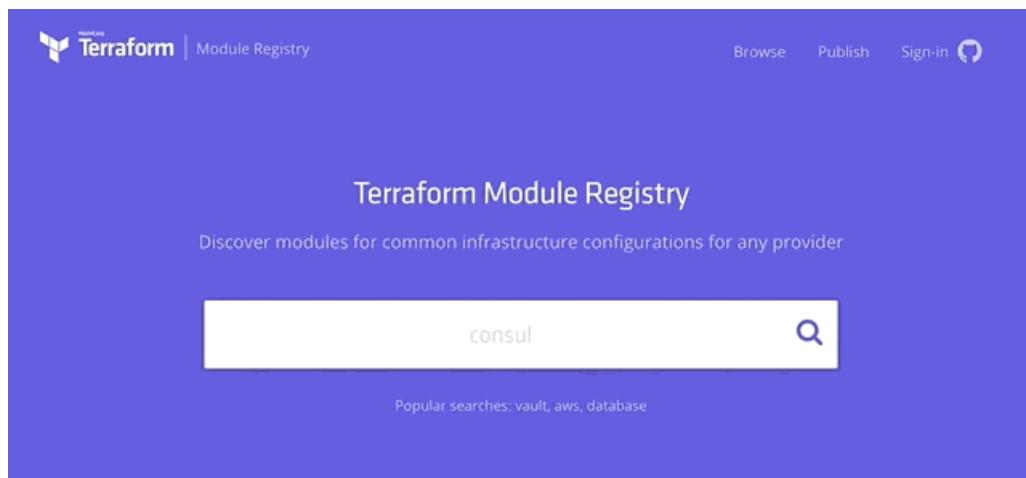
Publishing a Public Module

With the requirements met, you can publish a public module by going to the Terraform Registry (<https://registry.terraform.io>) and clicking the "Upload" link in the top navigation.

If you're not signed in, this will ask you to connect with GitHub. We only ask for access to public repositories, since the public registry may only publish public modules. We require access to hooks so we can register a webhook with your repository. We require access to your email address so that we can email you alerts about your module. We will not spam you.

The upload page will list your available repositories, filtered to those that match the naming convention described above. This is shown in the screenshot below. Select the repository of the module you want to add and click "Publish Module."

In a few seconds, your module will be created.



Releasing New Versions

The Terraform Registry uses tags to detect releases.

Tag names must be a valid semantic version (<http://semver.org>), optionally prefixed with a v. Example of valid tags are: v1.0.1 and 0.9.4. To publish a new module, you must already have at least one tag created.

To release a new version, create and push a new tag with the proper format. The webhook will notify the registry of the new version and it will appear on the registry usually in less than a minute.

If your version doesn't appear properly, you may force a sync with GitHub by viewing your module on the registry and clicking "Force GitHub Sync" under the "Manage Module" dropdown. This process may take a few minutes. Please only do this if you do not see the version appear, since it will cause the registry to resync *all versions* of your module.

Finding and Using Modules

The Terraform Registry (<https://registry.terraform.io>) makes it simple to find and use modules.

Finding Modules

Every page on the registry has a search field for finding modules. Enter any type of module you're looking for (examples: "vault", "vpc", "database") and resulting modules will be listed. The search query will look at module name, provider, and description to match your search terms. On the results page, filters can be used further refine search results.

By default, only verified modules (</docs/registry/modules/verified.html>) are shown in search results. Verified modules are reviewed by HashiCorp to ensure stability and compatibility. By using the filters, you can view unverified modules as well.

Using Modules

The Terraform Registry is integrated directly into Terraform. This makes it easy to reference any module in the registry. The syntax for referencing a registry module is <NAMESPACE>/<NAME>/<PROVIDER>. For example: hashicorp/consul/aws.

Note: Module registry integration was added in Terraform v0.10.6, and full versioning support in v0.11.0.

When viewing a module on the registry on a tablet or desktop, usage instructions are shown on the right side. You can copy and paste this to get started with any module. Some modules have required inputs you must set before being able to use the module.

```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

The `terraform init` command will download and cache any modules referenced by a configuration.

Private Registry Module Sources

You can also use modules from a private registry, like the one provided by Terraform Enterprise. Private registry modules have source strings of the form <HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>. This is the same format as the public registry, but with an added hostname prefix.

```
module "vpc" {
  source = "app.terraform.io/example_corp/vpc/aws"
  version = "0.9.3"
}
```

Depending on the registry you're using, you might also need to configure credentials to access modules. See your registry's documentation for details. Terraform Enterprise's private registry is documented here.

(</docs/enterprise/registry/index.html>)

Private registry module sources are supported in Terraform v0.11.0 and newer.

Module Versions

Each module in the registry is versioned. These versions syntactically must follow semantic versioning (<http://semver.org/>). In addition to pure syntax, we encourage all modules to follow the full guidelines of semantic versioning.

Terraform since version 0.11 will resolve any provided module version constraints (/docs/modules/usage.html#module-versions) and using them is highly recommended to avoid pulling in breaking changes.

Terraform versions after 0.10.6 but before 0.11 have partial support for the registry protocol, but always download the latest version instead of honoring version constraints.

Verified Modules

Verified modules are reviewed by HashiCorp and actively maintained by contributors to stay up-to-date and compatible with both Terraform and their respective providers.

The blue verification badge appears next to modules that are verified.



A screenshot of a Terraform module page. At the top left is a blue cloud icon with a gear inside. To its right is the word "consul" in bold black font, followed by a small blue hexagon containing a white checkmark. Below this is the text "AZURERM". In the top right corner is a button labeled "Version 0.0.1 ▾". The main description below the title reads: "A Terraform Module for how to run Consul on AzureRM using Terraform and Packer". Below this, there's a horizontal line. Underneath the line, the following information is listed: "Published September 15, 2017 by [hashicorp](#)", "Module managed by [gruntwork-team](#)", "Provisions in the last month: 100", and "Source: <https://github.com/hashicorp/terraform-azurerm-consul> (report an issue)".

If a module is verified, it is promised to be actively maintained and of high quality. It isn't indicative of flexibility or feature support; very simple modules can be verified just because they're great examples of modules. Likewise, an unverified module could be extremely high quality and actively maintained. An unverified module shouldn't be assumed to be poor quality, it only means it hasn't been created by a HashiCorp partner.

Module verification is currently a manual process restricted to a small group of trusted HashiCorp partners. In the coming months, we'll be expanding verification to enable the broader community to verify their modules.

When using registry modules (</docs/registry/modules/use.html>), there is no difference between a verified and unverified module; they are used the same way.

Private Registries

The registry at registry.terraform.io (<https://registry.terraform.io>) only hosts public modules, but most organizations have some modules that can't, shouldn't, or don't need to be public.

You can load private modules directly from version control and other sources (/docs/modules/sources.html), but those sources don't support version constraints (/docs/modules/usage.html#module-versions) or a browsable marketplace of modules, both of which are important for enabling a producers-and-consumers content model in a large organization.

If your organization is specialized enough that teams frequently use modules created by other teams, you will benefit from a private module registry.

Terraform Enterprise's Private Registry

Terraform Enterprise (<https://www.hashicorp.com/products/terraform>) (TFE) includes a private module registry, available at both Pro and Premium tiers.

It uses the same VCS-backed tagged release workflow as the Terraform Registry, but imports modules from your private VCS repos (on any of TFE's supported VCS providers) instead of requiring public GitHub repos. You can seamlessly reference private modules in your Terraform configurations (just include a hostname in the module source), and TFE's UI provides a searchable marketplace of private modules to help your users find the code they need.

Terraform Enterprise's private module registry is documented here. (/docs/enterprise/registry/index.html)

Other Private Registries

Terraform can use versioned modules from any service that implements the registry API (/docs/registry/api.html). The Terraform open source project does not provide a server implementation, but we welcome community members to create their own private registries by following the published protocol.

Command: state list

The `terraform state list` command is used to list resources within a Terraform state (</docs/state/index.html>).

Usage

Usage: `terraform state list [options] [address...]`

The command will list all resources in the state file matching the given addresses (if any). If no addresses are given, all resources are listed.

The resources listed are sorted according to module depth order followed by alphabetical. This means that resources that are in your immediate configuration are listed first, and resources that are more deeply nested within modules are listed last.

For complex infrastructures, the state can contain thousands of resources. To filter these, provide one or more patterns to the command. Patterns are in resource addressing format (</docs/commands/state/addressing.html>).

The command-line flags are all optional. The list of available flags are:

- `-state=path` - Path to the state file. Defaults to "terraform.tfstate". Ignored when remote state (</docs/state/remote.html>) is used.
- `-id=id` - ID of resources to show. Ignored when unset.

Example: All Resources

This example will list all resources, including modules:

```
$ terraform state list
aws_instance.foo
aws_instance.bar[0]
aws_instance.bar[1]
module.elb.aws_elb.main
```

Example: Filtering by Resource

This example will only list resources for the given name:

```
$ terraform state list aws_instance.bar
aws_instance.bar[0]
aws_instance.bar[1]
```

Example: Filtering by Module

This example will only list resources in the given module:

```
$ terraform state list module.elb  
module.elb.aws_elb.main
```

Example: Filtering by ID

This example will only list the resource whose ID is specified on the command line. This is useful to find where in your configuration a specific resource is located.

```
$ terraform state list -id=sg-1234abcd  
module.elb.aws_security_group.sg
```

Command: state mv

The `terraform state mv` command is used to move items in a Terraform state ([/docs/state/index.html](#)). This command can move single resources, single instances of a resource, entire modules, and more. This command can also move items to a completely different state file, enabling efficient refactoring.

Usage

Usage: `terraform state mv [options] SOURCE DESTINATION`

This command will move an item matched by the address given to the destination address. This command can also move to a destination address in a completely different state file.

This can be used for simple resource renaming, moving items to and from a module, moving entire modules, and more. And because this command can also move data to a completely new state, it can also be used for refactoring one configuration into multiple separately managed Terraform configurations.

This command will output a backup copy of the state prior to saving any changes. The backup cannot be disabled. Due to the destructive nature of this command, backups are required.

If you're moving an item to a different state file, a backup will be created for each state file.

This command requires a source and destination address of the item to move. Addresses are in resource addressing format ([/docs/commands/state/addressing.html](#)).

The command-line flags are all optional. The list of available flags are:

- `-backup=path` - Path where Terraform should write the backup for the original state. This can't be disabled. If not set, Terraform will write it to the same path as the statefile with a ".backup" extension.
- `-backup-out=path` - Path where Terraform should write the backup for the destination state. This can't be disabled. If not set, Terraform will write it to the same path as the destination state file with a backup extension. This only needs to be specified if `-state-out` is set to a different path than `-state`.
- `-state=path` - Path to the source state file to read from. Defaults to the configured backend, or "terraform.tfstate".
- `-state-out=path` - Path to the destination state file to write to. If this isn't specified the source state file will be used. This can be a new or existing path.

Example: Rename a Resource

The example below renames a single resource:

```
$ terraform state mv aws_instance.foo aws_instance.bar
```

Example: Move a Resource Into a Module

The example below moves a resource into a module. The module will be created if it doesn't exist.

```
$ terraform state mv aws_instance.foo module.web
```

Example: Move a Module Into a Module

The example below moves a module into another module.

```
$ terraform state mv module.foo module.parent.module.foo
```

Example: Move a Module to Another State

The example below moves a module into another state file. This removes the module from the original state file and adds it to the destination. The source and destination are the same meaning we're keeping the same name.

```
$ terraform state mv -state-out=other.tfstate \  
module.web module.web
```

Command: state pull

The `terraform state pull` command is used to manually download and output the state from remote state ([/docs/state/remote.html](#)). This command also works with local state.

Usage

Usage: `terraform state pull`

This command will download the state from its current location and output the raw format to stdout.

This is useful for reading values out of state (potentially pairing this command with something like `jq` (<https://stedolan.github.io/jq/>)). It is also useful if you need to make manual modifications to state.

Command: state push

The `terraform state push` command is used to manually upload a local state file to remote state ([/docs/state/remote.html](#)). This command also works with local state.

This command should rarely be used. It is meant only as a utility in case manual intervention is necessary with the remote state.

Usage

Usage: `terraform state push [options] PATH`

This command will push the state specified by PATH to the currently configured backend ([/docs/backends](#)).

If PATH is "--" then the state data to push is read from stdin. This data is loaded completely into memory and verified prior to being written to the destination state.

Terraform will perform a number of safety checks to prevent you from making changes that appear to be unsafe:

- **Differing lineage:** If the "lineage" value in the state differs, Terraform will not allow you to push the state. A differing lineage suggests that the states are completely different and you may lose data.
- **Higher remote serial:** If the "serial" value in the destination state is higher than the state being pushed, Terraform will prevent the push. A higher serial suggests that data is in the destination state that isn't accounted for in the local state being pushed.

Both of these safety checks can be disabled with the `-force` flag. **This is not recommended.** If you disable the safety checks and are pushing state, the destination state will be overwritten.

Command: state rm

The `terraform state rm` command is used to remove items from the Terraform state (/docs/state/index.html). This command can remove single resources, single instances of a resource, entire modules, and more.

Usage

Usage: `terraform state rm [options] ADDRESS...`

Remove one or more items from the Terraform state.

Items removed from the Terraform state are *not physically destroyed*. Items removed from the Terraform state are only no longer managed by Terraform. For example, if you remove an AWS instance from the state, the AWS instance will continue running, but `terraform plan` will no longer see that instance.

There are various use cases for removing items from a Terraform state file. The most common is refactoring a configuration to no longer manage that resource (perhaps moving it to another Terraform configuration/state).

The state will only be saved on successful removal of all addresses. If any specific address errors for any reason (such as a syntax error), the state will not be modified at all.

This command will output a backup copy of the state prior to saving any changes. The backup cannot be disabled. Due to the destructive nature of this command, backups are required.

This command requires one or more addresses that point to a resources in the state. Addresses are in resource addressing format (/docs/commands/state/addressing.html).

The command-line flags are all optional. The list of available flags are:

- `-backup=path` - Path where Terraform should write the backup state. This can't be disabled. If not set, Terraform will write it to the same path as the statefile with a backup extension.
- `-state=path` - Path to a Terraform state file to use to look up Terraform-managed resources. By default it will use the configured backend, or the default "terraform.tfstate" if it exists.

Example: Remove a Resource

The example below removes a single resource in a module:

```
$ terraform state rm module.foo.packet_device.worker[0]
```

Example: Remove a Module

The example below removes an entire module:

```
$ terraform state rm module.foo
```

Command: state show

The `terraform state show` command is used to show the attributes of a single resource in the Terraform state ([/docs/state/index.html](#)).

Usage

Usage: `terraform state show [options] ADDRESS`

The command will show the attributes of a single resource in the state file that matches the given address.

The attributes are listed in alphabetical order (with the except of "id" which is always at the top). They are outputted in a way that is easy to parse on the command-line.

This command requires a address that points to a single resource in the state. Addresses are in resource addressing format ([/docs/commands/state/addressing.html](#)).

The command-line flags are all optional. The list of available flags are:

- `-state=path` - Path to the state file. Defaults to "terraform.tfstate". Ignored when remote state ([/docs/state/remote.html](#)) is used.

Example: Show a Resource

The example below shows a resource:

```
$ terraform state show module.foo.packet_device.worker[0]
id          = 6015bg2b-b8c4-4925-aad2-f0671d5d3b13
billing_cycle = hourly
created      = 2015-12-17T00:06:56Z
facility     = ewr1
hostname     = prod-xyz01
locked       = false
...
```

Command: taint

The `terraform taint` command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.

This command *will not* modify infrastructure, but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next plan ([/docs/commands/plan.html](#)) will show that the resource will be destroyed and recreated and the next apply ([/docs/commands/apply.html](#)) will implement this change.

Forcing the recreation of a resource is useful when you want a certain side effect of recreation that is not visible in the attributes of a resource. For example: re-running provisioners will cause the node to be different or rebooting the machine from a base image will cause new startup scripts to run.

Note that tainting a resource for recreation may affect resources that depend on the newly tainted resource. For example, a DNS resource that uses the IP address of a server may need to be modified to reflect the potentially new IP address of a tainted server. The plan command ([/docs/commands/plan.html](#)) will show this if this is the case.

Usage

Usage: `terraform taint [options] name`

The `name` argument is the name of the resource to mark as tainted. The format of this argument is `TYPE.NAME`, such as `aws_instance.foo`.

The command-line flags are all optional. The list of available flags are:

- `-allow-missing` - If specified, the command will succeed (exit code 0) even if the resource is missing. The command can still error, but only in critically erroneous cases.
- `-backup=path` - Path to the backup file. Defaults to `-state-out` with the `".backup"` extension. Disabled by setting to `".`.
- `-lock=true` - Lock the state file when locking is supported.
- `-lock-timeout=0s` - Duration to retry a state lock.
- `-module=path` - The module path where the resource to taint exists. By default this is the root path. Other modules can be specified by a period-separated list. Example: "foo" would reference the module "foo" but "foo.bar" would reference the "bar" module in the "foo" module.
- `-no-color` - Disables output with coloring
- `-state=path` - Path to read and write the state file to. Defaults to `"terraform.tfstate"`. Ignored when remote state ([/docs/state/remote.html](#)) is used.
- `-state-out=path` - Path to write updated state file. By default, the `-state` path will be used. Ignored when remote state ([/docs/state/remote.html](#)) is used.

Example: Tainting a Single Resource

This example will taint a single resource:

```
$ terraform taint aws_security_group.allow_all
The resource aws_security_group.allow_all in the module root has been marked as tainted!
```

Example: Tainting a Resource within a Module

This example will only taint a resource within a module:

```
$ terraform taint -module=couchbase aws_instance.cb_node.9
The resource aws_instance.cb_node.9 in the module root.couchbase has been marked as tainted!
```

Command: untaint

The `terraform untaint` command manually unmarks a Terraform-managed resource as tainted, restoring it as the primary instance in the state. This reverses either a manual `terraform taint` or the result of provisioners failing on a resource.

This command *will not* modify infrastructure, but does modify the state file in order to unmark a resource as tainted.

NOTE on Tainted Indexes: In certain edge cases, more than one tainted instance can be present for a single resource. When this happens, the `-index` flag is required to select which of the tainted instances to restore as primary. You can use the `terraform show` command to inspect the state and determine which index holds the instance you'd like to restore. In the vast majority of cases, there will only be one tainted instance, and the `-index` flag can be omitted.

Usage

Usage: `terraform untaint [options] name`

The `name` argument is the name of the resource to mark as untainted. The format of this argument is `TYPE.NAME`, such as `aws_instance.foo`.

The command-line flags are all optional (with the exception of `-index` in certain cases, see above note). The list of available flags are:

- `-allow-missing` - If specified, the command will succeed (exit code 0) even if the resource is missing. The command can still error, but only in critically erroneous cases.
- `-backup=path` - Path to the backup file. Defaults to `-state-out` with the `".backup"` extension. Disabled by setting to `".`.
- `-index=n` - Selects a single tainted instance when there are more than one tainted instances present in the state for a given resource. This flag is required when multiple tainted instances are present. The vast majority of the time, there is a maximum of one tainted instance per resource, so this flag can be safely omitted.
- `-lock=true` - Lock the state file when locking is supported.
- `-lock-timeout=0s` - Duration to retry a state lock.
- `-module=path` - The module path where the resource to untaint exists. By default this is the root path. Other modules can be specified by a period-separated list. Example: "foo" would reference the module "foo" but "foo.bar" would reference the "bar" module in the "foo" module.
- `-no-color` - Disables output with coloring
- `-state=path` - Path to read and write the state file to. Defaults to `"terraform.tfstate"`. Ignored when remote state (`/docs/state/remote.html`) is used.
- `-state-out=path` - Path to write updated state file. By default, the `-state` path will be used. Ignored when remote state (`/docs/state/remote.html`) is used.

Command: validate

The `terraform validate` command is used to validate the syntax of the terraform files. Terraform performs a syntax check on all the terraform files in the directory, and will display an error if any of the files doesn't validate.

This command **does not** check formatting (e.g. tabs vs spaces, newlines, comments etc.).

The following can be reported:

- invalid HCL (<https://github.com/hashicorp/hcl>) syntax (e.g. missing trailing quote or equal sign)
- invalid HCL references (e.g. variable name or attribute which doesn't exist)
- same provider declared multiple times
- same module declared multiple times
- same resource declared multiple times
- invalid module name
- interpolation used in places where it's unsupported (e.g. `variable`, `depends_on`, `module.source`, `provider`)
- missing value for a variable (none of `-var foo=...` flag, `-var-file=foo.vars` flag, `TF_VAR_foo` environment variable, `terraform.tfvars`, or default value in the configuration)

Usage

Usage: `terraform validate [options] [dir]`

By default, validate requires no flags and looks in the current directory for the configurations.

The command-line flags are all optional. The available flags are:

- `-check-variables=true` - If set to true (default), the command will check whether all required variables have been specified.
- `-no-color` - Disables output with coloring.
- `-var 'foo=bar'` - Set a variable in the Terraform configuration. This flag can be set multiple times. Variable values are interpreted as HCL ([/docs/configuration/syntax.html#HCL](#)), so list and map values can be specified via this flag.
- `-var-file=foo` - Set variables in the Terraform configuration from a variable file ([/docs/configuration/variables.html#variable-files](#)). If "terraform.tfvars" is present, it will be automatically loaded first. Any files specified by `-var-file` override any values in a "terraform.tfvars". This flag can be used multiple times.

Command: workspace

The `terraform workspace` command is used to manage workspaces ([/docs/state/workspaces.html](#)).

This command is a container for further subcommands. These subcommands are listed in the navigation bar.

Usage

Usage: `terraform workspace <subcommand> [options] [args]`

Please choose a subcommand from the navigation for more information.

Command: workspace

The `terraform workspace` command is used to manage workspaces ([/docs/state/workspaces.html](#)).

This command is a container for further subcommands. These subcommands are listed in the navigation bar.

Usage

Usage: `terraform workspace <subcommand> [options] [args]`

Please choose a subcommand from the navigation for more information.

Command: workspace delete

The `terraform workspace delete` command is used to delete an existing workspace.

Usage

Usage: `terraform workspace delete [NAME]`

This command will delete the specified workspace.

To delete an workspace, it must already exist, it must have an empty state, and it must not be your current workspace. If the workspace state is not empty, Terraform will not allow you to delete it unless the `-force` flag is specified.

If you delete a workspace with a non-empty state (via `-force`), then resources may become "dangling". These are resources that physically exist but that Terraform can no longer manage. This is sometimes preferred: you want Terraform to stop managing resources so they can be managed some other way. Most of the time, however, this is not intended and so Terraform protects you from getting into this situation.

The command-line flags are all optional. The only supported flag is:

- `-force` - Delete the workspace even if its state is not empty. Defaults to false.

Example

```
$ terraform workspace delete example
Deleted workspace "example".
```

Command: workspace list

The `terraform workspace list` command is used to list all existing workspaces.

Usage

Usage: `terraform workspace list`

The command will list all existing workspaces. The current workspace is indicated using an asterisk (*) marker.

Example

```
$ terraform workspace list
  default
* development
  jsmith-test
```

Command: workspace new

The `terraform workspace new` command is used to create a new workspace.

Usage

Usage: `terraform workspace new [NAME]`

This command will create a new workspace with the given name. A workspace with this name must not already exist.

If the `-state` flag is given, the state specified by the given path will be copied to initialize the state for this new workspace.

The command-line flags are all optional. The only supported flag is:

- `-state=path` - Path to a state file to initialize the state of this environment.

Example: Create

```
$ terraform workspace new example
Created and switched to workspace "example"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Example: Create from State

To create a new workspace from a pre-existing local state file:

```
$ terraform workspace new -state=old.terraform.tfstate example
Created and switched to workspace "example"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Command: workspace select

The `terraform workspace select` command is used to choose a different workspace to use for further operations.

Usage

Usage: `terraform workspace select [NAME]`

This command will select another workspace. The named workspace must already exist.

Example

```
$ terraform workspace list
  default
* development
  jsmith-test

$ terraform workspace select default
Switched to workspace "default".
```

Configuration

Terraform uses text files to describe infrastructure and to set variables. These text files are called Terraform *configurations* and end in `.tf`. This section talks about the format of these files as well as how they're loaded.

The format of the configuration files are able to be in two formats: Terraform format and JSON. The Terraform format is more human-readable, supports comments, and is the generally recommended format for most Terraform files. The JSON format is meant for machines to create, modify, and update, but can also be done by Terraform operators if you prefer. Terraform format ends in `.tf` and JSON format ends in `.tf.json`.

Click a sub-section in the navigation to the left to learn more about Terraform configuration.

Data Source Configuration

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to build on information defined outside of Terraform, or defined by another separate Terraform configuration.

Providers (/docs/configuration/providers.html) are responsible in Terraform for defining and implementing data sources. Whereas a resource (/docs/configuration/resources.html) causes Terraform to create and manage a new infrastructure component, data sources present read-only views into pre-existing data, or they compute new values on the fly within Terraform itself.

For example, a data source may retrieve artifact information from Terraform Enterprise, configuration information from Consul, or look up a pre-existing AWS resource by filtering on its attributes and tags.

Every data source in Terraform is mapped to a provider based on longest-prefix matching. For example the `aws_ami` data source would map to the `aws` provider (if that exists).

This page assumes you're familiar with the configuration syntax (/docs/configuration/syntax.html) already.

Example

A data source configuration looks like the following:

```
# Find the latest available AMI that is tagged with Component = web
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

Description

The data block creates a data instance of the given TYPE (first parameter) and NAME (second parameter). The combination of the type and name must be unique.

Within the block (the `{ }`) is configuration for the data instance. The configuration is dependent on the type, and is documented for each data source in the providers section (/docs/providers/index.html).

Each data instance will export one or more attributes, which can be interpolated into other resources using variables of the form `data.TYPES.NAME.ATTR`. For example:

```
resource "aws_instance" "web" {
  ami           = "${data.aws_ami.web.id}"
  instance_type = "t1.micro"
}
```

Meta-parameters

As data sources are essentially a read only subset of resources they also support the same meta-parameters (<https://www.terraform.io/docs/configuration/resources.html#meta-parameters>) of resources except for the `lifecycle` configuration block (<https://www.terraform.io/docs/configuration/resources.html#lifecycle>).

Multiple Provider Instances

Similarly to resources (/docs/configuration/resources.html), the `provider` meta-parameter can be used where a configuration has multiple aliased instances of the same provider:

```
data "aws_ami" "web" {
  provider = "aws.west"

  # ...
}
```

See the "Multiple Provider Instances" (/docs/configuration/resources.html#multiple-provider-instances) documentation for resources for more information.

Data Source Lifecycle

If the arguments of a data instance contain no references to computed values, such as attributes of resources that have not yet been created, then the data instance will be read and its state updated during Terraform's "refresh" phase, which by default runs prior to creating a plan. This ensures that the retrieved data is available for use during planning and the diff will show the real values obtained.

Data instance arguments may refer to computed values, in which case the attributes of the instance itself cannot be resolved until all of its arguments are defined. In this case, refreshing the data instance will be deferred until the "apply" phase, and all interpolations of the data instance attributes will show as "computed" in the plan since the values are not yet known.

Environment Variables

TF_LOG

If set to any value, enables detailed logs to appear on stderr which is useful for debugging. For example:

```
export TF_LOG=TRACE
```

To disable, either unset it or set it to empty. When unset, logging will default to stderr. For example:

```
export TF_LOG=
```

For more on debugging Terraform, check out the section on [Debugging \(/docs/internals/debugging.html\)](#).

TF_LOG_PATH

This specifies where the log should persist its output to. Note that even when TF_LOG_PATH is set, TF_LOG must be set in order for any logging to be enabled. For example, to always write the log to the directory you're currently running terraform from:

```
export TF_LOG_PATH=./terraform.log
```

For more on debugging Terraform, check out the section on [Debugging \(/docs/internals/debugging.html\)](#).

TF_INPUT

If set to "false" or "0", causes terraform commands to behave as if the `-input=false` flag was specified. This is used when you want to disable prompts for variables that haven't had their values specified. For example:

```
export TF_INPUT=0
```

TF_MODULE_DEPTH

When given a value, causes terraform commands to behave as if the `-module-depth=VALUE` flag was specified. By setting this to 0, for example, you enable commands such as `plan` ([/docs/commands/plan.html](#)) and `graph` ([/docs/commands/graph.html](#)) to display more compressed information.

```
export TF_MODULE_DEPTH=0
```

For more information regarding modules, check out the section on [Using Modules \(/docs/modules/usage.html\)](#).

TF_VAR_name

Environment variables can be used to set variables. The environment variables must be in the format `TF_VAR_name` and this will be checked last for a value. For example:

```
export TF_VAR_region=us-west-1
export TF_VAR_ami=ami-049d8641
export TF_VAR_alist='[1,2,3]'
export TF_VAR_amap='{ foo = "bar", baz = "qux" }'
```

For more on how to use `TF_VAR_name` in context, check out the section on Variable Configuration ([/docs/configuration/variables.html](#)).

TF_CLI_ARGS and TF_CLI_ARGS_name

The value of `TF_CLI_ARGS` will specify additional arguments to the command-line. This allows easier automation in CI environments as well as modifying default behavior of Terraform on your own system.

These arguments are inserted directly *after* the subcommand (such as `plan`) and *before* any flags specified directly on the command-line. This behavior ensures that flags on the command-line take precedence over environment variables.

For example, the following command: `TF_CLI_ARGS="-input=false" terraform apply -force` is the equivalent to manually typing: `terraform apply -input=false -force`.

The flag `TF_CLI_ARGS` affects all Terraform commands. If you specify a named command in the form of `TF_CLI_ARGS_name` then it will only affect that command. As an example, to specify that only plans never refresh, you can set `TF_CLI_ARGS_plan="-refresh=false"`.

The value of the flag is parsed as if you typed it directly to the shell. Double and single quotes are allowed to capture strings and arguments will be separated by spaces otherwise.

TF_DATA_DIR

`TF_DATA_DIR` changes the location where Terraform keeps its per-working-directory data, such as the current remote backend configuration.

By default this data is written into a `.terraform` subdirectory of the current directory, but the path given in `TF_DATA_DIR` will be used instead if non-empty.

In most cases it should not be necessary to set this variable, but it may be useful to do so if e.g. the working directory is not writable.

The data directory is used to retain data that must persist from one command to the next, so it's important to have this variable set consistently throughout all of the Terraform workflow commands (starting with `terraform init`) or else Terraform may be unable to find providers, modules, and other artifacts.

TF_SKIP_REMOTE_TESTS

This can be set prior to running the unit tests to opt-out of any tests requiring remote network connectivity. The unit tests make an attempt to automatically detect when connectivity is unavailable and skip the relevant tests, but by setting this variable you can force these tests to be skipped.

```
export TF_SKIP_REMOTE_TESTS=1  
make test
```

Interpolation Syntax

Embedded within strings in Terraform, whether you're using the Terraform syntax or JSON syntax, you can interpolate other values. These interpolations are wrapped in \${}, such as \${var.foo}.

The interpolation syntax is powerful and allows you to reference variables, attributes of resources, call functions, etc.

You can perform simple math in interpolations, allowing you to write expressions such as \${count.index + 1}. And you can also use conditionals to determine a value based on some logic.

You can escape interpolation with double dollar signs: \$\$ {foo} will be rendered as a literal \${foo}.

Available Variables

There are a variety of available variable references you can use.

User string variables

Use the var. prefix followed by the variable name. For example, \${var.foo} will interpolate the foo variable value.

User map variables

The syntax is var.MAP["KEY"]. For example, \${var.amis["us-east-1"]} would get the value of the us-east-1 key within the amis map variable.

User list variables

The syntax is "\${var.LIST}". For example, "\${var.subnets}" would get the value of the subnets list, as a list. You can also return list elements by index: \${var.subnets[idx]}.

Attributes of your own resource

The syntax is self.ATTRIBUTE. For example \${self.private_ip} will interpolate that resource's private IP address.

Note: The self.ATTRIBUTE syntax is only allowed and valid within provisioners.

Attributes of other resources

The syntax is TYPE.NAME.ATTRIBUTE. For example, \${aws_instance.web.id} will interpolate the ID attribute from the aws_instance resource named web. If the resource has a count attribute set, you can access individual attributes with a zero-based index, such as \${aws_instance.web.0.id}. You can also use the splat syntax to get a list of all the attributes: \${aws_instance.web.*.id}.

Attributes of a data source

The syntax is `data.TYPE.NAME.ATTRIBUTE`. For example, `${data.aws_ami.ubuntu.id}` will interpolate the `id` attribute from the `aws_ami` data source ([/docs/configuration/data-sources.html](#)) named `ubuntu`. If the data source has a `count` attribute set, you can access individual attributes with a zero-based index, such as `${data.aws_subnet.example.0.cidr_block}`. You can also use the splat syntax to get a list of all the attributes: `${data.aws_subnet.example.*.cidr_block}`.

Outputs from a module

The syntax is `MODULE.NAME.OUTPUT`. For example `${module.foo.bar}` will interpolate the `bar` output from the `foo` module ([/docs/modules/index.html](#)).

Count information

The syntax is `count.FIELD`. For example, `${count.index}` will interpolate the current index in a multi-count resource. For more information on `count`, see the resource configuration page ([/docs/configuration/resources.html](#)).

Path information

The syntax is `path.TYPE`. `TYPE` can be `cwd`, `module`, or `root`. `cwd` will interpolate the current working directory. `module` will interpolate the path to the current module. `root` will interpolate the path of the root module. In general, you probably want the `path.module` variable.

Terraform meta information

The syntax is `terraform.FIELD`. This variable type contains metadata about the currently executing Terraform run. `FIELD` can currently only be `env` to reference the currently active state environment ([/docs/state/environments.html](#)).

Conditionals

Interpolations may contain conditionals to branch on the final value.

```
resource "aws_instance" "web" {
  subnet = "${var.env == "production" ? var.prod_subnet : var.dev_subnet}"
}
```

The conditional syntax is the well-known ternary operation:

```
CONDITION ? TRUEVAL : FALSEVAL
```

The condition can be any valid interpolation syntax, such as variable access, a function call, or even another conditional. The true and false value can also be any valid interpolation syntax. The returned types by the true and false side must be the same.

The supported operators are:

- Equality: `==` and `!=`

- Numerical comparison: `>`, `<`, `>=`, `<=`
- Boolean logic: `&&`, `||`, unary `!`

A common use case for conditionals is to enable/disable a resource by conditionally setting the count:

```
resource "aws_instance" "vpn" {
  count = "${var.something ? 1 : 0}"
}
```

In the example above, the "vpn" resource will only be included if "var.something" evaluates to true. Otherwise, the VPN resource will not be created at all.

Built-in Functions

Terraform ships with built-in functions. Functions are called with the syntax `name(arg, arg2, ...)`. For example, to read a file: `${file("path.txt")}`.

NOTE: Proper escaping is required for JSON field values containing quotes ("") such as environment values. If directly setting the JSON, they should be escaped as \" in the JSON, e.g. `"value": "I \\\"love\\\" escaped quotes"`. If using a Terraform variable value, they should be escaped as \\\\" in the variable, e.g. `value = "I \\\\"love\\\\\" escaped quotes"` in the variable and `"value": "${var.myvariable}"` in the JSON.

Supported built-in functions

The supported built-in functions are:

- `abs(float)` - Returns the absolute value of a given float. Example: `abs(1)` returns 1, and `abs(-1)` would also return 1, whereas `abs(-3.14)` would return 3.14. See also the `signum` function.
- `basename(path)` - Returns the last element of a path.
- `base64decode(string)` - Given a base64-encoded string, decodes it and returns the original string.
- `base64encode(string)` - Returns a base64-encoded representation of the given string.
- `base64gzip(string)` - Compresses the given string with gzip and then encodes the result to base64. This can be used with certain resource arguments that allow binary data to be passed with base64 encoding, since Terraform strings are required to be valid UTF-8.
- `base64sha256(string)` - Returns a base64-encoded representation of raw SHA-256 sum of the given string. **This is not equivalent** of `base64encode(sha256(string))` since `sha256()` returns hexadecimal representation.
- `base64sha512(string)` - Returns a base64-encoded representation of raw SHA-512 sum of the given string. **This is not equivalent** of `base64encode(sha512(string))` since `sha512()` returns hexadecimal representation.
- `bcrypt(password, cost)` - Returns the Blowfish encrypted hash of the string at the given cost. A default cost of 10 will be used if not provided.
- `ceil(float)` - Returns the least integer value greater than or equal to the argument.

- `chomp(string)` - Removes trailing newlines from the given string.
- `chunklist(list, size)` - Returns the list items chunked by size. Examples:
 - `chunklist(aws_subnet.foo.*.id, 1)`: will outputs `["id1"], ["id2"], ["id3"]`
 - `chunklist(var.list_of_strings, 2)`: will outputs `["id1", "id2"], ["id3", "id4"], ["id5"]`
- `cidrhost(iprange, hostnum)` - Takes an IP address range in CIDR notation and creates an IP address with the given host number. If given host number is negative, the count starts from the end of the range. For example, `cidrhost("10.0.0.0/8", 2)` returns `10.0.0.2` and `cidrhost("10.0.0.0/8", -2)` returns `10.255.255.254`.
- `cidrnetmask(iprange)` - Takes an IP address range in CIDR notation and returns the address-formatted subnet mask format that some systems expect for IPv4 interfaces. For example, `cidrnetmask("10.0.0.0/8")` returns `255.0.0.0`. Not applicable to IPv6 networks since CIDR notation is the only valid notation for IPv6.
- `cidrsubnet(iprange, newbits, netnum)` - Takes an IP address range in CIDR notation (like `10.0.0.0/8`) and extends its prefix to include an additional subnet number. For example, `cidrsubnet("10.0.0.0/8", 8, 2)` returns `10.2.0.0/16`; `cidrsubnet("2607:f298:6051:516c::/64", 8, 2)` returns `2607:f298:6051:516c:200::/72`.
- `coalesce(string1, string2, ...)` - Returns the first non-empty value from the given arguments. At least two arguments must be provided.
- `coalescelist(list1, list2, ...)` - Returns the first non-empty list from the given arguments. At least two arguments must be provided.
- `compact(list)` - Removes empty string elements from a list. This can be useful in some cases, for example when passing joined lists as module variables or when parsing module outputs. Example:
`compact(module.my_asg.load_balancer_names)`
- `concat(list1, list2, ...)` - Combines two or more lists into a single list. Example:
`concat(aws_instance.db.*.tags.Name, aws_instance.web.*.tags.Name)`
- `contains(list, element)` - Returns `true` if a list contains the given element and returns `false` otherwise. Examples:
`contains(var.list_of_strings, "an_element")`
- `dirname(path)` - Returns all but the last element of path, typically the path's directory.
- `distinct(list)` - Removes duplicate items from a list. Keeps the first occurrence of each element, and removes subsequent occurrences. This function is only valid for flat lists. Example: `distinct(var.usernames)`
- `element(list, index)` - Returns a single element from a list at the given index. If the index is greater than the number of elements, this function will wrap using a standard mod algorithm. This function only works on flat lists. Examples:
 - `element(aws_subnet.foo.*.id, count.index)`
 - `element(var.list_of_strings, 2)`
- `file(path)` - Reads the contents of a file into the string. Variables in this file are *not* interpolated. The contents of the file are read as-is. The path is interpreted relative to the working directory. Path variables can be used to reference paths relative to other base locations. For example, when using `file()` from inside a module, you generally want to make the path relative to the module base, like this: `file("${path.module}/file")`.
- `floor(float)` - Returns the greatest integer value less than or equal to the argument.
- `flatten(list of lists)` - Flattens lists of lists down to a flat list of primitive values, eliminating any nested lists

recursively. Examples:

- `flatten(data.github_user.user.*.gpg_keys)`
 - `format(format, args, ...)` - Formats a string according to the given format. The syntax for the format is standard `sprintf` syntax. Good documentation for the syntax can be found here (<https://golang.org/pkg/fmt/>). Example to zero-prefix a count, used commonly for naming servers: `format("web-%03d", count.index + 1)`.
 - `formatlist(format, args, ...)` - Formats each element of a list according to the given format, similarly to `format`, and returns a list. Non-list arguments are repeated for each list element. For example, to convert a list of DNS addresses to a list of URLs, you might use: `formatlist("https://%s:%s/", aws_instance.foo.*.public_dns, var.port)`. If multiple args are lists, and they have the same number of elements, then the formatting is applied to the elements of the lists in parallel. Example: `formatlist("instance %v has private ip %v", aws_instance.foo.*.id, aws_instance.foo.*.private_ip)`. Passing lists with different lengths to `formatlist` results in an error.
 - `indent(numspace, string)` - Prepends the specified number of spaces to all but the first line of the given multi-line string. May be useful when inserting a multi-line string into an already-indented context. The first line is not indented, to allow for the indented string to be placed after some sort of already-indented preamble. Example: `" \"items\" : ${ indent(4, "[\n \"item1\"\n]") }, "`
 - `index(list, elem)` - Finds the index of a given element in a list. This function only works on flat lists. Example: `index(aws_instance.foo.*.tags.Name, "foo-test")`
 - `join(delim, list)` - Joins the list with the delimiter for a resultant string. This function works only on flat lists.
- Examples:
- `join(",", aws_instance.foo.*.id)`
 - `join(",", var.ami_list)`
 - `jsonencode(value)` - Returns a JSON-encoded representation of the given value, which can contain arbitrarily-nested lists and maps. Note that if the value is a string then its value will be placed in quotes.
 - `keys(map)` - Returns a lexically sorted list of the map keys.
 - `length(list)` - Returns the number of members in a given list or map, or the number of characters in a given string.
 - `${length(split(", ", "a,b,c"))} = 3`
 - `${length("a,b,c")} = 5`
 - `${length(map("key", "val"))} = 1`
 - `list(items, ...)` - Returns a list consisting of the arguments to the function. This function provides a way of representing list literals in interpolation.
 - `${list("a", "b", "c")}` returns a list of "a", "b", "c".
 - `${list()}` returns an empty list.
 - `log(x, base)` - Returns the logarithm of x.
 - `lookup(map, key, [default])` - Performs a dynamic lookup into a map variable. The `map` parameter should be another variable, such as `var.amis`. If `key` does not exist in `map`, the interpolation will fail unless you specify a third argument, `default`, which should be a string value to return if no key is found in `map`. This function only works on flat maps and will return an error for maps that include nested lists or maps.

- `lower(string)` - Returns a copy of the string with all Unicode letters mapped to their lower case.
- `map(key, value, ...)` - Returns a map consisting of the key/value pairs specified as arguments. Every odd argument must be a string key, and every even argument must have the same type as the other values specified. Duplicate keys are not allowed. Examples:
 - `map("hello", "world")`
 - `map("us-east", list("a", "b", "c"), "us-west", list("b", "c", "d"))`
- `matchkeys(values, keys, searchset)` - For two lists `values` and `keys` of equal length, returns all elements from `values` where the corresponding element from `keys` exists in the `searchset` list. E.g.
`matchkeys(aws_instance.example.*.id, aws_instance.example.*.availability_zone, list("us-west-2a"))`
 will return a list of the instance IDs of the `aws_instance.example` instances in "us-west-2a". No match will result in empty list. Items of `keys` are processed sequentially, so the order of returned values is preserved.
- `max(float1, float2, ...)` - Returns the largest of the floats.
- `merge(map1, map2, ...)` - Returns the union of 2 or more maps. The maps are consumed in the order provided, and duplicate keys overwrite previous entries.
 - `${merge(map("a", "b"), map("c", "d"))}` returns `{"a": "b", "c": "d"}`
- `min(float1, float2, ...)` - Returns the smallest of the floats.
- `md5(string)` - Returns a (conventional) hexadecimal representation of the MD5 hash of the given string.
- `pathexpand(string)` - Returns a filepath string with `~` expanded to the home directory. Note: This will create a plan diff between two different hosts, unless the filepaths are the same.
- `pow(x, y)` - Returns the base `x` of exponential `y` as a float.

Example:

- `${pow(3,2)}` = 9
- `${pow(4,0)}` = 1
- `replace(string, search, replace)` - Does a search and replace on the given string. All instances of `search` are replaced with the value of `replace`. If `search` is wrapped in forward slashes, it is treated as a regular expression. If using a regular expression, `replace` can reference subcaptures in the regular expression by using `$n` where `n` is the index or name of the subcapture. If using a regular expression, the syntax conforms to the `re2` regular expression syntax (<https://github.com/google/re2/wiki/Syntax>).
- `rsadecrypt(string, key)` - Decrypts `string` using RSA. The padding scheme PKCS #1 v1.5 is used. The `string` must be base64-encoded. `key` must be an RSA private key in PEM format. You may use `file()` to load it from a file.
- `sha1(string)` - Returns a (conventional) hexadecimal representation of the SHA-1 hash of the given string. Example:
 `"${sha1("${aws_vpc.default.tags.customer}-s3-bucket")}"`
- `sha256(string)` - Returns a (conventional) hexadecimal representation of the SHA-256 hash of the given string.
 Example: `"${sha256("${aws_vpc.default.tags.customer}-s3-bucket")}"`
- `sha512(string)` - Returns a (conventional) hexadecimal representation of the SHA-512 hash of the given string.
 Example: `"${sha512("${aws_vpc.default.tags.customer}-s3-bucket")}"`

- `signum(integer)` - Returns -1 for negative numbers, 0 for 0 and 1 for positive numbers. This function is useful when you need to set a value for the first resource and a different value for the rest of the resources. Example:
`element(split(", ", var.r53_failover_policy), signum(count.index))` where the 0th index points to PRIMARY and 1st to FAILOVER
- `slice(list, from, to)` - Returns the portion of `list` between `from` (inclusive) and `to` (exclusive). Example:
`slice(var.list_of_strings, 0, length(var.list_of_strings) - 1)`
- `sort(list)` - Returns a lexicographically sorted list of the strings contained in the list passed as an argument. Sort may only be used with lists which contain only strings. Examples: `sort(aws_instance.foo.*.id)`, `sort(var.list_of_strings)`
- `split(delim, string)` - Splits the string previously created by `join` back into a list. This is useful for pushing lists through module outputs since they currently only support string values. Depending on the use, the string this is being performed within may need to be wrapped in brackets to indicate that the output is actually a list, e.g.
`a_resource_param = ["${split(", ", var.CSV_STRING)}"]`. Example: `split(", ", module.amod.server_ids)`
- `substr(string, offset, length)` - Extracts a substring from the input string. A negative offset is interpreted as being equivalent to a positive offset measured backwards from the end of the string. A length of -1 is interpreted as meaning "until the end of the string".
- `timestamp()` - Returns a UTC timestamp string in RFC 3339 format. This string will change with every invocation of the function, so in order to prevent diffs on every plan & apply, it must be used with the `ignore_changes` (`/docs/configuration/resources.html#ignore-changes`) lifecycle attribute.
- `timeadd(time, duration)` - Returns a UTC timestamp string corresponding to adding a given duration to `time` in RFC 3339 format.
For example, `timeadd("2017-11-22T00:00:00Z", "10m")` produces a value "2017-11-22T00:10:00Z".
- `title(string)` - Returns a copy of the string with the first characters of all the words capitalized.
- `transpose(map)` - Swaps the keys and list values in a map of lists of strings. For example, `transpose(map("a", list("1", "2"), "b", list("2", "3")))` produces a value equivalent to `map("1", list("a"), "2", list("a", "b"), "3", list("b"))`.
- `trimspace(string)` - Returns a copy of the string with all leading and trailing white spaces removed.
- `upper(string)` - Returns a copy of the string with all Unicode letters mapped to their upper case.
- `urlencode(string)` - Returns an URL-safe copy of the string.
- `uuid()` - Returns a random UUID string. This string will change with every invocation of the function, so in order to prevent diffs on every plan & apply, it must be used with the `ignore_changes` (`/docs/configuration/resources.html#ignore-changes`) lifecycle attribute.
- `values(map)` - Returns a list of the map values, in the order of the keys returned by the `keys` function. This function only works on flat maps and will return an error for maps that include nested lists or maps.
- `zipmap(list, list)` - Creates a map from a list of keys and a list of values. The keys must all be of type string, and the length of the lists must be the same. For example, to output a mapping of AWS IAM user names to the fingerprint of the key used to encrypt their initial password, you might use: `zipmap(aws_iam_user.users.*.name, aws_iam_user_login_profile.users.*.key_fingerprint)`.

Templates

Long strings can be managed using templates. Templates (/docs/providers/template/index.html) are data-sources (/docs/configuration/data-sources.html) defined by a filename and some variables to use during interpolation. They have a computed rendered attribute containing the result.

A template data source looks like:

```
data "template_file" "example" {
  template = "$${hello} $$${world}!"
  vars {
    hello = "goodnight"
    world = "moon"
  }
}

output "rendered" {
  value = "${data.template_file.example.rendered}"
}
```

Then the rendered value would be `goodnight moon!`.

You may use any of the built-in functions in your template. For more details on template usage, please see the `template_file` documentation (/docs/providers/template/d/file.html).

Using Templates with Count

Here is an example that combines the capabilities of templates with the interpolation from `count` to give us a parameterized template, unique to each resource instance:

```
variable "count" {
  default = 2
}

variable "hostnames" {
  default = {
    "0" = "example1.org"
    "1" = "example2.net"
  }
}

data "template_file" "web_init" {
  # Render the template once for each instance
  count      = "${length(var.hostnames)}"
  template   = "${file("templates/web_init.tpl")}"
  vars {
    # count.index tells us the index of the instance we are rendering
    hostname = "${var.hostnames[count.index]}"
  }
}

resource "aws_instance" "web" {
  # Create one instance for each hostname
  count      = "${length(var.hostnames)}

  # Pass each instance its corresponding template_file
  user_data = "${data.template_file.web_init.*.rendered[count.index]}"
}
```

With this, we will build a list of `template_file.web_init` data resources which we can use in combination with our list of `aws_instance.web` resources.

Math

Simple math can be performed in interpolations:

```
variable "count" {
  default = 2
}

resource "aws_instance" "web" {
  # ...

  count = "${var.count}"

  # Tag the instance with a counter starting at 1, ie. web-001
  tags {
    Name = "${format("web-%03d", count.index + 1)}"
  }
}
```

The supported operations are:

- *Add (+), Subtract (-), Multiply (*), and Divide (/)* for **float** types
- *Add (+), Subtract (-), Multiply (*), Divide (/), and Modulo (%)* for **integer** types

Operator precedences is the standard mathematical order of operations: *Multiply (*)*, *Divide (/)*, and *Modulo (%)* have precedence over *Add (+)* and *Subtract (-)*. Parenthesis can be used to force ordering.

```
"${2 * 4 + 3 * 3}" # computes to 17
"${3 * 3 + 2 * 4}" # computes to 17
"${2 * (4 + 3) * 3}" # computes to 42
```

You can use the `terraform console` ([/docs/commands/console.html](#)) command to try the math operations.

Note: Since Terraform allows hyphens in resource and variable names, it's best to use spaces between math operators to prevent confusion or unexpected behavior. For example, `${var.instance-count - 1}` will subtract **1** from the `instance-count` variable value, while `${var.instance-count-1}` will interpolate the `instance-count-1` variable value.

Load Order and Semantics

When invoking any command that loads the Terraform configuration, Terraform loads all configuration files within the directory specified in alphabetical order.

The files loaded must end in either `.tf` or `.tf.json` to specify the format that is in use. Otherwise, the files are ignored. Multiple file formats can be present in the same directory; it is okay to have one Terraform configuration file be Terraform syntax and another be JSON.

Override (/docs/configuration/override.html) files are the exception, as they're loaded after all non-override files, in alphabetical order.

The configuration within the loaded files are appended to each other. This is in contrast to being merged. This means that two resources with the same name are not merged, and will instead cause a validation error. This is in contrast to overrides (/docs/configuration/override.html), which do merge.

The order of variables, resources, etc. defined within the configuration doesn't matter. Terraform configurations are declarative (https://en.wikipedia.org/wiki/Declarative_programming), so references to other resources and variables do not depend on the order they're defined.

Getting Help

The modules in The Terraform Registry are provided and maintained by trusted HashiCorp partners and the Terraform Community. If you run into issues using a module or have additional contributions to make, you can find a link to the Module's GitHub issues on the module page.

 **consul**  AWS

Version 0.0.1 ▾

A Terraform Module for how to run Consul on AWS using Terraform and Packer

Published September 15, 2017 by [hashicorp](#)
Module managed by [gruntwork-team](#)
Provisions in the last month: 119
Source: <https://github.com/hashicorp/terraform-aws-consul> (report an issue)



Provision Instructions
Copy and paste into your Terraform configuration, insert the variables, and run `terraform init`:

```
module "consul" {  
    source = "hashicorp/consul"  
}
```

[Readme](#) [Inputs \(7\)](#) [Outputs \(15\)](#) [Dependencies \(0\)](#) [Resources \(17\)](#)

State

Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.

Terraform uses this local state to create plans and make changes to your infrastructure. Prior to any operation, Terraform does a refresh ([/docs/commands/refresh.html](#)) to update the state with the real infrastructure.

For more information on why Terraform requires state and why Terraform cannot function without state, please see the page state purpose ([/docs/state/purpose.html](#)).

Inspection and Modification

While the format of the state files are just JSON, direct file editing of the state is discouraged. Terraform provides the `terraform state` ([/docs/commands/state/index.html](#)) command to perform basic modifications of the state using the CLI.

The CLI usage and output of the state commands is structured to be friendly for Unix tools such as grep, awk, etc.

Additionally, the CLI insulates users from any format changes within the state itself. The Terraform project will keep the CLI working while the state format underneath it may shift.

Finally, the CLI manages backups for you automatically. If you make a mistake modifying your state, the state CLI will always have a backup available for you that you can restore.

Format

The state is in JSON format and Terraform will promise backwards compatibility with the state file. The JSON format makes it easy to write tools around the state if you want or to modify it by hand in the case of a Terraform bug. The "version" field on the state contents allows us to transparently move the format forward if we make modifications.

State Environments

The term *state environment*, or just *environment*, was used within the Terraform 0.9 releases to refer to the idea of having multiple distinct, named states associated with a single configuration directory.

After this concept was implemented, we received feedback that this terminology caused confusion due to other uses of the word "environment", both within Terraform itself and within organizations using Terraform.

As of 0.10, the preferred term is "workspace". For more information on workspaces, see the main Workspaces page ([/docs/state/workspaces.html](#)).

Import Existing Resources

Terraform is able to import existing infrastructure. This allows you take resources you've created by some other means and bring it under Terraform management.

To learn more about this, please visit the pages dedicated to import ([/docs/import/index.html](#)).

State Locking

If supported by your backend (/docs/backends), Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state.

State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, Terraform will not continue. You can disable state locking for most commands with the `-lock` flag but it is not recommended.

If acquiring the lock is taking longer than expected, Terraform will output a status message. If Terraform doesn't output a message, state locking is still occurring if your backend supports it.

Not all backends (/docs/backends) support locking. Please view the list of backend types (/docs/backends/types) for details on whether a backend supports locking or not.

Force Unlock

Terraform has a force-unlock command (/docs/commands/force-unlock.html) to manually unlock the state if unlocking failed.

Be very careful with this command. If you unlock the state when someone else is holding the lock it could cause multiple writers. Force unlock should only be used to unlock your own lock in the situation where automatic unlocking failed.

To protect you, the force-unlock command requires a unique lock ID. Terraform will output this lock ID if unlocking fails. This lock ID acts as a nonce (https://en.wikipedia.org/wiki/Cryptographic_nonce), ensuring that locks and unlocks target the correct lock.

Purpose of Terraform State

State is a necessary requirement for Terraform to function. It is often asked if it is possible for Terraform to work without state, or for Terraform to not use state and just inspect cloud resources on every run. This page will help explain why Terraform state is required.

As you'll see from the reasons below, state is required. And in the scenarios where Terraform may be able to get away without state, doing so would require shifting massive amounts of complexity from one place (state) to another place (the replacement concept).

Mapping to the Real World

Terraform requires some sort of database to map Terraform config to the real world. When you have a resource `resource "aws_instance" "foo"` in your configuration, Terraform uses this map to know that instance `i-abcd1234` is represented by that resource.

For some providers like AWS, Terraform could theoretically use something like AWS tags. Early prototypes of Terraform actually had no state files and used this method. However, we quickly ran into problems. The first major issue was a simple one: not all resources support tags, and not all cloud providers support tags.

Therefore, for mapping configuration to resources in the real world, Terraform uses its own state structure.

Metadata

Alongside the mappings between resources and remote objects, Terraform must also track metadata such as resource dependencies.

Terraform typically uses the configuration to determine dependency order. However, when you delete a resource from a Terraform configuration, Terraform must know how to delete that resource. Terraform can see that a mapping exists for a resource not in your configuration and plan to destroy. However, since the configuration no longer exists, the order cannot be determined from the configuration alone.

To ensure correct operation, Terraform retains a copy of the most recent set of dependencies within the state. Now Terraform can still determine the correct order for destruction from the state when you delete one or more items from the configuration.

One way to avoid this would be for Terraform to know a required ordering between resource types. For example, Terraform could know that servers must be deleted before the subnets they are a part of. The complexity for this approach quickly explodes, however: in addition to Terraform having to understand the ordering semantics of every resource for every cloud, Terraform must also understand the ordering *across providers*.

Terraform also stores other metadata for similar reasons, such as a pointer to the provider configuration that was most recently used with the resource in situations where multiple aliased providers are present.

Performance

In addition to basic mapping, Terraform stores a cache of the attribute values for all resources in the state. This is the most optional feature of Terraform state and is done only as a performance improvement.

When running a `terraform plan`, Terraform must know the current state of resources in order to effectively determine the changes that it needs to make to reach your desired configuration.

For small infrastructures, Terraform can query your providers and sync the latest attributes from all your resources. This is the default behavior of Terraform: for every plan and apply, Terraform will sync all resources in your state.

For larger infrastructures, querying every resource is too slow. Many cloud providers do not provide APIs to query multiple resources at once, and the round trip time for each resource is hundreds of milliseconds. On top of this, cloud providers almost always have API rate limiting so Terraform can only request a certain number of resources in a period of time. Larger users of Terraform make heavy use of the `-refresh=false` flag as well as the `-target` flag in order to work around this. In these scenarios, the cached state is treated as the record of truth.

Syncing

In the default configuration, Terraform stores the state in a file in the current working directory where Terraform was run. This is okay for getting started, but when using Terraform in a team it is important for everyone to be working with the same state so that operations will be applied to the same remote objects.

Remote state ([/docs/state/remote.html](#)) is the recommended solution to this problem. With a fully-featured state backend, Terraform can use remote locking as a measure to avoid two or more different users accidentally running Terraform at the same time, and thus ensure that each Terraform run begins with the most recent updated state.

Remote State

By default, Terraform stores state locally in a file named `terraform.tfstate`. When working with Terraform in a team, use of a local file makes Terraform usage complicated because each user must make sure they always have the latest state data before running Terraform and make sure that nobody else runs Terraform at the same time.

With *remote* state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team. Terraform supports storing state in Terraform Enterprise (<https://www.hashicorp.com/products/terraform/>), HashiCorp Consul (<https://www.consul.io/>), Amazon S3, and more.

Remote state is a feature of backends (/docs/backends). Configuring and using remote backends is easy and you can get started with remote state quickly. If you then want to migrate back to using local state, backends make that easy as well.

Delegation and Teamwork

Remote state gives you more than just easier version control and safer storage. It also allows you to delegate the outputs (/docs/configuration/outputs.html) to other teams. This allows your infrastructure to be more easily broken down into components that multiple teams can access.

Put another way, remote state also allows teams to share infrastructure resources in a read-only way without relying on any additional configuration store.

For example, a core infrastructure team can handle building the core machines, networking, etc. and can expose some information to other teams to run their own infrastructure. As a more specific example with AWS: you can expose things such as VPC IDs, subnets, NAT instance IDs, etc. through remote state and have other Terraform states consume that.

For example usage, see the `terraform_remote_state` data source (/docs/providers/terraform/d/remote_state.html).

While remote state is a convenient, built-in mechanism for sharing data between configurations, it is also possible to use more general stores to pass settings both to other configurations and to other consumers. For example, if your environment has HashiCorp Consul (<https://www.consul.io/>) then you can have one Terraform configuration that writes to Consul using `consul_key_prefix` (/docs/providers/consul/r/key_prefix.html) and then another that consumes those values using the `consul_keys` data source (/docs/providers/consul/d/keys.html).

Locking and Teamwork

For fully-featured remote backends, Terraform can also use state locking (/docs/state/locking.html) to prevent concurrent runs of Terraform against the same state.

Terraform Enterprise by HashiCorp (<https://www.hashicorp.com/products/terraform/>) is a commercial offering that supports an even stronger locking concept that can also detect attempts to create a new plan when an existing plan is already awaiting approval, by queuing Terraform operations in a central location. This allows teams to more easily coordinate and communicate about changes to infrastructure.

Sensitive Data in State

Terraform state can contain sensitive data depending on the resources in-use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.

Some resources (such as RDS databases) have options for PGP encrypting the values within the state. This is implemented on a per-resource basis and you should assume the value is plaintext unless otherwise documented.

When using local state, state is stored in plain-text JSON files. When using remote state (/docs/state/remote.html), state is only ever held in memory when used by Terraform. It may be encrypted at rest but this depends on the specific remote state backend.

It is important to keep this in mind if you do (or plan to) store sensitive data (e.g. database passwords, user passwords, private keys) as it may affect the risk of exposure of such sensitive data.

Recommendations

Storing state remotely may provide you encryption at rest depending on the backend you choose. As of Terraform 0.9, Terraform will only hold the state value in memory when remote state is in use. It is never explicitly persisted to disk.

For example, encryption at rest can be enabled with the S3 backend and IAM policies and logging can be used to identify any invalid access. Requests for the state go over a TLS connection.

Terraform Enterprise (<https://www.hashicorp.com/products/terraform/>) is a commercial product from HashiCorp that also acts as a backend (/docs/backends) and provides encryption at rest for state. Terraform Enterprise also knows the identity of the user requesting state and maintains a history of state changes. This can be used to provide access control and detect any breaches.

Future Work

Long term, the Terraform project wants to further improve the ability to secure sensitive data. There are plans to provide a generic mechanism for specific state attributes to be encrypted or even completely omitted from the state. These do not exist yet except on a resource-by-resource basis if documented.

Workspaces

Each Terraform configuration has an associated backend (</docs/backends/index.html>) that defines how operations are executed and where persistent data such as the Terraform state (<https://www.terraform.io/docs/state/purpose.html>) are stored.

The persistent data stored in the backend belongs to a *workspace*. Initially the backend has only one workspace, called "default", and thus there is only one Terraform state associated with that configuration.

Certain backends support *multiple* named workspaces, allowing multiple states to be associated with a single configuration. The configuration still has only one backend, but multiple distinct instances of that configuration to be deployed without configuring a new backend or changing authentication credentials.

Multiple workspaces are currently supported by the following backends:

- AzureRM (</docs/backends/types/azurerm.html>)
- Consul (</docs/backends/types/consul.html>)
- GCS (</docs/backends/types/gcs.html>)
- Local (</docs/backends/types/local.html>)
- Manta (</docs/backends/types/manta.html>)
- S3 (</docs/backends/types/s3.html>)

In the 0.9 line of Terraform releases, this concept was known as "environment". It was renamed in 0.10 based on feedback about confusion caused by the overloading of the word "environment" both within Terraform itself and within organizations that use Terraform.

Using Workspaces

Terraform starts with a single workspace named "default". This workspace is special both because it is the default and also because it cannot ever be deleted. If you've never explicitly used workspaces, then you've only ever worked on the "default" workspace.

Workspaces are managed with the `terraform workspace` set of commands. To create a new workspace and switch to it, you can use `terraform workspace new`; to switch workspaces you can use `terraform workspace select`; etc.

For example, creating a new workspace:

```
$ terraform workspace new bar
Created and switched to workspace "bar"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

As the command says, if you run `terraform plan`, Terraform will not see any existing resources that existed on the default (or any other) workspace. **These resources still physically exist**, but are managed in another Terraform workspace.

Current Workspace Interpolation

Within your Terraform configuration, you may include the name of the current workspace using the `${terraform.workspace}` interpolation sequence. This can be used anywhere interpolations are allowed.

Referencing the current workspace is useful for changing behavior based on the workspace. For example, for non-default workspaces, it may be useful to spin up smaller cluster sizes. For example:

```
resource "aws_instance" "example" {  
  count = "${terraform.workspace == "default" ? 5 : 1}"  
  
  # ... other arguments  
}
```

Another popular use case is using the workspace name as part of naming or tagging behavior:

```
resource "aws_instance" "example" {  
  tags {  
    Name = "web - ${terraform.workspace}"  
  }  
  
  # ... other arguments  
}
```

When to use Multiple Workspaces

Named workspaces allow conveniently switching between multiple instances of a *single* configuration within its *single* backend. They are convenient in a number of situations, but cannot solve all problems.

A common use for multiple workspaces is to create a parallel, distinct copy of a set of infrastructure in order to test a set of changes before modifying the main production infrastructure. For example, a developer working on a complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.

Non-default workspaces are often related to feature branches in version control. The default workspace might correspond to the "master" or "trunk" branch, which describes the intended state of production infrastructure. When a feature branch is created to develop a change, the developer of that feature might create a corresponding workspace and deploy into it a temporary "copy" of the main infrastructure so that changes can be tested without affecting the production infrastructure. Once the change is merged and deployed to the default workspace, the test infrastructure can be destroyed and the temporary workspace deleted.

When Terraform is used to manage larger systems, teams should use multiple separate Terraform configurations that correspond with suitable architectural boundaries within the system so that different components can be managed separately and, if appropriate, by distinct teams. Workspaces *alone* are not a suitable tool for system decomposition, because each subsystem should have its own separate configuration and backend, and will thus have its own distinct set of workspaces.

In particular, organizations commonly want to create a strong separation between multiple deployments of the same infrastructure serving different development stages (e.g. staging vs. production) or different internal teams. In this case, the backend used for each deployment often belongs to that deployment, with different credentials and access controls. Named

workspaces are *not* a suitable isolation mechanism for this scenario.

Instead, use one or more re-usable modules ([/docs/modules/index.html](#)) to represent the common elements, and then represent each instance as a separate configuration that instantiates those common elements in the context of a different backend. In that case, the root module of each configuration will consist only of a backend configuration and a small number of module blocks whose arguments describe any small differences between the deployments.

Where multiple configurations are representing distinct system components rather than multiple deployments, data can be passed from one component to another using paired resources types and data sources. For example:

- Where a shared Consul (<https://consul.io/>) cluster is available, use `consul_key_prefix` ([/docs/providers/consul/r/key_prefix.html](#)) to publish to the key/value store and `consul_keys` ([/docs/providers/consul/d/keys.html](#)) to retrieve those values in other configurations.
- In systems that support user-defined labels or tags, use a tagging convention to make resources automatically discoverable. For example, use the `aws_vpc` resource type ([/docs/providers/aws/r/vpc.html](#)) to assign suitable tags and then the `aws_vpc` data source ([/docs/providers/aws/d/vpc.html](#)) to query by those tags in other configurations.
- For server addresses, use a provider-specific resource to create a DNS record with a predictable name and then either use that name directly or use the `dns` provider ([/docs/providers/dns/index.html](#)) to retrieve the published addresses in other configurations.
- If a Terraform state for one configuration is stored in a remote backend that is accessible to other configurations then `terraform_remote_state` ([/docs/providers/terraform/d/remote_state.html](#)) can be used to directly consume its root module outputs from those other configurations. This creates a tighter coupling between configurations, but avoids the need for the "producer" configuration to explicitly publish its results in a separate system.

Workspace Internals

Workspaces are technically equivalent to renaming your state file. They aren't any more complex than that. Terraform wraps this simple notion with a set of protections and support for remote state.

For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d`. This directory should be treated similarly to local-only `terraform.tfstate`; some teams commit these files to version control, although using a remote backend instead is recommended when there are multiple collaborators.

For remote state ([/docs/state/remote.html](#)), the workspaces are stored directly in the configured backend ([/docs/backends](#)). For example, if you use Consul ([/docs/backends/types/consul.html](#)), the workspaces are stored by appending the workspace name to the state path. To ensure that workspace names are stored correctly and safely in all backends, the name must be valid to use in a URL path segment without escaping.

The important thing about workspace internals is that workspaces are meant to be a shared resource. They aren't a private, local-only notion (unless you're using purely local state and not committing it).

The "current workspace" name is stored only locally in the ignored `.terraform` directory. This allows multiple team members to work on different workspaces concurrently.

Download Terraform

Below are the available downloads for the latest version of Terraform (0.11.11). Please download the proper package for your operating system and architecture.

You can find the SHA256 checksums for Terraform 0.11.11

(https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_SHA256SUMS) online and you can verify the checksums signature file (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_SHA256SUMS.sig) which has been signed using HashiCorp's GPG key (<https://hashicorp.com/security.html>). You can also download older versions of Terraform (<https://releases.hashicorp.com/terraform/>) from the releases service.

Check out the v0.11.11 CHANGELOG (<https://github.com/hashicorp/terraform/blob/v0.11.11/CHANGELOG.md>) for information on the latest release.



64-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_darwin_amd64.zip)



32-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_freebsd_386.zip) |

64-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_freebsd_amd64.zip) |

Arm (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_freebsd_arm.zip)



32-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_linux_386.zip) |

64-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_linux_amd64.zip) |

Arm (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_linux_arm.zip)



32-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_openbsd_386.zip) |

64-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_openbsd_amd64.zip)



64-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_solaris_amd64.zip)



32-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_windows_386.zip) |

64-bit (https://releases.hashicorp.com/terraform/0.11.11/terraform_0.11.11_windows_amd64.zip)



(https://www.fastly.com?utm_source=hashicorp)

Terraform Guides

Welcome to the Terraform guides section! If you are just getting started with Terraform, please start with the Terraform introduction (</intro/index.html>) instead and then continue on to the guides. The guides provide examples for common Terraform workflows and actions for both beginner and advanced Terraform users.

The Core Terraform Workflow

The core Terraform workflow has three steps:

1. **Write** - Author infrastructure as code.
2. **Plan** - Preview changes before applying.
3. **Apply** - Provision reproducible infrastructure.

This guide walks through how each of these three steps plays out in the context of working as an individual practitioner, how they evolve when a team is collaborating on infrastructure, and how Terraform Enterprise enables this workflow to run smoothly for entire organizations.

Working as an Individual Practitioner

Let's first walk through how these parts fit together as an individual working on infrastructure as code.

Write

You write Terraform configuration just like you write code: in your editor of choice. It's common practice to store your work in a version controlled repository even when you're just operating as an individual.

```
# Create repository
$ git init my-infra && cd my-infra

Initialized empty Git repository in /.../my-infra/.git/

# Write initial config
$ vim main.tf

# Initialize Terraform
$ terraform init

Initializing provider plugins...
# ...
Terraform has been successfully initialized!
```

As you make progress on authoring your config, repeatedly running plans can help flush out syntax errors and ensure that your config is coming together as you expect.

```
# Make edits to config
$ vim main.tf

# Review plan
$ terraform plan

# Make additional edits, and repeat
$ vim main.tf
```

This parallels working on application code as an individual, where a tight feedback loop between editing code and running test commands is useful.

Plan

When the feedback loop of the Write step has yielded a change that looks good, it's time to commit your work and review the final plan.

```
$ git add main.tf  
$ git commit -m 'Managing infrastructure as code!'  
  
[master (root-commit) f735520] Managing infrastructure as code!  
1 file changed, 1 insertion(+)
```

Because `terraform apply` will display a plan for confirmation before proceeding to change any infrastructure, that's the command you run for final review.

```
$ terraform apply  
  
An execution plan has been generated and is shown below.  
# ...
```

Apply

After one last check, you are ready to tell Terraform to provision real infrastructure.

```
Do you want to perform these actions?  
  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.  
Enter a value: yes  
  
# ...  
  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

At this point, it's common to push your version control repository to a remote location for safekeeping.

```
$ git remote add origin https://github.com/*user*/repo*.git  
$ git push origin master
```

This core workflow is a loop; the next time you want to make changes, you start the process over from the beginning.

Notice how closely this workflow parallels the process of writing application code or scripts as an individual? This is what we mean when we talk about Terraform enabling infrastructure as code.

Working as a Team

Once multiple people are collaborating on Terraform configuration, new steps must be added to each part of the core workflow to ensure everyone is working together smoothly. You'll see that many of these steps parallel the workflow changes we make when we work on application code as teams rather than as individuals.

Write

While each individual on a team still makes changes to Terraform configuration in their editor of choice, they save their changes to version control *branches* to avoid colliding with each other's work. Working in branches enables team members to resolve mutually incompatible infrastructure changes using their normal merge conflict workflow.

```
$ git checkout -b add-load-balancer  
Switched to a new branch 'add-load-balancer'
```

Running iterative plans is still useful as a feedback loop while authoring configuration, though having each team member's computer able to run them becomes more difficult with time. As the team and the infrastructure grows, so does the number of sensitive input variables (e.g. API Keys, SSL Cert Pairs) required to run a plan.

To avoid the burden and the security risk of each team member arranging all sensitive inputs locally, it's common for teams to migrate to a model in which Terraform operations are executed in a shared Continuous Integration (CI) environment. The work needed to create such a CI environment is nontrivial, and is outside the scope of this core workflow overview, but a full deep dive on this topic can be found in our Running Terraform in Automation (<https://www.terraform.io/guides/running-terraform-in-automation.html>) guide.

This longer iteration cycle of committing changes to version control and then waiting for the CI pipeline to execute is often lengthy enough to prohibit using speculative plans as a feedback loop while authoring individual Terraform configuration changes. Speculative plans are still useful before new Terraform changes are applied or even merged to the main development branch, however, as we'll see in a minute.

Plan

For teams collaborating on infrastructure, Terraform's plan output creates an opportunity for team members to review each other's work. This allows the team to ask questions, evaluate risks, and catch mistakes before any potentially harmful changes are made.

The natural place for these reviews to occur is alongside pull requests within version control--the point at which an individual proposes a merge from their working branch to the shared team branch. If team members review proposed config changes alongside speculative plan output, they can evaluate whether the intent of the change is being achieved by the plan.

The problem becomes producing that speculative plan output for the team to review. Some teams that still run Terraform locally make a practice that pull requests should include an attached copy of speculative plan output generated by the change author. Others arrange for their CI system to post speculative plan output to pull requests automatically.

Initial resources #1

Open hashiadmin wants to merge 1 commit into `master` from `initial-resources`

Conversation 0 Commits 1 Files changed 1

hashiadmin commented 10 minutes ago

Adding initial resources for my application

Initial resources ... 5bfb27b

hashiadmin commented 2 minutes ago

Here is the Terraform plan output for this change:

```
An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:  
+ create  
  
Terraform will perform the following actions:  
  
+ null_resource.foo  
  id: <computed>  
  
Plan: 1 to add, 0 to change, 0 to destroy.
```

In addition to reviewing the plan for the proper expression of its author's intent, the team can also make an evaluation whether they want this change to happen now. For example, if a team notices that a certain change could result in service disruption, they may decide to delay merging its pull request until they can schedule a maintenance window.

Apply

Once a pull request has been approved and merged, it's important for the team to review the final concrete plan that's run against the shared team branch and the latest version of the state file.

This plan has the potential to be different than the one reviewed on the pull request due to issues like merge order or recent infrastructural changes. For example, if a manual change was made to your infrastructure since the plan was reviewed, the plan might be different when you merge.

It is at this point that the team asks questions about the potential implications of applying the change. Do we expect any service disruption from this change? Is there any part of this change that is high risk? Is there anything in our system that we should be watching as we apply this? Is there anyone we need to notify that this change is happening?

Depending on the change, sometimes team members will want to watch the apply output as it is happening. For teams that are running Terraform locally, this may involve a screen share with the team. For teams running Terraform in CI, this may involve gathering around the build log.

Just like the workflow for individuals, the core workflow for teams is a loop that plays out for each change. For some teams this loop happens a few times a week, for others, many times a day.

The Core Workflow Enhanced by Terraform Enterprise

While the above described workflows enable the safe, predictable, and reproducible creating or changing of infrastructure, there are multiple collaboration points that can be streamlined, especially as teams and organizations scale. We designed Terraform Enterprise to support and enhance the core Terraform workflow for anyone collaborating on infrastructure, from small teams to large organizations. Let's look at how Terraform Enterprise makes for a better experience at each step.

Write

Terraform Enterprise provides a centralized and secure location for storing input variables and state while also bringing back a tight feedback loop for speculative plans for config authors. Terraform configuration interacts with Terraform Enterprise via the "remote" backend ([/docs/backends/types/remote.html](#)).

```
terraform {  
  backend "remote" {  
    organization = "my-org"  
    workspaces {  
      prefix = "my-app-"  
    }  
  }  
}
```

Once the backend is wired up, a Terraform Enterprise API key is all that's needed by team members to be able to edit config and run speculative plans against the latest version of the state file using all the remotely stored input variables.

```
$ terraform workspace select my-app-dev  
Switched to workspace "my-app-dev".  
  
$ terraform plan  
  
Running plan remotely in Terraform Enterprise.  
  
Output will stream here. To view this plan in a browser, visit:  
  
https://app.terraform.io/my-org/my-app-dev/.../  
  
Refreshing Terraform state in-memory prior to plan...  
  
# ...  
  
Plan: 1 to add, 0 to change, 0 to destroy.
```

With the assistance of this plan output, team members can each work on authoring config until it is ready to propose as a change via a pull request.

Plan

Once a pull request is ready for review, Terraform Enterprise makes the process of reviewing a speculative plan easier for

team members. First, the plan is automatically run when the pull request is created. Status updates to the pull request indicate while the plan is in progress.

Once the plan is complete, the status update indicates whether there were any changes in the speculative plan, right from the pull request view.

Add a resource #4

The screenshot shows a GitHub pull request titled "Add a resource #4". The pull request is open and merges 1 commit into the master branch from the adding-bar branch. The commit message is "Add a second resource, bar". A comment from hashiadmin states "Add a second resource, bar". Below the commit, there is a link "Add a resource". The status bar at the bottom right shows "Verified" and a green checkmark next to the commit hash 4232b14.

hashiadmin commented 28 minutes ago

Add a second resource, bar

Add a resource

Verified ✓ 4232b14

Add more commits by pushing to the **adding-bar** branch on **core-workflow-guide/my-app-terraform**.

All checks have passed

1 successful check

atlas/my-org/my-app-terraform — Terraform plan: 1 to add, 0 to change, 0 to...

This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

For certain types of changes, this information is all that's needed for a team member to be able to approve the pull request. When a teammate needs to do a full review of the plan, clicking the link to Terraform Enterprise brings up a view that allows them to quickly analyze the full plan details.

The screenshot shows the HashiCorp Terraform Enterprise interface for a project named "my-app-terraform". The top navigation bar includes a logo, the organization name "my-org", and a "Queue Plan" button. The main menu has tabs for "Current Run", "Runs" (which is selected), "States", "Variables", "Settings", "Integrations", "Version Control", and "Access".

The "Runs" page displays a "Planned" resource addition titled "Add a resource". A message indicates that "hashiadmin triggered a pull request run (#3) from GitHub 3 minutes ago". The status is "Plan finished" 3 minutes ago, with 1 resource to add, 0 to change, and 0 to destroy. The execution history shows it was Queued, Started, and Finished 3 minutes ago. Below the log, a note states: "The refreshed state will be used to calculate this plan, but will not be persisted to local or remote state storage." The log output shows:

```
null_resource.foo: Refreshing state... (ID: 618339531673542557)
```

An execution plan has been generated and is shown below. Resource actions are indicated with the following symbols:

- + create

Terraform will perform the following actions:

- + null_resource.bar
id: <computed>

Plan: 1 to add, 0 to change, 0 to destroy.

A callout box with an info icon says: "Heads up: This run was started from a pull request and cannot be applied. To apply these changes the pull request must first be merged into the default branch."

Below the log, there's a "Comment" section with a "Comment" input field and a "Add Comment" button.



Support Terms Privacy Security © 2018 HashiCorp, Inc.

This page allows the reviewer to quickly determine if the plan is matching the config author's intent and evaluate the risk of the change.

Apply

After merge, Terraform Enterprise presents the concrete plan to the team for review and approval.

The screenshot shows the Terraform Enterprise web application interface. At the top, the URL is https://app.terraform.io/app/my-org/my-app-terraform/runs/run-bJ4NzSP9y32XUZjQ. The navigation bar includes tabs for Current Run, Runs (which is selected), States, Variables, Settings, Integrations, Version Control, and Access. A dropdown menu for 'my-org' is open. In the top right, there's a 'Queue Plan' button.

The main content area displays a run titled "Merge pull request #3 from core-wor...". A prominent orange banner at the top left of the run card says "! NEEDS CONFIRMATION". The run card shows that "hashiadmin triggered a run from GitHub 2 minutes ago".

The run status is "Plan finished" (indicated by a green checkmark) 2 minutes ago. It shows "Resources: 1 to add, 0 to change, 0 to destroy". Below this, the execution history is listed: "Queued 2 minutes ago > Started 2 minutes ago > Finished 2 minutes ago".

Below the history, there are buttons for "View raw log", "Top", "Bottom", "Expand", and "Full Screen". The log output shows:

```
The refreshed state will be used to calculate this plan, but will not be persisted to local or remote state storage.

null_resource.foo: Refreshing state... (ID: 618339531673542557)

-----
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

+ null_resource.bar
  id: <computed>

Plan: 1 to add, 0 to change, 0 to destroy.
```

At the bottom of the run card, there is a button labeled "Apply pending".

The team can discuss any outstanding questions about the plan before the change is made.

The screenshot shows a browser window for Terraform Enterprise at the URL <https://app.terraform.io/app/my-org/my-app-terraform/runs/run-bJ4NzSP9y32XUZjQ>. The main content area displays a completed Terraform run:

- hashiadmin triggered a run from GitHub 2 minutes ago**
- Plan finished** 2 minutes ago | Resources: 1 to add, 0 to change, 0 to destroy
- Queued 2 minutes ago > Started 2 minutes ago > Finished 2 minutes ago
- View raw log** | **Top** | **Bottom** | **Expand** | **Full Screen**
- The refreshed state will be used to calculate this plan, but will not be persisted to local or remote state storage.
- null_resource.foo: Refreshing state... (ID: 618339531673542557)
- An execution plan has been generated and is shown below. Resource actions are indicated with the following symbols:
 - + create
- Terraform will perform the following actions:
- + null_resource.bar
id: <computed>
- Plan: 1 to add, 0 to change, 0 to destroy.

Below the plan details, there is a section titled "Apply pending" with a dropdown arrow:

- phinze** 1 minute ago: Is "bar" what we wanted to call this resource?
- evanphx** a few seconds ago: This is for the construction account right? Sounds good!

A message box at the bottom says: **Needs Confirmation:** Check the plan and confirm to apply it, or discard the run.

Buttons at the bottom: **Confirm & Apply** (blue), **Discard Run**, **Add Comment**.

Once the Apply is confirmed, Terraform Enterprise displays the progress live to anyone who'd like to watch.

The screenshot shows a web browser window for Terraform Enterprise. The URL is <https://app.terraform.io/app/my-org/my-app-terraform/runs/run-bJ4NzSP9y32XUZjQ>. The main content area displays a terminal log for a run named "Apply running" which was "Queued" and "Started" a few seconds ago. The log output is as follows:

```
Terraform v0.11.8
Initializing plugins and modules...
2018/08/17 22:08:32 [DEBUG] Using modified User-Agent: Terraform/0.11.8 TFE/d012b76
null_resource.bar: Creating...
null_resource.bar: Creation complete after 0s (ID: 3423346902457632441)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Below the log, there is a comment section with two entries:

 **phinze** 1 minute ago
Is "bar" what we wanted to call this resource?

 **evanphx** a few seconds ago
This is for the construction account right? Sounds good!

 **phinze** a few seconds ago
It is! Ok cool let's ship this.

A green button labeled "Run confirmed" is visible. At the bottom, there is a "Comment" input field with placeholder text "Comment: Leave feedback or record a decision." and a "Add Comment" button.

Conclusion

There are many different ways to use Terraform: as an individual user, a single team, or an entire organization at scale. Choosing the best approach for the density of collaboration needed will provide the most return on your investment in the core Terraform workflow. For organizations using Terraform at scale, Terraform Enterprise introduces new layers that build on this core workflow to solve problems unique to teams and organizations.

Running Terraform in Automation

This is an advanced guide! When getting started with Terraform, it's recommended to use it locally from the command line. Automation can become valuable once Terraform is being used regularly in production, or by a larger team, but this guide assumes familiarity with the normal, local CLI workflow.

For teams that use Terraform as a key part of a change management and deployment pipeline, it can be desirable to orchestrate Terraform runs in some sort of automation in order to ensure consistency between runs, and provide other interesting features such as integration with version control hooks.

Automation of Terraform can come in various forms, and to varying degrees. Some teams continue to run Terraform locally but use *wrapper scripts* to prepare a consistent working directory for Terraform to run in, while other teams run Terraform entirely within an orchestration tool such as Jenkins.

This guide covers some things that should be considered when implementing such automation, both to ensure safe operation of Terraform and to accommodate some current limitations in Terraform's workflow that require careful attention in automation.

The guide assumes that Terraform will be running in an *non-interactive* environment, where it is not possible to prompt for input at the terminal. This is not necessarily true for wrapper scripts, but is often true when running in orchestration tools.

This is a general guide, giving an overview of things to consider when implementing orchestration of Terraform. Due to its general nature, it is not possible to go into specifics about any particular tools, though other tool-specific guides may be produced later if best practices emerge around such a tool.

Automated Workflow Overview

When running Terraform in automation, the focus is usually on the core plan/apply cycle. The main path, then, is broadly the same as for CLI usage:

1. Initialize the Terraform working directory.
2. Produce a plan for changing resources to match the current configuration.
3. Have a human operator review that plan, to ensure it is acceptable.
4. Apply the changes described by the plan.

Steps 1, 2 and 4 can be carried out using the familiar Terraform CLI commands, with some additional options:

- `terraform init -input=false` to initialize the working directory.
- `terraform plan -out=tfplan -input=false` to create a plan and save it to the local file `tfplan`.
- `terraform apply -input=false tfplan` to apply the plan stored in the file `tfplan`.

The `-input=false` option indicates that Terraform should not attempt to prompt for input, and instead expect all necessary values to be provided by either configuration files or the command line. It may therefore be necessary to use the `-var` and `-var-file` options on `terraform plan` to specify any variable values that would traditionally have been manually-entered under interactive usage.

It is strongly recommended to use a backend that supports remote state (</docs/state/remote.html>), since that allows Terraform to automatically save the state in a persistent location where it can be found and updated by subsequent runs. Selecting a backend that supports state locking (</docs/state/locking.html>) will additionally provide safety against race conditions that can be caused by concurrent Terraform runs.

Controlling Terraform Output in Automation

By default, some Terraform commands conclude by presenting a description of a possible next step to the user, often including a specific command to run next.

An automation tool will often abstract away the details of exactly which commands are being run, causing these messages to be confusing and un-actionable, and possibly harmful if they inadvertently encourage a user to bypass the automation tool entirely.

When the environment variable `TF_IN_AUTOMATION` is set to any non-empty value, Terraform makes some minor adjustments to its output to de-emphasize specific commands to run. The specific changes made will vary over time, but generally-speaking Terraform will consider this variable to indicate that there is some wrapping application that will help the user with the next step.

To reduce complexity, this feature is implemented primarily for the main workflow commands described above. Other ancillary commands may still produce command line suggestions, regardless of this setting.

Plan and Apply on different machines

When running in an orchestration tool, it can be difficult or impossible to ensure that the `plan` and `apply` subcommands are run on the same machine, in the same directory, with all of the same files present.

Running `plan` and `apply` on different machines requires some additional steps to ensure correct behavior. A robust strategy is as follows:

- After `plan` completes, archive the entire working directory, including the `.terraform` subdirectory created during `init`, and save it somewhere where it will be available to the `apply` step. A common choice is as a "build artifact" within the chosen orchestration tool.
- Before running `apply`, obtain the archive created in the previous step and extract it *at the same absolute path*. This recreates everything that was present after `plan`, avoiding strange issues where local files were created during the `plan` step.

Terraform currently makes some assumptions which must be accommodated by such an automation setup:

- The saved plan file can contain absolute paths to child modules and other data files referred to by configuration. Therefore it is necessary to ensure that the archived configuration is extracted at an identical absolute path. This is most commonly achieved by running Terraform in some sort of isolation, such as a Docker container, where the filesystem layout can be controlled.
- Terraform assumes that the plan will be applied on the same operating system and CPU architecture as where it was created. For example, this means that it is not possible to create a plan on a Windows computer and then apply it on a Linux server.
- Terraform expects the provider plugins that were used to produce a plan to be available and identical when the plan is

applied, to ensure that the plan is interpreted correctly. An error will be produced if Terraform or any plugins are upgraded between creating and applying a plan.

- Terraform can't automatically detect if the credentials used to create a plan grant access to the same resources used to apply that plan. If using different credentials for each (e.g. to generate the plan using read-only credentials) it is important to ensure that the two are consistent in which account on the corresponding service they belong to.

The plan file contains a full copy of the configuration, the state that the plan applies to, and any variables passed to `terraform plan`. If any of these contain sensitive data then the archived working directory containing the plan file should be protected accordingly. For provider authentication credentials, it is recommended to use environment variables instead where possible since these are *not* included in the plan or persisted to disk by Terraform in any other way.

Interactive Approval of Plans

Another challenge with automating the Terraform workflow is the desire for an interactive approval step between plan and apply. To implement this robustly, it is important to ensure that either only one plan can be outstanding at a time or that the two steps are connected such that approving a plan passes along enough information to the apply step to ensure that the correct plan is applied, as opposed to some later plan that also exists.

Different orchestration tools address this in different ways, but generally this is implemented via a *build pipeline* feature, where different steps can be applied in sequence, with later steps having access to data produced by earlier steps.

The recommended approach is to allow only one plan to be outstanding at a time. When a plan is applied, any other existing plans that were produced against the same state are invalidated, since they must now be recomputed relative to the new state. By forcing plans to be approved (or dismissed) in sequence, this can be avoided.

Auto-Approval of Plans

While manual review of plans is strongly recommended for production use-cases, it is sometimes desirable to take a more automatic approach when deploying in pre-production or development situations.

Where manual approval is not required, a simpler sequence of commands can be used:

- `terraform init -input=false`
- `terraform apply -input=false -auto-approve`

This variant of the apply command implicitly creates a new plan and then immediately applies it. The `-auto-approve` option tells Terraform not to require interactive approval of the plan before applying it.

When Terraform is empowered to make destructive changes to infrastructure, manual review of plans is always recommended unless downtime is tolerated in the event of unintended changes. Use automatic approval **only** with non-critical infrastructure.

Testing Pull Requests with `terraform plan`

`terraform plan` can be used as a way to perform certain limited verification of the validity of a Terraform configuration, without affecting real infrastructure. Although the plan step updates the state to match real resources, thus ensuring an accurate plan, the updated state is *not* persisted, and so this command can safely be used to produce "throwaway" plans that are created only to aid in code review.

When implementing such a workflow, hooks can be used within the code review tool in question (for example, Github Pull Requests) to trigger an orchestration tool for each new commit under review. Terraform can be run in this case as follows:

- `terraform plan -input=false`

As in the "main" workflow, it may be necessary to provide `-var` or `-var-file` as appropriate. The `-out` option is not used in this scenario because a plan produced for code review purposes will never be applied. Instead, a new plan can be created and applied from the primary version control branch once the change is merged.

Beware that passing sensitive/secret data to Terraform via variables or via environment variables will make it possible for anyone who can submit a PR to discover those values, so this flow must be used with care on an open source project, or on any private project where some or all contributors should not have direct access to credentials, etc.

Multi-environment Deployment

Automation of Terraform often goes hand-in-hand with creating the same configuration multiple times to produce parallel environments for use-cases such as pre-release testing or multi-tenant infrastructure. Automation in such a situation can help ensure that the correct settings are used for each environment, and that the working directory is properly configured before each operation.

The two most interesting commands for multi-environment orchestration are `terraform init` and `terraform workspace`. The former can be used with additional options to tailor the backend configuration for any differences between environments, while the latter can be used to safely switch between multiple states for the same config stored in a single backend.

Where possible, it's recommended to use a single backend configuration for all environments and use the `terraform workspace` command to switch between workspaces:

- `terraform init -input=false`
- `terraform workspace select QA`

In this usage model, a fixed naming scheme is used within the backend storage to allow multiple states to exist without any further configuration.

Alternatively, the automation tool can set the environment variable `TF_WORKSPACE` to an existing workspace name, which overrides any selection made with the `terraform workspace select` command. Using this environment variable is recommended only for non-interactive usage, since in a local shell environment it can be easy to forget the variable is set and apply changes to the wrong state.

In some more complex situations it is impossible to share the same backend configuration ([/docs/backends/config.html](#)) across environments. For example, the environments may exist in entirely separate accounts within the target service, and thus need to use different credentials or endpoints for the backend itself. In such situations, backend configuration settings can be overridden via the `-backend-config` option to `terraform init` ([/docs/commands/init.html#backend-config](#)).

Pre-installed Plugins

In default usage, `terraform init` (/docs/commands/init.html#backend-config) downloads and installs the plugins for any providers used in the configuration automatically, placing them in a subdirectory of the `.terraform` directory. This affords a simpler workflow for straightforward cases, and allows each configuration to potentially use different versions of plugins.

In automation environments, it can be desirable to disable this behavior and instead provide a fixed set of plugins already installed on the system where Terraform is running. This then avoids the overhead of re-downloading the plugins on each execution, and allows the system administrator to control which plugins are available.

To use this mechanism, create a directory somewhere on the system where Terraform will run and place into it the plugin executable files. The plugin release archives are available for download on releases.hashicorp.com (<https://releases.hashicorp.com/>). Be sure to download the appropriate archive for the target operating system and architecture.

After extracting the necessary plugins, the contents of the new plugin directory will look something like this:

```
$ ls -lah /usr/lib/custom-terraform-plugins
-rwxrwxr-x 1 user user 84M Jun 13 15:13 terraform-provider-aws-v1.0.0-x3
-rwxrwxr-x 1 user user 84M Jun 13 15:15 terraform-provider-rundeck-v2.3.0-x3
-rwxrwxr-x 1 user user 84M Jun 13 15:15 terraform-provider-mysql-v1.2.0-x3
```

The version information at the end of the filenames is important so that Terraform can infer the version number of each plugin. Multiple versions of the same provider plugin can be installed, and Terraform will use the newest one that matches the provider version constraints (/docs/configuration/providers.html#provider-versions) in the Terraform configuration.

With this directory populated, the usual auto-download and plugin discovery (/docs/extend/how-terraform-works.html#discovery) behavior can be bypassed using the `-plugin-dir` option to `terraform init`:

- `terraform init -input=false -plugin-dir=/usr/lib/custom-terraform-plugins`

When this option is used, only the plugins in the given directory are available for use. This gives the system administrator a high level of control over the execution environment, but on the other hand it prevents use of newer plugin versions that have not yet been installed into the local plugin directory. Which approach is more appropriate will depend on unique constraints within each organization.

Plugins can also be provided along with the configuration by creating a `terraform.d/plugins/OS_ARCH` directory, which will be searched before automatically downloading additional plugins. The `-get-plugins=false` flag can be used to prevent Terraform from automatically downloading additional plugins.

Terraform Enterprise

As an alternative to home-grown automation solutions, Hashicorp offers Terraform Enterprise (<https://www.hashicorp.com/products/terraform/>).

Internally, Terraform Enterprise runs the same Terraform CLI commands described above, using the same release binaries offered for download on this site.

Terraform Enterprise builds on the core Terraform CLI functionality to add additional features such as role-based access control, orchestration of the plan and apply lifecycle, a user interface for reviewing and approving plans, and much more.

It will always be possible to run Terraform via in-house automation, to allow for usage in situations where Terraform

Enterprise is not appropriate. It is recommended to consider Terraform Enterprise as an alternative to in-house solutions, since it provides an out-of-the-box solution that already incorporates the best practices described in this guide and can thus reduce time spent developing and maintaining an in-house alternative.

Terraform Provider Development Program

The Terraform Provider Development Program allows vendors to build Terraform providers that are officially approved and tested by HashiCorp and listed on the official Terraform website. The program is intended to be largely self-serve, with links to information sources, clearly defined steps, and checkpoints.

Building your own provider? If you're building your own provider and aren't interested in having HashiCorp officially approve and regularly test the provider, refer to the Writing Custom Providers guide (</guides/writing-custom-terraform-providers.html>) and the Extending Terraform (</docs/extend/index.html>) section.

What is a Terraform Provider?

Terraform is used to create, manage, and manipulate infrastructure resources. Examples of resources include physical machines, VMs, network switches, containers, etc. Almost any infrastructure noun can be represented as a resource in Terraform.

A provider is responsible for understanding API interactions with the underlying infrastructure like a cloud (AWS, GCP, Azure), a PaaS service (Heroku), a SaaS service (DNSimple, CloudFlare), or on-prem resources (vSphere). It then exposes these as resources users can code to. Terraform presently supports more than 70 providers, a number that has more than doubled in the past 12 months.

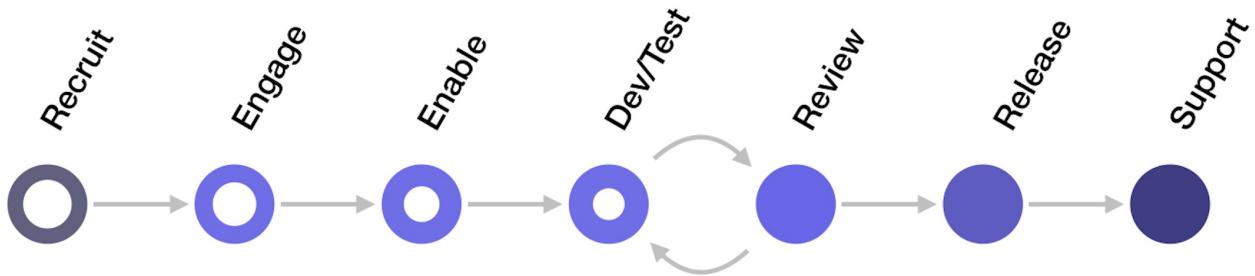
All providers integrate into and operate with Terraform exactly the same way. The table below is intended to help users understand who develops, maintains and tests a particular provider.

| Category | Developed /Maintained by | Reviewed by HashiCorp | Tested by | Listed on terraform.io | Examples |
|--------------------|--------------------------|-----------------------|-----------|--|---------------------------------------|
| HashiCorp / Vendor | HashiCorp / Vendor | Yes | HashiCorp | Yes | AWS, Azure, GCP, Oracle, Packet, etc. |
| Vendor / Community | Vendor / Community | No | Vendor | No | HP Oneview, etc. |

Note: This document is primarily intended for the "HashiCorp/Vendors" row in the table above. Community contributors who're interested in contributing to existing providers or building new providers should refer to the Writing Custom Providers guide (</guides/writing-custom-terraform-providers.html>).

Provider Development Process

The provider development process is divided into six steps below. By following these steps, providers can be developed alongside HashiCorp to ensure new providers are able to be published in Terraform as quickly as possible.



1. **Engage:** Initial contact between vendor and HashiCorp
2. **Enable:** Information and articles to aid with the provider development
3. **Dev/Test:** Provider development and test process
4. **Review:** HashiCorp code review and acceptance tests (iterative process)
5. **Release:** Provider availability and listing on [terraform.io](https://www.terraform.io) (<https://www.terraform.io>)
6. **Support:** Ongoing maintenance and support of the provider by the vendor.

1. Engage

Please begin by providing some basic information about the provider that is being built via a simple webform (<https://goo.gl/forms/iqfz6H9UK91X9LQp2>).

This information is captured upfront and used by HashiCorp to track the provider through various stages. The information is also used to notify the provider developer of any overlapping work, perhaps coming from the community.

Terraform has a large and active community and ecosystem of partners that may have already started working on the same provider. We'll do our best to connect similar parties to avoid duplicate work.

2. Enable

We've found the provider development to be fairly straightforward and simple when vendors pay close attention and follow to the resources below. Adopting the same structure and coding patterns helps expedite the review and release cycles.

- Writing custom providers guide (<https://www.terraform.io/guides/writing-custom-terraform-providers.html>)
- How-to build a provider video (<https://www.youtube.com/watch?v=2BvpqmFpchl>)
- Sample provider developed by partner (<http://container-solutions.com/write-terraform-provider-part-1/>)
- Example providers for reference: AWS (<https://github.com/terraform-providers/terraform-provider-aws>), OPC (<https://github.com/terraform-providers/terraform-provider-opc>)
- Contributing to Terraform guidelines
(<https://github.com/hashicorp/terraform/blob/master/.github/CONTRIBUTING.md>)
- Gitter HashiCorp-Terraform room (<https://gitter.im/hashicorp-terraformation/Lobby>).

3. Development & Test

Terraform providers are written in the Go (<https://golang.org/>) programming language. The Writing Custom Providers guide ([/guides/writing-custom-terraform-providers.html](#)) is a good resource for developers to begin writing a new provider.

The best approach to building a new provider is to be familiar with both the Writing Custom Providers ([/guides/writing-custom-terraform-providers.html](#)) guide and Extending Terraform ([/docs/extend/index.html](#)) section. The guide will give you an introduction in code structure and the basics of authoring a plugin that Terraform can interact with. The Extending Terraform section contains guides, best practices, and API reference for developers writing Terraform plugins. Additionally developers are encouraged to use the AWS provider (<https://github.com/terraform-providers/terraform-provider-aws>) as an implementation reference. Given the wide surface area of this provider, almost all resource types and preferred code constructs are covered in it.

It is recommended for vendors to first develop support for one or two resources and go through an initial review cycle before developing the code for the remaining resources. This helps catch any issues early on in the process and avoids errors from getting multiplied. In addition, it is advised to follow existing conventions you see in the codebase, and ensure your code is formatted with `go fmt`.

The provider code should include an acceptance test suite with tests for each individual resource that holistically tests its behavior. The Writing Acceptance Tests section in the Contributing to Terraform (<https://github.com/hashicorp/terraform/blob/master/.github/CONTRIBUTING.md>) document explains how to approach these. It is recommended to randomize the names of the tests as opposed to using unique static names, as that permits us to parallelize the test execution.

Each provider has a section in the Terraform documentation. You'll want to add new index file and individual pages for each resource supported by the provider.

While developing the provider code yourself is certainly possible, you can also choose to leverage one of the following development agencies who've developed Terraform providers in the past and are familiar with the requirements and process.

| Partner | Email | Website |
|--------------------|---|---|
| Crest Data Systems | malhar@crestdatasys.com
(mailto:malhar@crestdatasys.com) | www.crestdatasys.com
(http://www.crestdatasys.com) |
| DigitalOnUs | hashicorp@digitalonus.com
(mailto:hashicorp@digitalonus.com) | www.digitalonus.com
(http://www.digitalonus.com) |
| MustWin | bd@mustwin.com (mailto:bd@mustwin.com) | www.mustwin.com
(http://www.mustwin.com) |
| OpenCredo | hashicorp@opencredo.com
(mailto:hashicorp@opencredo.com) | www.opencredo.com
(http://www.opencredo.com) |

4. Review

During the review process, HashiCorp will provide feedback on the newly developed provider. **Please engage in the review process once one or two sample resources have been developed.** Begin the process by emailing `terraform-provider-dev@hashicorp.com` (`mailto:terraform-provider-dev@hashicorp.com`) with a URL to the public GitHub repo containing the code.

HashiCorp will then review the resource code, acceptance tests, and the documentation. When all the feedback has been addressed, support for the remaining resources can continue to be developed, along with the corresponding acceptance

tests and documentation.

The vendor is encouraged to send HashiCorp a rough list of resource names that are planned to be worked on along with the mapping to the underlying APIs, if possible. This information can be provided via the webform (<https://goo.gl/forms/iqfz6H9UK91X9LQp2>). It is preferred that the additional resources be developed and submitted as individual PRs in GitHub as that simplifies the review process.

Once the provider has been completed another email should be sent to terraform-provider-dev@hashicorp.com (<mailto:terraform-provider-dev@hashicorp.com>) along with a URL to the public GitHub repo containing the code requesting the final code review. HashiCorp will review the code and provide feedback about any changes that may be required. This is often an iterative process and can take some time to get done.

The vendor is also required to provide access credentials for the infrastructure (cloud or other) that is managed by the provider. Please encrypt the credentials using our public GPG key published at keybase.io/terraform (you can use the form at <https://keybase.io/encrypt#terraform> (<https://keybase.io/encrypt#terraform>)) and paste the encrypted message into the webform (<https://goo.gl/forms/iqfz6H9UK91X9LQp2>). Please do NOT enter plain-text credentials. These credentials are used during the review phase, as well as to test the provider as part of the regular testing HashiCorp conducts.

NOTE: It is strongly recommended to develop support for just one or two resources first and go through the review cycle before developing support for all the remaining resources. This approach helps catch any code construct issues early, and avoids the problem from multiplying across other resources. In addition, one of the common gaps is often the lack of a complete set of acceptance tests, which results in wasted time. It is recommended that you make an extra pass through the provider code and ensure that each resource has an acceptance test associated with it.

5. Release

At this stage, it is expected that the provider is fully developed, all tests and documentation are in place, the acceptance tests are all passing, and that HashiCorp has reviewed the provider.

HashiCorp will create a new GitHub repo under the `terraform-providers` GitHub organization for the new provider (example: `terraform-providers/terraform-provider-NAME`) and grant the owner of the original provider code write access to the new repo. A GitHub Pull Request should be created against this new repo with the provider code that had been reviewed in step-4 above. Once this is done HashiCorp will review and merge the PR, and get the new provider listed on [terraform.io](https://www.terraform.io) (<https://www.terraform.io>). This is also when the provider acceptance tests are added to the HashiCorp test harness (TeamCity) and tested at regular intervals.

Vendors whose providers are listed on [terraform.io](https://www.terraform.io) are permitted to use the HashiCorp Tested logo (`/assets/images/docs/hashicorp-tested-icon.png`) for their provider.

6. Support

Many vendors view the release step to be the end of the journey, while at HashiCorp we view it to be the start. Getting the provider built is just the first step in enabling users to use it against the infrastructure. Once this is done on-going effort is required to maintain the provider and address any issues in a timely manner.

The expectation is to resolve all critical issues within 48 hours and all other issues within 5 business days. HashiCorp Terraform has an extremely wide community of users and contributors and we encourage everyone to report issues however small, as well as help resolve them when possible.

Vendors who choose to not support their provider and prefer to make it a community supported provider will not be listed on [terraform.io](#).

Checklist

Below is an ordered checklist of steps that should be followed during the provider development process. This just reiterates the steps already documented in the section above.

- Fill out provider development program engagement webform (<https://goo.gl/forms/iqfz6H9UK91X9LQp2>)
- Refer to the example providers and model the new provider based on that
- Create the new provider with one or two sample resources along with acceptance tests and documentation
- Send email to terraform-provider-dev@hashicorp.com (<mailto:terraform-provider-dev@hashicorp.com>) to schedule an initial review
- Address review feedback and develop support for the other resources
- Send email to terraform-provider-dev@hashicorp.com (<mailto:terraform-provider-dev@hashicorp.com>) along with a pointer to the public GitHub repo containing the final code
- Provide HashiCorp with credentials for underlying infrastructure managed by the new provider via the webform (<https://goo.gl/forms/iqfz6H9UK91X9LQp2>)
- Address all review feedback, ensure that each resource has a corresponding acceptance test, and the documentation is complete
- Create a PR for the provider against the HashiCorp provided empty repo.
- Plan to continue supporting the provider with additional functionality as well as addressing any open issues.

Contact Us

For any questions or feedback please contact us at terraform-provider-dev@hashicorp.com (<mailto:terraform-provider-dev@hashicorp.com>).

Writing Custom Providers

This is an advanced guide! Following this guide is not required for regular use of Terraform and is only intended for advance users or Terraform contributors.

In Terraform, a "provider" is the logical abstraction of an upstream API. This guide details how to build a custom provider for Terraform.

Why?

There are a few possible reasons for authoring a custom Terraform provider, such as:

- An internal private cloud whose functionality is either proprietary or would not benefit the open source community.
- A "work in progress" provider being tested locally before contributing back.
- Extensions of an existing provider

Local Setup

Terraform supports a plugin model, and all providers are actually plugins. Plugins are distributed as Go binaries. Although technically possible to write a plugin in another language, almost all Terraform plugins are written in Go (<https://golang.org>). For more information on installing and configuring Go, please visit the Golang installation guide (<https://golang.org/doc/install>).

This post assumes familiarity with Golang and basic programming concepts.

As a reminder, all of Terraform's core providers are open source. When stuck or looking for examples, please feel free to reference the open source providers (<https://github.com/terraform-providers>) for help.

The Provider Schema

To start, create a file named `provider.go`. This is the root of the provider and should include the following boilerplate code:

```
package main

import (
    "github.com/hashicorp/terraform/helper/schema"
)

func Provider() *schema.Provider {
    return &schema.Provider{
        ResourcesMap: map[string]*schema.Resource{},
    }
}
```

The `helper/schema` (<https://godoc.org/github.com/hashicorp/terraform/helper/schema>) library is part of Terraform's core. It abstracts many of the complexities and ensures consistency between providers. The example above defines an empty provider (there are no *resources*).

The `*schema.Provider` type describes the provider's properties including:

- the configuration keys it accepts
- the resources it supports
- any callbacks to configure

Building the Plugin

Go requires a `main.go` file, which is the default executable when the binary is built. Since Terraform plugins are distributed as Go binaries, it is important to define this entry-point with the following code:

```
package main

import (
    "github.com/hashicorp/terraform/plugin"
    "github.com/hashicorp/terraform/terraform"
)

func main() {
    plugin.Serve(&plugin.ServeOpts{
        ProviderFunc: func() terraform.ResourceProvider {
            return Provider()
        },
    })
}
```

This establishes the main function to produce a valid, executable Go binary. The contents of the main function consume Terraform's plugin library. This library deals with all the communication between Terraform core and the plugin.

Next, build the plugin using the Go toolchain:

```
$ go build -o terraform-provider-example
```

The output name (`-o`) is **very important**. Terraform searches for plugins in the format of:

```
terraform-<TYPE>-<NAME>
```

In the case above, the plugin is of type "provider" and of name "example".

To verify things are working correctly, execute the binary just created:

```
$ ./terraform-provider-example
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically
```

This is the basic project structure and scaffolding for a Terraform plugin. To recap, the file structure is:

```
.  
└── main.go  
└── provider.go
```

Defining Resources

Terraform providers manage resources. A provider is an abstraction of an upstream API, and a resource is a component of that provider. As an example, the AWS provider supports `aws_instance` and `aws_elastic_ip`. DNSimple supports `dnsimple_record`. Fastly supports `fastly_service`. Let's add a resource to our fictitious provider.

As a general convention, Terraform providers put each resource in their own file, named after the resource, prefixed with `resource_`. To create an `example_server`, this would be `resource_server.go` by convention:

```
package main  
  
import (  
    "github.com/hashicorp/terraform/helper/schema"  
)  
  
func resourceServer() *schema.Resource {  
    return &schema.Resource{  
        Create: resourceServerCreate,  
        Read:   resourceServerRead,  
        Update: resourceServerUpdate,  
        Delete: resourceServerDelete,  
  
        Schema: map[string]*schema.Schema{  
            "address": &schema.Schema{  
                Type:     schema.TypeString,  
                Required: true,  
            },  
        },  
    }  
}
```

This uses the `schema.Resource` type (<https://godoc.org/github.com/hashicorp/terraform/helper/schema#Resource>). This structure defines the data schema and CRUD operations for the resource. Defining these properties are the only required thing to create a resource.

The schema above defines one element, "address", which is a required string. Terraform's schema automatically enforces validation and type casting.

Next there are four "fields" defined - Create, Read, Update, and Delete. The Create, Read, and Delete functions are required for a resource to be functional. There are other functions, but these are the only required ones. Terraform itself handles which function to call and with what data. Based on the schema and current state of the resource, Terraform can determine whether it needs to create a new resource, update an existing one, or destroy.

Each of the four struct fields point to a function. While it is technically possible to inline all functions in the resource schema, best practice dictates pulling each function into its own method. This optimizes for both testing and readability. Fill in those stubs now, paying close attention to method signatures.

```

func resourceServerCreate(d *schema.ResourceData, m interface{}) error {
    return nil
}

func resourceServerRead(d *schema.ResourceData, m interface{}) error {
    return nil
}

func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
    return nil
}

func resourceServerDelete(d *schema.ResourceData, m interface{}) error {
    return nil
}

```

Lastly, update the provider schema in `provider.go` to register this new resource.

```

func Provider() *schema.Provider {
    return &schema.Provider{
        ResourcesMap: map[string]*schema.Resource{
            "example_server": resourceServer(),
        },
    }
}

```

Build and test the plugin. Everything should compile as-is, although all operations are a no-op.

```

$ go build -o terraform-provider-example

$ ./terraform-provider-example
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically

```

The layout now looks like this:

```

.
├── main.go
└── provider.go
├── resource_server.go
└── terraform-provider-example

```

Invoking the Provider

Previous sections showed running the provider directly via the shell, which outputs a warning message like:

```

This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically

```

Terraform plugins should be executed by Terraform directly. To test this, create a `main.tf` in the working directory (the same place where the plugin exists).

```
resource "example_server" "my-server" {}
```

And execute `terraform plan`:

```
$ terraform plan

1 error(s) occurred:

* example_server.my-server: "address": required field is not set
```

This validates Terraform is correctly delegating work to our plugin and that our validation is working as intended. Fix the validation error by adding an address field to the resource:

```
resource "example_server" "my-server" {
  address = "1.2.3.4"
}
```

Execute `terraform plan` to verify the validation is passing:

```
$ terraform plan

+ example_server.my-server
  address: "1.2.3.4"

Plan: 1 to add, 0 to change, 0 to destroy.
```

It is possible to run `terraform apply`, but it will be a no-op because all of the resource options currently take no action.

Implement Create

Back in `resource_server.go`, implement the create functionality:

```
func resourceServerCreate(d *schema.ResourceData, m interface{}) error {
  address := d.Get("address").(string)
  d.SetId(address)
  return nil
}
```

This uses the `schema.ResourceData` API

(<https://godoc.org/github.com/hashicorp/terraform/helper/schema#ResourceData>) to get the value of "address" provided by the user in the Terraform configuration. Due to the way Go works, we have to typecast it to string. This is a safe operation, however, since our schema guarantees it will be a string type.

Next, it uses `SetId`, a built-in function, to set the ID of the resource to the address. The existence of a non-blank ID is what tells Terraform that a resource was created. This ID can be any string value, but should be a value that can be used to read the resource again.

Recompile the binary, then run `terraform plan` and `terraform apply`.

```
$ go build -o terraform-provider-example  
# ...
```

```
$ terraform plan  
  
+ example_server.my-server  
  address: "1.2.3.4"  
  
Plan: 1 to add, 0 to change, 0 to destroy.
```

```
$ terraform apply  
  
example_server.my-server: Creating...  
  address: "" => "1.2.3.4"  
example_server.my-server: Creation complete (ID: 1.2.3.4)  
  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Since the `Create` operation used `SetId`, Terraform believes the resource created successfully. Verify this by running `terraform plan`.

```
$ terraform plan  
Refreshing Terraform state in-memory prior to plan...  
The refreshed state will be used to calculate this plan, but will not be  
persisted to local or remote state storage.  
  
example_server.my-server: Refreshing state... (ID: 1.2.3.4)  
No changes. Infrastructure is up-to-date.  
  
This means that Terraform did not detect any differences between your  
configuration and real physical resources that exist. As a result, Terraform  
doesn't need to do anything.
```

Again, because of the call to `SetId`, Terraform believes the resource was created. When running `plan`, Terraform properly determines there are no changes to apply.

To verify this behavior, change the value of the `address` field and run `terraform plan` again. You should see output like this:

```
$ terraform plan  
example_server.my-server: Refreshing state... (ID: 1.2.3.4)  
  
~ example_server.my-server  
  address: "1.2.3.4" => "5.6.7.8"  
  
Plan: 0 to add, 1 to change, 0 to destroy.
```

Terraform detects the change and displays a diff with a `~` prefix, noting the resource will be modified in place, rather than created new.

Run `terraform apply` to apply the changes.

```
$ terraform apply
example_server.my-server: Refreshing state... (ID: 1.2.3.4)
example_server.my-server: Modifying... (ID: 1.2.3.4)
  address: "1.2.3.4" => "5.6.7.8"
example_server.my-server: Modifications complete (ID: 1.2.3.4)

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Since we did not implement the `Update` function, you would expect the `terraform plan` operation to report changes, but it does not! How were our changes persisted without the `Update` implementation?

Error Handling & Partial State

Previously our `Update` operation succeeded and persisted the new state with an empty function definition. Recall the current `update` function:

```
func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
    return nil
}
```

The `return nil` tells Terraform that the update operation succeeded without error. Terraform assumes this means any changes requested applied without error. Because of this, our state updated and Terraform believes there are no further changes.

To say it another way: if a callback returns no error, Terraform automatically assumes the entire diff successfully applied, merges the diff into the final state, and persists it.

Functions should *never* intentionally panic or call `os.Exit` - always return an error.

In reality, it is a bit more complicated than this. Imagine the scenario where our update function has to update two separate fields which require two separate API calls. What do we do if the first API call succeeds but the second fails? How do we properly tell Terraform to only persist half the diff? This is known as a *partial state* scenario, and implementing these properly is critical to a well-behaving provider.

Here are the rules for state updating in Terraform. Note that this mentions callbacks we have not discussed, for the sake of completeness.

- If the `Create` callback returns with or without an error without an ID set using `SetId`, the resource is assumed to not be created, and no state is saved.
- If the `Create` callback returns with or without an error and an ID has been set, the resource is assumed created and all state is saved with it. Repeating because it is important: if there is an error, but the ID is set, the state is fully saved.
- If the `Update` callback returns with or without an error, the full state is saved. If the ID becomes blank, the resource is destroyed (even within an update, though this shouldn't happen except in error scenarios).
- If the `Destroy` callback returns without an error, the resource is assumed to be destroyed, and all state is removed.
- If the `Destroy` callback returns with an error, the resource is assumed to still exist, and all prior state is preserved.
- If partial mode (covered next) is enabled when a create or update returns, only the explicitly enabled configuration

keys are persisted, resulting in a partial state.

Partial mode is a mode that can be enabled by a callback that tells Terraform that it is possible for partial state to occur. When this mode is enabled, the provider must explicitly tell Terraform what is safe to persist and what is not.

Here is an example of a partial mode with an update function:

```
func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
    // Enable partial state mode
    d.Partial(true)

    if d.HasChange("address") {
        // Try updating the address
        if err := updateAddress(d, m); err != nil {
            return err
        }

        d.SetPartial("address")
    }

    // If we were to return here, before disabling partial mode below,
    // then only the "address" field would be saved.

    // We succeeded, disable partial mode. This causes Terraform to save
    // all fields again.
    d.Partial(false)

    return nil
}
```

Note - this code will not compile since there is no `updateAddress` function. You can implement a dummy version of this function to play around with partial state. For this example, partial state does not mean much in this documentation example. If `updateAddress` were to fail, then the `address` field would not be updated.

Implementing Destroy

The `Destroy` callback is exactly what it sounds like - it is called to destroy the resource. This operation should never update any state on the resource. It is not necessary to call `d.SetId("")`, since any non-error return value assumes the resource was deleted successfully.

```
func resourceServerDelete(d *schema.ResourceData, m interface{}) error {
    // d.SetId("") is automatically called assuming delete returns no errors, but
    // it is added here for explicitness.
    d.SetId("")
    return nil
}
```

The `destroy` function should always handle the case where the resource might already be destroyed (manually, for example). If the resource is already destroyed, this should not return an error. This allows Terraform users to manually delete resources without breaking Terraform.

```
$ go build -o terraform-provider-example
```

Run `terraform destroy` to destroy the resource.

```
$ terraform destroy
Do you really want to destroy?
Terraform will delete all your managed infrastructure.
There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

example_server.my-server: Refreshing state... (ID: 5.6.7.8)
example_server.my-server: Destroying... (ID: 5.6.7.8)
example_server.my-server: Destruction complete

Destroy complete! Resources: 1 destroyed.
```

Implementing Read

The Read callback is used to sync the local state with the actual state (upstream). This is called at various points by Terraform and should be a read-only operation. This callback should never modify the real resource.

If the ID is updated to blank, this tells Terraform the resource no longer exists (maybe it was destroyed out of band). Just like the destroy callback, the Read function should gracefully handle this case.

```
func resourceServerRead(d *schema.ResourceData, m interface{}) error {
    client := m.(*MyClient)

    // Attempt to read from an upstream API
    obj, ok := client.Get(d.Id())

    // If the resource does not exist, inform Terraform. We want to immediately
    // return here to prevent further processing.
    if !ok {
        d.SetId("")
        return nil
    }

    d.Set("address", obj.Address)
    return nil
}
```

Next Steps

This guide covers the schema and structure for implementing a Terraform provider using the provider framework. As next steps, reference the internal providers for examples. Terraform also includes a full framework for testing providers.

General Rules

Dedicated Upstream Libraries

One of the biggest mistakes new users make is trying to conflate a client library with the Terraform implementation. Terraform should always consume an independent client library which implements the core logic for communicating with the upstream. Do not try to implement this type of logic in the provider itself.

Data Sources

While not explicitly discussed here, *data sources* are a special subset of resources which are read-only. They are resolved earlier than regular resources and can be used as part of Terraform's interpolation.

Introduction to Terraform

Welcome to the intro guide to Terraform! This guide is the best place to start with Terraform. We cover what Terraform is, what problems it can solve, how it compares to existing software, and contains a quick start for using Terraform.

If you are already familiar with the basics of Terraform, the documentation ([/docs/index.html](#)) provides a better reference guide for all available features as well as internals.

What is Terraform?

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

Examples work best to showcase Terraform. Please see the use cases ([/intro/use-cases.html](#)).

The key features of Terraform are:

Infrastructure as Code

Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.

Execution Plans

Terraform has a "planning" step where it generates an *execution plan*. The execution plan shows what Terraform will do when you call `apply`. This lets you avoid any surprises when Terraform manipulates infrastructure.

Resource Graph

Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources. Because of this, Terraform builds infrastructure as efficiently as possible, and operators get insight into dependencies in their infrastructure.

Change Automation

Complex changesets can be applied to your infrastructure with minimal human interaction. With the previously mentioned execution plan and resource graph, you know exactly what Terraform will change and in what order, avoiding many possible human errors.

Next Steps

See the page on Terraform use cases ([/intro/use-cases.html](#)) to see the multiple ways Terraform can be used. Then see how Terraform compares to other software ([/intro/vs/index.html](#)) to see how it fits into your existing infrastructure. Finally, continue onwards with the getting started guide ([/intro/getting-started/install.html](#)) to use Terraform to manage real infrastructure and to see how it works.

Example Configurations

The examples in this section illustrate some of the ways Terraform can be used.

All examples are ready to run as-is. Terraform will ask for input of things such as variables and API keys. If you want to continue using the example, you should save those parameters in a "terraform.tfvars" file or in a provider config block.

Warning! The examples use real providers that launch *real* resources. That means they can cost money to experiment with. To avoid unexpected charges, be sure to understand the price of resources before launching them, and verify any unneeded resources are cleaned up afterwards.

Experimenting in this way can help you learn how the Terraform lifecycle works, as well as how to repeatedly create and destroy infrastructure.

If you're completely new to Terraform, we recommend reading the getting started guide (</intro/getting-started/install.html>) before diving into the examples. However, due to the intuitive configuration Terraform uses it isn't required.

Examples

Our examples are distributed across several repos. This README file in the Terraform repo has links to all of them. (<https://github.com/hashicorp/terraform/tree/master/examples>)

To use these examples, Terraform must first be installed on your machine. You can install Terraform from the downloads page (</downloads.html>). Once installed, you can download, view, and run the examples.

To use an example, clone the repository that contains it and navigate to its directory. For example, to try the AWS two-tier architecture example:

```
git clone https://github.com/terraform-providers/terraform-provider-aws.git
cd terraform-provider-aws/examples/two-tier
```

You can then use your preferred code editor to browse and read the configurations. To try out an example, run Terraform's init and apply commands while in the example's directory:

```
$ terraform init
...
$ terraform apply
...
```

Terraform will interactively ask for variable input and potentially provider configuration, and will start executing.

When you're done with the example, run `terraform destroy` to clean up.

Two-Tier AWS Architecture

Example Source Code (<https://github.com/terraform-providers/terraform-provider-aws/tree/master/examples/two-tier>)

This provides a template for running a simple two-tier architecture on Amazon Web Services. The premise is that you have stateless app servers running behind an ELB serving traffic.

To simplify the example, it intentionally ignores deploying and getting your application onto the servers. However, you could do so either via provisioners (/docs/provisioners/index.html) and a configuration management tool, or by pre-baking configured AMIs with Packer (<https://www.packer.io>).

After you run `terraform apply` on this configuration, it will automatically output the DNS address of the ELB. After your instance registers, this should respond with the default Nginx web page.

As with all the examples, just copy and paste the example and run `terraform apply` to see it work.

Consul Example

Example Source Code (<https://github.com/terraform-providers/terraform-provider-consul/tree/master/examples/kv>)

Consul (<https://www.consul.io>) is a tool for service discovery, configuration and orchestration. The Key/Value store it provides is often used to store application configuration and information about the infrastructure necessary to process requests.

Terraform provides a Consul provider (</docs/providers/consul/index.html>) which can be used to interface with Consul from inside a Terraform configuration.

For our example, we use the Consul demo cluster (<https://demo.consul.io/>) to both read configuration and store information about a newly created EC2 instance. The size of the EC2 instance will be determined by the `tf_test/size` key in Consul, and will default to `m1.small` if that key does not exist. Once the instance is created the `tf_test/id` and `tf_test/public_dns` keys will be set with the computed values for the instance.

Before we run the example, use the Web UI (<https://demo.consul.io/ui/dc1/kv/>) to set the `tf_test/size` key to `t1.micro`. Once that is done, copy the configuration into a configuration file (`consul.tf` works fine). Either provide the AWS credentials as a default value in the configuration or invoke `apply` with the appropriate variables set.

Once the `apply` has completed, we can see the keys in Consul by visiting the Web UI (<https://demo.consul.io/ui/dc1/kv/>). We can see that the `tf_test/id` and `tf_test/public_dns` values have been set.

You can now tear down the infrastructure (</intro/getting-started/destroy.html>) Because we set the `delete` property of two of the Consul keys, Terraform will clean up those keys on destroy. We can verify this by using the Web UI.

This example has shown that Consul can be used with Terraform both to read existing values and to store generated results.

Inputs like AMI name, security groups, Puppet roles, bootstrap scripts, etc can all be loaded from Consul. This allows the specifics of an infrastructure to be decoupled from its overall architecture. This enables details to be changed without updating the Terraform configuration.

Outputs from Terraform can also be easily stored in Consul. One powerful feature this enables is using Consul for inventory management. If an application relies on ELB for routing, Terraform can update the application's configuration directly by setting the ELB address into Consul. Any resource attribute can be stored in Consul, allowing an operator to capture anything useful.

Count Example

Example Source Code (<https://github.com/terraform-providers/terraform-provider-aws/tree/master/examples/count>)

The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

Additionally, variables can be used to expand a list of resources for use elsewhere.

As with all the examples, just copy and paste the example and run `terraform apply` to see it work.

Local Value Configuration

Local values assign a name to an expression, that can then be used multiple times within a module.

Comparing modules to functions in a traditional programming language, if variables (/docs/configuration/variables.html) are analogous to function arguments and outputs (/docs/configuration/outputs.html) are analogous to function return values then *local values* are comparable to a function's local variables.

This page assumes you're already familiar with the configuration syntax (/docs/configuration/syntax.html).

Examples

Local values are defined in `locals` blocks:

```
# IDs for multiple sets of EC2 instances, merged together
locals {
    instance_ids = "${concat(aws_instance.blue.*.id, aws_instance.green.*.id)}"
}

# A computed default name prefix
locals {
    default_name_prefix = "${var.project_name}-web"
    name_prefix          = "${var.name_prefix != "" ? var.name_prefix : local.default_name_prefix}"
}

# Local values can be interpolated elsewhere using the "local." prefix.
resource "aws_s3_bucket" "files" {
    bucket = "${local.name_prefix}-files"
    # ...
}
```

Named local maps can be merged with local maps to implement common or default values:

```
# Define the common tags for all resources
locals {
    common_tags = {
        Component   = "awesome-app"
        Environment = "production"
    }
}

# Create a resource that blends the common tags with instance-specific tags.
resource "aws_instance" "server" {
    ami           = "ami-123456"
    instance_type = "t2.micro"

    tags = "${merge(
        local.common_tags,
        map(
            "Name", "awesome-app-server",
            "Role", "server"
        )
    )}"
}
```

Description

The `locals` block defines one or more local variables within a module. Each `locals` block can have as many locals as needed, and there can be any number of `locals` blocks within a module.

The names given for the items in the `locals` block must be unique throughout a module. The given value can be any expression that is valid within the current module.

The expression of a local value can refer to other locals, but as usual reference cycles are not allowed. That is, a local cannot refer to itself or to a variable that refers (directly or indirectly) back to it.

It's recommended to group together logically-related local values into a single block, particularly if they depend on each other. This will help the reader understand the relationships between variables. Conversely, prefer to define *unrelated* local values in *separate* blocks, and consider annotating each block with a comment describing any context common to all of the enclosed locals.

Module Configuration

Modules are used in Terraform to modularize and encapsulate groups of resources in your infrastructure. For more information on modules, see the dedicated modules section ([/docs/modules/index.html](#)).

This page assumes you're familiar with the configuration syntax ([/docs/configuration/syntax.html](#)) already.

Example

```
module "consul" {
  source  = "hashicorp/consul/aws"
  servers = 5
}
```

Description

A module block instructs Terraform to create an instance of a module, and in turn to instantiate any resources defined within it.

The name given in the block header is used to reference the particular module instance from expressions within the calling module, and to refer to the module on the command line. It has no meaning outside of a particular Terraform configuration.

Within the block body is the configuration for the module. All attributes within the block must correspond to variables ([/docs/configuration/variables.html](#)) within the module, with the exception of the following which Terraform treats as special:

- **source** - (Required) A module source ([/docs/modules/sources.html](#)) string specifying the location of the child module source code.
- **version** - (Optional) A version constraint ([/docs/modules/usage.html#module-versions](#)) string that specifies which versions of the referenced module are acceptable. The newest version matching the constraint will be used. **version** is supported only for modules retrieved from module registries.
- **providers** - (Optional) A map whose keys are provider configuration names that are expected by child module and whose values are corresponding provider names in the calling module. This allows provider configurations to be passed explicitly to child modules ([/docs/modules/usage.html#providers-within-modules](#)). If not specified, the child module inherits all of the default (un-aliased) provider configurations from the calling module.

Output Configuration

Outputs define values that will be highlighted to the user when Terraform applies, and can be queried easily using the output command ([/docs/commands/output.html](#)). Output usage is covered in more detail in the getting started guide ([/intro/getting-started/outputs.html](#)). This page covers configuration syntax for outputs.

Terraform knows a lot about the infrastructure it manages. Most resources have attributes associated with them, and outputs are a way to easily extract and query that information.

This page assumes you are familiar with the configuration syntax ([/docs/configuration/syntax.html](#)) already.

Example

A simple output configuration looks like the following:

```
output "address" {
  value = "${aws_instance.db.public_dns}"
}
```

This will output a string value corresponding to the public DNS address of the Terraform-defined AWS instance named "db". It is possible to export complex data types like maps and lists as well:

```
output "addresses" {
  value = ["${aws_instance.web.*.public_dns}"]
}
```

Description

The output block configures a single output variable. Multiple output variables can be configured with multiple output blocks. The NAME given to the output block is the name used to reference the output variable. It must conform to Terraform variable naming conventions if it is to be used as an input to other modules.

Within the block (the { }) is configuration for the output. These are the parameters that can be set:

- **value** (required) - The value of the output. This can be a string, list, or map. This usually includes an interpolation since outputs that are static aren't usually useful.
- **description** (optional) - A human-friendly description for the output. This is primarily for documentation for users using your Terraform configuration. A future version of Terraform will expose these descriptions as part of some Terraform CLI command.
- **depends_on** (list of strings) - Explicit dependencies that this output has. These dependencies will be created before this output value is processed. The dependencies are in the format of TYPE.NAME, for example aws_instance.web.
- **sensitive** (optional, boolean) - See below.

Syntax

The full syntax is:

```
output NAME {  
    value = VALUE  
}
```

Sensitive Outputs

Outputs can be marked as containing sensitive material by setting the `sensitive` attribute to `true`, like this:

```
output "sensitive" {  
    sensitive = true  
    value      = VALUE  
}
```

When outputs are displayed on-screen following a `terraform apply` or `terraform refresh`, sensitive outputs are redacted, with `<sensitive>` displayed in place of their value.

Limitations of Sensitive Outputs

- The values of sensitive outputs are still stored in the Terraform state, and available using the `terraform output` command, so cannot be relied on as a sole means of protecting values.
- Sensitivity is not tracked internally, so if the output is interpolated in another module into a resource, the value will be displayed.

Overrides

Terraform loads all configuration files within a directory and appends them together. Terraform also has a concept of *overrides*, a way to create files that are loaded last and *merged* into your configuration, rather than appended.

Overrides have a few use cases:

- Machines (tools) can create overrides to modify Terraform behavior without having to edit the Terraform configuration tailored to human readability.
- Temporary modifications can be made to Terraform configurations without having to modify the configuration itself.

Overrides names must be `override` or end in `_override`, excluding the extension. Examples of valid override files are `override.tf`, `override.tf.json`, `temp_override.tf`.

Override files are loaded last in alphabetical order.

Override files can be in Terraform syntax or JSON, just like non-override Terraform configurations.

Example

If you have a Terraform configuration `example.tf` with the contents:

```
resource "aws_instance" "web" {
  ami = "ami-408c7f28"
}
```

And you created a file `override.tf` with the contents:

```
resource "aws_instance" "web" {
  ami = "foo"
}
```

Then the AMI for the one resource will be replaced with "foo". Note that the override syntax can be Terraform syntax or JSON. You can mix and match syntaxes without issue.

Provider Configuration

Providers are responsible in Terraform for managing the lifecycle of a resource (/docs/configuration/resources.html): create, read, update, delete.

Most providers require some sort of configuration to provide authentication information, endpoint URLs, etc. Where explicit configuration is required, a provider block is used within the configuration as illustrated in the following sections.

By default, resources are matched with provider configurations by matching the start of the resource name. For example, a resource of type `vsphere_virtual_machine` is associated with a provider called `vsphere`.

This page assumes you're familiar with the configuration syntax (/docs/configuration/syntax.html) already.

Example

A provider configuration looks like the following:

```
provider "aws" {
  access_key = "foo"
  secret_key = "bar"
  region     = "us-east-1"
}
```

Description

A provider block represents a configuration for the provider named in its header. For example, provider `"aws"` above is a configuration for the `aws` provider.

Within the block body (between `{ }`) is configuration for the provider. The configuration is dependent on the type, and is documented for each provider (/docs/providers/index.html).

The arguments `alias` and `version`, if present, are special arguments handled by Terraform Core for their respective features described above. All other arguments are defined by the provider itself.

A provider block may be omitted if its body would be empty. Using a resource in configuration implicitly creates an empty provider configuration for it unless a provider block is explicitly provided.

Initialization

Each time a new provider is added to configuration -- either explicitly via a provider block or by adding a resource from that provider -- it's necessary to initialize that provider before use. Initialization downloads and installs the provider's plugin and prepares it to be used.

Provider initialization is one of the actions of `terraform init`. Running this command will download and initialize any providers that are not already initialized.

Providers downloaded by `terraform init` are only installed for the current working directory; other working directories can have their own installed provider versions.

Note that `terraform init` cannot automatically download providers that are not distributed by HashiCorp. See [Third-party Plugins](#) below for installation instructions.

For more information, see the `terraform init` command ([/docs/commands/init.html](#)).

Provider Versions

Providers are released on a separate rhythm from Terraform itself, and thus have their own version numbers. For production use, it is recommended to constrain the acceptable provider versions via configuration, to ensure that new versions with breaking changes will not be automatically installed by `terraform init` in future.

When `terraform init` is run *without* provider version constraints, it prints a suggested version constraint string for each provider:

```
The following providers do not have any version constraints in configuration,  
so the latest version was installed.
```

```
To prevent automatic upgrades to new major versions that may contain breaking  
changes, it is recommended to add version = "..." constraints to the  
corresponding provider blocks in configuration, with the constraint strings  
suggested below.
```

```
* provider.aws: version = "~> 1.0"
```

To constrain the provider version as suggested, add a `version` argument to the provider configuration block:

```
provider "aws" {  
  version = "~> 1.0"  
  
  access_key = "foo"  
  secret_key = "bar"  
  region     = "us-east-1"  
}
```

This special argument applies to *all* providers. `terraform providers` ([/docs/commands/providers.html](#)) can be used to view the specified version constraints for all providers used in the current configuration.

The `version` attribute value may either be a single explicit version or a version constraint expression. Constraint expressions use the following syntax to specify a *range* of versions that are acceptable:

- `>= 1.2.0`: version 1.2.0 or newer
- `<= 1.2.0`: version 1.2.0 or older
- `~> 1.2.0`: any non-beta version $\geq 1.2.0$ and $< 1.3.0$, e.g. 1.2.X
- `~> 1.2`: any non-beta version $\geq 1.2.0$ and $< 2.0.0$, e.g. 1.X.Y
- `>= 1.0.0, <= 2.0.0`: any version between 1.0.0 and 2.0.0 inclusive

When `terraform init` is re-run with providers already installed, it will use an already-installed provider that meets the constraints in preference to downloading a new version. To upgrade to the latest acceptable version of each provider, run `terraform init -upgrade`. This command also upgrades to the latest versions of all Terraform modules.

Multiple Provider Instances

You can define multiple configurations for the same provider in order to support multiple regions, multiple hosts, etc. The primary use case for this is using multiple cloud regions. Other use-cases include targeting multiple Docker hosts, multiple Consul hosts, etc.

To include multiple configurations for a given provider, include multiple provider blocks with the same provider name, but set the alias field to an instance name to use for each additional instance. For example:

```
# The default provider configuration
provider "aws" {
  # ...
}

# Additional provider configuration for west coast region
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
```

A provider block with out alias set is known as the *default* provider configuration. When alias is set, it creates an *additional* provider configuration. For providers that have no required configuration arguments, the implied *empty* configuration is also considered to be a *default* provider configuration.

Resources are normally associated with the default provider configuration inferred from the resource type name. For example, a resource of type aws_instance uses the *default* (un-aliased) aws provider configuration unless otherwise stated.

The provider argument within any resource or data block overrides this default behavior and allows an additional provider configuration to be selected using its alias:

```
resource "aws_instance" "foo" {
  provider = "aws.west"

  # ...
}
```

The value of the provider argument is always the provider name and an alias separated by a period, such as "aws.west" above.

Provider configurations may also be passed from a parent module into a child module, as described in *Providers within Modules* (/docs/modules/usage.html#providers-within-modules).

Interpolation

Provider configurations may use interpolation syntax (/docs/configuration/interpolation.html) to allow dynamic configuration:

```
provider "aws" {
  region = "${var.aws_region}"
}
```

Interpolation is supported only for the per-provider configuration arguments. It is not supported for the special `alias` and `version` arguments.

Although in principle it is possible to use any interpolation expression within a provider configuration argument, providers must be configurable to perform almost all operations within Terraform, and so it is not possible to use expressions whose value cannot be known until after configuration is applied, such as the `id` of a resource.

It is always valid to use input variables ([/docs/configuration/variables.html](#)) and data sources ([/docs/configuration/data-sources.html](#)) whose configurations do not in turn depend on as-yet-unknown values. Local values ([/docs/configuration/locals.html](#)) may also be used, but currently may cause errors when running `terraform destroy`.

Third-party Plugins

Anyone can develop and distribute their own Terraform providers. (See [Writing Custom Providers](#) ([/docs/extend/writing-custom-providers.html](#)) for more about provider development.) These third-party providers must be manually installed, since `terraform init` cannot automatically download them.

Install third-party providers by placing their plugin executables in the user plugins directory. The user plugins directory is in one of the following locations, depending on the host operating system:

| Operating system | User plugins directory |
|-------------------|--|
| Windows | <code>%APPDATA%\terraform.d\plugins</code> |
| All other systems | <code>~/.terraform.d/plugins</code> |

Once a plugin is installed, `terraform init` can initialize it normally.

Providers distributed by HashiCorp can also go in the user plugins directory. If a manually installed version meets the configuration's version constraints, Terraform will use it instead of downloading that provider. This is useful in airgapped environments and when testing pre-release provider builds.

Plugin Names and Versions

The naming scheme for provider plugins is `terraform-provider-<NAME>_vX.Y.Z`, and Terraform uses the name to understand the name and version of a particular provider binary.

If multiple versions of a plugin are installed, Terraform will use the newest version that meets the configuration's version constraints.

Third-party plugins are often distributed with an appropriate filename already set in the distribution archive, so that they can be extracted directly into the user plugins directory.

OS and Architecture Directories

Terraform plugins are compiled for a specific operating system and architecture, and any plugins in the root of the user plugins directory must be compiled for the current system.

If you use the same plugins directory on multiple systems, you can install plugins into subdirectories with a naming scheme of <OS>_<ARCH> (for example, darwin_amd64). Terraform uses plugins from the root of the plugins directory and from the subdirectory that corresponds to the current system, ignoring other subdirectories.

Terraform's OS and architecture strings are the standard ones used by the Go language. The following are the most common:

- darwin_amd64
- freebsd_386
- freebsd_amd64
- freebsd_arm
- linux_386
- linux_amd64
- linux_arm
- openbsd_386
- openbsd_amd64
- solaris_amd64
- windows_386
- windows_amd64

Provider Plugin Cache

By default, `terraform init` downloads plugins into a subdirectory of the working directory so that each working directory is self-contained. As a consequence, if you have multiple configurations that use the same provider then a separate copy of its plugin will be downloaded for each configuration.

Given that provider plugins can be quite large (on the order of hundreds of megabytes), this default behavior can be inconvenient for those with slow or metered Internet connections. Therefore Terraform optionally allows the use of a local directory as a shared plugin cache, which then allows each distinct plugin binary to be downloaded only once.

To enable the plugin cache, use the `plugin_cache_dir` setting in the CLI configuration file (<https://www.terraform.io/docs/commands/cli-config.html>). For example:

```
# (Note that the CLI configuration file is _not_ the same as the .tf files
# used to configure infrastructure.)

plugin_cache_dir = "$HOME/.terraform.d/plugin-cache"
```

This directory must already exist before Terraform will cache plugins; Terraform will not create the directory itself.

Please note that on Windows it is necessary to use forward slash separators (/) rather than the conventional backslash (\) since the configuration file parser considers a backslash to begin an escape sequence.

Setting this in the configuration file is the recommended approach for a persistent setting. Alternatively, the `TF_PLUGIN_CACHE_DIR` environment variable can be used to enable caching or to override an existing cache directory within a particular shell session:

```
export TF_PLUGIN_CACHE_DIR="$HOME/.terraform.d/plugin-cache"
```

When a plugin cache directory is enabled, the `terraform init` command will still access the plugin distribution server to obtain metadata about which plugins are available, but once a suitable version has been selected it will first check to see if the selected plugin is already available in the cache directory. If so, the already-downloaded plugin binary will be used.

If the selected plugin is not already in the cache, it will be downloaded into the cache first and then copied from there into the correct location under your current working directory.

When possible, Terraform will use hardlinks or symlinks to avoid storing a separate copy of a cached plugin in multiple directories. At present, this is not supported on Windows and instead a copy is always created.

The plugin cache directory must *not* be the third-party plugin directory or any other directory Terraform searches for pre-installed plugins, since the cache management logic conflicts with the normal plugin discovery logic when operating on the same directory.

Please note that Terraform will never itself delete a plugin from the plugin cache once it's been placed there. Over time, as plugins are upgraded, the cache directory may grow to contain several unused versions which must be manually deleted.

Resource Configuration

The most important thing you'll configure with Terraform are resources. Resources are a component of your infrastructure. It might be some low level component such as a physical server, virtual machine, or container. Or it can be a higher level component such as an email provider, DNS record, or database provider.

This page assumes you're familiar with the configuration syntax ([/docs/configuration/syntax.html](#)) already.

Example

A resource configuration looks like the following:

```
resource "aws_instance" "web" {  
  ami           = "ami-408c7f28"  
  instance_type = "t1.micro"  
}
```

Description

The resource block creates a resource of the given TYPE (first parameter) and NAME (second parameter). The combination of the type and name must be unique.

Within the block (the { }) is configuration for the resource. The configuration is dependent on the type, and is documented for each resource type in the providers section ([/docs/providers/index.html](#)).

Meta-parameters

There are **meta-parameters** available to all resources:

- **count** (int) - The number of identical resources to create. This doesn't apply to all resources. For details on using variables in conjunction with count, see [Using Variables with count](#) below.

Modules don't currently support the count parameter.

- **depends_on** (list of strings) - Explicit dependencies that this resource has. These dependencies will be created before this resource. For syntax and other details, see the section below on explicit dependencies.
- **provider** (string) - The name of a specific provider to use for this resource. The name is in the format of `TYPE.ALIAS`, for example, `aws.west`. Where `west` is set using the `alias` attribute in a provider. See [multiple provider instances](#).
- **lifecycle** (configuration block) - Customizes the lifecycle behavior of the resource. The specific options are documented below.

The `lifecycle` block allows the following keys to be set:

- **create_before_destroy** (bool) - This flag is used to ensure the replacement of a resource is created before the original instance is destroyed. As an example, this can be used to create a new DNS record before removing an old record.

- `prevent_destroy` (bool) - This flag provides extra protection against the destruction of a given resource. When this is set to `true`, any plan that includes a destroy of this resource will return an error message.
- `ignore_changes` (list of strings) - Customizes how diffs are evaluated for resources, allowing individual attributes to be ignored through changes. As an example, this can be used to ignore dynamic changes to the resource from external resources. Other meta-parameters cannot be ignored.

Ignored attribute names can be matched by their name, not state ID. For example, if an `aws_route_table` has two routes defined and the `ignore_changes` list contains "route", both routes will be ignored. Additionally you can also use a single entry with a wildcard (e.g. `"*"`) which will match all attribute names. Using a partial string together with a wildcard (e.g. `"rout*"`) is **not** supported.

Interpolations are not currently supported in the `lifecycle` configuration block (see issue #3116 (<https://github.com/hashicorp/terraform/issues/3116>))

Timeouts

Individual Resources may provide a `timeouts` block to enable users to configure the amount of time a specific operation is allowed to take before being considered an error. For example, the `aws_db_instance` (/docs/providers/aws/r/db_instance.html#timeouts) resource provides configurable timeouts for the `create`, `update`, and `delete` operations. Any Resource that provides Timeouts will document the default values for that operation, and users can overwrite them in their configuration.

Example overwriting the `create` and `delete` timeouts:

```
resource "aws_db_instance" "timeout_example" {
  allocated_storage = 10
  engine           = "mysql"
  engine_version   = "5.6.17"
  instance_class   = "db.t1.micro"
  name             = "mydb"

  # ...

  timeouts {
    create = "60m"
    delete = "2h"
  }
}
```

Individual Resources must opt-in to providing configurable Timeouts, and attempting to configure the timeout for a Resource that does not support Timeouts, or overwriting a specific action that the Resource does not specify as an option, will result in an error. Valid units of time are `s`, `m`, `h`.

Explicit Dependencies

Terraform ensures that dependencies are successfully created before a resource is created. During a destroy operation, Terraform ensures that this resource is destroyed before its dependencies.

A resource automatically depends on anything it references via interpolations ([/docs/configuration/interpolation.html](#)). The automatically determined dependencies are all that is needed most of the time. You can also use the `depends_on` parameter to explicitly define a list of additional dependencies.

The primary use case of explicit `depends_on` is to depend on a *side effect* of another operation. For example: if a provisioner creates a file, and your resource reads that file, then there is no interpolation reference for Terraform to automatically connect the two resources. However, there is a causal ordering that needs to be represented. This is an ideal case for `depends_on`. In most cases, however, `depends_on` should be avoided and Terraform should be allowed to determine dependencies automatically.

The syntax of `depends_on` is a list of resources and modules:

- Resources are `TYPE.NAME`, such as `aws_instance.web`.
- Modules are `module.NAME`, such as `module.foo`.

When a resource depends on a module, *everything* in that module must be created before the resource is created.

An example of a resource depending on both a module and resource is shown below. Note that `depends_on` can contain any number of dependencies:

```
resource "aws_instance" "web" {  
  depends_on = ["aws_instance.leader", "module.vpc"]  
}
```

Use sparingly! `depends_on` is rarely necessary. In almost every case, Terraform's automatic dependency system is the best-case scenario by having your resources depend only on what they explicitly use. Please think carefully before you use `depends_on` to determine if Terraform could automatically do this a better way.

Connection block

Within a resource, you can optionally have a **connection block**. Connection blocks describe to Terraform how to connect to the resource for provisioning ([/docs/provisioners/index.html](#)). This block doesn't need to be present if you're using only local provisioners, or if you're not provisioning at all.

Resources provide some data on their own, such as an IP address, but other data must be specified by the user.

The full list of settings that can be specified are listed on the provisioner connection page ([/docs/provisioners/connection.html](#)).

Provisioners

Within a resource, you can specify zero or more **provisioner blocks**. Provisioner blocks configure provisioners ([/docs/provisioners/index.html](#)).

Within the provisioner block is provisioner-specific configuration, much like resource-specific configuration.

Provisioner blocks can also contain a connection block (documented above). This connection block can be used to provide more specific connection info for a specific provisioner. An example use case might be to use a different user to log in for a single provisioner.

Using Variables With count

When declaring multiple instances of a resource using `count`, it is common to want each instance to have a different value for a given attribute.

You can use the `${count.index}` interpolation ([/docs/configuration/interpolation.html](#)) along with a map variable ([/docs/configuration/variables.html](#)) to accomplish this.

For example, here's how you could create three AWS Instances ([/docs/providers/aws/r/instance.html](#)) each with their own static IP address:

```
variable "instance_ips" {
  default = {
    "0" = "10.11.12.100"
    "1" = "10.11.12.101"
    "2" = "10.11.12.102"
  }
}

resource "aws_instance" "app" {
  count = "3"
  private_ip = "${lookup(var.instance_ips, count.index)}"
  # ...
}
```

To reference a particular instance of a resource you can use `resource.foo.*.id[#]` where `#` is the index number of the instance.

For example, to create a list of all AWS subnet ([/docs/providers/aws/r/subnet.html](#)) ids vs referencing a specific subnet in the list you can use this syntax:

```
resource "aws_vpc" "foo" {
  cidr_block = "198.18.0.0/16"
}

resource "aws_subnet" "bar" {
  count      = 2
  vpc_id     = "${aws_vpc.foo.id}"
  cidr_block = "${cidrsubnet(aws_vpc.foo.cidr_block, 8, count.index)}"
}

output "vpc_id" {
  value = "${aws_vpc.foo.id}"
}

output "all_subnet_ids" {
  value = "${aws_subnet.bar.*.id}"
}

output "subnet_id_0" {
  value = "${aws_subnet.bar.*.id[0]}"
}

output "subnet_id_1" {
  value = "${aws_subnet.bar.*.id[1]}"
}
```

Multiple Provider Instances

By default, a resource targets the provider based on its type. For example an `aws_instance` resource will target the "aws" provider. As of Terraform 0.5.0, a resource can target any provider by name.

The primary use case for this is to target a specific configuration of a provider that is configured multiple times to support multiple regions, etc.

To target another provider, set the `provider` field:

```
resource "aws_instance" "foo" {
  provider = "aws.west"

  # ...
}
```

The value of the field should be `TYPE` or `TYPE.ALIAS`. The `ALIAS` value comes from the `alias` field value when configuring the provider ([/docs/configuration/providers.html](#)).

```
provider "aws" {
  alias = "west"

  # ...
}
```

If no `provider` field is specified, the default provider is used.

Syntax

The full syntax is:

```
resource TYPE NAME {
  CONFIG ...
  [count = COUNT]
  [depends_on = [NAME, ...]]
  [provider = PROVIDER]

  [LIFECYCLE]

  [CONNECTION]
  [PROVISIONER ...]
}
```

where `CONFIG` is:

```
KEY = VALUE

KEY {
  CONFIG
}
```

where `LIFECYCLE` is:

```
lifecycle {  
    [create_before_destroy = true|false]  
    [prevent_destroy = true|false]  
    [ignore_changes = [ATTRIBUTE NAME, ...]]  
}
```

where CONNECTION is:

```
connection {  
    KEY = VALUE  
    ...  
}
```

where PROVISIONER is:

```
provisioner NAME {  
    CONFIG ...  
  
    [when = "create"|"destroy"]  
    [on_failure = "continue"|"fail"]  
  
    [CONNECTION]  
}
```

Configuration Syntax

The syntax of Terraform configurations is called HashiCorp Configuration Language (HCL) (<https://github.com/hashicorp/hcl>). It is meant to strike a balance between human readable and editable as well as being machine-friendly. For machine-friendliness, Terraform can also read JSON configurations. For general Terraform configurations, however, we recommend using the HCL Terraform syntax.

Terraform Syntax

Here is an example of Terraform's HCL syntax:

```
# An AMI
variable "ami" {
  description = "the AMI to use"
}

/* A multi
line comment. */
resource "aws_instance" "web" {
  ami           = "${var.ami}"
  count         = 2
  source_dest_check = false

  connection {
    user = "root"
  }
}
```

Basic bullet point reference:

- Single line comments start with #
- Multi-line comments are wrapped with /* and */
- Values are assigned with the syntax of key = value (whitespace doesn't matter). The value can be any primitive (string, number, boolean), a list, or a map.
- Strings are in double-quotes.
- Strings can interpolate other values using syntax wrapped in \${}, such as \${var.foo}. The full syntax for interpolation is documented here ([/docs/configuration/interpolation.html](#)).
- Multiline strings can use shell-style "here doc" syntax, with the string starting with a marker like <<EOF and then the string ending with EOF on a line of its own. The lines of the string and the end marker must *not* be indented.
- Numbers are assumed to be base 10. If you prefix a number with 0x, it is treated as a hexadecimal number.
- Boolean values: true, false.
- Lists of primitive types can be made with square brackets ([]). Example: ["foo", "bar", "baz"].
- Maps can be made with braces ({ }) and colons (:): { "foo": "bar", "bar": "baz" }. Quotes may be omitted on keys, unless the key starts with a number, in which case quotes are required. Commas are required between key/value pairs for single line maps. A newline between key/value pairs is sufficient in multi-line maps.

In addition to the basics, the syntax supports hierarchies of sections, such as the "resource" and "variable" in the example above. These sections are similar to maps, but visually look better. For example, these are nearly equivalent:

```
variable "ami" {
  description = "the AMI to use"
}
```

is equal to:

```
variable = [{
  "ami": {
    "description": "the AMI to use",
  }
}]
```

Notice how the top stanza visually looks a lot better? By repeating multiple `variable` sections, it builds up the `variable` list. When possible, use sections since they're visually clearer and more readable.

JSON Syntax

Terraform also supports reading JSON formatted configuration files. The above example converted to JSON:

```
{
  "variable": {
    "ami": {
      "description": "the AMI to use"
    }
  },
  "resource": {
    "aws_instance": {
      "web": {
        "ami": "${var.ami}",
        "count": 2,
        "source_dest_check": false,

        "connection": {
          "user": "root"
        }
      }
    }
  }
}
```

The conversion should be pretty straightforward and self-documented.

The downsides of JSON are less human readability and the lack of comments. Otherwise, the two are completely interoperable.

Terraform Push Configuration

Important: The `terraform push` command is deprecated, and only works with the legacy version of Terraform Enterprise (/docs/enterprise-legacy/index.html). In the current version of Terraform Enterprise, you can upload configurations using the API. See the docs about API-driven runs (/docs/enterprise/run/api.html) for more details.

The `terraform push` command (/docs/commands/push.html) uploads a configuration to a Terraform Enterprise (legacy) environment. The name of the environment (and the organization it's in) can be specified on the command line, or as part of the Terraform configuration in an `atlas` block.

The `atlas` block does not configure remote state; it only configures the push command. For remote state, use a `terraform { backend "<NAME>" { ... } }` block (/docs/backends/config.html).

This page assumes you're familiar with the configuration syntax (/docs/configuration/syntax.html) already.

Example

Terraform push configuration looks like the following:

```
atlas {  
  name = "mitchellh/production-example"  
}
```

Why is this called "atlas"? Atlas was previously a commercial offering from HashiCorp that included a full suite of enterprise products. The products have since been broken apart into their individual products, like **Terraform Enterprise**. While this transition is in progress, you may see references to "atlas" in the documentation. We apologize for the inconvenience.

Description

The `atlas` block configures the settings when Terraform is pushed (/docs/commands/push.html) to Terraform Enterprise. Only one `atlas` block is allowed.

Within the block (the `{ }`) is configuration for Atlas uploading. No keys are required, but the key typically set is `name`.

No value within the atlas block can use interpolations. Due to the nature of this configuration, interpolations are not possible. If you want to parameterize these settings, use the `Atlas` block to set defaults, then use the command-line flags of the `push` command (/docs/commands/push.html) to override.

Syntax

The full syntax is:

```
atlas {  
    name = VALUE  
}
```

Terraform Configuration

The `terraform` configuration section is used to configure Terraform itself, such as requiring a minimum Terraform version to execute a configuration.

This page assumes you're familiar with the configuration syntax ([/docs/configuration/syntax.html](#)) already.

Example

Terraform configuration looks like the following:

```
terraform {  
  required_version = "> 0.7.0"  
}
```

Description

The `terraform` block configures the behavior of Terraform itself.

The currently only allowed configurations within this block are `required_version` and `backend`.

`required_version` specifies a set of version constraints that must be met to perform operations on this configuration. If the running Terraform version doesn't meet these constraints, an error is shown. See the section below dedicated to this option.

See [backends](#) ([/docs/backends/index.html](#)) for more detail on the backend configuration.

No value within the `terraform` block can use interpolations. The `terraform` block is loaded very early in the execution of Terraform and interpolations are not yet available.

Specifying a Required Terraform Version

The `required_version` setting can be used to require a specific version of Terraform. If the running version of Terraform doesn't match the constraints specified, Terraform will show an error and exit.

When modules ([/docs/configuration/modules.html](#)) are used, all Terraform version requirements specified by the complete module tree must be satisfied. This means that the `required_version` setting can be used by a module to require that all consumers of a module also use a specific version.

The value of this configuration is a comma-separated list of constraints. A constraint is an operator followed by a version, such as `> 0.7.0`. Constraints support the following operations:

- `=` (or no operator): exact version equality
- `!=`: version not equal
- `>`, `>=`, `<`, `<=`: version comparison, where "greater than" is a larger version number
- `~>`: pessimistic constraint operator. Example: for `~> 0.9`, this means `>= 0.9, < 1.0`. Example: for `~> 0.8.4`, this means `>= 0.8.4, < 0.9`

For modules, a minimum version is recommended, such as `> 0.8.0`. This minimum version ensures that a module operates as expected, but gives the consumer flexibility to use newer versions.

Syntax

The full syntax is:

```
terraform {  
    required_version = VALUE  
}
```

Input Variable Configuration

Input variables serve as parameters for a Terraform module.

When used in the root module of a configuration, variables can be set from CLI arguments and environment variables. For *child* modules ([/docs/configuration/modules.html](#)), they allow values to pass from parent to child.

Input variable usage is introduced in the Getting Started guide section *Input Variables* ([/intro/getting-started/variables.html](#)).

This page assumes you're familiar with the configuration syntax ([/docs/configuration/syntax.html](#)) already.

Example

Input variables can be defined as follows:

```
variable "key" {
  type = "string"
}

variable "images" {
  type = "map"

  default = {
    us-east-1 = "image-1234"
    us-west-2 = "image-4567"
  }
}

variable "zones" {
  default = ["us-east-1a", "us-east-1b"]
}
```

Description

The `variable` block configures a single input variable for a Terraform module. Each block declares a single variable.

The name given in the block header is used to assign a value to the variable via the CLI and to reference the variable elsewhere in the configuration.

Within the block body (between `{ }`) is configuration for the variable, which accepts the following arguments:

- `type` (Optional) - If set this defines the type of the variable. Valid values are `string`, `list`, and `map`. If this field is omitted, the variable type will be inferred based on `default`. If no `default` is provided, the type is assumed to be `string`.
- `default` (Optional) - This sets a default value for the variable. If no `default` is provided, Terraform will raise an error if a value is not provided by the caller. The default value can be of any of the supported data types, as described below. If `type` is also set, the given value must be of the specified type.
- `description` (Optional) - A human-friendly description for the variable. This is primarily for documentation for users using your Terraform configuration. When a module is published in Terraform Registry (<https://registry.terraform.io/>), the given description is shown as part of the documentation.

The name of a variable can be any valid identifier. However, due to the interpretation of module configuration blocks (/docs/configuration/modules.html), the names `source`, `version` and `providers` are reserved for Terraform's own use and are thus not recommended for any module intended to be used as a child module.

The default value of an input variable must be a *literal* value, containing no interpolation expressions. To assign a name to an expression so that it may be re-used within a module, use Local Values (/docs/configuration/locals.html) instead.

Strings

String values are simple and represent a basic key to value mapping where the key is the variable name. An example is:

```
variable "key" {
  type    = "string"
  default = "value"
}
```

A multi-line string value can be provided using heredoc syntax.

```
variable "long_key" {
  type = "string"
  default = <<EOF
This is a long key.
Running over several lines.
EOF
}
```

Terraform performs automatic conversion from string values to numeric and boolean values based on context, so in practice string variables may be used to set arguments of any primitive type. For boolean values in particular there are some caveats, described under *Booleans* below.

Maps

A map value is a lookup table from string keys to string values. This is useful for selecting a value based on some other provided value.

A common use of maps is to create a table of machine images per region, as follows:

```
variable "images" {
  type    = "map"
  default = {
    "us-east-1" = "image-1234"
    "us-west-2" = "image-4567"
  }
}
```

Lists

A list value is an ordered sequence of strings indexed by integers starting with zero. For example:

```
variable "users" {
  type    = "list"
  default = ["admin", "ubuntu"]
}
```

Booleans

Although Terraform can automatically convert between boolean and string values, there are some subtle implications of these conversions that should be completely understood when using boolean values with input variables.

It is recommended for now to specify boolean values for variables as the strings "true" and "false", to avoid some caveats in the conversion process. A future version of Terraform will properly support boolean values and so relying on the current behavior could result in backwards-incompatibilities at that time.

For a configuration such as the following:

```
variable "active" {
  default = false
}
```

The `false` is converted to a string "`0`" when running Terraform.

Then, depending on where you specify overrides, the behavior can differ:

- Variables with boolean values in a `tfvars` file will likewise be converted to "`0`" and "`1`" values.
- Variables specified via the `-var` command line flag will be literal strings "true" and "false", so care should be taken to explicitly use "`0`" or "`1`".
- Variables specified with the `TF_VAR_` environment variables will be literal string values, just like `-var`.

A future version of Terraform will fully support first-class boolean types which will make the behavior of booleans consistent as you would expect. This may break some of the above behavior.

When passing boolean-like variables as parameters to resource configurations that expect boolean values, they are converted consistently:

- "`1`" and "true" become `true`
- "`0`" and "false" become `false`

The behavior of conversion in *this* direction (string to boolean) will *not* change in future Terraform versions. Therefore, using these string values rather than literal booleans is recommended when using input variables.

Environment Variables

Environment variables can be used to set the value of an input variable in the root module. The name of the environment variable must be `TF_VAR_` followed by the variable name, and the value is the value of the variable.

For example, given the configuration below:

```
variable "image" {}
```

The variable can be set via an environment variable:

```
$ TF_VAR_image=foo terraform apply
```

Maps and lists can be specified using environment variables as well using HCL ([/docs/configuration/syntax.html#HCL](#)) syntax in the value.

For a list variable like so:

```
variable "somelist" {
  type = "list"
}
```

The variable could be set like so:

```
$ TF_VAR_somelist='["ami-abc123", "ami-bcd234"]' terraform plan
```

Similarly, for a map declared like:

```
variable "somedmap" {
  type = "map"
}
```

The value can be set like this:

```
$ TF_VAR_somedmap='{foo = "bar", baz = "qux"}' terraform plan
```

Variable Files

Values for the input variables of a root module can be gathered in *variable definition files* and passed together using the `-var-file=FILE` option.

For all files which match `terraform.tfvars` or `*.auto.tfvars` present in the current directory, Terraform automatically loads them to populate variables. If the file is located somewhere else, you can pass the path to the file using the `-var-file` flag. It is recommended to name such files with names ending in `.tfvars`.

Variables files use HCL or JSON syntax to define variable values. Strings, lists or maps may be set in the same manner as the default value in a variable block in Terraform configuration. For example:

```
foo = "bar"
xyz = "abc"

somelist = [
  "one",
  "two",
]

somedmap = {
  foo = "bar"
  bax = "qux"
}
```

The `-var-file` flag can be used multiple times per command invocation:

```
$ terraform apply -var-file=foo.tfvars -var-file=bar.tfvars
```

Note: Variable files are evaluated in the order in which they are specified on the command line. If a particular variable is defined in more than one variable file, the last value specified is effective.

Variable Merging

When multiple values are provided for the same input variable, map values are merged while all other values are overridden by the last definition.

For example, if you define a variable twice on the command line:

```
$ terraform apply -var foo=bar -var foo=baz
```

Then the value of `foo` will be `baz`, since it was the last definition seen.

However, for maps, the values are merged:

```
$ terraform apply -var 'foo={quux="bar"}' -var 'foo={bar="baz"}'
```

The resulting value of `foo` will be:

```
{
  quux = "bar"
  bar = "baz"
}
```

There is no way currently to unset map values in Terraform. Whenever a map is modified either via variable input or being passed into a module, the values are always merged.

Variable Precedence

Both these files have the variable `baz` defined:

foo.tfvars

```
baz = "foo"
```

bar.tfvars

```
baz = "bar"
```

When they are passed in the following order:

```
$ terraform apply -var-file=foo.tfvars -var-file=bar.tfvars
```

The result will be that baz will contain the value bar because bar.tfvars has the last definition loaded.

Definition files passed using the `-var-file` flag will always be evaluated after those in the working directory.

Values passed within definition files or with `-var` will take precedence over `TF_VAR_` environment variables, as environment variables are considered defaults.

Terraform Enterprise

This is the documentation for Terraform Enterprise (TFE). Terraform Enterprise (<https://www.hashicorp.com/products/terraform/>) is a product to make it easier for teams to collaborate and govern Terraform changes.

If you are new to TFE, begin with the Getting Started Guide (</docs/enterprise/getting-started/access.html>).

Terraform Enterprise Features

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

Terraform Enterprise (<https://www.hashicorp.com/products/terraform/>) is a tool for safely and efficiently changing infrastructure across providers.

This is a list of features specific to Terraform Enterprise.

- Terraform Plans and Applies (/docs/enterprise-legacy/runs)
- Terraform Artifact Registry (/docs/enterprise-legacy/artifacts)
- Terraform Remote State Storage (/docs/enterprise-legacy/state)
- Terraform Run Notifications (/docs/enterprise-legacy/runs/notifications.html)

Terraform Enterprise Features

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

Terraform Enterprise (<https://www.hashicorp.com/products/terraform/>) is a tool for safely and efficiently changing infrastructure across providers.

This is a list of features specific to Terraform Enterprise.

- Terraform Plans and Applies (/docs/enterprise-legacy/runs)
- Terraform Artifact Registry (/docs/enterprise-legacy/artifacts)
- Terraform Remote State Storage (/docs/enterprise-legacy/state)
- Terraform Run Notifications (/docs/enterprise-legacy/runs/notifications.html)

Terraform Enterprise API Documentation

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise provides an API for a **subset of features** available. For questions or requests for new API features please email support@hashicorp.com (mailto:support@hashicorp.com).

The list of available endpoints are on the navigation.

Authentication

All requests must be authenticated with an X-Atlas-Token HTTP header. This token can be generated or revoked on the account tokens page. Your token will have access to all resources your account has access to.

For organization level resources, we recommend creating a separate user account that can be added to the organization with the specific privilege level required.

Response Codes

Standard HTTP response codes are returned. 404 Not Found codes are returned for all resources that a user does not have access to, as well as for resources that don't exist. This is done to avoid a potential attacker discovering the existence of a resource.

Errors

Errors are returned in JSON format:

```
{  
  "errors": {  
    "name": [  
      "has already been taken"  
    ]  
  }  
}
```

Versioning

The API currently resides under the /v1 prefix. Future APIs will increment this version leaving the /v1 API intact, though in the future certain features may be deprecated. In that case, ample notice to migrate to the new API will be provided.

Content Type

The API accepts namespaced attributes in either JSON or application/x-www-form-urlencoded. We recommend using JSON, but for simplicity form style requests are supported.

Configuration API

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

A configuration version represents versions of Terraform configuration. Each set of changes to Terraform HCL files or the scripts used in the files should have an associated configuration version.

When creating versions via the API, the variables attribute can be sent to include the necessary variables for the Terraform configuration. A configuration represents settings associated with a resource that runs Terraform with versions of Terraform configuration. Configurations have many configuration versions which represent versions of Terraform configuration templates and other associated configuration. Most operations take place on the configuration version, not the configuration.

Get Latest Configuration Version

This endpoint gets the latest configuration version.

| Method | Path |
|--------|---|
| GET | /terraform/configurations/:username/:name/versions/latest |

Parameters

- `:username` (string: <required>) - Specifies the username or organization name under which to get the latest configuration version. This username must already exist in the system, and the user must have permission to create new configuration versions under this namespace. This is specified as part of the URL.
- `:name` (string: <required>) - Specifies the name of the configuration for which to get the latest configuration. This is specified as part of the URL.

Sample Request

```
$ curl \
--header "X-Atlas-Token: ..." \
https://atlas.hashicorp.com/api/v1/terraform/configurations/my-organization/my-configuration/versions
/latest
```

Sample Response

```
{  
  "version": {  
    "version": 6,  
    "metadata": {  
      "foo": "bar"  
    },  
    "tf_vars": [],  
    "variables": {}  
  }  
}
```

- `version` (`int`) - the unique version instance number.
- `metadata` (`map<string|string>`) - a map of arbitrary metadata for this version.

Create Configuration Version

This endpoint creates a new configuration version.

| Method | Path |
|--------|--|
| POST | /terraform/configurations/:username/:name/versions |

Parameters

- `:username` (`string: <required>`) - Specifies the username or organization name under which to create this configuration version. This username must already exist in the system, and the user must have permission to create new configuration versions under this namespace. This is specified as part of the URL.
- `:name` (`string: <required>`) - Specifies the name of the configuration for which to create a new version. This is specified as part of the URL.
- `metadata` (`map<string|string>`) - Specifies an arbitrary hash of key-value metadata pairs. This is specified as the payload as JSON.
- `variables` (`map<string|string>`) - Specifies a hash of key-value pairs that will be made available as variables to this version.

Sample Payload

```
{  
  "version": {  
    "metadata": {  
      "git_branch": "master",  
      "remote_type": "atlas",  
      "remote_slug": "hashicorp/atlas"  
    },  
    "variables": {  
      "ami_id": "ami-123456",  
      "target_region": "us-east-1",  
      "consul_count": "5",  
      "consul_ami": "ami-123456"  
    }  
  }  
}
```

Sample Request

```
$ curl \  
  --request POST \  
  --header "X-Atlas-Token: ..." \  
  --header "Content-Type: application/json" \  
  --data @payload.json \  
  https://atlas.hashicorp.com/api/v1/terraform/configurations/my-organization/my-configuration/versions
```

Sample Response

```
{  
  "version": 6,  
  "upload_path": "https://binstore.hashicorp.com/ddbd7db6-f96c-4633-beb6-22fe2d74eed",  
  "token": "ddbd7db6-f96c-4633-beb6-22fe2d74eed"  
}
```

- **version (int)** - the unique version instance number. This is auto-incrementing.
- **upload_path (string)** - the path where the archive should be uploaded via a `POST` request.
- **token (string)** - the token that should be used when uploading the archive to the `upload_path`.

Check Upload Progress

This endpoint retrieves the progress for an upload of a configuration version.

| Method | Path |
|--------|------|
|--------|------|

| | |
|-----|---|
| GET | /terraform/configurations/:username/:name/versions/:progress/:token |
|-----|---|

Parameters

- `:username` (`string: <required>`) - Specifies the username or organization to read progress. This is specified as part of the URL.
- `:name` (`string: <required>`) - Specifies the name of the configuration for to read progress. This is specified as part of the URL.
- `:token` (`string: <required>`) - Specifies the token that was returned from the create option. **This is not an Atlas Token!** This is specified as part of the URL.

Sample Request

```
$ curl \  
  --header "X-Atlas-Token: ..." \  
  https://atlas.hashicorp.com/api/v1/terraform/configurations/my-organization/my-configuration/versions  
/progress/ddbd7db6-f96c-4633-beb6-22fe2d74eed
```

Sample Response

Environments API

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Environments represent running infrastructure managed by Terraform.

Environments can also be connected to Consul clusters. This documentation covers the environment interactions with Terraform.

Get Latest Configuration Version

This endpoint updates the Terraform variables for an environment. Due to the sensitive nature of variables, they are not returned on success.

| Method | Path |
|--------|---|
| PUT | /environments/:username/:name/variables |

Parameters

- `:username` (string: <required>) - Specifies the username or organization name under which to update variables. This username must already exist in the system, and the user must have permission to create new configuration versions under this namespace. This is specified as part of the URL.
- `:name` (string: <required>) - Specifies the name of the environment for which to update variables. This is specified as part of the URL.
- `variables` (map<string|string>) - Specifies a key-value map of Terraform variables to be updated. Existing variables will only be removed when their value is empty. Variables of the same key will be overwritten.

Note: Only string variables can be updated via the API currently. Creating or updating HCL variables is not yet supported.

Sample Payload

```
{  
  "variables": {  
    "desired_capacity": "15",  
    "foo": "bar"  
  }  
}
```

Sample Request

```
$ curl \  
  --header "X-Atlas-Token: ..." \  
  --header "Content-Type: application/json" \  
  --request PUT \  
  --data @payload.json \  
  https://atlas.hashicorp.com/api/v1/environments/my-organization/my-environment/variables
```

Sample Response

(empty body)

Runs API

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Runs in Terraform Enterprise represents a two step Terraform plan and a subsequent apply.

Runs are queued under environments (/docs/enterprise-legacy/api/environments.html) and require a two-step confirmation workflow. However, environments can be configured to auto-apply to avoid this.

Queue Run

Starts a new run (plan) in the environment. Requires a configuration version to be present on the environment to succeed, but will otherwise 404.

| Method | Path |
|--------|------------------------------------|
| POST | /environments/:username/:name/plan |

Parameters

- `:username` (string: <required>) - Specifies the username or organization name under which to get the latest configuration version. This username must already exist in the system, and the user must have permission to create new configuration versions under this namespace. This is specified as part of the URL.
- `:name` (string: <required>) - Specifies the name of the configuration for which to get the latest configuration. This is specified as part of the URL.
- `destroy` (bool: false) - Specifies if the plan should be a destroy plan.

Sample Payload

```
{  
  "destroy": false  
}
```

Sample Request

```
$ curl \  
  --request POST \  
  --header "X-Atlas-Token: ..." \  
  --header "Content-Type: application/json" \  
  --data @payload.json \  
  https://atlas.hashicorp.com/api/v1/environments/my-organization/my-environment/plan
```

Sample Response

```
{  
  "success": true  
}
```

State API

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

State represents the status of your infrastructure at the last time Terraform was run. States can be pushed to Terraform Enterprise from Terraform's CLI after an apply is done locally, or state is automatically stored if the apply is done in Terraform Enterprise.

List of States

This endpoint gets a list of states accessible to the user corresponding to the provided token.

| Method | Path |
|--------|------------------|
| GET | /terraform/state |

Parameters

- ?username (string: "") - Specifies the organization/username to filter states
- ?page (int: 1) - Specifies the pagination, which defaults to page 1.

Sample Requests

```
$ curl \  
  --header "X-Atlas-Token: ..." \  
  https://atlas.hashicorp.com/api/v1/terraform/state
```

```
$ curl \  
  --header "X-Atlas-Token: ..." \  
  https://atlas.hashicorp.com/api/v1/terraform/state?username=acme
```

Sample Response

```
{  
  "states": [  
    {  
      "updated_at": "2017-02-03T19:52:37.693Z",  
      "environment": {  
        "username": "my-organization",  
        "name": "docs-demo-one"  
      }  
    },  
    {  
      "updated_at": "2017-04-06T15:48:49.677Z",  
      "environment": {  
        "username": "my-organization",  
        "name": "docs-demo-two"  
      }  
    }  
  ]  
}
```

Users API

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Users are both users and organizations in Terraform Enterprise. They are the parent resource of all resources.

Currently, only the retrieval of users is available on the API. Additionally, only Vagrant box resources will be listed. Boxes will be returned based on permissions over the organization, or user.

Read User

This endpoint retrieves information about a single user.

| Method | Path |
|--------|-----------------|
| GET | /user/:username |

Parameters

- `:username` (string: <required>) - Specifies the username to search. This is specified as part of the URL.

Sample Request

```
$ curl \
--header "X-Atlas-Token: ..." \
https://atlas.hashicorp.com/api/v1/user/my-user
```

Sample Response

```
{
  "username": "sally-seashell",
  "avatar_url": "https://www.gravatar.com/avatar/...",
  "profile_html": "Sally is...",
  "profile_markdown": "Sally is...",
  "boxes": []
}
```

About Terraform Artifacts

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Terraform Enterprise can be used to store artifacts for use by Terraform. Typically, artifacts are stored with Packer (<https://packer.io/docs>).

Artifacts can be used in to deploy and manage images of configuration. Artifacts are generic, but can be of varying types like `amazon.image`. See the Packer `artifact_type` (https://packer.io/docs/post-processors/atlas.html#artifact_type) docs for more information.

Packer can create artifacts both while running in and out of Terraform Enterprise network. This is possible due to the post-processors use of the public artifact API to store the artifacts.

Cross Provider Example

Example Source Code (<https://github.com/hashicorp/terraform/tree/master/examples/cross-provider>)

This is a simple example of the cross-provider capabilities of Terraform.

This creates a Heroku application and points a DNS CNAME record at the result via DNSimple. A host query to the outputted hostname should reveal the correct DNS configuration.

As with all the examples, just copy and paste the example and run `terraform apply` to see it work.

Build Infrastructure

With Terraform installed, let's dive right into it and start creating some infrastructure.

We'll build infrastructure on AWS (<https://aws.amazon.com>) for the getting started guide since it is popular and generally understood, but Terraform can manage many providers (/docs/providers/index.html), including multiple providers in a single configuration. Some examples of this are in the use cases section (/intro/use-cases.html).

If you don't have an AWS account, create one now (<https://aws.amazon.com/free/>). For the getting started guide, we'll only be using resources which qualify under the AWS free-tier (<https://aws.amazon.com/free/>), meaning it will be free. If you already have an AWS account, you may be charged some amount of money, but it shouldn't be more than a few dollars at most.

Warning! If you're not using an account that qualifies under the AWS free-tier (<https://aws.amazon.com/free/>), you may be charged to run these examples. The most you should be charged should only be a few dollars, but we're not responsible for any charges that may incur.

Configuration

The set of files used to describe infrastructure in Terraform is simply known as a Terraform *configuration*. We're going to write our first configuration now to launch a single AWS EC2 instance.

The format of the configuration files is documented here (/docs/configuration/index.html). Configuration files can also be JSON (/docs/configuration/syntax.html), but we recommend only using JSON when the configuration is generated by a machine.

The entire configuration is shown below. We'll go over each part after. Save the contents to a file named `example.tf`. Verify that there are no other `*.tf` files in your directory, since Terraform loads all of them.

```
provider "aws" {
  access_key = "ACCESS_KEY_HERE"
  secret_key = "SECRET_KEY_HERE"
  region     = "us-east-1"
}

resource "aws_instance" "example" {
  ami          = "ami-2757f631"
  instance_type = "t2.micro"
}
```

Note: The above configuration is designed to work on most EC2 accounts, with access to a default VPC. For EC2 Classic users, please use `t1.micro` for `instance_type`, and `ami-408c7f28` for the `ami`. If you use a region other than `us-east-1` then you will need to choose an AMI in that region as AMI IDs are region specific.

Replace the `ACCESS_KEY_HERE` and `SECRET_KEY_HERE` with your AWS access key and secret key, available from this page (https://console.aws.amazon.com/iam/home#/security_credential). We're hardcoding them for now, but will extract these into variables later in the getting started guide.

Note: If you simply leave out AWS credentials, Terraform will automatically search for saved API credentials (for

example, in `~/.aws/credentials`) or IAM instance profile credentials. This option is much cleaner for situations where `tf` files are checked into source control or where there is more than one admin user. See details here (<https://aws.amazon.com/blogs/apn/terraform-beyond-the-basics-with-aws/>). Leaving IAM credentials out of the Terraform configs allows you to leave those credentials out of source control, and also use different IAM credentials for each user without having to modify the configuration files.

This is a complete configuration that Terraform is ready to apply. The general structure should be intuitive and straightforward.

The `provider` block is used to configure the named provider, in our case "aws". A provider is responsible for creating and managing resources. Multiple provider blocks can exist if a Terraform configuration is composed of multiple providers, which is a common situation.

The `resource` block defines a resource that exists within the infrastructure. A resource might be a physical component such as an EC2 instance, or it can be a logical resource such as a Heroku application.

The resource block has two strings before opening the block: the resource type and the resource name. In our example, the resource type is "aws_instance" and the name is "example." The prefix of the type maps to the provider. In our case "aws_instance" automatically tells Terraform that it is managed by the "aws" provider.

Within the resource block itself is configuration for that resource. This is dependent on each resource provider and is fully documented within our providers reference (/docs/providers/index.html). For our EC2 instance, we specify an AMI for Ubuntu, and request a "t2.micro" instance so we qualify under the free tier.

Initialization

The first command to run for a new configuration -- or after checking out an existing configuration from version control -- is `terraform init`, which initializes various local settings and data that will be used by subsequent commands.

Terraform uses a plugin based architecture to support the numerous infrastructure and service providers available. As of Terraform version 0.10.0, each "Provider" is its own encapsulated binary distributed separately from Terraform itself. The `terraform init` command will automatically download and install any Provider binary for the providers in use within the configuration, which in this case is just the aws provider:

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- downloading plugin for provider "aws"...
```

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add `version = "..."` constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

```
* provider.aws: version = "~> 1.0"
```

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running `"terraform plan"` to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your environment. If you forget, other
commands will detect it and remind you to do so if necessary.

The aws provider plugin is downloaded and installed in a subdirectory of the current working directory, along with various other book-keeping files.

The output specifies which version of the plugin was installed, and suggests specifying that version in configuration to ensure that running `terraform init` in future will install a compatible version. This step is not necessary for following the getting started guide, since this configuration will be discarded at the end.

Apply Changes

Note: The commands shown in this guide apply to Terraform 0.11 and above. Earlier versions require using the `terraform plan` command to see the execution plan before applying it. Use `terraform version` to confirm your running version.

In the same directory as the `example.tf` file you created, run `terraform apply`. You should see output similar to below, though we've truncated some of the output to save space:

```
$ terraform apply
# ...

+ aws_instance.example
  ami:                  "ami-2757f631"
  availability_zone:    "<computed>"
  ebs_block_device.#:   "<computed>"
  ephemeral_block_device.#: "<computed>"
  instance_state:       "<computed>"
  instance_type:        "t2.micro"
  key_name:             "<computed>"
  placement_group:      "<computed>"
  private_dns:          "<computed>"
  private_ip:           "<computed>"
  public_dns:           "<computed>"
  public_ip:            "<computed>"
  root_block_device.#:  "<computed>"
  security_groups.#:   "<computed>"
  source_dest_check:   "true"
  subnet_id:            "<computed>"
  tenancy:              "<computed>"
  vpc_security_group_ids.#: "<computed>"
```

This output shows the *execution plan*, describing which actions Terraform will take in order to change real infrastructure to match the configuration. The output format is similar to the diff format generated by tools such as Git. The output has a + next to `aws_instance.example`, meaning that Terraform will create this resource. Beneath that, it shows the attributes that will be set. When the value displayed is <computed>, it means that the value won't be known until the resource is created.

If `terraform apply` failed with an error, read the error message and fix the error that occurred. At this stage, it is likely to be a syntax error in the configuration.

If the plan was created successfully, Terraform will now pause and wait for approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here with no changes made to your infrastructure. In this case the plan looks acceptable, so type yes at the confirmation prompt to proceed.

Executing the plan will take a few minutes since Terraform waits for the EC2 instance to become available:

```
# ...
aws_instance.example: Creating...
  ami:                  "" => "ami-2757f631"
  instance_type:        "" => "t2.micro"
  [...]

aws_instance.example: Still creating... (10s elapsed)
aws_instance.example: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

# ...
```

After this, Terraform is all done! You can go to the EC2 console to see the created EC2 instance. (Make sure you're looking at the same region that was configured in the provider configuration!)

Terraform also wrote some data into the `terraform.tfstate` file. This state file is extremely important; it keeps track of the IDs of created resources so that Terraform knows what it is managing. This file must be saved and distributed to anyone who might run Terraform. It is generally recommended to setup remote state

(<https://www.terraform.io/docs/state/remote.html>) when working with Terraform, to share the state automatically, but this is not necessary for simple situations like this Getting Started guide.

You can inspect the current state using `terraform show`:

```
$ terraform show
aws_instance.example:
  id = i-32cf65a8
  ami = ami-2757f631
  availability_zone = us-east-1a
  instance_state = running
  instance_type = t2.micro
  private_ip = 172.31.30.244
  public_dns = ec2-52-90-212-55.compute-1.amazonaws.com
  public_ip = 52.90.212.55
  subnet_id = subnet-1497024d
  vpc_security_group_ids.# = 1
  vpc_security_group_ids.3348721628 = sg-67652003
```

You can see that by creating our resource, we've also gathered a lot of information about it. These values can actually be referenced to configure other resources or outputs, which will be covered later in the getting started guide.

Provisioning

The EC2 instance we launched at this point is based on the AMI given, but has no additional software installed. If you're running an image-based infrastructure (perhaps creating images with Packer (<https://www.packer.io>)), then this is all you need.

However, many infrastructures still require some sort of initialization or software provisioning step. Terraform supports provisioners, which we'll cover a little bit later in the getting started guide, in order to do this.

Next

Congratulations! You've built your first infrastructure with Terraform. You've seen the configuration syntax, an example of a basic execution plan, and understand the state file.

Next, we're going to move on to changing and destroying infrastructure (</intro/getting-started/change.html>).

Change Infrastructure

In the previous page, you created your first infrastructure with Terraform: a single EC2 instance. In this page, we're going to modify that resource, and see how Terraform handles change.

Infrastructure is continuously evolving, and Terraform was built to help manage and enact that change. As you change Terraform configurations, Terraform builds an execution plan that only modifies what is necessary to reach your desired state.

By using Terraform to change infrastructure, you can version control not only your configurations but also your state so you can see how the infrastructure evolved over time.

Configuration

Let's modify the ami of our instance. Edit the `aws_instance.example` resource in your configuration and change it to the following:

```
resource "aws_instance" "example" {
  ami           = "ami-b374d5a5"
  instance_type = "t2.micro"
}
```

Note: EC2 Classic users please use AMI `ami-656be372` and type `t1.micro`

We've changed the AMI from being an Ubuntu 16.04 LTS AMI to being an Ubuntu 16.10 AMI. Terraform configurations are meant to be changed like this. You can also completely remove resources and Terraform will know to destroy the old one.

Apply Changes

After changing the configuration, run `terraform apply` again to see how Terraform will apply this change to the existing resources.

```
$ terraform apply
# ...

-/+ aws_instance.example
  ami:                 "ami-2757f631" => "ami-b374d5a5" (forces new resource)
  availability_zone:   "us-east-1a" => "<computed>"
  ebs_block_device.#:  "0" => "<computed>"
  ephemeral_block_device.#: "0" => "<computed>"
  instance_state:      "running" => "<computed>"
  instance_type:        "t2.micro" => "t2.micro"
  private_dns:          "ip-172-31-17-94.ec2.internal" => "<computed>"
  private_ip:            "172.31.17.94" => "<computed>"
  public_dns:           "ec2-54-82-183-4.compute-1.amazonaws.com" => "<computed>"
  public_ip:             "54.82.183.4" => "<computed>"
  subnet_id:             "subnet-1497024d" => "<computed>"
  vpc_security_group_ids.#: "1" => "<computed>"
```

The prefix `-/+` means that Terraform will destroy and recreate the resource, rather than updating it in-place. While some

attributes can be updated in-place (which are shown with the ~ prefix), changing the AMI for an EC2 instance requires recreating it. Terraform handles these details for you, and the execution plan makes it clear what Terraform will do.

Additionally, the execution plan shows that the AMI change is what required resource to be replaced. Using this information, you can adjust your changes to possibly avoid destroy/create updates if they are not acceptable in some situations.

Once again, Terraform prompts for approval of the execution plan before proceeding. Answer yes to execute the planned steps:

```
# ...
aws_instance.example: Refreshing state... (ID: i-64c268fe)
aws_instance.example: Destroying...
aws_instance.example: Destruction complete
aws_instance.example: Creating...
  ami:          "" => "ami-b374d5a5"
  availability_zone:    "" => "<computed>"
  ebs_block_device.#:   "" => "<computed>"
  ephemeral_block_device.#: "" => "<computed>"
  instance_state:      "" => "<computed>"
  instance_type:        "" => "t2.micro"
  key_name:           "" => "<computed>"
  placement_group:     "" => "<computed>"
  private_dns:         "" => "<computed>"
  private_ip:          "" => "<computed>"
  public_dns:          "" => "<computed>"
  public_ip:           "" => "<computed>"
  root_block_device.#: "" => "<computed>"
  security_groups.#:   "" => "<computed>"
  source_dest_check:   "" => "true"
  subnet_id:          "" => "<computed>"
  tenancy:             "" => "<computed>"
  vpc_security_group_ids.#: "" => "<computed>"
aws_instance.example: Still creating... (10s elapsed)
aws_instance.example: Still creating... (20s elapsed)
aws_instance.example: Creation complete

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.

# ...
```

As indicated by the execution plan, Terraform first destroyed the existing instance and then created a new one in its place. You can use `terraform show` again to see the new values associated with this instance.

Next

You've now seen how easy it is to modify infrastructure with Terraform. Feel free to play around with this more before continuing. In the next section we're going to destroy our infrastructure (</intro/getting-started/destroy.html>).

Resource Dependencies

In this page, we're going to introduce resource dependencies, where we'll not only see a configuration with multiple resources for the first time, but also scenarios where resource parameters use information from other resources.

Up to this point, our example has only contained a single resource. Real infrastructure has a diverse set of resources and resource types. Terraform configurations can contain multiple resources, multiple resource types, and these types can even span multiple providers.

On this page, we'll show a basic example of multiple resources and how to reference the attributes of other resources to configure subsequent resources.

Assigning an Elastic IP

We'll improve our configuration by assigning an elastic IP to the EC2 instance we're managing. Modify your `example.tf` and add the following:

```
resource "aws_eip" "ip" {
  instance = "${aws_instance.example.id}"
}
```

This should look familiar from the earlier example of adding an EC2 instance resource, except this time we're building an "aws_eip" resource type. This resource type allocates and associates an elastic IP (<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html>) to an EC2 instance.

The only parameter for `aws_eip` (/docs/providers/aws/r/eip.html) is "instance" which is the EC2 instance to assign the IP to. For this value, we use an interpolation to use an attribute from the EC2 instance we managed earlier.

The syntax for this interpolation should be straightforward: it requests the "id" attribute from the "aws_instance.example" resource.

Apply Changes

Run `terraform apply` to see how Terraform plans to apply this change. The output will look similar to the following:

```
$ terraform apply

+ aws_eip.ip
  allocation_id:      "<computed>"
  association_id:    "<computed>"
  domain:            "<computed>"
  instance:          "${aws_instance.example.id}"
  network_interface: "<computed>"
  private_ip:        "<computed>"
  public_ip:         "<computed>"

+ aws_instance.example
  ami:                  "ami-b374d5a5"
  availability_zone:   "<computed>"
  ebs_block_device.#:  "<computed>"
  ephemeral_block_device.#: "<computed>"
  instance_state:      "<computed>"
  instance_type:       "t2.micro"
  key_name:            "<computed>"
  placement_group:    "<computed>"
  private_dns:         "<computed>"
  private_ip:          "<computed>"
  public_dns:          "<computed>"
  public_ip:           "<computed>"
  root_block_device.#: "<computed>"
  security_groups.#:  "<computed>"
  source_dest_check:  "true"
  subnet_id:           "<computed>"
  tenancy:             "<computed>"
  vpc_security_group_ids.#: "<computed>"
```

Terraform will create two resources: the instance and the elastic IP. In the "instance" value for the "aws_eip", you can see the raw interpolation is still present. This is because this variable won't be known until the "aws_instance" is created. It will be replaced at apply-time.

As usual, Terraform prompts for confirmation before making any changes. Answer yes to apply. The continued output will look similar to the following:

```
# ...
aws_instance.example: Creating...
  ami:      "" => "ami-b374d5a5"
  instance_type:      "" => "t2.micro"
  [...]
aws_instance.example: Still creating... (10s elapsed)
aws_instance.example: Creation complete
aws_eip.ip: Creating...
  allocation_id:      "" => "<computed>"
  association_id:    "" => "<computed>"
  domain:            "" => "<computed>"
  instance:          "" => "i-f3d77d69"
  network_interface: "" => "<computed>"
  private_ip:        "" => "<computed>"
  public_ip:         "" => "<computed>"
aws_eip.ip: Creation complete

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

As shown above, Terraform created the EC2 instance before creating the Elastic IP address. Due to the interpolation expression that passes the ID of the EC2 instance to the Elastic IP address, Terraform is able to infer a dependency, and knows it must create the instance first.

Implicit and Explicit Dependencies

By studying the resource attributes used in interpolation expressions, Terraform can automatically infer when one resource depends on another. In the example above, the expression `${aws_instance.example.id}` creates an *implicit dependency* on the `aws_instance` named `example`.

Terraform uses this dependency information to determine the correct order in which to create the different resources. In the example above, Terraform knows that the `aws_instance` must be created before the `aws_eip`.

Implicit dependencies via interpolation expressions are the primary way to inform Terraform about these relationships, and should be used whenever possible.

Sometimes there are dependencies between resources that are *not* visible to Terraform. The `depends_on` argument is accepted by any resource and accepts a list of resources to create *explicit dependencies* for.

For example, perhaps an application we will run on our EC2 instance expects to use a specific Amazon S3 bucket, but that dependency is configured inside the application code and thus not visible to Terraform. In that case, we can use `depends_on` to explicitly declare the dependency:

```
# New resource for the S3 bucket our application will use.
resource "aws_s3_bucket" "example" {
  # NOTE: S3 bucket names must be unique across _all_ AWS accounts, so
  # this name must be changed before applying this example to avoid naming
  # conflicts.
  bucket = "terraform-getting-started-guide"
  acl    = "private"
}

# Change the aws_instance we declared earlier to now include "depends_on"
resource "aws_instance" "example" {
  ami        = "ami-2757f631"
  instance_type = "t2.micro"

  # Tells Terraform that this EC2 instance must be created only after the
  # S3 bucket has been created.
  depends_on = ["aws_s3_bucket.example"]
}
```

Non-Dependent Resources

We can continue to build this configuration by adding another EC2 instance:

```
resource "aws_instance" "another" {
  ami        = "ami-b374d5a5"
  instance_type = "t2.micro"
}
```

Because this new instance does not depend on any other resource, it can be created in parallel with the other resources.

Where possible, Terraform will perform operations concurrently to reduce the total time taken to apply changes.

Before moving on, remove this new resource from your configuration and run `terraform apply` again to destroy it. We won't use this second instance any further in the getting started guide.

Next

In this page you were introduced to using multiple resources, interpolating attributes from one resource into another, and declaring dependencies between resources to define operation ordering.

In the next section, we'll use provisioners ([/intro/getting-started/provision.html](#)) to do some basic bootstrapping of our launched instance.

Destroy Infrastructure

We've now seen how to build and change infrastructure. Before we move on to creating multiple resources and showing resource dependencies, we're going to go over how to completely destroy the Terraform-managed infrastructure.

Destroying your infrastructure is a rare event in production environments. But if you're using Terraform to spin up multiple environments such as development, test, QA environments, then destroying is a useful action.

Destroy

Resources can be destroyed using the `terraform destroy` command, which is similar to `terraform apply` but it behaves as if all of the resources have been removed from the configuration.

```
$ terraform destroy  
# ...  
  
- aws_instance.example
```

The `-` prefix indicates that the instance will be destroyed. As with `apply`, Terraform shows its execution plan and waits for approval before making any changes.

Answer yes to execute this plan and destroy the infrastructure:

```
# ...  
aws_instance.example: Destroying...  
  
Apply complete! Resources: 0 added, 0 changed, 1 destroyed.  
  
# ...
```

Just like with `apply`, Terraform determines the order in which things must be destroyed. In this case there was only one resource, so no ordering was necessary. In more complicated cases with multiple resources, Terraform will destroy them in a suitable order to respect dependencies, as we'll see later in this guide.

Next

You now know how to create, modify, and destroy infrastructure from a local machine.

Next, we move on to features that make Terraform configurations slightly more useful: variables, resource dependencies, provisioning, and more ([/intro/getting-started/dependencies.html](#)).

Install Terraform

Terraform must first be installed on your machine. Terraform is distributed as a binary package (/downloads.html) for all supported platforms and architectures. This page will not cover how to compile Terraform from source, but compiling from source is covered in the documentation (/docs/index.html) for those who want to be sure they're compiling source they trust into the final binary.

Installing Terraform

To install Terraform, find the appropriate package (/downloads.html) for your system and download it. Terraform is packaged as a zip archive.

After downloading Terraform, unzip the package. Terraform runs as a single binary named `terraform`. Any other files in the package can be safely removed and Terraform will still function.

The final step is to make sure that the `terraform` binary is available on the PATH. See this page (<https://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux>) for instructions on setting the PATH on Linux and Mac. This page (<https://stackoverflow.com/questions/1618280/where-can-i-set-path-to-make-exe-on-windows>) contains instructions for setting the PATH on Windows.

Verifying the Installation

After installing Terraform, verify the installation worked by opening a new terminal session and checking that `terraform` is available. By executing `terraform` you should see help output similar to this:

```
$ terraform
Usage: terraform [--version] [--help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
  apply           Builds or changes infrastructure
  console         Interactive console for Terraform interpolations
# ...
```

If you get an error that `terraform` could not be found, your PATH environment variable was not set up properly. Please go back and ensure that your PATH variable contains the directory where Terraform was installed.

Next Steps

Time to build infrastructure (/intro/getting-started/build.html) using a minimal Terraform configuration file. You will be able to examine Terraform's execution plan before you deploy it to AWS.

Modules

Up to this point, we've been configuring Terraform by editing Terraform configurations directly. As our infrastructure grows, this practice has a few key problems: a lack of organization, a lack of reusability, and difficulties in management for teams.

Modules in Terraform are self-contained packages of Terraform configurations that are managed as a group. Modules are used to create reusable components, improve organization, and to treat pieces of infrastructure as a black box.

This section of the getting started will cover the basics of using modules. Writing modules is covered in more detail in the modules documentation ([/docs/modules/index.html](#)).

Warning! The examples on this page are *not eligible* for the AWS free tier (<https://aws.amazon.com/free/>). Do not try the examples on this page unless you're willing to spend a small amount of money.

Using Modules

If you have any instances running from prior steps in the getting started guide, use `terraform destroy` to destroy them, and remove all configuration files.

The Terraform Registry (<https://registry.terraform.io/>) includes a directory of ready-to-use modules for various common purposes, which can serve as larger building-blocks for your infrastructure.

In this example, we're going to use the Consul Terraform module for AWS (<https://registry.terraform.io/modules/hashicorp/consul/aws>), which will set up a complete Consul (<https://www.consul.io>) cluster. This and other modules can be found via the search feature on the Terraform Registry site.

Create a configuration file with the following contents:

```
provider "aws" {  
    access_key = "AWS ACCESS KEY"  
    secret_key = "AWS SECRET KEY"  
    region     = "us-east-1"  
}  
  
module "consul" {  
    source = "hashicorp/consul/aws"  
  
    num_servers = "3"  
}
```

The `module` block begins with the example given on the Terraform Registry page for this module, telling Terraform to create and manage this module. This is similar to a `resource` block: it has a name used within this configuration -- in this case, "`consul`" -- and a set of input values that are listed in the module's "Inputs" documentation (<https://registry.terraform.io/modules/hashicorp/consul/aws?tab=inputs>).

(Note that the `provider` block can be omitted in favor of environment variables. See the AWS Provider docs ([/docs/providers/aws/index.html](#)) for details. This module requires that your AWS account has a default VPC.)

The `source` attribute is the only mandatory argument for modules. It tells Terraform where the module can be retrieved. Terraform automatically downloads and manages modules for you.

In this case, the module is retrieved from the official Terraform Registry. Terraform can also retrieve modules from a variety

of sources, including private module registries or directly from Git, Mercurial, HTTP, and local files.

The other attributes shown are inputs to our module. This module supports many additional inputs, but all are optional and have reasonable values for experimentation.

After adding a new module to configuration, it is necessary to run (or re-run) `terraform init` to obtain and install the new module's source code:

```
$ terraform init  
# ...
```

By default, this command does not check for new module versions that may be available, so it is safe to run multiple times. The `-upgrade` option will additionally check for any newer versions of existing modules and providers that may be available.

Apply Changes

With the Consul module (and its dependencies) installed, we can now apply these changes to create the resources described within.

If you run `terraform apply`, you will see a large list of all of the resources encapsulated in the module. The output is similar to what we saw when using resources directly, but the resource names now have module paths prefixed to their names, like in the following example:

```
+ module.consul.module.consul_clients.aws_autoscaling_group.autoscaling_group  
  id:                      <computed>  
  arn:                     <computed>  
  default_cooldown:        <computed>  
  desired_capacity:        "6"  
  force_delete:            "false"  
  health_check_grace_period: "300"  
  health_check_type:       "EC2"  
  launch_configuration:    "${aws_launch_configuration.launch_configuration.name}"  
  max_size:                "6"  
  metrics_granularity:     "1Minute"  
  min_size:                "6"  
  name:                    <computed>  
  protect_from_scale_in:  "false"  
  tag.#:                  "2"  
  tag.2151078592.key:      "consul-clients"  
  tag.2151078592.propagate_at_launch: "true"  
  tag.2151078592.value:    "consul-example"  
  tag.462896764.key:       "Name"  
  tag.462896764.propagate_at_launch: "true"  
  tag.462896764.value:     "consul-example-client"  
  termination_policies.#:  "1"  
  termination_policies.0:   "Default"  
  vpc_zone_identifier.#:   "6"  
  vpc_zone_identifier.1880739334: "subnet-5ce4282a"  
  vpc_zone_identifier.3458061785: "subnet-16600f73"  
  vpc_zone_identifier.4176925006: "subnet-485abd10"  
  vpc_zone_identifier.4226228233: "subnet-40a9b86b"  
  vpc_zone_identifier.595613151:  "subnet-5131b95d"  
  vpc_zone_identifier.765942872:  "subnet-595ae164"  
  wait_for_capacity_timeout:  "10m"
```

The `module.consul.module.consul_clients` prefix shown above indicates not only that the resource is from the module

"consul" block we wrote, but in fact that this module has its own module "consul_clients" block within it. Modules can be nested to decompose complex systems into manageable components.

The full set of resources created by this module includes an autoscaling group, security groups, IAM roles and other individual resources that all support the Consul cluster that will be created.

Note that as we warned above, the resources created by this module are not eligible for the AWS free tier and so proceeding further will have some cost associated. To proceed with the creation of the Consul cluster, type yes at the confirmation prompt.

```
# ...

module.consul.module.consul_clients.aws_security_group.lc_security_group: Creating...
  description:      "" => "Security group for the consul-example-client launch configuration"
  egress.#:        "" => "<computed>"
  ingress.#:       "" => "<computed>"
  name:            "" => "<computed>"
  name_prefix:     "" => "consul-example-client"
  owner_id:        "" => "<computed>"
  revoke_rules_on_delete: "" => "false"
  vpc_id:          "" => "vpc-22099946"

# ...

Apply complete! Resources: 34 added, 0 changed, 0 destroyed.
```

After several minutes and many log messages about all of the resources being created, you'll have a three-server Consul cluster up and running. Without needing any knowledge of how Consul works, how to install Consul, or how to form a Consul cluster, you've created a working cluster in just a few minutes.

Module Outputs

Just as the module instance had input arguments such as `num_servers` above, module can also produce *output* values, similar to resource attributes.

The module's outputs reference (<https://registry.terraform.io/modules/hashicorp/consul/aws?tab=outputs>) describes all of the different values it produces. Overall, it exposes the id of each of the resources it creates, as well as echoing back some of the input values.

One of the supported outputs is called `asg_name_servers`, and its value is the name of the auto-scaling group that was created to manage the Consul servers.

To reference this, we'll just put it into our *own* output value. This value could actually be used anywhere: in another resource, to configure another provider, etc.

Add the following to the end of the existing configuration file created above:

```
output "consul_server_asg_name" {
  value = "${module.consul.asg_name_servers}"
}
```

The syntax for referencing module outputs is `${module.NAME.OUTPUT}`, where NAME is the module name given in the header of the module configuration block and OUTPUT is the name of the output to reference.

If you run `terraform apply` again, Terraform will make no changes to infrastructure, but you'll now see the "consul_server_asg_name" output with the name of the created auto-scaling group:

```
# ...  
  
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
consul_server_asg_name = tf-asg-2017103123350991200000000a
```

If you look in the Auto-scaling Groups section of the EC2 console you should find an autoscaling group of this name, and from there find the three Consul servers it is running. (If you can't find it, make sure you're looking in the right region!)

Destroy

Just as with top-level resources, we can destroy the resources created by the Consul module to avoid ongoing costs:

```
$ terraform destroy  
# ...  
  
Terraform will perform the following actions:  
  
- module.consul.module.consul_clients.aws_autoscaling_group.autoscaling_group  
- module.consul.module.consul_clients.aws_iam_instance_profile.instance_profile  
- module.consul.module.consul_clients.aws_iam_role.instance_role  
  
# ...
```

As usual, Terraform describes all of the actions it will take. In this case, it plans to destroy all of the resources that were created by the module. Type yes to confirm and, after a few minutes and even more log output, all of the resources should be destroyed:

```
Destroy complete! Resources: 34 destroyed.
```

With all of the resources destroyed, you can delete the configuration file we created above. We will not make any further use of it, and so this avoids the risk of accidentally re-creating the Consul cluster.

Next

For more information on modules, the types of sources supported, how to write modules, and more, read the in-depth module documentation ([/docs/modules/index.html](#)).

Next, we learn about Terraform's remote collaboration features ([/intro/getting-started/remote.html](#)).

Next Steps

That concludes the getting started guide for Terraform. Hopefully you're now able to not only see what Terraform is useful for, but you're also able to put this knowledge to use to improve building your own infrastructure.

We've covered the basics for all of these features in this guide.

As a next step, the following resources are available:

- Documentation (</docs/index.html>) - The documentation is an in-depth reference guide to all the features of Terraform, including technical details about the internals of how Terraform operates.
- Examples (</intro/examples/index.html>) - The examples have more full featured configuration files, showing some of the possibilities with Terraform.
- Import (</docs/import/index.html>) - The import section of the documentation covers importing existing infrastructure into Terraform.

Output Variables

In the previous section, we introduced input variables as a way to parameterize Terraform configurations. In this page, we introduce output variables as a way to organize data to be easily queried and shown back to the Terraform user.

When building potentially complex infrastructure, Terraform stores hundreds or thousands of attribute values for all your resources. But as a user of Terraform, you may only be interested in a few values of importance, such as a load balancer IP, VPN address, etc.

Outputs are a way to tell Terraform what data is important. This data is outputted when `apply` is called, and can be queried using the `terraform output` command.

Defining Outputs

Let's define an output to show us the public IP address of the elastic IP address that we create. Add this to any of your `*.tf` files:

```
output "ip" {
  value = "${aws_eip.ip.public_ip}"
}
```

This defines an output variable named "ip". The name of the variable must conform to Terraform variable naming conventions if it is to be used as an input to other modules. The `value` field specifies what the value will be, and almost always contains one or more interpolations, since the output data is typically dynamic. In this case, we're outputting the `public_ip` attribute of the elastic IP address.

Multiple output blocks can be defined to specify multiple output variables.

Viewing Outputs

Run `terraform apply` to populate the output. This only needs to be done once after the output is defined. The `apply` output should change slightly. At the end you should see this:

```
$ terraform apply
...
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

  ip = 50.17.232.209
```

`apply` highlights the outputs. You can also query the outputs after `apply`-time using `terraform output`:

```
$ terraform output ip
50.17.232.209
```

This command is useful for scripts to extract outputs.

Next

You now know how to parameterize configurations with input variables, extract important data using output variables, and bootstrap resources using provisioners.

Next, we're going to take a look at how to use modules ([/intro/getting-started/modules.html](#)), a useful abstraction to organize and reuse Terraform configurations.

Provision

You're now able to create and modify infrastructure. Now let's see how to use provisioners to initialize instances when they're created.

If you're using an image-based infrastructure (perhaps with images created with Packer (<https://www.packer.io>)), then what you've learned so far is good enough. But if you need to do some initial setup on your instances, then provisioners let you upload files, run shell scripts, or install and trigger other software like configuration management tools, etc.

Defining a Provisioner

To define a provisioner, modify the resource block defining the "example" EC2 instance to look like the following:

```
resource "aws_instance" "example" {
  ami           = "ami-b374d5a5"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo ${aws_instance.example.public_ip} > ip_address.txt"
  }
}
```

This adds a `provisioner` block within the `resource` block. Multiple `provisioner` blocks can be added to define multiple provisioning steps. Terraform supports multiple provisioners (</docs/provisioners/index.html>), but for this example we are using the `local-exec` provisioner.

The `local-exec` provisioner executes a command locally on the machine running Terraform. We're using this provisioner versus the others so we don't have to worry about specifying any connection info (</docs/provisioners/connection.html>) right now.

Running Provisioners

Provisioners are only run when a resource is *created*. They are not a replacement for configuration management and changing the software of an already-running server, and are instead just meant as a way to bootstrap a server. For configuration management, you should use Terraform provisioning to invoke a real configuration management solution.

Make sure that your infrastructure is destroyed (</intro/getting-started/destroy.html>) if it isn't already, then run `apply`:

```
$ terraform apply
# ...

aws_instance.example: Creating...
  ami:      "" => "ami-b374d5a5"
  instance_type: "" => "t2.micro"
aws_eip.ip: Creating...
  instance: "" => "i-213f350a"

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

Terraform will output anything from provisioners to the console, but in this case there is no output. However, we can verify everything worked by looking at the `ip_address.txt` file:

```
$ cat ip_address.txt  
54.192.26.128
```

It contains the IP, just as we asked!

Failed Provisioners and Tainted Resources

If a resource successfully creates but fails during provisioning, Terraform will error and mark the resource as "tainted". A resource that is tainted has been physically created, but can't be considered safe to use since provisioning failed.

When you generate your next execution plan, Terraform will not attempt to restart provisioning on the same resource because it isn't guaranteed to be safe. Instead, Terraform will remove any tainted resources and create new resources, attempting to provision them again after creation.

Terraform also does not automatically roll back and destroy the resource during the apply when the failure happens, because that would go against the execution plan: the execution plan would've said a resource will be created, but does not say it will ever be deleted. If you create an execution plan with a tainted resource, however, the plan will clearly state that the resource will be destroyed because it is tainted.

Destroy Provisioners

Provisioners can also be defined that run only during a destroy operation. These are useful for performing system cleanup, extracting data, etc.

For many resources, using built-in cleanup mechanisms is recommended if possible (such as init scripts), but provisioners can be used if necessary.

The getting started guide won't show any destroy provisioner examples. If you need to use destroy provisioners, please see the provisioner documentation ([/docs/provisioners](#)).

Next

Provisioning is important for being able to bootstrap instances. As another reminder, it is not a replacement for configuration management. It is meant to simply bootstrap machines. If you use configuration management, you should use the provisioning as a way to bootstrap the configuration management tool.

In the next section, we start looking at variables as a way to parameterize our configurations ([/intro/getting-started/variables.html](#)).

Remote Backends

We've now seen how to build, change, and destroy infrastructure from a local machine. This is great for testing and development, but in production environments it is more responsible to share responsibility for infrastructure. The best way to do this is by running Terraform in a remote environment with shared access to state.

Terraform supports team-based workflows with a feature known as remote backends (/docs/backends). Remote backends allow Terraform to use a shared storage space for state data, so any member of your team can use Terraform to manage the same infrastructure.

Depending on the features you wish to use, Terraform has multiple remote backend options. You could use Consul for state storage, locking, and environments. This is a free and open source option. You can use S3 which only supports state storage, for a low cost and minimally featured solution.

Terraform Enterprise (https://www.hashicorp.com/products/terraform/?utm_source=oss&utm_medium=getting-started&utm_campaign=terraform) is HashiCorp's commercial solution and also acts as a remote backend. Terraform Enterprise allows teams to easily version, audit, and collaborate on infrastructure changes. Each proposed change generates a Terraform plan which can be reviewed and collaborated on as a team. When a proposed change is accepted, the Terraform logs are stored, resulting in a linear history of infrastructure states to help with auditing and policy enforcement. Additional benefits to running Terraform remotely include moving access credentials off of developer machines and freeing local machines from long-running Terraform processes.

How to Store State Remotely

First, we'll use Consul (<https://www.consul.io>) as our backend. Consul is a free and open source solution that provides state storage, locking, and environments. It is a great way to get started with Terraform backends.

We'll use the demo Consul server (<https://demo.consul.io>) for this guide. This should not be used for real data. Additionally, the demo server doesn't permit locking. If you want to play with state locking (/docs/state/locking.html), you'll have to run your own Consul server or use a backend that supports locking.

First, configure the backend in your configuration:

```
terraform {
  backend "consul" {
    address = "demo.consul.io"
    path    = "getting-started-RANDOMSTRING"
    lock    = false
    scheme  = "https"
  }
}
```

Please replace "RANDOMSTRING" with some random text. The demo server is public and we want to try to avoid overlapping with someone else running through the getting started guide.

The backend section configures the backend you want to use. After configuring a backend, run `terraform init` to setup Terraform. It should ask if you want to migrate your state to Consul. Say "yes" and Terraform will copy your state.

Now, if you run `terraform apply`, Terraform should state that there are no changes:

```
$ terraform apply
# ...

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, Terraform
doesn't need to do anything.
```

Terraform is now storing your state remotely in Consul. Remote state storage makes collaboration easier and keeps state and secret information off your local disk. Remote state is loaded only in memory when it is used.

If you want to move back to local state, you can remove the backend configuration block from your configuration and run `terraform init` again. Terraform will once again ask if you want to migrate your state back to local.

Terraform Enterprise

Terraform Enterprise (https://www.hashicorp.com/products/terraform/?utm_source=oss&utm_medium=getting-started&utm_campaign=terraform) is a commercial solution which combines a predictable and reliable shared run environment with tools to help you work together on Terraform configurations and modules.

Although Terraform Enterprise can act as a standard remote backend to support Terraform runs on local machines, it works even better as a remote run environment. It supports two main workflows for performing Terraform runs:

- A VCS-driven workflow, in which it automatically queues plans whenever changes are committed to your configuration's VCS repo.
- An API-driven workflow, in which a CI pipeline or other automated tool can upload configurations directly.

For a hands-on introduction to Terraform Enterprise, follow the Terraform Enterprise getting started guide (</docs/enterprise/getting-started/index.html>).

Next

You now know how to create, modify, destroy, version, and collaborate on infrastructure. With these building blocks, you can effectively experiment with any part of Terraform.

We've now concluded the getting started guide, however there are a number of next steps (</intro/getting-started/next-steps.html>) to get started with Terraform.

Input Variables

You now have enough Terraform knowledge to create useful configurations, but we're still hard-coding access keys, AMIs, etc. To become truly shareable and version controlled, we need to parameterize the configurations. This page introduces input variables as a way to do this.

Defining Variables

Let's first extract our access key, secret key, and region into a few variables. Create another file `variables.tf` with the following contents.

Note: that the file can be named anything, since Terraform loads all files ending in `.tf` in a directory.

```
variable "access_key" {}
variable "secret_key" {}
variable "region" {
  default = "us-east-1"
}
```

This defines three variables within your Terraform configuration. The first two have empty blocks `{}`. The third sets a default. If a default value is set, the variable is optional. Otherwise, the variable is required. If you run `terraform plan` now, Terraform will prompt you for the values for unset string variables.

Using Variables in Configuration

Next, replace the AWS provider configuration with the following:

```
provider "aws" {
  access_key = "${var.access_key}"
  secret_key = "${var.secret_key}"
  region     = "${var.region}"
}
```

This uses more interpolations, this time prefixed with `var..`. This tells Terraform that you're accessing variables. This configures the AWS provider with the given variables.

Assigning Variables

There are multiple ways to assign variables. Below is also the order in which variable values are chosen. The following is the descending order of precedence in which variables are considered.

Command-line flags

You can set variables directly on the command-line with the `-var` flag. Any command in Terraform that inspects the configuration accepts this flag, such as `apply`, `plan`, and `refresh`:

```
$ terraform apply \
-var 'access_key=foo' \
-var 'secret_key=bar'
# ...
```

Once again, setting variables this way will not save them, and they'll have to be input repeatedly as commands are executed.

From a file

To persist variable values, create a file and assign variables within this file. Create a file named `terraform.tfvars` with the following contents:

```
access_key = "foo"
secret_key = "bar"
```

For all files which match `terraform.tfvars` or `*.auto.tfvars` present in the current directory, Terraform automatically loads them to populate variables. If the file is named something else, you can use the `-var-file` flag directly to specify a file. These files are the same syntax as Terraform configuration files. And like Terraform configuration files, these files can also be JSON.

We don't recommend saving usernames and password to version control, but you can create a local secret variables file and use `-var-file` to load it.

You can use multiple `-var-file` arguments in a single command, with some checked in to version control and others not checked in. For example:

```
$ terraform apply \
-var-file="secret.tfvars" \
-var-file="production.tfvars"
```

From environment variables

Terraform will read environment variables in the form of `TF_VAR_name` to find the value for a variable. For example, the `TF_VAR_access_key` variable can be set to set the `access_key` variable.

Note: Environment variables can only populate string-type variables. List and map type variables must be populated via one of the other mechanisms.

UI Input

If you execute `terraform apply` with certain variables unspecified, Terraform will ask you to input their values interactively. These values are not saved, but this provides a convenient workflow when getting started with Terraform. UI Input is not recommended for everyday use of Terraform.

Note: UI Input is only supported for string variables. List and map variables must be populated via one of the other mechanisms.

Variable Defaults

If no value is assigned to a variable via any of these methods and the variable has a `default` key in its declaration, that value will be used for the variable.

Lists

Lists are defined either explicitly or implicitly

```
# implicitly by using brackets [...]
variable "cidrs" { default = [] }

# explicitly
variable "cidrs" { type = "list" }
```

You can specify lists in a `terraform.tfvars` file:

```
cidrs = [ "10.0.0.0/16", "10.1.0.0/16" ]
```

Maps

We've replaced our sensitive strings with variables, but we still are hard-coding AMIs. Unfortunately, AMIs are specific to the region that is in use. One option is to just ask the user to input the proper AMI for the region, but Terraform can do better than that with *maps*.

Maps are a way to create variables that are lookup tables. An example will show this best. Let's extract our AMIs into a map and add support for the `us-west-2` region as well:

```
variable "amis" {
  type = "map"
  default = {
    "us-east-1" = "ami-b374d5a5"
    "us-west-2" = "ami-4b32be2b"
  }
}
```

A variable can have a `map` type assigned explicitly, or it can be implicitly declared as a map by specifying a `default` value that is a map. The above demonstrates both.

Then, replace the `aws_instance` with the following:

```
resource "aws_instance" "example" {
  ami           = "${lookup(var.amis, var.region)}"
  instance_type = "t2.micro"
}
```

This introduces a new type of interpolation: a function call. The `lookup` function does a dynamic lookup in a map for a key. The key is `var.region`, which specifies that the value of the `region` variable is the key.

While we don't use it in our example, it is worth noting that you can also do a static lookup of a map directly with `${var.amis["us-east-1"]}`.

Assigning Maps

We set defaults above, but maps can also be set using the `-var` and `-var-file` values. For example:

```
$ terraform apply -var 'amis={ us-east-1 = "foo", us-west-2 = "bar" }'  
# ...
```

Note: Even if every key will be assigned as input, the variable must be established as a map by setting its default to `{ }`.

Here is an example of setting a map's keys from a file. Starting with these variable definitions:

```
variable "region" {}  
variable "amis" {  
  type = "map"  
}
```

You can specify keys in a `terraform.tfvars` file:

```
amis = {  
  "us-east-1" = "ami-abc123"  
  "us-west-2" = "ami-def456"  
}
```

And access them via `lookup()`:

```
output "ami" {  
  value = "${lookup(var.amis, var.region)}"  
}
```

Like so:

```
$ terraform apply -var region=us-west-2

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

ami = ami-def456
```

Next

Terraform provides variables for parameterizing your configurations. Maps let you build lookup tables in cases where that makes sense. Setting and using variables is uniform throughout your configurations.

In the next section, we'll take a look at output variables ([/intro/getting-started/outputs.html](#)) as a mechanism to expose certain values more prominently to the Terraform operator.

Use Cases

Before understanding use cases, it's useful to know what Terraform is ([/intro/index.html](#)). This page lists some concrete use cases for Terraform, but the possible use cases are much broader than what we cover. Due to its extensible nature, providers and provisioners can be added to further extend Terraform's ability to manipulate resources.

Heroku App Setup

Heroku is a popular PaaS for hosting web apps. Developers create an app, and then attach add-ons, such as a database, or email provider. One of the best features is the ability to elastically scale the number of dynos or workers. However, most non-trivial applications quickly need many add-ons and external services.

Terraform can be used to codify the setup required for a Heroku application, ensuring that all the required add-ons are available, but it can go even further: configuring DNSimple to set a CNAME, or setting up Cloudflare as a CDN for the app. Best of all, Terraform can do all of this in under 30 seconds without using a web interface.

Multi-Tier Applications

A very common pattern is the N-tier architecture. The most common 2-tier architecture is a pool of web servers that use a database tier. Additional tiers get added for API servers, caching servers, routing meshes, etc. This pattern is used because the tiers can be scaled independently and provide a separation of concerns.

Terraform is an ideal tool for building and managing these infrastructures. Each tier can be described as a collection of resources, and the dependencies between each tier are handled automatically; Terraform will ensure the database tier is available before the web servers are started and that the load balancers are aware of the web nodes. Each tier can then be scaled easily using Terraform by modifying a single count configuration value. Because the creation and provisioning of a resource is codified and automated, elastically scaling with load becomes trivial.

Self-Service Clusters

At a certain organizational size, it becomes very challenging for a centralized operations team to manage a large and growing infrastructure. Instead it becomes more attractive to make "self-serve" infrastructure, allowing product teams to manage their own infrastructure using tooling provided by the central operations team.

Using Terraform, the knowledge of how to build and scale a service can be codified in a configuration. Terraform configurations can be shared within an organization enabling customer teams to use the configuration as a black box and use Terraform as a tool to manage their services.

Software Demos

Modern software is increasingly networked and distributed. Although tools like Vagrant (<https://www.vagrantup.com/>) exist to build virtualized environments for demos, it is still very challenging to demo software on real infrastructure which more closely matches production environments.

Software writers can provide a Terraform configuration to create, provision and bootstrap a demo on cloud providers like AWS. This allows end users to easily demo the software on their own infrastructure, and even enables tweaking parameters like cluster size to more rigorously test tools at any scale.

Disposable Environments

It is common practice to have both a production and staging or QA environment. These environments are smaller clones of their production counterpart, but are used to test new applications before releasing in production. As the production environment grows larger and more complex, it becomes increasingly onerous to maintain an up-to-date staging environment.

Using Terraform, the production environment can be codified and then shared with staging, QA or dev. These configurations can be used to rapidly spin up new environments to test in, and then be easily disposed of. Terraform can help tame the difficulty of maintaining parallel environments, and makes it practical to elastically create and destroy them.

Software Defined Networking

Software Defined Networking (SDN) is becoming increasingly prevalent in the datacenter, as it provides more control to operators and developers and allows the network to better support the applications running on top. Most SDN implementations have a control layer and infrastructure layer.

Terraform can be used to codify the configuration for software defined networks. This configuration can then be used by Terraform to automatically setup and modify settings by interfacing with the control layer. This allows configuration to be versioned and changes to be automated. As an example, AWS VPC (<https://aws.amazon.com/vpc/>) is one of the most commonly used SDN implementations, and can be configured by Terraform (/docs/providers/aws/r/vpc.html).

Resource Schedulers

In large-scale infrastructures, static assignment of applications to machines becomes increasingly challenging. To solve that problem, there are a number of schedulers like Borg, Mesos, YARN, and Kubernetes. These can be used to dynamically schedule Docker containers, Hadoop, Spark, and many other software tools.

Terraform is not limited to physical providers like AWS. Resource schedulers can be treated as a provider, enabling Terraform to request resources from them. This allows Terraform to be used in layers: to setup the physical infrastructure running the schedulers as well as provisioning onto the scheduled grid.

Multi-Cloud Deployment

It's often attractive to spread infrastructure across multiple clouds to increase fault-tolerance. By using only a single region or cloud provider, fault tolerance is limited by the availability of that provider. Having a multi-cloud deployment allows for more graceful recovery of the loss of a region or entire provider.

Realizing multi-cloud deployments can be very challenging as many existing tools for infrastructure management are cloud-specific. Terraform is cloud-agnostic and allows a single configuration to be used to manage multiple providers, and to even handle cross-cloud dependencies. This simplifies management and orchestration, helping operators build large-scale multi-

cloud infrastructures.

Terraform vs. Other Software

Terraform provides a flexible abstraction of resources and providers. This model allows for representing everything from physical hardware, virtual machines, and containers, to email and DNS providers. Because of this flexibility, Terraform can be used to solve many different problems. This means there are a number of existing tools that overlap with the capabilities of Terraform. We compare Terraform to a number of these tools, but it should be noted that Terraform is not mutually exclusive with other systems. It can be used to manage a single application, or the entire datacenter.

Use the navigation on the left to read comparisons of Terraform versus other specific systems.

Terraform vs. Boto, Fog, etc.

Libraries like Boto, Fog, etc. are used to provide native access to cloud providers and services by using their APIs. Some libraries are focused on specific clouds, while others attempt to bridge them all and mask the semantic differences. Using a client library only provides low-level access to APIs, requiring application developers to create their own tooling to build and manage their infrastructure.

Terraform is not intended to give low-level programmatic access to providers, but instead provides a high level syntax for describing how cloud resources and services should be created, provisioned, and combined. Terraform is very flexible, using a plugin-based model to support providers and provisioners, giving it the ability to support almost any service that exposes APIs.

Terraform vs. Chef, Puppet, etc.

Configuration management tools install and manage software on a machine that already exists. Terraform is not a configuration management tool, and it allows existing tooling to focus on their strengths: bootstrapping and initializing resources.

Using provisioners, Terraform enables any configuration management tool to be used to setup a resource once it has been created. Terraform focuses on the higher-level abstraction of the datacenter and associated services, without sacrificing the ability to use configuration management tools to do what they do best. It also embraces the same codification that is responsible for the success of those tools, making entire infrastructure deployments easy and reliable.

Terraform vs. CloudFormation, Heat, etc.

Tools like CloudFormation, Heat, etc. allow the details of an infrastructure to be codified into a configuration file. The configuration files allow the infrastructure to be elastically created, modified and destroyed. Terraform is inspired by the problems they solve.

Terraform similarly uses configuration files to detail the infrastructure setup, but it goes further by being both cloud-agnostic and enabling multiple providers and services to be combined and composed. For example, Terraform can be used to orchestrate an AWS and OpenStack cluster simultaneously, while enabling 3rd-party providers like Cloudflare and DNSimple to be integrated to provide CDN and DNS services. This enables Terraform to represent and manage the entire infrastructure with its supporting services, instead of only the subset that exists within a single provider. It provides a single unified syntax, instead of requiring operators to use independent and non-interoperable tools for each platform and service.

Terraform also separates the planning phase from the execution phase, by using the concept of an execution plan. By running `terraform plan`, the current state is refreshed and the configuration is consulted to generate an action plan. The plan includes all actions to be taken: which resources will be created, destroyed or modified. It can be inspected by operators to ensure it is exactly what is expected. Using `terraform graph`, the plan can be visualized to show dependent ordering. Once the plan is captured, the execution phase can be limited to only the actions in the plan. Other tools combine the planning and execution phases, meaning operators are forced to mentally reason about the effects of a change, which quickly becomes intractable in large infrastructures. Terraform lets operators apply changes with confidence, as they know exactly what will happen beforehand.

Terraform vs. Custom Solutions

Most organizations start by manually managing infrastructure through simple scripts or web-based interfaces. As the infrastructure grows, any manual approach to management becomes both error-prone and tedious, and many organizations begin to home-roll tooling to help automate the mechanical processes involved.

These tools require time and resources to build and maintain. As tools of necessity, they represent the minimum viable features needed by an organization, being built to handle only the immediate needs. As a result, they are often hard to extend and difficult to maintain. Because the tooling must be updated in lockstep with any new features or infrastructure, it becomes the limiting factor for how quickly the infrastructure can evolve.

Terraform is designed to tackle these challenges. It provides a simple, unified syntax, allowing almost any resource to be managed without learning new tooling. By capturing all the resources required, the dependencies between them can be resolved automatically so that operators do not need to remember and reason about them. Removing the burden of building the tool allows operators to focus on their infrastructure and not the tooling.

Furthermore, Terraform is an open source tool. In addition to HashiCorp, the community around Terraform helps to extend its features, fix bugs and document new use cases. Terraform helps solve a problem that exists in every organization and provides a standard that can be adopted to avoid reinventing the wheel between and within organizations. Its open source nature ensures it will be around in the long term.

User-agent: *\nDisallow: /404\nDisallow: /500

Terraform Security

We understand that many users place a high level of trust in HashiCorp and the tools we build. We apply best practices and focus on security to make sure we can maintain the trust of the community.

We deeply appreciate any effort to disclose vulnerabilities responsibly.

If you would like to report a vulnerability, please see the HashiCorp security page (<https://www.hashicorp.com/security.html>), which has the proper email to communicate with as well as our PGP key. Please **do not create an GitHub issue for security concerns.**

If you need to report a non-security related bug, please open and issue on the Terraform GitHub repository (<https://github.com/hashicorp/terraform>).

Artifact Provider

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Terraform has a provider (<https://terraform.io/docs/providers/index.html>) for managing Terraform Enterprise artifacts called `atlas_artifact`.

This is used to make data stored in Artifacts available to Terraform for interpolation. In the following example, an artifact is defined and references an AMI ID stored in Terraform Enterprise.

Why is this called "atlas"? Atlas was previously a commercial offering from HashiCorp that included a full suite of enterprise products. The products have since been broken apart into their individual products, like **Terraform Enterprise**. While this transition is in progress, you may see references to "atlas" in the documentation. We apologize for the inconvenience.

```
provider "atlas" {
  # You can also set the atlas token by exporting ATLAS_TOKEN into your env
  token = "${var.atlas_token}"
}

data "atlas_artifact" "web-worker" {
  name      = "my-username/web-worker"
  type      = "amazon.image"
  version   = "latest"
}

resource "aws_instance" "worker-machine" {
  ami           = "${atlas_artifact.web-worker.metadata_full.region-us-east-1}"
  instance_type = "m1.small"
}
```

This automatically pulls the "latest" artifact version.

Following a new artifact version being created via a Packer build, the following diff would be generated when running `terraform plan`.

```
-/+ aws_instance.worker-machine
  ami:          "ami-168f9d7e" => "ami-2f3a9df2" (forces new resource)
  instance_type: "m1.small" => "m1.small"
```

This allows you to reference changing artifacts and trigger new deployments upon pushing subsequent Packer builds.

Read more about artifacts in the Terraform documentation (<https://terraform.io/docs/providers/terraform-enterprise/r/artifact.html>).

Creating AMI Artifacts with Packer and Terraform Enterprise

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Currently, the best way to create AWS AMI artifacts is with Packer.

We detail how to do this in the Packer section of the documentation (</docs/enterprise-legacy/packer/artifacts/creating-amis.html>).

Managing Artifact Versions

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Artifacts stored in Terraform Enterprise are versioned and assigned a version number. Versions are useful to roll back, audit and deploy images specific versions of images to certain environments in a targeted way.

This assumes you are familiar with the artifact provider (<https://terraform.io/docs/providers/terraform-enterprise/index.html>) in Terraform.

Finding the Version of an Artifact

Artifact versions can be found with the `terraform show` command (<https://terraform.io/docs/commands/show.html>), or by looking at the Packer logs generated during builds. After a successful artifact upload, version numbers are displayed. "latest" can be used to use the latest version of the artifact.

The following output is from `terraform show`.

```
atlas_artifact.web-worker:
  id = us-east-1:ami-3a0a1d52
  build = latest
  metadata_full.# = 1
  metadata_full.region-us-east-1 = ami-3a0a1d52
  name = my-username/web-worker
  slug = my-username/web-worker/amazon.image/7
  type = amazon.image
```

In this case, the version is 7 and can be found in the persisted slug attribute.

Pinning Artifacts to Specific Versions

You can pin artifacts to a specific version. This allows for a targeted deploy.

```
data "atlas_artifact" "web-worker" {
  name  = "my-username/web-worker"
  type  = "amazon.image"
  version = 7
}
```

This will use version 7 of the web-worker artifact.

Pinning Artifacts to Specific Builds

Artifacts can also be pinned to an Terraform build number. This is only possible if Terraform Enterprise was used to build the artifact with Packer.

```
data "atlas_artifact" "web-worker" {  
  name  = "my-username/web-worker"  
  type   = "amazon.image"  
  build  = 5  
}
```

It's recommended to use versions, instead of builds, as it will be easier to track when building outside of the Terraform Enterprise environment.

Frequently Asked Questions

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

Monolithic Artifacts (</docs/enterprise-legacy/faq/monolithic-artifacts.html>) - *How do I build multiple applications into one artifact?*

Rolling Deployments (</docs/enterprise-legacy/faq/rolling-deployments.html>) - *How do I configure rolling deployments?*

Vagrant Cloud Migration (</docs/enterprise-legacy/faq/vagrant-cloud-migration.html>) - *How can I prepare for the Vagrant Cloud Mirgration?*

Monolithic Artifacts

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

How do I build multiple applications into one artifact?

Create your new Applications in Terraform Enterprise using the application compilation feature.

You can either link each Application to the single Build Template you will be using to create the monolithic artifact, or run periodic Packer builds.

Each time an Application is pushed, it will store the new application version in the artifact registry as a tarball. These will be available for you to download at build-time on the machines they belong.

Here's an example `compile.json` template that you will include with the rest of your application files that do the compiling:

```
{
  "variables": {
    "app_slug": "{{ env `ATLAS_APPLICATION_SLUG` }}"
  },
  "builders": [
    {
      "type": "docker",
      "image": "ubuntu:14.04",
      "commit": true
    }
  ],
  "provisioners": [
    {
      "type": "shell",
      "inline": [
        "apt-get -y update"
      ]
    },
    {
      "type": "file",
      "source": ".",
      "destination": "/tmp/app"
    },
    {
      "type": "shell",
      "inline": [
        "cd /tmp/app",
        "make"
      ]
    },
    {
      "type": "file",
      "source": "/tmp/compiled-app.tar.gz",
      "destination": "compiled-app.tar.gz",
      "direction": "download"
    }
  ],
  "post-processors": [
    [
      {
        "type": "artifice",
        "files": ["compiled-app.tar.gz"]
      },
      {
        "type": "atlas",
        "artifact": "{{user `app_slug` }}",
        "artifact_type": "archive"
      }
    ]
  ]
}
```

In your Packer template, you can download each of the latest applications artifacts onto the host using the shell provisioner:

```
$ curl -L -H "X-Atlas-Token: ${ATLAS_TOKEN}" https://atlas.hashicorp.com/api/v1/artifacts/hashicorp/example/archive/latest/file -o example.tar.gz
```

Here's an example Packer template:

```
{
  "variables": {
```

```

variables : [
  "atlas_username": "{{env `ATLAS_USERNAME`}}",
  "aws_access_key": "{{env `AWS_ACCESS_KEY_ID`}}",
  "aws_secret_key": "{{env `AWS_SECRET_ACCESS_KEY`}}",
  "aws_region": "{{env `AWS_DEFAULT_REGION`}}",
  "instance_type": "c3.large",
  "source_ami": "ami-9a562df2",
  "name": "example",
  "ssh_username": "ubuntu",
  "app_dir": "/app"
},
"push": {
  "name": "{{user `atlas_username`}}/{{user `name`}}",
  "vcs": false
},
"builders": [
  {
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "{{user `aws_region`}}",
    "vpc_id": "",
    "subnet_id": "",
    "instance_type": "{{user `instance_type`}}",
    "source_ami": "{{user `source_ami`}}",
    "ami_regions": [],
    "ami_name": "{{user `name`}} {{timestamp}}",
    "ami_description": "{{user `name`}} AMI",
    "run_tags": { "ami-create": "{{user `name`}} " },
    "tags": { "ami": "{{user `name`}} " },
    "ssh_username": "{{user `ssh_username`}}",
    "ssh_timeout": "10m",
    "ssh_private_ip": false,
    "associate_public_ip_address": true
  }
],
"provisioners": [
  {
    "type": "shell",
    "execute_command": "echo {{user `ssh_username`}} | {{ .Vars }} sudo -E -S sh '{{ .Path }}'",
    "inline": [
      "apt-get -y update",
      "apt-get -y upgrade",
      "apt-get -y install curl unzip tar",
      "mkdir -p {{user `app_dir`}}",
      "chmod a+w {{user `app_dir`}}",
      "cd /tmp",
      "curl -L -H 'X-Atlas-Token: ${ATLAS_TOKEN}' https://atlas.hashicorp.com/api/v1/artifacts/{{user `atlas_username`}}/{{user `name`}}/archive/latest/file -o example.tar.gz",
      "tar -xzf example.tar.gz -C {{user `app_dir`}}"
    ]
  }
],
"post-processors": [
  {
    "type": "atlas",
    "artifact": "{{user `atlas_username`}}/{{user `name`}}",
    "artifact_type": "amazon.image",
    "metadata": {
      "created_at": "{{timestamp}}"
    }
  }
]
}

```

Once downloaded, you can place each application slug where it needs to go to produce the monolithic artifact your are accustom to.

Rolling Deployments

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

How do I configure rolling deployments?

User are able to quickly change out an Artifact version that is being utilized by Terraform, using variables within Terraform Enterprise. This is particularly useful when testing specific versions of the given artifact without performing a full rollout. This configuration also allows one to deploy any version of an artifact with ease, simply by changing a version variable in Terraform and re-deploying.

Here is an example:

```
variable "type"          { default = "amazon.image" }
variable "region"         {}
variable "atlas_username" {}
variable "pinned_name"   {}
variable "pinned_version" { default = "latest" }

data "atlas_artifact" "pinned" {
  name      = "${var.atlas_username}/${var.pinned_name}"
  type      = "${var.type}"
  version   = "${var.pinned_version}"

  lifecycle { create_before_destroy = true }

  metadata {
    region = "${var.region}"
  }
}

output "pinned" { value = "${atlas_artifact.pinned.metadata_full.ami_id}" }
```

In the above example we have an `atlas_artifact` resource where you pass in the version number via the variable `pinned_version`. (note: this variable defaults to `latest`). If you ever want to deploy any other version, you just update the variable `pinned_version` and redeploy.

Below is similar to the first example, but it is in the form of a module that handles the creation of artifacts:

```

variable "type"          { default = "amazon.image" }
variable "region"         {}
variable "atlas_username" {}
variable "artifact_name" {}
variable "artifact_version" { default = "latest" }

data "atlas_artifact" "artifact" {
  name    = "${var.atlas_username}/${var.artifact_name}"
  type    = "${var.type}"
  count   = "${length(split(",", var.artifact_version))}"
  version = "${element(split(",", var.artifact_version), count.index)}"

  lifecycle { create_before_destroy = true }
  metadata  { region = "${var.region}" }
}

output "amis" { value = "${join("", atlas_artifact.artifact.*.metadata_full.ami_id)}" }

```

One can then use the module as follows (*note: the source will likely be different depending on the location of the module*):

```

module "artifact_consul" {
  source = "../../modules/aws/util/artifact"

  type      = "${var.artifact_type}"
  region    = "${var.region}"
  atlas_username = "${var.atlas_username}"
  artifact_name = "${var.consul_artifact_name}"
  artifact_version = "${var.consul_artifacts}"
}

```

In the above example, we have created artifacts for Consul. In this example, we can create two versions of the artifact, "latest" and "pinned". This is useful when rolling a cluster (like Consul) one node at a time, keeping some nodes pinned to current version and others deployed with the latest Artifact.

There are additional details for implementing rolling deployments in the Best-Practices Repo (https://github.com/hashicorp/best-practices/blob/master/terraform/providers/aws/us_east_1_prod/us_east_1_prod.tf#L105-L123), as there are some things uncovered in this FAQ (i.e Using the Terraform Enterprise Artifact in an instance).

Vagrant Cloud Migration

Vagrant-related functionality will be moved from Terraform Enterprise into its own product, Vagrant Cloud. This migration is currently planned for **June 27th, 2017**.

All existing Vagrant boxes will be moved to the new system on that date. All users, organizations, and teams will be copied as well.

Authentication Tokens

No existing Terraform Enterprise authentication tokens will be transferred. To prevent a disruption of service for Vagrant-related operations, users must create a new authentication token and check "Migrate to Vagrant Cloud" and begin using these tokens for creating and modifying Vagrant boxes. These tokens will be moved on the migration date.

Creating a token via `vagrant login` will also mark a token as "Migrate to Vagrant Cloud".

More Information

At least 1 month prior to the migration, we will be releasing more information on the specifics and impact of the migration.

Glossary

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

Terraform Enterprise, and this documentation, covers a large set of terminology adopted from tools, industry standards and the community. This glossary seeks to define as many of those terms as possible to help increase understanding in interfacing with the platform and reading documentation.

Authentication Tokens

Authentication tokens are tokens used to authenticate with Terraform Enterprise via APIs or through tools. Authentication tokens can be revoked, expired or created under any user.

ACL

ACL is an acronym for access control list. This defines access to a set of resources. Access to an object in Terraform Enterprise limited to "read" for certain users is an example of an ACL.

Alert

An alert represents a health check status change on a Consul node that is sent to Terraform Enterprise, and then recorded and distributed to various notification methods.

Application

An application is a set of code that represents an application that should be deployed. Applications can be linked to builds to be made available in the Packer environment.

Apply

An apply is the second step of the two steps required for Terraform to make changes to infrastructure. The apply is the process of communicating with external APIs to make the changes.

Artifact

An artifact is an abstract representation of something you wish to store and use again that has undergone configuration, compilation or some other build process. An artifact is typically an image created by Packer that is then deployed by Terraform, or used locally with Vagrant.

Box

Boxes are a Vagrant specific package format. Vagrant can install and uses images in box format.

Build

Builds are resources that represent Packer configurations. A build is a generic name, sometimes called a "Build Configuration" when defined in the Terraform Enterprise UI.

Build Configuration

A build configuration are settings associated with a resource that creates artifacts via builds. A build configuration is the name in `packer push -name acemeinc/web`.

Catalog

The box catalog is a publicly available index of Vagrant Boxes that can be downloaded from Terraform Enterprise and used for development.

Consul

Consul (<https://consul.io>) is a HashiCorp tool for service discovery, configuration, and orchestration. Consul enables rapid deployment, configuration, monitoring and maintenance of service-oriented architectures.

Datacenter

A datacenter represents a group of nodes in the same network or datacenter within Consul.

Environment

Environments show the real-time status of your infrastructure, any pending changes, and its change history. Environments can be configured to use any or all of these three components.

Environments are the namespace of your Terraform Enterprise managed infrastructure. As an example, if you want to have a production environment for a company named Acme Inc., your environment may be named `my-username/production`.

To read more about features provided under environments, read the Terraform (/docs/enterprise) sections.

Environment Variables

Environment variables injected into the environment of Packer builds or Terraform Runs (plans and applies).

Flapping

Flapping is something entering and leaving a healthy state rapidly. It is typically associated with a health checks that briefly report unhealthy status before recovering.

Health Check

Health checks trigger alerts by changing status on a Consul node. That status change is seen by Terraform Enterprise, when connected, and an associated alert is recorded and sent to any configured notification methods, like email.

Infrastructure

An infrastructure is a stateful representation of a set of Consul datacenters.

Operator

An operator is a person who is making changes to infrastructure or settings.

Packer

Packer (<https://packer.io>) is a tool for creating images for platforms such as Amazon AWS, OpenStack, VMware, VirtualBox, Docker, and more — all from a single source configuration.

Packer Template

A Packer template is a JSON file that configure the various components of Packer in order to create one or more machine images.

Plan

A plan is the second step of the two steps required for Terraform to make changes to infrastructure. The plan is the process of determining what changes will be made to.

Providers

Providers are often referenced when discussing Packer or Terraform. Terraform providers manage resources in Terraform.
Read more (<https://terraform.io/docs/providers/index.html>).

Post-Processors

The post-processor section within a Packer template configures any post-processing that will be done to images built by the builders. Examples of post-processing would be compressing files, uploading artifacts, etc..

Registry

Often referred to as the "Artifact Registry", the registry stores artifacts, be it images or IDs for cloud provider images.

Run

A run represents a two step Terraform plan and a subsequent apply.

Service

A service in Consul represents an application or service, which could be active on any number of nodes.

Share

Shares are let you instantly share public access to your running Vagrant environment (virtual machine).

State

Terraform state is the state of your managed infrastructure from the last time Terraform was run. By default this state is stored in a local file named `terraform.tfstate`, but it can also be stored in Terraform Enterprise and is then called "Remote state".

Terraform

Terraform (<https://terraform.io>) is a tool for safely and efficiently changing infrastructure across providers.

Terraform Configuration

Terraform configuration is the configuration files and any files that may be used in provisioners like `remote-exec`.

Terraform Variables

Variables in Terraform, uploaded with `terraform push` or set in the UI. These differ from environment variables as they are a first class Terraform variable used in interpolation.

Organizations in Terraform Enterprise

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Organizations are a group of users in Terraform Enterprise that have access and ownership over shared resources. When operating within a team, we recommend creating an organization to manage access control, auditing, billing and authorization.

Each individual member of your organization should have their own account.

Set an Organization Authentication Policy

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Because organization membership affords members access to potentially sensitive resources, owners can set organization-wide authentication policy in Terraform Enterprise.

Requiring Two-Factor Authentication

Organization owners can require that all organization team members use two-factor authentication (</docs/enterprise-legacy/user-accounts/authentication.html>). Those that lack two-factor authentication will be locked out of the web interface until they enable it or leave the organization.

Visit your organization's configuration page to enable this feature. All organization owners must have two-factor authentication enabled to require the practice organization-wide. Note: locked-out users are still be able to interact with Terraform Enterprise using their ATLAS_TOKEN.

Disabling Two-Factor Authentication Requirement

Organization owners can disable the two-factor authentication requirement from their organization's configuration page. Locked-out team members (those who have not enabled two-factor authentication) will have their memberships reinstated.

Create an Organization Account

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

To create an organization:

1. Create a personal account. You'll use this to create and administrate the organization. You'll be able to add other users as owners of the organization, so it won't be tied solely to your account.
2. Visit your new organization page to create the organization.

Add credit card details to an organization

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

To setup automated billing for your Terraform usage, you must add a credit card to your organization's account. To do so, go into your account settings, then go to the proper organization settings in the left navigation. Select billing in the organization settings, and then enter your credit card information.

If you have any questions regarding billing or payment, contact sales@hashicorp.com (<mailto:sales@hashicorp.com>).

Membership

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise allows collaboration in organizations. However, there are several membership levels within organizations.

Admin

Admin is the top-most access level within Terraform Enterprise and is reserved for Terraform install admins. Admins can manage users, organizations, and jobs. For example, our support team can clear out stuck plans within Terraform environments or view organization data in order to resolve requests. Admins do not have access to sensitive variables.

Owner

Owners are the top-most access level within an organization and have complete administrative rights. An owner can manage teams and memberships, and can enforce two factor authentication across the organization.

Every organization has a special **owners** team, made up of every user that has owner permissions. The **owners** team has implied access for all of the organization's resources, but also has the ability to manage the organization.

Owners can also restrict access to resources (Terraform Environments, Packer Builds, and Artifacts). To manage access go to the Access settings for the specific resource and enter the team or collaborators name, then specify whether they should have `read`, `write`, or `admin`.

Member

Members are Terraform Enterprise users with access to your organization. Owners can organize members into teams and restrict their permissions via Access. See the documentation on access controls (</docs/enterprise-legacy/user-accounts/access.html>) for details.

Migrate Organization

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

To migrate an existing user account to an organization:

1. Create or retrieve the username of a new personal account. You'll add this account as an "owner" for the new organization during the migration process. If you already have another account, write down your username.
2. Sign in as the account you wish to migrate and visit the migration page (<https://atlas.hashicorp.com/account/migrate>).
3. Put the username of the personal account you wish to make an owner of the organization into the username text field and press "Migrate".
4. You will be logged out and receive a confirmation email with the personal account you migrated to.
5. Now, sign in with the personal account created during step 1. Visit your account settings (<https://atlas.hashicorp.com/settings/resources>). In the resource list, you should see your migrated organization available to administrate.

Start a trial

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise offers organizations 30-day trials for Terraform Enterprise (<https://www.hashicorp.com/products/terraform/>), Consul Enterprise (<https://www.hashicorp.com/consul.html>), and Vagrant Enterprise. Note that trials are available for organizations, not users.

Request a trial (<https://www.hashicorp.com/products/terraform/>) for your organization.

About Packer and Artifacts

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

Packer creates and uploads artifacts to Terraform Enterprise. This is done with the post-processor (<https://packer.io/docs/post-processors/atlas.html>).

Artifacts can then be used to deploy services or access via Vagrant. Artifacts are generic, but can be of varying types. These types define different behavior within Terraform Enterprise.

For uploading artifacts `artifact_type` can be set to any unique identifier, however, the following are recommended for consistency.

- `amazon.image`
- `azure.image`
- `digitalocean.image`
- `docker.image`
- `google.image`
- `openstack.image`
- `parallels.image`
- `qemu.image`
- `virtualbox.image`
- `vmware.image`
- `custom.image`
- `application.archive`
- `vagrant.box`

Packer can create artifacts when running in Terraform Enterprise or locally. This is possible due to the post-processors use of the public artifact API to store the artifacts.

You can read more about artifacts and their use in the Terraform section (</docs/enterprise-legacy/>) of the documentation.

Creating AMI Artifacts with Terraform Enterprise

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

In an immutable infrastructure workflow, it's important to version and store full images (artifacts) to be deployed. This section covers storing AWS AMI (<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html>) images in Terraform Enterprise to be queried and used later.

Note the actual AMI does *not get stored*. Terraform Enterprise simply keeps the AMI ID as a reference to the target image. Tools like Terraform can then use this in a deploy.

Steps

If you run Packer in Terraform Enterprise, the following will happen after a push (/docs/enterprise-legacy/packer/builds/startng.html):

1. Terraform Enterprise will run `packer build` against your template in our infrastructure. This spins up an AWS instance in your account and provisions it with any specified provisioners
2. Packer stops the instance and stores the result as an AMI in AWS under your account. This then returns an ID (the artifact) that it passes to the post-processor
3. The post-processor creates and uploads the new artifact version with the ID in Terraform Enterprise of the type `amazon.image` for use later

Example

Below is a complete example Packer template that starts an AWS instance.

```
{  
  "push": {  
    "name": "my-username/frontend"  
  },  
  "provisioners": [],  
  "builders": [  
    {  
      "type": "amazon-ebs",  
      "access_key": "",  
      "secret_key": "",  
      "region": "us-east-1",  
      "source_ami": "ami-2ccc7a44",  
      "instance_type": "c3.large",  
      "ssh_username": "ubuntu",  
      "ami_name": "Terraform Enterprise Example {{ timestamp }}"  
    }  
  ],  
  "post-processors": [  
    {  
      "type": "atlas",  
      "artifact": "my-username/web-server",  
      "artifact_type": "amazon.image"  
    }  
  ]  
}
```

Creating Vagrant Boxes with Packer

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing these features yourself.

We recommend using Packer to create boxes, as it is fully repeatable and keeps a strong history of changes within Terraform Enterprise.

Getting Started

Using Packer requires more up front effort, but the repeatable and automated builds will end any manual management of boxes. Additionally, all boxes will be stored and served from Terraform Enterprise, keeping a history along the way.

Post-Processors

Packer uses post-processors (<https://packer.io/docs/templates/post-processors.html>) to define how to process images and artifacts after provisioning. Both the `vagrant` and `atlas` post-processors must be used in order to upload Vagrant Boxes to Terraform Enterprise via Packer.

It's important that they are sequenced (<https://packer.io/docs/templates/post-processors.html>) in the Packer template so they run in order. This is done by nesting arrays:

```
{
  "post-processors": [
    [
      {
        "type": "vagrant"
        // ...
      },
      {
        "type": "atlas"
        // ...
      }
    ]
  }
}
```

Sequencing automatically passes the resulting artifact from one post-processor to the next – in this case, the `.box` file.

Vagrant Post-Processor

The Vagrant post-processor (<https://packer.io/docs/post-processors/vagrant.html>) is required to package the image from the build (an `.ovf` file, for example) into a `.box` file before passing it to the `atlas` post-processor.

```
{  
  "type": "vagrant",  
  "keep_input_artifact": false  
}
```

The input artifact (i.e and .ovf file) does not need to be kept when building Vagrant Boxes, as the resulting .box will contain it.

Post-Processor

The post-processor (<https://packer.io/docs/post-processors/atlas.html>) takes the resulting .box file and uploads it adding metadata about the box version.

```
{  
  "type": "atlas",  
  "artifact": "my-username/dev-environment",  
  "artifact_type": "vagrant.box",  
  "metadata": {  
    "provider": "vmware_desktop",  
    "version": "0.0.1"  
  }  
}
```

Attributes Required

These are all of the attributes for that post-processor required for uploading Vagrant Boxes. A complete example is shown below.

- **artifact:** The username and box name (username/name) you're creating the version of the box under. If the box doesn't exist, it will be automatically created
- **artifact_type:** This must be `vagrant.box`. Terraform Enterprise uses this to determine how to treat this artifact.

For `vagrant.box` type artifacts, you can specify keys in the metadata block:

- **provider:** The Vagrant provider for the box. Common providers are `virtualbox`, `vmware_desktop`, `aws` and so on *(required)*
- **version:** This is the Vagrant box version and is constrained to the same formatting as in the web UI: `*.*.*` *(optional, but required for boxes with multiple providers)*. *The version will increment on the minor version if left blank (e.g the initial version will be set to 0.1.0, the subsequent version will be set to 0.2.0).*
- **description:** This is the description that will be shown with the version of the box. You can use Markdown for links and style. *(optional)*

Example

An example post-processor block for Terraform Enterprise and Vagrant is below. In this example, the build runs on both VMware and Virtualbox creating two different providers for the same box version (0.0.1).

```
{  
  "post-processors": [  
    [  
      {  
        "type": "vagrant",  
        "keep_input_artifact": false  
      },  
      {  
        "type": "atlas",  
        "only": ["vmware-iso"],  
        "artifact": "my-username/dev-environment",  
        "artifact_type": "vagrant.box",  
        "metadata": {  
          "provider": "vmware_desktop",  
          "version": "0.0.1"  
        }  
      },  
      {  
        "type": "atlas",  
        "only": ["virtualbox-iso"],  
        "artifact": "my-username/dev-environment",  
        "artifact_type": "vagrant.box",  
        "metadata": {  
          "provider": "virtualbox",  
          "version": "0.0.1"  
        }  
      }  
    ]  
  ]  
}
```

About Builds

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Builds are instances of `packer build` being run within Terraform Enterprise. Every build belongs to a build configuration.

Build configurations represent a set of Packer configuration versions and builds run. It is used as a namespace within Terraform Enterprise, Packer commands and URLs. Packer configuration sent to Terraform Enterprise are stored and versioned under these build configurations.

These **versions** of Packer configuration can contain:

- The Packer template, a JSON file which define one or more builds by configuring the various components of Packer
- Any provisioning scripts or packages used by the template
- Applications that use the build as part of the pipeline and merged into the version prior to running Packer on it

When a new version of Packer configuration and associated scripts from GitHub or `packer push` is received, it automatically starts a new Packer build. That Packer build runs in an isolated machine environment with the contents of that version available to it.

You can be alerted of build events with Build Notifications (</docs/enterprise-legacy/packer/builds/notifications.html>).

Packer Build Environment

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

This page outlines the environment that Packer runs in within Terraform Enterprise.

Supported Builders

Terraform Enterprise currently supports running the following Packer builders:

- amazon-chroot
- amazon-ebs
- amazon-instance
- digitalocean
- docker
- googlecompute
- null
- openstack
- qemu
- virtualbox-iso
- vmware-iso

Files

All files in the uploading package (via Packer push or GitHub (</docs/enterprise-legacy/packer/builds/starting.html>)), and the application from the build pipeline are available on the filesystem of the build environment.

You can use the file icon on the running build to show a list of available files.

Files can be copied to the destination image Packer is provisioning with Packer Provisioners (<https://packer.io/docs/templates/provisioners.html>).

An example of this with the Shell provisioner is below.

```
{  
  "provisioners": [  
    {  
      "type": "shell",  
      "scripts": [  
        "scripts/vagrant.sh",  
        "scripts/dependencies.sh",  
        "scripts/cleanup.sh"  
      ]  
    }  
  ]  
}
```

We encourage use of relative paths over absolute paths to maintain portability between Terraform Enterprise and local builds.

The total size of all files in the package being uploaded via Packer push or GitHub (/docs/enterprise-legacy/packer/builds/start.html) must be 5 GB or less.

If you need to upload objects that are larger, such as dmgs, see the `packer push` "Limits" documentation (<https://packer.io/docs/command-line/push.html>) for ways around this limitation.

Hardware Limitations

Currently, each builder defined in the Packer template receives the following hardware resources. This is subject to change.

- 1 CPU core
- 2 GB of memory
- 20 GBs of disk space

Environment Variables

You can set any number of environment variables that will be injected into your build environment at runtime. These variables can be used to configure your build with secrets or other key value configuration.

Variables are encrypted and stored securely.

Additionally, the following environment variables are automatically injected. All injected environment variables will be prefixed with `ATLAS_`

- `ATLAS_TOKEN` - This is a unique, per-build token that expires at the end of build execution (e.g. "abcd.atlasv1.ghjkl...")
- `ATLAS_BUILD_ID` - This is a unique identifier for this build (e.g. "33")
- `ATLAS_BUILD_NUMBER` - This is a unique identifier for all builds in the same scope (e.g. "12")
- `ATLAS_BUILD_NAME` - This is the name of the build (e.g. "mybuild").
- `ATLAS_BUILD_SLUG` - This is the full name of the build (e.g. "company/mybuild").
- `ATLAS_BUILD_USERNAME` - This is the username associated with the build (e.g. "sammy")

- ATLAS_BUILD_CONFIGURATION_VERSION - This is the unique, auto-incrementing version for the Packer build configuration ([/docs/enterprise-legacy/glossary/index.html](#)) (e.g. "34").
- ATLAS_BUILD_GITHUB_BRANCH - This is the name of the branch that the associated Packer build configuration version was ingressed from (e.g. `master`).
- ATLAS_BUILD_GITHUB_COMMIT_SHA - This is the full commit hash of the commit that the associated Packer build configuration version was ingressed from (e.g. "abcd1234...").
- ATLAS_BUILD_GITHUB_TAG - This is the name of the tag that the associated Packer build configuration version was ingressed from (e.g. "`v0.1.0`").

If the build was triggered by a new application version, the following environment variables are also available:

- ATLAS_APPLICATION_NAME - This is the name of the application connected to the Packer build (e.g. "`myapp`").
- ATLAS_APPLICATION_SLUG - This is the full name of the application connected to the Packer build (e.g. "`company/myapp`").
- ATLAS_APPLICATION_USERNAME - This is the username associated with the application connected to the Packer build (e.g. "`sammy`")
- ATLAS_APPLICATION_VERSION - This is the version of the application connected to the Packer build (e.g. "2").
- ATLAS_APPLICATION_GITHUB_BRANCH - This is the name of the branch that the associated application version was ingressed from (e.g. `master`).
- ATLAS_APPLICATION_GITHUB_COMMIT_SHA - This is the full commit hash of the commit that the associated application version was ingressed from (e.g. "abcd1234...").
- ATLAS_APPLICATION_GITHUB_TAG - This is the name of the tag that the associated application version was ingressed from (e.g. "`v0.1.0`").

For any of the GITHUB_ attributes, the value of the environment variable will be the empty string ("") if the resource is not connected to GitHub or if the resource was created outside of GitHub (like using `packer push` or `vagrant push`).

Base Artifact Variable Injection

A base artifact can be selected on the "Settings" page for a build configuration. During each build, the latest artifact version will have its external ID (such as an AMI for AWS) injected as an environment variable for the environment.

The keys for the following artifact types will be injected:

- `aws.ami`: ATLAS_BASE_ARTIFACT_AWS_AMI_ID
- `amazon.ami`: ATLAS_BASE_ARTIFACT_AMAZON_AMI_ID
- `amazon.image`: ATLAS_BASE_ARTIFACT_AMAZON_IMAGE_ID
- `google.image`: ATLAS_BASE_ARTIFACT_GOOGLE_IMAGE_ID

You can then reference this artifact in your Packer template, like this AWS example:

```
{  
  "variables": {  
    "base_ami": "{{env `ATLAS_BASE_ARTIFACT_AWS_AMI_ID`}}"  
  },  
  "builders": [  
    {  
      "type": "amazon-ebs",  
      "access_key": "",  
      "secret_key": "",  
      "region": "us-east-1",  
      "source_ami": "{{user `base_ami`}}"  
    }  
  ]  
}
```

Notes on Security

Packer environment variables in Terraform Enterprise are encrypted using Vault (<https://vaultproject.io>) and closely guarded and audited. If you have questions or concerns about the safety of your configuration, please contact our security team at security@hashicorp.com (<mailto:security@hashicorp.com>).

Upgrade Guides

Terraform's major releases can include an upgrade guide to help upgrading users walk through backwards compatibility issues and changes to expect. See the navigation for the available upgrade guides.

Upgrading to Terraform v0.10

Terraform v0.10 is a major release and thus includes some changes that you'll need to consider when upgrading. This guide is intended to help with that process.

The goal of this guide is to cover the most common upgrade concerns and issues that would benefit from more explanation and background. The exhaustive list of changes will always be the Terraform Changelog (<https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md>). After reviewing this guide, we recommend reviewing the Changelog to check on specific notes about the resources and providers you use.

This guide focuses on changes from v0.9 to v0.10. Each previous major release has its own upgrade guide, so please consult the other guides (available in the navigation) if you are upgrading directly from an earlier version.

Separated Provider Plugins

As of v0.10, provider plugins are no longer included in the main Terraform distribution. Instead, they are distributed separately and installed automatically by the `terraform init` command (</docs/commands/init.html>).

In the long run, this new approach should be beneficial to anyone who wishes to upgrade a specific provider to get new functionality without also upgrading another provider that may have introduced incompatible changes. In the short term, it just means a smaller distribution package and thus avoiding the need to download tens of providers that may never be used.

Provider plugins are now also versioned separately from Terraform itself. Version constraints (</docs/configuration/providers.html#provider-versions>) can be specified in configuration to ensure that new major releases (which may have breaking changes) are not automatically installed.

Action: After upgrading, run `terraform init` in each Terraform configuration working directory to install the necessary provider plugins. If running Terraform in automation, this command should be run as the first step after a Terraform configuration is cloned from version control, and will also install any necessary modules and configure any remote backend.

Action: For "production" configurations, consider adding provider version constraints (</docs/configuration/providers.html#provider-versions>), as suggested by the `terraform init` output, to prevent new major versions of plugins from being automatically installed in future.

Third-party Provider Plugins

This initial release of separated provider plugins applies only to the providers that are packaged and released by Hashicorp. The goal is to eventually support a similar approach for third-party plugins, but we wish to ensure the robustness of the installation and versioning mechanisms before generalizing this feature.

In the mean time, the prior mechanisms for installing third-party providers (</docs/plugins/basics.html#installing-a-plugin>) are still supported. Maintainers of third-party providers may optionally make use of the new versioning mechanism by naming provider binaries using the scheme `terraform-provider-NAME_v0.0.1`, where "0.0.1" is an example version. Terraform expects providers to follow the semantic versioning (<http://semver.org/>) methodology.

Although third-party providers with versions cannot currently be automatically installed, Terraform 0.10 *will* verify that the installed version matches the constraints in configuration and produce an error if an acceptable version is unavailable.

Action: No immediate action required, but third-party plugin maintainers may optionally begin using version numbers in their binary distributions to help users deal with changes over time.

Recursive Module Targeting with -target

It is possible to target all of the resources in a particular module by passing a module address to the `-target` argument:

```
$ terraform plan -out=tfplan -target=module.example
```

Prior to 0.10, this command would target only the resources *directly* in the given module. As of 0.10, this behavior has changed such that the above command also targets resources in *descendent* modules.

For example, if `module.example` contains a module itself, called `module.examplechild`, the above command will target resources in both `module.example` *and* `module.example.module.examplechild`.

This also applies to other Terraform features that use resource addressing ([/docs/internals/resource-addressing.html](#)) syntax. This includes some of the subcommands of `terraform state` ([/docs/commands/state/index.html](#)).

Action: If running Terraform with `-target` in automation, review usage to ensure that selecting additional resources in child modules will not have ill effects. Be sure to review plan output when `-target` is used to verify that only the desired resources have been targeted for operations. Please note that it is not recommended to routinely use `-target`; it is provided for exceptional uses and manual intervention.

Interactive Approval in `terraform apply`

Starting with Terraform 0.10 `terraform apply` has a new mode where it will present the plan, pause for interactive confirmation, and then apply the plan only if confirmed. This is intended to get similar benefits to separately running `terraform plan`, but to streamline the workflow for interactive command-line use.

For 0.10 this feature is disabled by default, to avoid breaking any wrapper scripts that are expecting the old behavior. To opt-in to this behavior, pass `-auto-approve=false` when running `terraform apply` without an explicit plan file.

It is planned that a future version of Terraform will make this behavior the default. Although no immediate action is required, we strongly recommend adjusting any Terraform automation or wrapper scripts to prepare for this upcoming change in behavior, in the following ways:

- Non-interactive automation around production systems should *always* separately run `terraform plan -out=tfplan` and then (after approval) `terraform apply tfplan`, to ensure operators have a chance to review the plan before applying it.
- If running `terraform apply` *without* a plan file in automation for a *non-production* system, add `-auto-approve=true` to the command line soon, to preserve the current 0.10 behavior once auto-approval is no longer enabled by default.

We are using a staged deprecation for this change because we are aware that many teams use Terraform in wrapper scripts and automation, and we wish to ensure that such teams have an opportunity to update those tools in preparation for the future change in behavior.

Action: 0.10 preserves the previous behavior as the default, so no immediate action is required. However, maintainers of tools that wrap Terraform, either in automation or in alternative command-line UI, should consider which behavior is appropriate for their use-case and explicitly set the `-auto-approve=...` flag to ensure that behavior in future versions.

Upgrading to Terraform v0.11

Terraform v0.11 is a major release and thus includes some changes that you'll need to consider when upgrading. This guide is intended to help with that process.

The goal of this guide is to cover the most common upgrade concerns and issues that would benefit from more explanation and background. The exhaustive list of changes will always be the Terraform Changelog (<https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md>). After reviewing this guide, we recommend reviewing the Changelog to check on specific notes about the resources and providers you use.

This guide focuses on changes from v0.10 to v0.11. Each previous major release has its own upgrade guide, so please consult the other guides (available in the navigation) if you are upgrading directly from an earlier version.

Interactive Approval in `terraform apply`

Terraform 0.10 introduced a new mode for `terraform apply` (when run without an explicit plan file) where it would show a plan and prompt for approval before proceeding, similar to `terraform destroy`.

Terraform 0.11 adopts this as the default behavior for this command, which means that for interactive use in a terminal it is not necessary to separately run `terraform plan -out=...` to safely review and apply a plan.

The new behavior also has the additional advantage that, when using a backend that supports locking, the state lock will be held throughout the refresh, plan, confirmation and apply steps, ensuring that a concurrent execution of `terraform apply` will not invalidate the execution plan.

A consequence of this change is that `terraform apply` is now interactive by default unless a plan file is provided on the command line. When running Terraform in automation (/guides/running-terraform-in-automation.html) it is always recommended to separate plan from apply, but if existing automation was running `terraform apply` with no arguments it may now be necessary to update it to either generate an explicit plan using `terraform plan -out=...` or to run `terraform apply -auto-approve` to bypass the interactive confirmation step. The latter should be done only in unimportant environments.

Action: For interactive use in a terminal, prefer to use `terraform apply` with out an explicit plan argument rather than `terraform plan -out=tfplan` followed by `terraform apply tfplan`.

Action: Update any automation scripts that run Terraform non-interactively so that they either use separated plan and apply or override the confirmation behavior using the `-auto-approve` option.

Relative Paths in Module source

Terraform 0.11 introduces full support for module installation from Terraform Registry (<https://registry.terraform.io/>) as well as other private, in-house registries using concise module source strings like `hashicorp/consul/aws`.

As a consequence, module source strings like "child" are no longer interpreted as relative paths. Instead, relative paths must be expressed explicitly by beginning the string with either `./` (for a module in a child directory) or `../` (for a module in the parent directory).

Action: Update existing module source values containing relative paths to start either `./` or `../` to prevent misinterpretation of the source as a Terraform Registry module.

Interactions Between Providers and Modules

Prior to Terraform 0.11 there were several limitations in deficiencies in how providers interact with child modules, such as:

- Ancestor module provider configurations always overrode the associated settings in descendent modules.
- There was no well-defined mechanism for passing "aliased" providers from an ancestor module to a descendent, where the descendent needs access to multiple provider instances.

Terraform 0.11 changes some of the details of how each resource block is associated with a provider configuration, which may change how Terraform interprets existing configurations. This is notably true in the following situations:

- If the same provider is configured in both an ancestor and a descendent module, the ancestor configuration no longer overrides attributes from the descendent and the descendent no longer inherits attributes from its ancestor. Instead, each configuration is entirely distinct.
- If a provider block is present in a child module, it must either contain a complete configuration for its associated provider or a configuration must be passed from the parent module using the new providers attribute (/docs/modules/usage.html#providers-within-modules). In the latter case, an empty provider block is a placeholder that declares that the child module requires a configuration to be passed from its parent.
- When a module containing its own provider blocks is removed from its parent module, Terraform will no longer attempt to associate it with another provider of the same name in a parent module, since that would often cause undesirable effects such as attempting to refresh resources in the wrong region. Instead, the resources in the module resources must be explicitly destroyed *before* removing the module, so that the provider configuration is still available: `terraform destroy -target=module.example`.

The recommended design pattern moving forward is to place all explicit provider blocks in the root module of the configuration, and to pass providers explicitly to child modules so that the associations are obvious from configuration:

```
provider "aws" {
  region = "us-east-1"
  alias   = "use1"
}

provider "aws" {
  region = "us-west-1"
  alias   = "usw1"
}

module "example-use1" {
  source = "./example"

  providers = {
    "aws" = "aws.use1"
  }
}

module "example-usw1" {
  source = "./example"

  providers = {
    "aws" = "aws.usw1"
  }
}
```

With the above configuration, any aws provider resources in the module `./example` will use the us-east-1 provider configuration for `module.example-use1` and the us-west-1 provider configuration for `module.example-usw1`.

When a default (non-aliased) provider is used, and not explicitly declared in a child module, automatic inheritance of that provider is still supported.

Action: In existing configurations where both a descendent module and one of its ancestor modules both configure the same provider, copy any settings from the ancestor into the descendent because provider configurations now inherit only as a whole, rather than on a per-argument basis.

Action: In existing configurations where a descendent module inherits *aliased* providers from an ancestor module, use the new `providers` attribute ([/docs/modules/usage.html#providers-within-modules](#)) to explicitly pass those aliased providers.

Action: Consider refactoring existing configurations so that all provider configurations are set in the root module and passed explicitly to child modules, as described in the following section.

Moving Provider Configurations to the Root Module

With the new provider inheritance model, it is strongly recommended to refactor any configuration where child modules define their own `provider` blocks so that all explicit configuration is defined in the *root* module. This approach will ensure that removing a module from the configuration will not cause any provider configurations to be removed along with it, and thus ensure that all of the module's resources can be successfully refreshed and destroyed.

A common configuration is where two child modules have different configurations for the same provider, like this:

```
# root.tf

module "network-use1" {
  source = "./network"
  region = "us-east-1"
}

module "network-usw2" {
  source = "./network"
  region = "us-west-2"
}
```

```
# network/network.tf

variable "region" {}

provider "aws" {
  region = "${var.region}"
}

resource "aws_vpc" "example" {
  # ...
}
```

The above example is problematic because removing either `module.network-use1` or `module.network-usw2` from the root module will make the corresponding provider configuration no longer available, as described in issue #15762 (<https://github.com/hashicorp/terraform/issues/15762>), which prevents Terraform from refreshing or destroying that module's `aws_vpc.example` resource.

This can be addressed by moving the provider blocks into the root module as *additional configurations*, and then passing them down to the child modules as *default configurations* via the explicit providers map:

```
# root.tf

provider "aws" {
  region = "us-east-1"
  alias   = "use1"
}

provider "aws" {
  region = "us-west-2"
  alias   = "usw2"
}

module "network-use1" {
  source = "./network"

  providers = {
    "aws" = "aws.use1"
  }
}

module "network-usw2" {
  source = "./network"

  providers = {
    "aws" = "aws.usw2"
  }
}
```

```
# network/network.tf

# Empty provider block signals that we expect a default (unaliased) "aws"
# provider to be passed in from the caller.
provider "aws" {}

resource "aws_vpc" "example" {
  # ...
}
```

After the above refactoring, run `terraform apply` to re-synchronize Terraform's record (in the Terraform state (`/docs/state/index.html`)) of the location of each resource's provider configuration. This should make no changes to actual infrastructure, since no resource configurations were changed.

For more details on the explicit providers map, and discussion of more complex possibilities such as child modules with additional (aliased) provider configurations, see *Providers Within Modules* (`/docs/modules/usage.html#providers-within-modules`).

Error Checking for Output Values

Prior to Terraform 0.11, if an error occurred when evaluating the value expression within an output block then it would be silently ignored and the empty string used as the result. This was inconvenient because it made it very hard to debug errors within output expressions.

To give better feedback, Terraform now halts and displays an error message when such errors occur, similar to the behavior for expressions elsewhere in the configuration.

Unfortunately, this means that existing configurations may have erroneous outputs lurking that will become fatal errors after upgrading to Terraform 0.11. The prior behavior is no longer available; to apply such a configuration with Terraform 0.11 will require adjusting the configuration to avoid the error.

Action: If any existing output value expressions contain errors, change these expressions to fix the error.

Referencing Attributes from Resources with count = 0

A common pattern for conditional resources is to conditionally set count to either 0 or 1 depending on the result of a boolean expression:

```
resource "aws_instance" "example" {
  count = "${var.create_instance ? 1 : 0}"

  # ...
}
```

When using this pattern, it's required to use a special idiom to access attributes of this resource to account for the case where no resource is created at all:

```
output "instance_id" {
  value = "${element(concat(aws_instance.example.*.id, list(""))), 0}"
}
```

Accessing aws_instance.example.id directly is an error when count = 0. This is true for all situations where interpolation expressions are allowed, but previously *appeared* to work for outputs due to the suppression of the error. Existing outputs that access non-existent resources must be updated to use the idiom above after upgrading to 0.11.0.

Upgrading to Terraform v0.12

Terraform 0.12 has not yet been released. This guide is proactive to help users understand what the upgrade path to 0.12 will be like. This guide will be updated with more detail up until the release of 0.12.

Terraform v0.12 will be a major release (<https://hashicorp.com/blog/terraform-0-1-2-preview>) focused on configuration language improvements and thus will include some changes that you'll need to consider when upgrading. The goal of this guide is to cover the most common upgrade concerns and issues. For the majority of users, no steps will need to be taken to upgrade. The sections below explain which users are likely to be in the small group who will need to make manual changes to upgrade to 0.12.

This guide focuses on changes from v0.11 to v0.12. Each previous major release has its own upgrade guide, so please consult the other guides (available in the navigation) if you are upgrading directly from an earlier version.

Upgrading Terraform configuration

The majority of users will not need to make manual changes to their Terraform configurations to upgrade to 0.12. The users who will need to make manual changes are users who use language workarounds in previous Terraform versions. Examples of these workarounds include:

- Treating block types like attributes in an attempt to work around Terraform not supporting generating nested blocks dynamically. (#7034 (<https://github.com/hashicorp/terraform/issues/7034>))
- Wrapping redundant list brackets ([and]) around splat expressions in order to force them to be interpreted as lists even when there are unknown items in the list.

Note that these workarounds are not "wrong", but rather clever solutions by dedicated community members! These folks have been the inspiration for HCL2 and these solutions have given guidance on how to make the Terraform language more flexible to meet the needs of complex infrastructure.

Terraform 0.12 will be released with a migration tool that will make most of the required updates automatically, and also provide guidance on any changes that require human input.

For users who follow the examples in the Terraform documentation, there should be no required changes. However, we still recommend to run the migration tool to upgrade to the more readable syntax conventions supported in this release, and to draw attention to any potential issues.

Upgrading Terraform providers

We've updated the RPC protocol used by Terraform plugins to support typed data and schema transfer.

In Terraform 0.12 Terraform will have an awareness of the schemas used by both provider and provisioner plugins. This V1 for the RPC plugin protocols will still use the old-style passing of `map[string]interface{}` for config and `map[string]string` with `flatmap` for state and diff.

To avoid the need to atomically upgrade both Terraform Core and the providers all at once, new provider versions will be released that are compatible with both v0.12 and prior versions. Users will need to upgrade to the new v0.12-compatible provider releases before upgrading Terraform Core, but can use these newer provider releases with prior Terraform

versions to transition gradually and reduce risk.

This compatibility support will be handled automatically by a new version of the plugin SDK so that provider maintainers need only to upgrade to the latest version and rebuild.

The dual-protocol compatibility will be retained at least until the next major release of the plugin SDK, in order to give users time to perform a gradual upgrade of each provider and of Terraform Core itself.

Upgrading modules

Module authors will need to complete several steps to get their modules ready for v0.12.

1. Follow the steps in "Upgrading Terraform configurations" above to get the module code upgraded
2. The migration tool will automatically add a `>= 0.12.0` Terraform version constraint to indicate that the module has been upgraded to use v0.12-only features.
3. If the module is published in a module registry, publish a new major version of the module to indicate that the new version is not compatible with older versions of Terraform. If you are not using a registry, be sure that downstream consumers of the module are aware of the update.

Module consumers can then upgrade to the new versions of the module by upgrading their configurations to 0.12 and updating the module version constraint in each configuration to refer to the new major version.

Upgrading Sentinel policies

Terraform Enterprise users of Sentinel will need to complete the below steps to upgrade Sentinel to work with Terraform 0.12.

1. Update Terraform configurations to 0.12
2. Update Sentinel policies

Because Sentinel is applied across all workspaces in Terraform Enterprise, all workspaces must be upgraded to Terraform 0.12 otherwise Sentinel policies will fail on versions below 0.12.

More details on this upgrade process will be added prior to the final release.

Upgrading to Terraform v0.7

Terraform v0.7 is a major release, and thus includes some backwards incompatibilities that you'll need to consider when upgrading. This guide is meant to help with that process.

The goal of this guide is to cover the most common upgrade concerns and issues that would benefit from more explanation and background. The exhaustive list of changes will always be the Terraform Changelog (<https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md>). After reviewing this guide, review the Changelog to check on specific notes about the resources and providers you use.

Plugin Binaries

Before v0.7, Terraform's built-in plugins for providers and provisioners were each distributed as separate binaries.

```
terraform          # core binary
terraform-provider-* # provider plugins
terraform-provisioner-* # provisioner plugins
```

These binaries needed to all be extracted to somewhere in your \$PATH or in the ~/.terraform.d directory for Terraform to work.

As of v0.7, all built-in plugins ship embedded in a single binary. This means that if you just extract the v0.7 archive into a path, you may still have the old separate binaries in your \$PATH. You'll need to remove them manually.

For example, if you keep Terraform binaries in /usr/local/bin you can clear out the old external binaries like this:

```
rm /usr/local/bin/terraform-*
```

External plugin binaries continue to work using the same pattern, but due to updates to the RPC protocol, they will need to be recompiled to be compatible with Terraform v0.7.x.

Maps in Displayed Plans

When displaying a plan, Terraform now distinguishes attributes of type map by using a % character for the "length field".

Here is an example showing a diff that includes both a list and a map:

```
somelist.#: "0" => "1"
somelist.0: "" => "someitem"
somedmap.%: "0" => "1"
somedmap.foo: "" => "bar"
```

Interpolation Changes

There are a few changes to Terraform's interpolation language that may require updates to your configs.

String Concatenation

The `concat()` interpolation function used to work for both lists and strings. It now only works for lists.

```
"${concat(var.foo, "-suffix")}"      # => Error! No longer supported.
```

Instead, you can use variable interpolation for string concatenation.

```
"${var.foo}-suffix"
```

Nested Quotes and Escaping

Escaped quotes inside of interpolations were supported to retain backwards compatibility with older versions of Terraform that allowed them.

Now, escaped quotes will no longer work in the interpolation context:

```
"${lookup(var.somemap, \"somekey\")}"      # => Syntax Error!
```

Instead, treat each set of interpolation braces (`{}{}`) as a new quoting context:

```
"${lookup(var.somemap, "somekey")}"
```

This allows double quote characters to be expressed properly within strings inside of interpolation expressions:

```
"${upper(\"quoted\")}"      # => "QUOTED"
```

Safer terraform plan Behavior

Prior to v0.7, the `terraform plan` command had the potential to write updates to the state if changes were detected during the Refresh step (which happens by default during `plan`). Some configurations have metadata that changes with every read, so Refresh would always result in changes to the state, and therefore a write.

In collaborative environments with shared remote state, this potential side effect of `plan` would cause unnecessary contention over the state, and potentially even interfere with active `apply` operations if they were happening simultaneously elsewhere.

Terraform v0.7 addresses this by changing the Refresh process that is run during `terraform plan` to always be an in-memory only refresh. New state information detected during this step will not be persisted to permanent state storage.

If the `-out` flag is used to produce a Plan File, the updated state information *will* be encoded into that file, so that the resulting `terraform apply` operation can detect if any changes occurred that might invalidate the plan.

For most users, this change will not affect your day-to-day usage of Terraform. For users with automation that relies on the old side effect of `plan`, you can use the `terraform refresh` command, which will still persist any changes it discovers.

Migrating to Data Sources

With the addition of Data Sources (/docs/configuration/data-sources.html), there are several resources that were acting as Data Sources that are now deprecated. Existing configurations will continue to work, but will print a deprecation warning when a data source is used as a resource.

- atlas_artifact
- template_file
- template_cloudinit_config
- tls_cert_request

Migrating to the equivalent Data Source is as simple as changing the `resource` keyword to `data` in your declaration and prepending `data.` to attribute references elsewhere in your config.

For example, given a config like:

```
resource "template_file" "example" {  
    template = "someconfig"  
}  
resource "aws_instance" "example" {  
    user_data = "${template_file.example.rendered}"  
    # ...  
}
```

A config using the equivalent Data Source would look like this:

```
data "template_file" "example" {  
    template = "someconfig"  
}  
resource "aws_instance" "example" {  
    user_data = "${data.template_file.example.rendered}"  
    # ...  
}
```

Referencing remote state outputs has also changed. The `.output` keyword is no longer required.

For example, a config like this:

```
resource "terraform_remote_state" "example" {  
    # ...  
}  
  
resource "aws_instance" "example" {  
    ami = "${terraform_remote_state.example.output.ami_id}"  
    # ...  
}
```

Would now look like this:

```

data "terraform_remote_state" "example" {
  # ...
}

resource "aws_instance" "example" {
  ami = "${data.terraform_remote_state.example.ami_id}"
  # ...
}

```

Migrating to native lists and maps

Terraform 0.7 now supports lists and maps as first-class constructs. Although the patterns commonly used in previous versions still work (excepting any compatibility notes), there are now patterns with cleaner syntax available.

For example, a common pattern for exporting a list of values from a module was to use an output with a `join()` interpolation, like this:

```

output "private_subnets" {
  value = "${join(", ", aws_subnet.private.*.id)}"
}

```

When using the value produced by this output in another module, a corresponding `split()` would be used to retrieve individual elements, often parameterized by `count.index`, for example:

```
subnet_id = "${element(split("", var.private_subnets), count.index)}"
```

Using Terraform 0.7, list values can now be passed between modules directly. The above example can read like this for the output:

```

output "private_subnets" {
  value = ["${aws_subnet.private.*.id}"]
}

```

And then when passed to another module as a list type variable, we can index directly using `[]` syntax:

```
subnet_id = "${var.private_subnets[count.index]}"
```

Note that indexing syntax does not wrap around if the extent of a list is reached - for example if you are trying to distribute 10 instances across three private subnets. For this behaviour, `element` can still be used:

```
subnet_id = "${element(var.private_subnets, count.index)}"
```

Map value overrides

Previously, individual elements in a map could be overridden by using a dot notation. For example, if the following variable was declared:

```
variable "amis" {
  type = "map"
  default = {
    us-east-1 = "ami-123456"
    us-west-2 = "ami-456789"
    eu-west-1 = "ami-789123"
  }
}
```

The key "us-west-2" could be overridden using `-var "amis.us-west-2=overridden_value"` (or equivalent in an environment variable or `tfvars` file). The syntax for this has now changed - instead maps from the command line will be merged with the default value, with maps from flags taking precedence. The syntax for overriding individual values is now:

```
-var 'amis = { us-west-2 = "overridden_value" }'
```

This will give the map the effective value:

```
{
  us-east-1 = "ami-123456"
  us-west-2 = "overridden_value"
  eu-west-1 = "ami-789123"
}
```

It's also possible to override the values in a variables file, either in any `terraform.tfvars` file, an `.auto.tfvars` file, or specified using the `-var-file` flag.

Upgrading to Terraform v0.8

Terraform v0.8 is a major release and thus includes some backwards incompatibilities that you'll need to consider when upgrading. This guide is meant to help with that process.

The goal of this guide is to cover the most common upgrade concerns and issues that would benefit from more explanation and background. The exhaustive list of changes will always be the Terraform Changelog (<https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md>). After reviewing this guide, we recommend reviewing the Changelog to check on specific notes about the resources and providers you use.

Newlines in Strings

Newlines are no longer allowed in strings unless it is a heredoc or an interpolation. This improves the performance of IDE syntax highlighting of Terraform configurations and simplifies parsing.

Behavior that no longer works in Terraform 0.8:

```
resource "null_resource" "foo" {
  value = "foo
bar"
}
```

Valid Terraform 0.8 configuration:

```
resource "null_resource" "foo" {
  value = "foo\nbar"

  value2 = <<EOF
foo
bar
EOF

  # You can still have newlines within interpolations.
  value3 = "${lookup(
    var.foo, var.bar)}"
}
```

Action: Use heredocs or escape sequences when you have a string with newlines.

Math Order of Operations

Math operations now follow standard mathematical order of operations. Prior to 0.8, math ordering was simply left-to-right. With 0.8, *, /, and % are done before +, -.

Some examples are shown below:

```
 ${1+5*2}  => 11 (was 12 in 0.7)
 ${4/2*5}  => 10 (was 10 in 0.7)
 ${(1+5)*2} => 12 (was 12 in 0.7)
```

Action: Use parentheses where necessary to be explicit about ordering.

Escaped Variables in Templates

The `template_file` resource now requires that any variables specified in an inline `template` attribute are now escaped. This *does not affect* templates read from files either via `file()` or the `filename` attribute.

Inline variables must be escaped using two dollar signs. `${foo}` turns into `$$ {foo}`.

This is necessary so that Terraform doesn't try to interpolate the values before executing the template (for example using standard Terraform interpolations). In Terraform 0.7, we had special case handling to ignore templates, but this would cause confusion and poor error messages. Terraform 0.8 requires explicitly escaping variables.

Behavior that no longer works in Terraform 0.8:

```
data "template_file" "foo" {
  template = "${foo}"

  vars { foo = "value" }
}
```

Valid Terraform 0.8 template:

```
data "template_file" "foo" {
  template = "$${foo}"

  vars { foo = "value" }
}
```

Action: Escape variables in inline templates in `template_file` resources.

Escape Sequences Within Interpolations

Values within interpolations now only need to be escaped once.

The exact behavior prior to 0.8 was inconsistent. In many cases, users just added `\` until it happened to work. The behavior is now consistent: single escape any values that need to be escaped.

For example:

```
 ${replace(var.foo, "\\", "\\\\"})}
```

This will now replace `\` with `\\\` throughout `var.foo`. Note that `\` and `\\\` are escaped exactly once. Prior to 0.8, this required double the escape sequences to function properly.

A less complicated example:

```
 ${replace(var.foo, "\n", "")}
```

This does what you expect by replacing newlines with empty strings. Prior to 0.8, you'd have to specify `\n`, which could be confusing.

Action: Escape sequences within interpolations only need to be escaped once.

New Internal Graphs

The core graphs used to execute Terraform operations have been changed to support new features. These require no configuration changes and should work as normal.

They were tested extensively during 0.7.x behind experimental flags and using the shadow graph. However, it is possible that there are still edge cases that aren't properly handled.

While we believe it will be unlikely, if you find that something is not working properly, you may use the `-Xlegacy-graph` flag on any Terraform operation to use the old code path.

This flag will be removed prior to 0.9 (the next major release after 0.8), so please report any issues that require this flag so we can make sure they become fixed.

Warning: Some features (such as `depends_on` referencing modules) do not work on the legacy graph code path.

Specifically, any features introduced in Terraform 0.8 won't work with the legacy code path. These features will only work with the new, default graphs introduced with Terraform 0.8.

Upgrading to Terraform v0.9

Terraform v0.9 is a major release and thus includes some changes that you'll need to consider when upgrading. This guide is meant to help with that process.

The goal of this guide is to cover the most common upgrade concerns and issues that would benefit from more explanation and background. The exhaustive list of changes will always be the Terraform Changelog (<https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md>). After reviewing this guide, we recommend reviewing the Changelog to check on specific notes about the resources and providers you use.

Remote State

Remote state has been overhauled to be easier and safer to configure and use. **The new changes are backwards compatible** with existing remote state and you'll be prompted to migrate to the new remote backend system.

An in-depth guide for migrating to the new backend system is available here (</docs/backends/legacy-0-8.html>). This includes backing up your existing remote state and also rolling back if necessary.

The only non-backwards compatible change is in the CLI: the existing `terraform remote config` command is now gone. Remote state is now configured via the "backend" section within the Terraform configuration itself.

Example configuring a Consul remote backend:

```
terraform {  
  backend "consul" {  
    address      = "demo.consul.io"  
    datacenter   = "nyc3"  
    path         = "tfdemo"  
    scheme       = "https"  
  }  
}
```

Action: Nothing immediately, everything will continue working except scripts using `terraform remote config`. As soon as possible, upgrade to backends (</docs/backends/legacy-0-8.html>).

State Locking

Terraform 0.9 now will acquire a lock for your state if your backend supports it. **This change is backwards compatible**, but may require enhanced permissions for the authentication used with your backend.

Backends that support locking as of the 0.9.0 release are: local files, Amazon S3, HashiCorp Consul, and Terraform Enterprise (atlas). If you don't use these backends, you can ignore this section.

Specific notes for each affected backend:

- **Amazon S3:** DynamoDB is used for locking. The AWS access keys must have access to Dynamo. You may disable locking by omitting the `lock_table` key in your backend configuration.
- **HashiCorp Consul:** Sessions are used for locking. If an auth token is used it must have permissions to create and destroy sessions. You may disable locking by specifying `lock = false` in your backend configuration.

Action: Update your credentials or configuration if necessary.

How Packer Builds Run in Terraform Enterprise

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

This briefly covers the internal process of running builds in Terraform Enterprise. It's not necessary to know this information, but may be valuable to help understand implications of running or debugging failing builds.

Steps of Execution

1. A Packer template and directory of files is uploaded via Packer Push or GitHub
2. Terraform Enterprise creates a version of the build configuration and waits for the upload to complete. At this point, the version will be visible in the UI even if the upload has not completed
3. Once the upload finishes, the build is queued. This is potentially split across multiple machines for faster processing
4. In the build environment, the package including the files and Packer template are downloaded
5. `packer build` is run against the template in the build environment
6. Logs are streamed into the UI and stored
7. Any artifacts as part of the build are then uploaded via the public artifact API, as they would be if Packer was executed locally
8. The build completes, the environment is teared down and status updated

Installing Software

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Please review the Packer Build Environment (/docs/enterprise-legacy/packer/builds/build-environment.html) specification for important information on isolation, security, and hardware limitations before continuing.

In some cases, it may be necessary to install custom software to build your artifact using Packer. The easiest way to install software on the Packer builder is via the `shell-local` provisioner. This will execute commands on the host machine running Packer.

```
{
  "provisioners": [
    {
      "type": "shell-local",
      "command": "sudo apt-get install -y customsoftware"
    }
  ]
}
```

Please note that nothing is persisted between Packer builds, so you will need to install custom software on each run.

The Packer builders run the latest version of Ubuntu LTS.

Managing Packer Versions

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Terraform Enterprise does not automatically upgrade the version of Packer used to run builds or compiles. This is intentional, as occasionally there can be backwards incompatible changes made to Packer that cause templates to stop building properly, or new versions that produce some other unexpected behavior.

All upgrades must be performed by a user, but Terraform Enterprise will display a notice above any builds run with out of date versions. We encourage the use of the latest version when possible.

Upgrading Packer

1. Go the Settings tab of a build configuration or application
2. Go to the "Packer Version" section and select the version you wish to use
3. Review the changelog for that version and previous versions
4. Click the save button. At this point, future builds will use that version

About Packer Build Notifications

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Terraform Enterprise can send build notifications to your organization for the following events:

- **Starting** - The build has begun.
- **Finished** - All build jobs have finished successfully.
- **Errored** - An error has occurred during one of the build jobs.
- **Canceled** - A user has canceled the build.

Emails will include logs for the **Finished** and **Errored** events.

You can toggle notifications for each of these events on the "Integrations" tab of a build configuration.

Rebuilding Builds

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Sometimes builds fail due to temporary or remotely controlled conditions.

In this case, it may make sense to "rebuild" a Packer build. To do so, visit the build you wish to run again and click the Rebuild button. This will take that exact version of configuration and run it again.

You can rebuild at any point in history, but this may cause side effects that are not wanted. For example, if you were to rebuild an old version of a build, it may create the next version of an artifact that is then released, causing a rollback of your configuration to occur.

Schedule Periodic Builds in Terraform Enterprise

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Terraform Enterprise can automatically run a Packer build and create artifacts on a specified schedule. This option is disabled by default and can be enabled by an organization owner on a per-environment (</docs/enterprise-legacy/glossary#environment>) basis.

On the specified interval, builds will be automatically queued that run Packer for you, creating any artifacts and sending the appropriate notifications.

If your artifacts are used in any other environments and you have activated the plan on artifact upload feature, this may also queue Terraform plans.

This feature is useful for maintenance of images and automatic updates, or to build nightly style images for staging or development environments.

Enabling Periodic Builds

To enable periodic builds for a build, visit the build settings page and select the desired interval and click the save button to persist the changes. An initial build may immediately run, depending on the history, and then will automatically build at the specified interval.

If you have run a build separately, either manually or triggered from GitHub or Packer configuration version uploads, Terraform Enterprise will not queue a new build until the allowed time after the manual build ran. This ensures that a build has been executed at the specified schedule.

Starting Packer Builds in Terraform Enterprise

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (/docs/enterprise/upgrade/index.html) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Packer builds can be started in two ways: `packer push` to upload the template and directory or via a GitHub connection that retrieves the contents of a repository after changes to the default branch (usually master).

Packer Push

Packer push is a Packer command (<https://packer.io/docs/command-line/push.html>) that packages and uploads a Packer template and directory. This then creates a build which performs `packer build` against the uploaded template and packaged directory.

The directory is included in order to run any associated provisioners, builds or post-processors that all might use local files. For example, a shell script or set of Puppet modules used in a Packer build needs to be part of the upload for Packer to be run remotely.

By default, everything in your directory is uploaded as part of the push.

However, it's not always the case that the entire directory should be uploaded. Often, temporary or cache directories and files like `.git`, `.tmp` will be included by default. This can cause builds to fail at certain sizes and should be avoided. You can specify exclusions (<https://packer.io/docs/templates/push.html#exclude>) to avoid this situation.

Packer also allows for a VCS option (<https://packer.io/docs/templates/push.html#vcs>) that will detect your VCS (if there is one) and only upload the files that are tracked by the VCS. This is useful for automatically excluding ignored files. In a VCS like git, this basically does a `git ls-files`.

GitHub Webhooks

Optionally, GitHub can be used to import Packer templates and configurations. When used within an organization, this can be extremely valuable for keeping differences in environments and last mile changes from occurring before an upload.

After you have connected your GitHub account (/docs/enterprise-legacy/vcs/github.html) to Terraform Enterprise, you can connect your Build Configuration (/docs/enterprise-legacy/glossary#build-configuration) to the target GitHub repository. The GitHub repository will be linked to the Packer configuration, and GitHub will start sending webhooks. Certain GitHub webhook events, detailed below, will cause the repository to be automatically ingressed into Terraform Enterprise and stored, along with references to the GitHub commits and authorship information.

After each ingress the configuration will automatically build.

You can disable an ingress by adding the text `[atlas skip]` or `[ci skip]` to your commit message.

Supported GitHub webhook events:

- `push` (on by default)
 - `ingress` when a tag is created

- ingress when the default branch is updated
- note: the default branch is either configured on your configuration's integrations tab in Terraform Enterprise, or if that is blank it is the GitHub repository's default branch
- create (off by default)
 - ingress when a tag is created
 - note: if you want to only run on tag creation, turn on create events and turn off push events

Troubleshooting Failing Builds

Deprecation warning: The Packer, Artifact Registry and Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (</docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise and our guide on building immutable infrastructure with Packer on CI/CD (<https://www.packer.io/guides/packer-on-cicd/>) for ideas on implementing the Packer and Artifact features yourself.

Packer builds can fail in Terraform Enterprise for a number of reasons – improper configuration, transient networking errors, and hardware constraints are all possible. Below is a list of debugging options you can use.

Verbose Packer Logging

You can set a variable (</docs/enterprise-legacy/packer/builds/build-environment.html#environment-variables>) in the UI that increases the logging verbosity in Packer. Set the `PACKER_LOG` key to a value of 1 to accomplish this.

After setting the variable, you'll need to rebuild (</docs/enterprise-legacy/packer/builds/rebuilding.html>).

Verbose logging will be much louder than normal Packer logs and isn't recommended for day-to-day operations. Once enabled, you'll be able to see in further detail why things failed or what operations Packer was performing.

This can also be used locally:

```
$ PACKER_LOG=1 packer build ...
```

Hanging Builds

Some VM builds, such as VMware or VirtualBox, may hang at various stages, most notably Waiting for SSH....

Things to pay attention to when this happens:

- SSH credentials must be properly configured. AWS keypairs should match, SSH usernames should be correct, passwords should match, etc.
- Any VM pre-seed configuration should have the same SSH configuration as your template defines

A good way to debug this is to manually attempt to use the same SSH configuration locally, running with `packer build --debug`. See more about debugging Packer builds (<https://packer.io/docs/other/debugging.html>).

Hardware Limitations

Your build may be failing by requesting larger memory or disk usage than is available. Read more about the build environment (</docs/enterprise-legacy/packer/builds/build-environment.html#hardware-limitations>).

Typically Packer builds that fail due to requesting hardware limits that exceed Terraform Enterprise's hardware limitations (</docs/enterprise-legacy/packer/builds/build-environment.html#hardware-limitations>) will fail with a *The operation was canceled* error message as shown below:

```
# ...
==> vmware-iso: Starting virtual machine...
vmware-iso: The VM will be run headless, without a GUI. If you want to
vmware-iso: view the screen of the VM, connect via VNC without a password to
vmware-iso: 127.0.0.1:5918
==> vmware-iso: Error starting VM: VMware error: Error: The operation was canceled
==> vmware-iso: Waiting 4.604392397s to give VMware time to clean up...
==> vmware-iso: Deleting output directory...
Build 'vmware-iso' errored: Error starting VM: VMware error: Error: The operation was canceled

==> Some builds didn't complete successfully and had errors:
--> vmware-iso: Error starting VM: VMware error: Error: The operation was canceled
```

Local Debugging

Sometimes it's faster to debug failing builds locally. In this case, you'll want to install Packer (<https://www.packer.io/intro/getting-started/install.html>) and any providers (like Virtualbox) necessary.

Because Terraform Enterprise runs the open source version of Packer, there should be no difference in execution between the two, other than the environment that Packer is running in. For more on hardware constraints in the Terraform Enterprise environment read below.

Once your builds are running smoothly locally you can push it up to Terraform Enterprise for versioning and automated builds.

Internal Errors

This is a short list of internal errors and what they mean.

- SIC-001: Your data was being ingressed from GitHub but failed to properly unpack. This can be caused by bad permissions, using symlinks or very large repository sizes. Using symlinks inside of the packer directory, or the root of the repository, if the packer directory is unspecified, will result in this internal error.

Note: Most often this error occurs when applications or builds are linked to a GitHub repository and the directory and/or template paths are incorrect. Double check that the paths specified when you linked the GitHub repository match the actual paths to your template file.
- SEC-001: Your data was being unpacked from a tarball uploaded and encountered an error. This can be caused by bad permissions, using symlinks or very large tarball sizes.

Community Resources

Packer is an open source project with an active community. If you're having an issue specific to Packer, the best avenue for support is the mailing list or IRC. All bug reports should go to GitHub.

- Website: packer.io (<https://packer.io>)
- GitHub: github.com/mitchellh/packer (<https://github.com/mitchellh/packer>)
- IRC: #packer-tool on Freenode
- Mailing list: Google Groups (<http://groups.google.com/group/packer-tool>)

Getting Support

If you believe your build is failing as a result of a bug in Terraform Enterprise, or would like other support, please email us (<mailto:support@hashicorp.com>).

About Terraform Enterprise Runs

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

A "run" represents the logical grouping of two Terraform steps - a "plan" and an "apply". The distinction between these two phases of a Terraform run are documented below.

When a new run is created (/docs/enterprise-legacy/runs/start.html), Terraform Enterprise automatically queues a Terraform plan. Because a plan does not change the state of infrastructure, it is safe to execute a plan multiple times without consequence. An apply executes the output of a plan and actively changes infrastructure. To prevent race conditions, the platform will only execute one plan/apply at a time (plans for validating GitHub Pull Requests are allowed to happen concurrently, as they do not modify state). You can read more about Terraform plans and applies below.

Plan

During the plan phase of a run, the command `terraform plan` is executed. Terraform performs a refresh and then determines what actions are necessary to reach the desired state specified in the Terraform configuration files. A successful plan outputs an executable file that is securely stored in Terraform Enterprise and may be used in the subsequent apply.

Terraform plans do not change the state of infrastructure, so it is safe to execute a plan multiple times. In fact, there are a number of components that can trigger a Terraform plan. You can read more about this in the starting runs (/docs/enterprise-legacy/runs/start.html) section.

Apply

During the apply phase of a run, the command `terraform apply` is executed with the executable result of the prior Terraform plan. This phase **can change infrastructure** by applying the changes required to reach the desired state specified in the Terraform configuration file.

While Terraform plans are safe to run multiple times, Terraform applies often change active infrastructure. Because of this, the default behavior is to require user confirmation as part of the Terraform run execution (/docs/enterprise-legacy/runs/how-runs-execute.html). Upon user confirmation, the Terraform apply will be queued and executed. It is also possible to configure automatic applies (/docs/enterprise-legacy/runs/automatic-applies.html), but this option is disabled by default.

Environment Locking

During run execution, the environment will lock to prevent other plans and applies from executing simultaneously. When the run completes, the next pending run, if any, will be started.

An administrator of the environment can also manually lock the environment, for example during a maintenance period.

You can see the lock status of an environment, and lock/unlock the environment by visiting that environment's settings page.

Notifications

To receive alerts when user confirmation is needed or for any other phase of the run process, you can enable run notifications (</docs/enterprise-legacy/runs/notifications.html>) for your organization or environment.

Automatic Terraform Applies

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

You can automatically apply successful Terraform plans to your infrastructure. This option is disabled by default and can be enabled by an organization owner on a per-environment basis.

This is an advanced feature that enables changes to active infrastructure without user confirmation. Please understand the implications to your infrastructure before enabling.

Enabling Auto-Apply

To enable auto-apply for an environment, visit the environment settings page check the box labeled "auto apply" and click the save button to persist the changes. The next successful Terraform plan for the environment will automatically apply without user confirmation.

How Terraform Runs Execute

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

This briefly covers the internal process of running Terraform plan and applies. It is not necessary to know this information, but may be valuable to help understand implications of running or debugging failed runs.

Steps of Execution

1. A set of Terraform configuration and directory of files is uploaded via Terraform Push or GitHub
2. Terraform Enterprise creates a version of the Terraform configuration and waits for the upload to complete. At this point, the version will be visible in the UI even if the upload has not completed
3. Once the upload finishes, Terraform Enterprise creates a run and queues a `terraform plan`
4. In the run environment, the package including the files and Terraform configuration are downloaded
5. `terraform plan` is run against the configuration in the run environment
6. Logs are streamed into the UI and stored
7. The `.tfplan` file created in the plan is uploaded and stored
8. Once the plan completes, the environment is torn down and status is updated in the UI
9. The plan then requires confirmation by an operator. It can optionally be discarded and ignored at this stage
10. Once confirmed, the run then executes a `terraform apply` in a new environment against the saved `.tfplan` file
11. The logs are streamed into the UI and stored
12. Once the apply completes, the environment is torn down, status is updated in the UI and changed state is saved back

Note: In the case of a failed apply, it's safe to re-run. This is possible because Terraform saves partial state and can "pick up where it left off".

Customizing Terraform Execution

As described in the steps above, Terraform will be run against your configuration when changes are pushed via GitHub, `terraform push`, or manually queued in the UI. There are a few options available to customize the execution of Terraform. These are:

- The directory that contains your environment's Terraform configuration can be customized to support directory structures with more than one set of Terraform configuration files. To customize the directory for your Environment, set the *Terraform Directory* property in the *GitHub Integration* (</docs/enterprise-legacy/vcs/github.html>) settings for your environment. This is equivalent to passing the `[dir]` argument when running Terraform in your local shell.

- The directory in which Terraform is executed from can be customized to support directory structures with nested sub-directories or configurations that use Terraform modules with relative paths. To customize the directory used for Terraform execution in your Environment, set the TF_ATLAS_DIR environment variable ([/docs/enterprise-legacy/runs/variables-and-configuration.html#environment-variables](#)) to the relative path of the directory - ie. `terraform/production`. This is equivalent to changing directories to the appropriate path in your local shell and then executing Terraform.

Installing Custom Software

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

The machines that run Terraform exist in an isolated environment and are destroyed on each use. In some cases, it may be necessary to install certain software on the Terraform worker, such as a configuration management tool like Chef, Puppet, Ansible, or Salt.

The easiest way to install software on the Terraform worker is via the `local-exec` provisioner. This will execute commands on the host machine running Terraform.

```
resource "null_resource" "local-software" {
  provisioner "local-exec" {
    command = <<EOH
sudo apt-get update
sudo apt-get install -y ansible
EOH
  }
}
```

Please note that nothing is persisted between Terraform runs, so you will need to install custom software on each run.

The Terraform workers run the latest version of Ubuntu LTS.

Managing Terraform Versions

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise does not automatically upgrade the version of Terraform used to execute plans and applies. This is intentional, as occasionally there can be backwards incompatible changes made to Terraform that cause state and plans to differ based on the same configuration, or new versions that produce some other unexpected behavior.

All upgrades must be performed by a user, but Terraform Enterprise will display a notice above any plans or applies run with out of date versions. We encourage the use of the latest version when possible.

Note that regardless of when an upgrade is performed, the version of Terraform used in a plan will be used in the subsequent apply.

Upgrading Terraform

1. Go the Settings tab of an environment
2. Go to the "Terraform Version" section and select the version you wish to use
3. Review the changelog for that version and previous versions
4. Click the save button. At this point, future builds will use that version

AWS Multi-Factor Authentication for Terraform Runs in Terraform Enterprise

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

You can optionally configure Terraform plans and applies to use multi-factor authentication using AWS Secure Token Service (<http://docs.aws.amazon.com/STS/latest/APIReference>Welcome.html>).

This option is disabled by default and can be enabled by an organization owner.

This is an advanced feature that enables changes to active infrastructure without user confirmation. Please understand the implications to your infrastructure before enabling.

Setting Up AWS Multi-Factor Authentication

Before you are able to set up multi-factor authentication in Terraform Enterprise, you must set up an IAM user in AWS. More details about creating an IAM user can be found here (http://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_mfa_enable.html). Setting up an AWS IAM user will provide you with the serial number and access keys that you will need in order to connect to AWS Secure Token Service.

In order to set up multi-factor authentication for your organization, you must have the following environment variables in your configuration: 'AWS_ACCESS_KEY_ID', 'AWS_SECRET_ACCESS_KEY', 'AWS_MFA_SERIAL_NUMBER'. You can set these variables at /settings/organization_variables.

Enabling AWS Multi-Factor Authentication

To enable multi-factor authentication, visit the environment settings page:

```
/terraform/:organization/environments/:environment/settings
```

Use the drop down labeled "AWS Multi-Factor Authentication ". There are currently three levels available: "never", "applies only", and "plans and applies". Once you have selected your desired level, save your settings. All subsequent runs on the environment will now require the selected level of authentication.

Using AWS Multi-Factor Authentication

Once you have elected to use AWS MFA for your Terraform Runs, you will then be prompted to enter a token code each time you plan or apply the run depending on your settings. Your one time use token code will be sent to you via the method you selected when setting up your IAM account (http://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_mfa_enable.html).

If you have selected "applies only", you will be able to queue and run a plan without entering your token code. Once the run finishes, you will need to enter your token code and click "Authenticate" before the applying the plan. Once you submit your token code, the apply will start, and you will see "Authenticated with MFA by user" in the UI. If for any case there is an error when submitting your token code, the lock icon in the UI will turn red, and an error will appear alerting you to the failure.

If you have selected "plans and applies", you will be prompted to enter your token before queueing your plan. Once you enter the token and click "Authenticate", you will see "Authenticated with MFA by user" appear in the UI logs. The plan will queue and you may run the plan once it is queued. Then, before applying, you will be asked to authenticate with MFA again. Enter your token, click Authenticate, and note that "Authenticated with MFA by user" appears in the UI log after the apply begins. If for any case there is an error authenticating, the lock icon in the UI will turn red, and an error will appear alerting you to the failure.

Using AWS Multi-Factor Authentication with AWS STS AssumeRole

The AWS Secure Token Service can be used to return a set of temporary security credentials that a user can use to access resources that they might not normally have access to (known as AssumeRole). The AssumeRole workflow is compatible with AWS multi-factor authentication in Terraform Enterprise.

To use AssumeRole, you first need to create an IAM role and edit the trust relationship policy document to contain the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::[INT]:user/[USER]"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "Bool": {
          "aws:MultiFactorAuthPresent": "true"
        }
      }
    }
  ]
}
```

You can then configure the Terraform AWS provider to assume a given role by specifying the role ARN within the nested `assume_role` block:

```
provider "aws" {
  # ...

  assume_role {
    role_arn = "arn:aws:iam::[INT]:role/[ROLE]"
  }
}
```

Terraform Run Notifications

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise can send run notifications, the following events are configurable:

- **Needs Confirmation** - The plan phase has succeeded, and there are changes that need to be confirmed before applying.
- **Confirmed** - A plan has been confirmed, and it will begin applying shortly.
- **Discarded** - A user has discarded the plan.
- **Applying** - The plan has begun to apply and make changes to your infrastructure.
- **Applied** - The plan was applied successfully.
- **Errored** - An error has occurred during the plan or apply phase.

Emails will include logs for the **Needs Confirmation**, **Applied**, and **Errored** events.

You can toggle notifications for each of these events on the "Integrations" tab of an environment.

Schedule Periodic Plan Runs

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform can automatically run a plan against your infrastructure on a specified schedule. This option is disabled by default and can be enabled by an organization owner on a per-environment basis.

On the specified interval, a plan can be run that for you, determining any changes and sending the appropriate notifications.

When used with automatic applies (/docs/enterprise-legacy/runs/automatic-applies.html), this feature can help converge changes to infrastructure without human input.

Runs will not be queued while another plan or apply is in progress, or if the environment has been manually locked. See Environment Locking (/docs/enterprise-legacy/runs#environment-locking) for more information.

Enabling Periodic Plans

To enable periodic plans for an environment, visit the environment settings page and select the desired interval and click the save button to persist the changes. An initial plan may immediately run, depending on the state of your environment, and then will automatically plan at the specified interval.

If you have manually run a plan separately, a new plan will not be queued until the allotted time after the manual plan ran. This means that the platform simply ensures that a plan has been executed at the specified schedule.

Starting Terraform Runs

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

There are a variety of ways to queue a Terraform run in Terraform Enterprise. In addition to `terraform push`, you can connect your environment to GitHub and runs based on new commits. You can also intelligently queue new runs when linked artifacts are uploaded or changed. Remember from the previous section about Terraform runs (/docs/enterprise-legacy/runs) that it is safe to trigger many plans without consequence since Terraform plans do not change infrastructure.

Terraform Push

Terraform push is a Terraform command (<https://terraform.io/docs/commands/push.html>) that packages and uploads a set of Terraform configuration and directory to the platform. This then creates a run which performs `terraform plan` and `terraform apply` against the uploaded configuration.

The directory is included in order to run any associated provisioners, that might use local files. For example, a remote-exec provisioner that executes a shell script.

By default, everything in your directory is uploaded as part of the push.

However, it's not always the case that the entire directory should be uploaded. Often, temporary or cache directories and files like `.git`, `.tmp` will be included by default, which can cause failures at certain sizes and should be avoided. You can specify exclusions (<https://terraform.io/docs/commands/push.html>) to avoid this situation.

Terraform also allows for a VCS option (https://terraform.io/docs/commands/push.html#_vcs_true) that will detect your VCS (if there is one) and only upload the files that are tracked by the VCS. This is useful for automatically excluding ignored files. In a VCS like git, this basically does a `git ls-files`.

GitHub Webhooks

Optionally, GitHub can be used to import Terraform configuration. When used within an organization, this can be extremely valuable for keeping differences in environments and last mile changes from occurring before an upload.

After you have connected your GitHub account to Terraform Enterprise (/docs/enterprise-legacy/vcs/github.html), you can connect your environment to the target GitHub repository. The GitHub repository will be linked to the Terraform Enterprise configuration, and GitHub will start sending webhooks. Certain GitHub webhook events, detailed below, will cause the repository to be automatically ingressed into Terraform and stored, along with references to the GitHub commits and authorship information.

Currently, an environment must already exist to be connected to GitHub. You can create the environment with `terraform push`, detailed above, and then link it to GitHub.

Each ingress will trigger a Terraform plan. If you have auto-apply enabled then the plan will also be applied.

You can disable an ingress by adding the text `[atlas skip]` or `[ci skip]` to your commit message.

Supported GitHub webhook events:

- pull_request (on by default)
 - ingress when opened or reopened
 - ingress when synchronized (new commits are pushed to the branch)
- push (on by default)
 - ingress when a tag is created
 - ingress when the default branch is updated
 - note: the default branch is either configured on your configuration's integrations tab, or if that is blank it is the GitHub repository's default branch
- create (off by default)
 - ingress when a tag is created
 - note: if you want to only run on tag creation, turn on create events and turn off push events

Artifact Uploads

Upon successful completion of a Terraform run, the remote state is parsed and any artifacts (/docs/enterprise-legacy/artifacts/artifact-provider.html) are detected that were referenced. When new versions of those referenced artifacts are uploaded, you have the option to automatically queue a new Terraform run.

For example, consider the following Terraform configuration which references an artifact named "worker":

```
resource "aws_instance" "worker" {  
  ami           = "${atlas_artifact.worker.metadata_full.region-us-east-1}"  
  instance_type = "m1.small"  
}
```

When a new version of the artifact "worker" is uploaded either manually or as the output of a Packer build (/docs/enterprise-legacy/packer/builds/start.html), a Terraform plan can be automatically triggered with this new artifact version. You can enable this feature on a per-environment basis from the environment settings page.

Combined with Terraform auto apply (/docs/enterprise-legacy/runs/automatic-applies.html), you can continuously deliver infrastructure using Terraform and Terraform Enterprise.

Terraform Plugins

If you are using a custom Terraform Plugin (<https://www.terraform.io/docs/plugins/index.html>) binary for a provider or provisioner that's not currently in a released version of Terraform, you can still use this in Terraform Enterprise.

All you need to do is include a Linux AMD64 binary for the plugin in the directory in which Terraform commands are run from; it will then be used next time you `terraform push` or `ingress` from GitHub.

Terraform Variables and Configuration

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

There are several ways to configure Terraform runs:

1. Terraform variables
2. Environment variables
3. Personal Environment and Personal Organization variables

You can add, edit, and delete all Terraform, Environment, and Personal Environment variables from the "Variables" page on your environment:

Terraform Variables

These variables are sent with `terraform push`.

[Edit](#)

There are no variables set.

Environment Variables

These variables will be set in your environment.

[Edit](#)

`ATLAS_TOKEN`

There are no custom environment variables set.

Personal Environment Variables for `terraform-run-canary`

These environment variables will be included in Terraform runs for the `terraform-run-canary` environment. They will override all other variables.

[Edit](#)

There are no user environment variables set for `terraform-run-canary`.

Personal Organization variables can be managed in your Account Settings under "Organization Variables":

These environment variables will be included in Terraform runs that you initiate for the `slothcorp` organization.

| | | |
|-------------------------------------|---------------------------------------|------------------------------------|
| <code>SECRET_GATE_ACCESS_KEY</code> | <input type="text" value="REDACTED"/> | <input type="checkbox"/> Sensitive |
|-------------------------------------|---------------------------------------|------------------------------------|

Variable types

Terraform Variables

Terraform variables are first-class configuration in Terraform. They define the parameterization of Terraform configurations and are important for sharing and removal of sensitive secrets from version control.

Variables are sent with the `terraform push` command. Any variables in your local `.tfvars` files are securely uploaded. Once variables are uploaded, Terraform will prefer the stored variables over any changes you make locally. Please refer to the Terraform push documentation (<https://www.terraform.io/docs/commands/push.html>) for more information.

You can also add, edit, and delete variables. To update Terraform variables, visit the "variables" page on your environment.

The maximum size for the value of Terraform variables is 256kb.

For detailed information about Terraform variables, please read the Terraform variables (<https://terraform.io/docs/configuration/variables.html>) section of the Terraform documentation.

Environment Variables

Environment variables are injected into the virtual environment that Terraform executes in during the `plan` and `apply` phases.

You can add, edit, and delete environment variables from the "variables" page on your environment.

Additionally, the following environment variables are automatically injected by Terraform Enterprise. All injected environment variables will be prefixed with `ATLAS_`

- `ATLAS_TOKEN` - This is a unique, per-run token that expires at the end of run execution (e.g. `"abcd.atlasv1.ghjkl..."`).
- `ATLAS_RUN_ID` - This is a unique identifier for this run (e.g. `"33"`).
- `ATLAS_CONFIGURATION_NAME` - This is the name of the configuration used in this run. Unless you have configured it differently, this will also be the name of the environment (e.g. `"production"`).
- `ATLAS_CONFIGURATION_SLUG` - This is the full slug of the configuration used in this run. Unless you have configured it differently, this will also be the name of the environment (e.g. `"company/production"`).
- `ATLAS_CONFIGURATION_VERSION` - This is the unique, auto-incrementing version for the Terraform configuration (e.g. `"34"`).
- `ATLAS_CONFIGURATION_VERSION_GITHUB_BRANCH` - This is the name of the branch that the associated Terraform

configuration version was ingressed from (e.g. master).

- ATLAS_CONFIGURATION_VERSION_GITHUB_COMMIT_SHA - This is the full commit hash of the commit that the associated Terraform configuration version was ingressed from (e.g. "abcd1234...").
- ATLAS_CONFIGURATION_VERSION_GITHUB_TAG - This is the name of the tag that the associated Terraform configuration version was ingressed from (e.g. "v0.1.0").

For any of the GITHUB_ attributes, the value of the environment variable will be the empty string ("") if the resource is not connected to GitHub or if the resource was created outside of GitHub (like using `terraform push`).

Personal Environment and Personal Organization Variables

Personal variables can be created at the Environment or Organization level and are private and scoped to the user that created them. Personal Environment variables are scoped to just the environment they are attached to, while Personal Organization variables are applied across any environment a user triggers a Terraform run in. Just like shared Environment variables, they are injected into the virtual environment during the plan and apply phases.

Both Personal Environment and Personal Organization variables can be used to override Environment variables on a per-user basis.

Variable Hierarchy

It is possible to create the same variable in multiple places for more granular control. Variables are applied in the following order from least to most precedence:

1. Environment
2. Personal Organization
3. Personal Environment

Here's an example:

- For the SlothCorp/petting_zoo environment, User 1 creates an Environment variable called SECRET_GATE_ACCESS_KEY and sets the value to "orange-turtleneck"
- User 2 adds a Personal Environment variable for SECRET_GATE_ACCESS_KEY and sets the value to "pink-overalls"
- When User 2 submits a plan or apply, the SECRET_GATE_ACCESS_KEY will use "pink-overalls"
- When User 1, or any other user, submits a plan or apply, the SECRET_GATE_ACCESS_KEY will use "orange-turtleneck"

Managing Secret Multi-Line Files

Terraform Enterprise has the ability to store multi-line files as variables. The recommended way to manage your secret or sensitive multi-line files (private key, SSL cert, SSL private key, CA, etc.) is to add them as Terraform Variables or Environment Variables.

Just like secret strings, it is recommended that you never check in these multi-line secret files to version control by following the below steps.

Set the variables (<https://www.terraform.io/docs/configuration/variables.html>) in your Terraform template that resources utilizing the secret file will reference:

```
variable "private_key" {}

resource "aws_instance" "example" {
  # ...

  provisioner "remote-exec" {
    connection {
      host        = "${self.private_ip}"
      private_key = "${var.private_key}"
    }
  }

  # ...
}
```

`terraform push` any "Terraform Variables":

```
$ terraform push -name $ATLAS_USERNAME/example -var "private_key=$MY_PRIVATE_KEY"
```

`terraform push` any "Environment Variables":

```
$ TF_VAR_private_key=$MY_PRIVATE_KEY terraform push -name $ATLAS_USERNAME/example
```

Alternatively, you can add or update variables manually by going to the "Variables" section of your Environment and pasting the contents of the file in as the value.

Now, any resource that consumes that variable will have access to the variable value, without having to check the file into version control. If you want to run Terraform locally, that file will still need to be passed in as a variable in the CLI. View the Terraform Variable Documentation (<https://www.terraform.io/docs/configuration/variables.html>) for more info on how to accomplish this.

A few things to note...

The `.tfvars` file does not support multi-line files. You can still use `.tfvars` to define variables, however, you will not be able to actually set the variable in `.tfvars` with the multi-line file contents like you would a variable in a `.tf` file.

If you are running Terraform locally, you can pass in the variables at the command line:

```
$ terraform apply -var "private_key=$MY_PRIVATE_KEY"
$ TF_VAR_private_key=$MY_PRIVATE_KEY terraform apply
```

You can update variables locally by using the `-overwrite` flag with your `terraform push` command:

```
$ terraform push -name $ATLAS_USERNAME/example -var "private_key=$MY_PRIVATE_KEY" -overwrite=private_key
$ TF_VAR_private_key=$MY_PRIVATE_KEY terraform push -name $ATLAS_USERNAME/example -overwrite=private_key
```

Notes on Security

Terraform variables and environment variables are encrypted using Vault (<https://vaultproject.io>) and closely guarded and audited. If you have questions or concerns about the safety of your configuration, please contact our security team at security@hashicorp.com (<mailto:security@hashicorp.com>).

State

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise stores the state of your managed infrastructure from the last time Terraform was run. The state is stored remotely, which works better in a team environment, allowing you to store, version and collaborate on state.

Remote state gives you more than just easier version control and safer storage. It also allows you to delegate the outputs to other teams. This allows your infrastructure to be more easily broken down into components that multiple teams can access.

Remote state is automatically updated when you run `apply` (/docs/commands/apply.html) locally. It is also updated when an `apply` is executed in a Terraform Enterprise Run (/docs/enterprise-legacy/runs/index.html).

Read more about remote state (/docs/state/remote.html).

Collaborating on Terraform Remote State

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise is one of a few options to store remote state (</docs/state/remote.html>).

Remote state gives you the ability to version and collaborate on Terraform changes. It stores information about the changes Terraform makes based on configuration.

In order to collaborate safely on remote state, we recommend creating an organization (</docs/enterprise-legacy/organizations/create.html>) to manage teams of users.

Then, following a Terraform Enterprise Run (</docs/enterprise-legacy/runs>) or apply (</docs/commands/apply.html>) you can view state versions in the States list of the environment.

Pushing Terraform Remote State to Terraform Enterprise

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise is one of a few options to store remote state (</docs/enterprise-legacy/state>).

Remote state gives you the ability to version and collaborate on Terraform changes. It stores information about the changes Terraform makes based on configuration.

To use Terraform Enterprise to store remote state, you'll first need to have the ATLAS_TOKEN environment variable set and run the following command.

NOTE: `terraform remote config` command has been deprecated in 0.9.X. Remote configuration is now managed as a backend configuration (</docs/backends/config.html>).

```
$ terraform remote config \
  -backend-config="name=$USERNAME/product"
```

Resolving Conflicts in Remote States

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Resolving state conflicts can be time consuming and error prone, so it's important to approach it carefully.

There are several tools provided by Terraform Enterprise to help resolve conflicts and fix remote state issues. First, you can navigate between state versions in the changes view of your environment (after toggling on the remote state checkbox) and view plain-text differences between versions.

This allows you to pinpoint where things may have gone wrong and make an educated decision about resolving the conflict.

Rolling Back to a Specific State Version

The rollback feature allows you to choose a new version to set as the "Head" version of the state. Rolling back to a version means it will then return that state upon request from a client. It will not increment the serial in the state, but perform a hard rollback to the exact version of the state provided.

This allows you to reset the state to an older version, essentially forgetting changes made in versions after that point.

To roll back to a specific version, navigate to it in the changes view and use the rollback link. You'll need to confirm the version number to perform the operation.

Using Terraform Locally

Another way to resolve remote state conflicts is by manual intervention of the state file.

Use the `state pull` (/docs/commands/state/pull.html) subcommand to pull the remote state into a local state file.

```
$ terraform state pull > example.tfstate
```

Once a conflict has been resolved locally by editing the state file, the serial can be incremented past the current version and pushed with the `state push` (/docs/commands/state/push.html) subcommand:

```
$ terraform state push example.tfstate
```

This will upload the manually resolved state and set it as the head version.

Contacting Support

All users of Terraform Enterprise are urged to email feedback, questions or requests to the HashiCorp team.

Free Support

We do not currently publish support SLAs for free accounts, but endeavor to respond as quickly as possible. We respond to most requests within less than 24 hours.

HashiCorp Tools Support

It's often the case that Terraform Enterprise questions or feedback relates to the HashiCorp tooling. We encourage all Terraform Enterprise users to search for related issues and problems in the open source repositories and mailing lists prior to contacting us to help make our support more efficient and to help resolve problems faster.

Visit the updating tools section for a list of our tools and their project websites.

Documentation Feedback

Due to the dynamic nature of Terraform Enterprise and the broad set of features it provides, there may be information lacking in the documentation.

In this case, we appreciate any feedback to be emailed to us so we can make improvements. Please email feedback to (<mailto:support@hashicorp.com>)support@hashicorp.com (<mailto:support@hashicorp.com>).

User Accounts

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Users are the main identity system in Terraform Enterprise. A user can be a member of multiple organizations (</docs/enterprise-legacy/organizations/index.html>), as well as individually collaborate on various resources.

Access

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise allows collaboration on resources, with several access levels available for users.

Read

Read is the lowest level of access within Terraform Enterprise. Users with read access on a resource can view it, but are unable to modify or delete it.

Write

Write access allows collaborating users the ability to view the resource and perform create, update, and edit actions on a resource. Users with write access are not able to destroy resources. For example, users with write access are able to view and edit Terraform environment variables as well as push new configuration.

Admin

Admin is the top-most user access level within Terraform Enterprise and is reserved for organization owners and select user accounts. Beyond read and write access, admins are given ownership permissions on the resource that allow them to manage collaborations and destroy the resource.

Authentication

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise requires a username and password to sign up and login. However, there are several ways to authenticate with your account.

Authentication Tokens

Authentication tokens are keys used to access your account via tools or over the various APIs used in Terraform Enterprise.

You can create new tokens in the token section of your account settings. It's important to keep tokens secure, as they are essentially a password and can be used to access your account or resources. Additionally, token authentication bypasses two factor authentication.

Authenticating Tools

All HashiCorp tools look for the ATLAS_TOKEN environment variable:

```
$ export ATLAS_TOKEN=TOKEN
```

This will automatically authenticate all requests against this token. This is the recommended way to authenticate with our various tools. Care should be given to how this token is stored, as it is as good as a password.

Two Factor Authentication

You can optionally enable Two Factor authentication, requiring an SMS or TOTP one-time code every time you log in, after entering your username and password.

You can enable Two Factor authentication in the security section of your account settings.

Be sure to save the generated recovery codes. Each backup code can be used once to sign in if you do not have access to your two-factor authentication device.

Sudo Mode

When accessing certain admin-level pages (adjusting your user profile, for example), you may notice that you're prompted for your password, even though you're already logged in. This is by design, and aims to help guard protect you if your screen is unlocked and unattended.

Session Management

You can see a list of your active sessions on your security settings page. From here, you can revoke sessions, in case you have lost access to a machine from which you were accessing.

Account Recovery

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

If you have lost access to your Terraform Enterprise account, use the reset password form on the login page to send yourself a link to reset your password.

If an email is unknown, contact us (<mailto:support@hashicorp.com>) for further help.

Integration with Version Control Software

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Terraform Enterprise can integrate with your version control software to automatically execute Terraform with your latest Terraform configuration as you commit changes to source control.

Different capabilities within Terraform Enterprise are available depending on the integration in use. The available integration options are on the sidebar navigation.

Bitbucket Cloud

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Bitbucket Cloud can be used to import Terraform configuration, automatically queuing runs when changes are merged into a repository's default branch. Additionally, plans are run when a pull request is created or updated. Terraform Enterprise will update the pull request with the result of the Terraform plan providing quick feedback on proposed changes.

Setup an Organization to use Bitbucket Cloud

Create a Bitbucket Cloud OAuth Consumer

You need to register Terraform Enterprise as an OAuth consumer within your Bitbucket Cloud account. Proceed to <https://bitbucket.org/account/user/your-username/oauth-consumers/new> (<https://bitbucket.org/account/user/your-username/oauth-consumers/new>). Fill out the form with the following information.

- **Name:** Terraform Enterprise (or whatever you want)
- **Callback URL:** Skip this one for now. You will need to come back and fill it in later.

The following **Permissions** are required:

- **Account:** Email, Read, Write
- **Repositories:** Read, Write, Admin
- **Pull requests:** Read, Write
- **Webhooks:** Read and write

Upon saving, you will be redirected to <https://bitbucket.org/account/user/your-username/api> (<https://bitbucket.org/account/user/your-username/api>).

1. Scroll down to the **OAuth consumers** section and click on the consumer you just created
2. Copy the **Key** and **Secret**
3. Leave this tab open in your browser as you will need to return to it in a moment.

Create a Terraform Enterprise OAuth Client

In a new tab, navigate to <https://atlas.hashicorp.com/settings> (<https://atlas.hashicorp.com/settings>) and, in the left-side panel, select the Organization that you'd like to setup to use with Bitbucket Cloud. Then click on **Configuration** in the left-side panel.

Scroll down to the **Add OAuthClient** pane and fill out the form with the following information.

- **Oauth client:** Bitbucket Cloud
- **Application key:** Key (from Bitbucket Cloud in the previous section)
- **Base url:** <https://bitbucket.org> (<https://bitbucket.org>)
- **Api url:** <https://api.bitbucket.org/2.0> (<https://api.bitbucket.org/2.0>)
- **Application secret:** Secret (from Bitbucket Cloud in the previous section)

Once you have created your client, you will be redirected back to the **Configuration** page for your chosen Organization. On that page, find the **OAuth Clients** pane and copy the **Callback url** for Bitbucket Cloud. Leave this tab open in your browser as you will need to return to it in a moment.

Back in the open Bitbucket tab, select the Terraform Enterprise OAuth consumer and click edit. Enter the **Callback url** you just copied in the field labeled **Callback URL**. Save the OAuth consumer.

Connect a Bitbucket Cloud User to Organization

Back on the **Configuration** page for your Terraform Enterprise Organization, in the **OAuth Connections** pane, you can now connect your organization to Bitbucket Cloud by clicking **Connect**. You will be briefly redirected to Bitbucket Cloud in order to authenticate the client. You should be successfully redirected back to Terraform Enterprise. If you are not, check the values in your OAuth client and make sure they match exactly with the values associated with your Bitbucket OAuth consumer.

The Terraform Enterprise Bitbucket Cloud integration is now ready your Organization to start using.

Connecting Configurations

Once you have linked a Bitbucket installation to your Organization, you are ready to begin creating Packer Builds and Terraform Environments linked to your desired Bitbucket Cloud repository.

Terraform Enterprise environments are linked to individual Bitbucket Cloud repositories. However, a single Bitbucket Cloud repository can be linked to multiple environments allowing a single set of Terraform configuration to be used across multiple environments.

Environments can be linked when they're initially created using the New Environment process. Existing environments can be linked by setting Bitbucket Cloud details in their **Integrations**.

To link a Terraform Enterprise environment to a Bitbucket Cloud repository, you need three pieces of information:

- **Bitbucket Cloud repository** - The location of the repository being imported in the format *username/repository*.
- **Bitbucket Cloud branch** - The branch from which to ingress new versions. This defaults to the value Bitbucket Cloud provides as the default branch for this repository.
- **Path to directory of Terraform files** - The repository's subdirectory that contains its terraform files. This defaults to the root of the repository.

Note: Users creating, updating, or deleting webhooks via the API must have `owner` or `admin` permissions enabled on the target Bitbucket Cloud repository. To update user permissions on the target repository the repository owner can visit: <https://bitbucket.org/your-username/your-repository/admin/access> (<https://bitbucket.org/your-username/your-repository/admin/access>)

repository/admin/access)

Connecting a Bitbucket Cloud Repository to a Terraform Environment

Navigate to <https://atlas.hashicorp.com/configurations/import> (<https://atlas.hashicorp.com/configurations/import>) and select Link to Bitbucket Cloud. A menu will appear asking you to name the environment. Then use the autocomplete field for repository and select the repository for which you'd like to create a webhook & environment. If you do not see the repository you would like to connect to in the drop down, manually enter it using the format: username/repository. If necessary, fill out information about the VCS branch to pull from as well as the directory where the Terraform files live within the repository. Click Create and Continue.

Upon success, you will be redirected to the environment's runs page (<https://atlas.hashicorp.com/terraform/your-organization/environments/your-environment/changes/runs> (<https://atlas.hashicorp.com/terraform/your-organization/environments/your-environment/changes/runs>)). A message will display letting you know that the repository is ingressing from Bitbucket and once finished you will be able to Queue, Run, & Apply a Terraform Plan. Depending on your webhook settings, changes will be triggered through git events on the specified branch.

The events currently supported are repository and branch push, pull request, and merge.

Connecting a Bitbucket Cloud Repository to a Packer Build Configuration

Navigate to <https://atlas.hashicorp.com/builds/new> (<https://atlas.hashicorp.com/builds/new>) and select the organization for which you'd like to create a build configuration. Name your build & select Connect build configuration to a Git Repository. A form will appear asking you to select your Git Host. Select Bitbucket Cloud.

Choose the repository for which you'd like to create a webhook. Fill out any other information in the form such as preferred branch to build from (your default branch will be selected should this field be left blank), Packer directory, and Packer Template.

Upon clicking Create you will be redirected to the build configuration (<https://atlas.hashicorp.com/packer/your-organization/build-configurations/your-build-configuration> (<https://atlas.hashicorp.com/packer/your-organization/build-configurations/your-build-configuration>)). On this page, you will have the opportunity to make any changes to your packer template, push changes via the CLI, or manually queue a Packer build.

Depending on your webhook settings, changes will be triggered through git events on the specified branch. The events currently supported are repository and branch push, pull request, and merge.

Git Integration

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

Git repositories can be integrated with Terraform Enterprise by using `terraform push` (/docs/commands/push.html) to import Terraform configuration when changes are committed. When Terraform configuration is imported using `terraform push` a plan is automatically queued.

This integration is for Git repositories **not** hosted on GitHub. For GitHub, please see the GitHub documentation instead.

Setup

Terraform configuration can be manually imported by running `terraform push` like below:

```
$ terraform push -name=$USERNAME/ENV_NAME
```

A better option than having to manually run `terraform push` is to run it using a git commit hook. A client-side pre-push hook is suitable and will push your Terraform configuration when you push local changes to your Git server.

Client-side Commit Hook

The script below will execute `terraform push` when you push local changes to your Git server. Place the script at `.git/pre-push` in your local Git repository, set the necessary variables, and ensure the script is executable.

```
#!/bin/bash
#
# An example hook script to push Terraform configuration to Terraform Enterprise.
#
# Set the following variables for your project:
# - ENV_NAME - your environment name (e.g. org/env)
# - TERRAFORM_DIR - the local directory to push
# - DEFAULT_BRANCH - the branch to push. Other branches will be ignored.

ENV_NAME="YOUR_ORG/YOUR_ENV"
TERRAFORM_DIR="terraform"
DEFAULT_BRANCH=""

if [[ -z "$ENV_NAME" || -z "$TERRAFORM_DIR" || -z "$DEFAULT_BRANCH" ]]; then
    echo 'pre-push hook: One or more variables are undefined. Canceling push.'
    exit 1
fi

current_branch=$(git symbolic-ref HEAD | sed -e 's,.*\/\(.*\'),\1,')

if [ "$current_branch" == "$DEFAULT_BRANCH" ]; then
    echo "pre-push hook: Pushing branch [$current_branch] to environment [$ENV_NAME]."
    terraform push -name="$ENV_NAME" $TERRAFORM_DIR
else
    echo "pre-push hook: NOT pushing branch [$current_branch] to environment [$ENV_NAME]."
fi
```

GitHub Integration

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

GitHub can be used to import Terraform configuration, automatically queuing runs when changes are merged into a repository's default branch. Additionally, plans are run when a pull request is created or updated. Terraform Enterprise will update the pull request with the result of the Terraform plan providing quick feedback on proposed changes.

Setup

Terraform Enterprise environments are linked to individual GitHub repositories. However, a single GitHub repository can be linked to multiple environments allowing a single set of Terraform configuration to be used across multiple environments.

Environments can be linked when they're initially created using the New Environment process. Existing environments can be linked by setting GitHub details in their **Integrations**.

To link a Terraform Enterprise environment to a GitHub repository, you need three pieces of information:

- **GitHub repository** - The location of the repository being imported in the format *username/repository*.
- **GitHub branch** - The branch from which to ingress new versions. This defaults to the value GitHub provides as the default branch for this repository.
- **Path to directory of Terraform files** - The repository's subdirectory that contains its terraform files. This defaults to the root of the repository.

GitLab.com, GitLab Community, & GitLab Enterprise

Deprecation warning: Terraform Enterprise (Legacy) features of Atlas will no longer be actively developed or maintained and will be fully decommissioned on Thursday, May 31, 2018. Please see our Upgrading From Terraform Enterprise (Legacy) (<https://www.terraform.io/docs/enterprise/upgrade/index.html>) guide to migrate to the new Terraform Enterprise.

GitLab can be used to import Terraform configuration, automatically queuing runs when changes are merged into a repository's default branch. Additionally, plans are run when a pull request is created or updated. Terraform Enterprise will update the pull request with the result of the Terraform plan providing quick feedback on proposed changes.

Setup an Organization to use GitLab

Create a GitLab OAuth Application

You will need to register Terraform Enterprise as an OAuth application within your GitLab account. Proceed to <https://gitlab.com/profile/applications> (<https://gitlab.com/profile/applications>) (for GitLab.com). Fill out the form with the following information.

- **Name:** Terraform Enterprise (or whatever you want)
- **Redirect URI:** <http://example.com> (<http://example.com>) (You will need to come back and fill in the real value later)
- **Scopes:** None

Upon saving, you will be redirected to the OAuth application view. Copy the **Application Id** and **Secret**. You will need to enter these values into Terraform Enterprise. Leave this tab open in your browser as you will need to return to it in a moment.

Create a Terraform Enterprise OAuth Client

In a new tab, navigate to <https://atlas.hashicorp.com/settings> (<https://atlas.hashicorp.com/settings>) and, in the left-side panel, select the Organization that you'd like to setup to use with GitLab. Then click on **Configuration** in the left-side panel.

Scroll down to the **Add OAuthClient** pane and fill out the form with the following information.

- **Oauth client:** Your GitLab installation type (e.g. GitLab.com, GitLab Community Edition, or GitLab Enterprise)
- **Application key:** Application Id (from GitLab in the previous section)
- **Base url:** <https://gitlab.com> (<https://gitlab.com>) (for GitLab.com; If you are using GitLab Community Edition or GitLab Enterprise your URL will be different)
- **Api url:** <https://gitlab.com/api/v4> (<https://gitlab.com/api/v4>) (for GitLab.com; If you are using GitLab Community Edition or GitLab Enterprise your URL will be different)
- **Application secret:** Secret (from GitLab in the previous section)

Once you have created your client, you will be redirected back to the **Configuration** page for your chosen Organization. On

that page, find the **OAuth Clients** pane and copy the **Callback url** for GitLab. Leave this tab open in your browser as you will need to return to it in a moment.

Back in the open GitLab tab, select the Terraform Enterprise OAuth application and click edit. Enter the **Callback url** you just copied in the field labeled **Redirect URI**. Save the OAuth application.

Connect a GitLab User to Organization

Back on the **Configuration** page for your Terraform Enterprise Organization, in the **OAuth Connections** pane, you can now connect your organization to GitLab by clicking **Connect**. You will be briefly redirected to GitLab in order to authenticate the client. You should be successfully redirected back to Terraform Enterprise. If you are not, check the values in your OAuth client and make sure they match exactly with the values associated with your GitLab OAuth application.

The Terraform Enterprise GitLab integration is now ready your Organization to start using.

Connecting Configurations

Once you have linked a GitLab installation to your Organization, you are ready to begin creating Packer Builds and Terraform Environments linked to your desired GitLab repository.

Terraform Enterprise environments are linked to individual GitLab repositories. However, a single GitLab repository can be linked to multiple environments allowing a single set of Terraform configuration to be used across multiple environments.

Environments can be linked when they're initially created using the New Environment process. Existing environments can be linked by setting GitLab details in their **Integrations**.

To link a Terraform Enterprise environment to a GitLab repository, you need three pieces of information:

- **GitLab repository** - The location of the repository being imported in the format *username/repository*.
- **GitLab branch** - The branch from which to ingress new versions. This defaults to the value GitLab provides as the default branch for this repository.
- **Path to directory of Terraform files** - The repository's subdirectory that contains its terraform files. This defaults to the root of the repository.

Connecting a GitLab Repository to a Terraform Environment

Navigate to <https://atlas.hashicorp.com/configurations/import> (<https://atlas.hashicorp.com/configurations/import>) and select Link to GitLab.com (or your preferred GitLab installation). A Menu will appear asking you to name the environment. Then use the autocomplete field for repository and select the repository for which you'd like to create a webhook & environment. If necessary, fill out information about the VCS branch to pull from as well as the directory where the Terraform files live within the repository. Click Create and Continue.

Upon success, you will be redirected to the environment's runs page (<https://atlas.hashicorp.com/terraform/your-organization/environments/your-environment/changes/runs> (<https://atlas.hashicorp.com/terraform/your-organization/environments/your-environment/changes/runs>)). A message will display letting you know that the repository is

ingressing from GitLab and once finished you will be able to Queue, Run, & Apply a Terraform Plan. Depending on your webhook settings, changes will be triggered through git events on the specified branch. The events currently supported are repository and branch push, merge request, and merge.

Connecting a GitLab Repository to a Packer Build Configuration

Navigate to <https://atlas.hashicorp.com/builds/new> (<https://atlas.hashicorp.com/builds/new>) and select the organization for which you'd like to create a build configuration. Name your build & select Connect build configuration to a Git Repository. A form will appear asking you to select your Git Host. Select your preferred GitLab integration. Choose the repository for which you'd like to create a webhook. Fill out any other information in the form such as preferred branch to build from (your default branch will be selected should this field be left blank), Packer directory, and Packer Template.

Upon clicking Create you will be redirected to the build configuration (<https://atlas.hashicorp.com/packer/your-organization/build-configurations/your-build-configuration>). On this page, you will have the opportunity to make any changes to your packer template, push changes via the CLI, or manually queue a Packer build. Depending on your webhook settings, changes will be triggered through git events on the specified branch. The events currently supported are repository and branch push, merge request, and merge.

Terraform Enterprise API Documentation

Note: These API endpoints are in beta and are subject to change.

Terraform Enterprise (TFE) provides an API for a subset of its features. If you have any questions or want to request new API features, please email support@hashicorp.com (mailto:support@hashicorp.com).

See the navigation sidebar for the list of available endpoints.

Note: Before planning an API integration, consider whether the tfe Terraform provider (/docs/providers/tfe/index.html) meets your needs. It can't create or approve runs in response to arbitrary events, but it's a useful tool for managing your organizations, teams, and workspaces as code.

Authentication

All requests must be authenticated with a bearer token. Use the HTTP header `Authorization` with the value `Bearer <token>`. If the token is absent or invalid, TFE responds with HTTP status 401 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401>) and a JSON API error object (<http://jsonapi.org/format/#error-objects>). The 401 status code is reserved for problems with the authentication token; forbidden requests with a valid token result in a 404.

There are three kinds of token available:

- User tokens (/docs/enterprise/users-teams-organizations/users.html#api-tokens) — each TFE user can have any number of API tokens, which can make requests on their behalf.
- Team tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts) — each team has an associated service account, which can have one API token at a time. This is intended for performing plans and applies via a CI/CD pipeline.
- Organization tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts) — each organization also has a service account, which can have one API token at a time. This is intended for automating the management of teams, team membership, and workspaces. The organization token cannot perform plans and applies.

Response Codes

This API returns standard HTTP response codes.

We return 404 Not Found codes for resources that a user doesn't have access to, as well as for resources that don't exist. This is to avoid telling a potential attacker that a given resource exists.

Versioning

The API documented in these pages is the second version of TFE's API, and resides under the /v2 prefix. For documentation of the /v1 endpoints, see the Terraform Enterprise (legacy) API docs. (/docs/enterprise-legacy/api/index.html)

Future APIs will increment this version, leaving the /v1 API intact, though in the future we might deprecate certain features. In that case, we'll provide ample notice to migrate to the new API.

Paths

All V2 API endpoints use /api/v2 as a prefix unless otherwise specified.

For example, if the API endpoint documentation defines the path /runs then the full path is /api/v2/runs.

JSON API Formatting

The TFE endpoints use the JSON API specification (<http://jsonapi.org/>), which specifies key aspects of the API. Most notably:

- HTTP error codes (<http://jsonapi.org/examples/#error-objects-error-codes>)
- Error objects (<http://jsonapi.org/examples/#error-objects-basics>)
- Document structure (<http://jsonapi.org/format/#document-structure>)
- HTTP request/response headers (<http://jsonapi.org/format/#content-negotiation>)

JSON API Documents

Since our API endpoints use the JSON API spec, most of them return JSON API documents (<http://jsonapi.org/format/#document-structure>).

Endpoints that use the POST method also require a JSON API document as the request payload. A request object usually looks something like this:

```
{  
  "data": {  
    "type": "vars",  
    "attributes": {  
      "key": "some_key",  
      "value": "some_value",  
      "category": "terraform",  
      "hcl": false,  
      "sensitive": false  
    },  
    "relationships": {  
      "workspace": {  
        "data": {  
          "id": "ws-4j8p6jX1w33MiDC7",  
          "type": "workspaces"  
        }  
      }  
    }  
  }  
}
```

These objects always include a top-level data property, which:

- Must have a type property to indicate what type of API object you're interacting with.

- Often has an `attributes` property to specify attributes of the object you're creating or modifying.
- Sometimes has a `relationships` property to specify other objects that are linked to what you're working with.

In the documentation for each API method, we use dot notation to explain the structure of nested objects in the request. For example, the properties of the request object above are listed as follows:

| Key path | Type | Default | Description |
|---|--------|---------|---|
| <code>data.type</code> | string | | Must be "vars". |
| <code>data.attributes.key</code> | string | | The name of the variable. |
| <code>data.attributes.value</code> | string | | The value of the variable. |
| <code>data.attributes.category</code> | string | | Whether this is a Terraform or environment variable. Valid values are "terraform" or "env". |
| <code>data.attributes.hcl</code> | bool | false | Whether to evaluate the value of the variable as a string of HCL code. Has no effect for environment variables. |
| <code>data.attributes.sensitive</code> | bool | false | Whether the value is sensitive. If true then the variable is written once and not visible thereafter. |
| <code>data.relationships.workspace.data.type</code> | string | | Must be "workspaces". |
| <code>data.relationships.workspace.data.id</code> | string | | The ID of the workspace that owns the variable. |

We also always include a sample payload object, to show the document structure more visually.

Query Parameters

Although most of our API endpoints use the POST method and receive their parameters as a JSON object in the request payload, some of them use the GET method. These GET endpoints sometimes require URL query parameters, in the standard `...path?key1=value1&key2=value2` format.

Since these parameters were originally designed as part of a JSON object, they sometimes have characters that must be percent-encoded (<https://en.wikipedia.org/wiki/Percent-encoding>) in a query parameter. For example, [becomes %5B and] becomes %5D.

For more about URI structure and query strings, see the specification (RFC 3986) (<https://tools.ietf.org/html/rfc3986>) or the Wikipedia page on URIs (https://en.wikipedia.org/wiki/Uniform_Resource_Identifier).

Pagination

Most of the endpoints that return lists of objects support pagination. A client may pass the following query parameters to control pagination on supported endpoints:

| Parameter | Description |
|---------------------------|--|
| <code>page[number]</code> | Optional. If omitted, the endpoint will return the first page. |
| <code>page[size]</code> | Optional. If omitted, the endpoint will return 20 items per page. |

Additional data is returned in the `links` and `meta` top level attributes of the response.

```
{  
  "data": [...],  
  "links": {  
    "self": "https://app.terraform.io/api/v2/organizations/hashicorp/workspaces?page%5Bnumber%5D=1&page%5Bsize%5D=20",  
    "first": "https://app.terraform.io/api/v2/organizations/hashicorp/workspaces?page%5Bnumber%5D=1&page%5Bsize%5D=20",  
    "prev": null,  
    "next": "https://app.terraform.io/api/v2/organizations/hashicorp/workspaces?page%5Bnumber%5D=2&page%5Bsize%5D=20",  
    "last": "https://app.terraform.io/api/v2/organizations/hashicorp/workspaces?page%5Bnumber%5D=2&page%5Bsize%5D=20"  
  },  
  "meta": {  
    "pagination": {  
      "current-page": 1,  
      "prev-page": null,  
      "next-page": 2,  
      "total-pages": 2,  
      "total-count": 21  
    }  
  }  
}
```

Inclusion of Related Resources

Some of the API's GET endpoints can return additional information about nested resources by adding an `include` query parameter, whose value is a comma-separated list of resource types.

The related resource options are listed in each endpoint's documentation where available.

The related resources will appear in an `included` section of the response.

Example:

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request GET \  
  https://app.terraform.io/api/v2/teams/team-n8UQ6wfhyym25sMe?include=users
```

```
{
  "data": {
    "id": "team-n8UQ6wfhyym25sMe",
    "type": "teams",
    "attributes": {
      "name": "owners",
      "users-count": 1
      ...
    },
    "relationships": {
      "users": {
        "data": [
          {
            "id": "user-62goNpx1ThQf689e",
            "type": "users"
          }
        ]
      } ...
    }
    ...
  },
  "included": [
    {
      "id": "user-62goNpx1ThQf689e",
      "type": "users",
      "attributes": {
        "username": "hashibot"
        ...
      } ...
    }
  ]
}
```

Rate Limiting

You can make up to 30 requests per second to the API as an authenticated or unauthenticated request. If you reach the rate limit then your access will be throttled and an error response will be returned.

Authenticated requests are allocated to the user associated with the authentication token. This means that a user with multiple tokens will still be limited to 30 requests per second, additional tokens will not allow you to increase the requests per second permitted.

Unauthenticated requests are associated with the requesting IP address.

| Status | Response | Reason |
|---|---|------------------------------|
| 429 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429) | JSON API error object (http://jsonapi.org/format/#error-objects) | Rate limit has been reached. |

```
{  
  "errors": [  
    {  
      "detail": "You have exceeded the API's rate limit of 30 requests per second.",  
      "status": 429,  
      "title": "Too many requests"  
    }  
  ]  
}
```

Client libraries and tools

HashiCorp maintains go-tfe (<https://github.com/hashicorp/go-tfe>), a Go client for TFE's API.

Additionally, the community of Terraform Enterprise users and vendors have built client libraries in other languages. These client libraries and tools are not tested nor officially maintained by HashiCorp, but are listed below in order to help users find them easily.

If you have built a client library and would like to add it to this community list, please contribute (<https://github.com/hashicorp/terraform-website#contributions-welcome>) to this page (<https://github.com/hashicorp/terraform-website/blob/master/content/source/docs/enterprise/api/index.html.md>).

- tf_api_gateway (https://github.com/PlethoraOfHate/tf_api_gateway): Python API library and console app
- terraform-enterprise-client (<https://github.com/skierkowski/terraform-enterprise-client>): Ruby API library and console app

Account API

Note: These API endpoints are in beta and are subject to change.

Account represents the current user interacting with Terraform.

Get your account details

GET /account/details

| Status | Response | Reason |
|---|---|----------------------------|
| 200 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document (https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | The request was successful |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request GET \  
  https://app.terraform.io/api/v2/account/details
```

Sample Response

```
{
  "data": {
    "id": "user-V3R563qtJNcExAkN",
    "type": "users",
    "attributes": {
      "username": "admin",
      "is-service-account": false,
      "avatar-url": "https://www.gravatar.com/avatar/9babb00091b97b9ce9538c45807fd35f?s=100&d=mm",
      "v2-only": false,
      "is-site-admin": true,
      "is-sso-login": false,
      "email": "admin@hashicorp.com",
      "unconfirmed-email": null,
      "permissions": {
        "can-create-organizations": true,
        "can-change-email": true,
        "can-change-username": true
      }
    },
    "relationships": {
      "authentication-tokens": {
        "links": {
          "related": "/api/v2/users/user-V3R563qtJNcExAkN/authentication-tokens"
        }
      }
    },
    "links": {
      "self": "/api/v2/users/user-V3R563qtJNcExAkN"
    }
  }
}
```

Update your account info

Your username and email address can be updated with this endpoint.

`PATCH /account/update`

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | Your info was successfully updated |
| 401
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401) | JSON API error object (http://jsonapi.org/format/#error-objects) | Unauthorized |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|--------------------------|--------|---------|---|
| data.type | string | | Must be "users" |
| data.attributes.username | string | | New username |
| data.attributes.email | string | | New email address (must be confirmed afterwards to take effect) |

Sample Payload

```
{  
  "data": {  
    "type": "users",  
    "attributes": {  
      "email": "admin@example.com",  
      "username": "admin"  
    }  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request PATCH \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/account/update
```

Sample Response

```
{
  "data": {
    "id": "user-V3R563qtJNcExAkN",
    "type": "users",
    "attributes": {
      "username": "admin",
      "is-service-account": false,
      "avatar-url": "https://www.gravatar.com/avatar/9babb00091b97b9ce9538c45807fd35f?s=100&d=mm",
      "v2-only": false,
      "is-site-admin": true,
      "is-sso-login": false,
      "email": "admin@hashicorp.com",
      "unconfirmed-email": null,
      "permissions": {
        "can-create-organizations": true,
        "can-change-email": true,
        "can-change-username": true
      }
    },
    "relationships": {
      "authentication-tokens": {
        "links": {
          "related": "/api/v2/users/user-V3R563qtJNcExAkN/authentication-tokens"
        }
      }
    },
    "links": {
      "self": "/api/v2/users/user-V3R563qtJNcExAkN"
    }
  }
}
```

Change your password

PATCH /account/password

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | Your password was successfully changed |
| 401
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401) | JSON API error object (http://jsonapi.org/format/#error-objects) | Unauthorized |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|---------------------------------------|--------|---------|-----------------------------|
| data.type | string | | Must be "users" |
| data.attributes.current-password | string | | Current password |
| data.attributes.password | string | | New password |
| data.attributes.password-confirmation | string | | New password (confirmation) |

Sample Payload

```
{
  "data": {
    "type": "users",
    "attributes": {
      "current-password": "current password 2:C)e'G4{D\n06:[d1~y",
      "password": "new password 34rk492+jgLL0@xhfyisj",
      "password-confirmation": "new password 34rk492+jLL0@xhfyisj",
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/account/password
```

Sample Response

```
{  
  "data": {  
    "id": "user-V3R563qtJNcExAkN",  
    "type": "users",  
    "attributes": {  
      "username": "admin",  
      "is-service-account": false,  
      "avatar-url": "https://www.gravatar.com/avatar/9babb00091b97b9ce9538c45807fd35f?s=100&d=mm",  
      "v2-only": false,  
      "is-site-admin": true,  
      "is-sso-login": false,  
      "email": "admin@hashicorp.com",  
      "unconfirmed-email": null,  
      "permissions": {  
        "can-create-organizations": true,  
        "can-change-email": true,  
        "can-change-username": true  
      }  
    },  
    "relationships": {  
      "authentication-tokens": {  
        "links": {  
          "related": "/api/v2/users/user-V3R563qtJNcExAkN/authentication-tokens"  
        }  
      },  
      "links": {  
        "self": "/api/v2/users/user-V3R563qtJNcExAkN"  
      }  
    }  
  }  
}
```

Private Terraform Enterprise Admin API Documentation

Note: These API endpoints are in beta and are subject to change.

These API endpoints are available in Private Terraform Enterprise as of version 201807-1.

Private Terraform Enterprise provides an API to allow administrators to configure and support their installation.

See the navigation sidebar for the list of available endpoints.

Authentication

With the exception of the user impersonation endpoints ([/docs/enterprise/api/admin/users.html#impersonate-another-user](#)), all requests must be authenticated with a bearer token belonging to a site administrator. Use the HTTP Header `Authorization` with the value `Bearer <token>`. This token can be generated or revoked on the `tokens` tab of the user settings page ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)). In the context of the Admin API, your token has management access to all resources in the system.

For more information on authentication behavior, see the API overview section ([/docs/enterprise/api/index.html#authentication](#)).

Admin Organizations API

Note: These API endpoints are in beta and are subject to change.

These API endpoints are available in Private Terraform Enterprise as of version 201807-1.

The Organizations Admin API contains endpoints to help site administrators manage organizations.

List all organizations

GET /admin/organizations

This endpoint lists all organizations in the Terraform Enterprise installation.

| Status | Response | Reason |
|--|--|-----------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "organizations") | Successfully listed organizations |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is not an administrator. |

Query Parameters

These are standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|--------------|--|
| q | Optional. A search query string. Organizations are searchable by name and notification email. |
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 organizations per page. |

Available Related Resources

This GET endpoint can optionally return related resources, if requested with the `include` query parameter (/docs/enterprise/api/index.html#include-related-resources). The following resource types are available:

| Resource Name | Description |
|---------------|---|
| owners | A list of owners for each organization. |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  "https://app.terraform.io/api/v2/admin/organizations"
```

Sample Response

```
{
  "data": [
    {
      "id": "my-organization",
      "type": "organizations",
      "attributes": {
        "name": "my-organization",
        "enterprise-plan": "pro",
        "trial-expires-at": "2018-05-22T00:00:00.000Z",
        "notification-email": "my-organization@example.com"
      },
      "relationships": {
        "owners": {
          "data": [
            {
              "id": "user-mVPjPn2hRJFtHMF5",
              "type": "users"
            }
          ]
        }
      },
      "links": {
        "self": "/api/v2/organizations/my-organization"
      }
    }
  ],
  "links": {
    "self": "https://app.terraform.io/api/v2/admin/organizations?page%5Bnumber%5D=1&page%5Bsize%5D=20",
    "first": "https://app.terraform.io/api/v2/admin/organizations?page%5Bnumber%5D=1&page%5Bsize%5D=20",
    "prev": null,
    "next": null,
    "last": "https://app.terraform.io/api/v2/admin/organizations?page%5Bnumber%5D=1&page%5Bsize%5D=20"
  },
  "meta": {
    "pagination": {
      "current-page": 1,
      "prev-page": null,
      "next-page": null,
      "total-pages": 1,
      "total-count": 1
    },
    "status-counts": {
      "total": 1,
      "active-trial": 0,
      "expired-trial": 0,
      "pro": 1,
      "premium": 0,
      "disabled": 0
    }
  }
}
```

Show an organization

GET /admin/organizations/:name

| Parameter | Description | |
|--|--|---|
| :name | The name of the organization to show | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "organizations") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found, or client is not an administrator |

Available Related Resources

This GET endpoint can optionally return related resources, if requested with the `include` query parameter (</docs/enterprise/api/index.html#inclusion-of-related-resources>). The following resource types are available:

| Resource Name | Description |
|---------------|--|
| owners | A list of owners for the organization. |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  https://app.terraform.io/api/v2/admin/organizations/my-organization
```

Sample Response

```
{
  "data": {
    "id": "my-organization",
    "type": "organizations",
    "attributes": {
      "name": "my-organization",
      "notification-email": "my-organization@example.com"
    },
    "relationships": {
      "owners": {
        "data": [
          {
            "id": "user-mVPjPn2hRJFtHMF5",
            "type": "users"
          }
        ]
      }
    },
    "links": {
      "self": "/api/v2/organizations/my-organization"
    }
  }
}
```

Delete an organization

`DELETE /admin/organizations/:name`

| Parameter | Description | |
|---|--|---|
| <code>:name</code> | The name of the organization to delete | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Empty response | The organization was successfully deleted |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object
(http://jsonapi.org/format/#error-objects) | Organization not found, or client is not an administrator |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
https://app.terraform.io/api/v2/admin/organizations/my-organization
```

Admin Runs API

Note: These API endpoints are in beta and are subject to change.

These API endpoints are available in Private Terraform Enterprise as of version 201807-1.

The Runs Admin API contains endpoints to help site administrators manage runs.

List all runs

GET /admin/runs

This endpoint lists all runs in the Terraform Enterprise installation.

| Status | Response | Reason |
|--|---|---------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "runs") | Successfully listed runs |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is not an administrator. |

Query Parameters

These are standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|----------------|--|
| q | Optional. A search query string. Runs are searchable by ID, workspace name, organization name or email, and VCS repository identifier. |
| filter[status] | Optional. A comma-separated list of Run statuses to restrict results to, including any of the following: "pending", "planning", "planned", "confirmed", "applying", "applied", "discarded", "errored", "canceled", "policy_checking", "policy_override", and/or "policy_checked". |
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 runs per page. |

A VCS repository identifier is a reference to a VCS repository in the format :org/:repo, where :org and :repo refer to the organization (or project) and repository in your VCS provider.

Available Related Resources

This GET endpoint can optionally return related resources, if requested with the `include` query parameter (/docs/enterprise/api/index.html#inclusion-of-related-resources). The following resource types are available:

| Resource Name | Description |
|------------------------|---|
| workspace | The workspace this run belongs in. |
| workspace.organization | The organization of the associated workspace. |

Sample Request

```
curl \  
--header "Authorization: Bearer $TOKEN" \  
--header "Content-Type: application/vnd.api+json" \  
"https://app.terraform.io/api/v2/admin/runs"
```

Sample Response

```
{  
  "data": [  
    {  
      "id": "run-VCsNJXa59eUza53R",  
      "type": "runs",  
      "attributes": {  
        "status": "pending",  
        "status-timestamps": {  
          "planned-at": "2018-03-02T23:42:06+00:00",  
          "discarded-at": "2018-03-02T23:42:06+00:00"  
        },  
        "has-changes": true,  
        "created-at": "2018-03-02T23:42:06.651Z"  
      },  
      "relationships": {  
        "workspace": {  
          "data": {  
            "id": "ws-mJtb6bXGybq5zbf3",  
            "type": "workspaces"  
          }  
        }  
      },  
      "links": {  
        "self": "/api/v2/runs/run-VCsNJXa59eUza53R"  
      }  
    }  
  ],  
  "links": {  
    "self": "https://app.terraform.io/api/v2/admin/runs?page%5Bnumber%5D=1&page%5Bsize%5D=20",  
    "first": "https://app.terraform.io/api/v2/admin/runs?page%5Bnumber%5D=1&page%5Bsize%5D=20",  
    "prev": null,  
    "next": null,  
    "last": "https://app.terraform.io/api/v2/admin/runs?page%5Bnumber%5D=1&page%5Bsize%5D=20"  
  },  
  "meta": {  
    "pagination": {  
      "current-page": 1,  
      "prev-page": null,  
      "next-page": null,  
      "total-pages": 1,  
      "total-count": 1  
    },  
    "status-counts": {  
      "pending": 1,  
      "planning": 0,  
      "planned": 0,  
      "confirmed": 0,  
      "applying": 0,  
      "applied": 0,  
      "discarded": 0,  
      "errored": 0,  
      "canceled": 0,  
      "policy-checking": 0,  
      "policy-override": 0,  
      "policy-checked": 0,  
      "total": 1  
    }  
  }  
}
```

Force a run into the "cancelled" state

POST /admin/runs/:id/actions/force-cancel

| Parameter | Description | |
|-----------|------------------------------|--|
| :id | The ID of the run to cancel. | |

This endpoint forces a run (and its plan/apply, if applicable) into the "cancelled" state. This action should only be performed for runs that are stuck and no longer progressing normally, as there is a risk of lost state data if a progressing apply is force-canceled. Healthy runs can be requested for cancellation by end-users (/docs/enterprise/run/states.html).

| Status | Response | Reason |
|--|---|---|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "runs") | Successfully canceled the run. |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Run not found, or client is not an administrator. |

Request body

This POST endpoint allows an optional JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|----------|--------|---------|---|
| comment | string | null | An optional explanation for why the run was force-canceled. |

Sample Payload

```
{  
  "comment": "This run was stuck and would never finish."  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  "https://app.terraform.io/api/v2/admin/runs/run-VCsNJXa59eUza53R/actions/force-cancel"
```

Sample Response

```
{  
  "data": {  
    "id": "run-VCsNJXa59eUza53R",  
    "type": "runs",  
    "attributes": {  
      "status": "errored",  
      "status-timestamps": {  
        "planned-at": "2018-03-02T23:42:06Z"  
      },  
      "has-changes": true,  
      "created-at": "2018-03-02T23:42:06.651Z"  
    },  
    "relationships": {  
      "workspace": {  
        "data": {  
          "id": "ws-mJtb6bXGybq5zbf3",  
          "type": "workspaces"  
        }  
      }  
    },  
    "links": {  
      "self": "/api/v2/runs/run-VCsNJXa59eUza53R"  
    }  
  }  
}
```

Private Terraform Enterprise Settings API

Note: These API endpoints are in beta and are subject to change.

Note: These endpoints are only available in Private Terraform instances and only accessible by site administrators.

These API endpoints are available in Private Terraform Enterprise as of version 201807-1.

List General Settings

GET /api/v2/admin/general-settings

| Status | Response | Reason |
|--|---|--------------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "general-settings") | Successfully listed General settings |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request GET \  
  https://app.terraform.io/api/v2/admin/general-settings
```

Sample Response

```
{  
  "data": {  
    "id": "general",  
    "type": "general-settings",  
    "attributes": {  
      "limit-user-organization-creation": true,  
      "support-email-address": "support@hashicorp.com",  
      "api-rate-limiting-enabled": true,  
      "api-rate-limit": 30  
    }  
  }  
}
```

Update General Settings

PATCH /api/v2/admin/general-settings

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document (https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "general-settings") | Successfully updated the General settings |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|--|---------|-------------------------|--|
| data.attributes.limit-user-organization-creation | bool | true | When set to true, limits the ability to create organizations to users with the site-admin permission only. |
| data.attributes.support-email-address | string | "support@hashicorp.com" | The support address for outgoing emails. |
| data.attributes.api-rate-limiting-enabled | bool | true | Whether or not rate limiting is enabled for API requests. To learn more about API Rate Limiting, refer to the rate limiting documentation (/docs/enterprise/api/index.html#rate-limiting) |
| data.attributes.api-rate-limit | integer | 30 | The number of allowable API requests per second for any client. This value cannot be less than 30. To learn more about API Rate Limiting, refer to the rate limiting documentation (/docs/enterprise/api/index.html#rate-limiting) |

Sample Payload

```
{
  "data": {
    "attributes": {
      "limit-user-organization-creation": true,
      "support-email-address": "support@hashicorp.com",
      "api-rate-limiting-enabled": true,
      "api-rate-limit": 50
    }
  }
}
```

Sample Request

```
curl \  
--header "Authorization: Bearer $TOKEN" \  
--header "Content-Type: application/vnd.api+json" \  
--request PATCH \  
--data @payload.json \  
https://app.terraform.io/api/v2/admin/general-settings
```

Sample Response

```
{  
  "data": {  
    "id": "general",  
    "type": "general-settings",  
    "attributes": {  
      "limit-user-organization-creation": true,  
      "support-email-address": "support@hashicorp.com",  
      "api-rate-limiting-enabled": true,  
      "api-rate-limit": 50  
    }  
  }  
}
```

List SAML Settings

GET /api/v2/admin/saml-settings

| Status | Response | Reason |
|--|--|-------------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "saml-settings") | Successfully listed SAML settings |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |

Sample Request

```
curl \  
--header "Authorization: Bearer $TOKEN" \  
--header "Content-Type: application/vnd.api+json" \  
--request GET \  
https://app.terraform.io/api/v2/admin/saml-settings
```

Sample Response

```
{
  "data": {
    "id": "saml",
    "type": "saml-settings",
    "attributes": {
      "enabled": true,
      "debug": false,
      "idp-cert": "SAMPLE-CERTIFICATE",
      "slo-endpoint-url": "https://example.com/slo",
      "sso-endpoint-url": "https://example.com/sso",
      "attr-username": "Username",
      "attr-groups": "MemberOf",
      "attr-site-admin": "SiteAdmin",
      "site-admin-role": "site-admins",
      "sso-api-token-session-timeout": 1209600,
      "acs-consumer-url": "https://example.com/users/saml/auth",
      "metadata-url": "https://example.com/users/saml/metadata"
    }
  }
}
```

Update SAML Settings

PATCH /api/v2/admin/saml-settings

| Status | Response | Reason |
|--|---|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document (https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "saml-settings") | Successfully updated SAML settings |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

If `data.attributes.enabled` is set to `true`, all remaining attributes must have valid values. You can omit attributes if they have a default value, or if a value was set by a previous update. Omitted attributes keep their previous values.

See [SAML Configuration](#) (/docs/enterprise/saml/configuration.html) for more details on attribute values.

| Key path | Type | Default | Description |
|--------------------------------------|------|---------|---|
| <code>data.attributes.enabled</code> | bool | false | Allows SAML to be used. If true, all remaining attributes must have valid values. |
| <code>data.attributes.debug</code> | bool | false | Enables a SAML debug dialog that allows an admin to see the SAMLResponse XML and processed values during login. |

| Key path | Type | Default | Description |
|---|---------|---------------|---|
| data.attributes.idp-cert | string | | Identity Provider Certificate specifies the PEM encoded X.509 Certificate as provided by the IdP configuration. |
| data.attributes.slo-endpoint-url | string | | Single Log Out URL specifies the HTTPS endpoint on your IdP for single logout requests. This value is provided by the IdP configuration. |
| data.attributes.sso-endpoint-url | string | | Single Sign On URL specifies the HTTPS endpoint on your IdP for single sign-on requests. This value is provided by the IdP configuration. |
| data.attributes.attr-username | string | "Username" | Username Attribute Name specifies the name of the SAML attribute that determines the user's username. |
| data.attributes.attr-groups | string | "MemberOf" | Team Attribute Name specifies the name of the SAML attribute that determines team membership. |
| data.attributes.attr-site-admin | string | "SiteAdmin" | Specifies the role for site admin access. Overrides the "Site Admin Role" method. |
| data.attributes.site-admin-role | string | "site-admins" | Specifies the role for site admin access, provided in the list of roles sent in the Team Attribute Name attribute. |
| data.attributes.sso-api-token-session-timeout | integer | 1209600 | Specifies the Single Sign On session timeout in seconds. Defaults to 14 days. |

```
{
  "data": {
    "attributes": {
      "enabled": true,
      "debug": false,
      "idp-cert": "SAMPLE-CERTIFICATE",
      "slo-endpoint-url": "https://example.com/slo",
      "sso-endpoint-url": "https://example.com/sso",
      "attr-username": "Username",
      "attr-groups": "MemberOf",
      "attr-site-admin": "SiteAdmin",
      "site-admin-role": "site-admins",
      "sso-api-token-session-timeout": 1209600
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/admin/saml-settings
```

Sample Response

```
{
  "data": {
    "id": "saml",
    "type": "saml-settings",
    "attributes": {
      "enabled": true,
      "debug": false,
      "idp-cert": "SAMPLE-CERTIFICATE",
      "slo-endpoint-url": "https://example.com/slo",
      "sso-endpoint-url": "https://example.com/sso",
      "attr-username": "Username",
      "attr-groups": "MemberOf",
      "attr-site-admin": "SiteAdmin",
      "site-admin-role": "site-admins",
      "sso-api-token-session-timeout": 1209600,
      "acs-consumer-url": "https://example.com/users/saml/auth",
      "metadata-url": "https://example.com/users/saml/metadata"
    }
  }
}
```

List SMTP Settings

GET /api/v2/admin/smtp-settings

| Status | Response | Reason |
|--|--|--------------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "smtp-settings") | Successfully listed
SMTP settings |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/admin/smtp-settings
```

Sample Response

```
{
  "data": {
    "id": "smtp",
    "type": "smtp-settings",
    "attributes": {
      "enabled": true,
      "host": "example.com",
      "port": 25,
      "sender": "sample_user@example.com",
      "auth": "login",
      "username": "sample_user"
    }
  }
}
```

Update SMTP Settings

PATCH /api/v2/admin/smtp-settings

When a request to this endpoint is submitted, a test message will be sent to the specified `test-email-address`. If the test message delivery fails, the API will return an error code indicating the reason for the failure.

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "smtp-settings") | Successfully updated the SMTP settings |
| 401
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401) | JSON API error object (http://jsonapi.org/format/#error-objects) | SMTP user credentials are invalid |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |
| 500
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/504) | JSON API error object (http://jsonapi.org/format/#error-objects) | SMTP server returned a server error |
| 504
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/504) | JSON API error object (http://jsonapi.org/format/#error-objects) | SMTP server timed out |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

If `data.attributes.enabled` is set to `true`, all remaining attributes must have valid values. You can omit attributes if they have a default value, or if a value was set by a previous update. Omitted attributes keep their previous values.

| Key path | Type | Default | Description |
|------------------------------------|---------|---------|---|
| data.attributes.enabled | bool | false | Allows SMTP to be used. If true, all remaining attributes must have valid values. |
| data.attributes.host | string | | The host address of the SMTP server. |
| data.attributes.port | integer | | The port of the SMTP server. |
| data.attributes.sender | string | | The desired sender address. |
| data.attributes.auth | string | "none" | The authentication type. Valid values are "none", "plain", and "login". |
| data.attributes.username | string | | The username used to authenticate to the SMTP server. Only required if data.attributes.auth is set to "login" or "plain". |
| data.attributes.password | string | | The username used to authenticate to the SMTP server. Only required if data.attributes.auth is set to "login" or "plain". |
| data.attributes.test-email-address | string | | The email address to send a test message to. Not persisted and only used during testing. |

Sample Payload

```
{
  "data": {
    "attributes": {
      "enabled": true,
      "host": "example.com",
      "port": 25,
      "sender": "sample_user@example.com",
      "auth": "login",
      "username": "sample_user",
      "password": "sample_password",
      "test-email-address": "test@example.com"
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/admin/smtp-settings
```

Sample Response

```
{
  "data": {
    "id": "smtp",
    "type": "smtp-settings",
    "attributes": {
      "enabled": true,
      "host": "example.com",
      "port": 25,
      "sender": "sample_user@example.com",
      "auth": "login",
      "username": "sample_user"
    }
  }
}
```

List Twilio Settings

GET /api/v2/admin/twilio-settings

| Status | Response | Reason |
|--|--|-------------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "twilio-settings") | Successfully listed Twilio settings |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/admin/twilio-settings
```

Sample Response

```
{
  "data": {
    "id": "twilio",
    "type": "twilio-settings",
    "attributes": {
      "enabled": true,
      "account-sid": "12345abcd",
      "from-number": "555-555-5555"
    }
  }
}
```

Update Twilio Settings

PATCH /api/v2/admin/twilio-settings

| Status | Response | Reason |
|--|---|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document (https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "twilio-settings") | Successfully listed Twilio settings |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

If `data.attributes.enabled` is set to `true`, all remaining attributes must have valid values. You can omit attributes if they have a default value, or if a value was set by a previous update. Omitted attributes keep their previous values.

| Key path | Type | Default | Description |
|--|--------|--------------------|---|
| <code>data.attributes.enabled</code> | bool | <code>false</code> | Allows Twilio to be used. If true, all remaining attributes must have valid values. |
| <code>data.attributes.account-sid</code> | string | | The Twilio account id. |
| <code>data.attributes.auth-token</code> | string | | The Twilio authentication token. |
| <code>data.attributes.from-number</code> | string | | The Twilio registered phone number that will be used to send the message. |

```
{
  "data": {
    "attributes": {
      "enabled": true,
      "account-sid": "12345abcd",
      "auth-token": "sample_token",
      "from-number": "555-555-5555"
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/admin/twilio-settings
```

Sample Response

```
{
  "data": {
    "id": "twilio",
    "type": "twilio-settings",
    "attributes": {
      "enabled": true,
      "account-sid": "12345abcd",
      "from-number": "555-555-5555"
    }
  }
}
```

Verify Twilio Settings

`POST /api/v2/admin/twilio-settings/verify`

Uses the `test-number` attribute to send a test SMS when Twilio is enabled.

| Status | Response | Reason |
|---|---|--|
| 200 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | none | Twilio test message sent successfully |
| 400 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Verification settings invalid (missing test number, Twilio disabled, etc.) |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User unauthorized to perform action |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|--|--------|---------|---|
| <code>data.attributes.test-number</code> | string | | The target phone number for the test SMS. Not persisted and only used during testing. |

```
{  
  "data": {  
    "attributes": {  
      "test-number": "555-555-0000"  
    }  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/admin/twilio-settings/verify
```

Admin Terraform Versions API

Note: These API endpoints are in beta and are subject to change.

These API endpoints are available in Private Terraform Enterprise as of version 201807-1.

The Terraform Versions Admin API contains endpoints to help site administrators manage known versions of Terraform.

List all Terraform versions

GET /admin/terraform-versions

This endpoint lists all known versions of Terraform.

| Status | Response | Reason |
|--|---|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "terraform-versions") | Successfully listed Terraform versions |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is not an administrator. |

Query Parameters

This endpoint supports pagination with standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|--------------|---|
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 Terraform versions per page. |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  "https://app.terraform.io/api/v2/admin/terraform-versions"
```

Sample Response

```
{
  "data": [
    {
      "id": "tool-L4oe7rNwn7J4E5Yr",
      "type": "terraform-versions",
      "attributes": {
        "version": "0.11.8",
        "url": "https://releases.hashicorp.com/terraform/0.11.8/terraform_0.11.8_linux_amd64.zip",
        "sha": "84ccfb8e13b5fce63051294f787885b76a1fedef6bdbecf51c5e586c9e20c9b7",
        "official": true,
        "enabled": true,
        "beta": false,
        "usage": 0,
        "created-at": "2018-08-15T22:34:24.561Z"
      }
    },
    {
      "id": "tool-qcbYn12vuRKPgPpy",
      "type": "terraform-versions",
      "attributes": {
        "version": "0.11.7",
        "url": "https://releases.hashicorp.com/terraform/0.11.7/terraform_0.11.7_linux_amd64.zip",
        "sha": "6b8ce67647a59b2a3f70199c304abca0ddec0e49fd060944c26f666298e23418",
        "official": true,
        "enabled": true,
        "beta": false,
        "usage": 2,
        "created-at": null
      }
    }
  ],
  "links": {
    "self": "https://tfe.example.com/api/v2/admin/terraform-versions?page%5Bnumber%5D=1&page%5Bsize%5D=20",
    "first": "https://tfe.example.com/api/v2/admin/terraform-versions?page%5Bnumber%5D=1&page%5Bsize%5D=20",
    "prev": null,
    "next": "https://tfe.example.com/api/v2/admin/terraform-versions?page%5Bnumber%5D=2&page%5Bsize%5D=20",
    "last": "https://tfe.example.com/api/v2/admin/terraform-versions?page%5Bnumber%5D=4&page%5Bsize%5D=20"
  },
  "meta": {
    "pagination": {
      "current-page": 1,
      "prev-page": null,
      "next-page": 2,
      "total-pages": 4,
      "total-count": 70
    }
  }
}
```

Create a Terraform version

POST /admin/terraform-versions

| Status | Response | Reason |
|--|---|--|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "terraform-versions") | The Terraform version was successfully created |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is not an administrator |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Validation errors |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------------|--------|---------|--|
| data.type | string | | Must be "terraform-versions" |
| data.attributes.version | string | | A semantic version string (e.g. "0.11.0") |
| data.attributes.url | string | | The URL where a ZIP-compressed 64-bit Linux binary of this version can be downloaded |
| data.attributes.sha | string | | The SHA-256 checksum of the compressed Terraform binary |
| data.attributes.official | bool | false | Whether or not this is an official release of Terraform |
| data.attributes.enabled | bool | true | Whether or not this version of Terraform is enabled for use in Terraform Enterprise |
| data.attributes.beta | bool | false | Whether or not this version of Terraform is a beta pre-release |

Sample Payload

```
{
  "data": {
    "type": "terraform-versions",
    "attributes": {
      "version": "0.11.8",
      "url": "https://releases.hashicorp.com/terraform/0.11.8/terraform_0.11.8_linux_amd64.zip",
      "sha": "84ccfb8e13b5fce63051294f787885b76a1fedef6bdbecf51c5e586c9e20c9b7",
      "official": true,
      "enabled": true,
      "beta": false
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/admin/terraform-versions
```

Sample Response

```
{
  "data": {
    "id": "tool-L4oe7rNwn7J4E5Yr",
    "type": "terraform-versions",
    "attributes": {
      "version": "0.11.8",
      "url": "https://releases.hashicorp.com/terraform/0.11.8/terraform_0.11.8_linux_amd64.zip",
      "sha": "84ccfb8e13b5fce63051294f787885b76a1fedef6bdbecf51c5e586c9e20c9b7",
      "official": true,
      "enabled": true,
      "beta": false,
      "usage": 0,
      "created-at": "2018-08-15T22:34:24.561Z"
    }
  }
}
```

Show a Terraform version

GET /admin/terraform-versions/:id

| Parameter | Description | |
|--|---|--|
| :id | The ID of the Terraform version to show | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "terraform-versions") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Terraform version not found, or client is not an administrator |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/admin/terraform-versions/tool-L4oe7rNwn7J4E5Yr
```

Sample Response

```
{
  "data": {
    "id": "tool-L4oe7rNwn7J4E5Yr",
    "type": "terraform-versions",
    "attributes": {
      "version": "0.11.8",
      "url": "https://releases.hashicorp.com/terraform/0.11.8/terraform_0.11.8_linux_amd64.zip",
      "sha": "84ccfb8e13b5fce63051294f787885b76a1fedef6bdbecf51c5e586c9e20c9b7",
      "official": true,
      "enabled": true,
      "beta": false,
      "usage": 0,
      "created-at": "2018-08-15T22:34:24.561Z"
    }
  }
}
```

Update a Terraform version

PATCH /admin/terraform-versions/:id

| Parameter | Description | |
|--|---|--|
| :id | The ID of the Terraform version to update | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "terraform-versions") | The Terraform version was successfully updated |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Terraform version not found, or client is not an administrator |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Validation errors |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------------|--------|------------------|--|
| data.type | string | | Must be "terraform-versions" |
| data.attributes.version | string | (previous value) | A semantic version string (e.g. "0.11.0") |
| data.attributes.url | string | (previous value) | The URL where a ZIP-compressed 64-bit Linux binary of this version can be downloaded |
| data.attributes.sha | string | (previous value) | The SHA-256 checksum of the compressed Terraform binary |
| data.attributes.official | bool | (previous value) | Whether or not this is an official release of Terraform |
| data.attributes.enabled | bool | (previous value) | Whether or not this version of Terraform is enabled for use in Terraform Enterprise |
| data.attributes.beta | bool | (previous value) | Whether or not this version of Terraform is a beta pre-release |

Sample Payload

```
{
  "data": {
    "type": "terraform-versions",
    "attributes": {
      "official": true,
      "beta": false
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/admin/terraform-versions/tool-L4oe7rNwn7J4E5Yr
```

Sample Response

```
{
  "data": {
    "id": "tool-L4oe7rNwn7J4E5Yr",
    "type": "terraform-versions",
    "attributes": {
      "version": "0.11.8",
      "url": "https://releases.hashicorp.com/terraform/0.11.8/terraform_0.11.8_linux_amd64.zip",
      "sha": "84ccfb8e13b5fce63051294f787885b76a1fedef6bdbecf51c5e586c9e20c9b7",
      "official": true,
      "enabled": true,
      "beta": false,
      "usage": 0,
      "created-at": "2018-08-15T22:34:24.561Z"
    }
  }
}
```

Delete a Terraform version

`DELETE /admin/terraform-versions/:id`

This endpoint removes a Terraform version from Terraform Enterprise. Versions cannot be removed if they are labeled as official versions of Terraform or if there are workspaces using them.

| Parameter | Description | |
|---|--|---|
| <code>:id</code> | The ID of the Terraform version to delete | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Empty response | The Terraform version was successfully deleted |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object
(http://jsonapi.org/format/#error-objects) | Terraform version not found, or client is not an administrator |
| 422 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object
(http://jsonapi.org/format/#error-objects) | The Terraform version cannot be removed (it is official or is in use) |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
https://app.terraform.io/api/v2/admin/terraform-versions/tool-L4oe7rNwn7J4E5Yr
```

Admin Users API

Note: These API endpoints are in beta and are subject to change.

These API endpoints are available in Private Terraform Enterprise as of version 201807-1.

The Users Admin API contains endpoints to help site administrators manage user accounts.

List all users

GET /admin/users

This endpoint lists all user accounts in the Terraform Enterprise installation.

| Status | Response | Reason |
|--|--|---------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | Successfully listed users |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is not an administrator. |

Query Parameters

These are standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|-------------------|--|
| q | Optional. A search query string. Users are searchable by username and email address. |
| filter[admin] | Optional. Can be "true" or "false" to show only administrators or non-administrators. |
| filter[suspended] | Optional. Can be "true" or "false" to show only suspended users or users who are not suspended. |
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 users per page. |

Available Related Resources

This GET endpoint can optionally return related resources, if requested with the `include` query parameter (/docs/enterprise/api/index.html#inclusion-of-related-resources). The following resource types are available:

| Resource Name | Description |
|---------------|--|
| organizations | A list of organizations that each user is a member of. |

| Resource Name | Description |
|---------------|-------------|
|---------------|-------------|

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  "https://app.terraform.io/api/v2/admin/users"
```

Sample Response

```
{
  "data": [
    {
      "id": "user-ZL4MsEKnd6iTigTb",
      "type": "users",
      "attributes": {
        "username": "myuser",
        "email": "myuser@example.com",
        "avatar-url": "https://www.gravatar.com/avatar/3a23b75d5aa41029b88b73f47a0d90db?s=100&d=mm",
        "is-admin": true,
        "is-suspended": false,
        "is-service-account": false
      },
      "relationships": {
        "organizations": {
          "data": [
            {
              "id": "my-organization",
              "type": "organizations"
            }
          ]
        }
      },
      "links": {
        "self": "/api/v2/users/myuser"
      }
    }
  ],
  "links": {
    "self": "https://app.terraform.io/api/v2/admin/users?page%5Bnumber%5D=1&page%5Bsize%5D=20",
    "first": "https://app.terraform.io/api/v2/admin/users?page%5Bnumber%5D=1&page%5Bsize%5D=20",
    "prev": null,
    "next": null,
    "last": "https://app.terraform.io/api/v2/admin/users?page%5Bnumber%5D=1&page%5Bsize%5D=20"
  },
  "meta": {
    "pagination": {
      "current-page": 1,
      "prev-page": null,
      "next-page": null,
      "total-pages": 1,
      "total-count": 1
    },
    "status-counts": {
      "total": 1,
      "suspended": 0,
      "admin": 1
    }
  }
}
}
```

Delete a user account

`DELETE /admin/users/:id`

This endpoint deletes a user's account from Terraform Enterprise. To prevent unowned organizations, a user cannot be deleted if they are the sole owner of any organizations. The organizations must be given a new owner or deleted first.

| Parameter | Description | |
|--|--|--|
| :id | The ID of the user to delete. | |
| Status | Response | Reason |
| 204
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Empty body | Successfully removed the user account. |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object
(http://jsonapi.org/format/#error-objects) | Client is not an administrator. |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object
(http://jsonapi.org/format/#error-objects) | The user cannot be deleted because they are the sole owner of one or more organizations. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
"https://app.terraform.io/api/v2/admin/users/user-ZL4MsEKnd6iTigTb"
```

Suspend a user

POST /admin/users/:id/actions/suspend

| Parameter | Description | |
|--|--|--|
| :id | The ID of the user to suspend. | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | Successfully suspended the user's account. |
| 400
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400) | JSON API error object (http://jsonapi.org/format/#error-objects) | User is already suspended. |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is not an administrator. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
"https://app.terraform.io/api/v2/admin/users/user-ZL4MsEKnd6iTigTb/actions/suspend"
```

Sample Response

```
{
  "data": {
    "id": "user-ZL4MsEKnd6iTigTb",
    "type": "users",
    "attributes": {
      "username": "myuser",
      "email": "myuser@example.com",
      "avatar-url": "https://www.gravatar.com/avatar/3a23b75d5aa41029b88b73f47a0d90db?s=100&d=mm",
      "is-admin": false,
      "is-suspended": true,
      "is-service-account": false
    },
    "relationships": {
      "organizations": {
        "data": [
          {
            "id": "my-organization",
            "type": "organizations"
          }
        ]
      }
    },
    "links": {
      "self": "/api/v2/users/myuser"
    }
  }
}
```

Re-activate a suspended user

POST /admin/users/:id/actions/unsuspend

| Parameter | Description | Reason |
|-----------|------------------------------------|---|
| :id | The ID of the user to re-activate. | Successfully re-activated the user's account. |

This endpoint re-activates a suspended user's account, allowing them to resume authenticating and accessing resources.

| Status | Response | Reason |
|--|---|--|
| 400
(Status/400">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/400) | JSON API error object (http://jsonapi.org/format/#error-objects) | User is not suspended. |
| 404
(Status/404">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not found, or client is not an administrator. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
"https://app.terraform.io/api/v2/admin/users/user-ZL4MsEKnd6iTigTb/actions/unsuspend"
```

Sample Response

```
{
  "data": {
    "id": "user-ZL4MsEKnd6iTigTb",
    "type": "users",
    "attributes": {
      "username": "myuser",
      "email": "myuser@example.com",
      "avatar-url": "https://www.gravatar.com/avatar/3a23b75d5aa41029b88b73f47a0d90db?s=100&d=mm",
      "is-admin": false,
      "is-suspended": false,
      "is-service-account": false
    },
    "relationships": {
      "organizations": {
        "data": [
          {
            "id": "my-organization",
            "type": "organizations"
          }
        ]
      }
    },
    "links": {
      "self": "/api/v2/users/myuser"
    }
  }
}
```

Grant a user administrative privileges

POST /admin/users/:id/actions/grant_admin

| Parameter | Description | |
|--|--|--|
| :id | The ID of the user to make an administrator. | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | Successfully made the user an administrator. |
| 400
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400) | JSON API error object (http://jsonapi.org/format/#error-objects) | User is already an administrator. |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not found, or client is not an administrator. |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Validation errors |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
"https://app.terraform.io/api/v2/admin/users/user-ZL4MsEKnd6iTigTb/actions/grant_admin"
```

Sample Response

```
{
  "data": {
    "id": "user-ZL4MsEKnd6iTigTb",
    "type": "users",
    "attributes": {
      "username": "myuser",
      "email": "myuser@example.com",
      "avatar-url": "https://www.gravatar.com/avatar/3a23b75d5aa41029b88b73f47a0d90db?s=100&d=mm",
      "is-admin": true,
      "is-suspended": false,
      "is-service-account": false
    },
    "relationships": {
      "organizations": {
        "data": [
          {
            "id": "my-organization",
            "type": "organizations"
          }
        ]
      }
    },
    "links": {
      "self": "/api/v2/users/myuser"
    }
  }
}
```

Revoke an user's administrative privileges

POST /admin/users/:id/actions/revoke_admin

| Parameter | Description | |
|--|--|--|
| :id | The ID of the administrator to demote. | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | Successfully made the user an administrator. |
| 400
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400) | JSON API error object (http://jsonapi.org/format/#error-objects) | User is not an administrator. |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not found, or client is not an administrator. |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  "https://app.terraform.io/api/v2/admin/users/user-ZL4MsEKnd6iTigTb/actions/revoke_admin"
```

Sample Response

```
{  
  "data": {  
    "id": "user-ZL4MsEKnd6iTigTb",  
    "type": "users",  
    "attributes": {  
      "username": "myuser",  
      "email": "myuser@example.com",  
      "avatar-url": "https://www.gravatar.com/avatar/3a23b75d5aa41029b88b73f47a0d90db?s=100&d=mm",  
      "is-admin": false,  
      "is-suspended": false,  
      "is-service-account": false  
    },  
    "relationships": {  
      "organizations": {  
        "data": [  
          {  
            "id": "my-organization",  
            "type": "organizations"  
          }  
        ]  
      }  
    },  
    "links": {  
      "self": "/api/v2/users/myuser"  
    }  
  }  
}
```

Disable a user's two-factor authentication

POST /admin/users/:id/actions/disable_two_factor

| Parameter | Description |
|-----------|--|
| :id | The ID of the user to disable 2FA for. |

This endpoint disables a user's two-factor authentication in the situation where they have lost access to their device and recovery codes. Before disabling a user's two-factor authentication, completing a security verification process is recommended to ensure the request is legitimate.

| Status | Response | Reason |
|--------|----------|--------|
|--------|----------|--------|

| Status | Response | Reason |
|--|--|---|
| 200
(Status/200">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | Successfully disabled the user's two-factor authentication. |
| 400
(Status/400">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/400) | JSON API error object (http://jsonapi.org/format/#error-objects) | User does not have two-factor authentication enabled. |
| 404
(Status/404">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not found, or client is not an administrator. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
"https://app.terraform.io/api/v2/admin/users/user-ZL4MsEKnd6iTigTb/actions/disable_two_factor"
```

Sample Response

```
{
  "data": {
    "id": "user-ZL4MsEKnd6iTigTb",
    "type": "users",
    "attributes": {
      "username": "myuser",
      "email": "myuser@example.com",
      "avatar-url": "https://www.gravatar.com/avatar/3a23b75d5aa41029b88b73f47a0d90db?s=100&d=mm",
      "is-admin": false,
      "is-suspended": false,
      "is-service-account": false
    },
    "relationships": {
      "organizations": {
        "data": [
          {
            "id": "my-organization",
            "type": "organizations"
          }
        ]
      }
    },
    "links": {
      "self": "/api/v2/users/myuser"
    }
  }
}
```

Impersonate another user

POST /admin/users/:id/actions/impersonate

| Parameter | Description |
|-----------|--|
| :id | The ID of the user to impersonate. It is not possible to impersonate service accounts or your own account. |

Impersonation allows an admin to begin a new session as another user in the system; for more information, see Impersonating a User ([/docs/enterprise/private/admin/resources.html#impersonating-a-user](#)) in the Private Terraform Enterprise administration section.

Note: Impersonation is intended as a UI feature ([/docs/enterprise/private/admin/resources.html#impersonating-a-user](#)), and this endpoint exists to support that UI. We do not recommend impersonating users via the API.

This endpoint does not respond with a body, but the response does include a `Set-Cookie` header to persist a new session.

Important: Impersonating via the API requires you to switch to an alternate authentication flow that is not based on TFE's normal API tokens; instead, you must acquire and persist both an `_atlas_session_data=...` cookie and an `X-CSRF-Token` header. Instructions for doing this are beyond the scope of this document. If you believe you need to automate user impersonation, please contact HashiCorp Support ([/docs/enterprise/private/faq.html#support-for-private-terraform-enterprise](#)) for assistance.

| Status | Response | Reason |
|---|---|--|
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Empty body | Successfully impersonated the user. |
| 400 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400) | JSON API error object (http://jsonapi.org/format/#error-objects) | A reason for impersonation is required. |
| 403 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/403) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is already impersonating another user. |
| 403 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/403) | JSON API error object (http://jsonapi.org/format/#error-objects) | User cannot be impersonated. |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not found, or client is not an administrator. |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|----------|--------|---------|--|
| reason | string | | A reason for impersonation, which will be recorded in the Audit Log. |

Sample Payload

```
{  
  "reason": "Reason for impersonation"  
}
```

Sample Request

```
curl \  
  --header "Cookie: $COOKIE" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/admin/users/user-ZL4MsEKnd6iTigTb/actions/impersonate
```

End an impersonation session

POST /admin/users/actions/unimpersonate

When an admin has used the above endpoint to begin an impersonation session, they can make a request to this endpoint, using the cookie provided originally, in order to end that session and log out as the impersonated user.

This endpoint does not respond with a body, but the response does include a Set-Cookie header to persist a new session as the original admin user. As such, this endpoint will have no effect unless the client is able to persist and use cookies.

| Status | Response | Reason |
|---|--|---|
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Empty body | Successfully ended the impersonation session. |
| 400 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400) | JSON API error object
(http://jsonapi.org/format/#error-objects) | Client is not in an impersonation session. |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object
(http://jsonapi.org/format/#error-objects) | Client is not an administrator. |

Sample Request

```
curl \  
  --header "Cookie: $COOKIE" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  https://app.terraform.io/api/v2/admin/users/actions/unimpersonate
```

Admin Workspaces API

Note: These API endpoints are in beta and are subject to change.

These API endpoints are available in Private Terraform Enterprise as of version 201807-1.

The Workspaces Admin API contains endpoints to help site administrators manage workspaces.

List all workspaces

GET /admin/workspaces

This endpoint lists all workspaces in the Terraform Enterprise installation.

| Status | Response | Reason |
|--|---|---------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "workspaces") | Successfully listed workspaces |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is not an administrator. |

Query Parameters

These are standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|-----------------------------|---|
| q | Optional. A search query string. Workspaces are searchable by name and organization name. |
| filter[current_run][status] | Optional. A comma-separated list of Run statuses to restrict results to, including of any of the following: "pending", "planning", "planned", "confirmed", "applying", "applied", "discarded", "errored", "canceled", "policy_checking", "policy_override", and/or "policy_checked". |
| sort | Optional. Allows sorting the returned workspaces. Valid values are "name" (the default) and "current-run.created-at" (which sorts by the time of the current run). Prepending a hyphen to the sort parameter will reverse the order (e.g. "-name" to reverse the default order) |
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 workspaces per page. |

Available Related Resources

This GET endpoint can optionally return related resources, if requested with the `include` query parameter (/docs/enterprise/api/index.html#include-related-resources). The following resource types are available:

| Resource Name | Description |
|---------------------|--|
| organization | The organization for each returned workspace. |
| organization.owners | A list of owners for each workspace's associated organization. |
| current_run | The current run for each returned workspace. |

Sample Request

```
curl \
  --header "Authorization: Bearer $TOKEN" \
  --header "Content-Type: application/vnd.api+json" \
  "https://app.terraform.io/api/v2/admin/workspaces"
```

Sample Response

```
{  
  "data": [  
    {  
      "id": "ws-2HRvNs49EWPjDqT1",  
      "type": "workspaces",  
      "attributes": {  
        "name": "my-workspace",  
        "locked": false,  
        "vcs-repo": {  
          "identifier": "my-organization/my-repository"  
        }  
      },  
      "relationships": {  
        "organization": {  
          "data": {  
            "id": "my-organization",  
            "type": "organizations"  
          }  
        },  
        "current-run": {  
          "data": {  
            "id": "run-jm8ekSaW3F52FACN",  
            "type": "runs"  
          }  
        }  
      },  
      "links": {  
        "self": "/api/v2/organizations/my-organization/workspaces/my-workspace"  
      }  
    }  
,  
    "links": {  
      "self": "http://localhost:3000/api/v2/admin/workspaces?page%5Bnumber%5D=1&page%5Bsize%5D=20",  
      "first": "http://localhost:3000/api/v2/admin/workspaces?page%5Bnumber%5D=1&page%5Bsize%5D=20",  
      "prev": null,  
      "next": null,  
      "last": "http://localhost:3000/api/v2/admin/workspaces?page%5Bnumber%5D=1&page%5Bsize%5D=20"  
    },  
    "meta": {  
      "pagination": {  
        "current": 1,  
        "total": 1  
      }  
    }  
  ]  
}
```

```

        "current-page": 1,
        "prev-page": null,
        "next-page": null,
        "total-pages": 0,
        "total-count": 1
    },
    "status-counts": {
        "pending": 1,
        "planning": 0,
        "planned": 0,
        "confirmed": 0,
        "applying": 0,
        "applied": 0,
        "discarded": 0,
        "errored": 0,
        "canceled": 0,
        "policy-checking": 0,
        "policy-override": 0,
        "policy-checked": 0,
        "none": 0,
        "total": 1,
    }
}
}
}

```

Show a workspace

GET /admin/workspaces/:id

This endpoint lists all workspaces in the Terraform Enterprise installation.

| Status | Response | Reason |
|--|---|---------------------------------|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "workspaces") | Successfully listed workspaces |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Client is not an administrator. |

Query Parameters

| Parameter | Description |
|---------------|------------------|
| :workspace_id | The workspace ID |

Available Related Resources

This GET endpoint can optionally return related resources, if requested with the `include` query parameter (</docs/enterprise/api/index.html#inclusion-of-related-resources>). The following resource types are available:

| Resource Name | Description |
|---------------------|--|
| organization | The organization for each returned workspace. |
| organization.owners | A list of owners for each workspace's associated organization. |
| current_run | The current run for each returned workspace. |

Sample Request

```
curl \  
--header "Authorization: Bearer $TOKEN" \  
--header "Content-Type: application/vnd.api+json" \  
"https://app.terraform.io/api/v2/admin/workspaces/ws-2HRvNs49EWPjDqT1"
```

Sample Response

```
{  
  "data": {  
    "id": "ws-2HRvNs49EWPjDqT1",  
    "type": "workspaces",  
    "attributes": {  
      "name": "my-workspace",  
      "locked": false,  
      "vcs-repo": {  
        "identifier": "my-organization/my-repository"  
      }  
    },  
    "relationships": {  
      "organization": {  
        "data": {  
          "id": "my-organization",  
          "type": "organizations"  
        }  
      },  
      "current-run": {  
        "data": {  
          "id": "run-jm8ekSaW3F52FACN",  
          "type": "runs"  
        }  
      }  
    },  
    "links": {  
      "self": "/api/v2/organizations/my-organization/workspaces/my-workspace"  
    }  
  }  
}
```

Destroy a workspace

DELETE /admin/workspaces/:id

| Parameter | Description | |
|---|---|--|
| :workspace_id | The workspace ID | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | | The workspace was successfully destroyed |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace not found or user unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request DELETE \  
  https://app.terraform.io/api/v2/admin/workspaces/ws-2HRvNs49EWPjDqT1
```

Sample Response

The response body will be empty if successful.

Applies API

Note: These API endpoints are in beta and are subject to change.

An apply represents the results of applying a Terraform Run's execution plan.

Show an apply

GET /applies/:id

| Parameter | Description | |
|-----------|------------------------------|--|
| id | The ID of the apply to show. | |

There is no endpoint to list applies. You can find the ID for an apply in the `relationships.apply` property of a run object.

| Status | Response | Reason |
|--|--|---|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "applies") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Apply not found, or user unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  https://app.terraform.io/api/v2/applies/apply-47MBvjwzBG8YKc2v
```

Sample Response

```
{  
  "data": {  
    "id": "apply-47MBvjwzBG8YKc2v",  
    "type": "applies",  
    "attributes": {  
      "status": "finished",  
      "status-timestamps": {  
        "queued-at": "2018-10-17T18:58:27+00:00",  
        "started-at": "2018-10-17T18:58:29+00:00",  
        "finished-at": "2018-10-17T18:58:37+00:00"  
      },  
      "log-read-url": "https://archivist.terraform.io/v1/object/dmF1bHQ6djE60FA1eEdlSFVHRSs4YUcwaW83a1dRR  
DA0U2E3T3FiWk1HM2NyQlNtcS9JS1hHN3dmTXJmaFhEYTLhdTF1ZlgxZ2wzVC9kVTLNcjRPOEJkK050VFI3U3dvS2ZuaUhFSGpVenJVUF  
YzSFVZQ1VZYno3T3UyYjdDRVRPRE5pbWJDVTIrNllQTENyTndYd1Y0ak1DL1dPVlN1VlnxKzYzbWlIcnJPa2dRRkJZZGtFeTNiaU84YlZ  
4QWs2QzLLY3VJb3lmWlIrajF4a1hYZTlsWnFYemRkL2pNOG9Zc0ZDakdVMCtURUE3dDNMODRsRnY4cWl1dUN5dUVuUzdnZzFwL3BNeHlw  
bXNXZWWrUDhXdzhGNnF4c3dqaxLzs29oL3FKakI5dm9uYU5ZKzAybnloREdnQ3J2Rk5WmlBJemZQTg",  
      "resource-additions": 1,  
      "resource-changes": 0,  
      "resource-destructions": 0  
    },  
    "relationships": {  
      "state-versions": {  
        "data": [  
          {  
            "id": "sv-TpnsuD3iewwsfeRD",  
            "type": "state-versions"  
          },  
          {  
            "id": "sv-Fu1n6a3TgJ1Typq9",  
            "type": "state-versions"  
          }  
        ]  
      }  
    },  
    "links": {  
      "self": "/api/v2/applies/apply-47MBvjwzBG8YKc2v"  
    }  
  }  
}
```

Configuration Versions API

Note: These API endpoints are in beta and are subject to change.

Note: Before working with the runs or configuration versions APIs, read the API-driven run workflow ([/docs/enterprise/run/api.html](#)) page, which includes both a full overview of this workflow and a walkthrough of a simple implementation of it.

A configuration version (configuration-version) is a resource used to reference the uploaded configuration files. It is associated with the run to use the uploaded configuration files for performing the plan and apply.

List Configuration Versions

GET /workspaces/:workspace_id/configuration-versions

| Parameter | Description |
|---------------|--|
| :workspace_id | The ID of the workspace to list configurations from. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace (/docs/enterprise/api/workspaces.html#show-workspace) endpoint. |

Query Parameters

This endpoint supports pagination with standard URL query parameters ([/docs/enterprise/api/index.html#query-parameters](#)); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|--------------|---|
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 configuration versions per page. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/workspaces/ws-2Qhk7LHgbMrm3grF/configuration-versions
```

Sample Response

```
{
  "data": [
    {
      "id": "cv-ntv3HbhJqvFzamy7",
      "type": "configuration-versions",
      "attributes": {
        "error": null,
        "error-message": null,
        "source": "gitlab",
        "status": "uploaded",
        "status-timestamps": []
      },
      "relationships": {
        "ingress-attributes": {
          "data": {
            "id": "ia-i4MrTxmQXYxH2nYD",
            "type": "ingress-attributes"
          },
          "links": {
            "related": "/api/v2/configuration-versions/cv-ntv3HbhJqvFzamy7/ingress-attributes"
          }
        },
        "links": {
          "self": "/api/v2/configuration-versions/cv-ntv3HbhJqvFzamy7"
        }
      }
    }
  ]
}
```

Show a Configuration Version

GET /configuration-versions/:configuration-id

| Parameter | Description |
|-------------------|--------------------------------------|
| :configuration-id | The id of the configuration to show. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/configuration-versions/cv-ntv3HbhJqvFzamy7
```

Sample Response

```
{
  "data": {
    "id": "cv-ntv3HbhJqvFzamy7",
    "type": "configuration-versions",
    "attributes": {
      "error": null,
      "error-message": null,
      "source": "gitlab",
      "status": "uploaded",
      "status-timestamps": {}
    },
    "relationships": {
      "ingress-attributes": {
        "data": {
          "id": "ia-i4MrTxmQXYxH2nYD",
          "type": "ingress-attributes"
        },
        "links": {
          "related": "/api/v2/configuration-versions/cv-ntv3HbhJqvFzamy7/ingress-attributes"
        }
      }
    },
    "links": {
      "self": "/api/v2/configuration-versions/cv-ntv3HbhJqvFzamy7"
    }
  }
}
```

Create a Configuration Version

`POST /workspaces/:workspace_id/configuration-versions`

| Parameter | Description |
|----------------------------|---|
| <code>:workspace_id</code> | The ID of the workspace to create the new configuration version in. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace (/docs/enterprise/api/workspaces.html#show-workspace) endpoint. |

Note: This endpoint cannot be accessed with organization tokens ([/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts](#)). You must access it with a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) or team token ([/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts](#)).

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|---------------------------------|---------|---------|---|
| data.attributes.auto-queue-runs | boolean | true | When true, runs are queued automatically when the configuration version is uploaded. |
| data.attributes.speculative | boolean | false | When true, this configuration version may only be used to create runs which are speculative, that is, can neither be confirmed nor applied. |

Sample Payload

```
{  
  "data": {  
    "type": "configuration-versions",  
    "attributes": {  
      "auto-queue-runs": true  
    }  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/workspaces/ws-2Qhk7LHgbMrm3grF/configuration-versions
```

Sample Response

```
{
  "data": {
    "id": "cv-UYwHEakurukz85nW",
    "type": "configuration-versions",
    "attributes": {
      "auto-queue-runs": true,
      "error": null,
      "error-message": null,
      "source": "tfe-api",
      "status": "pending",
      "status-timestamps": {},
      "upload-url":
        "https://archivist.terraform.io/v1/object/9224c6b3-2e14-4cd7-adff-ed484d7294c2"
    },
    "relationships": {
      "ingress-attributes": {
        "data": null,
        "links": {
          "related":
            "/api/v2/configuration-versions/cv-UYwHEakurukz85nW/ingress-attributes"
        }
      }
    },
    "links": { "self": "/api/v2/configuration-versions/cv-UYwHEakurukz85nW" }
  }
}
```

Configuration Version Status

A configuration version will be in the pending status when initially created. It will remain pending until configuration files are supplied via upload, and while they are processed.

If upload and processing succeed, the configuration version status will then be uploaded. An uploaded configuration version is ready for use.

If upload and processing fail, the status will be errored, indicating that something went wrong.

Runs cannot be created using pending or errored configuration versions.

Upload Configuration Files

Note: If auto-queue-runs was either not provided or set to true during creation of the configuration version, a run using this configuration version will be automatically queued on the workspace. If auto-queue-runs was set to false explicitly, then it is necessary to create a run on the workspace ([/docs/enterprise/api/run.html#create-a-run](#)) manually after the configuration version is uploaded.

PUT `https://archivist.terraform.io/v1/object/<UNIQUE OBJECT ID>`

The URL is provided in the `upload-url` attribute in the configuration-versions resource.

Sample Request

@filename is the name of configuration file you wish to upload.

```
curl \  
  --header "Content-Type: application/octet-stream" \  
  --request PUT \  
  --data-binary @filename \  
  https://archivist.terraform.io/v1/object/4c44d964-eba7-4dd5-ad29-1ece7b99e8da
```

Available Related Resources

The GET endpoints above can optionally return related resources, if requested with the `include` query parameter ([/docs/enterprise/api/index.html#inclusion-of-related-resources](#)). The following resource types are available:

- `ingress_attributes` - The commit information used in the configuration

Registry Modules API

Note: These API endpoints are in beta and are subject to change.

Listing and reading modules, providers and versions

The Terraform Enterprise Module Registry implements the Registry standard API (</docs/registry/api.html>) for consuming the modules. Refer to the Module Registry HTTP API (</docs/registry/api.html>) to perform the following:

- Browse available modules
- Search modules by keyword
- List available versions for a specific module
- Download source code for a specific module version
- List latest version of a module for all providers
- Get the latest version for a specific module provider
- Get a specific module
- Download the latest version of a module

The TFE Module Registry endpoints differs from the Module Registry endpoints in the following ways:

- The `:namespace` parameter should be replaced with the organization name.
- The module registry discovery endpoints have the path prefix provided in the discovery document (</docs/registry/api.html#service-discovery>) which is currently `/api/registry/v1`.
- Authentication (</docs/enterprise/api/index.html#authentication>) is handled the same as all other TFE endpoints.

Sample Request

List available versions for the `consul` module for the `aws` provider on the module registry published from the Github organization `my-gh-repo-org`:

```
$ curl https://registry.terraform.io/v1/modules/my-gh-repo-org/consul/aws/versions
```

The same request for the same module and provider on the TFE module registry for `my-tfe-org` TFE organization:

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
https://app.terraform.io/api/registry/v1/modules/my-tfe-org/consul/aws/versions
```

Publish a Module from a VCS

POST /registry-modules

Publishes a new registry module from a VCS repository, with module versions managed automatically by the repository's tags. The publishing process will fetch all tags in the source repository that look like SemVer (<https://semver.org/>) versions with optional 'v' prefix. For each version, the tag is cloned and the config parsed to populate module details (input and output variables, readme, submodules, etc.). The Module Registry Requirements (/docs/registry/modules/publish.html#requirements) define additional requirements on naming, standard module structure and tags for releases.

| Status | Response | Reason |
|--|---|--|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "registry-modules") | Successfully published module |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not authorized |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|---|--------|---------|--|
| data.type | string | | Must be "registry-modules". |
| data.attributes.vcs-
repo.identifier | string | | The repository from which to ingress the configuration. |
| data.attributes.vcs-
repo.oauth-token-id | string | | The VCS Connection (OAuth Connection + Token) to use as identified. This ID can be obtained from the oauth-tokens (/docs/enterprise/api/oauth-tokens.html) endpoint. |

A VCS repository identifier is a reference to a VCS repository in the format :org/:repo, where :org and :repo refer to the organization (or project key, for Bitbucket Server) and repository in your VCS provider.

The OAuth Token ID identifies the VCS connection, and therefore the organization, that the module will be created in.

Sample Payload

```
{  
  "data": {  
    "attributes": {  
      "vcs-repo": {  
        "identifier": "SKI/terraform-aws-instance",  
        "oauth-token-id": "ot-hmAyP66qk2AMVdbJ"  
      }  
    },  
    "type": "registry-modules"  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/registry-modules
```

Sample Response

```
{  
  "data": {  
    "id": "mod-1JqHG3j71bwoukuX",  
    "type": "registry-modules",  
    "attributes": {  
      "name": "instance",  
      "provider": "aws",  
      "status": "pending",  
      "version-statuses": [],  
      "created-at": "2017-11-30T00:00:52.386Z",  
      "updated-at": "2017-11-30T00:00:52.386Z",  
      "permissions": {  
        "can-delete": true,  
        "can-resync": true,  
        "can-retry": true  
      }  
    },  
    "relationships": {  
      "organization": {  
        "data": {  
          "id": "org-QpXoEnULx3r2r1CA",  
          "type": "organizations"  
        }  
      }  
    },  
    "links": {  
      "self": "/api/v2/registry-modules/show/skierkowski-v2/instance/aws"  
    }  
  }  
}
```

Create a Module

POST /organizations/:organization_name/registry-modules

| Parameter | Description |
|--------------------|--|
| :organization_name | The name of the organization to create a module in. The organization must already exist, and the token authenticating the API request must belong to the "owners" team or a member of the "owners" team. |

Creates a new registry module without a backing VCS repository. After creating a module, a version must be created and uploaded in order to be usable. Modules created this way do not automatically update with new versions; instead, you must explicitly create and upload each new version with the Create a Module Version endpoint.

| Status | Response | Reason |
|--|---|--|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "registry-modules") | Successfully published module |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not authorized |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------------|--------|---------|--|
| data.type | string | | Must be "registry-modules". |
| data.attributes.name | string | | The name of this module. May contain alphanumeric characters and dashes. |
| data.attributes.provider | string | | Specifies the Terraform provider that this module is used for. |

For example, aws is a provider.

Sample Payload

```
{  
  "data": {  
    "type": "registry-modules",  
    "attributes": {  
      "name": "my-module",  
      "provider": "aws"  
    }  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/organizations/my-organization/registry-modules
```

Sample Response

```
{  
  "data": {  
    "attributes": {  
      "created-at": "2018-09-24T20:45:13.614Z",  
      "name": "my-module",  
      "permissions": {  
        "can-delete": true,  
        "can-resync": true,  
        "can-retry": true  
      },  
      "provider": "aws",  
      "status": "pending",  
      "updated-at": "2018-09-24T20:45:13.614Z",  
      "version-statuses": []  
    },  
    "id": "mod-kno8GMqyUFAdbExr",  
    "links": {  
      "self": "/api/v2/registry-modules/show/my-organization/my-module/aws"  
    },  
    "relationships": {  
      "organization": {  
        "data": {  
          "id": "org-qScjTapEAMHut5ky",  
          "type": "organizations"  
        }  
      }  
    },  
    "type": "registry-modules"  
  }  
}
```

Create a Module Version

POST /registry-modules/:organization_name/:name/:provider/versions

| Parameter | Description |
|--------------------|--|
| :organization_name | The name of the organization to create a module in. The organization must already exist, and the token authenticating the API request must belong to the "owners" team or a member of the "owners" team. |
| :name | The name of the module for which the version is being created. |
| :provider | The name of the provider for which the version is being created. |

Creates a new registry module version. This endpoint only applies to modules without a VCS repository; VCS-linked modules automatically create new versions for new tags. After creating the version, the module should be uploaded to the returned upload link.

| Status | Response | Reason |
|--|---|--|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "registry-module-versions") | Successfully published module version |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not authorized |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-------------------------|--------|---------|-------------------------------------|
| data.type | string | | Must be "registry-module-versions". |
| data.attributes.version | string | | A valid semver version string. |

Sample Payload

```
{  
  "data": {  
    "type": "registry-module-versions",  
    "attributes": {  
      "version": "1.2.3"  
    }  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/registry-modules/my-organization/my-module/aws/versions
```

Sample Response

```
{  
  "data": {  
    "id": "modver-qjjF7ArLXJSWU3WU",  
    "type": "registry-module-versions",  
    "attributes": {  
      "source": "tfe-api",  
      "status": "pending",  
      "version": "1.2.3",  
      "created-at": "2018-09-24T20:47:20.931Z",  
      "updated-at": "2018-09-24T20:47:20.931Z"  
    },  
    "relationships": {  
      "registry-module": {  
        "data": {  
          "id": "1881",  
          "type": "registry-modules"  
        }  
      }  
    },  
    "links": {  
      "upload": "https://archivist.terraform.io/v1/object/dmF1bHQ6djE6NWJPbHQ4QjV4R1ox..."  
    }  
  }  
}
```

Upload a Module Version

PUT <https://archivist.terraform.io/v1/object/<UNIQUE OBJECT ID>>

The URL is provided in the upload links attribute in the registry-module-versions resource.

Expected Archive Format

Terraform Enterprise expects the module version uploaded to be a tarball with the module in the root (not in a subdirectory).

Given the following folder structure:

```
terraform-null-test
├── README.md
├── examples
│   └── default
│       ├── README.md
│       └── main.tf
└── main.tf
```

Package the files in an archive format by running `tar zcvf module.tar.gz *` in the module's directory.

```
~$ cd terraform-null-test
terraform-null-test$ tar zcvf module.tar.gz *
a README.md
a examples
a examples/default
a examples/default/main.tf
a examples/default/README.md
a main.tf
```

Sample Request

```
curl \
  --header "Content-Type: application/octet-stream" \
  --request PUT \
  --data-binary @module.tar.gz \
  https://archivist.terraform.io/v1/object/dmF1bHQ6djE6NWJPbHQ4QjV4R1ox...
```

After the registry module version is successfully parsed by TFE, its status will become "ok".

Delete a Module

- POST /registry-modules/actions/delete/:organization_name/:name/:provider/:version
- POST /registry-modules/actions/delete/:organization_name/:name/:provider
- POST /registry-modules/actions/delete/:organization_name/:name

Parameters

| Parameter | Description |
|-----------|-------------|
|-----------|-------------|

| Parameter | Description |
|--------------------|--|
| :organization_name | The name of the organization to delete a module from. The organization must already exist, and the token authenticating the API request must belong to the "owners" team or a member of the "owners" team. |
| :name | The module name that the deletion will affect. |
| :provider | If specified, the provider for the module that the deletion will affect. |
| :version | If specified, the version for the module and provider that will be deleted. |

When removing modules, there are three versions of the endpoint, depending on how many parameters are specified.

- If all parameters (module, provider, and version) are specified, the specified version for the given provider of the module is deleted.
- If module and provider are specified, the specified provider for the given module is deleted along with all its versions.
- If only module is specified, the entire module is deleted.

If a version deletion would leave a provider with no versions, the provider will be deleted. If a provider deletion would leave a module with no providers, the module will be deleted.

| Status | Response | Reason |
|---|---|---|
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | Success |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Module, provider, or version not found or user not authorized |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
https://app.terraform.io/api/v2/registry-modules/actions/delete/skierkowski-v2/instance
```

OAuth Clients API

Note: These API endpoints are in beta and are subject to change.

An OAuth Client represents the connection between an organization and a VCS provider.

List OAuth Clients

GET /organizations/:organization_name/oauth-clients

| Parameter | Description | |
|--------------------|-------------------------------|--|
| :organization_name | The name of the organization. | |

This endpoint allows you to list VCS connections between an organization and a VCS provider (GitHub, Bitbucket, or GitLab) for use when creating or setting up workspaces.

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "oauth-clients") | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found, or user unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request GET \  
  https://app.terraform.io/api/v2/organizations/my-organization/oauth-clients
```

Sample Response

```
{
  "data": [
    {
      "id": "oc-XKFwG6ggfA9n7t1K",
      "type": "oauth-clients",
      "attributes": {
        "created-at": "2018-04-16T20:42:53.771Z",
        "callback-url": "https://app.terraform.io/auth/35936d44-842c-4ddd-b4d4-7c741383dc3a/callback",
        "connect-path": "/auth/35936d44-842c-4ddd-b4d4-7c741383dc3a?organization_id=1",
        "service-provider": "github",
        "service-provider-display-name": "GitHub",
        "http-url": "https://github.com",
        "api-url": "https://api.github.com",
        "key": null,
        "rsa-public-key": null
      },
      "relationships": {
        "organization": {
          "data": {
            "id": "my-organization",
            "type": "organizations"
          },
          "links": {
            "related": "/api/v2/organizations/my-organization"
          }
        },
        "oauth-tokens": {
          "data": [],
          "links": {
            "related": "/api/v2/oauth-tokens/oc-XKFwG6ggfA9n7t1K"
          }
        }
      }
    }
  ]
}
```

Show an OAuth Client

GET /oauth-clients/:id

| Parameter | Description | |
|--|--|--|
| :id | The ID of the OAuth Client to show | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "oauth-clients") | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | OAuth Client not found, or user unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/oauth-clients/oc-XKFwG6ggfA9n7t1K
```

Sample Response

```
{
  "data": {
    "id": "oc-XKFwG6ggfA9n7t1K",
    "type": "oauth-clients",
    "attributes": {
      "created-at": "2018-04-16T20:42:53.771Z",
      "callback-url": "https://app.terraform.io/auth/35936d44-842c-4ddd-b4d4-7c741383dc3a/callback",
      "connect-path": "/auth/35936d44-842c-4ddd-b4d4-7c741383dc3a?organization_id=1",
      "service-provider": "github",
      "service-provider-display-name": "GitHub",
      "http-url": "https://github.com",
      "api-url": "https://api.github.com",
      "key": null,
      "rsa-public-key": null
    },
    "relationships": {
      "organization": {
        "data": {
          "id": "my-organization",
          "type": "organizations"
        },
        "links": {
          "related": "/api/v2/organizations/my-organization"
        }
      },
      "oauth-tokens": {
        "data": [],
        "links": {
          "related": "/api/v2/oauth-tokens/oc-XKFwG6ggfA9n7t1K"
        }
      }
    }
  }
}
```

Create an OAuth Client

POST /organizations/:organization_name/oauth-clients

| Parameter | Description |
|--------------------|--|
| :organization_name | The name of the organization that will be connected to the VCS provider. The organization must already exist in the system, and the user must have permissions to initiate the connection. |

This endpoint allows you to create a VCS connection between an organization and a VCS provider (GitHub, Bitbucket, or

GitLab) for use when creating or setting up workspaces. By using this API endpoint, you can provide a pre-generated OAuth token string instead of going through the process of creating a GitHub/GitLab OAuth Application or Bitbucket App Link. To learn how to generate one of these token strings for your VCS provider, you can read the following documentation:

- GitHub and GitHub Enterprise (<https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/>)
- Bitbucket Cloud (<https://confluence.atlassian.com/bitbucket/app-passwords-828781300.html>)
- GitLab, GitLab Community Edition, and GitLab Enterprise Edition
(https://docs.gitlab.com/ce/user/profile/personal_access_tokens.html#creating-a-personal-access-token)

Note: This endpoint does not currently support creation of a Bitbucket Server OAuth Client.

| Status | Response | Reason |
|--|--|--|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "oauth-clients") | OAuth Client successfully created |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found or user unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|------------------------------------|--------|---------|---|
| data.type | string | | Must be "oauth-clients". |
| data.attributes.service-provider | string | | The VCS provider being connected with. Valid options are "github", "github_enterprise", "bitbucket_hosted", "gitlab_hosted", "gitlab_community_edition", or "gitlab_enterprise_edition". |
| data.attributes.http-url | string | | The homepage of your VCS provider (e.g. " https://github.com " or " https://ghe.example.com ") |
| data.attributes.api-url | string | | The base URL of your VCS provider's API (e.g. https://api.github.com or " https://ghe.example.com/api/v3 ") |
| data.attributes.oauth-token-string | string | | The token string you were given by your VCS provider |

Sample Payload

```
{  
  "data": {  
    "type": "oauth-clients",  
    "attributes": {  
      "service-provider": "github",  
      "http-url": "https://github.com",  
      "api-url": "https://api.github.com",  
      "oauth-token-string": "4306823352f0009d0ed81f1b654ac17a"  
    }  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/organizations/my-organization/oauth-clients
```

Sample Response

```
{
  "data": {
    "id": "oc-XKFwG6ggfA9n7t1K",
    "type": "oauth-clients",
    "attributes": {
      "created-at": "2018-04-16T20:42:53.771Z",
      "callback-url": "https://app.terraform.io/auth/35936d44-842c-4ddd-b4d4-7c741383dc3a/callback",
      "connect-path": "/auth/35936d44-842c-4ddd-b4d4-7c741383dc3a?organization_id=1",
      "service-provider": "github",
      "service-provider-display-name": "GitHub",
      "http-url": "https://github.com",
      "api-url": "https://api.github.com",
      "key": null,
      "rsa-public-key": null
    },
    "relationships": {
      "organization": {
        "data": {
          "id": "my-organization",
          "type": "organizations"
        },
        "links": {
          "related": "/api/v2/organizations/my-organization"
        }
      },
      "oauth-tokens": {
        "data": [],
        "links": {
          "related": "/api/v2/oauth-tokens/oc-XKFwG6ggfA9n7t1K"
        }
      }
    }
  }
}
```

Update an OAuth Client

PATCH /oauth-clients/:id

| Parameter | Description |
|-----------|---------------------------------------|
| :id | The ID of the OAuth Client to update. |

When a VCS service provider changes their API URL, use this endpoint to update the value of `data.attributes.api-url`. Use caution when changing other attributes with this endpoint; editing an OAuth client that workspaces are currently using can have unexpected effects.

| Status | Response | Reason |
|--|--|---|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "oauth-clients") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | OAuth Client not found or user unauthorized to perform action |

| Status | Response | Reason |
|--|---|--|
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|----------------------------------|--------|------------------|---|
| data.type | string | | Must be "oauth-clients". |
| data.attributes.service-provider | string | (previous value) | The VCS provider being connected with. Valid options are "github", "github_enterprise", "bitbucket_hosted", "gitlab_hosted", "gitlab_community_edition", or "gitlab_enterprise_edition". |
| data.attributes.http-url | string | (previous value) | The homepage of your VCS provider (e.g. " https://github.com " or " https://ghe.example.com ") |
| data.attributes.api-url | string | (previous value) | The base URL of your VCS provider's API (e.g. " https://api.github.com " or " https://ghe.example.com/api/v3 ") |
| data.attributes.key | string | (previous value) | The OAuth client key. |
| data.attributes.secret | string | (previous value) | The OAuth client secret. |

Sample Payload

```
{
  "data": {
    "id": "oc-XKFwG6ggfA9n7t1K",
    "type": "oauth-clients",
    "attributes": {
      "service-provider": "github",
      "http-url": "https://github.com",
      "api-url": "https://api.github.com",
      "key": "key",
      "secret": "secret"
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/oauth-clients/oc-XKFwG6ggfA9n7t1K
```

Sample Response

```
{
  "data": {
    "id": "oc-XKFwG6ggfA9n7t1K",
    "type": "oauth-clients",
    "attributes": {
      "created-at": "2018-04-16T20:42:53.771Z",
      "callback-url": "https://app.terraform.io/auth/35936d44-842c-4ddd-b4d4-7c741383dc3a/callback",
      "connect-path": "/auth/35936d44-842c-4ddd-b4d4-7c741383dc3a?organization_id=1",
      "service-provider": "github",
      "service-provider-display-name": "GitHub",
      "http-url": "https://github.com",
      "api-url": "https://api.github.com",
      "key": null,
      "rsa-public-key": null
    },
    "relationships": {
      "organization": {
        "data": {
          "id": "my-organization",
          "type": "organizations"
        },
        "links": {
          "related": "/api/v2/organizations/my-organization"
        }
      },
      "oauth-tokens": {
        "data": [],
        "links": {
          "related": "/api/v2/oauth-tokens/oc-XKFwG6ggfA9n7t1K"
        }
      }
    }
  }
}
```

Destroy an OAuth Client

`DELETE /oauth-clients/:id`

| Parameter | Description |
|------------------|---------------------------------------|
| <code>:id</code> | The ID of the OAuth Client to destroy |

This endpoint allows you to remove an existing connection between an organization and a VCS provider (GitHub, Bitbucket, or GitLab).

Note: Removing the OAuth Client will unlink workspaces that use this connection from their repositories, and these workspaces will need to be manually linked to another repository.

| Status | Response | Reason |
|---|---|--|
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Empty response | The OAuth Client was successfully destroyed |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization or OAuth Client not found, or user unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request DELETE \  
  https://app.terraform.io/api/v2/oauth-clients/oc-XKFwG6ggfA9n7t1K
```

OAuth Tokens

Note: These API endpoints are in beta and are subject to change.

The oauth-token object represents a VCS configuration which includes the OAuth connection and the associated OAuth token. This object is used when creating a workspace to identify which VCS connection to use.

List OAuth Tokens

List all the OAuth Tokens for a given OAuth Client

GET /oauth-clients/:oauth_client_id/oauth-tokens

| Parameter | Description | |
|--|---|--|
| :oauth_client_id | The ID of the OAuth Client | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "oauth-tokens") | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | OAuth Client not found, or user unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  https://app.terraform.io/api/v2/oauth-clients/oc-GhHqb5rkeK19mLB8/oauth-tokens
```

Sample Response

```
{
  "data": [
    {
      "id": "ot-hmAyP66qk2AMVdbJ",
      "type": "oauth-tokens",
      "attributes": {
        "created-at": "2017-11-02T06:37:49.284Z",
        "service-provider-user": "skierkowski",
        "has-ssh-key": false
      },
      "relationships": {
        "oauth-client": {
          "data": {
            "id": "oc-GhHqb5rkeK19mLB8",
            "type": "oauth-clients"
          },
          "links": {
            "related": "/api/v2/oauth-clients/oc-GhHqb5rkeK19mLB8"
          }
        },
        "links": {
          "self": "/api/v2/oauth-tokens/ot-hmAyP66qk2AMVdbJ"
        }
      }
    }
  ]
}
```

Show an OAuth Token

GET /oauth-tokens/:id

| Parameter | Description | |
|--|---|---|
| :id | The ID of the OAuth token to show | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "oauth-tokens") | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | OAuth Token not found, or user unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/oauth-tokens/ot-29t7xkUKiNC2XasL
```

Sample Response

```
{  
  "data": {  
    "id": "ot-29t7xkUKiNC2XasL",  
    "type": "oauth-tokens",  
    "attributes": {  
      "created-at": "2018-08-29T14:07:22.144Z",  
      "service-provider-user": "EM26Jj0ikRsIFFh3fE5C",  
      "has-ssh-key": false  
    },  
    "relationships": {  
      "oauth-client": {  
        "data": {  
          "id": "oc-WMipGbuW8q7xCrmJ",  
          "type": "oauth-clients"  
        },  
        "links": {  
          "related": "/api/v2/oauth-clients/oc-WMipGbuW8q7xCrmJ"  
        }  
      }  
    },  
    "links": {  
      "self": "/api/v2/oauth-tokens/ot-29t7xkUKiNC2XasL"  
    }  
  }  
}
```

Update an OAuth Token

PATCH /oauth-tokens/:id

| Parameter | Description | |
|--|---|--|
| :id | The ID of the OAuth token to update | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "oauth-tokens") | OAuth Token successfully updated |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | OAuth Token not found or user unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-------------------------|--------|---------|------------------------------|
| data.type | string | | Must be "oauth-tokens". |
| data.attributes.ssh-key | string | | Optional. The SSH key |

Sample Payload

```
{
  "data": {
    "id": "ot-29t7xkUKiNC2XasL",
    "type": "oauth-tokens",
    "attributes": {
      "ssh-key": "..."
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/oauth-tokens/ot-29t7xkUKiNC2XasL
```

Sample Response

```
{
  "data": {
    "id": "ot-29t7xkUKiNC2XasL",
    "type": "oauth-tokens",
    "attributes": {
      "created-at": "2018-08-29T14:07:22.144Z",
      "service-provider-user": "EM26Jj0ikRsIFFh3fE5C",
      "has-ssh-key": false
    },
    "relationships": {
      "oauth-client": {
        "data": {
          "id": "oc-WMipGbuW8q7xCrmJ",
          "type": "oauth-clients"
        },
        "links": {
          "related": "/api/v2/oauth-clients/oc-WMipGbuW8q7xCrmJ"
        }
      }
    },
    "links": {
      "self": "/api/v2/oauth-tokens/ot-29t7xkUKiNC2XasL"
    }
  }
}
```

Destroy an OAuth Token

`DELETE /oauth-tokens/:id`

| Parameter | Description | |
|---|---|---|
| <code>:id</code> | The ID of the OAuth Token to destroy | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Empty response | The OAuth Token was successfully destroyed |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | OAuth Token not found, or user unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
https://app.terraform.io/api/v2/oauth-tokens/ot-29t7xkUKiNC2XasL
```

Organization Token API

Note: These API endpoints are in beta and are subject to change.

Generate a new organization token

POST /organizations/:organization_name/authentication-token

| Parameter | Description | |
|--------------------|---|--|
| :organization_name | The name of the organization to generate a token for. | |

Generates a new organization token, replacing any existing token. This token can be used to act as the organization service account (/docs/enterprise/users-teams-organizations/service-accounts.html).

Only members of the owners team, the owners team service account (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts), and the organization service account (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts) can use this endpoint.

| Status | Response | Reason |
|--|--|---------------------|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "authentication-tokens") | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not authorized |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  https://app.terraform.io/api/v2/organizations/my-organization/authentication-token
```

Sample Response

```
{
  "data": {
    "id": "4111756",
    "type": "authentication-tokens",
    "attributes": {
      "created-at": "2017-11-29T19:11:28.075Z",
      "last-used-at": null,
      "description": null,
      "token": "ZgqYdzuvlv8Iyg.atlasv1.6nV7t10yFls341jo1xdZTP72fN0uu9VL55ozqzekfmToGFbhoFvvygIRy2mwVAXom0E"
    },
    "relationships": {
      "created-by": {
        "data": {
          "id": "user-62goNpx1ThQf689e",
          "type": "users"
        }
      }
    }
  }
}
```

Delete the organization token

`DELETE /organizations/:organization/authentication-token`

| Parameter | Description |
|---------------------------------|---|
| <code>:organization_name</code> | Which organization's token should be deleted. |

Only members of the owners team, the owners team service account, and the organization service account can use this endpoint.

| Status | Response | Reason |
|---|---|---------------------|
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | Success |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not authorized |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
https://app.terraform.io/api/v2/organizations/my-organization/authentication-token
```

Organizations API

Note: These API endpoints are in beta and are subject to change.

The Organizations API is used to list, show, create, update, and destroy organizations.

List Organizations

GET /organizations

| Status | Response | Reason |
|--|--|---|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "organizations") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found or user unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request GET \  
  https://app.terraform.io/api/v2/organizations
```

Sample Response

```
{
  "data": [
    {
      "id": "hashicorp",
      "type": "organizations",
      "attributes": {
        "name": "hashicorp",
        "created-at": "2017-09-07T14:34:40.492Z",
        "email": "user@example.com",
        "session-timeout": null,
        "session-remember": null,
        "collaborator-auth-policy": "password",
        "enterprise-plan": "pro",
        "permissions": {
          "can-update": true,
          "can-destroy": true,
          "can-create-team": true,
          "can-create-workspace": true,
          "can-update-oauth": true,
          "can-update-api-token": true,
          "can-update-sentinel": true,
          "can-traverse": true,
          "can-create-workspace-migration": true
        }
      },
      "links": {
        "self": "/api/v2/organizations/hashicorp"
      }
    }
  ]
}
```

Show an Organization

GET /organizations/:organization_name

| Parameter | Description | |
|--|--|---|
| :organization_name | The name of the organization to show | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "organizations") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found or user unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/organizations/hashicorp
```

Sample Response

```
{
  "data": {
    "id": "hashicorp",
    "type": "organizations",
    "attributes": {
      "name": "hashicorp",
      "created-at": "2017-09-07T14:34:40.492Z",
      "email": "user@example.com",
      "session-timeout": null,
      "session-remember": null,
      "collaborator-auth-policy": "password",
      "enterprise-plan": "pro",
      "permissions": {
        "can-update": true,
        "can-destroy": true,
        "can-create-team": true,
        "can-create-workspace": true,
        "can-update-oauth": true,
        "can-update-api-token": true,
        "can-update-sentinel": true,
        "can-traverse": true,
        "can-create-workspace-migration": true
      }
    },
    "links": {
      "self": "/api/v2/organizations/hashicorp"
    }
  }
}
```

Create an Organization

POST /organizations

| Status | Response | Reason |
|--|--|---|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "organizations") | The organization was successfully created |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found or user unauthorized to perform action |

| Status | Response | Reason |
|--|---|--|
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-----------------------|--------|---------|--------------------------|
| data.type | string | | Must be "organizations" |
| data.attributes.name | string | | Name of the organization |
| data.attributes.email | string | | Admin email address |

Sample Payload

```
{
  "data": {
    "type": "organizations",
    "attributes": {
      "name": "hashicorp",
      "email": "user@example.com"
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/organizations
```

Sample Response

```
{
  "data": {
    "id": "hashicorp",
    "type": "organizations",
    "attributes": {
      "name": "hashicorp",
      "created-at": "2017-09-07T14:34:40.492Z",
      "email": "user@example.com",
      "session-timeout": null,
      "session-remember": null,
      "collaborator-auth-policy": "password",
      "enterprise-plan": "pro",
      "permissions": {
        "can-update": true,
        "can-destroy": true,
        "can-create-team": true,
        "can-create-workspace": true,
        "can-update-oauth": true,
        "can-update-api-token": true,
        "can-update-sentinel": true,
        "can-traverse": true,
        "can-create-workspace-migration": true
      }
    },
    "links": {
      "self": "/api/v2/organizations/hashicorp"
    }
  }
}
```

Update an Organization

PATCH /organizations/:organization_name

| Parameter | Description | |
|--|--|--|
| :organization_name | The name of the organization to update | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "organizations") | The organization was successfully updated |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found or user unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|--|---------|----------|---|
| data.type | string | | Must be "organizations" |
| data.attributes.name | string | | Name of the organization |
| data.attributes.email | string | | Admin email address |
| data.attributes.session-timeout | integer | 20160 | Session timeout after inactivity (minutes) |
| data.attributes.session-remember | integer | 20160 | Session expiration (minutes) |
| data.attributes.collaborator-auth-policy | string | password | Authentication policy (password or two_factor_mandatory) |
| data.attributes.owners-team-saml-role-id | string | owners | SAML only The name of the "owners" team
(/docs/enterprise/saml/team-membership.html#managing-membership-of-the-owners-team) |

Sample Payload

```
{
  "data": {
    "type": "organizations",
    "attributes": {
      "email": "admin@example.com"
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/organizations/hashicorp
```

Sample Response

```
{
  "data": {
    "id": "hashicorp",
    "type": "organizations",
    "attributes": {
      "name": "hashicorp",
      "created-at": "2017-09-07T14:34:40.492Z",
      "email": "admin@example.com",
      "session-timeout": null,
      "session-remember": null,
      "collaborator-auth-policy": "password",
      "enterprise-plan": "pro",
      "permissions": {
        "can-update": true,
        "can-destroy": true,
        "can-create-team": true,
        "can-create-workspace": true,
        "can-update-oauth": true,
        "can-update-api-token": true,
        "can-update-sentinel": true,
        "can-traverse": true,
        "can-create-workspace-migration": true
      }
    },
    "links": {
      "self": "/api/v2/organizations/hashicorp"
    }
  }
}
```

Destroy an Organization

`DELETE /organizations/:organization_name`

| Parameter | Description | |
|---|---|---|
| <code>:organization_name</code> | The name of the organization to destroy | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | | The organization was successfully destroyed |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found or user unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
https://app.terraform.io/api/v2/organizations/hashicorp
```

Sample Response

The response body will be empty if successful.

Plans API

Note: These API endpoints are in beta and are subject to change.

A plan represents the execution plan of a Run in a Terraform workspace.

Show a plan

GET /plans/:id

| Parameter | Description |
|-----------|-----------------------------|
| id | The ID of the plan to show. |

There is no endpoint to list plans. You can find the ID for a plan in the `relationships.plan` property of a run object.

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "plans") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Plan not found, or user unauthorized to perform action |

Sample Request

```
curl \  
--header "Authorization: Bearer $TOKEN" \  
--header "Content-Type: application/vnd.api+json" \  
https://app.terraform.io/api/v2/plans/plan-8F5JFydVYAmtTjET
```

Sample Response

```
{  
  "data": {  
    "id": "plan-8F5JFydVYAmTjET",  
    "type": "plans",  
    "attributes": {  
      "has-changes": true,  
      "resource-additions": 0,  
      "resource-changes": 1,  
      "resource-destructions": 0,  
      "status": "finished",  
      "status-timestamps": {  
        "queued-at": "2018-07-02T22:29:53+00:00",  
        "pending-at": "2018-07-02T22:29:53+00:00",  
        "started-at": "2018-07-02T22:29:54+00:00",  
        "finished-at": "2018-07-02T22:29:58+00:00"  
      },  
      "log-read-url": "https://archivist.terraform.io/v1/object/dmF1bHQ6djE60FA1eEdlSFVHRSs4YUcwaW83a1dRR  
DA0U2E3T3FiWk1HM2NyQlNtcS9JS1hHN3dmTXJmaFhEYtlHdTF1ZlgxZ2wzVC9kVtNcjRPOEJkK050VFI3U3dvS2ZuaUhFSGpVenJVUF  
YzSFVZQ1VZYno3T3UyYjdDRVPRRE5pbWJDVTIrNllQTENyTndYd1Y0ak1DL1dPVln1VlnxKzYzbWlIcnJPa2dRRkJZZGtFeTNiaU84YlZ  
4QWs2QzLLY3VJb3lmWlIrajF4a1hYZTlsWnFYemRkL2pNOG9Zc0ZDakdVMCtURUE3dDNMODRsRnY4cWl1dUN5dUVuUzdnZzFwL3BNeHlw  
bXNXZWrrUDhXdhGNnF4c3dqaxlzs29oL3FKakI5dm9uYU5KzAybnloREdnQ3J2Rk5WMLBjemZQTg"  
    },  
    "relationships": {  
      "state-versions": {  
        "data": []  
      }  
    },  
    "links": {  
      "self": "/api/v2/plans/plan-8F5JFydVYAmTjET"  
    }  
  }  
}
```

Policies API

Note: These API endpoints are in beta and are subject to change.

Sentinel Policy as Code (/docs/enterprise/sentinel/index.html) is an embedded policy as code framework integrated with Terraform Enterprise.

Policies are configured on a per-organization level and are organized and grouped into policy sets (/docs/enterprise/sentinel/manage-policies.html#organizing-policies-with-policy-sets), which define the workspaces on which policies are enforced during runs. In these workspaces, the plan's changes are validated against the relevant policies after the plan step. (For details, see Run States and Stages (/docs/enterprise/run/states.html).)

This page documents the API endpoints to create, read, update, and delete the Sentinel policies in an organization. To view and manage the results of a specific run's policy check, use the Runs API (/docs/enterprise/api/run.html).

Create a Policy

POST /organizations/:organization_name/policies

| Parameter | Description |
|--------------------|--|
| :organization_name | The organization to create the policy in. The organization must already exist in the system, and the token authenticating the API request must belong to the "owners" team or a member of the "owners" team. |

This creates a new policy object for the organization, but does not upload the actual policy code. After creation, you must use the Upload a Policy endpoint (below) with the new policy's upload path. (This endpoint's response body includes the upload path in its `links.upload` property.)

| Status | Response | Reason |
|--|---|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "policies") | Successfully created a policy |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found, or user unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|---------------------------------------|---------------|----------------|--|
| data.type | string | | Must be "policies". |
| data.attributes.name | string | | The name of the policy, which cannot be modified after creation. Can include letters, numbers, -, and _. |
| data.attributes.enforce | array[object] | | An array of enforcement configurations which map Sentinel file paths to their enforcement modes. Currently policies only support a single file, so this array will consist of a single element. If the path in the enforcement map does not match the Sentinel policy (<NAME>.sentinel), then the default hard-mandatory will be used. |
| data.attributes.enforce[].path | string | | Must be <NAME>.sentinel, where <NAME> has the same value as data.attributes.name. |
| data.attributes.enforce[].mode | string | hard-mandatory | The enforcement level of the policy. Valid values are "hard-mandatory", "soft-mandatory", and "advisory". For more details, see Managing Policies (/docs/enterprise/sentinel/manage-policies.html). |
| data.relationships.policy-sets.data[] | array[object] | [] | A list of resource identifier objects to define which policy sets the new policy will be a member of. These objects must contain id and type properties, and the type property must be policy-sets (e.g. { "id": "polset-3yVQZvHzf5j3WRJ1", "type": "policy-sets" }). |

Sample Payload

```
{
  "data": {
    "attributes": {
      "enforce": [
        {
          "path": "my-example-policy.sentinel",
          "mode": "hard-mandatory"
        }
      ],
      "name": "my-example-policy"
    },
    "relationships": {
      "policy-sets": {
        "data": [
          { "id": "polset-3yVQZvHzf5j3WRJ1", "type": "policy-sets" }
        ]
      }
    },
    "type": "policies"
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/organizations/my-organization/policies
```

Sample Response

```
{
  "data": {
    "id": "pol-u3S5p2Uwk21keu1s",
    "type": "policies",
    "attributes": {
      "name": "my-example-policy",
      "enforce": [
        {
          "path": "my-example-policy.sentinel",
          "mode": "advisory"
        }
      ],
      "policy-set-count": 1,
      "updated-at": null
    },
    "relationships": {
      "organization": {
        "data": { "id": "my-organization", "type": "organizations" }
      },
      "policy-sets": {
        "data": [
          { "id": "polset-3yVQZvHzf5j3WRJ1", "type": "policy-sets" }
        ]
      }
    },
    "links": {
      "self": "/api/v2/policies/pol-u3S5p2Uwk21keu1s",
      "upload": "/api/v2/policies/pol-u3S5p2Uwk21keu1s/upload"
    }
  }
}
```

Show a Policy

GET /policies/:policy_id

| Parameter | Description | |
|--|---|----------------------------|
| :policy_id | The ID of the policy to show. Use the "List Policies" endpoint to find IDs. | |
| Status | Response | Reason |
| 200
(Status/200">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "policies") | The request was successful |

| Status | Response | Reason |
|--|---|---|
| 404
(Status/404">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy not found or user unauthorized to perform action |

Sample Request

```
curl --request GET \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/policies/pol-oXUppaX2ximkqp8w
```

Sample Response

```
{
  "data": {
    "id": "pol-oXUppaX2ximkqp8w",
    "type": "policies",
    "attributes": {
      "name": "my-example-policy",
      "enforce": [
        {
          "path": "my-example-policy.sentinel",
          "mode": "soft-mandatory"
        }
      ],
      "policy-set-count": 1,
      "updated-at": "2018-09-11T18:21:21.784Z"
    },
    "relationships": {
      "organization": {
        "data": { "id": "my-organization", "type": "organizations" }
      },
      "policy-sets": {
        "data": [
          { "id": "polset-3yVQZvHzf5j3WRJ1", "type": "policy-sets" }
        ]
      }
    },
    "links": {
      "self": "/api/v2/policies/pol-oXUppaX2ximkqp8w",
      "upload": "/api/v2/policies/pol-oXUppaX2ximkqp8w/upload",
      "download": "/api/v2/policies/pol-oXUppaX2ximkqp8w/download"
    }
  }
}
```

Upload a Policy

PUT /policies/:policy_id/upload

| Parameter | Description |
|------------|--|
| :policy_id | The ID of the policy to upload code to. Use the "List Policies" endpoint (or the response to a "Create Policy" request) to find IDs. |

This endpoint uploads code to an existing Sentinel policy.

Note: This endpoint does not use JSON-API's conventions for HTTP headers and body serialization.

Note: This endpoint limits the size of uploaded policies to 10MB. If a larger payload is uploaded, an HTTP 413 error will be returned, and the policy will not be saved. Consider refactoring policies into multiple smaller, more concise documents if you reach this limit.

Request Body

This PUT endpoint requires the text of a valid Sentinel policy, with a Content-Type of application/octet-stream.

See Defining Policies (/docs/enterprise/sentinel/import/index.html) for details about writing Sentinel code.

Sample Payload

```
main = rule { true }
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/octet-stream" \  
  --request PUT \  
  --data-binary @payload.sentinel \  
  https://app.terraform.io/api/v2/policies/pol-u3S5p2Uwk21keu1s/upload
```

Update a Policy

PATCH /policies/:policy_id

| Parameter | Description |
|------------|---|
| :policy_id | The ID of the policy to update. Use the "List Policies" endpoint to find IDs. |

This endpoint can update the enforcement mode of an existing policy. To update the policy code itself, use the upload endpoint.

| Status | Response | Reason |
|--|---|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "policies") | Successfully updated the policy |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy not found, or user unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------------------|---------------|----------------|---|
| data.type | string | | Must be "policies". |
| data.attributes.name | string | (Current name) | Ignored if present. |
| data.attributes.enforce | array[object] | | An array of enforcement configurations which map Sentinel file paths to their enforcement modes. Currently policies only support a single file, so this array will consist of a single element. The value provided replaces the enforcement map. To make an incremental update, you can first fetch the current value of this map from the show endpoint and modify it. If the path in the enforcement map does not match the Sentinel policy (<NAME>.sentinel), then the default hard-mandatory will be used. |
| data.attributes.enforce[].path | string | | Must be <NAME>.sentinel, where <NAME> matches the original value of data.attributes.name. |
| data.attributes.enforce[].mode | string | hard-mandatory | The enforcement level of the policy. Valid values are "hard-mandatory", "soft-mandatory", and "advisory". For more details, see Managing Policies (/docs/enterprise/sentinel/manage-policies.html). |

Sample Payload

```
{  
  "data": {  
    "attributes": {  
      "enforce": [  
        {  
          "path": "my-example-policy.sentinel",  
          "mode": "soft-mandatory"  
        }  
      ],  
      "type": "policies"  
    }  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request PATCH \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/policies/pol-u3S5p2Uwk21keu1s
```

Sample Response

```
{  
  "data": {  
    "id": "pol-u3S5p2Uwk21keu1s",  
    "type": "policies",  
    "attributes": {  
      "name": "my-example-policy",  
      "enforce": [  
        {  
          "path": "my-example-policy.sentinel",  
          "mode": "soft-mandatory"  
        }  
      ],  
      "policy-set-count": 0,  
      "updated-at": "2017-10-10T20:58:04.621Z"  
    },  
    "relationships": {  
      "organization": {  
        "data": { "id": "my-organization", "type": "organizations" }  
      },  
    },  
    "links": {  
      "self": "/api/v2/policies/pol-u3S5p2Uwk21keu1s",  
      "upload": "/api/v2/policies/pol-u3S5p2Uwk21keu1s/upload",  
      "download": "/api/v2/policies/pol-u3S5p2Uwk21keu1s/download"  
    }  
  }  
}
```

List Policies

GET /organizations/:organization_name/policies

| Parameter | Description | |
|--|---|--|
| :organization_name | The organization to list policies for. | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | Array of JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents)s (type: "policies") | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found, or user unauthorized to perform action |

Query Parameters

This endpoint supports pagination with standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|--------------|---|
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 policies per page. |
| search[name] | Optional. Allows searching the organization's policies by name. |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  https://app.terraform.io/api/v2/organizations/my-organization/policies
```

Sample Response

```
{
  "data": [
    {
      "attributes": {
        "enforce": [
          {
            "mode": "advisory",
            "path": "my-example-policy.sentinel"
          }
        ],
        "name": "my-example-policy",
        "policy-set-count": 0,
        "updated-at": "2017-10-10T20:52:13.898Z"
      },
      "id": "pol-u3S5p2Uwk21keu1s",
      "relationships": {
        "organization": {
          "data": { "id": "my-organization", "type": "organizations" }
        },
        "links": {
          "download": "/api/v2/policies/pol-u3S5p2Uwk21keu1s/download",
          "self": "/api/v2/policies/pol-u3S5p2Uwk21keu1s",
          "upload": "/api/v2/policies/pol-u3S5p2Uwk21keu1s/upload"
        },
        "type": "policies"
      }
    }
  ]
}
```

Delete a Policy

`DELETE /policies/:policy_id`

| Parameter | Description | |
|---|---|--|
| <code>:policy_id</code> | The ID of the policy to delete. Use the "List Policies" endpoint to find IDs. | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | Successfully deleted the policy |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy not found, or user unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--request DELETE \
https://app.terraform.io/api/v2/policies/pl-u3S5p2Uwk21keu1s
```

Available Related Resources

The GET endpoints above can optionally return related resources, if requested with the `include` query parameter ([/docs/enterprise/api/index.html#inclusion-of-related-resources](#)). The following resource types are available:

- `policy-sets` - Policy sets that any returned policies are members of.

Policy Checks API

Note: These API endpoints are in beta and are subject to change.

List policy checks

This endpoint lists the policy checks in a run.

| Method | Path |
|--------|-----------------------------|
| GET | /runs/:run_id/policy-checks |

Parameters

- `run_id` (string: <required>) - specifies the run ID for which to list policy checks

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  https://app.terraform.io/api/v2/runs/run-CZcmD7eaghjhyXavN/policy-checks
```

Sample Response

```
{  
  "data": [  
    {  
      "id": "polchk-9VYRc9bpfJEsnwum",  
      "type": "policy-checks",  
      "attributes": {  
        "result": {  
          "result": false,  
          "passed": 0,  
          "total-failed": 1,  
          "hard-failed": 0,  
          "soft-failed": 1,  
          "advisory-failed": 0,  
          "duration-ms": 0,  
          "sentinel": {  
            "can-override": true,  
            "error": null,  
            "policies": [  
              {  
                "allowed-failure": false,  
                "error": null,  
                "policy": "contains-billing-tag.sentinel",  
                "result": false,  
                "trace": {  
                  "description": "",  
                  "-----"  
                }  
              }  
            ]  
          }  
        }  
      }  
    }  
  ]  
}
```

```

    "error": null,
    "print": "",
    "result": false,
    "rules": {
        "main": {
            "ident": "main",
            "root": {
                "children": [
                    {
                        "children": null,
                        "expression": "r.applied contains \"tags\"",
                        "value": "false"
                    }
                ],
                "expression": "all tfplan.resources.aws_instance as _, instances {\n\tall instances as _, r {\n\t\ttr.applied contains \"tags\" and r.applied.tags contains \"billing-id\"\n\t}\n}\n",
                "value": "false"
            },
            "string": "Rule \"main\" (byte offset 18) = false\n  false (offset 120): r.applied contains \"tags\"\n"
        }
    }
},
"scope": "organization",
"status": "soft_failed",
"status-timestamps": {
    "queued-at": "2017-11-29T20:02:17+00:00",
    "soft-failed-at": "2017-11-29T20:02:20+00:00"
},
"actions": {
    "is-overridable": true
},
"permissions": {
    "can-override": false
},
"links": {
    "output": "/api/v2/policy-checks/polchk-9VYRc9bpfJEsnwum/output"
}
}
]
}

```

Override Policy

This endpoint overrides a soft-mandatory or warning policy.

| Method | Path |
|--------|--|
| POST | /policy-checks/:policy_check_id/actions/override |

Parameters

- `policy_check_id(string: <required>)` - specifies the ID for the policy check to override

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  https://app.terraform.io/api/v2/policy-checks/polchk-EasPB4Srx5NAiWAU/actions/override
```

Sample Response

```
{  
  "data": {  
    "id": "polchk-EasPB4Srx5NAiWAU",  
    "type": "policy-checks",  
    "attributes": {  
      "result": {  
        "result": false,  
        "passed": 0,  
        "total-failed": 1,  
        "hard-failed": 0,  
        "soft-failed": 1,  
        "advisory-failed": 0,  
        "duration-ms": 0,  
        "sentinel": {  
          "can-override": true,  
          "error": null,  
          "policies": [  
            {  
              "allowed-failure": false,  
              "error": null,  
              "policy": "contains-billing-tag.sentinel",  
              "result": false,  
              "trace": {  
                "description": "",  
                "error": null,  
                "print": "",  
                "result": false,  
                "rules": {  
                  "main": {  
                    "ident": "main",  
                    "root": {  
                      "children": [  
                        {  
                          "children": null,  
                          "expression": "r.applied contains \"tags\"",  
                          "value": "false"  
                        }  
                      ],  
                      "expression": "all tfplan.resources.aws_instance as _, instances {\n\tall instances  
as _, r {\n\t\ttr.applied contains \"tags\" and r.applied.tags contains \"billing-id\"\n\t}\n}\n",  
                      "value": "false"  
                    },  
                    "string": "Rule \"main\" (byte offset 18) = false\n  false (offset 120): r.applied co  
ntains \"tags\"\n"                }  
              }  
            }  
          ]  
        }  
      }  
    }  
  }
```

```
        }
      ],
      "result": false
    }
  },
  "scope": "organization",
  "status": "overridden",
  "status-timestamps": {
    "queued-at": "2017-11-29T20:13:37+00:00",
    "soft-failed-at": "2017-11-29T20:13:40+00:00",
    "overridden-at": "2017-11-29T20:14:11+00:00"
  },
  "actions": {
    "is-overridable": true
  },
  "permissions": {
    "can-override": false
  }
},
"links": {
  "output": "/api/v2/policy-checks/polchk-EasPB4Srx5NAiWAU/output"
}
}
}
```

Policy Sets API

Note: These API endpoints are in beta and are subject to change.

Sentinel Policy as Code (/docs/enterprise/sentinel/index.html) is an embedded policy as code framework integrated with Terraform Enterprise.

Policy sets are groups of policies that are applied together to related workspaces. By using policy sets, you can group your policies by attributes such as environment or region. Individual policies that are members of policy sets will only be checked for workspaces that the policy set is attached to. In order for a policy to be active and checked during runs, it must be a member of at least one policy set that is attached to workspaces.

This page documents the API endpoints to create, read, update, and delete policy sets in an organization. To view and manage policies, use the Policies API (/docs/enterprise/api/policies.html).

Create a Policy Set

POST /organizations/:organization_name/policy-sets

| Parameter | Description | |
|--|--|--|
| :organization_name | The organization to create the policy set in. The organization must already exist in the system, and the token authenticating the API request must belong to the "owners" team or a member of the "owners" team. | |
| Status | Response | Reason |
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "policy-sets") | Successfully created a policy set |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found, or user unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-----------|--------|---------|------------------------|
| data.type | string | | Must be "policy-sets". |

| Key path | Type | Default | Description |
|--------------------------------------|------------------|---------|---|
| data.attributes.name | string | | The name of the policy set. Can include letters, numbers, -, and _. |
| data.attributes.description | string | null | A description of the set's purpose. This field supports Markdown and will be rendered in the Terraform Enterprise UI. |
| data.attributes.global | boolean | false | Whether or not this policies in this set should be checked on all of the organization's workspaces, or only on workspaces the policy set is attached to. |
| data.relationships.policies.data[] | array[object] [] | | A list of resource identifier objects that defines which policies will be members of the new set. These objects must contain id and type properties, and the type property must be policies (e.g. { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" }). |
| data.relationships.workspaces.data[] | array[object] [] | | A list of resource identifier objects that defines which workspaces the new set will be attached to. These objects must contain id and type properties, and the type property must be workspaces (e.g. { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }). Obtain workspace IDs from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace (/docs/enterprise/api/workspaces.html#show-workspace) endpoint. |

Sample Payload

```
{
  "data": {
    "attributes": {
      "name": "production",
      "description": "This set contains policies that should be checked on all production infrastructure workspaces.",
      "global": false
    },
    "relationships": {
      "policies": {
        "data": [
          { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" }
        ]
      },
      "workspaces": {
        "data": [
          { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }
        ]
      }
    },
    "type": "policy-sets"
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/organizations/my-organization/policy-sets
```

Sample Response

```
{
  "data": {
    "id": "polset-3yVQZvHzf5j3WRJ1",
    "type": "policy-sets",
    "attributes": {
      "name": "production",
      "description": "This set contains policies that should be checked on all production infrastructure workspaces.",
      "global": false,
      "policy-count": 1,
      "workspace-count": 1,
      "created-at": "2018-09-11T18:21:21.784Z",
      "updated-at": "2018-09-11T18:21:21.784Z",
    },
    "relationships": {
      "organization": {
        "data": { "id": "my-organization", "type": "organizations" }
      },
      "policies": {
        "data": [
          { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" }
        ]
      },
      "workspaces": {
        "data": [
          { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }
        ]
      }
    },
    "links": {
      "self": "/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1"
    }
  }
}
```

List policy sets

GET /organizations/:organization_name/policy-sets

| Parameter | Description |
|--------------------|---|
| :organization_name | The organization to list policy sets for. |

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "policy-sets") | Request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found, or user unauthorized to perform action |

Query Parameters

This endpoint supports pagination with standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|--------------|--|
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 policy sets per page. |
| search[name] | Optional. Allows searching the organization's policy sets by name. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
https://app.terraform.io/api/v2/organizations/my-organization/policy-sets
```

Sample Response

```
{
  "data": [
    {
      "id": "polset-3yVQZvHzf5j3WRJ1",
      "type": "policy-sets",
      "attributes": {
        "name": "production",
        "description": "This set contains policies that should be checked on all production infrastructure workspaces.",
        "global": false,
        "policy-count": 1,
        "workspace-count": 1,
        "created-at": "2018-09-11T18:21:21.784Z",
        "updated-at": "2018-09-11T18:21:21.784Z",
      },
      "relationships": {
        "organization": {
          "data": { "id": "my-organization", "type": "organizations" }
        },
        "policies": {
          "data": [
            { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" }
          ]
        },
        "workspaces": {
          "data": [
            { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }
          ]
        }
      },
      "links": {
        "self": "/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1"
      }
    },
  ],
}
}
```

Show a Policy Set

GET /policy-sets/:id

| Parameter | Description |
|-----------|--|
| :id | The ID of the policy set to show. Use the "List Policy Sets" endpoint to find IDs. |

| Status | Response | Reason |
|--|--|---|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "policy-sets") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy set not found or user unauthorized to perform action |

Sample Request

```
curl --request GET \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1
```

Sample Response

```
{
  "data": {
    "id": "polset-3yVQZvHzf5j3WRJ1",
    "type": "policy-sets",
    "attributes": {
      "name": "production",
      "description": "This set contains policies that should be checked on all production infrastructure workspaces.",
      "global": false,
      "policy-count": 1,
      "workspace-count": 1,
      "created-at": "2018-09-11T18:21:21.784Z",
      "updated-at": "2018-09-11T18:21:21.784Z",
    },
    "relationships": {
      "organization": {
        "data": { "id": "my-organization", "type": "organizations" }
      },
      "policies": {
        "data": [
          { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" }
        ]
      },
      "workspaces": {
        "data": [
          { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }
        ]
      }
    },
    "links": {
      "self": "/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1"
    }
  }
}
```

Update a Policy Set

PATCH /policy-sets/:id

| Parameter | Description |
|-----------|--|
| :id | The ID of the policy set to update. Use the "List Policy Sets" endpoint to find IDs. |

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "policy-sets") | The request was successful |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy set not found or user unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This PATCH endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-----------------------------|---------|------------------|---|
| data.type | string | | Must be "policy-sets". |
| data.attributes.name | string | (previous value) | The name of the policy set. Can include letters, numbers, -, and _. |
| data.attributes.description | string | (previous value) | A description of the set's purpose. This field supports Markdown and will be rendered in the Terraform Enterprise UI. |
| data.attributes.global | boolean | (previous value) | Whether or not this policies in this set should be checked on all of the organization's workspaces, or only on workspaces directly attached to the set. |

Sample Payload

```
{
  "data": {
    "attributes": {
      "name": "a-global-set",
      "description": "**WARNING:** Any policies added to this set will be checked in _all_ workspaces!",
      "global": true
    },
    "type": "policy-sets"
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1
```

Sample Response

```
{
  "data": {
    "id": "polset-3yVQZvHzf5j3WRJ1",
    "type": "policy-sets",
    "attributes": {
      "name": "a-global-set",
      "description": "**WARNING:** Any policies added to this set will be checked in _all_ workspaces!",
      "global": true,
      "policy-count": 1,
      "workspace-count": 4,
      "created-at": "2018-09-11T18:21:21.784Z",
      "updated-at": "2018-09-11T18:21:21.784Z",
    },
    "relationships": {
      "organization": {
        "data": { "id": "my-organization", "type": "organizations" }
      },
      "policies": {
        "data": [
          { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" }
        ]
      },
      "workspaces": {
        "data": [
          { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" },
          { "id": "ws-UZuU7aTTjch2TG19", "type": "workspaces" },
          { "id": "ws-sVHFvWAF2wRkGzD7", "type": "workspaces" },
          { "id": "ws-nt3Jm4hSFtuHF5fi", "type": "workspaces" }
        ]
      }
    },
    "links": {
      "self": "/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1"
    }
  }
}
```

Add Policies to the Policy Set

POST /policy-sets/:id/relationships/policies

| Parameter | Description |
|-----------|---|
| :id | The ID of the policy set to add policies to. Use the "List Policy Sets" endpoint to find IDs. |

| Status | Response | Reason |
|---|---|--|
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | The request was successful |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy set not found or user unauthorized to perform action |
| 422 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (one or more policies not found, wrong types, etc.) |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|----------|---------------|---------|--|
| data[] | array[object] | | A list of resource identifier objects that defines which policies will be added to the set. These objects must contain id and type properties, and the type property must be policies (e.g. { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" }). |

Sample Payload

```
{
  "data": [
    { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" },
    { "id": "pol-2HRvNs49EWPjDqT1", "type": "policies" }
  ]
}
```

Sample Request

```
curl \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1/relationships/policies
```

Attach a Policy Set to workspaces

POST /policy-sets/:id/relationships/workspaces

| Parameter | Description | |
|---|---|--|
| :id | The ID of the policy set to attach to workspaces. Use the "List Policy Sets" endpoint to find IDs. | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | The request was successful |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy set not found or user unauthorized to perform action |
| 422 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (one or more workspaces not found, wrong types, etc.) |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key | Type | Default | Description |
|--------|---------------|---------|---|
| data[] | array[object] | | A list of resource identifier objects that defines the workspaces the policy set will be attached to. These objects must contain id and type properties, and the type property must be workspaces (e.g. { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }). |

Sample Payload

```
{
  "data": [
    { "id": "ws-u3S5p2Uwk21keu1s", "type": "workspaces" },
    { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }
  ]
}
```

Sample Request

```
curl \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1/relationships/workspaces
```

Remove Policies from the Policy Set

DELETE /policy-sets/:id/relationships/policies

| Parameter | Description | |
|---|---|---|
| :id | The ID of the policy set to remove policies from. Use the "List Policy Sets" endpoint to find IDs. | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | The request was successful |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy set not found or user unauthorized to perform action |
| 422 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (wrong types, etc.) |

Request Body

This DELETE endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key | Type | Default | Description |
|--------|---------------|---------|--|
| data[] | array[object] | | A list of resource identifier objects that defines which policies will be removed from the set. These objects must contain id and type properties, and the type property must be policies (e.g. { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" }). |

Sample Payload

```
{
  "data": [
    { "id": "pol-u3S5p2Uwk21keu1s", "type": "policies" },
    { "id": "pol-2HRvNs49EWPjDqT1", "type": "policies" }
  ]
}
```

Sample Request

```
curl \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/vnd.api+json" \
--request DELETE \
--data @payload.json \
https://app.terraform.io/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1/relationships/policies
```

Detach the Policy Set from workspaces

DELETE /policy-sets/:id/relationships/workspaces

| Parameter | Description | |
|---|---|---|
| :id | The ID of the policy set to detach from workspaces. Use the "List Policy Sets" endpoint to find IDs. | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | The request was successful |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy set not found or user unauthorized to perform action |
| 422 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (wrong types, etc.) |

Request Body

This DELETE endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key | Type | Default | Description |
|--------|---------------|---------|--|
| data[] | array[object] | | A list of resource identifier objects that defines which workspaces the policy set will be detached from. These objects must contain id and type properties, and the type property must be workspaces (e.g. { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }). Obtain workspace IDs from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace (/docs/enterprise/api/workspaces.html#show-workspace) endpoint. |

Sample Payload

```
{
  "data": [
    { "id": "ws-u3S5p2Uwk21keu1s", "type": "workspaces" },
    { "id": "ws-2HRvNs49EWPjDqT1", "type": "workspaces" }
  ]
}
```

Sample Request

```
curl \  
  -H "Authorization: Bearer $TOKEN" \  
  -H "Content-Type: application/vnd.api+json" \  
  --request DELETE \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1/relationships/workspaces
```

Delete a Policy Set

DELETE /policy-sets/:id

| Parameter | Description | |
|---|---|--|
| :id | The ID of the policy set to delete. Use the "List Policy Sets" endpoint to find IDs. | |
| Status | Response | Reason |
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | Successfully deleted the policy set |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Policy set not found, or user unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --request DELETE \  
  https://app.terraform.io/api/v2/policy-sets/polset-3yVQZvHzf5j3WRJ1
```

Available Related Resources

The GET endpoints above can optionally return related resources, if requested with the `include` query parameter (</docs/enterprise/api/index.html#inclusion-of-related-resources>). The following resource types are available:

- `policies` - Policies that are a member of this set.
- `workspaces` - Workspaces that this set are attached to.

Runs API

Note: These API endpoints are in beta and are subject to change.

Note: Before working with the runs or configuration versions APIs, read the API-driven run workflow ([/docs/enterprise/run/api.html](#)) page, which includes both a full overview of this workflow and a walkthrough of a simple implementation of it.

Performing a run on a new configuration is a multi-step process.

1. Create a configuration version on the workspace ([/docs/enterprise/api/configuration-versions.html#create-a-configuration-version](#)).
2. Upload configuration files to the configuration version ([/docs/enterprise/api/configuration-versions.html#upload-configuration-files](#)).
3. Create a run on the workspace; this is done automatically when a configuration file is uploaded.
4. Create and queue an apply on the run; if auto-apply is not enabled.

Alternatively, you can create a run with a pre-existing configuration version, even one from another workspace. This is useful for promoting known good code from one workspace to another.

Create a Run

POST /runs

A run performs a plan and apply, using a configuration version and the workspace's current variables. You can specify a configuration version when creating a run; if you don't provide one, the run defaults to the workspace's most recently used version. (A configuration version is "used" when it is created or used for a run in this workspace.)

Note: This endpoint cannot be accessed with organization tokens ([/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts](#)). You must access it with a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) or team token ([/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts](#)).

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|----------------------------|------|---------|---|
| data.attributes.is-destroy | bool | false | Specifies if this plan is a destroy plan, which will destroy all provisioned resources. |

| Key path | Type | Default | Description |
|--|---|--|--|
| data.attributes.message | string | "Queued manually via the Terraform Enterprise API" | Specifies the message to be associated with this run. |
| data.relationships.workspace.data.id | string | | Specifies the workspace ID where the run will be executed. |
| data.relationships.configuration-version.data.id | string | (nothing) | Specifies the configuration version to use for this run. If the configuration-version object is omitted, the run will be created using the workspace's latest configuration version. |
| Status | Response | Reason | |
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "runs") | | Successfully created a run |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | | Organization or workspace not found, or user unauthorized to perform action |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | | Malformed request body (missing attributes, wrong types, etc.) |

Sample Payload

```
{
  "data": {
    "attributes": {
      "is-destroy": false,
      "message": "Custom message"
    },
    "type": "runs",
    "relationships": {
      "workspace": {
        "data": {
          "type": "workspaces",
          "id": "ws-LLGHCr4SWy28wyGN"
        }
      },
      "configuration-version": {
        "data": {
          "type": "configuration-versions",
          "id": "cv-n4XQPBa2QnecZJ4G"
        }
      }
    }
  }
}
```

Sample Request

```
curl \  
--header "Authorization: Bearer $TOKEN" \  
--header "Content-Type: application/vnd.api+json" \  
--request POST \  
--data @payload.json \  
https://app.terraform.io/api/v2/runs
```

Sample Response

```
{
  "data": {
    "id": "run-CZcmD7eagjhyX0vN",
    "type": "runs",
    "attributes": {
      "auto-apply": false,
      "error-text": null,
      "is-destroy": false,
      "message": "Custom Message",
      "metadata": {},
      "source": "tfe-ui",
      "status": "pending",
      "status-timestamps": {},
      "terraform-version": "0.10.8",
      "created-at": "2017-11-29T19:56:15.205Z",
      "has-changes": false,
      "actions": {
        "is-cancelable": true,
        "is-confirmable": false,
        "is-discardable": false,
      },
      "permissions": {
        "can-apply": true,
        "can-cancel": true,
        "can-discard": true,
        "can-force-execute": true
      }
    },
    "relationships": {
      "apply": {...},
      "canceled-by": { ... },
      "configuration-version": {...},
      "confirmed-by": {...},
      "created-by": {...},
      "input-state-version": {...},
      "plan": {...},
      "run-events": {...},
      "policy-checks": {...},
      "workspace": {...},
      "comments": {...},
      "workspace-run-alerts": {...}
    }
  },
  "links": {
    "self": "/api/v2/runs/run-CZcmD7eagjhyX0vN"
  }
}
}
```

Apply a Run

POST /runs/:run_id/actions/apply

| Parameter | Description |
|-----------|---------------------|
| run_id | The run ID to apply |

Applies a run that is paused waiting for confirmation after a plan. This includes runs in the "needs confirmation" and "policy checked" states. This action is only required for workspaces without auto-apply enabled.

This endpoint queues the request to perform an apply; the apply might not happen immediately.

This endpoint represents an action as opposed to a resource. As such, the endpoint does not return any object in the response body.

Note: This endpoint cannot be accessed with organization tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts). You must access it with a user token (/docs/enterprise/users-teams-organizations/users.html#api-tokens) or team token (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts).

| Status | Response | Reason(s) |
|---|---|---|
| 202 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/202) | none | Successfully queued an apply request. |
| 409 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409) | JSON API error object (http://jsonapi.org/format/#error-objects) | Run was not paused for confirmation; apply not allowed. |

Request Body

This POST endpoint allows an optional JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|----------|--------|---------|------------------------------------|
| comment | string | null | An optional comment about the run. |

Sample Payload

This payload is optional, so the curl command will work without the --data @payload.json option too.

```
{  
  "comment": "Looks good to me"  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/runs/run-DQGdmrWMX8z9yWQB/actions/apply
```

List Runs in a Workspace

GET /workspaces/:workspace_id/runs

| Parameter | Description | |
|--------------|--|--------------------------|
| workspace_id | The workspace ID to list runs for. | |
| Status | Response | Reason |
| [200][] | Array of JSON API document (https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents)s (type: "runs") | Successfully listed runs |

Query Parameters

This endpoint supports pagination with standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|--------------|---|
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 runs per page. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/workspaces/ws-yF7z4gyEQRhaCNG9/runs
```

Sample Response

```
{  
  "data": [  
    {  
      "id": "run-bWSq4YeYpfrW4mx7",  
      "type": "runs",  
      "attributes": {  
        "auto-apply": false,  
        "error-text": null,  
        "is-destroy": false,  
        "message": "",  
        "metadata": {},  
        "source": "tfe-configuration-version",  
        "status": "planned",  
        "status-timestamps": {  
          "planned-at": "2017-11-28T22:52:51+00:00"  
        },  
        "terraform-version": "0.11.0",  
        "created-at": "2017-11-28T22:52:46.711Z",  
        "has-changes": true,  
        "actions": {  
          "is-cancelable": false,  
          "is-confirmable": true,  
          "is-discardable": true,  
          "is-force-cancelable": false  
        },  
        "permissions": {  
          "can-apply": true,  
          "can-cancel": true,  
          "can-discard": true,  
          "can-force-cancel": false,  
          "can-force-execute": true  
        }  
      },  
      "relationships": {  
        "workspace": {...},  
        "apply": {...},  
        "canceled-by": {...},  
        "configuration-version": {...},  
        "confirmed-by": {...},  
        "created-by": {...},  
        "input-state-version": {...},  
        "plan": {...},  
        "run-events": {...},  
        "policy-checks": {...},  
        "comments": {...},  
        "workspace-run-alerts": {...}  
      },  
      "links": {  
        "self": "/api/v2/runs/run-bWSq4YeYpfrW4mx7"  
      }  
    },  
    {...}  
  ]  
}
```

Get run details

GET /runs/:run_id

| Parameter | Description |
|-----------|--------------------|
| :run_id | The run ID to get. |

This endpoint is used for showing details of a specific run.

| Status | Response | Reason |
|--|---|--------------------------------------|
| [200][] | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "runs") | Success |
| 404
(Status/404">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Run not found or user not authorized |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  https://app.terraform.io/api/v2/runs/run-bWSq4YeYpfrW4mx7
```

Sample Response

```
{
  "data": {
    "id": "run-bWSq4YeYpfrW4mx7",
    "type": "runs",
    "attributes": {
      "auto-apply": false,
      "error-text": null,
      "is-destroy": false,
      "message": "",
      "metadata": {},
      "source": "tfe-configuration-version",
      "status": "planned",
      "status-timestamps": {
        "planned-at": "2017-11-28T22:52:51+00:00"
      },
      "terraform-version": "0.11.0",
      "created-at": "2017-11-28T22:52:46.711Z",
      "has-changes": true,
      "actions": {
        "is-cancelable": false,
        "is-confirmable": true,
        "is-discardable": true,
        "is-force-cancelable": false
      },
      "permissions": {
        "can-apply": true,
        "can-cancel": true,
        "can-discard": true,
        "can-force-cancel": false,
        "can-force-execute": true
      }
    },
    "relationships": {
      "workspace": {...},
      "apply": {...},
      "canceled-by": {...},
      "configuration-version": {...},
      "confirmed-by": {...},
      "created-by": {...},
      "input-state-version": {...},
      "plan": {...},
      "run-events": {...},
      "policy-checks": {...},
      "comments": {...},
      "workspace-run-alerts": {...}
    },
    "links": {
      "self": "/api/v2/runs/run-bWSq4YeYpfrW4mx7"
    }
  }
}
```

Discard a Run

POST /runs/:run_id/actions/discard

| Parameter | Description |
|-----------|-----------------------|
| run_id | The run ID to discard |

The discard action can be used to skip any remaining work on runs that are paused waiting for confirmation or priority. This includes runs in the "pending," "needs confirmation," "policy checked," and "policy override" states.

This endpoint queues the request to perform a discard; the discard might not happen immediately. After discarding, the run is completed and later runs can proceed.

This endpoint represents an action as opposed to a resource. As such, it does not return any object in the response body.

Note: This endpoint cannot be accessed with organization tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts). You must access it with a user token (/docs/enterprise/users-teams-organizations/users.html#api-tokens) or team token (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts).

| Status | Response | Reason(s) |
|---|---|---|
| 202 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/202) | none | Successfully queued a discard request. |
| 409 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409) | JSON API error object (http://jsonapi.org/format/#error-objects) | Run was not paused for confirmation or priority; discard not allowed. |

Request Body

This POST endpoint allows an optional JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|----------|--------|---------|--|
| comment | string | null | An optional explanation for why the run was discarded. |

Sample Payload

This payload is optional, so the curl command will work without the --data @payload.json option too.

```
{  
  "comment": "This run was discarded"  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/runs/run-DQGdmrWMX8z9yWQB/actions/discard
```

Cancel a Run

POST /runs/:run_id/actions/cancel

| Parameter | Description |
|-----------|----------------------|
| run_id | The run ID to cancel |

The `cancel` action can be used to interrupt a run that is currently planning or applying. Performing a `cancel` is roughly equivalent to hitting `ctrl+c` during a Terraform plan or apply on the CLI. The running Terraform process is sent an `INT` signal, which instructs Terraform to end its work and wrap up in the safest way possible.

This endpoint queues the request to perform a `cancel`; the `cancel` might not happen immediately. After canceling, the run is completed and later runs can proceed.

This endpoint represents an action as opposed to a resource. As such, it does not return any object in the response body.

Note: This endpoint cannot be accessed with organization tokens ([/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts](#)). You must access it with a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) or team token ([/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts](#)).

| Status | Response | Reason(s) |
|---|---|--|
| 202 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/202) | none | Successfully queued a <code>cancel</code> request. |
| 409 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409) | JSON API error object (http://jsonapi.org/format/#error-objects) | Run was not planning or applying; <code>cancel</code> not allowed. |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Run was not found or user not authorized. |

Request Body

This POST endpoint allows an optional JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|----------|--------|---------|---|
| comment | string | null | An optional explanation for why the run was canceled. |

Sample Payload

This payload is optional, so the `curl` command will work without the `--data @payload.json` option too.

```
{  
  "comment": "This run was stuck and would never finish."  
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/runs/run-DQGdmrWMX8z9yWQB/actions/cancel
```

Forcefully cancel a run

POST /runs/:run_id/actions/force-cancel

| Parameter | Description |
|-----------|----------------------|
| run_id | The run ID to cancel |

The `force-cancel` action is like `cancel`, but ends the run immediately. Once invoked, the run is placed into a `canceled` state, and the running Terraform process is terminated. The workspace is immediately unlocked, allowing further runs to be queued. The `force-cancel` operation requires workspace admin privileges.

This endpoint enforces a prerequisite that a non-forceful cancel is performed first, and a cool-off period has elapsed. To determine if this criteria is met, it is useful to check the `data.attributes.is-force-cancelable` value of the run details endpoint. The time at which the `force-cancel` action will become available can be found using the `run details` endpoint, in the key `data.attributes.force_cancel_available_at`. Note that this key is only present in the payload after the initial cancel has been initiated.

This endpoint represents an action as opposed to a resource. As such, it does not return any object in the response body.

Note: This endpoint cannot be accessed with organization tokens ([/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts](#)). You must access it with a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) or team token ([/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts](#)).

Warning: This endpoint has potentially dangerous side-effects, including loss of any in-flight state in the running Terraform process. Use this operation with extreme caution.

| Status | Response | Reason(s) |
|--|--|--|
| 202
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/202) | none | Successfully queued a cancel request. |
| 409
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409) | JSON API error object
(http://jsonapi.org/format/#error-objects) | Run was not planning or applying, has not been canceled non-forcefully, or the cool-off period has not yet passed. |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object
(http://jsonapi.org/format/#error-objects) | Run was not found or user not authorized. |

Request Body

This POST endpoint allows an optional JSON object with the following properties as a request payload.

| Key path | Type | Default | Description |
|----------|--------|---------|---|
| comment | string | null | An optional explanation for why the run was canceled. |

Sample Payload

This payload is optional, so the `curl` command will work without the `--data @payload.json` option too.

```
{  
  "comment": "This run was stuck and would never finish."  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/runs/run-DQGdmrWMX8z9yWQB/actions/force-cancel
```

Available Related Resources

The GET endpoints above can optionally return related resources, if requested with the `include` query parameter ([/docs/enterprise/api/index.html#inclusion-of-related-resources](#)). The following resource types are available:

- `plan` - Additional information about plans.
- `apply` - Additional information about applies.
- `created_by` - Full user records of the users responsible for creating the runs.
- `configuration_version` - The configuration record used in the run.
- `configuration_version.ingress_attributes` - The commit information used in the run.

SSH Keys

Note: These API endpoints are in beta and are subject to change.

The ssh-key object represents an SSH key which includes a name and the SSH private key. An organization can have multiple SSH keys available.

SSH keys can be used in two places:

- They can be assigned to VCS provider integrations (available in the API as oauth-tokens (/docs/enterprise/api/oauth-tokens.html)). Bitbucket Server requires an SSH key; other providers only need an SSH key if your repositories include submodules that are only accessible via SSH (instead of your VCS provider's API).
- They can be assigned to workspaces (/docs/enterprise/api/workspaces.html#assign-an-ssh-key-to-a-workspace) and used when Terraform needs to clone modules from a Git server. This is only necessary when your configurations directly reference modules from a Git server; you do not need to do this if you use Terraform Enterprise's private module registry (/docs/enterprise/registry/index.html).

Important: The list and read methods on this API only provide metadata about SSH keys. The actual private key text is write-only, and Terraform Enterprise never provides it to users via the API or UI.

List SSH Keys

GET /organizations/:organization_name/ssh-keys

| Parameter | Description |
|--------------------|--|
| :organization_name | The name of the organization to list SSH keys for. |

Note: This endpoint cannot be accessed with organization tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts). You must access it with a user token (/docs/enterprise/users-teams-organizations/users.html#api-tokens) or team token (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts).

| Status | Response | Reason |
|--|---|---|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | Array of JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents s (type: "ssh-keys")) | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found or user not authorized |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  https://app.terraform.io/api/v2/organizations/my-organization/ssh-keys
```

Sample Response

```
{  
  "data": [  
    {  
      "attributes": {  
        "name": "SSH Key"  
      },  
      "id": "sshkey-GxrePWre1Ezug7aM",  
      "links": {  
        "self": "/api/v2/ssh-keys/sshkey-GxrePWre1Ezug7aM"  
      },  
      "type": "ssh-keys"  
    }  
  ]  
}
```

Get an SSH Key

GET /ssh-keys/:ssh_key_id

| Parameter | Description |
|-------------|------------------------|
| :ssh_key_id | The SSH key ID to get. |

This endpoint is for looking up the name associated with an SSH key ID. It does not retrieve the key text.

Note: This endpoint cannot be accessed with organization tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts). You must access it with a user token (/docs/enterprise/users-teams-organizations/users.html#api-tokens) or team token (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts).

| Status | Response | Reason |
|---|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "ssh-keys") | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | SSH key not found or user not authorized |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
https://app.terraform.io/api/v2/ssh-keys/sshkey-GxrePWre1Ezug7aM
```

Sample Response

```
{
  "data": {
    "attributes": {
      "name": "SSH Key"
    },
    "id": "sshkey-GxrePWre1Ezug7aM",
    "links": {
      "self": "/api/v2/ssh-keys/sshkey-GxrePWre1Ezug7aM"
    },
    "type": "ssh-keys"
  }
}
```

Create an SSH Key

POST /organizations/:organization_name/ssh-keys

| Parameter | Description |
|--------------------|--|
| :organization_name | The name of the organization to create an SSH key in. The organization must already exist, and the token authenticating the API request must belong to the "owners" team or a member of the "owners" team. |

Note: This endpoint cannot be accessed with organization tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts). You must access it with a user token (/docs/enterprise/users-teams-organizations/users.html#api-tokens) or team token (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts).

| Status | Response | Reason |
|--|---|--|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "ssh-keys") | Success |
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not authorized |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-----------------------|--------|---------|----------------------------------|
| data.type | string | | Must be "ssh-keys". |
| data.attributes.name | string | | A name to identify the SSH key. |
| data.attributes.value | string | | The text of the SSH private key. |

Sample Payload

```
{
  "data": {
    "type": "ssh-keys",
    "attributes": {
      "name": "SSH Key",
      "value": "-----BEGIN RSA PRIVATE KEY-----\nMIIEowIBAAKCAQEA...m6+JVgl..."
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/organizations/my-organization/ssh-keys
```

Sample Response

```
{
  "data": {
    "attributes": {
      "name": "SSH Key"
    },
    "id": "sshkey-GxrePWre1Ezug7aM",
    "links": {
      "self": "/api/v2/ssh-keys/sshkey-GxrePWre1Ezug7aM"
    },
    "type": "ssh-keys"
  }
}
```

Update an SSH Key

PATCH /ssh-keys/:ssh_key_id

| Parameter | Description |
|-------------|---------------------------|
| :ssh_key_id | The SSH key ID to update. |

This endpoint replaces the name and/or key text of an existing SSH key. Existing workspaces that use the key will be updated with the new values.

Only members of the owners team (or the owners team service account) can edit SSH keys.

Note: This endpoint cannot be accessed with organization tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts). You must access it with a user token (/docs/enterprise/users-teams-organizations/users.html#api-tokens) or team token (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts).

| Status | Response | Reason |
|--|---|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "ssh-keys") | Success |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | SSH key not found or user not authorized |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-----------------------|--------|-----------|---|
| data.type | string | | Must be "ssh-keys". |
| data.attributes.name | string | (nothing) | A name to identify the SSH key. If omitted, the existing name is preserved. |
| data.attributes.value | string | (nothing) | The text of the SSH private key. If omitted, the existing value is preserved. |

Sample Payload

```
{
  "data": {
    "attributes": {
      "name": "SSH Key for GitHub"
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/ssh-keys/sshkey-GxrePWre1Ezug7aM
```

Sample Response

```
{
  "data": {
    "attributes": {
      "name": "SSH Key for GitHub"
    },
    "id": "sshkey-GxrePWre1Ezug7aM",
    "links": {
      "self": "/api/v2/ssh-keys/sshkey-GxrePWre1Ezug7aM"
    },
    "type": "ssh-keys"
  }
}
```

Delete an SSH Key

DELETE /ssh-keys/:ssh_key_id

| Parameter | Description | |
|-------------|---------------------------|--|
| :ssh_key_id | The SSH key ID to delete. | |

Only members of the owners team (or the owners team service account) can delete SSH keys.

Note: This endpoint cannot be accessed with organization tokens ([/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts](#)). You must access it with a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) or team token ([/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts](#)).

| Status | Response | Reason |
|---|---|--|
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Nothing | Success |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | SSH key not found or user not authorized |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --request DELETE \  
  https://app.terraform.io/api/v2/ssh-keys/sshkey-GxrePWre1Ezug7aM
```

State Versions API

Note: These API endpoints are in beta and are subject to change.

Create a State Version

POST /workspaces/:workspace_id/state-versions

| Parameter | Description |
|---------------|--|
| :workspace_id | The workspace ID to create the new state version in. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace (/docs/enterprise/api/workspaces.html#show-workspace) endpoint. |

Creates a state version and sets it as the current state version for the given workspace. The workspace must be locked by the user creating a state version. The workspace may be locked with the API (/docs/enterprise/api/workspaces.html#lock-a-workspace) or with the UI (/docs/enterprise/workspaces/settings.html#workspace-lock). This is most useful for migrating existing state from open source Terraform into a new TFE workspace.

Warning: Use caution when uploading state to workspaces that have already performed Terraform runs. Replacing state improperly can result in orphaned or duplicated infrastructure resources.

Note: This endpoint cannot be accessed with organization tokens (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts). You must access it with a user token (/docs/enterprise/users-teams-organizations/users.html#api-tokens) or team token (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts).

| Status | Response | Reason |
|--|--|--|
| 201
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) | Successfully created a state version |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace not found, or user unauthorized to perform action |
| 409
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409) | JSON API error object (http://jsonapi.org/format/#error-objects) | Conflict; check the error object for more information |
| 412
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/412) | JSON API error object (http://jsonapi.org/format/#error-objects) | Precondition failed; check the error object for more information |

| Status | Response | Reason |
|--|---|--|
| 422
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------------------|---------|-----------|--|
| data.type | string | | Must be "state-versions". |
| data.attributes.serial | integer | | The serial of the state version. Must match the serial value extracted from the raw state file. |
| data.attributes.md5 | string | | An MD5 hash of the raw state version |
| data.attributes.lineage | string | (nothing) | Optional Lineage of the state version. Should match the lineage extracted from the raw state file. Early versions of terraform did not have the concept of lineage, so this is an optional attribute. |
| data.attributes.state | string | | Base64 encoded raw state file |
| data.relationships.run.data.id | string | (nothing) | Optional The ID of the run to associate with the state version. |

Sample Payload

```
{
  "data": {
    "type": "state-versions",
    "attributes": {
      "serial": 1,
      "md5": "d41d8cd98f00b204e9800998ecf8427e",
      "lineage": "871d1b4a-e579-fb7c-ffdb-f0c858a647a7",
      "state": "..."
    },
    "relationships": {
      "run": {
        "data": {
          "type": "runs",
          "id": "run-bWSq4YeYpfrW4mx7"
        }
      }
    }
  }
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/workspaces/ws-6fHMCom98SDXSQUv/state-versions
```

Sample Response

Note: The hosted-state-download-url attribute provides a url from which you can download the raw state.

```
{
  "data": {
    "id": "sv-DmoXecHePnNznaA4",
    "type": "state-versions",
    "attributes": {
      "vcs-commit-sha": null,
      "vcs-commit-url": null,
      "created-at": "2018-07-12T20:32:01.490Z",
      "hosted-state-download-url": "https://terraform.io/v1/object/f55b739b-ff03-4716-b436-726466b96dc4",
      "serial": 1
    },
    "links": {
      "self": "/api/v2/state-versions/sv-DmoXecHePnNznaA4"
    }
  }
}
```

List State Versions for a Workspace

GET /state-versions

Query Parameters

This endpoint supports pagination with standard URL query parameters (/docs/enterprise/api/index.html#query-parameters); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|----------------------------|---|
| filter[workspace][name] | Required The name of one workspace to list versions for. |
| filter[organization][name] | Required The name of the organization that owns the desired workspace. |
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 20 state versions per page. |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/state-versions?filter%5Bworkspace%5D%5Bname%5D=my-workspace&filter%5Borganization%5D%5Bname%5D=my-organization
```

Sample Response

Note: The hosted-state-download-url attribute provides a url from which you can download the raw state.

```
{
  "data": [
    {
      "id": "sv-SDboVZC8TCxXEneJ",
      "type": "state-versions",
      "attributes": {
        "vcs-commit-sha": null,
        "vcs-commit-url": null,
        "created-at": "2018-08-27T14:49:47.902Z",
        "hosted-state-download-url": "https://archivist.terraform.io/v1/object/...",
        "serial": 3
      },
      "relationships": {
        "run": {
          "data": {
            "type": "runs"
          }
        },
        "created-by": {
          "data": {
            "id": "api-org-my-organization",
            "type": "users"
          }
        },
        "links": {
          "related": "/api/v2/runs/sv-SDboVZC8TCxXEneJ/created-by"
        }
      }
    },
    {
      "links": {
        "self": "/api/v2/state-versions/sv-SDboVZC8TCxXEneJ"
      }
    },
    {
      "id": "sv-UdqGARTddt8SEJEi",
      "type": "state-versions",
      "attributes": {
        "vcs-commit-sha": null,
        "vcs-commit-url": null,
        "created-at": "2018-08-27T14:49:46.102Z",
        "hosted-state-download-url": "https://archivist.terraform.io/v1/object/...",
        "serial": 2
      },
      "relationships": {
        "run": {
          "data": {
            "type": "runs"
          }
        }
      }
    }
  ]
}
```

```

        }
    },
    "created-by": {
        "data": {
            "id": "api-org-my-organization",
            "type": "users"
        },
        "links": {
            "related": "/api/v2/runs/sv-UdqGARTddt8SEJEi/created-by"
        }
    }
},
"links": {
    "self": "/api/v2/state-versions/sv-UdqGARTddt8SEJEi"
}
}
],
"links": {
    "self": "https://app.terraform.io/api/v2/state-versions?filter%5Borganization%5D%5Bname%5D=my-organization&filter%5Bworkspace%5D%5Bname%5D=my-workspace&page%5Bnumber%5D=1&page%5Bsize%5D=20",
    "first": "https://app.terraform.io/api/v2/state-versions?filter%5Borganization%5D%5Bname%5D=my-organization&filter%5Bworkspace%5D%5Bname%5D=my-workspace&page%5Bnumber%5D=1&page%5Bsize%5D=20",
    "prev": null,
    "next": null,
    "last": "https://app.terraform.io/api/v2/state-versions?filter%5Borganization%5D%5Bname%5D=my-organization&filter%5Bworkspace%5D%5Bname%5D=my-workspace&page%5Bnumber%5D=1&page%5Bsize%5D=20"
},
"meta": {
    "pagination": {
        "current-page": 1,
        "prev-page": null,
        "next-page": null,
        "total-pages": 1,
        "total-count": 2
    }
}
}
}

```

Fetch the Current State Version for a Workspace

GET /workspaces/:workspace_id/current-state-version

| Parameter | Description |
|-----------|-------------|
|-----------|-------------|

| | |
|---------------|---|
| :workspace_id | The ID for the workspace whose current state version you want to fetch. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace (/docs/enterprise/api/workspaces.html#show-workspace) endpoint. |
|---------------|---|

Fetches the current state version for the given workspace. This state version will be the input state when running terraform operations.

| Status | Response | Reason |
|--|--|---|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) | Successfully returned current state version for the given workspace |

| Status | Response | Reason |
|--|---|--|
| 404
(Status/404">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace not found, workspace does not have a current state version, or user unauthorized to perform action |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/workspaces/ws-6fHMCm98SDXSQUV/current-state-version
```

Sample Response

Note: The `hosted-state-download-url` attribute provides a url from which you can download the raw state.

```
{
  "data": {
    "id": "sv-SDboVZC8TCxXEneJ",
    "type": "state-versions",
    "attributes": {
      "vcs-commit-sha": null,
      "vcs-commit-url": null,
      "created-at": "2018-08-27T14:49:47.902Z",
      "hosted-state-download-url": "https://archivist.terraform.io/v1/object/...",
      "serial": 3
    },
    "relationships": {
      "run": {
        "data": {
          "type": "runs"
        }
      },
      "created-by": {
        "data": {
          "id": "api-org-hashicorp",
          "type": "users"
        }
      },
      "links": {
        "related": "/api/v2/runs/sv-SDboVZC8TCxXEneJ/created-by"
      }
    },
    "links": {
      "self": "/api/v2/state-versions/sv-SDboVZC8TCxXEneJ"
    }
  }
}
```

Team access API

Note: These API endpoints are in beta and are subject to change.

The team access APIs are used to associate a team to permissions on a workspace. A single `team-workspace` resource contains the relationship between the Team and Workspace, including the privileges the team has on the workspace.

List Team Access to Workspaces

GET /team-workspaces

Query Parameters

These are standard URL query parameters ([/docs/enterprise/api/index.html#query-parameters](#)); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|------------------------------------|--|
| <code>filter[workspace][id]</code> | Required. The workspace ID to list team access for. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace (/docs/enterprise/api/workspaces.html#show-workspace) endpoint. |

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
"https://app.terraform.io/api/v2/team-workspaces?filter%5Bworkspace%5D%5Bid%5D=ws-5vBKrazjYR36gcYX"
```

Sample Response

```
{
  "data": [
    {
      "id": "131",
      "type": "team-workspaces",
      "attributes": {
        "access": "read"
      },
      "relationships": {
        "team": {
          "data": {
            "id": "team-BUHBEM97xboT8TVz",
            "type": "teams"
          },
          "links": {
            "related": "/api/v2/teams/devs"
          }
        },
        "workspace": {
          "data": {
            "id": "ws-5vBKrazjYR36gcYX",
            "type": "workspaces"
          },
          "links": {
            "related": "/api/v2/organizations/my-organization/workspaces/ws-5vBKrazjYR36gcYX"
          }
        }
      },
      "links": {
        "self": "/api/v2/team-workspaces/131"
      }
    }
  ]
}
```

Add Team Access to a Workspace

`POST /team-workspaces`

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|---|--------|---------|--|
| <code>data.type</code> | string | | Must be "team-workspaces". |
| <code>data.attributes.access</code> | string | | The type of access to grant. Valid values are <code>read</code> , <code>write</code> , or <code>admin</code> . |
| <code>data.relationships.workspace.data.type</code> | string | | Must be <code>workspaces</code> . |
| <code>data.relationships.workspace.data.id</code> | string | | The workspace ID to which the team is to be added. |
| <code>data.relationships.team.data.type</code> | string | | Must be <code>teams</code> . |

| Key path | Type | Default | Description |
|---------------------------------|--------|---------|---|
| data.relationships.team.data.id | string | | The ID of the team to add to the workspace. |

Sample Payload

```
{  
  "data": {  
    "attributes": {  
      "access": "read"  
    },  
    "relationships": {  
      "workspace": {  
        "data": {  
          "type": "workspaces",  
          "id": "ws-5vBKrazjYR36gcYX"  
        }  
      },  
      "team": {  
        "data": {  
          "type": "teams",  
          "id": "team-BUHBEM97xboT8TVz"  
        }  
      }  
}
```

Sample Request

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  

```

Sample Response

```
{  
  "data": {  
    "id": "131",  
    "type": "team-workspaces",  
    "attributes": {  
      "access": "read"  
    },  
    "relationships": {  
      "team": {  
        "data": {  
          "id": "team-BUHBEM97xboT8TVz",  
          "type": "teams"  
        },  
        "links": {  
          "related": "/api/v2/teams/devs"  
        }  
      },  
      "workspace": {  
        "data": {  
          "id": "ws-5vBKrazjYR36gcYX",  
          "type": "workspaces"  
        },  
        "links": {  
          "related": "/api/v2/organizations/my-organization/workspaces/ws-5vBKrazjYR36gcYX"  
        }  
      }  
    },  
    "links": {  
      "self": "/api/v2/team-workspaces/131"  
    }  
  }  
}
```

Show Team Access to a Workspace

GET /team-workspaces/:id`

Parameter Description

:id The ID of the team/workspace relationship. Obtain this from the list team access action described above.

Sample Request

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request GET \  
  https://app.terraform.io/api/v2/team-workspaces/257525
```

Sample Response

```
{  
  "data": {  
    "type": "team-workspaces",  
    "id": "1",  
    "attributes": { "permission": "read" }  
  }  
}
```

Remove Team Access to a Workspace

DELETE /team-workspaces/:id

| Parameter | Description |
|-----------|-------------|
|-----------|-------------|

| | |
|-----|--|
| :id | The ID of the team/workspace relationship. Obtain this from the list team access action described above. |
|-----|--|

Sample Request

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request DELETE \  
  https://app.terraform.io/api/v2/team-workspaces/257525
```

Team Membership API

Note: These API endpoints are in beta and are subject to change.

The Team Membership API is used to add or remove users from teams. The Team API (/docs/enterprise/api/teams.html) is used to create or destroy teams.

Add a User to Team

This method adds multiple users to a team. Both users and teams must already exist.

`POST /teams/:team_id/relationships/users`

| Parameter | Description |
|-----------------------|---------------------|
| <code>:team_id</code> | The ID of the team. |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------------|--------|---------|----------------------------------|
| <code>data[].type</code> | string | | Must be "users". |
| <code>data[].id</code> | string | | The username of the user to add. |

Sample Payload

```
{
  "data": [
    {
      "type": "users",
      "id": "myuser1"
    },
    {
      "type": "users",
      "id": "myuser2"
    }
  ]
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/teams/257525/relationships/users
```

Delete a User from Team

This method removes multiple users from a team. Both users and teams must already exist. This DOES NOT delete the user; it only removes them from this team.

`DELETE /teams/:team_id/relationships/users`

| Parameter | Description |
|-----------------------|---------------------|
| <code>:team_id</code> | The ID of the team. |

Request Body

This DELETE endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------------|--------|---------|-------------------------------------|
| <code>data[].type</code> | string | | Must be "users". |
| <code>data[].id</code> | string | | The username of the user to remove. |

Sample Payload

```
{  
  "data": [  
    {  
      "type": "users",  
      "id": "myuser1"  
    },  
    {  
      "type": "users",  
      "id": "myuser2"  
    }  
  ]  
}
```

Sample Request

```
$ curl \  
--header "Authorization: Bearer $TOKEN" \  
--header "Content-Type: application/vnd.api+json" \  
--request DELETE \  
--data @payload.json \  
https://app.terraform.io/api/v2/teams/257525/relationships/users
```

Team Token API

Note: These API endpoints are in beta and are subject to change.

Generate a new team token

Generates a new team token and overrides existing token if one exists.

| Method | Path |
|--------|--------------------------------------|
| POST | /teams/:team_id/authentication-token |

Parameters

- `:team_id(string: <required>)` - specifies the team ID for generating the team token

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  https://app.terraform.io/api/v2/teams/team-BUHBEM97xboT8TVz/authentication-token
```

Sample Response

```
{  
  "data": {  
    "id": "4111797",  
    "type": "authentication-tokens",  
    "attributes": {  
      "created-at": "2017-11-29T19:18:09.976Z",  
      "last-used-at": null,  
      "description": null,  
      "token": "QnbSxjjhVMHJgw.atlasv1.gxZnWIjI5j752DGqdwEUVLOFF0mtyaQ00H9bA1j90qWb254lEkQy0dfqqcq9zzL7Sm  
0"  
    },  
    "relationships": {  
      "created-by": {  
        "data": {  
          "id": "user-62goNpx1ThQf689e",  
          "type": "users"  
        }  
      }  
    }  
  }  
}
```

Delete the team token

| Method | Path |
|--------|--------------------------------------|
| DELETE | /teams/:team_id/authentication-token |

Parameters

- `:team_id(string: <required>)` - specifies the team_id from which to delete the token

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request DELETE \  
  https://app.terraform.io/api/v2/teams/team-BUHBEM97xboT8TVz/authentication-token
```

Teams API

Note: These API endpoints are in beta and are subject to change.

The Teams API is used to create and destroy teams. The Team Membership API ([/docs/enterprise/api/team-members.html](#)) is used to add or remove users from a team. To give a team access to a workspace use the Team Access API ([/docs/enterprise/api/team-access.html](#)) to associate a team with privileges on a workspace.

List teams

GET `organizations/:organization_name/teams`

| Parameter | Description |
|---------------------------------|--|
| <code>:organization_name</code> | The name of the organization to list teams from. |

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/organizations/my-organization/teams
```

Sample Response

```
{
  "data": [
    {
      "id": "257529",
      "type": "teams",
      "attributes": {
        "name": "owners",
        "users-count": 1,
        "permissions": {
          "can-update-membership": false,
          "can-destroy": false
        }
      },
      "relationships": {
        "users": {
          "data": [
            {
              "id": "user-62goNpx1ThQf689e",
              "type": "users"
            }
          ]
        },
        "authentication-token": {
          "meta": {}
        }
      },
      "links": {
        "self": "/api/v2/teams/team-n8UQ6wfhyym25sMe"
      }
    }
  ]
}
```

Create a Team

POST /organizations/:organization_name/teams

| Parameter | Description | |
|--|--|--|
| :organization_name | The name of the organization to create the team in. The organization must already exist in the system, and the user must have permissions to create new teams. | |
| Status | Response | Reason |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "teams") | Successfully created a team |
| 400
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400) | JSON API error object (http://jsonapi.org/format/#error-objects) | Invalid include parameter |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Organization not found, or user unauthorized to perform action |

| Status | Response | Reason |
|--|---|--|
| 422
(Status/422">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |
| 500
(Status/500">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/500) | JSON API error object (http://jsonapi.org/format/#error-objects) | Failure during team creation |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|----------------------|--------|---------|---|
| data.type | string | | Must be "teams". |
| data.attributes.name | string | | The name of the team, which can only include letters, numbers, -, and _. This will be used as an identifier and must be unique in the organization. |

Sample Payload

```
{
  "data": {
    "type": "teams",
    "attributes": {
      "name": "team-creation-test"
    }
  }
}
```

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/organizations/my-organization/teams
```

Sample Response

```
{  
  "id": "257528",  
  "type": "teams",  
  "attributes": {  
    "name": "team-creation-test",  
    "users-count": 0  
  },  
  "relationships": {  
    "users": {  
      "data": []  
    }  
  },  
  "links": {  
    "self": "/api/v2/teams/257528"  
  }  
}
```

Show Team Information

GET /teams/:team_id

| Parameter | Description |
|-----------|--------------------------|
| :team_id | The team ID to be shown. |

Sample Request

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request GET \  
  https://app.terraform.io/api/v2/teams/257529
```

Sample Response

```
{  
  "data": {  
    "id": "257529",  
    "type": "teams",  
    "attributes": {  
      "name": "owners",  
      "users-count": 1,  
      "permissions": {  
        "can-update-membership": false,  
        "can-destroy": false  
      }  
    },  
    "relationships": {  
      "users": {  
        "data": [  
          {  
            "id": "user-62goNpx1ThQf689e",  
            "type": "users"  
          }  
        ]  
      },  
      "authentication-token": {  
        "meta": {}  
      }  
    },  
    "links": {  
      "self": "/api/v2/teams/257529"  
    }  
  }  
}
```

Delete a Team

`DELETE /teams/:team_id`

| Parameter | Description |
|-----------------------|----------------------------|
| <code>:team_id</code> | The team ID to be deleted. |

Sample Request

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request DELETE \  
  https://app.terraform.io/api/v2/teams/257529
```

Available Related Resources

The GET endpoints above can optionally return related resources, if requested with the `include` query parameter ([/docs/enterprise/api/index.html#inclusion-of-related-resources](#)). The following resource types are available:

- **users (string)** - Returns the full user record for every member of a team.

User Tokens API

Note: These API endpoints are in beta and are subject to change.

List User Tokens

GET /api/v2/users/:user_id/authentication-tokens

| Parameter | Description |
|-----------|---------------------|
| :user_id | The ID of the User. |

The objects returned by this endpoint only contain metadata, and do not include the secret text of any authentication tokens. A token is only shown upon creation, and cannot be recovered later.

Note: You must access this endpoint with a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)), and it will only return useful data for that token's user account.

| Status | Response | Reason |
|--|--|---|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "authentication-tokens") | The request was successful |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | Empty JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (no type) | User has no authentication tokens, or request was made by someone other than the user |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not found |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/users/user-MA4GL63FmYRpSFxa/authentication-tokens
```

Sample Response

```
{
  "data": [
    {
      "id": "at-QmATJea6aWj1xR2t",
      "type": "authentication-tokens",
      "attributes": {
        "created-at": "2018-11-06T22:56:10.203Z",
        "last-used-at": null,
        "description": null,
        "token": null
      },
      "relationships": {
        "created-by": {
          "data": null
        }
      }
    },
    {
      "id": "at-6yEmxNAhaoQLH1Da",
      "type": "authentication-tokens",
      "attributes": {
        "created-at": "2018-11-25T22:31:30.624Z",
        "last-used-at": "2018-11-26T20:27:54.931Z",
        "description": "api",
        "token": null
      },
      "relationships": {
        "created-by": {
          "data": {
            "id": "user-MA4GL63FmYRpSFxa",
            "type": "users"
          }
        }
      }
    }
  ]
}
```

Show a User Token

GET /api/v2/authentication-tokens/:id

| Parameter | Description |
|-----------|---------------------------|
| :id | The ID of the User Token. |

The objects returned by this endpoint only contain metadata, and do not include the secret text of any authentication tokens. A token is only shown upon creation, and cannot be recovered later.

Note: You must access this endpoint with a user token (</docs/enterprise/users-teams-organizations/users.html#api-tokens>), and it will only return useful data for that token's user account.

| Status | Response | Reason |
|--|--|--|
| 200
(Status/200">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "authentication-tokens") | The request was successful |
| 404
(Status/404">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User Token not found, or unauthorized to view the User Token |

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request GET \
https://app.terraform.io/api/v2/authentication-tokens/at-6yEmxNAhaoQLH1Da
```

Sample Response

```
{
  "data": {
    "id": "at-6yEmxNAhaoQLH1Da",
    "type": "authentication-tokens",
    "attributes": {
      "created-at": "2018-11-25T22:31:30.624Z",
      "last-used-at": "2018-11-26T20:34:59.487Z",
      "description": "api",
      "token": null
    },
    "relationships": {
      "created-by": {
        "data": {
          "id": "user-MA4GL63FmYRpSFxa",
          "type": "users"
        }
      }
    }
  }
}
```

Create a User Token

POST /api/v2/users/:user_id/authentication-tokens

| Parameter | Description |
|-----------|---------------------|
| :user_id | The ID of the User. |

This endpoint returns the secret text of the created authentication token. A token is only shown upon creation, and cannot

be recovered later.

Note: You must access this endpoint with a user token (</docs/enterprise/users-teams-organizations/users.html#api-tokens>), and it will only create new tokens for that token's user account.

| Status | Response | Reason |
|--|--|--|
| 201
(Status/201">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/201) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "authentication-tokens") | The request was successful |
| 404
(Status/404">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not found or user unauthorized to perform action |
| 422
(Status/422">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/422) | JSON API error object (http://jsonapi.org/format/#error-objects) | Malformed request body (missing attributes, wrong types, etc.) |
| 500
(Status/500">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/500) | JSON API error object (http://jsonapi.org/format/#error-objects) | Failure during User Token creation |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-----------------------------|--------|---------|-------------------------------------|
| data.type | string | | Must be "authentication-tokens". |
| data.attributes.description | string | | The description for the User Token. |

Sample Payload

```
{  
  "data": {  
    "type": "authentication-tokens",  
    "attributes": {  
      "description": "api"  
    }  
  }  
}
```

Sample Request

```
curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
--data @payload.json \
https://app.terraform.io/api/v2/users/user-MA4GL63FmYRpSFxa/authentication-tokens
```

Sample Response

```
{
  "data": {
    "id": "at-MKD1X3i4HS3AuD41",
    "type": "authentication-tokens",
    "attributes": {
      "created-at": "2018-11-26T20:48:35.054Z",
      "last-used-at": null,
      "description": "api",
      "token": "6tL24nM38M7XWQ.atlasv1.KmWckRfzeNmUVFNvpvwUEChKaLGznCSD6fPf3VPzqMMVzmSxFU0p2Ibzpo2h5eTGwpu"
    },
    "relationships": {
      "created-by": {
        "data": {
          "id": "user-MA4GL63FmYRpSFxa",
          "type": "users"
        }
      }
    }
  }
}
```

Destroy a User Token

`DELETE /api/v2/authentication-tokens/:id`

| Parameter | Description |
|------------------|--------------------------------------|
| <code>:id</code> | The ID of the User Token to destroy. |

Note: You must access this endpoint with a user token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)), and it will only delete tokens for that token's user account.

| Status | Response | Reason |
|---|---|--|
| 204 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204) | Empty response | The User Token was successfully destroyed |
| 404 (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User Token not found, or user unauthorized to perform action |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request DELETE \  
  https://app.terraform.io/api/v2/authentication-tokens/at-6yEmxNAhaoQLH1Da
```

Users API

Note: These API endpoints are in beta and are subject to change.

Show a User

Shows details for a user. The ID for a user can be obtained from the Team (/docs/enterprise/api/teams.html#list-teams) endpoint. ?include=users should be included in the query string in order to have usernames be included in the response.

GET /users/:user_id

| Status | Response | Reason |
|--|--|--|
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "users") | The request was successful |
| 401
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401) | JSON API error object (http://jsonapi.org/format/#error-objects) | Unauthorized |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | User not found, or unauthorized to view the user |

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request GET \  
  https://app.terraform.io/api/v2/users/user-MA4GL63FmYRpSFxa
```

Sample Response

```
{  
  "data": {  
    "id": "user-MA4GL63FmYRpSFxa",  
    "type": "users",  
    "attributes": {  
      "username": "admin",  
      "is-service-account": false,  
      "avatar-url": "https://www.gravatar.com/avatar/fa1f0c9364253d351bf1c7f5c534cd40?s=100&d=mm",  
      "v2-only": true,  
      "permissions": {  
        "can-create-organizations": false,  
        "can-change-email": true,  
        "can-change-username": true  
      }  
    },  
    "relationships": {  
      "authentication-tokens": {  
        "links": {  
          "related": "/api/v2/users/user-MA4GL63FmYRpSFxa/authentication-tokens"  
        }  
      }  
    },  
    "links": {  
      "self": "/api/v2/users/user-MA4GL63FmYRpSFxa"  
    }  
  }  
}
```

Variables API

Note: These API endpoints are in beta and are subject to change.

This set of APIs covers create, update, list and delete operations on variables.

Create a Variable

POST /vars

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--|--------|---------|--|
| data.type | string | | Must be "vars". |
| data.attributes.key | string | | The name of the variable. |
| data.attributes.value | string | | The value of the variable. |
| data.attributes.category | string | | Whether this is a Terraform or environment variable. Valid values are "terraform" or "env". |
| data.attributes.hcl | bool | false | Whether to evaluate the value of the variable as a string of HCL code. Has no effect for environment variables. |
| data.attributes.sensitive | bool | false | Whether the value is sensitive. If true then the variable is written once and not visible thereafter. |
| data.relationships.workspace.data.type | string | | Must be "workspaces". |
| data.relationships.workspace.data.id | string | | The ID of the workspace that owns the variable. Obtain workspace IDs from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace (/docs/enterprise/api/workspaces.html#show-workspace) endpoint. |

Deprecation warning: The custom filter properties are replaced by JSON API relationships and will be removed from future versions of the API!

| Key path | Type | Default | Description |
|--------------------------|--------|---------|---|
| filter.workspace.name | string | | The name of the workspace that owns the variable. |
| filter.organization.name | string | | The name of the organization that owns the workspace. |

Sample Payload

```
{  
  "data": {  
    "type": "vars",  
    "attributes": {  
      "key": "some_key",  
      "value": "some_value",  
      "category": "terraform",  
      "hcl": false,  
      "sensitive": false  
    },  
    "relationships": {  
      "workspace": {  
        "data": {  
          "id": "ws-4j8p6jX1w33MiDC7",  
          "type": "workspaces"  
        }  
      }  
    }  
  }  
}
```

Sample Request

```
curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/vars
```

Sample Response

```
{
  "data": {
    "id": "var-EavQ1LztoRTQHSNT",
    "type": "vars",
    "attributes": {
      "key": "some_key",
      "value": "some_value",
      "sensitive": false,
      "category": "terraform",
      "hcl": false
    },
    "relationships": {
      "configurable": {
        "data": {
          "id": "ws-4j8p6jX1w33MiDC7",
          "type": "workspaces"
        },
        "links": {
          "related": "/api/v2/organizations/my-organization/workspaces/my-workspace"
        }
      }
    },
    "links": {
      "self": "/api/v2/vars/var-EavQ1LztoRTQHSNT"
    }
  }
}
```

List Variables

GET /vars

Query Parameters

These are standard URL query parameters ([/docs/enterprise/api/index.html#query-parameters](#)); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|-----------------------------|---|
| filter[workspace] [name] | Optional The name of one workspace to list variables for. If included, you must also include the organization name filter. |
| filter[organization] [name] | Optional The name of the organization that owns the desired workspace. If included, you must also include the workspace name filter. |

These two parameters are optional but linked; if you include one, you must include both. Without a filter, this method lists variables for all workspaces you can access.

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
"https://app.terraform.io/api/v2/vars?filter%5Borganization%5D%5Bname%5D=my-organization&filter%5Bworkspace%5D%5Bname%5D=my-workspace"
# ?filter[organization][name]=my-organization&filter[workspace][name]=demo01
```

Sample Response

```
{
  "data": [
    {
      "id": "var-AD4pibb9nxo1468E",
      "type": "vars", "attributes": {
        "key": "name",
        "value": "hello",
        "sensitive": false,
        "category": "terraform",
        "hcl": false
      },
      "relationships": {
        "configurable": {
          "data": {
            "id": "ws-cZE9LERN3rGPRAmH",
            "type": "workspaces"
          },
          "links": {
            "related": "/api/v2/organizations/my-organization/workspaces/my-workspace"
          }
        },
        "links": {
          "self": "/api/v2/vars/var-AD4pibb9nxo1468E"
        }
      }
    }
  ]
}
```

Update Variables

PATCH /vars/:variable_id

| Parameter | Description |
|--------------|---------------------------------------|
| :variable_id | The ID of the variable to be updated. |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|-----------------|--------|---------|---|
| data.type | string | | Must be "vars". |
| data.id | string | | The ID of the variable to update. |
| data.attributes | object | | New attributes for the variable. This object can include key, value, category, hcl, and sensitive properties, which are described above under create a variable. All of these properties are optional; if omitted, a property will be left unchanged. |

Sample Payload

```
{  
  "data": {  
    "id": "var-yRmifb4PJj7cLkMG",  
    "attributes": {  
      "key": "name",  
      "value": "mars",  
      "category": "terraform",  
      "hcl": false,  
      "sensitive": false  
    },  
    "type": "vars"  
  }  
}
```

Sample Request

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request PATCH \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/vars/var-yRmifb4PJj7cLkMG
```

Sample Response

```
{
  "data": {
    "id": "var-yRmifb4PJj7cLkMG",
    "type": "vars",
    "attributes": {
      "key": "name",
      "value": "mars",
      "sensitive": false,
      "category": "terraform",
      "hcl": false
    },
    "relationships": {
      "configurable": {
        "data": {
          "id": "ws-4j8p6jX1w33MiDC7",
          "type": "workspaces"
        },
        "links": {
          "related": "/api/v2/organizations/workspace-v2-06/workspaces/workspace-v2-06"
        }
      }
    },
    "links": {
      "self": "/api/v2/vars/var-yRmifb4PJj7cLkMG"
    }
  }
}
```

Delete Variables

`DELETE /vars/:variable_id`

| Parameter | Description |
|---------------------------|---------------------------------------|
| <code>:variable_id</code> | The ID of the variable to be deleted. |

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
https://app.terraform.io/api/v2/vars/var-yRmifb4PJj7cLkMG
```

Workspaces API

Note: These API endpoints are in beta and are subject to change.

Workspaces represent running infrastructure managed by Terraform.

Create a Workspace

POST /organizations/:organization_name/workspaces

| Parameter | Description |
|--------------------|--|
| :organization_name | The name of the organization to create the workspace in. The organization must already exist in the system, and the user must have permissions to create new workspaces. |

Note: Workspace creation is restricted to members of the owners team, the owners team service account (/docs/enterprise/users-teams-organizations/service-accounts.html#team-service-accounts), and the organization service account (/docs/enterprise/users-teams-organizations/service-accounts.html#organization-service-accounts).

Note: Migrating legacy workspaces (with the migration-environment attribute) can only be done with a user token (/docs/enterprise/users-teams-organizations/users.html#api-tokens). The user must be a member of the owners team in both the legacy organization and the new organization.

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

By supplying the necessary attributes under a vcs-repository object, you can create a workspace that is configured against a VCS Repository.

By supplying a migration-environment attribute, you can create a workspace which is migrated from a legacy environment. When you do this, the following will happen:

- Environment and Terraform variables will be copied to the workspace.
- Teams which are associated with the legacy environment will be created in the destination workspace's organization, if teams with those names don't already exist.
- Members of those teams will be added as members of the corresponding teams in the destination workspace's organization.
- Each team will be given the same access level on the workspace as it had on the legacy environment.
- The latest state of the legacy environment will be copied over into the workspace and set as the workspace's current state.

- VCS repo ingress settings (like branch and working directory) will be copied over into the workspace.

| Key path | Type | Default | Description |
|---|---------|-----------|---|
| data.type | string | | Must be "workspaces". |
| data.attributes.name | string | | The name of the workspace, which can only include letters, numbers, -, and _. This will be used as an identifier and must be unique in the organization. |
| data.attributes.auto-apply | boolean | false | Whether to automatically apply changes when a Terraform plan is successful. |
| data.attributes.migration-environment | string | (nothing) | The legacy TFE environment to use as the source of the migration, in the form organization/environment. Omit this unless you are migrating a legacy environment. |
| data.attributes.queue-all-runs | boolean | false | Whether runs should be queued immediately after workspace creation. When set to false, runs triggered by a VCS change will not be queued until at least one run is manually queued. |
| data.attributes.terraform-version | string | (nothing) | The version of Terraform to use for this workspace. Upon creating a workspace, the latest version is selected unless otherwise specified (e.g. "0.11.1"). |
| data.attributes.working-directory | string | (nothing) | A relative path that Terraform will execute within. This defaults to the root of your repository and is typically set to a subdirectory matching the environment when multiple environments exist within the same repository. |
| data.attributes.vcs-repo | object | (nothing) | Settings for the workspace's VCS repository. If omitted, the workspace is created without a VCS repo. If included, you must specify at least the oauth-token-id and identifier keys below. |
| data.attributes.vcs-repo.oauth-token-id | string | | The VCS Connection (OAuth Connection + Token) to use. This ID can be obtained from the oauth-tokens (/docs/enterprise/api/oauth-tokens.html) endpoint. |
| data.attributes.vcs-repo.branch | string | (nothing) | The repository branch that Terraform will execute from. If omitted or submitted as an empty string, this defaults to the repository's default branch (e.g. master). |
| data.attributes.vcs-repo.ingress-submodules | boolean | false | Whether submodules should be fetched when cloning the VCS repository. |
| data.attributes.vcs-repo.identifier | string | | A reference to your VCS repository in the format :org/:repo where :org and :repo refer to the organization and repository in your VCS provider. |

Sample Payload

Without a VCS repository

```
{  
  "data": {  
    "attributes": {  
      "name": "workspace-1"  
    },  
    "type": "workspaces"  
  }  
}
```

With a VCS repository

```
{  
  "data": {  
    "attributes": {  
      "name": "workspace-2",  
      "terraform_version": "0.11.1",  
      "working-directory": "",  
      "vcs-repo": {  
        "identifier": "skierkowski/terraform-test-proj",  
        "oauth-token-id": "ot-hmAyP66qk2AMVdbJ",  
        "branch": "",  
        "default-branch": true  
      }  
    },  
    "type": "workspaces"  
  }  
}
```

Migrating a legacy environment

```
{  
  "data": {  
    "attributes": {  
      "name": "workspace-2",  
      "migration-environment":  
        "legacy-hashicorp-organization/legacy-environment",  
      "vcs-repo": {  
        "identifier": "skierkowski/terraform-test-proj",  
        "oauth-token-id": "ot-hmAyP66qk2AMVdbJ"  
      }  
    },  
    "type": "workspaces"  
  }  
}
```

Sample Request

```
$ curl \  
--header "Authorization: Bearer $TOKEN" \  
--header "Content-Type: application/vnd.api+json" \  
--request POST \  
--data @payload.json \  
https://app.terraform.io/api/v2/organizations/my-organization/workspaces
```

Sample Response

Without a VCS repository

```
{  
  "data": {  
    "id": "ws-YnyXLq9fy38afEeb",  
    "type": "workspaces",  
    "attributes": {  
      "auto-apply": false,  
      "can-queue-destroy-plan": false,  
      "created-at": "2017-11-18T00:43:59.384Z",  
      "environment": "default",  
      "locked": false,  
      "name": "workspace-1",  
      "permissions": {  
        "can-update": true,  
        "can-destroy": false,  
        "can-queue-destroy": false,  
        "can-queue-run": false,  
        "can-update-variable": false,  
        "can-lock": false,  
        "can-read-settings": true  
      },  
      "queue-all-runs": false,  
      "terraform-version": "0.11.0",  
      "vcs-repo": null,  
      "working-directory": ""  
    },  
    "relationships": {  
      "organization": {  
        "data": {  
          "id": "my-organization",  
          "type": "organizations"  
        }  
      },  
      "ssh-key": {  
        "data": null  
      },  
      "latest-run": {  
        "data": null  
      }  
    },  
    "links": {  
      "self": "/api/v2/organizations/my-organization/workspaces/workspace-1"  
    }  
  }  
}
```

With a VCS repository

```
{
  "data": {
    "id": "ws-SihZTyXKfNXUWuUa",
    "type": "workspaces",
    "attributes": {
      "auto-apply": false,
      "can-queue-destroy-plan": true,
      "created-at": "2017-11-02T23:55:16.142Z",
      "environment": "default",
      "locked": false,
      "name": "workspace-2",
      "permissions": {
        "can-update": true,
        "can-destroy": false,
        "can-queue-destroy": false,
        "can-queue-run": false,
        "can-update-variable": false,
        "can-lock": false,
        "can-read-settings": true
      },
      "queue-all-runs": false,
      "terraform-version": "0.10.8",
      "vcs-repo": {
        "identifier": "skierkowski/terraform-test-proj",
        "branch": "",
        "oauth-token-id": "ot-hmAyP66qk2AMVdbJ",
        "ingress-submodules": false
      },
      "working-directory": null
    },
    "relationships": {
      "organization": {
        "data": {
          "id": "my-organization",
          "type": "organizations"
        }
      },
      "ssh-key": {
        "data": null
      },
      "latest-run": {
        "data": null
      }
    },
    "links": {
      "self": "/api/v2/organizations/my-organization/workspaces/workspace-2"
    }
  }
}
```

Migrating a legacy environment

```
{
  "data": {
    "id": "ws-SihZTyXKfNXUWuUa",
    "type": "workspaces",
    "attributes": {
      "auto-apply": false,
      "can-queue-destroy-plan": true,
      "created-at": "2017-11-02T23:55:16.142Z",
      "environment": "default",
      "locked": false,
      "name": "workspace-2",
      "permissions": {
        "can-update": true,
        "can-destroy": false,
        "can-queue-destroy": false,
        "can-queue-run": false,
        "can-update-variable": false,
        "can-lock": false,
        "can-read-settings": true
      },
      "queue-all-runs": false,
      "terraform-version": "0.10.8",
      "vcs-repo": {
        "identifier": "skierkowski/terraform-test-proj",
        "branch": "",
        "oauth-token-id": "ot-hmAyP66qk2AMVdbJ",
        "ingress-submodules": false
      },
      "working-directory": null
    },
    "relationships": {
      "organization": {
        "data": {
          "id": "my-organization",
          "type": "organizations"
        }
      },
      "ssh-key": {
        "data": null
      },
      "latest-run": {
        "data": null
      }
    },
    "links": {
      "self": "/api/v2/organizations/my-organization/workspaces/workspace-2"
    }
  }
}
```

Update a Workspace

PATCH /organizations/:organization_name/workspaces/:name

| Parameter | Description |
|--------------------|--|
| :organization_name | The name of the organization to create the workspace in. The organization must already exist in the system, and the user must have permissions to create new workspaces. |

| Parameter | Description |
|-----------|--|
| :name | The name of the workspace to update, which can only include letters, numbers, -, and _. This will be used as an identifier and must be unique in the organization. |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

Note that workspaces without an associated VCS repository only use the `auto-apply`, `terraform-version`, and `working-directory`.

| Key path | Type | Default | Description |
|---|----------------|------------------|---|
| data.type | string | | Must be "workspaces". |
| data.attributes.name | string | (previous value) | A new name for the workspace, which can only include letters, numbers, -, and _. This will be used as an identifier and must be unique in the organization. Warning: Changing a workspace's name changes its URL in the API and UI. |
| data.attributes.auto-apply | boolean | (previous value) | Whether to automatically apply changes when a Terraform plan is successful. |
| data.attributes.queue-all-runs | boolean | (previous value) | Whether runs should be queued immediately after workspace creation. When set to false, runs triggered by a VCS change will not be queued until at least one run is manually queued. |
| data.attributes.terraform-version | string | (previous value) | The version of Terraform to use for this workspace. |
| data.attributes.working-directory | string | (previous value) | A relative path that Terraform will execute within. This defaults to the root of your repository and is typically set to a subdirectory matching the environment when multiple environments exist within the same repository. |
| data.attributes.vcs-repo | object or null | (previous value) | To delete a workspace's existing VCS repo, specify null instead of an object. To modify a workspace's existing VCS repo, include whichever of the keys below you wish to modify. To add a new VCS repo to a workspace that didn't previously have one, include at least the <code>oauth-token-id</code> and <code>identifier</code> keys. |
| data.attributes.vcs-repo.oauth-token-id | string | (previous value) | The VCS Connection (OAuth Connection + Token) to use. This ID can be obtained from the <code>oauth-tokens</code> (/docs/enterprise/api/oauth-tokens.html) endpoint. |
| data.attributes.vcs-repo.branch | string | (previous value) | The repository branch that Terraform will execute from. |
| data.attributes.vcs-repo.ingress-submodules | boolean | (previous value) | Whether submodules should be fetched when cloning the VCS repository. |
| data.attributes.vcs-repo.identifier | string | (previous value) | A reference to your VCS repository in the format <code>:org/:repo</code> where <code>:org</code> and <code>:repo</code> refer to the organization and repository in your VCS provider. |

Sample Payload

```
{  
  "data": {  
    "attributes": {  
      "name": "workspace-2",  
      "terraform_version": "0.11.1",  
      "working_directory": "",  
      "vcs_repo": {  
        "identifier": "skierkowski/terraform-test-proj",  
        "branch": "",  
        "ingress_submodules": false,  
        "oauth_token_id": "ot-hmAyP66qk2AMVdbJ"  
      }  
    },  
    "type": "workspaces"  
  }  
}
```

Sample Request

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request PATCH \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/organizations/my-organization/workspaces/workspace-2
```

Sample Response

```
{
  "data": {
    "id": "ws-SihZTyXKfNXUWuUa",
    "type": "workspaces",
    "attributes": {
      "auto-apply": false,
      "can-queue-destroy-plan": false,
      "created-at": "2017-11-02T23:24:05.997Z",
      "environment": "default",
      "ingress-trigger-attributes": {
        "branch": "",
        "default-branch": true,
        "ingress-submodules": false
      },
      "locked": false,
      "name": "workspace-2",
      "queue-all-runs": false,
      "terraform-version": "0.10.8",
      "working-directory": ""
    },
    "relationships": {
      "organization": {
        "data": {
          "id": "my-organization",
          "type": "organizations"
        }
      },
      "ssh-key": {
        "data": null
      },
      "latest-run": {
        "data": null
      }
    },
    "links": {
      "self": "/api/v2/organizations/my-organization/workspaces/workspace-2"
    }
  }
}
```

List workspaces

This endpoint lists workspaces in the organization.

`GET /organizations/:organization_name/workspaces`

| Parameter | Description |
|---------------------------------|--|
| <code>:organization_name</code> | The name of the organization to create the workspace in. The organization must already exist in the system, and the user must have permissions to create new workspaces. |

Query Parameters

This endpoint supports pagination with standard URL query parameters ([/docs/enterprise/api/index.html#query-parameters](#)); remember to percent-encode [as %5B and] as %5D if your tooling doesn't automatically encode URLs.

| Parameter | Description |
|--------------|--|
| page[number] | Optional. If omitted, the endpoint will return the first page. |
| page[size] | Optional. If omitted, the endpoint will return 150 workspaces per page. |

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/organizations/my-organization/workspaces
```

Sample Response

```
{
  "data": [
    {
      "id": "ws-SihZTyXKfNXUWuUa",
      "type": "workspaces",
      "attributes": {
        "auto-apply": false,
        "can-queue-destroy-plan": false,
        "created-at": "2017-11-02T23:24:05.997Z",
        "environment": "default",
        "ingress-trigger-attributes": {
          "branch": "",
          "default-branch": true,
          "ingress-submodules": false
        },
        "locked": false,
        "name": "workspace-2",
        "queue-all-runs": false,
        "terraform-version": "0.10.8",
        "working-directory": ""
      },
      "relationships": {
        "organization": {
          "data": {
            "id": "my-organization",
            "type": "organizations"
          }
        },
        "ssh-key": {
          "data": null
        },
        "latest-run": {
          "data": null
        }
      },
      "links": {
        "self": "/api/v2/organizations/my-organization/workspaces/workspace-2"
      }
    },
    {
      "id": "ws-YnyXLq9fy38afEeb",
      "type": "workspaces",
      "attributes": {
        "auto-apply": false,
        "can-queue-destroy-plan": false,
        "created-at": "2017-11-02T23:24:05.997Z",
        "environment": "default",
        "ingress-trigger-attributes": {
          "branch": "",
          "default-branch": true,
          "ingress-submodules": false
        },
        "locked": false,
        "name": "workspace-2",
        "queue-all-runs": false,
        "terraform-version": "0.10.8",
        "working-directory": ""
      },
      "relationships": {
        "organization": {
          "data": {
            "id": "my-organization",
            "type": "organizations"
          }
        },
        "ssh-key": {
          "data": null
        },
        "latest-run": {
          "data": null
        }
      },
      "links": {
        "self": "/api/v2/organizations/my-organization/workspaces/workspace-2"
      }
    }
  ]
}
```

```

"attributes": {
    "auto-apply": false,
    "can-queue-destroy-plan": false,
    "created-at": "2017-11-02T23:23:53.765Z",
    "environment": "default",
    "ingress-trigger-attributes": {
        "branch": "",
        "default-branch": true,
        "ingress-submodules": false
    },
    "locked": false,
    "name": "workspace-1",
    "queue-all-runs": false,
    "terraform-version": "0.10.8",
    "working-directory": ""
},
"relationships": {
    "organization": {
        "data": {
            "id": "my-organization",
            "type": "organizations"
        }
    },
    "ssh-key": {
        "data": null
    },
    "latest-run": {
        "data": null
    }
},
"links": {
    "self": "/api/v2/organizations/my-organization/workspaces/workspace-1"
}
}
]
}

```

Show workspace

This endpoint shows details for a workspace in the organization.

`GET /organizations/:organization_name/workspaces/:name`

| Parameter | Description |
|---------------------------------|--|
| <code>:organization_name</code> | The name of the organization to create the workspace in. The organization must already exist in the system, and the user must have permissions to create new workspaces. |
| <code>:name</code> | The name of the workspace to show details for, which can only include letters, numbers, <code>-</code> , and <code>_</code> . |

Sample Request

```

$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
https://app.terraform.io/api/v2/organizations/my-organization/workspaces/workspace-1

```

Sample Response

```
{  
  "data": {  
    "id": "ws-mD5bmJ8ry3uTzuHi",  
    "type": "workspaces",  
    "attributes": {  
      "actions": {  
        "is-destroyable": true  
      },  
      "auto-apply": false,  
      "created-at": "2018-03-08T22:30:00.404Z",  
      "environment": "default",  
      "locked": false,  
      "name": "workspace-1",  
      "permissions": {  
        "can-update": true,  
        "can-destroy": true,  
        "can-queue-destroy": true,  
        "can-queue-run": true,  
        "can-update-variable": true,  
        "can-lock": true,  
        "can-read-settings": true  
      },  
      "queue-all-runs": false,  
      "terraform-version": "0.11.3",  
      "working-directory": null  
    },  
    "relationships": {  
      "organization": {  
        "data": {  
          "id": "my-organization",  
          "type": "organizations"  
        }  
      },  
      "latest-run": {  
        "data": null  
      },  
      "current-run": {  
        "data": null  
      }  
    },  
    "links": {  
      "self": "/api/v2/organizations/my-organization/workspaces/workspace-1"  
    }  
  }  
}
```

Delete a workspace

This endpoint deletes a workspace.

`DELETE /organizations/:organization_name/workspaces/:name`

| Parameter | Description |
|---------------------------------|--|
| <code>:organization_name</code> | The name of the organization to create the workspace in. The organization must already exist in the system, and the user must have permissions to create new workspaces. |

| Parameter | Description |
|-----------|---|
| :name | The name of the workspace to delete, which can only include letters, numbers, -, and _. |

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
https://app.terraform.io/api/v2/organizations/my-organization/workspaces/workspace-1
```

Lock a workspace

This endpoint locks a workspace.

POST /workspaces/:workspace_id/actions/lock

| Parameter | Description | |
|--|---|---|
| :workspace_id | The workspace ID to lock. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace endpoint. | |
| Status | Response | Reason(s) |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "workspaces") | Successfully locked the workspace |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace not found, or user unauthorized to perform action |
| 409
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace already locked |

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|----------|--------|---------|---------------------------------------|
| reason | string | "" | The reason for locking the workspace. |

Sample Payload

```
{  
  "reason": "Locking workspace-1"  
}
```

Sample Request

```
$ curl \  
  --header "Authorization: Bearer $TOKEN" \  
  --header "Content-Type: application/vnd.api+json" \  
  --request POST \  
  --data @payload.json \  
  https://app.terraform.io/api/v2/workspaces/ws-SihZTyXKfNXUWuUa/actions/lock
```

Sample Response

```
{
  "data": {
    "attributes": {
      "auto-apply": false,
      "can-queue-destroy-plan": false,
      "created-at": "2017-11-02T23:23:53.765Z",
      "environment": "default",
      "locked": true,
      "name": "workspace-2",
      "permissions": {
        "can-destroy": true,
        "can-lock": true,
        "can-queue-destroy": true,
        "can-queue-run": true,
        "can-read-settings": true,
        "can-update": true,
        "can-update-variable": true
      },
      "queue-all-runs": false,
      "terraform-version": "0.10.8",
      "vcs-repo": {
        "branch": "",
        "identifier": "my-organization/my-repository",
        "ingress-submodules": false,
        "oauth-token-id": "ot-hmAyP66qk2AMVdbJ"
      },
      "working-directory": null
    },
    "id": "ws-SihZTyXKfNXUWuUa",
    "relationships": {
      "locked-by": {
        "data": {
          "id": "my-user",
          "type": "users"
        },
        "links": {
          "related": "/api/v2/users/my-user"
        }
      }
    },
    "type": "workspaces"
  }
}
```

Unlock a workspace

This endpoint unlocks a workspace.

`POST /workspaces/:workspace_id/actions/unlock`

| Parameter | Description |
|----------------------------|---|
| <code>:workspace_id</code> | The workspace ID to unlock. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace endpoint. |

| Status | Response | Reason(s) |
|--|---|---|
| 200
(Status/200">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "workspaces") | Successfully unlocked the workspace |
| 404
(Status/404">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace not found, or user unauthorized to perform action |
| 409
(Status/409">https://developer.mozilla.org/en-US/docs/Web/HTTP>Status/409) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace already unlocked, or locked by a different user |

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
https://app.terraform.io/api/v2/workspaces/ws-SihZTyXKfNXUWuUa/actions/unlock
```

Sample Response

```
{
  "data": {
    "attributes": {
      "auto-apply": false,
      "can-queue-destroy-plan": false,
      "created-at": "2017-11-02T23:23:53.765Z",
      "environment": "default",
      "locked": false,
      "name": "workspace-2",
      "permissions": {
        "can-destroy": true,
        "can-lock": true,
        "can-queue-destroy": true,
        "can-queue-run": true,
        "can-read-settings": true,
        "can-update": true,
        "can-update-variable": true
      },
      "queue-all-runs": false,
      "terraform-version": "0.10.8",
      "vcs-repo": {
        "branch": "",
        "identifier": "my-organization/my-repository",
        "ingress-submodules": false,
        "oauth-token-id": "ot-hmAyP66qk2AMVdbJ"
      },
      "working-directory": null
    },
    "id": "ws-SihZTyXKfNXUWuUa",
    "type": "workspaces"
  }
}
```

Force Unlock a workspace

This endpoint force unlocks a workspace. Only users with admin access are authorized to force unlock a workspace.

`POST /workspaces/:workspace_id/actions/force-unlock`

| Parameter | Description | |
|--|---|---|
| <code>:workspace_id</code> | The workspace ID to force unlock. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace endpoint. | |
| Status | Response | Reason(s) |
| 200
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) | JSON API document
(https://www.terraform.io/docs/enterprise/api/index.html#json-api-documents) (type: "workspaces") | Successfully force unlocked the workspace |
| 404
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace not found, or user unauthorized to perform action |
| 409
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409) | JSON API error object (http://jsonapi.org/format/#error-objects) | Workspace already unlocked |

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request POST \
https://app.terraform.io/api/v2/workspaces/ws-SihZTyXKfNXUWuUs/actions/force-unlock
```

Sample Response

```
{
  "data": {
    "attributes": {
      "auto-apply": false,
      "can-queue-destroy-plan": false,
      "created-at": "2017-11-02T23:23:53.765Z",
      "environment": "default",
      "locked": false,
      "name": "workspace-2",
      "permissions": {
        "can-destroy": true,
        "can-lock": true,
        "can-queue-destroy": true,
        "can-queue-run": true,
        "can-read-settings": true,
        "can-update": true,
        "can-update-variable": true
      },
      "queue-all-runs": false,
      "terraform-version": "0.10.8",
      "vcs-repo": {
        "branch": "",
        "identifier": "my-organization/my-repository",
        "ingress-submodules": false,
        "oauth-token-id": "ot-hmAyP66qk2AMVdbJ"
      },
      "working-directory": null
    },
    "id": "ws-SihZTyXKfNXUWuUs",
    "type": "workspaces"
  }
}
```

Assign an SSH key to a workspace

This endpoint assigns an SSH key to a workspace.

PATCH /workspaces/:workspace_id/relationships/ssh-key

| Parameter | Description |
|-----------|-------------|
|-----------|-------------|

| | |
|---------------|--|
| :workspace_id | The workspace ID to assign the SSH key to. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace endpoint. |
|---------------|--|

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------|--------|---------|--|
| data.type | string | | Must be "workspaces". |
| data.attributes.id | string | | The SSH key ID to assign. Obtain this from the ssh-keys (/docs/enterprise/api/ssh-keys.html) endpoint. |

Sample Payload

```
{
  "data": {
    "attributes": {
      "id": "sshkey-GxrePWre1Ezug7aM"
    },
    "type": "workspaces"
  }
}
```

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/workspaces/ws-SihZTyXKfNXUWuUa/relationships/ssh-key
```

Sample Response

```
{
  "data": {
    "attributes": {
      "auto-apply": false,
      "can-queue-destroy-plan": false,
      "created-at": "2017-11-02T23:24:05.997Z",
      "environment": "default",
      "ingress-trigger-attributes": {
        "branch": "",
        "default-branch": true,
        "ingress-submodules": false
      },
      "locked": false,
      "name": "workspace-2",
      "queue-all-runs": false,
      "terraform-version": "0.10.8",
      "working-directory": ""
    },
    "id": "ws-SihZTyXKfNXUWuUa",
    "links": {
      "self": "/api/v2/organizations/my-organization/workspaces/workspace-2"
    },
    "relationships": {
      "latest-run": {
        "data": null
      },
      "organization": {
        "data": {
          "id": "my-organization",
          "type": "organizations"
        }
      },
      "ssh-key": {
        "data": {
          "id": "sshkey-GxrePWre1Ezug7aM",
          "type": "ssh-keys"
        },
        "links": {
          "related": "/api/v2/ssh-keys/sshkey-GxrePWre1Ezug7aM"
        }
      }
    },
    "type": "workspaces"
  }
}
```

Unassign an SSH key from a workspace

This endpoint unassigns the currently assigned SSH key from a workspace.

PATCH /workspaces/:workspace_id/relationships/ssh-key

| Parameter | Description |
|-----------|-------------|
|-----------|-------------|

| | |
|---------------|--|
| :workspace_id | The workspace ID to assign the SSH key to. Obtain this from the workspace settings (/docs/enterprise/workspaces/settings.html) or the Show Workspace endpoint. |
|---------------|--|

Request Body

This POST endpoint requires a JSON object with the following properties as a request payload.

Properties without a default value are required.

| Key path | Type | Default | Description |
|--------------------|--------|---------|-----------------------|
| data.type | string | | Must be "workspaces". |
| data.attributes.id | string | | Must be null. |

Sample Payload

```
{
  "data": {
    "attributes": {
      "id": null
    },
    "type": "workspaces"
  }
}
```

Sample Request

```
$ curl \
--header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request PATCH \
--data @payload.json \
https://app.terraform.io/api/v2/workspaces/ws-SihZTyXKfNXUWuUa/relationships/ssh-key
```

Sample Response

```
{
  "data": {
    "attributes": {
      "auto-apply": false,
      "can-queue-destroy-plan": false,
      "created-at": "2017-11-02T23:24:05.997Z",
      "environment": "default",
      "ingress-trigger-attributes": {
        "branch": "",
        "default-branch": true,
        "ingress-submodules": false
      },
      "locked": false,
      "name": "workspace-2",
      "queue-all-runs": false,
      "terraform-version": "0.10.8",
      "working-directory": ""
    },
    "id": "ws-erEAnPmgm5Jr77",
    "links": {
      "self": "/api/v2/organizations/my-organization/workspaces/workspace-2"
    },
    "relationships": {
      "latest-run": {
        "data": null
      },
      "organization": {
        "data": {
          "id": "my-organization",
          "type": "organizations"
        }
      },
      "ssh-key": {
        "data": null
      }
    },
    "type": "workspaces"
  }
}
```

Available Related Resources

The GET endpoints above can optionally return related resources, if requested with the `include` query parameter ([/docs/enterprise/api/index.html#inclusion-of-related-resources](#)). The following resource types are available:

- `organization` - The full organization record.
- `current_run` - Additional information about the current run.
- `current_run.plan` - The plan used in the current run.
- `current_run.configuration_version` - The configuration used in the current run.
- `current_run.configuration_version.ingress_attributes` - The commit information used in the current run.

Getting Started with Terraform Enterprise

Welcome to Terraform Enterprise!

Terraform Enterprise helps teams collaborate on infrastructure. It combines a predictable and reliable shared run environment with tools to help you work together on Terraform configurations and modules. To learn more about its features and functionality, and to request a free trial, see the Terraform Enterprise product page (<https://www.hashicorp.com/products/terraform>).

This getting started guide introduces Terraform Enterprise's core workflows and UI, and walks you through their basic usage. The first page is about how to access Terraform Enterprise. (/docs/enterprise/getting-started/access.html)

Accessing Terraform Enterprise

To start using Terraform Enterprise (TFE), you must:

- Make sure you have a TFE account.
- Contact HashiCorp sales to request access to TFE.
- Navigate to TFE.
- Create a new organization for working with TFE.

Creating a TFE Account

If you don't already have a TFE account, you must create one. You can use the same user account to access the new TFE and the previous legacy version of TFE, although you'll use separate organizations.

Click here to request a free trial of TFE (https://www.hashicorp.com/products/terraform/?utm_source=oss&utm_medium=header-nav&utm_campaign=terraform&_ga=2.40850658.1512399790.1504740058-931972891.1498668200#terraform-contact-form), or contact HashiCorp sales to purchase a TFE subscription.

Navigating to TFE

In your web browser, go to:

- app.terraform.io (<https://app.terraform.io>) if you've registered for the SaaS version of TFE.
- The hostname of your private instance if you're using private Terraform Enterprise.

Creating an Organization

Note: If someone else has already created a TFE organization and added you to it, you can skip this process. You'll be taken to the organization's front page when you first navigate to TFE.

After you've navigated to TFE, it will prompt you to create a new organization. Enter a name (distinct from your legacy TFE organization, if any) and an admin email address at the prompt:



New Organization

ORGANIZATION NAME name

This will be part of your resource names used in various tools, i.e. `hashicorp/www-prod`.

EMAIL ADDRESS email

The organization email is used for any future notifications, such as billing, and the organization avatar, via [gravatar.com](#).

Create organization

Adding Other Users to an Organization

To collaborate with your colleagues in TFE, you'll all need access to the same TFE organization. You can add users to an organization by creating a *team* and adding users to it.

First, navigate to the settings page for your organization — you can reach it from the "Settings" link found at the top of every page. Once there, click the "Teams" link in the sidebar navigation.

The list of teams starts with just one team, named "owners." Don't add users to this team yet; instead, enter a new team name (like "core-infrastructure") in the text field and click the "Create team" button. This will take you to the new team's settings page:

ORGANIZATION SETTINGS

nicktech

Profile

Teams

VCS Providers

Integrations

API Tokens

Authentication

Manage SSH Keys

Sentinel Policies

Team: test_team

Add a New Team Member

USERNAME

[Add member](#)

The username of the user you wish to add to this team. Share the signup link with new users:

<https://app.terraform.io/account/new> 

Members (3)



kfishner

2FA



nfagerlund

2FA



skierkowski

2FA



Team API Token

Treat this token like a password, as it can be used to access your account without a username, password, or two-factor authentication.

Last used **4 months ago**

Created **4 months ago** by user **nfagerlund**

[Regenerate token](#)[Delete token](#)

Delete this Team

Warning! This will permanently delete this team and any permissions associated with it.

[Delete test_team](#)

Add as many users as you'd like by typing their TFE username in the text field and clicking "Add member". Added users won't receive a notification, but your organization will be available the next time they access TFE.

Team membership is how TFE controls access to workspaces. Later, you can create more teams and assign them different permissions on a per-workspace basis. For more information, see Teams (/docs/enterprise/users-teams-organizations/teams.html).

Next Steps

After you've created a TFE organization, you should configure version control access. (/docs/enterprise/getting-started/vcs.html)

Creating and Managing Terraform Policies

Prerequisites: Before starting this guide, make sure you've successfully completed a run (</docs/enterprise/getting-started/runs.html>).

About Sentinel Policies

Policies in TFE are composed of Sentinel policies (</docs/enterprise/sentinel/index.html>) with some extra features. Sentinel (<https://www.hashicorp.com/sentinel>) is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products. It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources. Within TFE, you can use Sentinel to apply checks to your runs.

A policy consists of:

- The Sentinel policy code
- An enforcement mode that changes how a policy affects the run lifecycle

Currently, policies in an organization apply to **all** workspaces. You'll want to consider that when writing your policies so they don't cause unintended failures. In the future, TFE will provide a way to manage policies so you can apply them per-workspace.

Creating a Policy

First, make sure you're viewing the organization settings. If you're still on a run page (or any other page), click the "Settings" button in the top navigation bar.

The screenshot shows the HashiCorp Enterprise interface. At the top, there's a blue navigation bar with the HashiCorp logo, the workspace name "hashicorp-v2", and dropdown menus for "Workspaces", "Modules", and "Settings". The "Settings" button is highlighted with a red circle. Below the navigation bar, the page title is "Organization Profile". On the left, there's a sidebar with "ORGANIZATION SETTINGS" and a user profile section labeled "hashicorp-v2". The main content area is currently empty, displaying the text "The policy list shows all of the policies you have access to; if you haven't created any, it's empty." and a note about creating a new policy.

The policy list shows all of the policies you have access to; if you haven't created any, it's empty.

To create your first policy, click the "Create new policy" button in the upper right.

The screenshot shows the Hashicorp Enterprise UI interface. At the top, there's a navigation bar with links for 'hashicorp-v2', 'Workspaces', 'Modules', 'Settings', 'Documentation', 'Status', and a user icon. On the left, a sidebar titled 'ORGANIZATION SETTINGS' lists various options like Profile, Teams, VCS Providers, Integrations, API Tokens, Authentication, Manage SSH Keys, and 'Sentinel Policies'. The 'Sentinel Policies' link is highlighted with a blue background. The main content area is titled 'Sentinel Policies' and contains a brief description of what Sentinel Policies are. Below the description, it says 'You do not currently have any Sentinel policies configured for your organization.' In the top right corner of the main content area, there's a blue button labeled 'Create new policy' which is circled in red.

On the "Create a new Sentinel Policy" page, you need to enter at least two items: a policy name, and the policy code. When you've finished, click the "Create Policy" button.

The screenshot shows the 'Create new Sentinel Policy' page. At the top, there's a navigation bar with links for 'hashicorp-v2', 'Workspaces', 'Modules', 'Settings', 'Documentation', 'Status', and a user icon. On the left, a sidebar titled 'ORGANIZATION SETTINGS' lists various options like Profile, Teams, VCS Providers, Integrations, API Tokens, Authentication, Manage SSH Keys, and 'Sentinel Policies'. The 'Sentinel Policies' link is highlighted with a blue background. The main content area is titled 'Create new Sentinel Policy'. It includes a callout box with the following text: 'Heads up: Sentinel is designed to enable **policy as code**. As such, it is not recommended to use only the Terraform Enterprise UI to manage policy. However, this interface can serve to demonstrate or manage very simple Sentinel policies. Consider integrating with the Terraform Enterprise API in CI to test and upload policy files. In the future, Terraform Enterprise will integrate directly with VCS providers for the Sentinel workflow.' Below the callout, there's a note stating 'Policy will be enforced on every workspace run to validate the terraform plan and corresponding resources are in compliance with company policies.' There are two input fields: 'POLICY NAME' containing 'passthrough' and 'ENFORCEMENT MODE' set to 'soft-mandatory (can override)'. Below these, a note says 'Only users on the owners team in this organization can override `soft-mandatory` checks.' At the bottom, there's a code editor for 'POLICY CODE' containing the following text:

```
1 main = rule { true }
```

 and a blue 'Create policy' button.

Policy Name

A policy name should tell your colleagues what the policy is for. Examples could be "Require tags on all instances" or "Enforce network ACLs". If a policy is for a particular workspace, or environment within a workspace, include the name of the workspace and/or environment.

For this example, we'll just create a sample passthrough policy that will allow all runs to "PASS" our policy check.

In this example, we're using a configuration named "minimum" and we're deploying it in a production environment, so we named it passthrough.

Enforcement Mode

Enforcement Mode alters how a policy result affects your run; "hard-mandatory" will always stop a run if a policy fails, "soft-mandatory" will pause a run and allow a failure to be overridden, and "advisory" will log failures but always allow a run to continue. Use "soft-mandatory" for now so we can see how the "Policy Override" feature works in practice.

Policy Code

You can paste the following code into the code input box. It always resolves to `true` and will allow all runs to pass. This is a good way to see policy checks applied to your run without having a policy "FAIL".

```
main = rule { true }
```

Later, you can switch `true` to `false` to see how a "FAIL" during a policy check can affect your run. You can find many more examples in our Example Policies section (</docs/enterprise/sentinel/examples.html>).

What Happens in a New Policy

When you create a new policy, it will be applied to all future runs. Runs that are currently queued or in progress will not be affected.

See a Policy Check in a Run

Once your policy is created, you can view its effect on the run page. Start a new run to watch it play out. Again, you can start a run with the "Queue Plan" button at the upper right of the workspace page, or using the "Save & Plan" button when editing the workspace's variables.

Your policy code will be applied to this run and you'll see a new policy check section in the run's timeline. Expand the section and you should see that our new "passthrough" policy has been run with a "true" result, allowing the run to continue.

 Policy check passed 5 days ago Policies: 1 passed, 0 failed ^

Queued 5 days ago > Passed 5 days ago

[View raw log](#)

Top Bottom Expand Full Screen

```
Sentinel Result: true

This result means that Sentinel policies returned true and the protected behavior is allowed by Sentinel policies.

1 policies evaluated.

## Policy 1: a-test.sentinel (hard-mandatory)

Result: true

TRUE - a-test.sentinel:1:1 - Rule "main"
```

If you change your policy code to `true false`, the "soft-mandatory" option will allow you to override at this stage and continue to the apply stage of the run.

Finished

You've now configured TFE and experienced its core workflows — you know how to create new workspaces, automatically and manually trigger runs on a workspace, review and monitor runs, approve plans, and add policy checks to your runs.

Running Terraform in TFE

Prerequisites: Before starting this guide, make sure you've created and configured at least one workspace (</docs/enterprise/getting-started/workspaces.html>).

Once you've created and configured a workspace, you can manage infrastructure resources by doing Terraform runs in that workspace.

About Terraform Runs in TFE

TFE enforces Terraform's division between *plan* and *apply* operations, and by default it waits for user approval before applying new plans. You can enable automatic applies in a workspace's settings.

Each workspace has its own queue of runs, which are processed one at a time in order. When a run is queued, it grabs the current version of Terraform code and the current variable values, and will use those when it eventually runs; if you make more commits or modify variables before the run occurs, it will still use the data it was originally queued with.

Most runs in TFE are started automatically — whenever new commits are added to a workspace's VCS branch, it queues a new plan. You can also manually queue plans, usually after editing variables.

Note that TFE needs network access to any infrastructure providers managed by your Terraform configurations. If you're using the SaaS version of TFE and you use Terraform to manage private cloud resources, those providers must be internet accessible.

For more details, see the following pages:

- About Terraform Runs in Terraform Enterprise (</docs/enterprise/run/index.html>)
- Run States and Stages (</docs/enterprise/run/states.html>)
- The UI- and VCS-driven Run Workflow (</docs/enterprise/run/ui.html>)
- The API-driven Run Workflow (</docs/enterprise/run/api.html>)
- CLI-driven run workflow (</docs/enterprise/run/cli.html>)

Starting a Run from Version Control

To queue a plan for your new workspace, make a new commit to the VCS branch you chose when creating the workspace. In the previous page's example, we chose the `nfagerlund/terraform-minimum` repo and left it on the default branch, so we could start a run by pushing one or more new commits to the `master` branch of that repo.

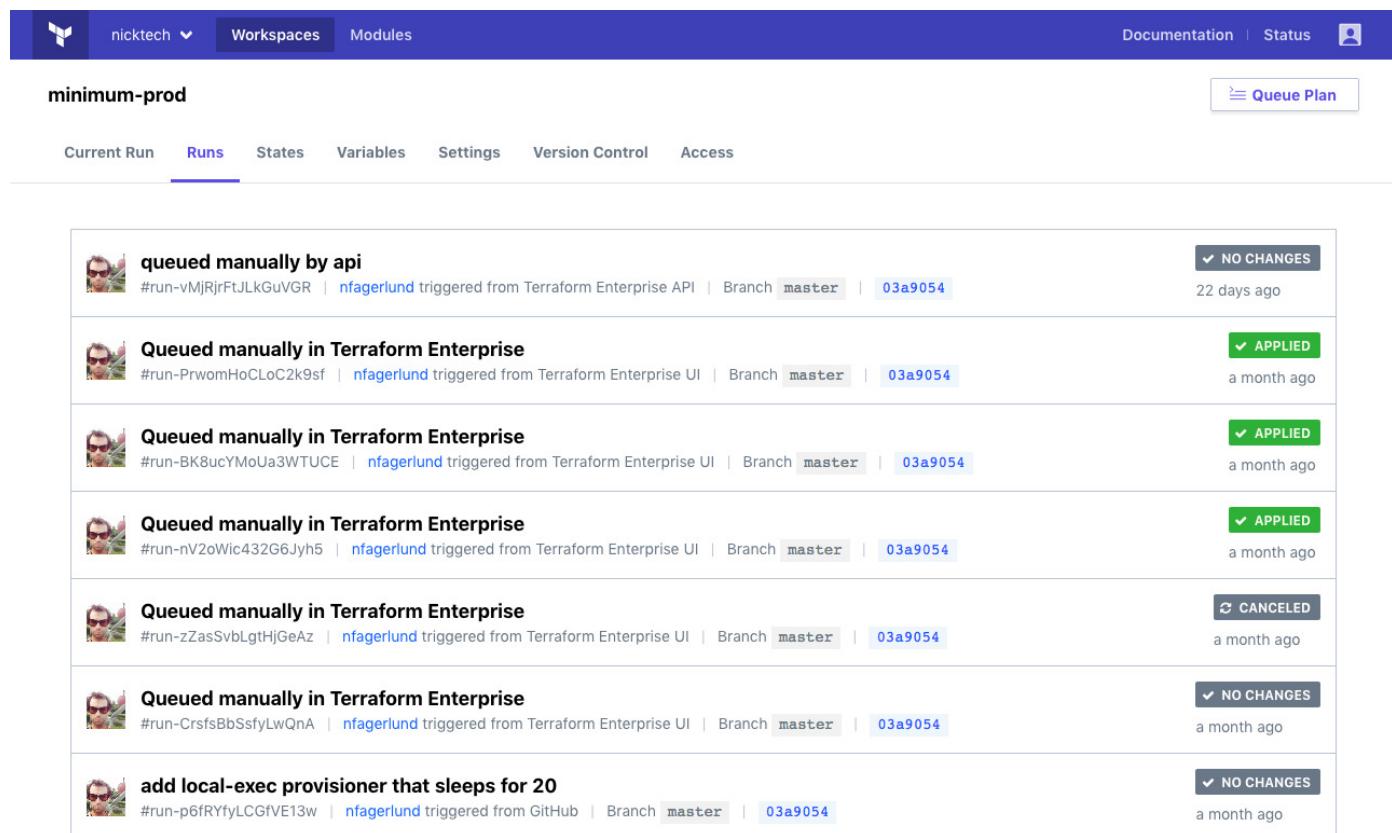
Starting a Run from the UI

Before continuing with your first run, queue another plan using TFE's UI. You can do this with the "Queue Plan" at the upper right of the workspace page, or using the "Save & Plan" button when editing the workspace's variables.

The best way to see how TFE's run queue works is to make changes before queueing a second run. If your workspace's Terraform configuration uses variables, change one of them and queue a plan. Alternately, you could make changes to the code and push new commits to VCS, which would also put a new plan in the queue.

Viewing the Run List

At the top of the workspace's page, click the "Runs" link, which goes to the full list of runs.



The screenshot shows the 'Runs' tab selected in the navigation bar. A button labeled 'Queue Plan' is visible in the top right. Below the header, there are tabs for 'Current Run', 'Runs' (selected), 'States', 'Variables', 'Settings', 'Version Control', and 'Access'. The main area displays a list of seven runs:

| Run ID | Description | Status | Time Ago |
|---------|---|--------------|-------------|
| 03a9054 | queued manually by api | ✓ NO CHANGES | 22 days ago |
| 03a9054 | Queued manually in Terraform Enterprise | ✓ APPLIED | a month ago |
| 03a9054 | Queued manually in Terraform Enterprise | ✓ APPLIED | a month ago |
| 03a9054 | Queued manually in Terraform Enterprise | ✓ APPLIED | a month ago |
| 03a9054 | Queued manually in Terraform Enterprise | ✖ CANCELED | a month ago |
| 03a9054 | Queued manually in Terraform Enterprise | ✓ NO CHANGES | a month ago |
| 03a9054 | add local-exec provisioner that sleeps for 20 | ✓ NO CHANGES | a month ago |

From the list of runs, you can click to view or interact with an individual run. Choose the run you started first, which will probably be in the "Pending," "Planning," or "Planned" state. Note that your second run is "Pending."

Viewing a Run Page

An individual run page shows the progress and outcomes of each stage of the run.

minimum-prod

 Queue Plan 

Current Run  Runs States Variables Settings Integrations Version Control Access

✓ APPLIED Add empty `random` provider block for resource destruction

 nfagerlund triggered a run from GitHub 9 months ago 

 **Plan finished** 9 months ago 

 **Apply finished** 9 months ago 

 nfagerlund 9 months ago
let's try this again!

 Run confirmed

Comment: Leave feedback or record a decision.

Add Comment



Support Terms Privacy Security © 2018 HashiCorp, Inc.

Once this run has finished planning, it will ask you to confirm or discard the plan. Click the "Confirm & Apply" button to finish applying the changes from your new commits.

 **Plan finished** 34 minutes ago Resources: 1 to add, 0 to change, 1 to destroy 

Queued 34 minutes ago > Started 34 minutes ago > Finished 34 minutes ago

 View raw log  Top  Bottom  Expand  Full Screen

```
-/+ destroy and then create replacement
Terraform will perform the following actions:

-/+ random_id.random (new resource required)
  id:      "3zkmEftYik8" => <computed> (forces new resource)
  b64:    "3zkmEftYik8" => <computed>
  b64_std: "3zkmEftYik8" => <computed>
  b64_url: "3zkmEftYik8" => <computed>
  byte_length: "8" => "8"
  dec:      "16084929395824298575" => <computed>
  hex:     "df3926105b588af" => <computed>
  keepers.%: "1" => "1"
  keepers.uuid: "b5bfe2b7-ca6e-c0cd-7a01-b659aac398f7" => "f87d124d-9155-90d1-c994-a14f4fffc8adf" (forces new resource)

Plan: 1 to add, 0 to change, 1 to destroy.
```

 **Apply pending**

Needs Confirmation: Check the plan and confirm to apply it, or discard the run.

 Confirm & Apply  Discard Run  Add Comment

Note that until you apply or discard a plan, TFE can't start another run in that workspace, and your second run will stay "Pending."

After the first run has finished applying, you can go back to the runs list and see the second run starting.

Next Steps

Now that you've completed and viewed a run, you're ready to start applying policy checks ([/docs/enterprise/getting-started/policies.html](#)).

Configuring Version Control Access

Prerequisites: At this point, you should have gotten access to Terraform Enterprise (/docs/enterprise/getting-started/access.html), and created an organization if necessary.

Before you can use TFE, it needs access to the version control system (VCS) you use for Terraform code.

About VCS Access

Most workspaces in TFE are associated with a VCS repository, which provides Terraform configurations for that workspace. To find out which repos are available, access their contents, and create webhooks, TFE needs access to your VCS service.

Although TFE's API lets you create workspaces and push configurations to them without a VCS connection, the primary workflow expects every workspace to be backed by a repository. If you plan to use TFE's GUI to create workspaces, you must configure VCS access first.

Configuring VCS Access

In general, you enable VCS access by creating a new application configuration on your VCS service, telling TFE how to reach your VCS, exchanging some secret information between them, and requesting access.

Each supported VCS service has slightly different instructions for this. Open the VCS Integrations page (/docs/enterprise/vcs/index.html) in a new tab, select your VCS, and follow the instructions to connect it. When you've finished, continue to the next page in this guide.

Next Steps

After you've configured VCS access, you can start creating workspaces (/docs/enterprise/getting-started/workspaces.html).

Creating and Managing Terraform Workspaces

Prerequisites: Before starting this guide, you should get access to Terraform Enterprise (/docs/enterprise/getting-started/access.html) and configure VCS access (/docs/enterprise/getting-started/vcs.html).

At this point, you've done all the setup TFE needs, and can start using it for real work. Your first task should be to set up some workspaces.

About Workspaces

Workspaces are how TFE organizes infrastructure. If you've used the legacy version of TFE, workspaces used to be called environments.

A workspace consists of:

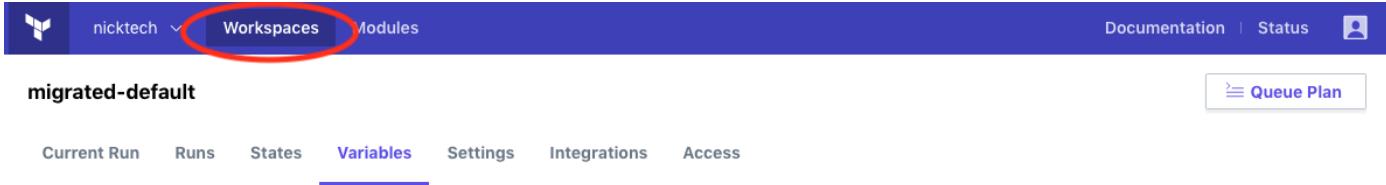
- A collection of Terraform configurations (retrieved from a VCS repo).
- Values for any variables those configurations require.
- Persistent stored state for the resources it manages.
- Historical state and run logs.

A well-designed Terraform workflow uses multiple configurations, so you can manage each logical grouping of infrastructure with its own code. Additionally, it's common to use the same configuration multiple times (with different values for variables) to manage different environments.

Creating a Workspace

Note: Only members of the "owners" team can create new workspaces.

First, make sure you're viewing the workspace list. If you're still on the VCS settings (or any other page), click the "Workspaces" button in the top navigation bar.



The screenshot shows the Terraform Enterprise dashboard. At the top, there's a dark blue header with the user icon, the username 'nicktech', and a dropdown arrow. To the right of the dropdown are the 'Documentation' and 'Status' links, and a user profile icon. Below the header is a navigation bar with tabs: 'Workspaces' (which is highlighted and circled in red), 'Modules', 'migrated-default' (the active workspace name), 'Queue Plan' (with a small icon), and 'Current Run', 'Runs', 'States', 'Variables' (which is underlined in blue), 'Settings', 'Integrations', and 'Access'. The main content area below the navigation bar is currently empty, indicating no workspaces have been created yet.

The workspace list shows all of the workspaces you have access to; if you haven't created any, it's empty.

To create your first workspace, click the "+ New Workspace" button in the upper right.

| Workspaces 10 total | | | | |
|---------------------|-------------|---------------|---------------------|------------------------------|
| All (10) | Success (2) | Error (0) | Needs Attention (7) | Running (0) |
| WORKSPACE NAME | RUN STATUS | LATEST CHANGE | RUN | REPO |
| migrated-default | NO CHANGES▼ | a month ago | run-Skof | nfagerlund/terraform-minimum |
| migrated-second | NO CHANGES▼ | a month ago | run-EP9C | nfagerlund/terraform-minimum |
| migrated-first | NO CHANGES▼ | a month ago | run-8kaS | nfagerlund/terraform-minimum |

On the "Create a new Workspace" page, you need to enter at least two items: a workspace name, and a VCS repository. (You can search for a repo by typing part of its name.) When you've finished, click the "Create Workspace" button.

This workspace will be created under the current organization, **nicktech**.

WORKSPACE NAME
e.g. workspace-name

The name of your workspace is unique and used in tools, routing, and UI. Dashes, underscores, and alphanumeric characters are permitted. Learn more about [naming workspaces](#).

SOURCE

None Bitbucket Server Bitbucket Cloud GitLab.com
 GitHub +

REPOSITORY
e.g. organization/repository-name

The repository identifier in the format `username/repository`. Only the most recently updated repositories will appear with autocomplete; however, all repositories are available for use.

More options (working directory, VCS branch, ingress submodules)

Create Workspace **Cancel**

Note: You can also create workspaces without a VCS repository, but doing so requires a different workflow for Terraform runs. For more information, see [About Terraform Runs in Terraform Enterprise](#) (/docs/enterprise/run/index.html).

Workspace Name

A workspace name should tell your colleagues what the workspace is for. Most workspaces are *a particular environment of a particular Terraform configuration*, so the name should include both the name of the configuration and the name of the

environment.

In this example, we're using a configuration named "minimum" and we're deploying it in a production environment, so we named it `minimum-prod`.

Repository

The "repository" field is linked to data from your VCS service. You can start typing the name of a repo, then select the correct repo from the resulting list.

In this example, we're using the `nfagerlund/terraform-minimum` repo.

Other Settings

Optionally, you can set three other settings for a new workspace:

- **Terraform Working Directory:** The directory in this repo where TFE will execute Terraform commands. If the Terraform configuration files you want aren't in the root of the repository, use this to specify where they are.
- **VCS Branch:** Which branch to use. Defaults to the repo's default branch, which is usually `master`.

What Happens in a New Workspace

When you create a new workspace, a few things happen:

- TFE *doesn't* immediately queue a plan for the workspace. Instead, it presents you with a dialog, with shortcut links to either queue a plan or edit variables.

A screenshot of the Terraform Workspaces interface. At the top, there is a navigation bar with tabs: Current Run, Runs, States, Variables, Settings, Version Control, and Access. The 'Runs' tab is currently selected, indicated by a blue underline. Below the navigation bar, a success message is displayed: **Configuration uploaded successfully**. A small green checkmark icon is to the left of the message. Below the message, a note reads: "Your configuration has been uploaded. Next, you probably want to configure variables (such as access keys or configuration values). If your configuration doesn't require variables, you can queue your first plan now." At the bottom of the screen, there are two buttons: "Configure Variables" (in blue) and "Queue Plan".

- TFE automatically registers a webhook with your VCS service. The next time new commits appear in the selected branch of that repo, TFE will automatically queue a Terraform plan for the workspace.

Editing Variables

For almost any workspace, you'll want to edit the Terraform variables and environment variables used by the code.

You can edit variables as soon as you've created a workspace, by clicking the workspace's "Variables" tab.

minimum-prod

 Queue PlanCurrent Run Runs States **Variables** Settings Integrations Version Control Access

Variables

These variables are used for all plans and applies in this workspace. Workspaces using Terraform 0.10.0 or later can also load default values from any `*.auto.tfvars` files in the configuration.

Sensitive variables are hidden from view in the UI and API, and can't be edited. (To change a sensitive variable, delete and replace it.) Sensitive variables can still appear in Terraform logs if your configuration is designed to output them.

When setting many variables at once, the [TFE CLI tool's](#) `pushvars` command or the [variables API](#) can often save time.

Terraform Variables

These [Terraform variables](#) are set using a `terraform.tfvars` file. To use interpolation or set a non-string value for a variable, click its HCL checkbox.

| | | |
|--------------------------------|----------------------|---|
| username | service-deploy |   |
| prior_workspace | nicktech/minimum-dev |   |
| + Add Variable | | |

Environment Variables

These variables are set in Terraform's shell environment using `export`.

| | | |
|--------------------------------|------------------------|---|
| EXAMPLE_PROVIDER_ACCESS_TOKEN | sensitive - write only |   |
| + Add Variable | | |

The variables page has two sections: Terraform variables (as declared in your Terraform configurations) and environment variables.

Click the "+ Add Variable" button to create a new variable, or click an existing variable's text fields or pencil icon to change its values and settings. After editing a variable, be sure to click its "Save Variable" button.

Terraform Variables

These [Terraform variables](#) are set using a `terraform.tfvars` file. To use interpolation or set a non-string value for a variable, click its HCL checkbox.

username HCL Sensitive

Save Variable Cancel

prior_workspace nicktech/minimum-dev

+ Add Variable

Environment Variables

These variables are set in Terraform's shell environment using `export`.

EXAMPLE_PROVIDER_ACCESS_TOKEN sensitive - write only

+ Add Variable

Terraform variables start as basic string values, but you can also enter array or map values if you click the "HCL" checkbox for that variable. You can write these values with the same syntax you'd use in a Terraform configuration.

For more information, see [Variables](#) (/docs/enterprise/workspaces/variables.html).

Granting Access

The user account that created your organization is part of the "owners" team, which can do any action on any workspace. Most of your colleagues don't need that level of access; instead, an administrator can give them access to the specific workspaces they need to use.

TFE manages access with *teams*. In your organization's settings, you can create any number of teams and add users to them — earlier in this getting started guide, you might have already created a team (/docs/enterprise/getting-started/access.html#adding-other-users-to-an-organization) to bring other users into your organization. If you haven't done that yet, do it now.

Once you have some teams, you can add them to a workspace by clicking the "Access" tab:

The screenshot shows the 'Access' tab selected in the navigation bar. The 'Add a team' section contains a 'TEAM NAME' dropdown set to 'owners' and a 'PERMISSIONS' dropdown set to 'read'. Below these are descriptive text labels and a 'Add team' button. The 'Teams' section lists 'Owners of nicktech' with a 'DEFAULT' permission level and 'test_team (admin)' with a 'Remove' button.

A newly created workspace can't be used by anyone but the "owners" team; other teams can't even see it. To enable collaboration on a workspace, you must add other teams to it.

To add a team to a workspace, select it from the dropdown menu and choose which permissions the team should have. There are three levels of permissions available:

- **Read** access lets team members view the workspace's variables and settings, view its run history, and view its StateVersions and ConfigurationVersions. They can't queue run plans, apply existing run plans, or change any variables or settings.
- **Write** access lets team members create and approve runs, and lock or unlock the workspace. It doesn't let them change variables or settings.
- **Admin** access lets team members change variables and settings, add other teams to the workspace, remove other teams from the workspace, and delete the workspace.

To change a team's permissions on a workspace, you must first delete their existing permissions, then add their new permissions.

Note: These permissions only affect actions via TFE's UI and API. If a person can push commits to a workspace's VCS repo and the workspace is set to automatically apply changes, that person can still make infrastructure changes. Take this into account when planning your teams and workspace permissions.

Configuring a Workspace

Each workspace has a "Settings" page (available from the top links when viewing that workspace), where you can change the behavior of the workspace. Currently, this page has the following settings:

- **Auto apply vs. manual apply:** By default, TFE only performs a Terraform plan when you start a run (either manually or by adding new commits to the repo). You can then view the outcome of the plan and decide whether to go forward with applying it.

This is called "manual apply." If you'd rather have TFE automatically apply successful plans, you can switch to "auto apply."

- **Terraform version:** TFE can use any released version of Terraform to manage a workspace. Different workspaces can use different versions, and TFE won't automatically upgrade a workspace.

To stay up to date, you should visit workspace settings periodically and update the Terraform version.

- **Workspace lock:** A user with write privileges on a workspace can *lock* the workspace, which prevents plans and applies from running. Use this when recovering from a bad commit or performing other maintenance.
- **Workspace delete:** When deleting a workspace, you usually also want to de-provision any infrastructure it's managing. This section of the settings has a button for queueing a destroy plan (to de-provision), and a button for deleting the workspace data. Note that destroy plans require an extra environment variable for confirmation; see the settings page for more details.

Navigating Workspaces

Most of your time in TFE is spent in two views:

- The workspace list. Use this to get an overview of the workspaces you're responsible for, and to navigate between workspaces.

| WORKSPACE NAME | RUN STATUS | LATEST CHANGE | RUN | REPO |
|------------------|----------------------|---------------|----------|-------------------------------|
| exceed-limit | ✓ APPLIED | 5 months ago | run-B8Ac | NICKF/terraform-minimum |
| filetest-dev | ✗ ERRORED | 3 months ago | run-SLSz | nfagerlund/terraform-filetest |
| migrated-default | ✓ PLANNED | 5 months ago | run-BVj | nfagerlund/terraform-minimum |
| migrated-first | ✓ PLANNED | 5 months ago | run-A2sp | nfagerlund/terraform-minimum |
| migrated-second | ✓ PLANNED | 5 months ago | run-KqNV | nfagerlund/terraform-minimum |
| migrated-solo | ✓ APPLIED | 5 months ago | run-1RkX | NICKF/terraform-minimum |
| migrated-solo2 | ✓ PLANNED | 5 months ago | run-Rih7 | nfagerlund/terraform-minimum |
| migrate-first-2 | ! NEEDS CONFIRMATION | 3 months ago | run-hR57 | nfagerlund/terraform-minimum |

To get back to the workspace list at any time, click the "Workspaces" button in the top navigation bar.

- The individual workspace pages, which provide more detail about a workspace's status, and let you manage runs, variables, and settings.

To reach a workspace page, click that workspace's entry on the workspace list.

The screenshot shows the HashiCorp Terraform Cloud interface. At the top, there's a navigation bar with a logo, the workspace name "nicktech", and tabs for "Workspaces" (which is selected), "Modules", "Documentation", "Status", and a user icon. Below the navigation is a header for the "minimum-prod" workspace, with a "Queue Plan" button. The main content area is titled "Runs" and lists four recent runs:

- Revert "Merge pull request #2 from nfagerlund/pipeline"** This reverts commit `c9127b08145769d049b8a5ec79880f5400c8ad7c`, reversing changes made to `1e980e31dd19b59f3b8e5d10ac2689358a4cf594`.
Triggered by `#run-LuUPedxmoVZTUiC` | `nfagerlund` triggered from GitHub | Branch `master` | `e19eebb`
Status: `NO CHANGES✓` a month ago
- Queued manually in Terraform Enterprise**
Triggered by `#run-YnHVgMpyf87vC8m8` | `nfagerlund` triggered from Terraform Enterprise UI | Branch `master` | `c9127b0`
Status: `NO CHANGES✓` a month ago
- Queued manually in Terraform Enterprise**
Triggered by `#run-5qj9vzZJQ9ptvf4P` | `nfagerlund` triggered from Terraform Enterprise UI | Branch `master` | `c9127b0`
Status: `NO CHANGES✓` 2 months ago
- Queued manually in Terraform Enterprise**
Triggered by `#run-7KedSC4EjfGsWaNJ` | `nfagerlund` triggered from Terraform Enterprise UI | Branch `master` | `c9127b0`
Status: `APPLIED✓` 2 months ago

Next Steps

Now that you've created and configured at least one workspace, you're ready to start performing Terraform runs (</docs/enterprise/getting-started/runs.html>).

Terraform Recommended Practices

This guide is meant for enterprise users looking to advance their Terraform usage from a few individuals to a full organization.

Introduction

HashiCorp specializes in helping IT organizations adopt cloud technologies. Based on what we've seen work well, we believe the best approach to provisioning is **collaborative infrastructure as code**, using Terraform as the core workflow and Terraform Enterprise to manage the boundaries between your organization's different teams, roles, applications, and deployment tiers.

The collaborative infrastructure as code workflow is built on many other IT best practices (like using version control and preventing manual changes), and you must adopt these foundations before you can fully adopt our recommended workflow. Achieving state-of-the-art provisioning practices is a journey, with several distinct stops along the way.

This guide describes our recommended Terraform practices and how to adopt them. It covers the steps to start using our tools, with special attention to the foundational practices they rely on.

- Part 1: An Overview of Our Recommended Workflow (</docs/enterprise/guides/recommended-practices/part1.html>) is a holistic overview of Terraform Enterprise's collaborative infrastructure as code workflow. It describes how infrastructure is organized and governed, and how people interact with it.
- Part 2: Evaluating Your Current Provisioning Practices (</docs/enterprise/guides/recommended-practices/part2.html>) is a series of questions to help you evaluate the state of your own infrastructure provisioning practices. We define four stages of operational maturity around provisioning to help you orient yourself and understand which foundational practices you still need to adopt.
- Part 3: How to Evolve Your Provisioning Practices (</docs/enterprise/guides/recommended-practices/part3.html>) is a guide for how to advance your provisioning practices through the four stages of operational maturity. Many organizations are already partway through this process, so use what you learned in part 2 to determine where you are in this journey.

This part is split into four pages:

- Part 3.1: How to Move from Manual Changes to Semi-Automation (</docs/enterprise/guides/recommended-practices/part3.1.html>)
- Part 3.2: How to Move from Semi-Automation to Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.2.html>)
- Part 3.3: How to Move from Infrastructure as Code to Collaborative Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.3.html>)
- Part 3.4: Advanced Improvements to Collaborative Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.4.html>)

Next

Begin reading with Part 1: An Overview of Our Recommended Workflow (</docs/enterprise/guides/recommended-practices/part1.html>).

Part 1: An Overview of Our Recommended Workflow

Terraform's purpose is to provide one workflow to provision any infrastructure. In this section, we'll show you our recommended practices for organizing Terraform usage across a large organization. This is the set of practices that we call "collaborative infrastructure as code."

Fundamental Challenges in Provisioning

There are two major challenges everyone faces when trying to improve their provisioning practices: technical complexity and organizational complexity.

1. Technical complexity — Different infrastructure providers use different interfaces to provision new resources, and the inconsistency between these interfaces imposes extra costs on daily operations. These costs get worse as you add more infrastructure providers and more collaborators.

Terraform addresses this complexity by separating the provisioning workload. It uses a single core engine to read infrastructure as code configurations and determine the relationships between resources, then uses many provider plugins (<https://www.terraform.io/docs/providers/index.html>) to create, modify, and destroy resources on the infrastructure providers. These provider plugins can talk to IaaS (e.g. AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. GitHub, DNSimple, Cloudflare).

In other words, Terraform uses a model of workflow-level abstraction, rather than resource-level abstraction. It lets you use a single workflow for managing infrastructure, but acknowledges the uniqueness of each provider instead of imposing generic concepts on non-equivalent resources.

2. Organizational complexity — As infrastructure scales, it requires more teams to maintain it. For effective collaboration, it's important to delegate ownership of infrastructure across these teams and empower them to work in parallel without conflict. Terraform and Terraform Enterprise can help delegate infrastructure in the same way components of a large application are delegated.

To delegate a large application, companies often split it into small, focused microservice components that are owned by specific teams. Each microservice provides an API, and as long as those APIs don't change, microservice teams can make changes in parallel despite relying on each others' functionality.

Similarly, infrastructure code can be split into smaller Terraform configurations, which have limited scope and are owned by specific teams. These independent configurations use output variables (<https://www.terraform.io/docs/configuration/outputs.html>) to publish information and remote state resources (https://www.terraform.io/docs/providers/terraform/d/remote_state.html) to access output data from other workspaces. Just like microservices communicate and connect via APIs, Terraform workspaces connect via remote state.

Once you have loosely-coupled Terraform configurations, you can delegate their development and maintenance to different teams. To do this effectively, you need to control access to that code. Version control systems can regulate who can commit code, but since Terraform affects real infrastructure, you also need to regulate who can run the code.

This is how Terraform Enterprise (TFE) solves the organizational complexity of provisioning: by providing a centralized run environment for Terraform that supports and enforces your organization's access control decisions across all workspaces. This helps you delegate infrastructure ownership to enable parallel development.

Personas, Responsibilities, and Desired User Experiences

There are four main personas for managing infrastructure at scale. These roles have different responsibilities and needs, and Terraform Enterprise supports them with different tools and permissions.

Central IT

This team is responsible for defining common infrastructure practices, enforcing policy across teams, and maintaining shared services.

Central IT users want a single dashboard to view the status and compliance of all infrastructure, so they can quickly fix misconfigurations or malfunctions. Since Terraform Enterprise is tightly integrated with Terraform's run data and is designed around Terraform's concepts of workspaces and runs, it offers a more integrated workflow experience than a general-purpose CI system.

Organization Architect

This team defines how global infrastructure is divided and delegated to the teams within the business unit. This team also enables connectivity between workspaces by defining the APIs each workspace must expose, and sets organization-wide variables and policies.

Organization Architects want a single dashboard to view the status of all workspaces and the graph of connectivity between them.

Workspace Owner

This individual owns a specific set of workspaces, which build a given Terraform configuration across several environments. They are responsible for the health of those workspaces, managing the full change lifecycle through dev, UAT, staging, and production. They are the main approver of changes to production within their domain.

Workspace Owners want:

- A single dashboard to view the status of all workspaces that use their infrastructure code.
- A streamlined way to promote changes between environments.
- An interface to set variables used by a Terraform configuration across environments.

Workspace Contributor

Contributors submit changes to workspaces by making updates to the infrastructure as code configuration. They usually do not have approval to make changes to production, but can make changes in dev, UAT, and staging.

Workspace Contributors want a simple workflow to submit changes to a workspace and promote changes between workspaces. They can edit a subset of workspace variables and their own personal variables.

Workspace contributors are often already familiar with Terraform's operating model and command line interface, and can usually adapt quickly to TFE's web interface.

The Recommended Terraform Workspace Structure

About Workspaces

Terraform Enterprise's main unit of organization is a workspace. A workspace is a collection of everything Terraform needs to run: a Terraform configuration (usually from a VCS repo), values for that configuration's variables, and state data to keep track of operations between runs.

In Terraform open source, a workspace is just an independent state file on the local disk. In TFE, they're persistent shared resources; you can assign them their own access controls, monitor their run states, and more.

One Workspace Per Environment Per Terraform Configuration

Workspaces are TFE's primary tool for delegating control, which means their structure should match your organizational permissions structure. The best approach is to use one workspace for each environment of a given infrastructure component. Or in other words, Terraform configurations * environments = workspaces.

This is different from how some other tools view environments; notably, you shouldn't use a single Terraform workspace to manage everything that makes up your production or staging environment. Instead, make smaller workspaces that are easy to delegate. This also means not every configuration has to use the exact same environments; if a UAT environment doesn't make sense for your security infrastructure, you aren't forced to use one.

Name your workspaces with both their component and their environment. For example, if you have a Terraform configuration for managing an internal billing app and another for your networking infrastructure, you could name the workspaces as follows:

- billing-app-dev
- billing-app-stage
- billing-app-prod
- networking-dev
- networking-stage
- networking-prod

Delegating Workspaces

Since each workspace is one environment of one infrastructure component, you can use per-workspace access controls to delegate ownership of components and regulate code promotion across environments. For example:

- Teams that help manage a component can start Terraform runs and edit variables in dev or staging.
- The owners or senior contributors of a component can start Terraform runs in production, after reviewing other contributors' work.
- Central IT and organization architects can administer permissions on all workspaces, to ensure everyone has what they need to work.

- Teams that have no role managing a given component don't have access to its workspaces.

To use TFE effectively, you must make sure the division of workspaces and permissions matches your organization's division of responsibilities. If it's difficult to separate your workspaces effectively, it might reveal an area of your infrastructure where responsibility is muddled and unclear. If so, this is a chance to disentangle the code and enforce better boundaries of ownership.

Promoting Changes Between Related Workspaces (Coming Soon)

In a future version, TFE will let you create automatic promotion pipelines across workspaces, to help guarantee that high environments only run known good code.

Today, you have two options for manually promoting configurations from one workspace to another:

- Once a new configuration has passed a testing environment, update the code in the next environment's VCS repository to match that configuration. Usually this is done by with a branch merging workflow, but there are other ways to accomplish it.
- Use the runs API to handle promotion. Each run is associated with a static configuration version, and the API allows you to specify an existing version by its ID. You can create your high-environment workspaces without a backing VCS repository (to prevent automatic plans), look up the configuration version from the last successful run in a lower environment, then re-use that known-good configuration version when starting a run in the higher environment.

This method is advanced, but it takes advantage of the same APIs that the upcoming pipelines feature will use. For more information about using the runs API, see:

- The API-driven Run Workflow (</docs/enterprise/run/api.html>)
- The Runs API reference (</docs/enterprise/api/run.html>) (in particular, the "Create a Run" endpoint and the "List Runs in a Workspace" endpoint)

Next

Now that you're familiar with the outlines of the Terraform Enterprise workflow, it's time to assess your organization's provisioning practices. Continue on to Part 2: Evaluating Your Current Provisioning Practices (</docs/enterprise/guides/recommended-practices/part2.html>).

Part 2: Evaluating Your Current Provisioning Practices

Terraform Enterprise depends on several foundational IT practices. Before you can implement Terraform Enterprise's collaborative infrastructure as code workflows, you need to understand which of those practices you're already using, and which ones you still need to implement.

We've written the section below in the form of a quiz or interview, with multiple-choice answers that represent the range of operational maturity levels we've seen across many organizations. You should read it with a notepad handy, and take note of any questions where your organization can improve its use of automation and collaboration.

This quiz doesn't have a passing or failing score, but it's important to know your organization's answers. Once you know which of your IT practices need the most attention, Section 3 will guide you from your current state to our recommended practices in the most direct way.

Four Levels of Operational Maturity

Each question has several answers, each of which aligns with a different level of operational maturity. Those levels are as follows:

1. Manual

- Infrastructure is provisioned through a UI or CLI.
- Configuration changes do not leave a traceable history, and aren't always visible.
- Limited or no naming standards in place.

2. Semi-automated

- Infrastructure is provisioned through a combination of UI/CLI, infrastructure as code, and scripts or configuration management.
- Traceability is limited, since different record-keeping methods are used across the organization.
- Rollbacks are hard to achieve due to differing record-keeping methods.

3. Infrastructure as code

- Infrastructure is provisioned using Terraform OSS.
- Provisioning and deployment processes are automated.
- Infrastructure configuration is consistent, with all necessary details fully documented (nothing siloed in a sysadmin's head).
- Source files are stored in version control to record editing history, and, if necessary, roll back to older versions.
- Some Terraform code is split out into modules, to promote consistent reuse of your organization's more common architectural patterns.

4. Collaborative infrastructure as code

- Users across the organization can safely provision infrastructure with Terraform, without conflicts and with clear understanding of their access permissions.

- Expert users within an organization can produce standardized infrastructure templates, and beginner users can consume those to follow infrastructure best practices for the organization.
- Per-workspace access control helps committers and approvers on workspaces protect production environments.
- Functional groups that don't directly write Terraform code have visibility into infrastructure status and changes through Terraform Enterprise's UI.

By the end of this section, you should have a clear understanding of which operational maturity stage you are in. Section 3 will explain the recommended steps to move from your current stage to the next one.

Answering these questions will help you understand your organization's method for provisioning infrastructure, its change workflow, its operation model, and its security model.

Once you understand your current practices, you can identify the remaining steps for implementing Terraform Enterprise.

Your Current Configuration and Provisioning Practices

How does your organization configure and provision infrastructure today? Automated and consistent practices help make your infrastructure more knowable and reliable, and reduce the amount of time spent on troubleshooting.

The following questions will help you evaluate your current level of automation for configuration and provisioning.

Q1. How do you currently manage your infrastructure?

1. Through a UI or CLI. This might seem like the easiest option for one-off tasks, but for recurring operations it is a big consumer of valuable engineering time. It's also difficult to track and manage changes.
2. Through reusable command line scripts, or a combination of UI and infrastructure as code. This is faster and more reliable than pure ad-hoc management and makes recurring operations repeatable, but the lack of consistency and versioning makes it difficult to manage over time.
3. Through an infrastructure as code tool (Terraform, CloudFormation). Infrastructure as code enables scalable, repeatable, and versioned infrastructure. It dramatically increases the productivity of each operator and can enforce consistency across environments when used appropriately.
4. Through a general-purpose automation framework (i.e. Jenkins + scripts / Jenkins + Terraform). This centralizes the management workflow, albeit with a tool that isn't built specifically for provisioning tasks.

Q2. What topology is in place for your service provider accounts?

1. Flat structure, single account. All infrastructure is provisioned within the same account.
2. Flat structure, multiple accounts. Infrastructure is provisioned using different infrastructure providers, with an account per environment.
3. Tree hierarchy. This features a master billing account, an audit/security/logging account, and project/environment-specific infrastructure accounts.

Q3. How do you manage the infrastructure for different environments?

1. Manual. Everything is manual, with no configuration management in place.
2. Siloed. Each application team has its own way of managing infrastructure — some manually, some using infrastructure as code or custom scripts.
3. Infrastructure as code with different code bases per environment. Having different code bases for infrastructure as code configurations can lead to untracked changes from one environment to the other if there is no promotion within environments.
4. Infrastructure as code with a single code base and differing environment variables. All resources, regardless of environment, are provisioned with the same code, ensuring that changes promote through your deployment tiers in a predictable way.

Q4. How do teams collaborate and share infrastructure configuration and code?

1. N/A. Infrastructure as code is not used.
2. Locally. Infrastructure configuration is hosted locally and shared via email, documents or spreadsheets.
3. Ticketing system. Code is shared through journal entries in change requests or problem/incident tickets.
4. Centralized without version control. Code is stored on a shared filesystem and secured through security groups. Changes are not versioned. Rollbacks are only possible through restores from backups or snapshots.
5. Configuration stored and collaborated in a version control system (VCS) (Git repositories, etc.). Teams collaborate on infrastructure configurations within a VCS workflow, and can review infrastructure changes before they enter production. This is the most mature approach, as it offers the best record-keeping and cross-department/cross-team visibility.

Q5. Do you use reusable modules for writing infrastructure as code?

1. Everything is manual. No infrastructure as code currently used.
2. No modularity. Infrastructure as code is used, but primarily as one-off configurations. Users usually don't share or reuse code.
3. Teams use modules internally but do not share them across teams.
4. Modules are shared organization-wide. Similar to shared software libraries, a module for a common infrastructure pattern can be updated once and the entire organization benefits.

Your Current Change Control Workflow

Change control is a formal process to coordinate and approve changes to a product or system. The goals of a change control process include:

- Minimizing disruption to services.
- Reducing rollbacks.

- Reducing the overall cost of changes.
- Preventing unnecessary changes.
- Allowing users to make changes without impacting changes made by other users.

The following questions will help you assess the maturity of your change control workflow.

Q6. How do you govern the access to control changes to infrastructure?

1. Access is not restricted or audited. Everyone in the platform team has the flexibility to create, change, and destroy all infrastructure. This leads to a complex system that is unstable and hard to manage.
2. Access is not restricted, only audited. This makes it easier to track changes after the fact, but doesn't proactively protect your infrastructure's stability.
3. Access is restricted based on service provider account level. Members of the team have admin access to different accounts based on the environment they are responsible for.
4. Access is restricted based on user roles. All access is restricted based on user roles at infrastructure provider level.

Q7. What is the process for changing existing infrastructure?

1. Manual changes by remotely logging into machines. Repetitive manual tasks are inefficient and prone to human errors.
2. Runtime configuration management (Puppet, Chef, etc.). Configuration management tools let you make fast, automated changes based on readable and auditible code. However, since they don't produce static artifacts, the outcome of a given configuration version isn't always 100% repeatable, making rollbacks only partially reliable.
3. Immutable infrastructure (images, containers). Immutable components can be replaced for every deployment (rather than being updated in-place), using static deployment artifacts. If you maintain sharp boundaries between ephemeral layers and state-storing layers, immutable infrastructure can be much easier to test, validate, and roll back.

Q8. How do you deploy applications?

1. Manually (SSH, WinRM, rsync, robocopy, etc.). Repetitive manual tasks are inefficient and prone to human errors.
2. With scripts (Fabric, Capistrano, custom, etc.).
3. With a configuration management tool (Chef, Puppet, Ansible, Salt, etc.), or by passing userdata scripts to CloudFormation Templates or Terraform configuration files.
4. With a scheduler (Kubernetes, Nomad, Mesos, Swarm, ECS, etc.).

Your Current Security Model

Q9. How are infrastructure service provider credentials managed?

1. By hardcoding them in the source code. This is highly insecure.
2. By using infrastructure provider roles (like EC2 instance roles for AWS). Since the service provider knows the identity of the machines it's providing, you can grant some machines permission to make API requests without giving them a copy of your actual credentials.
3. By using a secrets management solution (like Vault, Keywhis, or PAR). We recommend this.
4. By using short-lived tokens. This is one of the most secure methods, since the temporary credentials you distribute expire quickly and are very difficult to exploit. However, this can be more complex to use than a secrets management solution.

Q10. How do you control users and objects hosted by your infrastructure provider (like logins, access and role control, etc.)?

1. A common 'admin' or 'superuser' account shared by engineers. This increases the possibility of a breach into your infrastructure provider account.
2. Individual named user accounts. This makes a loss of credentials less likely and easier to recover from, but it doesn't scale very well as the team grows.
3. LDAP and/or Active Directory integration. This is much more secure than shared accounts, but requires additional architectural considerations to ensure that the provider's access into your corporate network is configured correctly.
4. Single sign-on through OAuth or SAML. This provides token-based access into your infrastructure provider while not requiring your provider to have access to your corporate network.

Q11. How do you track the changes made by different users in your infrastructure provider's environments?

1. No logging in place. Auditing and troubleshooting can be very difficult without a record of who made which changes when.
2. Manual changelog. Users manually write down their changes to infrastructure in a shared document. This method is prone to human error.
3. By logging all API calls to an audit trail service or log management service (like CloudTrail, Loggly, or Splunk). We recommend this. This ensures that an audit trail is available during troubleshooting and/or security reviews.

Q12. How is the access of former employees revoked?

1. Immediately, manually. If you don't use infrastructure as code, the easiest and quickest way is by removing access for that employee manually using the infrastructure provider's console.
2. Delayed, as part of the next release. If your release process is extremely coupled and most of your security changes have to pass through a CAB (Change Advisory Board) meeting in order to be executed in production, this could be delayed.
3. Immediately, writing a hot-fix in the infrastructure as code. This is the most secure and recommended option. Before the employee leaves the building, access must be removed.

Assessing the Overall Maturity of Your Provisioning Practices

After reviewing all of these questions, look back at your notes and assess your organization's overall stage of maturity: are your practices mostly manual, semi-automated, infrastructure as code, or collaborative infrastructure as code?

Keep your current state in mind as you read the next section.

Next

Now that you've taken a hard look at your current practices, it's time to begin improving them. Continue on to [Part 3: How to Evolve Your Provisioning Practices \(/docs/enterprise/guides/recommended-practices/part3.html\)](#).

Part 3.1: How to Move from Manual Changes to Semi-Automation

Building infrastructure manually (with CLI or GUI tools) results in infrastructure that is hard to audit, hard to reproduce, hard to scale, and hard to share knowledge about.

If your current provisioning practices are largely manual, your first goal is to begin using open source Terraform in a small, manageable subset of your infrastructure. Once you've gotten some small success using Terraform, you'll have reached the semi-automated stage of provisioning maturity, and can begin to scale up and expand your Terraform usage.

Allow one individual (or a small group) in your engineering team to get familiar with Terraform by following these steps:

1. Install Terraform

Follow the instructions here to install Terraform OSS (<https://learn.hashicorp.com/terraform/getting-started/install>).

2. Write Some Code

Write your first Terraform Configuration file (<https://learn.hashicorp.com/terraform/getting-started/build>).

3. Read the Getting Started Guide

Follow the rest of the Terraform getting started guide (<https://learn.hashicorp.com/terraform/getting-started/change>). These pages will walk you through changing (<https://learn.hashicorp.com/terraform/getting-started/change>) and destroying (<https://learn.hashicorp.com/terraform/getting-started/destroy>) resources, working with resource dependencies (<https://learn.hashicorp.com/terraform/getting-started/dependencies>), and more.

4. Implement a Real Infrastructure Project

Choose a small real-life project and implement it with Terraform. Look at your organization's list of upcoming projects, and designate one to be a Terraform proof-of-concept. Alternately, you can choose some existing infrastructure to re-implement with Terraform.

The key is to choose a project with limited scope and clear boundaries, such as provisioning infrastructure for a new application on AWS. This helps keep your team from getting overwhelmed with features and possibilities. You can also look at some example projects (<https://github.com/hashicorp/terraform/tree/master/examples/>) to get a feel for your options. (The AWS two-tier example (<https://github.com/terraform-providers/terraform-provider-aws/tree/master/examples/two-tier>) is often a good start.)

Your goal here is to build a small but reliable core of expertise with Terraform, and demonstrate its benefits to others in the organization.

Next

At this point, you've reached a semi-automated stage of provisioning practices — one or more people in the organization can write Terraform code to provision and modify resources, and a small but meaningful subset of your infrastructure is being managed as code. This is a good time to provide a small demo to the rest of team to show how easy it is to write and provision infrastructure with Terraform.

Next, it's time to transition to a more complete infrastructure as code workflow. Continue on to Part 3.2: How to Move from Semi-Automation to Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.2.html>).

Part 3.2: How to Move from Semi-Automation to Infrastructure as Code

We define semi-automated provisioning as a mix of at least two of the following practices:

- Infrastructure as code with Terraform.
- Manual CLI or GUI processes.
- Scripts.

If that describes your current provisioning practices, your next goal is to expand your use of Terraform, reduce your use of manual processes and imperative scripts, and make sure you've adopted the foundational practices that make infrastructure as code more consistent and useful.

Note: If you aren't already using infrastructure as code for some portion of your infrastructure, make sure you follow the steps in the previous section first.

1. Use Version Control

Choose and implement a version control system (VCS) if your organization doesn't already use a VCS.

You might be able to get by with a minimalist Git/Mercurial/SVN server, but we recommend adopting a more robust collaborative VCS application that supports code reviews/approvals and has APIs for accessing data and administering repositories and accounts. Bitbucket, GitLab, and GitHub are popular tools in this space.

If you already have established VCS workflows, layouts, and access control practices, great! If not, this is a good time to make these decisions. (We consider this advice (<https://www.drupalwatchdog.com/volume-4/issue-2/version-control-workflow-strategies>) to be a good starting point.) Make sure you have a plan for who is allowed to merge changes and under what circumstances — since this code will be managing your whole infrastructure, it's important to maintain its integrity and quality.

Also, make sure to write down your organization's expectations and socialize them widely among your teams.

Make sure you've picked a VCS system that Terraform Enterprise will be able to access. Currently, Terraform Enterprise supports integrations with GitHub, GitLab and Atlassian Bitbucket (both Server and Cloud).

2. Put Terraform Code in VCS Repos

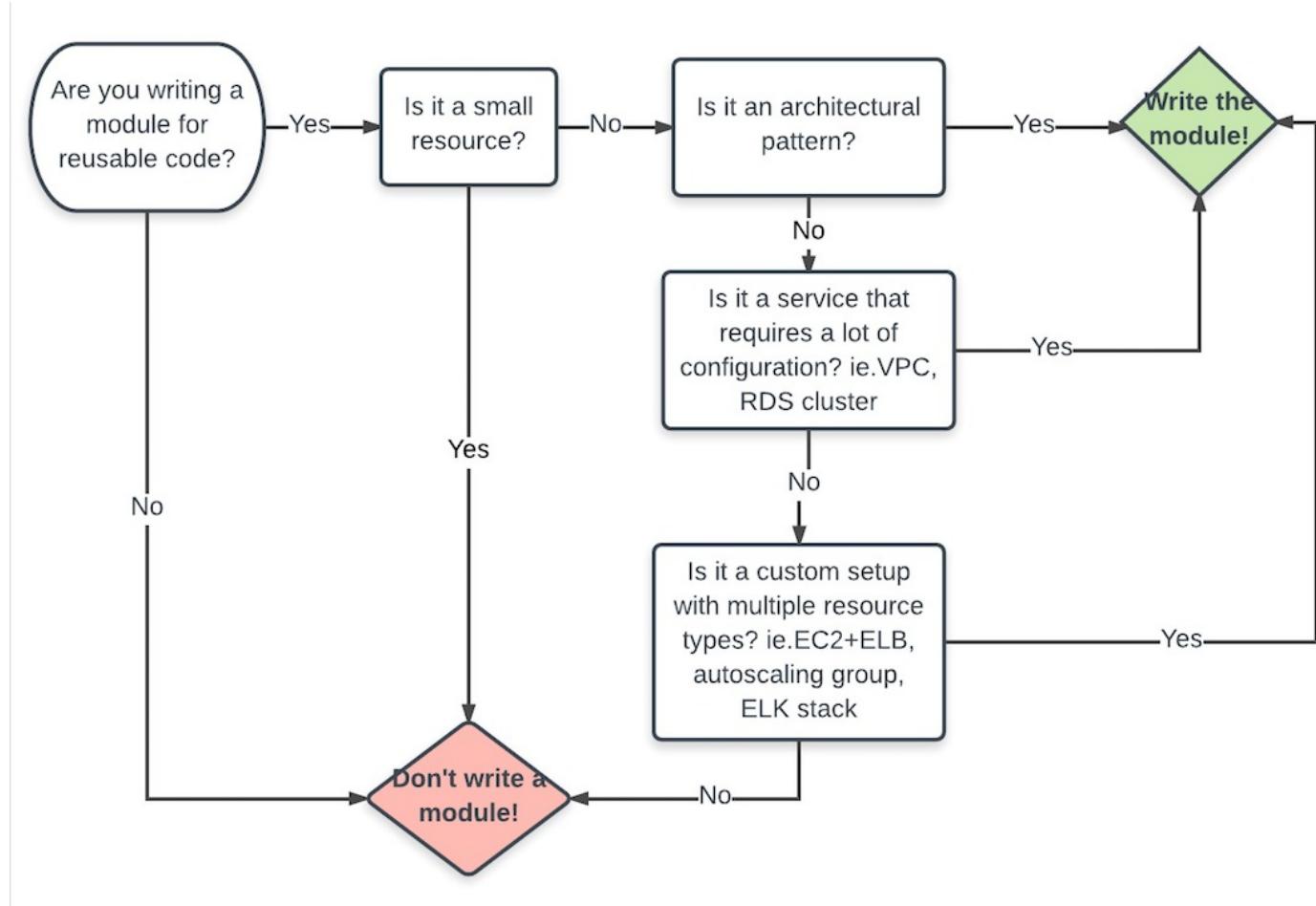
Start moving infrastructure code into version control. New Terraform code should all be going into version control; if you have existing Terraform code that's outside version control, start moving it in so that everyone in your organization knows where to look for things and can track the history and purpose of changes.

Note: There are several ways to structure Terraform repositories. If you want to learn more about how your repo structure can affect your Terraform Enterprise workflows, see VCS Repository Structure (<https://www.terraform.io/docs/enterprise/workspaces/repo-structure.html>) in the Terraform Enterprise documentation.

3. Create Your First Module

Terraform modules (<https://www.terraform.io/docs/modules/usage.html>) are reusable configuration units. They let you manage pieces of infrastructure as a single package you can call and define multiple times in the main configuration for a workspace. Examples of a good Terraform module candidate would be an auto-scaling group on AWS that wraps a launch configuration, auto-scaling group, and EC2 Elastic Load Balancer (ELB). If you are already using Terraform modules, make sure you're following the best practices and keep an eye on places where your modules could improve.

The diagram below can help you decide when to write a module:



4. Share Knowledge

Spread Terraform skills to additional teams, and improve the skills of existing infrastructure teams. In addition to internal training and self-directed learning, you might want to consider:

- Sign your teams up for official HashiCorp Training (<https://www.hashicorp.com/training/>).
- Make available resources such as Terraform Up and Running: Writing Infrastructure as Code (https://www.amazon.com/Terraform-Running-Writing-Infrastructure-Code-ebook/dp/B06XKHGJHP/ref=sr_1_1?ie=UTF8&qid=1496138592&sr=8-1&keywords=terraform+up+and+running) or Getting Started with Terraform (https://www.amazon.com/Getting-Started-Terraform-Kirill-Shirinkin/dp/1786465108/ref=sr_1_1?ie=UTF8&qid=1496138892&sr=8-1&keywords=Getting+Started+with+Terraform). These are especially valuable when nobody in your organization has used Terraform before.

5. Set Guidelines

Create standard build architectures to use as guidelines for writing Terraform code. Modules work best when they're shared across an organization, and sharing is more effective if everyone has similar expectations around how to design infrastructure. Your IT architects should design some standardized build architectures specific to your organizational needs, to encourage building with high availability, elasticity and disaster recovery in mind, and to support consistency across teams.

Here are a few examples of good build patterns from several cloud providers:

- AWS: Well Architected Frameworks (https://d0.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf) and the Architecture Center (<https://aws.amazon.com/architecture/>).
- Azure: deploying Azure Reference Architectures (<https://github.com/mspnp/reference-architectures>) and Azure Architecture Center (<https://docs.microsoft.com/en-us/azure/architecture/>).
- GCP: Building scalable and resilient web applications. (<https://cloud.google.com/solutions/scalable-and-resilient-apps>)
- Oracle Public Cloud: Best Practices for Using Oracle Cloud. (<https://docs.oracle.com/cloud/latest/stcompute/STCSG/GUID-C37FDFF1-7C48-4DA8-B31F-D7D7B35674A8.html#STCSG-GUID-C37FDFF1-7C48-4DA8-B31F-D7D7B35674A8>)

6. Integrate Terraform With Configuration Management

If your organization already has a configuration management tool, then it's time to integrate it with Terraform — you can use Terraform's provisioners (<https://www.terraform.io/docs/provisioners/index.html>) to pass control to configuration management after a resource is created. Terraform should handle the infrastructure, and other tools should handle user data and applications.

If your organization doesn't use a configuration management tool yet, and the configuration of the infrastructure being managed is mutable, you should consider adopting a configuration management tool. This might be a large task, but it supports the same goals that drove you to infrastructure as code, by making application configuration more controllable, understandable, and repeatable across teams.

If you're just getting started, try this tutorial on how to create a Chef cookbook (https://www.vagrantup.com/docs/provisioning/chef_solo.html) and test it locally with Vagrant. We also recommend this article about how to decide what configuration management tool (<http://www.intigua.com/blog/puppet-vs.-chef-vs.-ansible-vs.-saltstack>) is best suited for your organization.

7. Manage Secrets

Integrate Terraform with Vault (<https://www.terraform.io/docs/providers/vault/index.html>) or another secret management tool. Secrets like service provider credentials must stay secret, but they also must be easy to use when needed. The best way to address those needs is to use a dedicated secret management tool. We believe HashiCorp's Vault is the best choice for most people, but Terraform can integrate with other secret management tools as well.

Next

At this point, your organization has a VCS configured, is managing key infrastructure with Terraform, and has at least one reusable Terraform module. Compared to a semi-automated practice, your organization has much better visibility into infrastructure configuration, using a consistent language and workflow.

Next, you need an advanced workflow that can scale and delegate responsibilities to many contributors. Continue on to Part 3.3: How to Move from Infrastructure as Code to Collaborative Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.3.html>).

Part 3.3: How to Move from Infrastructure as Code to Collaborative Infrastructure as Code

Using version-controlled Terraform configurations to manage key infrastructure eliminates a great deal of technical complexity and inconsistency. Now that you have the basics under control, you're ready to focus on other problems.

Your next goals are to:

- Adopt consistent workflows for Terraform usage across teams
- Expand the benefits of Terraform beyond the core of engineers who directly edit Terraform code.
- Manage infrastructure provisioning permissions for users and teams.

Terraform Enterprise (TFE) (<https://www.hashicorp.com/products/terraform/>) is the product we've built to help you address these next-level problems. The following section describes how to start using it most effectively.

Note: If you aren't already using mature Terraform code to manage a significant portion of your infrastructure, make sure you follow the steps in the previous section first.

1. Install or Sign Up for TFE

You have two options for installing Terraform Enterprise: SaaS or private install. If you have chosen the SaaS version then you can skip this step; otherwise visit the installation guide (<https://github.com/hashicorp/terraform-enterprise-modules/blob/master/INSTALLING.md>) to get started. Terraform will display the URL to your TFE servers as an output.

2. Learn TFE's Run Environment

Get familiar with how Terraform runs work in TFE. With Terraform OSS, you generally use external VCS tools to get code onto the filesystem, then execute runs from the command line or from a general purpose CI system.

TFE does things differently: a workspace is associated directly with a VCS repo, and you use TFE's UI or API to start and monitor runs. To get familiar with this operating model:

- Read the documentation on how to perform and configure Terraform runs (</docs/enterprise/getting-started/runs.html>) in Terraform Enterprise.
- Create a proof-of-concept workspace, associate it with Terraform code in a VCS repo, set variables as needed, and use Terraform Enterprise to perform some Terraform runs with that code.

3. Design Your Organization's Workspace Structure

In TFE, each Terraform configuration should manage a specific infrastructure component, and each environment of a given configuration should be a separate workspace — in other words, Terraform configurations * environments = workspaces. A workspace name should be something like "networking-dev," so you can tell at a glance which infrastructure and environment it manages.

The definition of an “infrastructure component” depends on your organization’s structure. A given workspace might manage an application, a service, or a group of related services; it might provision infrastructure used by a single engineering team, or it might provision shared, foundational infrastructure used by the entire business.

You should structure your workspaces to match the divisions of responsibility in your infrastructure. You will probably end up with a mixture: some components, like networking, are foundational infrastructure controlled by central IT staff; others are application-specific and should be controlled by the engineering teams that rely on them.

Also, keep in mind:

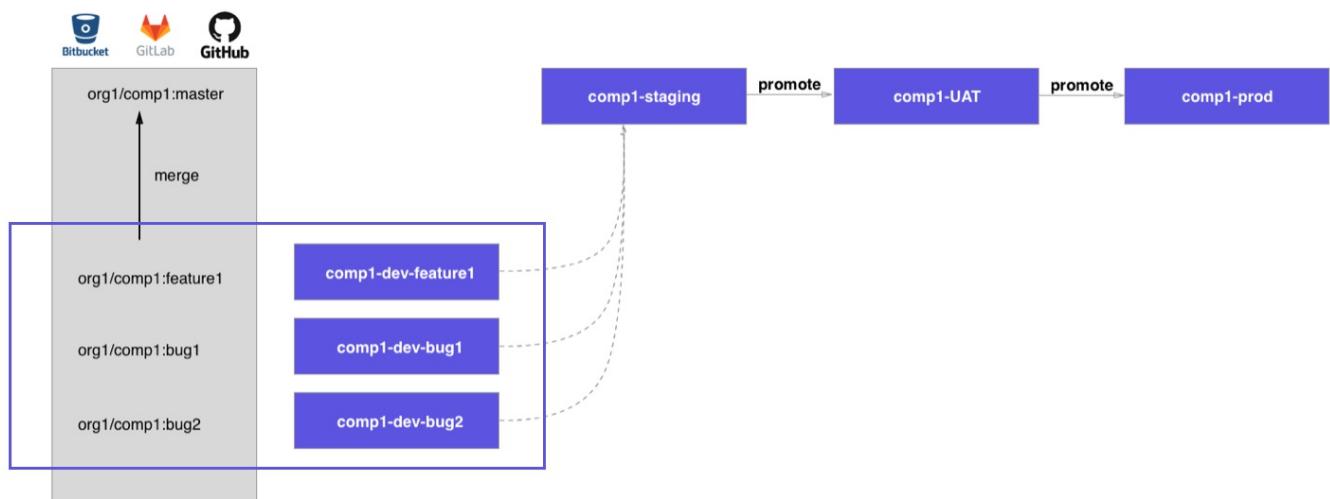
- Some workspaces publish output data to be used by other workspaces.
- The workspaces that make up a configuration’s environments (app1-dev, app1-stage, app1-prod) should be run in order, to ensure code is properly verified.

The first relationship, a relationship between workspaces for different components but the same environment, creates a graph of dependencies between workspaces, and you should stay aware of it. The second relationship, a relationship between workspaces for the same component but different environments, creates a pipeline between workspaces. TFE doesn’t currently have the ability to act on these dependencies, but features like cascading updates and promotion are coming soon, and you’ll be able to use them more easily if you already understand how your workspaces relate.

4. Create Workspaces

Create workspaces in TFE, and map VCS repositories to them. Each workspace reads its Terraform code from your version control system. You’ll need to assign a repository and branch to each workspace.

We recommend using the same repository and branch for every environment of a given app or service — write your Terraform code such that you can differentiate the environments via variables, and set those variables appropriately per workspace. This might not be practical for your existing code yet, in which case you can use different branches per workspace and handle promotion through your merge strategy, but we believe a model of one canonical branch works best.



5. Plan and Create Teams

TFE manages workspace access with teams, which are groups of user accounts.

Your TFE teams should match your understanding of who's responsible for which infrastructure. That isn't always an exact

match for your org chart, so make sure you spend some time thinking about this and talking to people across the organization. Keep in mind:

- Some teams need to administer many workspaces, and others only need permissions on one or two.
- A team might not have the same permissions on every workspace they use; for example, application developers might have read/write access to their app's dev and stage environments, but read-only access to prod.

Managing an accurate and complete map of how responsibilities are delegated is one of the most difficult parts of practicing collaborative infrastructure as code.

When managing team membership, you have two options:

- Manage user accounts with SAML single sign-on (</docs/enterprise/saml/index.html>). SAML support is available with the Premium tier on private installs, and lets users log into TFE via your organization's existing identity provider. If your organization is at a scale where you use a SAML-compatible identity provider, we recommend this option.

If your identity provider already has information about your colleagues' teams or groups, you can manage team membership via your identity provider (</docs/enterprise/saml/team-membership.html>). Otherwise, you can add users to teams with the UI or with the team membership API (</docs/enterprise/api/team-members.html>).

- Manage user accounts in TFE. Your colleagues must create their own TFE user accounts, and you can add them to your organization by adding their username to at least one team. You can manage team membership with the UI or with the team membership API (</docs/enterprise/api/team-members.html>).

6. Assign Permissions

Assign workspace ownership and permissions to teams. Each workspace has three levels of permissions you can grant to any user or team: admin, read/write, and read-only. Admins effectively own the workspace, and can change the permissions of other users on it.

Most workspaces will give access to multiple teams with different permissions.

| Workspace | Team Permissions |
|-----------------|--|
| app1-dev | Team-eng-app1: Read/write
Team-owners-app1: Admin
Team-central-IT: Admin |
| app1-prod | Team-eng-app1: Read-only
Team-owners-app1: Read/write
Team-central-IT: Admin |
| networking-dev | Team-eng-networking: Read/write
Team-owners-networking: Admin
Team-central-IT: Admin |
| networking-prod | Team-eng-networking: Read-only
Team-owners-networking: Read/write
Team-central-IT: Admin |

7. Restrict Non-Terraform Access

Restrict access to cloud provider UIs and APIs. Since Terraform Enterprise is now your organization's primary interface for

infrastructure provisioning, you should restrict access to any alternate interface that bypasses TFE. For almost all users, it should be impossible to manually modify infrastructure without using the organization's agreed-upon Terraform workflow.

As long as no one can bypass Terraform, your code review processes and your TFE workspace permissions are the definitive record of who can modify which infrastructure. This makes everything about your infrastructure more knowable and controllable. Terraform Enterprise is one workflow to learn, one workflow to secure, and one workflow to audit for provisioning any infrastructure in your organization.

Next

At this point, you have successfully adopted a collaborative infrastructure as code workflow with Terraform Enterprise. You can provision infrastructure across multiple providers using a single workflow, and you have a shared interface that helps manage your organization's standards around access control and code promotion.

Next, you can make additional improvements to your workflows and practices. Continue on to Part 3.4: Advanced Improvements to Collaborative Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.4.html>).

Part 3.4: Advanced Improvements to Collaborative Infrastructure as Code

Now that you have a collaborative interface and workflow for provisioning, you have a solid framework for improving your practices even further.

The following suggestions don't have to be done in order, and some of them might not make sense for every business. We present them as possibilities for when you find yourself asking what's next.

- Move more processes and resources into TFE. Even after successfully implementing TFE, there's a good chance you still have manual or semi-automated workflows and processes. We suggest holding a discovery meeting with all of the teams responsible for keeping infrastructure running, to identify future targets for automation. You can also use your notes from the questions in section 2 as a guide, or go through old change requests or incident tickets.
- Adopt HashiCorp Packer (<https://www.packer.io/>) for image creation. Packer helps you build machine images in a maintainable and repeatable way, and can amplify Terraform's usefulness.
- Apply policy to your Terraform configurations with Sentinel (</docs/enterprise/sentinel/index.html>) to enforce compliance with business and regulatory rules.
- Monitor and retain TFE's audit logs. Learn more about logging in Terraform Enterprise private installs here. (</docs/enterprise/private/logging.html>)
- Add infrastructure monitoring and performance metrics. This can help make environment promotion safer, and safeguard the performance of your applications. There are many tools available in this space, and we recommend monitoring both the infrastructure itself, and the user's-eye-view performance of your applications.
- Use the TFE API. The TFE API (</docs/enterprise/api/index.html>) can be used to integrate with general-purpose CI/CD tools to trigger Terraform runs in response to a variety of events.

Part 3: How to Evolve Your Provisioning Practices

This section describes the steps necessary to move an organization from manual provisioning processes to a collaborative infrastructure as code workflow. For each stage of operational maturity, we give instructions for moving your organization to the next stage, eventually arriving at our recommended workflow.

We've split this section into multiple pages, so you can skip instructions that you've already implemented. Look back at your notes from Part 2, and start with the page about your current level of operational maturity.

- Part 3.1: How to Move from Manual Changes to Semi-Automation (</docs/enterprise/guides/recommended-practices/part3.1.html>)
- Part 3.2: How to Move from Semi-Automation to Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.2.html>)
- Part 3.3: How to Move from Infrastructure as Code to Collaborative Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.3.html>)
- Part 3.4: Advanced Improvements to Collaborative Infrastructure as Code (</docs/enterprise/guides/recommended-practices/part3.4.html>)

Migrating State from Terraform Open Source

If you already use Terraform to manage infrastructure, you're probably managing some infrastructure that you want to transfer to Terraform Enterprise (TFE). By migrating your Terraform state (</docs/state/index.html>) to TFE, you can hand off infrastructure without de-provisioning anything.

Important: These instructions are for migrating state in a basic working directory that only uses the default workspace.

If you use multiple workspaces (</docs/state/workspaces.html>) in one working directory, the instructions are different; see [Migrating State from Multiple Terraform Workspaces](/docs/enterprise/migrate/workspaces.html) (</docs/enterprise/migrate/workspaces.html>) instead.

API: See the State Versions API (</docs/enterprise/api/state-versions.html>). Be sure to stop Terraform runs before migrating state to TFE, and only import state into TFE workspaces that have never performed a run.

Step 1: Gather Credentials, Data, and Code

Make sure you have all of the following:

- The location of the VCS repository for the Terraform configuration in question.
- A local working copy of the Terraform configuration in question.
- The existing state data. The location of the state depends on which backend (</docs/backends/index.html>) you've been using:
 - If you were using the default `local` backend, your state is a file on disk. You need the original working directory where you've been running Terraform, or a copy of the `terraform.tfstate` file to copy into a fresh working directory.
 - For remote backends, you need the path to the particular storage being used (usually already included in the configuration) and access credentials (which you usually must set as an environment variable).
- A TFE user account which is a member of your organization's owners team, so you can create workspaces.
- A user API token (</docs/enterprise/users-teams-organizations/users.html#api-tokens>) for your TFE user account. (Organization and team tokens will not work; the token must be associated with an individual user.)

In your shell, set an `ATLAS_TOKEN` environment variable with your API token as the value.

```
export ATLAS_TOKEN=<USER TOKEN>
```

Step 2: Create a New TFE Workspace

Create a new workspace in your TFE organization to take over management of this infrastructure. Set the VCS repository and any necessary variable values appropriately. You can set team access permissions now or later, whichever is more convenient.

Do not perform any runs in this workspace yet, and consider locking the workspace to be sure. If an `apply` happens prematurely, you'll need to destroy the workspace and start the process over.

Step 3: Stop Terraform Runs

However you currently run Terraform to manage this infrastructure, stop. Let any current runs finish, then make sure there won't be more.

This might involve locking or deleting CI jobs, restricting access to the state backend, or just communicating very clearly with your colleagues.

Step 4: Prepare Your Terraform Working Directory

In your shell, go to the directory with the Terraform configuration you're migrating.

- If you've already been doing Terraform runs in this directory, you should be ready to go.
- If you had to retrieve a `terraform.tfstate` file from elsewhere, copy it into the root of the working directory and run `terraform init`.
- If you use a remote backend and had to check out a fresh working directory from version control, run `terraform init`.

Step 5: Edit the Backend Configuration

If the Terraform configuration has an existing backend block, delete it now.

Add a `terraform { backend ... }` block to the configuration.

```
terraform {
  backend "atlas" {
    name = "<TFE ORG>/<WORKSPACE NAME>"
    address = "https://<TFE HOSTNAME>"
  }
}
```

- Use the `atlas` backend. (This is TFE's backend; the `Atlas` name is used here for historical reasons.)
- In the `name` attribute, specify the name of your TFE organization and the target workspace, separated by a slash. (For example, `example_corp/database-prod`.)
- The `address` attribute is only necessary with private TFE instances. You can omit it if you're using the SaaS version of TFE.

Note: The `atlas` backend block is temporary; it isn't needed once TFE takes over Terraform runs. It's easiest to put the backend block in a separate `backend.tf` file which isn't checked into version control.

Step 6: Run `terraform init` and Answer "Yes"

Run `terraform init`.

The `init` command will notice that your new TFE workspace doesn't have any state, and will offer to migrate the previous state to it. The prompt usually looks like this:

```
Do you want to copy existing state to the new backend?  
Pre-existing state was found while migrating the previous "local" backend to the  
newly configured "atlas" backend. No existing state was found in the newly  
configured "atlas" backend. Do you want to copy this state to the new "atlas"  
backend? Enter "yes" to copy and "no" to start with an empty state.
```

Answer "yes," and Terraform will migrate your state.

After the `init` has finished, you can delete the temporary `atlas` backend block.

Step 7: Enable Runs in the New Workspace

In TFE, unlock the new workspace and queue a plan. Examine the results.

If all went well, the plan should result in no changes or very small changes. TFE can now take over all Terraform runs for this infrastructure.

Troubleshooting

- If the plan would create an entirely new set of infrastructure resources, you probably have the wrong state file.

In the case of a wrong state file, you can recover by fixing your local working directory and trying again. You'll need to re-set to the local backend, run `terraform init`, replace the state file with the correct one, change back to the `atlas` backend, run `terraform init` again, and confirm that you want to replace the remote state with the current local state.

- If the plan recognizes the existing resources but would make unexpected changes, check whether the designated VCS branch for the workspace is the same branch you've been running Terraform on, and update it, if it is not. You can also check whether variables in the TFE workspace have the correct values.

Migrating State from Multiple Terraform Workspaces

Terraform can manage multiple groups of infrastructure in one working directory by switching workspaces ([/docs/state/workspaces.html](#)).

These workspaces, managed with the `terraform workspace` command, aren't the same thing as Terraform Enterprise (TFE)'s workspaces. TFE workspaces act more like completely separate working directories; CLI workspaces are just alternate state files.

If you use multiple workspaces, you'll need to migrate each one to a separate TFE workspace.

Migrating multiple workspaces is similar to migrating a single workspace ([/docs/enterprise/migrate/index.html](#)), but it requires some extra steps.

API: See the State Versions API ([/docs/enterprise/api/state-versions.html](#)). Be sure to stop Terraform runs before migrating state to TFE, and only import state into TFE workspaces that have never performed a run.

Step 1: Gather Credentials, Data, and Code

Make sure you have all of the following:

- The location of the VCS repository for the Terraform configuration in question.
- A local working copy of the Terraform configuration in question.
- The existing state data. The location of the state depends on which backend ([/docs/backends/index.html](#)) you've been using:
 - If you were using the `local` backend, your state is files on disk. You need the original working directory where you've been running Terraform, or a copy of the `terraform.tfstate` and `terraform.tfstate.d` files to copy into a fresh working directory.
 - For remote backends, you need the path to the particular storage being used (usually already included in the configuration) and access credentials (which you usually must set as an environment variable).
- A TFE user account which is a member of your organization's owners team, so you can create workspaces.
- A user API token ([/docs/enterprise/users-teams-organizations/users.html#api-tokens](#)) for your TFE user account. (Organization and team tokens will not work; the token must be associated with an individual user.)

In your shell, set an `ATLAS_TOKEN` environment variable with your API token as the value.

```
export ATLAS_TOKEN=<USER TOKEN>
```

Step 2: Create New TFE Workspaces

For each workspace you're migrating, create a new workspace in your TFE organization to take over management of the infrastructure. Set the VCS repository and any necessary variable values appropriately. You can set team access permissions now or later, whichever is more convenient.

Do not perform any runs in these workspaces yet, and consider locking them to be sure. If an apply happens prematurely, you'll need to destroy the workspace and start the process over.

Step 3: Stop Terraform Runs

However you currently run Terraform to manage this infrastructure, stop. Let any current runs finish, then make sure there won't be more.

This might involve locking or deleting CI jobs, restricting access to the state backend, or just communicating very clearly with your colleagues.

Step 4: Prepare Your Terraform Working Directory

In your shell, go to the directory with the Terraform configuration you're migrating.

- If you've already been doing Terraform runs in this directory, you should be ready to go.
- If you had to retrieve state files from elsewhere, copy them into the root of the working directory and run `terraform init`.
- If you use a remote backend and had to check out a fresh working directory from version control, run `terraform init`.

Step 5: Migrate to the local Backend, if Necessary

If you use a remote backend that supports multiple workspaces, migrate to the local backend now. You need copies of each state file on disk, and this is the easiest way to get them.

To migrate to local, delete the `terraform { backend { ... } }` configuration block and run `terraform init`. Confirm that you want to copy existing state to the new local backend, even if there's already data in it. (Any existing local data is probably out of date.)

Step 6: Back Up Your State Files

Copy the following files to outside your working directory:

- `terraform.tfstate`
- `terraform.tfstate.d` (directory)

It's easy to delete data when switching backends and workspaces, and the local copy of the default workspace's state is usually deleted during the next steps. Save copies now in case you need to start over.

Step 7: Switch to the default Workspace

If you're not currently on the default workspace, run `terraform workspace select default` to activate it.

Step 8: Set the Backend Configuration for the Default Workspace

Add a `terraform { backend ...}` block to the configuration. Specify the TFE workspace that corresponds to the default workspace.

```
terraform {  
  backend "atlas" {  
    name = "<TFE ORG>/<WORKSPACE NAME>"  
    address = "https://<TFE HOSTNAME>"  
  }  
}
```

- Use the `atlas` backend. (This is TFE's backend; the `Atlas` name is used here for historical reasons.)
- In the `name` attribute, specify the name of your TFE organization and the target workspace, separated by a slash. (For example, `example_corp/database-prod`.)
- The `address` attribute is only necessary with private TFE instances. You can omit it if you're using the SaaS version of TFE.

Note: The `atlas` backend block is temporary; it isn't needed once TFE takes over Terraform runs. It's easiest to put the backend block in a separate `backend.tf` file which isn't checked into version control.

Step 9: Migrate default by Running `terraform init`

Run `terraform init` to migrate the default workspace.

Terraform will ask two questions:

- "**Do you want to copy only your current workspace?**" Answer "yes."

Terraform init can only handle the default workspace, so you'll use a different command for the others.

- "**Do you want to copy existing state to the new backend?**" Answer "yes."

The default workspace is now migrated.

You can test the migration by unlocking the corresponding TFE workspace and queueing a plan; if the plan results in no changes or very small changes, the migration probably worked correctly.

Step 10: Edit the Backend to Change Workspaces

Edit your backend configuration, and change the `name` to the next TFE workspace you want to migrate. For example, if the next local workspace is `dev1`, its new name in TFE might be something like `example_corp/database-dev1`.

At this point, it's a good idea to use a checklist of workspaces to keep track of your progress.

Step 11: Run `terraform init` and Answer "No"

Run `terraform init`.

When asked if you want to migrate state, **answer "no."**

```
Do you want to copy existing state to the new backend?  
Pre-existing state was found while migrating the previous "atlas" backend to the  
newly configured "atlas" backend. No existing state was found in the newly  
configured "atlas" backend. Do you want to copy this state to the new "atlas"  
backend? Enter "yes" to copy and "no" to start with an empty state.
```

If you answer yes, Terraform will copy the state from the last workspace you migrated, which is not what you want.

Step 12: Run `terraform state push <FILE>`

Locate the state file for the workspace you want to migrate. Its local path should be something like
`./terraform.tfstate.d/<WORKSPACE NAME>/terraform.tfstate`.

Run `terraform state push ./terraform.tfstate.d/<WORKSPACE NAME>/terraform.tfstate`.

The workspace is now migrated. You can check the migration by unlocking the TFE workspace and queuing a plan.

Step 13: Repeat Steps 10-12 as Needed

Repeat the last three steps for each remaining workspace, until every workspace is migrated.

Step 14: Enable Runs in the New Workspaces

After checking the results of the migrations, unlock each new workspace to allow TFE to take over management of this infrastructure.

Private Terraform Enterprise

Private Terraform Enterprise is software provided to customers that allows full use of Terraform Enterprise in a private, isolated environment.

This section is intended to help guide customers on how to properly install Private Terraform Enterprise. This includes required prerequisites, steps to install the software, and the basic configuration that will need to be done after install.

- Installation (Installer) (</docs/enterprise/private/install-installer.html>) - For customers using the Installer
- Installation (AMI) (</docs/enterprise/private/install-ami.html>) - For customers using the older AMI
- Configuration and Validation (</docs/enterprise/private/config.html>)
- Frequently Asked Questions (</docs/enterprise/private/faq.html>)

Please Note: This documentation is provided for customers who feel confident, after reading the full documentation, that they can successfully deploy Private Terraform Enterprise on their own. If you are unsure, or have questions, please talk to your Solutions Engineer (pre-sales, POC or trial) or Technical Account Manager (existing customers). If you have read the documentation and are ready to schedule your install, please inform your Sales Engineer (pre-sales, POC or trial) or Technical Account Manager (existing customers) of your install time window so they can make sure they are available if necessary.