This provides a good segway to discuss Consul.
<table>
<thead>
<tr>
<th>Agenda</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction to Consul</td>
</tr>
<tr>
<td>2. Architecture and overview</td>
</tr>
<tr>
<td>3. Registering nodes and services</td>
</tr>
<tr>
<td>4. Adding health checks</td>
</tr>
<tr>
<td>5. Constructing templates from Consul data</td>
</tr>
<tr>
<td>6. Further reading</td>
</tr>
</tbody>
</table>
Introduction
Consul has a number of goals. First and foremost, Consul is a service discovery and registration tool.

Those service discovery results are also closely tied to health checks of services.

Seemingly unrelated, Consul also provides a distributed key-value store.

And all of these things are accessible via the HTTP API.
**Consul vs. Other Tools**

- Opinionated framework for service discovery using DNS or HTTP
- Scalable gossip system that links Consul server nodes and clients
- Distributed health checking with edge triggered updates
- Globally aware with multi-datacenter support
- Operationally simple

Consul is very opinionated and slightly different than other software.
Consul has multiple components, but as a whole, it is a tool for discovering and configuring services in your infrastructure. It provides several key features:

- **Service Discovery**: Clients of Consul can provide a service, such as api or mysql, and other clients can use Consul to discover providers of a given service. Using either DNS or HTTP, applications can easily find the services they depend upon.

- **Health Checking**: Consul clients can provide any number of health checks, either associated with a given service ("is the webserver returning 200 OK"), or with the local node ("is memory utilization below 90\% "). This information can be used by an operator to monitor cluster health, and it is used by the service discovery components to route traffic away from unhealthy hosts.

- **KV Store**: Applications can make use of Consul's hierarchical key/value store for any number of purposes, including dynamic configuration, feature flagging, coordination, leader election, and more. The simple HTTP API makes it easy to use.

- **Multi Datacenter**: Consul supports multiple datacenters out of the box. This means users of Consul do not have to worry about building additional layers of abstraction to grow to multiple regions.
Glossary

Agent

Long-running daemon on every member of the Consul cluster. The agent is able to run in either client or server mode.
Glossary

Agent (Client)

Agent that forwards all RPCs to a server and participates in the LAN gossip pool.
**Agent (Server)**

Agent that maintains cluster state, responds to RPC queries, exchanges WAN gossip with other datacenters, and forwards queries to leaders of remote datacenters.

Always want an odd number of server agents (3, 5, 7, etc.)
Glossary

**Consensus**

Agreement upon the elected leader
Glossary

**Gossip**

Random node-to-node communication primarily over UDP that provides membership, failure detection, and event broadcast information to the cluster. Built on Serf. Consul has both LAN and WAN Gossip.
**Catalog**

The catalog is the "database" of all registered nodes, services, checks, and queries.
Glossary

**Datacenter**

Networking environment that is private, low latency, and high bandwidth. A Consul cluster is run per datacenter, so it's important to have low latency for the gossip protocol.
**Node**

A node represents the "physical" machine on which an agent is running. This could be a bare metal device, VM, or container.
Glossary

Service

A service is an application or process that is registered in Consul's catalog for discovery. Services may optionally have a health check, and they can be internal (inside your datacenter) or external (like an RDS cluster).
Glossary

Check
A check is a *locally run* command or operation that returns the status of the object to which it is attached. Checks may be attached to nodes and services.
Architecture
Here is the architecture for a single data center:

1. Clients communicate with each other via the gossip network
2. Clients communicate to any server via RPC calls
3. Server followers forward requests to the leader
4. The leader replicates data to the followers
In a multi-datacenter setup, the servers participate in the WAN gossip. If a client in the first data center wants to communicate to a client in the second data center, this communicate must go through the servers.
Agent Functionality
<table>
<thead>
<tr>
<th>Agent Functionality (Client &amp; Server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC, HTTP, DNS APIs</td>
</tr>
<tr>
<td>Health checks</td>
</tr>
<tr>
<td>Event execution</td>
</tr>
<tr>
<td>Gossip participation (membership &amp; failure detection)</td>
</tr>
</tbody>
</table>

All of the agents, regardless of being in a client or server mode provide the same set of RPC, HTTP, and DNS APIs. As a client of Consul, the difference is transparent. The queries are appropriately forwarded to a server, either in the local or a remote datacenter if appropriate. All the agents run health checks and handle even execution locally and push results to the servers as appropriate.

The Gossip pool the agents participate in is based on Serf. Serf can be used an independent Cli or embedded as a library like in Consuls case. It provides membership, failure detection and an event system.
The server nodes are the "brains" of the cluster. They maintain the state, and replicate data between themselves. They are also responsible for responding to queries. As a detail, they make use of the Raft Consensus protocol, which is a simplification of Paxos. They also participate in a WAN gossip pool, this means the servers are aware of the other data centers and can forward requests as appropriate.
Exercise: List `consul members`

SSH into your workstation using the provided credentials.

```bash
ssh velocity@<your.ip.address>
password: i<3london
```

There is already a Consul agent configured and running locally. Run the `consul members` command to see the registered agents in the cluster.
Here is a sample solution.
Bootstrapping Clusters
Server-Client Responsibilities

Consul agents can run in both client and server mode.

Server nodes are responsible for **consensus** and storing cluster state.

Client nodes are mostly stateless.

Servers do most of the "work".
Connecting Servers

One server node must be elected the **leader** before the cluster can service requests.

Usually the first nodes in the cluster are server nodes, but clusters can provision in parallel and use `-retry-join`.

Bootstrapping is the process of joining initial server nodes to form a cluster.
The **-bootstrap-expect** flag informs Consul of the expected number of servers to prevent inconsistencies and split-brain situations.

To prevent split-brain, all servers should specify the same value.

There should be an odd number of servers (3, 5, 7, etc.), and having a single server is highly discouraged.

The recommended way to bootstrap is to use the `-bootstrap-expect` configuration option. This option informs Consul of the expected number of server nodes and automatically bootstraps when that many servers are available. To prevent inconsistencies and split-brain situations (that is, clusters where multiple servers consider themselves leader), all servers should either specify the same value for `-bootstrap-expect` or specify no value at all. Only servers that specify a value will attempt to bootstrap the cluster.

To prevent inconsistencies and split-brain situations (that is, clusters where multiple servers consider themselves leader), all servers should either specify the same value for `-bootstrap-expect`. 
Server Recommendations

We recommend 3 or 5 server nodes per datacenter.

Single-server deployments are fine for local testing, but are highly discouraged in production.

We recommend 3 or 5 total servers per datacenter. A single server deployment is highly discouraged as data loss is inevitable in a failure scenario. Please refer to the deployment table for more detail.
This is a table that shows quorum size and failure tolerance for various cluster sizes. The recommended deployment is either 3 or 5 servers. A single server deployment is highly discouraged as data loss is inevitable in a failure scenario.
Consensus

Consul uses a consensus algorithm (Raft) to provide the CP behavior of CAP.

Only server agents participate in Raft.

Quorum is the majority of server members, and quorum is required to commit a Raft log entry (instruction).
Cluster Formation

To trigger a leader election, servers must be joined together to form a cluster.

Two options exist for cluster formation: manual, automatic.
Cluster Formation - Manual

1. Start one server (s1) with `-bootstrap-expect=N`
2. Capture s1 internal IP or DNS address
3. Start servers 2-N (sN) with same `-bootstrap-expect=N`
4. Execute `consul join <s1-address>` on each
Cluster Formation - Manual

Since the join operation is symmetric, it does not matter which node initiates it - \( s_1 \) could join \( s_2 \) or \( s_2 \) could join \( s_1 \).

Information is shared on join, so \( s_2 \) can join \( s_1 \) and \( s_3 \) can join \( s_2 \) to learn about \( s_1 \).

Servers will not service requests until \( N \) from \(-\text{bootstrap}-\text{expect}\) servers are joined.
Here is an example of manual joining.

```bash
$ consul join 1.2.3.4 5.6.7.8
```
Cluster Formation - Automatic

Requires cloud-provider-specific integrations.

Use `-retry-join` with a string of configuration options.

Supports AWS, Azure, Google Cloud, and IBM SoftLayer.
Here is some sample output for what auto-joining looks like on various technologies.

```
Terminal

$ consul agent \
  -retry-join "provider=aws tag_key=... tag_value=..."

$ consul agent \
  -retry-join "provider=azure tag_name=... tag_value=... tenant_id=..."

$ consul agent \
  -retry-join "provider=gce project_name=... tag_value=..."

$ consul agent \
  -retry-join "provider=softlayer datacenter=... tag_value=..."
```
Stop the Consul service (`sudo systemctl stop consul`).
Run `consul members` and observe result.
Start the Consul service (`sudo systemctl start consul`).
Run `consul members` again and observe result.
Terminal

```
$ sudo systemctl stop consul
consul stop/waiting

$ consul members
Error retrieving members: Get http://127.0.0.1:8500/v1/agent/members?segme...

$ sudo systemctl start consul
consul start/running, process 4305

$ consul members
```

<table>
<thead>
<tr>
<th>Node</th>
<th>Address</th>
<th>Status</th>
<th>Type</th>
<th>Build</th>
<th>Protocol</th>
<th>DC</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>server-0</td>
<td>10.1.1.169:8301</td>
<td>alive</td>
<td>server</td>
<td>0.9.3</td>
<td>2</td>
<td>dcl</td>
<td>&lt;all&gt;</td>
</tr>
<tr>
<td>server-1</td>
<td>10.1.2.16:8301</td>
<td>alive</td>
<td>server</td>
<td>0.9.3</td>
<td>2</td>
<td>dcl</td>
<td>&lt;all&gt;</td>
</tr>
<tr>
<td>server-2</td>
<td>10.1.1.199:8301</td>
<td>alive</td>
<td>server</td>
<td>0.9.3</td>
<td>2</td>
<td>dcl</td>
<td>&lt;all&gt;</td>
</tr>
<tr>
<td>goldfish</td>
<td>10.1.1.185:8301</td>
<td>alive</td>
<td>client</td>
<td>0.9.3</td>
<td>2</td>
<td>dcl</td>
<td>&lt;default&gt;</td>
</tr>
<tr>
<td>grasshopper</td>
<td>10.1.2.166:8301</td>
<td>alive</td>
<td>client</td>
<td>0.9.3</td>
<td>2</td>
<td>dcl</td>
<td>&lt;default&gt;</td>
</tr>
<tr>
<td>llama</td>
<td>10.1.1.148:8301</td>
<td>alive</td>
<td>client</td>
<td>0.9.3</td>
<td>2</td>
<td>dcl</td>
<td>&lt;default&gt;</td>
</tr>
</tbody>
</table>

Here is a sample solution.
What Happened?

How did we automatically join the cluster?
Inspect the configuration at /etc/consul.d/config.json.
This tells Consul to automatically join instances which have a tagged named "consul_join" with a value of "training". All your instances have been pre-configured with these tags, thus the cluster forms automatically.
Service Discovery
In microservice oriented architectures, applications are not guaranteed to live on the same hosts or IP addresses.

Containers complicate this even more, with overlay networks and software-defined networks.

Service discovery makes these services addressable, even as they move throughout the system.
Imagine an E-Commerce company we know nothing about. A reasonable service decomposition is to have a web front end, order processing, order history and forecasting services.

In a traditional application running on physical hardware, the network locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated.

In a modern, cloud-based microservices application, however, this is a much more difficult problem to solve as we add load balancers, reverse proxies, and dynamic assignments.

Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of auto-scaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.
The first problem when moving to an SOA model is discovery. How does the web app find the order processing system? This is the most basic question service discovery tries to solve.
Next, even if we can find the order processing system, how do we properly load balance requests to it?
A classic anti-pattern is to slam a load balancer in front, and hard code the address of it. This removes location transparency, meaning we cannot move things around, but also adds a single point of failure. This defeats the availability properties of an SOA system.
Typically the LB will do some form of health checking. If we remove the LB, we still have this problem. We must make sure requests are not routed to unhealthy hosts. So health checking is a core issue.
Service Discovery with Consul

Consul acts as a de-centralized registry for all services.
Service discovery is integrated with health checks.
Unhealthy hosts are removed from query results to the registry.
Multiple services are automatically load-balanced.
Service Discovery is at the heart of Consul. Consul uniquely blends service discovery with health checking, which is absolutely crucial. Registering services with Consul requires no code, instead a simple JSON file is provided that describes the service and optionally provides health checks. The checks are Nagios compatible, which means almost every conceivable service already has a pre-written check. Querying data out of Consul is done using either a DNS interface or the HTTP API.
Service Discovery (DNS)
Consul's DNS interface is **zero touch**

Randomized Round-Robin DNS

Filters on Health Checks

Using a DNS based discovery approach allows for a zero-touch integration with Consul. No application code changes. JSON files are provided to register services, and discovery is done transparently at the DNS level.

Additionally, the DNS records returned are randomized, so you get a round-robin form of load balancing. Entries for services that are failing health checks are automatically filtered out to avoid routing to unhealthy hosts.
DNS is a first-class mechanism to doing service discovery with Consul. Here we can issue a simple request to lookup one of the node's from before. Consul responds with an A record providing the IP address of the instance.
DNS Suffixes

.consul is the TLD

.node.consul queries for nodes.

.service.consul queries for logical services.

.query.consul executes prepared queries.
Exercise: Query Node

Query for a node's information using **dig** and the DNS interface.

**HINT:** You may need to use **consul members** to get a node name.
But these instances have dnsmasq installed and configured to listen on the ".consul" suffix, so you can omit the port.

dnsmasq is a common approach and examples are on Consul's website. We will discuss this more later. This is discussed in detail later.
It's also possible to add the "+short" flag, which will just return the IP addresses as a list, each on their own line. There are other tools like "host" too.
Registering Nodes

Nodes are registered by the agent when an agent joins the cluster.
Exercise: Query Service

Identify the IP address(es) of the service named "training" using the DNS interface.
Here is a sample answer. Again, I used "+short" here, but you could have used full dig or another tool.
Registering Services

Services are registered using JSON definition or via the HTTP API.

JSON files are put into a dot-d directory and loaded by Consul.

The previous exercise was not very exciting because we could have gotten the IP address directly just using `consul members`.

Consul's power really comes from the service registration and integrated discovery portion.
Here is an example service definition.

```json
{
  "service": {
    "name": "my-service",
    "tags": ["tag-1", "tag-2"],
    "port": 1234
  }
}
```
Exposed via DNS and HTTP APIs
Referred to as the "logical service" name

dig my-service.service.consul
Exposed via DNS and HTTP APIs

dig tag-1.my-service.service.consul
Exposed via DNS and HTTP APIs

dig SRV my-service.service.consul
Testing Service Configuration

Configuration errors sometimes happen (humans after-all).

Configuration syntax is verified using `consul validate`.

Each argument is the path to a file or folder to validate.
We can validate the configuration file using the validate command.
It's possible to specify multiple arguments and each will be validated.
Or just check the entire directory.
Detecting Changes

Consul does not automatically watch for changes to files in the configuration directory.

Consul must be informed and reloaded using the `consul reload` command.

Reloading is an online operation (no downtime).
Terminal

$ consul reload
Configuration reload triggered
Exercise: Register & Query Service

There is a pre-written service in /etc/consul.d/web.service.

Rename the file web.json.

Check the syntax using the validate command.

Query that service using the DNS interface.

HINT: Do not forget to reload consul!
Here is a sample solution.
$ mv /etc/consul.d/web.service /etc/consul.d/web.json
Then we need to reload the consul service to pick up the new configuration.

If the student restarted consul, they need to run `join` again.
Then we need to reload the consul service to pick up the new configuration.

If the student restarted consul, they need to run `join` again.
Notice that no results are returned. Why is that?

Instructor note: give the class some time to think about this.

Answer: Consul only returns **healthy** hosts in the DNS response. This means traffic is never routed to an unhealthy host. (More on this later)
If we query the registered services, we can see that "web" is now registered, so why is it not being returned?
Registering Checks

Checks are registered using JSON definition or via the HTTP API. Both host-level checks and service-level checks exist. JSON files are put into a dot-d directory and loaded by Consul.

**Unhealthy hosts and services are removed from discovery.**
A health check is any command that returns a specific status check. If the script exits with "0", it's considered passing. If the script exits as "1", it's considered "warning", and any other value is considered a failure.
The monitoring features are deeply integrated into the service discovery of Consul. When building Consul, we wanted to ensure that the checks would be compatible with the existing nagios ecosystem, but unlike nagios we wanted the checks to be scalable, devops friendly and actionable.

As a result, we spent a lot of time designing this feature. We wanted to ensure Consul could scale to thousands of machines, so instead of the central servers using SSH to run the checks remotely in effect a "Pull" model, Consul runs the health checks on the edges and uses a "Push" model. This allows for a major optimization which is to only do edge triggering. If a health check is in a stable state, we don’t need to update the servers. This provides multiple orders of magnitude reduction in write volume.
Here is an example host-level check. This check ensures the host has access to the public Internet by pinging a well-known IP address every 30s.
And here is an example of a service-level health check. This ensures that our service is responding on port 1234.
The **consul monitor** command will stream Consul's logs to the main terminal.

Sometimes the logs are on disk, but they may be streamed to an external service.

If we didn't know why the check was failing, we could look at the logs using the `consul monitor` command.
The consul logs will live where configured, but the consul monitor command does not care. Consul is running under upstart in these systems, so the log files are at /var/log/upstart/consul.log and you need sudo to read them. We don't need sudo to use consul members.
You’ll see a message like this every 5s. Press ctrl-c to exit.
The log is "info" by default, and a reload is at the debug level.
Exercise: Fix Health Check

Install a web server like nginx or apache so our health check starts to pass.
Terminal

$ sudo apt-get -y install nginx
Terminal

$ dig +short web.service.consul
10.1.1.148
10.1.1.185
10.1.2.106
Health Checks: Scaling

Consul's architecture allows for massive scale compared to traditional monitoring solutions.
One of the problems with traditional health checking is that it's a push-pull model.
Traditional Health Checking (pull)

The monitoring server is constantly asking all the nodes if they are healthy.
Traditional Health Checking (pull)

And the nodes are responding with a status
This happens at some regular interval
But the problem is that this creates a bottleneck in your system. As the number of nodes scales, so too does the amount of traffic to your monitoring service.
This can result of thousands of requests per minute. This clogs your local network traffic and puts unnecessary load on your cluster.
Consul health checking behaves slightly different. Consul uses a push-based model that only sends events on status changes.
Because statuses are only sent on change, it means that even a reasonably large cluster will have very few requests.
The problem with edge triggered monitoring is that we have no liveness heartbeats. Meaning in the absence of any updates, we don't know if the checks are simply in a stable state or if the server has died. Consul get's around this by using a gossip-based failure detector. All cluster members take part in a background gossip, which has a constant load regardless of cluster size. This enables Consul to scale to thousands of machines with negligible load on CPU or the network.
Forwarding DNS
DNS Defaults

By default, DNS runs on port 53.

This is a reserved port on most operating systems, which would require Consul run with very elevated privileges.

Instead we recommend running Consul's DNS on an unprivileged port and delegating a suffix to Consul.

Consul's DNS runs on port 8600 by default, but is configurable.
Here is an example BIND configuration. We assume Consul is running with default settings and is serving DNS on port 8600.
Here is an example configuration for dnsmasq. This will route anything in the .consul suffix to 127.0.0.1:8600.

This is how these machines are configured.
On Linux systems that support it, incoming requests and requests to the local host can use iptables to forward ports on the same machine without a secondary service. Since Consul, by default, only resolves the .consul TLD, it is especially important to use the recursors option if you wish the iptables setup to resolve for other domains. The recursors should not include the local host as the redirects would just intercept the requests.

The iptables method is suited for situations where an external DNS service is already running in your infrastructure and is used as the recursor or if you want to use an existing DNS server as your query endpoint and forward requests for the consul domain to the Consul server. In both of those cases you may want to query the Consul server but not need the overhead of a separate service on the Consul host.
To see this, query using `dig` but specify the "port".
Service Discovery (CLI)
Consul CLI provides basic interactions with service discovery

Do not use the CLI to build tooling (use API instead)

Allows easily querying services

Currently only supports the catalog

May want to explain the difference between the "agent" and "catalog" endpoints.
Exercise: Get Help

Run `consul catalog -h` to see the list of available CLI service discovery commands.
Usage: consul catalog <subcommand> [options] [args]

This command has subcommands for interacting with Consul's catalog. The catalog should not be confused with the agent, although the APIs and responses may be similar.

Here are some simple examples, and more detailed examples are available in the subcommands or the documentation.

List all datacenters:
$ consul catalog datacenters

List all nodes:
$ consul catalog nodes
Exercise: List Datacenters

List all registered datacenters using the service discovery CLI.
Here is a sample response. Note that in this training there is only one datacenter. If additional datacenters were joined by WAN, they would appear in this list.
Exercise: List Nodes with Service

List all the nodes which provide the service "web" using the service discovery CLI.
Here is a sample response. This shows all available nodes which provide the service. It does NOT filter by health.

```
$ consul catalog nodes -detailed -service=web
Node    ID      Address    DC
testing-llama  67ec06d2  10.1.1.148  dc1
```
Exercise: List Services with Tags

List all registered services with their tags using the service discovery CLI.
Here is a sample response.
Exercise: List Services on Node

List all services registered on a node using the service discovery CLI.

HINT: You can get the name of a node from `consul members`
Here is a sample response.

```bash
$ consul catalog services -node=llama
training
web
```
Service Discovery (API)
HTTP API

HTTP API is a JSON API
Allows easy construction of custom integrations
Edge-triggering gets "as close to real time" as possible
Many different consistency modes are supported

In addition to the DNS interface, there is also a rich HTTP API. The API can be used to deeper more intelligent integrations. This allow clients to customize their routing logic, connection pooling, etc. It also makes it simple to build tooling around Consul.
The HTTP API gives applications the most detailed amount of control over their interactions with Consul, but it also requires applications to be Consul-aware. This may not be ideal, since it can make local development more complex.

In a later section, we will talk about middle-ground solutions that do this work for us.
Here is an example of using the HTTP API to look for all services named web. Notice that a response is returned. This is different from the DNS API.
Appending `?pretty=true` to a URL will cause Consul to return formatted JSON.

`jq` is installed on these systems (recommended).

`jq` (a command-line JSON parser) is installed on these machines. You can pipe output to ``jq .` to format it pretty.
Always query the local agent (default: `127.0.0.1:8500`)

Avoids the need for applications to know about Consul servers

Local agents cache and multiplex connections to servers

When integrating with the HTTP API, it's tempting to want to communicate with the Consul server directly. **Do not do this!** This presents a catch-22 where you need to know the IP addresses of your Consul servers, which may change over time. Instead, you should always query `127.0.0.1:8500`. The local Consul client also caches and multiplexes connections to the servers.
Exercise: Query for Service

Query the HTTP API for all healthy (passing) services named training.
Terminal

```bash
$ curl http://127.0.0.1:8500/v1/health/service/training
{
  "Node": {
    "ID": "19efebce-4047-894a-1b8f-ddae07cd74ad",
    "Node": "goldfish",
    "Address": "10.1.1.185",
    "Datacenter": "dc1",
    "TaggedAddresses": {
      "lan": "10.1.1.185",
      "wan": "13.59.15.136"
    },
    "Meta": {
      "consul-network-segment": ""
    },
    "CreateIndex": 119,
    "ModifyIndex": 132
  }
}
```
Exercise: List Members

Query the HTTP for all members of the cluster.

This is the API the CLI uses for the `consul members` command.
$ curl http://127.0.0.1:8500/v1/agent/members
{
  "Name": "grasshopper",
  "Addr": "10.1.2.166",
  "Port": 8301,
  "Tags": {
    "build": "0.9.3:112c640",
    "dc": "dc1",
    "id": "c4881ac9-685d-5ae9-1f8d-ca291038e6f1",
    "role": "node",
    "segment": "",
    "vsn": "3",
    "vsn_max": "3",
    "vsn_min": "2"
  },
  "Status": 1,
  "ProtocolMin": 1,
Exercise: Reload Agent

Reload the Consul agent using the HTTP API.

This is the API the CLI uses for the `consul reload` command.
Terminal

$ curl --request PUT http://127.0.0.1:8500/v1/agent/reload
(empty response)
Almost all CLI commands are wrappers around the HTTP API.
You can build custom automations using the HTTP API.

Because all commands go through the HTTP API, it's easy to build your own custom integrations or solutions into Consul's HTTP API.
Many requests to Consul return additional metadata in the header. Information may be informative, diagnostic, or used for blocking queries.

Curl can print headers with the `--verbose` flag.
If we query for the headers using the verbose flag...

$ curl --verbose http://127.0.0.1:8500/v1/health/service/training
Consul will return extra information. I've highlighted the relevant information in bold here.

This information will become more important when we discuss blocking queries.
K/V Store
K/V Store

- Highly-available, globally accessible key-value store
- Folder-like architecture allows for easy organization
- Optional ACLs can enforce policy and access
- Accessible via HTTP API (no DNS interface)
- Can be used via the CLI or via a tool like `curl`
K/V Store: Use Cases

- Runtime configuration data (max connections)
- Secrets or sensitive application data (see also: Vault)
Here is an example of writing a value into the key-value store. The values for `<DATA>` and `<KEY>` should be replaced.

The top example shows using the CLI, and the bottom example shows using the API through a tool like curl.
Here is an example of writing a value of "test" into the key "example".

```bash
$ consul kv put example test
Success! Data written to: example

$ curl --request PUT --data 'test' http://localhost:8500/v1/kv/example
true
```
To read a value back out, we just curl without any arguments with the name of the key.
Here is an example from before with the key "example"
When using the CLI, you'll notice we get back the value as we put in, but not when interacting with the API. This is because, internally, contents are stored as base64 encoded. There is no requirement that the data stored in consul be representable as a string, so for safe transport and storage it is base64 encoded. The CLI automatically decodes this into text, but the API returns a JSON response.

You may also notice the API returns more information that the CLI.
We can ask for more detailed information by specifying the `-detailed` flag to the CLI.
Exercise: Create KV Data

Create two new key-value pairs in the store.

Keep in mind that everyone is using the same Consul servers, so choose a unique name that won't conflict with another user.

Read those values back out.
Here is a sample solution

```
$ consul kv put sethvargo/foo bar
Success! Data written to: sethvargo/foo

$ curl -X PUT -d 'test' http://localhost:8500/v1/kv/sethvargo/bar
true

$ consul kv get sethvargo/foo
bar

$ consul kv get sethvargo/bar
test
```
Exercise: Interact with KV

List detailed information about one of your keys.
List all the keys (but not their values) in the cluster.
List all keys that begin with the letter "s".
Here is one example of getting detailed information about a key.
Here is how to list all keys, but not their values.
Here is how to list all keys that begin with the letter s.

$ consul kv get -recurse s
Many OSS UIs are available
ACLs can enforce policy on paths
Data can be snapshotted and restored
Data can automatically be snapshotted (Enterprise feature)
Tools exist for backing up and replicating data
Policy Example

Try to read, list, or write to the **`vault/`** path in Consul's KV store.
The ACL token on these systems does not allow you to write to vault/ (which is where Vault's data is stored).
Blocking Queries
A blocking query waits for a change using long-polling.

Not all endpoints support blocking queries, but all are clearly documented.

Endpoints that support blocking queries will return an **X-Consul-Index** header.
Blocking Queries

The value of X-Consul-Index is the "current state".

Future requests set the index query string to wait for changes.

The HTTP request will "hang" until a change in the system occurs or a timeout is reached.

Timeout is configurable via the wait query parameter.
Blocking Queries

The return of a blocking request does not guarantee a change.

Timeouts and idempotent writes can trigger changes that do not alter the underlying structure.
Terminal

$ curl --verbose http://127.0.0.1:8500/v1/health/service/training
  # ...
  < X-Consul-Index: 522

$ curl http://127.0.0.1:8500/v1/health/service/training?index=522
  (hangs)
  (eventually a response returns)
Here's the same thing in graphic form. Initially, when an HTTP request is made to the Consul server...
Consul will immediately return the response along with some additional metadata. One piece of that metadata will be a WaitIndex.
Another request is then made to Consul with that WaitIndex parameter included in the request. This tells Consul to issue a "blocking query".
At this point, the HTTP request is still open and "hanging", similar to when your Internet is really slow and you're waiting for a page to render. But in this example, the server is intentionally holding the response, waiting for a state change in the system.
When there is a state change, Consul "renders" that open HTTP request, immediately sending the response to the client. This gives us as close to real-time reactive infrastructure as possible.
Consistency Modes
Consistency Modes

Most read query endpoints support multiple levels of consistency:

- default
- consistent
- stale

Write endpoints only support strong consistency.
The *default* consistency mode (unspecified) is almost always "strong"

Small window during leader election where stale data is served

Minimal effort on clients

Balance: edge case for stale reads being non-consistent
Consistency Mode: consistent

The **consistent** mode requires strong consistency with no caveats.

Requires a leader verify with a quorum of peers that it is still leader.

Introduces additional round trip time.

Balance: increased latency, guaranteed freshness.
Consistency Mode: stale

The **stale** mode allows any server to respond to a read request, even if it is not the leader.

Reads can be arbitrarily stale (configurable at request-time).

Servers can respond without an elected leader.

Balance: fast and scalable reads, higher probability of stale values.
Consistency Modes

Specified at request-time via a query parameter:

- default = (none)
- consistent = ?consistent
- stale = ?stale
Here is an example of a consistent query. Notice the "Lastcontact" header is 0. That means the node we queried was either the leader or was replicated very recently.
In this example, the stale query is 23ms stale.

**Instructor note:** this is actually very hard to reproduce on a cluster of this size. Usually it'll be 0, even with stale queries, because of the limited number of nodes and the size of the instances.
Consul UI

There is a basic UI that comes packaged with Consul.

There is a community-maintained UI for both Consul and Nomad called hashi-ui.
Exercise: Visit UI

Visit https://consul.hashicorp.rocks/ in your browser.
Exercise: Use Consul UI

Browse the Consul UI.
Create, update, and delete keys from the key-value store.
Examine health check output from various services.
Consul Template
About Consul Template

Consul Template handles the HTTP API flow with Consul
Retrieves keys and services from Consul and renders them into a template
Optionally integration with HashiCorp Vault as well
Consul Template takes an input template, parses the template and queries the resulting dependencies, and then renders the resulting template to disk at the specified path. Optionally, Consul Template can run a command whenever the output template changes.
Consul Template is given an input template like this. The `key` key tells Consul Template to read from the given key in Consul's KV store. Consul Template makes a request to the Consul server to retrieve the information. The Consul server returns the data back to Consul Template. Consul Template then writes this value to disk and listens for edge-triggered updates from Consul.
Exercise: Create Template

Create and execute a Consul Template template that iterates over all the healthy services named "web" and prints out the IP address.

HINT: Consul Template's documentation is very verbose and probably has examples that you can follow.
Here is a sample solution. Note that the health filtering is built in, so we do not need to explicitly filter "healthy".
Now we can run this command using the consul-template process.

```bash
$ consul-template -template=in.tpl:out.txt
```
Consul Template runs in the foreground, so we will need to stop it.
Finally, we can print the output of the out.txt file to see what was rendered.
Envconsul
Envconsul handles the HTTP API flow with Consul.

Retrieves keys (but not services) from Consul and places their values in environment variables which are injected into a subprocess.

Optionally integration with HashiCorp Vault as well.
Envconsul takes a list of secret paths in Vault and injects them into the subprocess environment. This is very handy for twelve-factor applications.
Envconsul is given a list of prefixes to watch in Consul. It makes a request to the Consul server to retrieve the information. The Consul server returns the data back to Envconsul. Envconsul then executes a subprocess (command) injecting the responses from Consul into the subprocess' environment variables.
Exercise: Run Envconsul

Execute `envconsul` with the `env` command and the prefix "/".
Terminal

$ envconsul -prefix="/" env
TERM=xterm-256color
SHELL=/bin/bash
SSH_TTY=/dev/pts/1
USER=training
foo=bar
zip=zap

Here is an example with the response. You may notice that the result includes the parent process environment variables and is also lowercase. Let's fix that.
Exercise: Tune Envconsul

Execute `envconsul` on the prefix "/" with the `env` command.
Ensure all environment variable keys are uppercase.
Do not include any of the parent processes environment variables.
Hint: `envconsul -h`
We need to use the `-upcase` and `-pristine` commands to do this.
Exercise: Interactive Envconsul

Execute `envconsul` on the prefix "/" with an interactive shell such as `/bin/bash`.

Ensure all environment variable keys are uppercase.

Do **not** include any of the parent processes environment variables.
Here, you are in a subshell of envconsul. You can run the `env` command to see
Nearest Neighbor
Calculating Round Trip Times

Consul 0.6+ has the ability to measure "distances" between nodes using physics.

The `consul rtt` command accepts two node IDs and returns the estimated round-trip time between them.
$ consul rtt node1 node2
Estimated node1 <-> node2 rtt: 2.347 ms (using LAN coordinates)
Exercise: Calculate RTT

Calculate the round-trip time between any two nodes.

HINT: You may need to use `consul members` to find node IDs.
Here is a sample response. You can grab any two node IDs from the consul members output.
Time permitting, you may want to skip this section.
Orchestration is the last, but also the least solved issue. There is not broad adoption of specific tooling for this problem. Likely because it's the easiest to ignore.

When it comes to Orchestration, Consul provides an entire toolkit of features. All of these features "just work" out of the box, so that developers and operators don’t have to write code to get started.
Watches are the simplest way to react to changes using Consul. A watcher is basically a pairing between a "view of data" and a custom handler. The "data" you care about can be KV data, services, nodes, health checks, or custom events. The handler can be any executable, letting operators customize behavior. Watches can be configured either statically using configuration or with the "consul watch" command dynamically.

- Watch for changes in K/V, services, nodes, checks, and events
- Execute custom handlers on changes
As an example, a watcher can be setup to watch for changes to the nodes list. Any time a new node joins or leaves the cluster, the handler will be invoked. If we invoke it without a handler, we just get the current result set. We can also specify a handler which gets updates over stdin and is free to react in any way. For example, we could update our BIND configuration.
Building on watches, Consul includes a flexible event system that can be used for any purpose. Events are fired with a name and payload, and handlers can be registered to respond to events. Events can filter on services, nodes and tags, so that you can target the machines that should process the event. Because events are built on watches, they operate in the same way. Either statically configured or using consul watch.
As an example, here we are firing an event named "test", targeting the "redis" service and providing a "sample-payload". If we had been running a watcher, it could filter to only "test" events. Here we can see the handler receives the event, with the base64 encoded payload.
In addition to the generic event system, Consul also provides a remote execution tool. This allows operators to invoke commands like parallel SSH, while filtering on services, tags, or nodes. This can be used to avoid writing a specific event handler when a more generic remote exec is suitable.
This is a simple example of running the "uptime" command against any node that is providing the "redis" service. The event system is used to notify nodes of the remote execution, and then the request and all the response data is sent back using the KV store. This avoids an expensive parallel SSH to all the nodes, and allows for very efficient execution.
Lastly, to support more complex orchestration needs, Consul supports distributed locking. This is supported within the Key/Value store and allows applications to implement client-side leader election or semaphores. This can either be integrated within a client application or lock command can be used. Ultimately this frees developers from needing to develop their own consensus protocols.
This is a simple example of running the "lock" command with the "foo" service.
The Key/Value prefix of "service/foo/lock" is used to store the lock itself, and once the lock is acquired the foo service is invoked. As you can see, only a single instance is running on "node1". If we killed that node, Consul would handle the failover to "node2".
If we run the command again, it will block because the lock cannot be acquired.
Terminal

server-01: $ consul lock -n=3 kernel/upgrade/lock sudo reboot
System is shutting down...

server-02: $ consul lock -n=3 kernel/upgrade/lock sudo reboot
System is shutting down...

server-03: $ consul lock -n=3 kernel/upgrade/lock sudo reboot
System is shutting down...

server-04: $ consul lock -n=3 kernel/upgrade/lock sudo reboot
<blocks until the semaphore is released...>
Terminal

$ consul exec consul lock -n=3 kernel/upgrade/lock sudo reboot
server-01: System is shutting down...
server-02: System is shutting down...
server-03: System is shutting down...
<time passes>
server-04: System is shutting down...
This provides a good segway to discuss Consul.