Complete Consul Integration Explanation - From Beginner to Expert

Table of Contents

---

1. What is Service Discovery and Why Do We Need It? {#what-is-service-discovery}

Traditional Approach (What You Had Before)

```json
// Your old appsettings.json
"product-cluster": {
    "Destinations": {
        "product-destination": {
            "Address": "http://localhost:5001/"  // HARDCODED!
        }
    }
}
```

Problems with this approach:

- If Product API moves to port 5003, you must manually update the gateway
- If you run multiple instances of Product API, you must manually add each one
- If a service crashes, gateway keeps trying to send requests to it
- No health monitoring

Service Discovery Approach (What We Built)

- Services tell Consul: "Hey, I'm Product API and I'm running on port 5001"
- Gateway asks Consul: "Where are all the Product APIs?"
- Consul responds: "Here's a list of healthy Product APIs and their addresses"
- Everything updates automatically!

---

2. What is Consul? {#what-is-consul}

Think of Consul as a **phone book for your microservices**:

- **Service Registration**: Services add themselves to the phone book
- **Health Checking**: Consul calls each service regularly to check if it's still alive
- **Service Discovery**: Other services can look up addresses in the phone book
- **Key/Value Store**: Can also store configuration (we're not using this feature)

When you run `consul agent -dev -ui`:

- `agent`: Starts Consul as a service
- `-dev`: Development mode (stores data in memory, resets when stopped)
- `-ui`: Enables the web interface at http://localhost:8500

---

3. The Problem We're Solving {#the-problem}

Your Original Architecture

```
Client → Gateway (Port 5000) → Product API (HARDCODED: 5001)
                             → Order API   (HARDCODED: 5002)
```

Problems:

1. **Static Configuration**: Ports are hardcoded in appsettings.json
2. **No Fault Tolerance**: If a service dies, gateway doesn't know
3. **No Load Balancing**: Can't easily add multiple instances
4. **Manual Updates**: Change a port? Update config files manually

Our Solution with Consul

```
Client → Gateway (Port 5000) → Consul (Where is Product API?)
                             ← Consul (It's at localhost:5001)
                             → Product API (Dynamic discovery!)
```

---

4. How Our Solution Works - The Big Picture {#big-picture}

Step 1: Services Register Themselves

When Product API starts:

1. It tells Consul: "I'm product-api running at localhost:5001"
2. It also says: "Check my health at http://localhost:5001/health"
3. Consul adds it to its registry

Step 2: Gateway Discovers Services

Every 5 seconds, the Gateway:

1. Asks Consul: "What services are available?"
2. Gets a list of healthy services with their addresses
3. Updates its routing table dynamically

Step 3: Health Monitoring

Every 10 seconds, Consul:

1. Calls each service's /health endpoint
2. If a service doesn't respond, marks it as unhealthy
3. Unhealthy services are removed from discovery

---

5. Deep Dive: Order/Product API Code {#api-code-explanation}

Let's understand every line of `ConsulRegistration.cs`:

csharp

```csharp
public class ConsulRegistration : IHostedService
{
    private readonly IConfiguration _config;
    private readonly IConsulClient _consul;
    private string _serviceId;
```

What's happening here:

- `IHostedService`: This is a .NET interface that lets our code run when the app starts/stops
- `IConfiguration`: Gives us access to appsettings.json and other config
- `IConsulClient`: This is how we talk to Consul
- `_serviceId`: Unique ID for this specific instance (important for multiple instances)

csharp

```csharp
public ConsulRegistration(IConfiguration config)
{
    _config = config;
    _consul = new ConsulClient(c => c.Address = new Uri("http://localhost:8500"));
}
```

Constructor explanation:

- Gets configuration injected by .NET's dependency injection
- Creates a Consul client pointing to localhost:8500 (where Consul is running)

csharp

```csharp
public async Task StartAsync(CancellationToken cancellationToken)
{
    var uri = new Uri(_config["urls"] ?? "http://localhost:5002");
    _serviceId = $"order-{Guid.NewGuid()}";
```

StartAsync - Called when your API starts:

- `_config["urls"]`: Gets the URL your service is running on (from launchSettings.json)
- `?? "http://localhost:5002"`: If no URL configured, use this default
- `Guid.NewGuid()`: Creates unique ID like "order-a4f2c3d1-2345-6789"
- Why unique ID? So you can run multiple instances of the same service

csharp

```csharp
    var registration = new AgentServiceRegistration
    {
        ID = _serviceId,                        // Unique instance ID
        Name = "order-api",                     // Service type name
        Address = uri.Host,                     // "localhost"
        Port = uri.Port,                        // 5002
        Check = new AgentServiceCheck
        {
            HTTP = $"{uri}health",              // "http://localhost:5002/health"
            Interval = TimeSpan.FromSeconds(10)  // Check every 10 seconds
        }
    };
```

Registration object:

- This is all the info Consul needs about your service
- `Name`: Groups similar services (all order APIs have same name)
- `ID`: Identifies this specific instance
- `Check`: Tells Consul how to verify the service is healthy

csharp

```csharp
    await _consul.Agent.ServiceRegister(registration);
}
```

Actually registers with Consul:

- Sends all the registration info to Consul
- After this, your service appears in Consul's registry

csharp

```csharp
public async Task StopAsync(CancellationToken cancellationToken)
{
    await _consul.Agent.ServiceDeregister(_serviceId);
}
```

StopAsync - Called when your API stops:

- Removes this service from Consul

- Important for clean shutdown
- Prevents Consul from trying to route to dead services

In Program.cs:

csharp

```csharp
builder.Services.AddHealthChecks();
builder.Services.AddHostedService<ConsulRegistration>();
```

What these do:

- `AddHealthChecks()`: Adds the /health endpoint that returns 200 OK
- `AddHostedService<ConsulRegistration>()`: Tells .NET to run our ConsulRegistration

csharp

```csharp
app.MapHealthChecks("/health");
```

Creates the health endpoint:

- When accessed, returns 200 OK if service is running
- Consul uses this to check if your service is alive

---

6. Deep Dive: Gateway Code {#gateway-code-explanation}

The Gateway code is more complex because it needs to:

1. Discover services from Consul
2. Update YARP's routing configuration dynamically
3. Handle the authorization policies

Let's break down `ConsulConfig.cs`:

csharp

```csharp
public class ConsulConfig : BackgroundService, Yarp.ReverseProxy.Configuration.IProxyConfigProv
{
```

Class declaration:

- `BackgroundService`: Runs continuously in the background

- `IProxyConfigProvider`: Interface that YARP uses to get routing configuration

csharp

```csharp
private readonly IConsulClient _consul = new ConsulClient(c => c.Address = new Uri("http://loca
private volatile Config _config = new Config(
    new List<Yarp.ReverseProxy.Configuration.RouteConfig>(),
    new List<Yarp.ReverseProxy.Configuration.ClusterConfig>()
);
```

Fields:

- `_consul`: Client to talk to Consul
- `volatile`: Keyword that ensures changes are visible across threads
- `_config`: Current routing configuration (starts empty)

csharp

```csharp
public Yarp.ReverseProxy.Configuration.IProxyConfig GetConfig() => _config;
```

GetConfig:

- YARP calls this to get current routing configuration
- Returns our `_config` object

csharp

```csharp
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
```

ExecuteAsync:

- Runs continuously in the background
- `stoppingToken`: Signals when the app is shutting down
- The while loop keeps running until the app stops

csharp

```csharp
var services = await _consul.Agent.Services();
```

Getting services from Consul:

- Asks Consul for ALL registered services
- Returns a dictionary of services with their details

csharp

```csharp
var routes = new List<Yarp.ReverseProxy.Configuration.RouteConfig>
{
    new Yarp.ReverseProxy.Configuration.RouteConfig
    {
        RouteId = "product-route",
        ClusterId = "product-cluster",
        AuthorizationPolicy = "AdminPolicy",
        Match = new Yarp.ReverseProxy.Configuration.RouteMatch
        {
            Path = "/api/product/{**catch-all}"
        }
    },
```

Creating routes:

- These define URL patterns and which cluster handles them
- `RouteId`: Unique identifier for this route
- `ClusterId`: Which cluster of servers handles this route
- `AuthorizationPolicy`: Your security policy (Admin users only)
- `Path`: URL pattern - `{**catch-all}` means "anything after /api/product"

csharp

```csharp
var productServices = services.Response.Values.Where(s => s.Service == "product-api").ToList();
if (productServices.Any())
{
    clusters.Add(new Yarp.ReverseProxy.Configuration.ClusterConfig
    {
        ClusterId = "product-cluster",
        Destinations = productServices.ToDictionary(
            s => s.ID,
            s => new Yarp.ReverseProxy.Configuration.DestinationConfig
            {
                Address = $"http://{s.Address}:{s.Port}/"
            }
        )
    });
}
```

Creating clusters from Consul data:

- Filters services to find all "product-api" instances
- Creates a cluster with all healthy instances
- `ToDictionary`: Converts list to dictionary with ID as key
- Each destination is a specific instance of your service

csharp

```csharp
var oldConfig = _config;
_config = new Config(routes, clusters);
oldConfig.SignalChange();
```

Updating configuration:

- Saves old config
- Creates new config with updated routes/clusters
- `SignalChange()`: Tells YARP the config has changed

csharp

```csharp
await Task.Delay(5000, stoppingToken);
```

Wait before next update:

- Pauses 5 seconds before checking Consul again
- Prevents overwhelming Consul with requests

The Config Inner Class:

```csharp
private class Config : Yarp.ReverseProxy.Configuration.IProxyConfig
{
    private readonly CancellationTokenSource _cts = new CancellationTokenSource();
```

Config class:

- Implements YARP's config interface
- `CancellationTokenSource`: Used to signal when config changes

```csharp
public IChangeToken ChangeToken { get; }

public void SignalChange() => _cts.Cancel();
```

Change notification:

- `ChangeToken`: YARP watches this to know when config changes
- `SignalChange()`: Triggers the token, telling YARP to reload config

---

7. The Complete Flow {#complete-flow}

Let me walk you through exactly what happens when everything runs:

1. Starting Consul

```bash
consul agent -dev -ui
```

- Consul starts and listens on port 8500
- Web UI available at http://localhost:8500
- Ready to accept service registrations

2. Starting Order API

```
1. Order API starts on port 5002
2. ConsulRegistration.StartAsync() runs
3. Creates registration with ID "order-a4f2c3d1-2345-6789"
4. Sends registration to Consul
5. Consul adds to registry and starts health checks
```

3. Starting Product API

```
Same process but with:
- Port 5001
- Service name "product-api"
- Different unique ID
```

4. Starting Gateway

```
1. Gateway starts on port 5000
2. ConsulConfig background service starts
3. First update cycle:
   - Queries Consul for all services
   - Finds "order-api" and "product-api"
   - Creates routes and clusters
   - Updates YARP configuration
4. Repeats every 5 seconds
```

5. Client Makes Request

```
1. Client: GET http://localhost:5000/api/product
2. Gateway: Matches route "/api/product/{**catch-all}"
3. Gateway: Route says use "product-cluster"
4. Gateway: Cluster has destination "http://localhost:5001/"
5. Gateway: Forwards request to Product API
6. Product API: Processes and returns response
7. Gateway: Returns response to client
```

6. When Product API Port Changes

```
1. Stop Product API
2. Change port to 5003 in launchSettings.json
3. Start Product API
4. Product API registers with new port
5. Gateway's next update cycle (within 5 seconds):
   - Gets updated info from Consul
   - Updates cluster with new address
6. Next request automatically goes to port 5003!
```

---

8. How to Customize {#customization}

Adding a New Service (e.g., Customer API)

Step 1: Create CustomerApi project with ConsulRegistration.cs:

csharp

```
// Change the service name and port
_serviceId = $"customer-{Guid.NewGuid()}";
var registration = new AgentServiceRegistration
{
    Name = "customer-api",  // Different service name
    Port = 5003,           // Different port
    // ... rest same
};
```

Step 2: Update Gateway's ConsulConfig.cs:

csharp

```csharp
// Add new route
new RouteConfig
{
    RouteId = "customer-route",
    ClusterId = "customer-cluster",
    AuthorizationPolicy = "UserPolicy",  // Or whatever policy
    Match = new RouteMatch { Path = "/api/customer/{**catch-all}" }
}

// Add cluster discovery
var customerServices = services.Response.Values.Where(s => s.Service == "customer-api").ToList(
if (customerServices.Any())
{
    clusters.Add(new ClusterConfig
    {
        ClusterId = "customer-cluster",
        Destinations = customerServices.ToDictionary(/*...*/)
    });
}
```

Running Multiple Instances

For load balancing, just start multiple instances:

bash

```bash
# Terminal 1: Order API on port 5002
dotnet run

# Terminal 2: Order API on port 5003
dotnet run --urls="http://localhost:5003"

# Both register as "order-api" with different IDs
# Gateway automatically load balances between them!
```

Changing Health Check Interval

csharp

```
Check = new AgentServiceCheck
{
    HTTP = $"{uri}health",
    Interval = TimeSpan.FromSeconds(30),  // Check every 30 seconds
    Timeout = TimeSpan.FromSeconds(5),    // Timeout after 5 seconds
    DeregisterCriticalServiceAfter = TimeSpan.FromMinutes(1)  // Remove if unhealthy for 1 minu
}
```

Using Configuration Files

In appsettings.json:

json

```
{
  "Consul": {
    "Address": "http://localhost:8500",
    "CheckInterval": 10,
    "UpdateInterval": 5
  }
}
```

In code:

csharp

```
var consulAddress = _config["Consul:Address"];
var checkInterval = int.Parse(_config["Consul:CheckInterval"]);
```

---

9. Common Patterns and Best Practices {#best-practices}

1. Service Naming Convention

csharp

```
// Good: Descriptive, lowercase, hyphenated
"product-api"
"order-service"
"customer-api"

// Bad: Inconsistent
"ProductAPI"
"ORDERSERVICE"
"CustomerApi"
```

2. Health Check Best Practices

csharp

```
// Basic health check (what we have)
app.MapHealthChecks("/health");

// Advanced health check with dependencies
builder.Services.AddHealthChecks()
    .AddSqlServer(connectionString)  // Check database
    .AddRedis(redisConnection)       // Check Redis
    .AddCheck("Custom", () =>
    {
        // Custom logic
        return HealthCheckResult.Healthy();
    });
```

3. Environment-Specific Configuration

csharp

```
public async Task StartAsync(CancellationToken cancellationToken)
{
    var environment = Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT");

    var registration = new AgentServiceRegistration
    {
        Name = $"order-api-{environment}",  // order-api-dev, order-api-prod
        Tags = new[] { environment, "api", "v1" }  // Tags for filtering
    };
}
```

4. Handling Consul Failures

```csharp
try
{
    await _consul.Agent.ServiceRegister(registration);
}
catch (Exception ex)
{
    // Log error but don't crash the service
    Console.WriteLine($"Failed to register with Consul: {ex.Message}");
    // Service continues running even if Consul is down
}
```

5. Load Balancing Strategies

In Gateway's ConsulConfig, you can add:

```csharp
clusters.Add(new ClusterConfig
{
    ClusterId = "product-cluster",
    LoadBalancingPolicy = LoadBalancingPolicies.RoundRobin,  // Default
    // Other options: Random, LeastRequests, PowerOfTwoChoices
    Destinations = productServices.ToDictionary(/*...*/)
});
```

Summary

What we built is a **dynamic service discovery system** where:

1. **Services self-register**: Each API tells Consul where it is
2. **Health monitoring**: Consul checks services are alive
3. **Dynamic discovery**: Gateway finds services from Consul
4. **Automatic updates**: Changes propagate automatically
5. **Load balancing**: Multiple instances work automatically
6. **Fault tolerance**: Dead services are removed automatically

This is how modern microservices communicate - no hardcoded addresses, everything dynamic and resilient!

The beauty is that you can now:

- Change ports without updating config files
- Add/remove service instances on the fly
- Monitor service health in Consul UI
- Scale services up/down easily
- Deploy to different environments without code changes

This is production-ready patterns used by companies like Netflix, Uber, and many others! routing configuration

- Returns our `_config` object

csharp

```csharp
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
```

ExecuteAsync:

- Runs continuously in the background
- `stoppingToken`: Signals when the app is shutting down
- The while loop keeps running until the app stops

csharp

```csharp
var
```