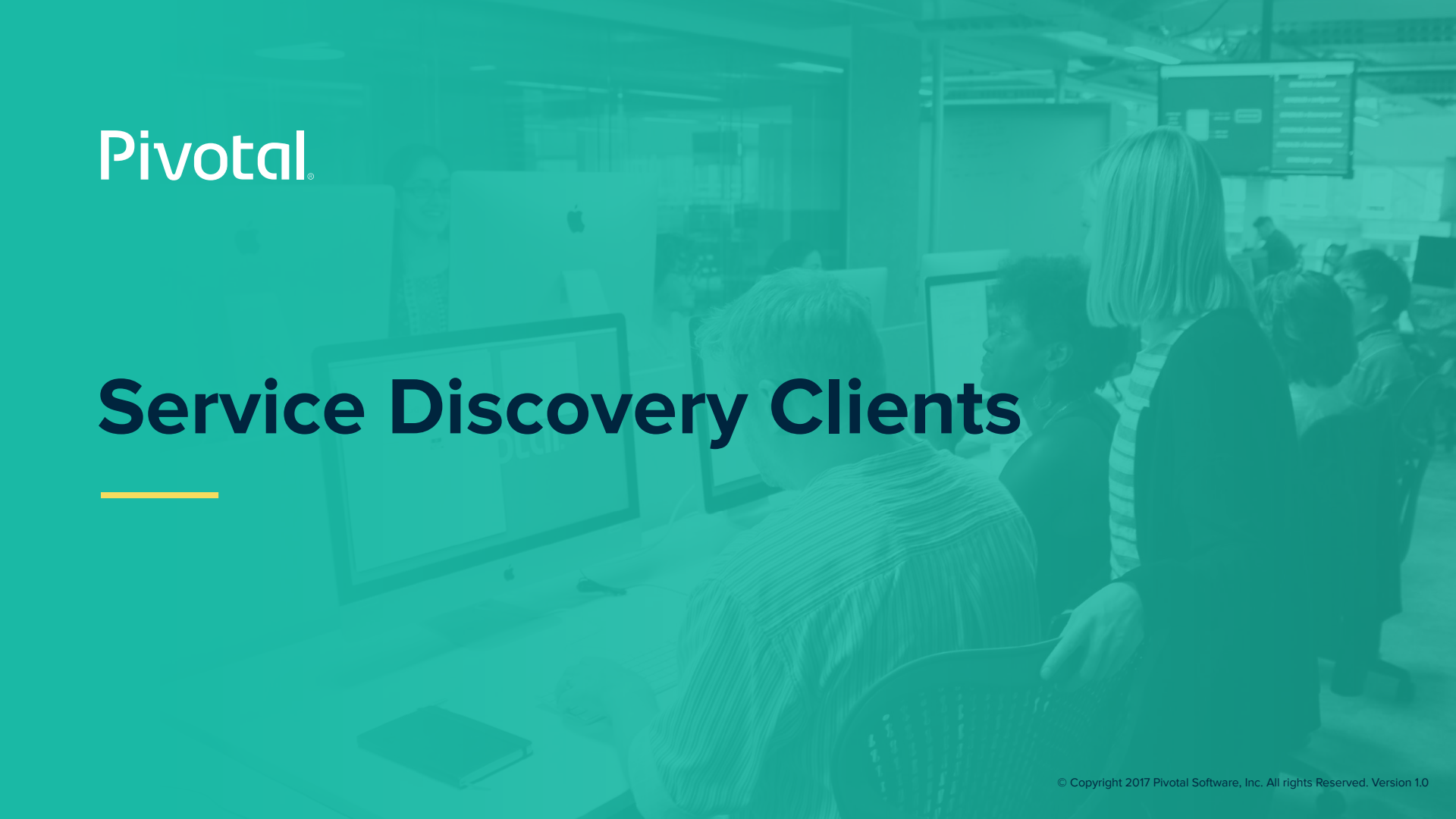
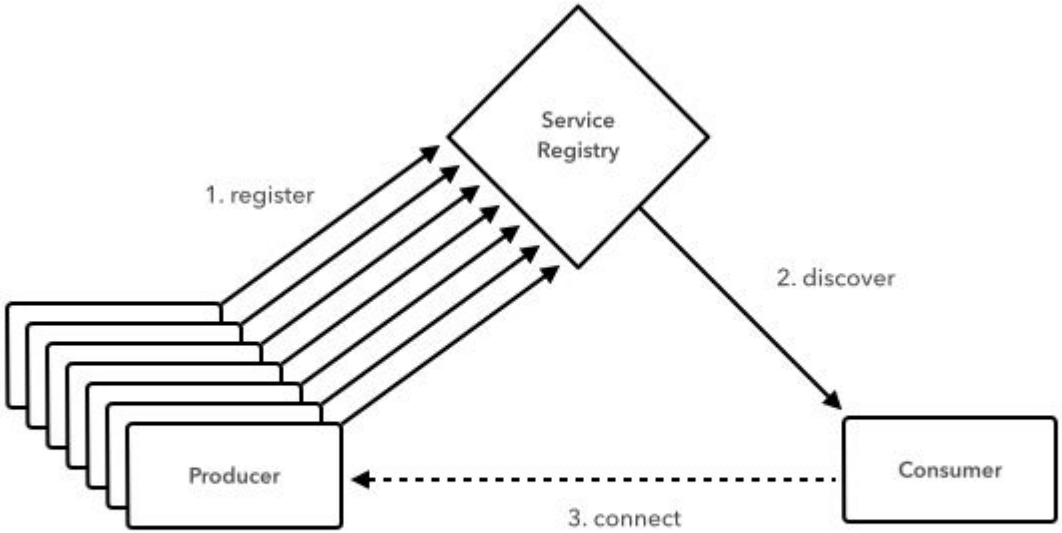
Pivotal

Service Discovery Clients



Service Registry Revisited



Using Service Registry

- *Consumers* and *Producers* need to talk to a given *Service Registry* implementation.
- The *Consumers* and *Producers* should have a simple, consistent mechanism to talk to its *Service Registry* implementation.

Single Abstraction for Multiple Service Registries

- The idea is to abstract away Service Registry implementation from the *Consumers* and *Producers*.
- *DiscoveryClient* interface
- Currently supported service discovery client implementations:
  - Eureka
  - Zookeeper
  - Consul
  - Kubernetes

DiscoveryClient

- Single client abstraction `DiscoveryClient`
- Simplified interface to fetch list of endpoints and associated services

```
List<ServiceInstance> getInstances(String serviceId);
```

```
List<String> getServices();
```

- Conforms to SRP and ISP principles, no client load balancing

Adapters

- Each Spring Cloud Discovery implementation provides adapter of `DiscoveryClient`:
  - Eureka: EurekaDiscoveryClient -> EurekaClient
  - Zookeeper: ZookeeperDiscoveryClient -> ServiceDiscovery
  - Consul: ConsulDiscoveryClient -> ConsulClient
  - Kubernetes: KubernetesDiscoveryClient -> KubernetesClient

Discovery by Composition

- Default discovery implementation is a composite client lookup mechanism `CompositeDiscoveryClient`
- Aggregates lookups from wired discovery clients

Trade-Offs of Using DiscoveryClient vs. EurekaClient

- `EurekaClient`
  - Tightly couples to a specific single service registry…
  - …But gives basic round-robin load-balancing logic

    `getNextServerFromEureka()`
- Spring Cloud `DiscoveryClient`
  - Abstracts details of service registry from clients
  - Allows multiple service registry implementations with composite discovery client
  - No client load balancing
  - Must explicitly use `@LoadBalanced` or `LoadBalancerClient` interceptors