# Desinging and implementing Microservices

# 1. Design patterns

# Reviewing Behavioral Patterns

- Behavioral patterns focus on algorithms and the distribution of responsibility between classes.

| Pattern | Primary Function |
| --- | --- |
| Strategy | Encapsulates a family of algorithms for interchangeable use |
| Command | Encapsulates requests in objects for execution by another object |
| Iterator | Traverses any collection in a loosely coupled way |
| Observer | Provides notification of events without polling |

# Reviewing Creational Patterns

- Creational patterns hide the details of object creation.

| Pattern | Primary Function |
| --- | --- |
| Factory Method | Creates objects of a class or its subclasses through a method call |
| Abstract Factory | Creates a family of objects through a single interface |
| Singleton | Restricts a class to one globally accessible instance |

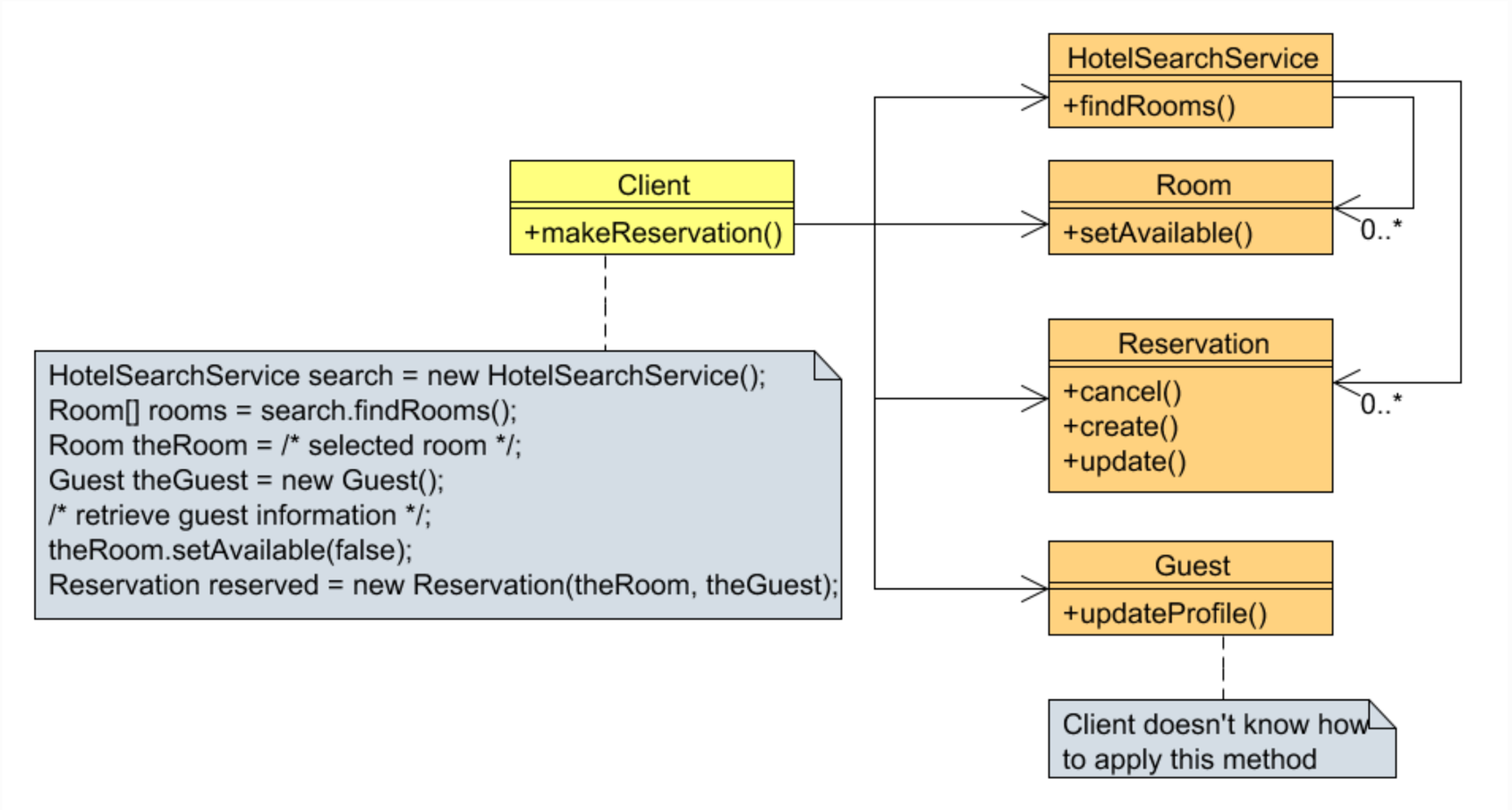
# Reviewing Structural Patterns

- Structural patterns define ways to compose classes that add or modify functionality.

| Pattern | Primary Function |
|---|---|
| Facade | Provides a simplified interface to a subsystem |
| Proxy | Provides an intermediate object that controls access to another object |
| Adapter | Enables a caller to use an object that has an incompatible interface |
| Composite | Composes objects into part-whole tree structures |
| Decorator | Attaches new functionality to an object dynamically |

# Applying the Facade Pattern: Example Problem

- Client is coupled to many classes in the reservation subsystem.



# Applying the Facade Pattern: Problem Forces

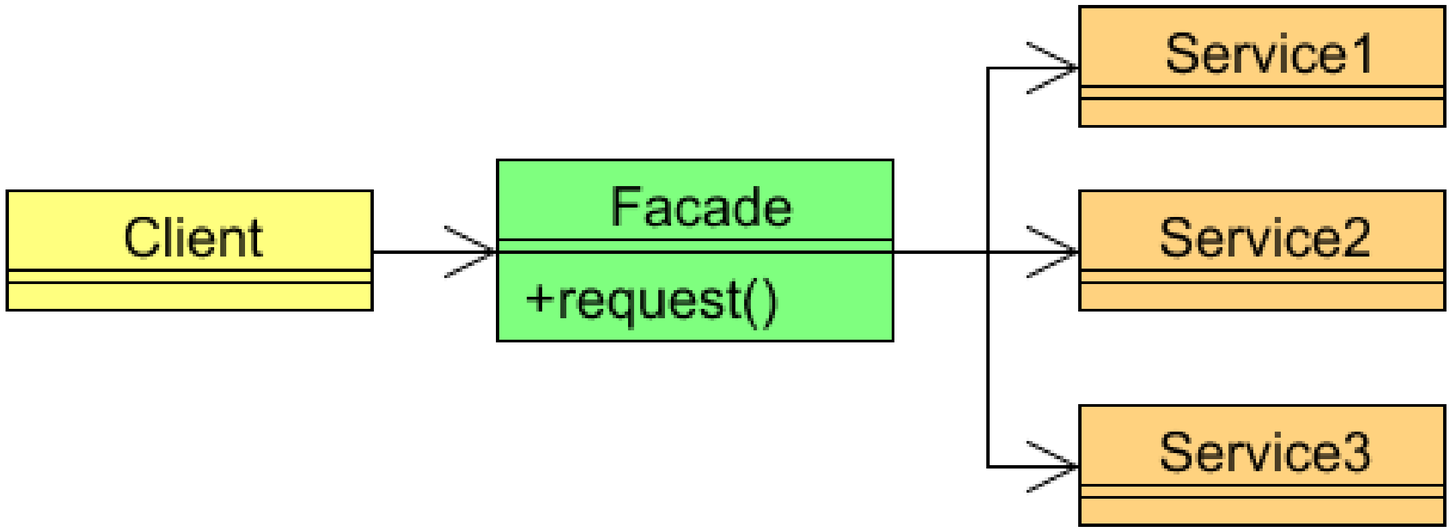


- When a client directly links to many classes:
  - It is vulnerable to changes in any of those classes
  - It must know each one's interface
  - It must devise an order for calling operations and delegating parameters to them
- In a distributed environment, many fine-grained client calls can also increase network overhead.

# Applying the Facade Pattern: Solution

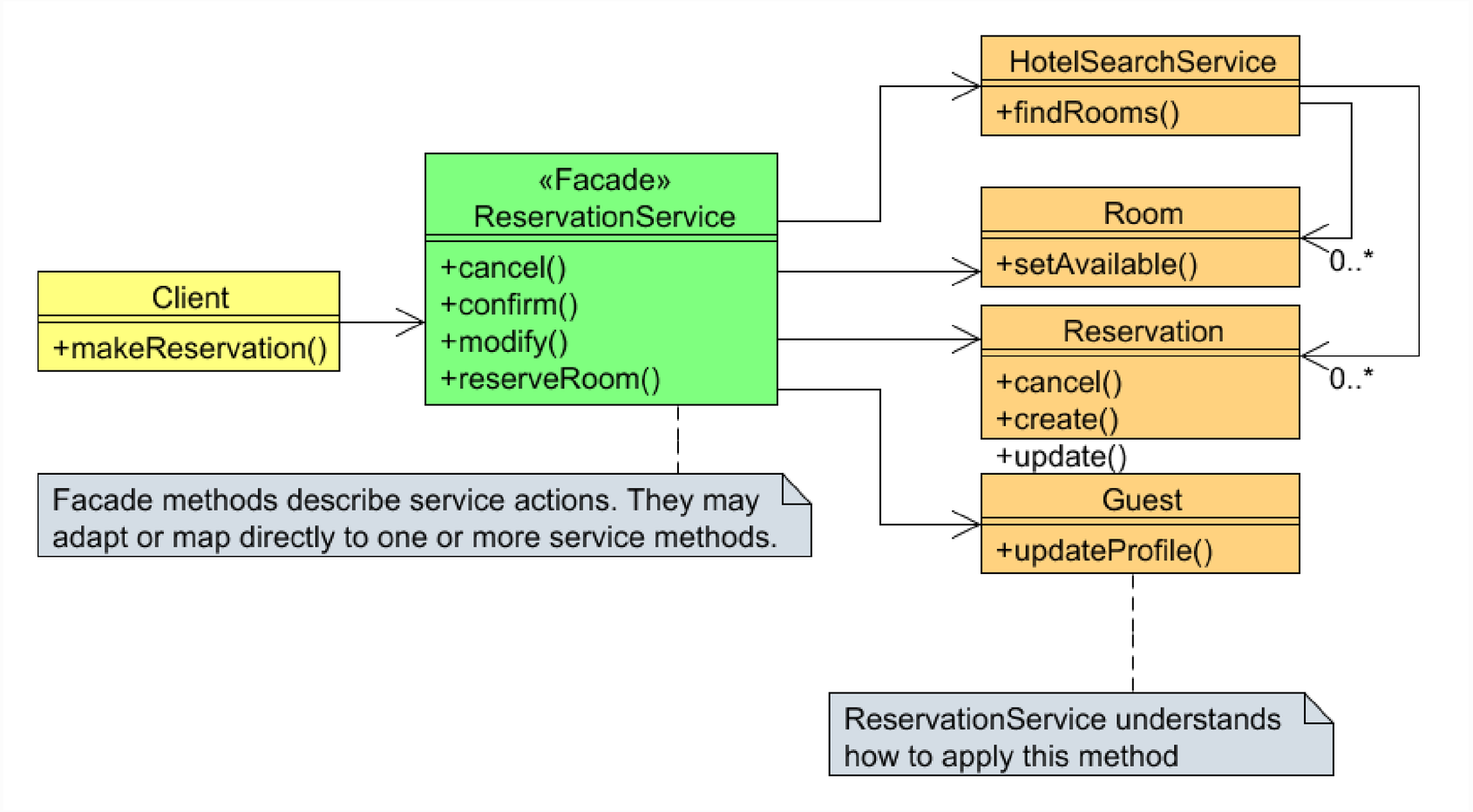
- Provide a `Facade` class that has a simple, high-level interface to a subsystem.
  - Clients use a `Facade` object instead of communicating with each subsystem object individually.
  - The `Facade` contains the logic required for sequencing, managing state, and communicating with each subsystem object.
  - The `Client` may also be allowed to communicate directly with other subsystem classes for additional functionality.

# Applying the Facade Pattern: Structure



# Applying the Facade Pattern: Example Solution





# Applying the Facade Pattern: Consequences

Advantages:

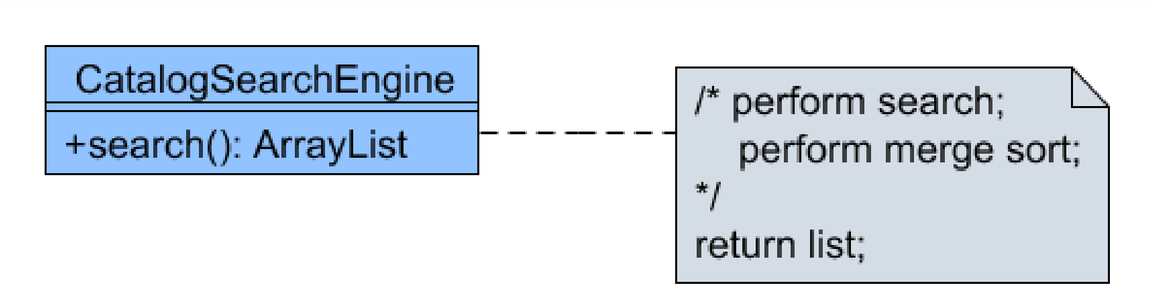
- The client is decoupled from the subsystem components.
- The client does not have to know the underlying process.
- There are fewer network trips in a distributed environment.
- Clients can still communicate directly with the subsystems.

Disadvantage:

- A layer of indirection obscures the underlying process.

# Applying the Strategy Pattern: Example Problem

- A library system's `search` method wants to use multiple algorithms for sort results.
- Replacing or adding algorithms requires code modification.



# Applying the Strategy Pattern: Problem Forces

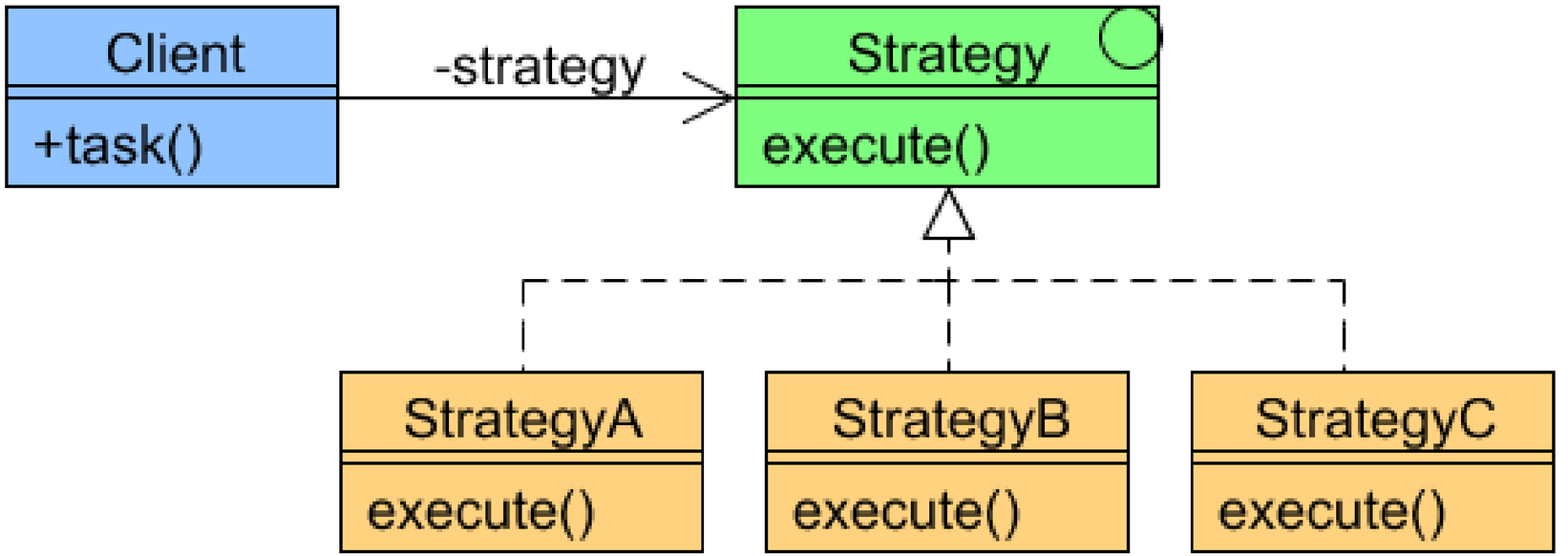


- Classes often need to choose from multiple algorithms.
- The details of these algorithms should be hidden from client classes.
- Separating choices by conditional logic is hard to maintain.

# Applying the Strategy Pattern: Solution

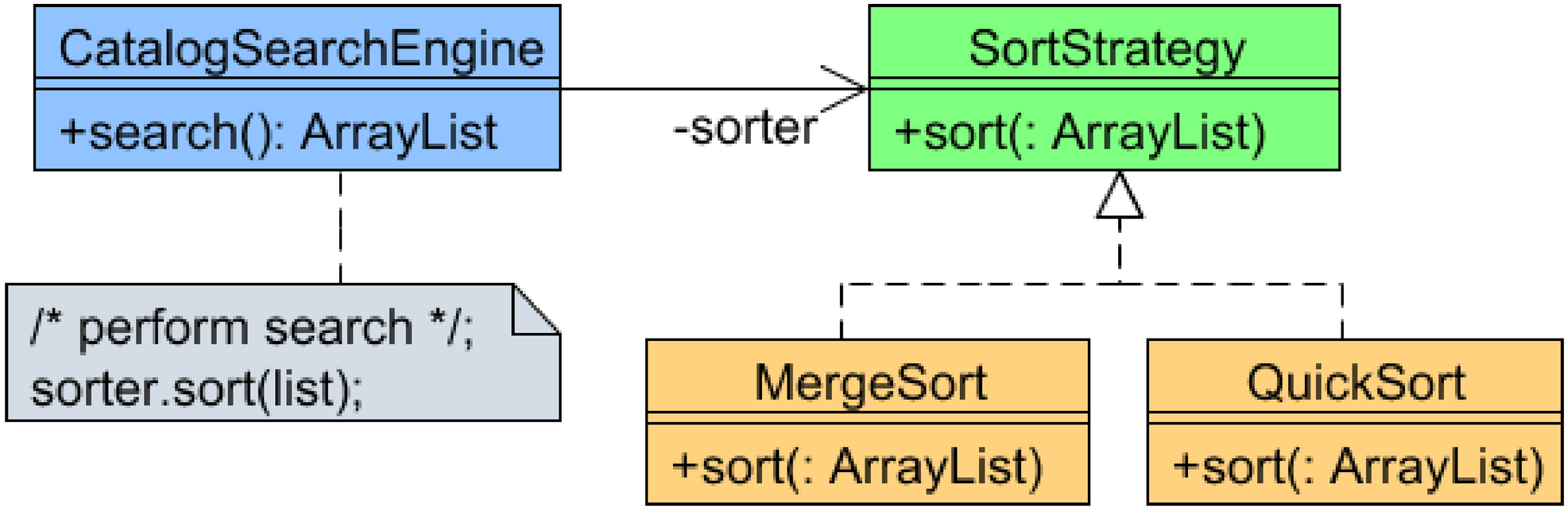
- Each algorithm is implemented in a separate class that implements the `Strategy` interface.
- The `Client` object:
  - References a `Strategy` object
  - Delegates the request to the reference
  - May specify the `Strategy` subtype to use (optional)

# Applying the Strategy Pattern: Structure



# Applying the Strategy Pattern: Example Solution





# Applying the Strategy Pattern: Client Code

```
1  public class CatalogSearchEngine {
2
3     private SortStrategy sorter;
4
5     public CatalogSearchEngine(SortStrategy
ss) {
6        sorter = ss;
7     }
8     public ArrayList search() {
9        ArrayList list = //perform search
10       sorter.sort(list);
11       return list;
12    }
13 }
```

# Applying the Strategy Pattern: Implementing the Interface

```
1  public interface SortStrategy {
2   public void sort(ArrayList al);
3  }
```

```
1  public class QuickSort implements
SortStrategy {
2    public void sort(ArrayList al) {
3  //implement Quick sort code
4    }
5  }
```

# Applying the Strategy Pattern: Consequences

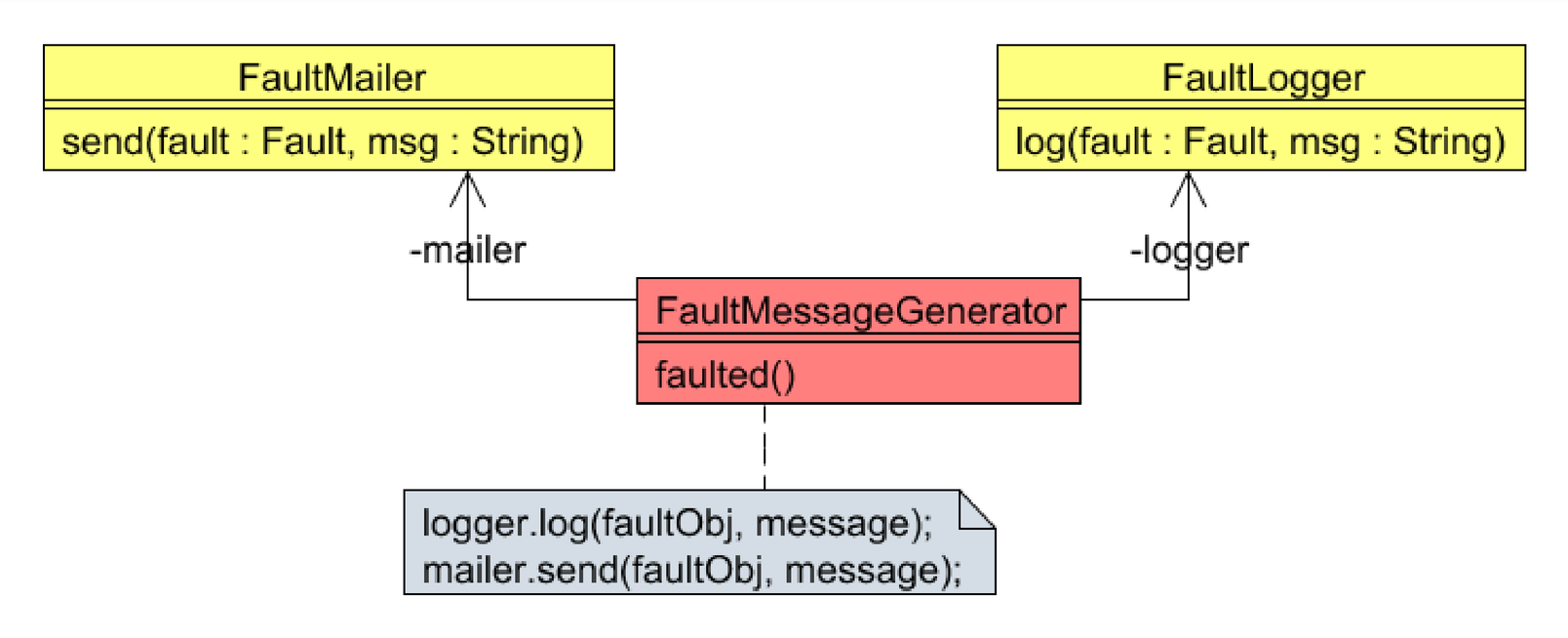


- Advantages:
  - Adding new strategies does affect the existing code.
  - Branch logic that can grow over time can be avoided.
  - Each algorithm is expressed separately.
  - Change to one algorithm is isolated from other algorithms.
- Disadvantages:
  - Clients must know which strategies are available.
  - Naive clients may expect default logic.
    - A `Strategy` interface can instead be a parent class with a default implementation.

# Applying the Observer Pattern: Example Problem



- The `faulted()` method delegates a message to services that are coupled to the `FaultMessageGenerator` class.
- Other fault handlers may be needed.



# Applying the Observer Pattern: Problem Forces

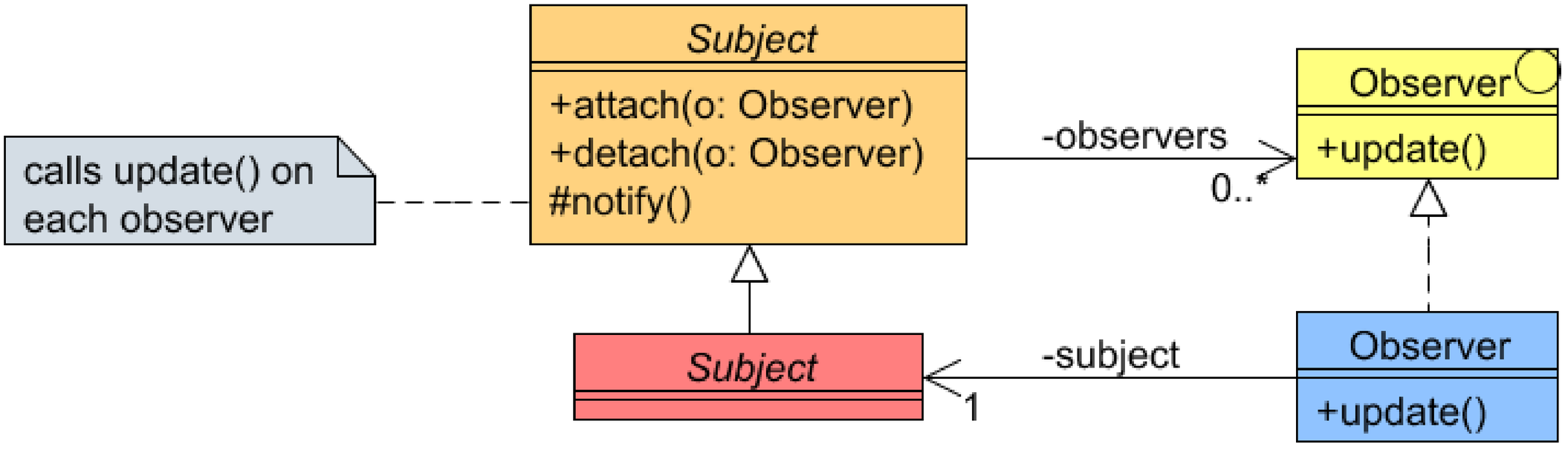


- Several classes want to monitor changes in another class:
  - Synchronous polling for asynchronous state changes may waste resources.
  - Multiple observers polling an observable object may also require synchronization controls.
  - Observers must know how to get to an observable object.
  - You want to add observers as needed.

# Applying the Observer Pattern: Solution

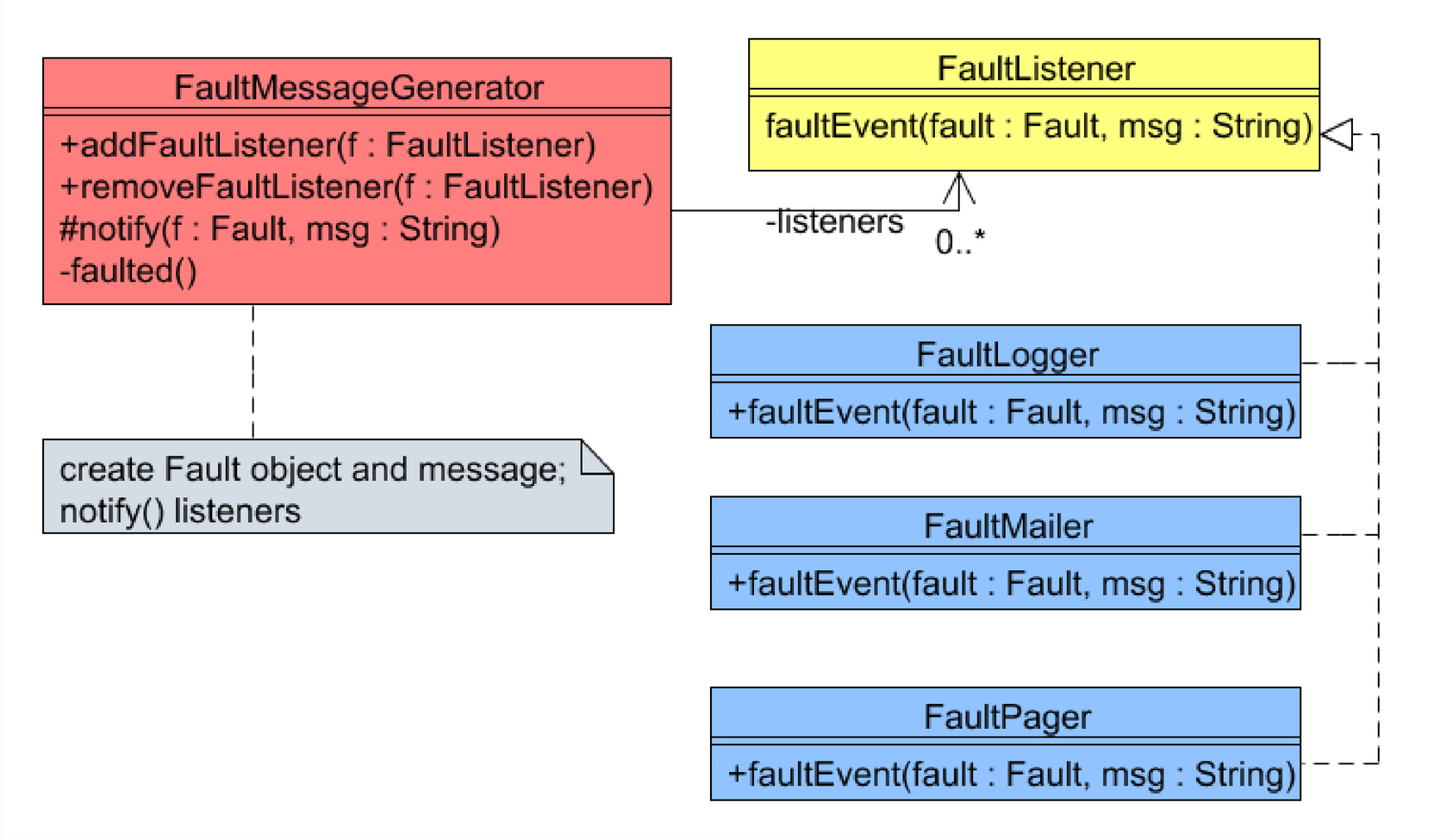
- Use an Observer interface that lets an observable `Subject` call on `Observer` implementations.
- `Subject` keeps a collection of objects that implement `Observer`.
- The `Observer` objects register with `Subject`.
- The `Subject` notifies the registered `Observer` objects when it changes state.

# Applying the Observer Pattern: Structure



# Applying the Observer Pattern: Example Solution





# Applying the Observer Pattern: Observable Code

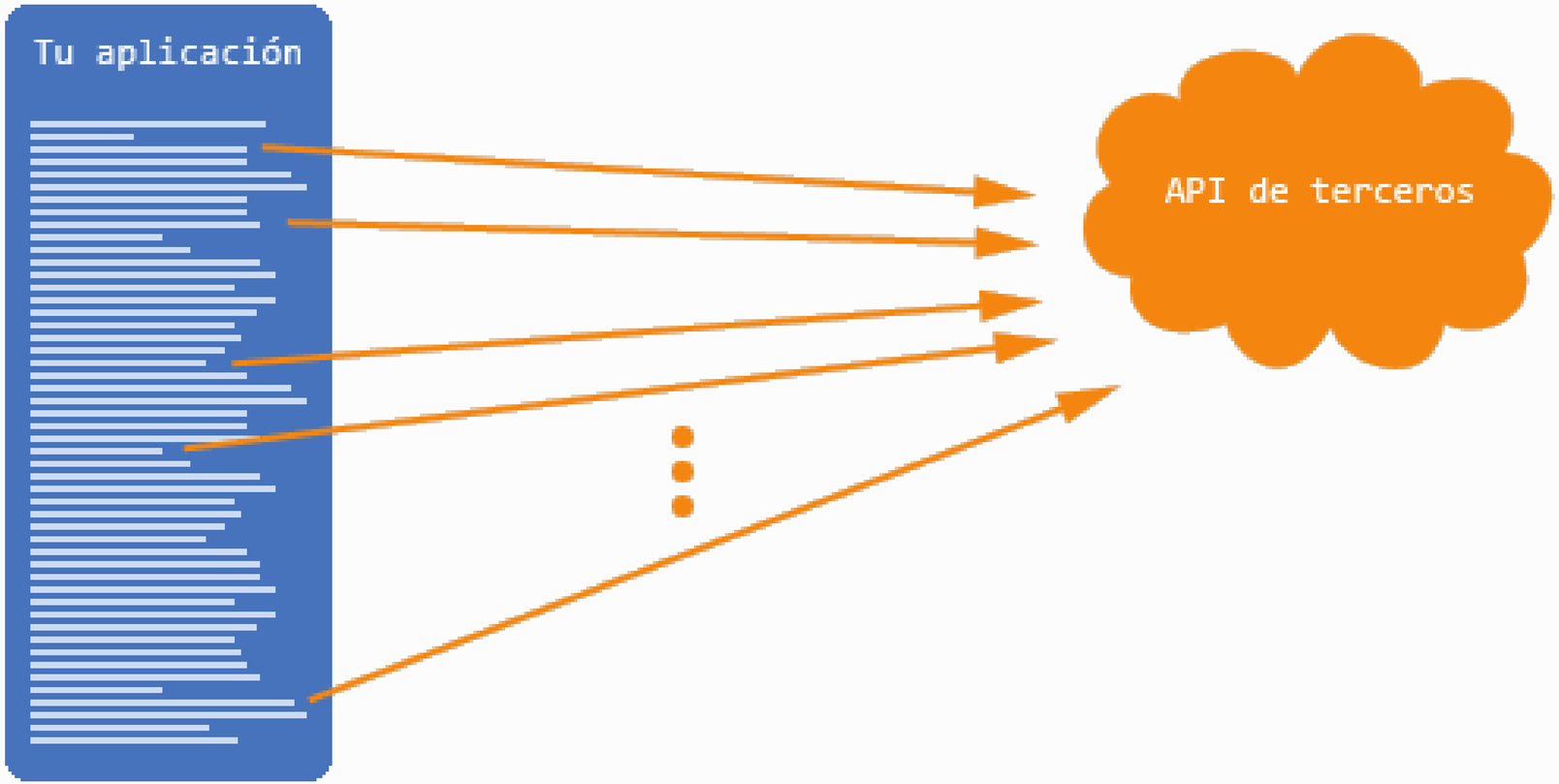
```
1 import java.util.*;
2 public class FaultMessageGenerator {
3  private ArrayList<FaultListener> listeners =
    new ArrayList<FaultListener>();
4
5  public void addFaultListener(FaultListener listener) {
6   listeners.add(listener);
7  }
8  protected void notify() {
9   for (FaultListener fl: listeners) {
10    fl.faultEvent(faultObj, faultMsg);
11  }
12  public void removeFaultListener(FaultListener listener) {
13   listeners.remove(listener);
14  }
15 }
```

# Applying the Observer Pattern: Observer Code

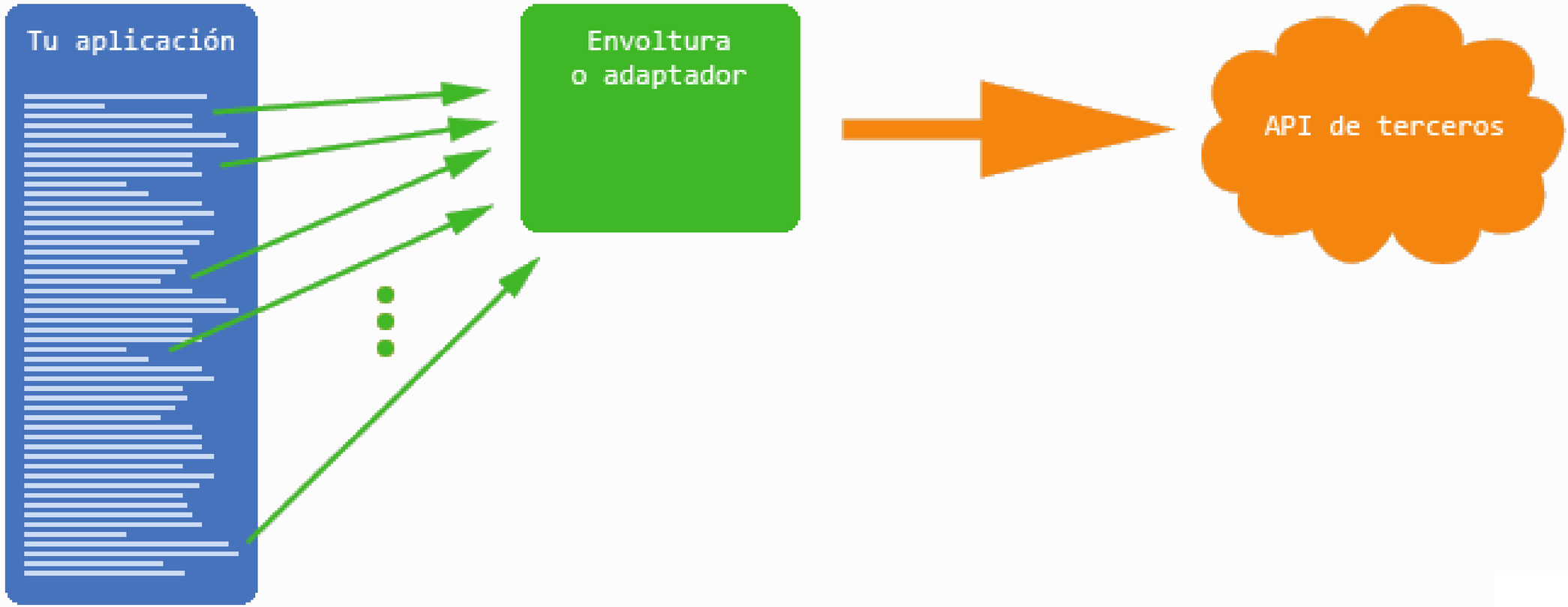
```
1 import java.util.logging.*;
2
3 public class FaultLogger implements FaultListener{
4   private static Logger logger =
Logger.getLogger("");
5
6   public FaultLogger(FaultMessageGenerator fmg) {
7     fmg.addFaultListener(this);
8   }
9   public void faultEvent(Fault fault, String msg) {
10   logger.log(Level.WARNING,
11     "A " + fault.getType() + " occurred: " + msg);
12  }
13 }
```

# 2. Design Patterns applied to Microservices

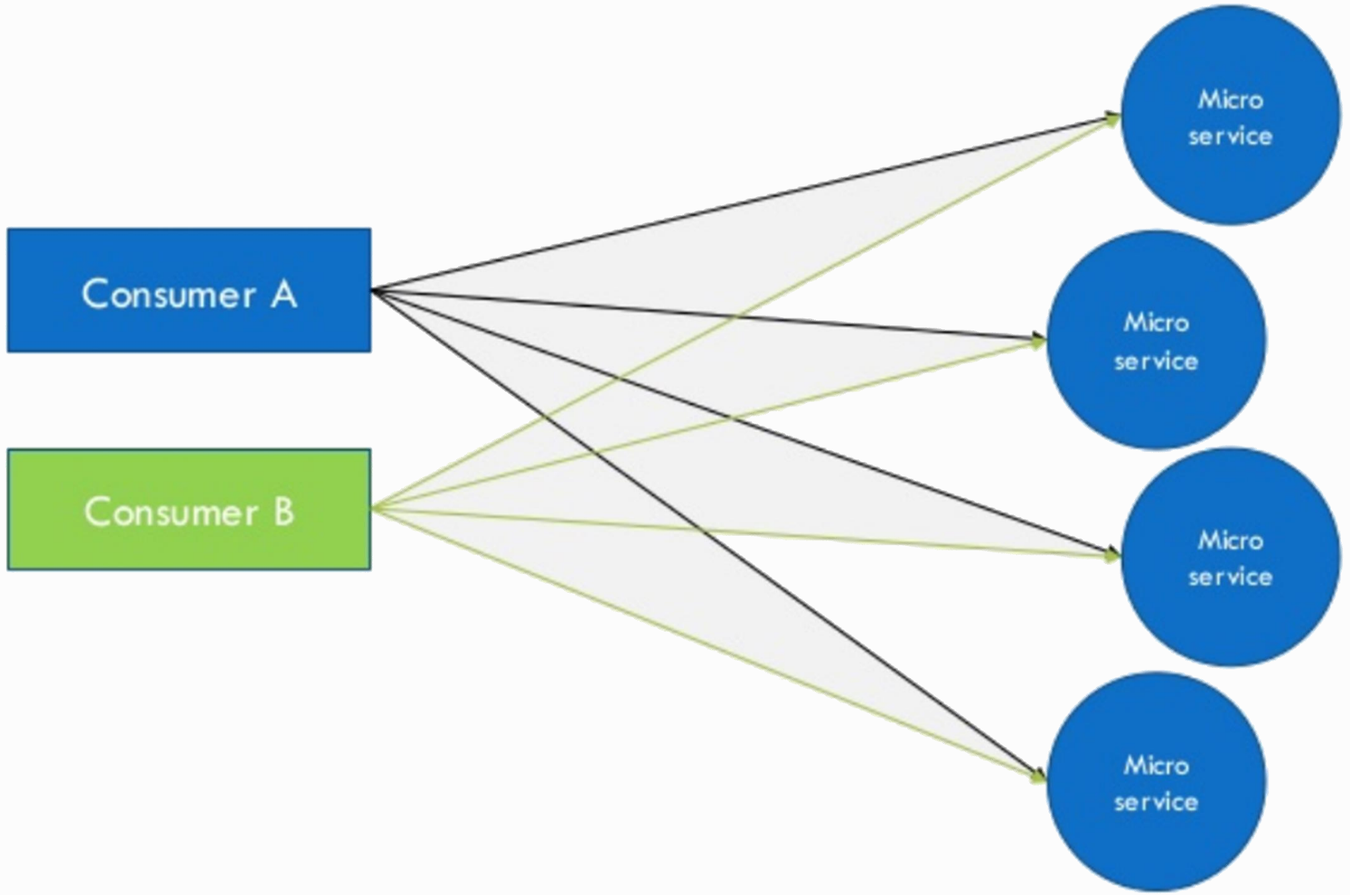
# Adapter



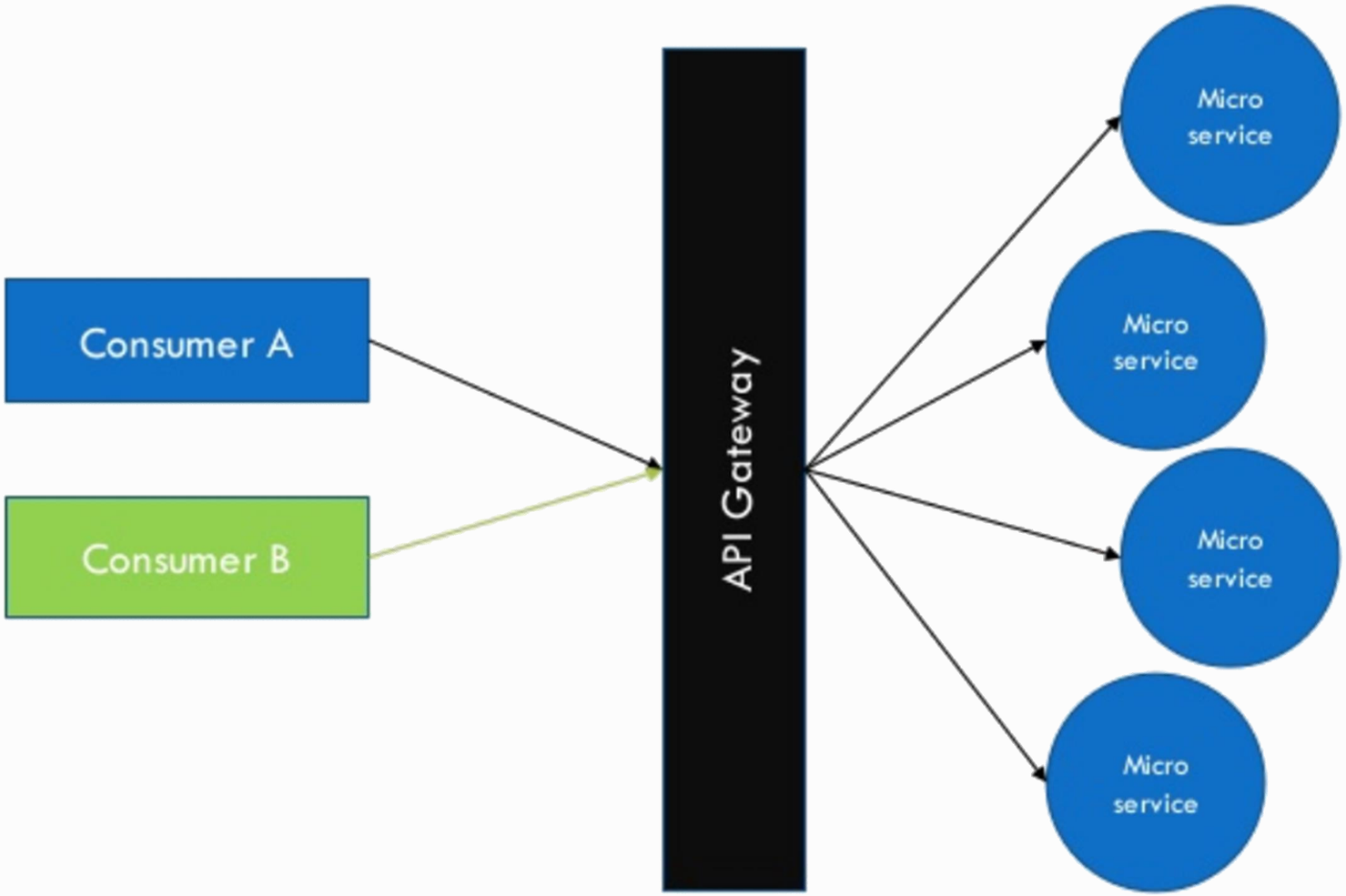
# Adapter



# Direct Communication



# API Gateway



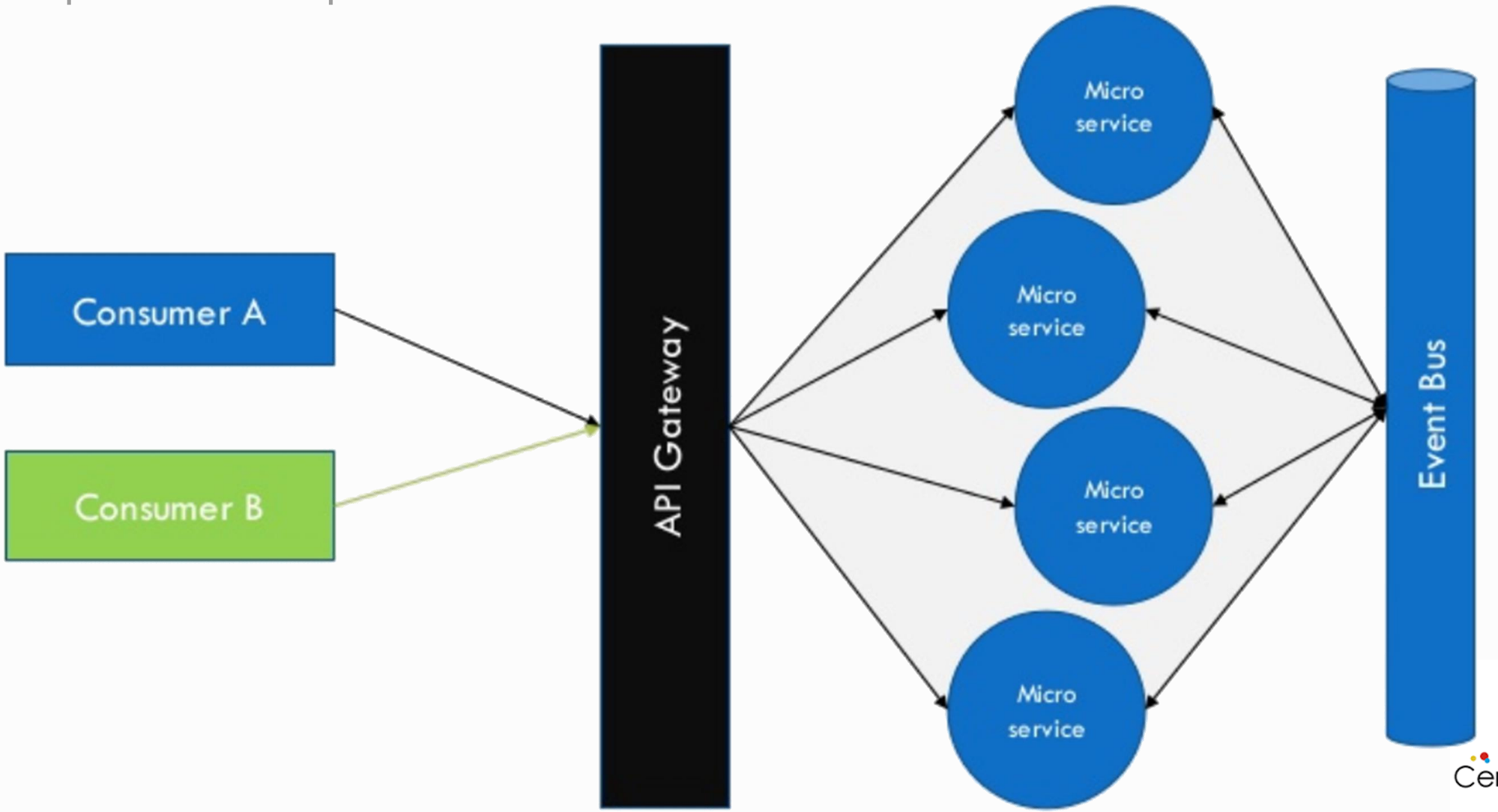

Who is consuming our services?

Who was consuming what?

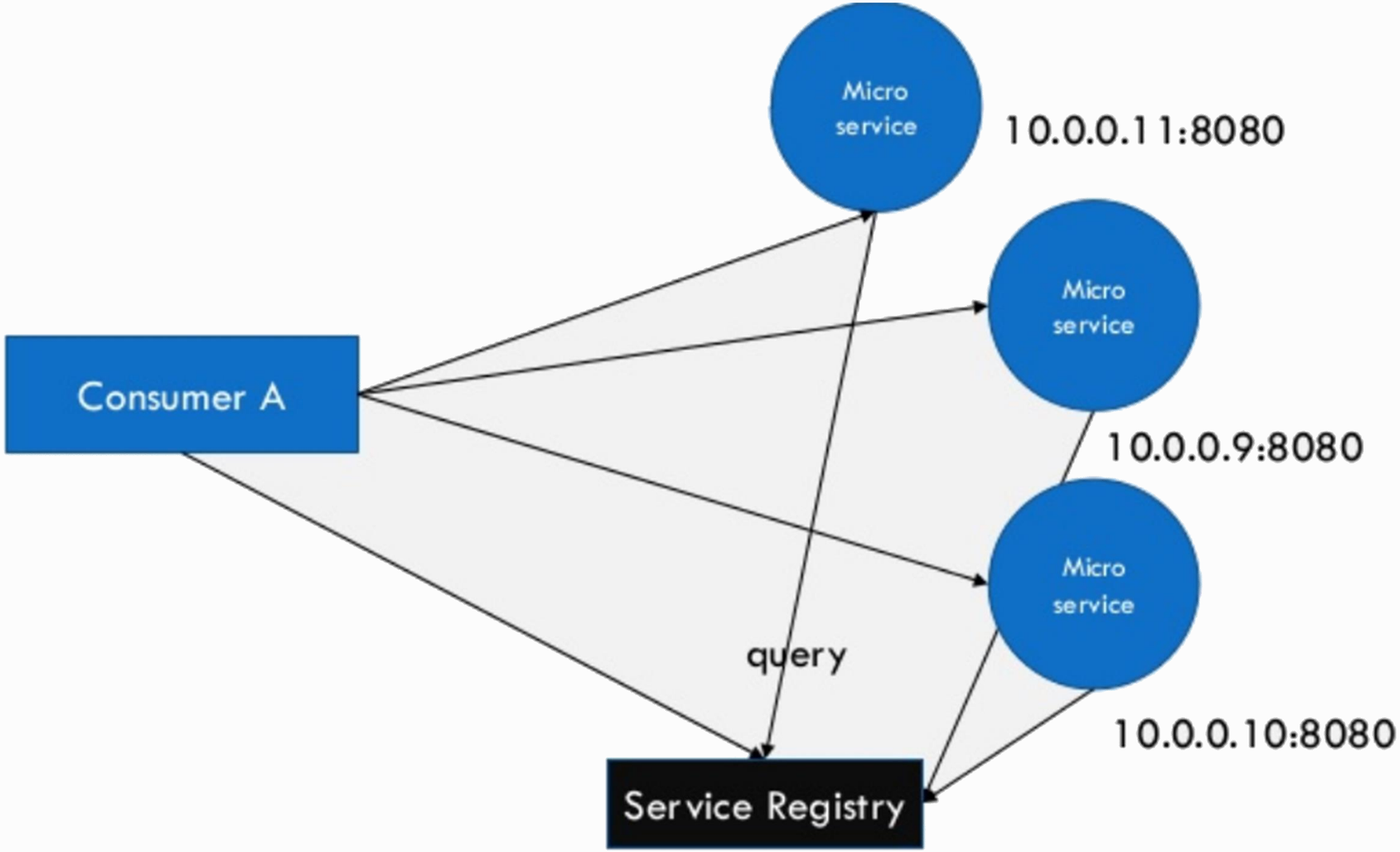
What rate?

What time?

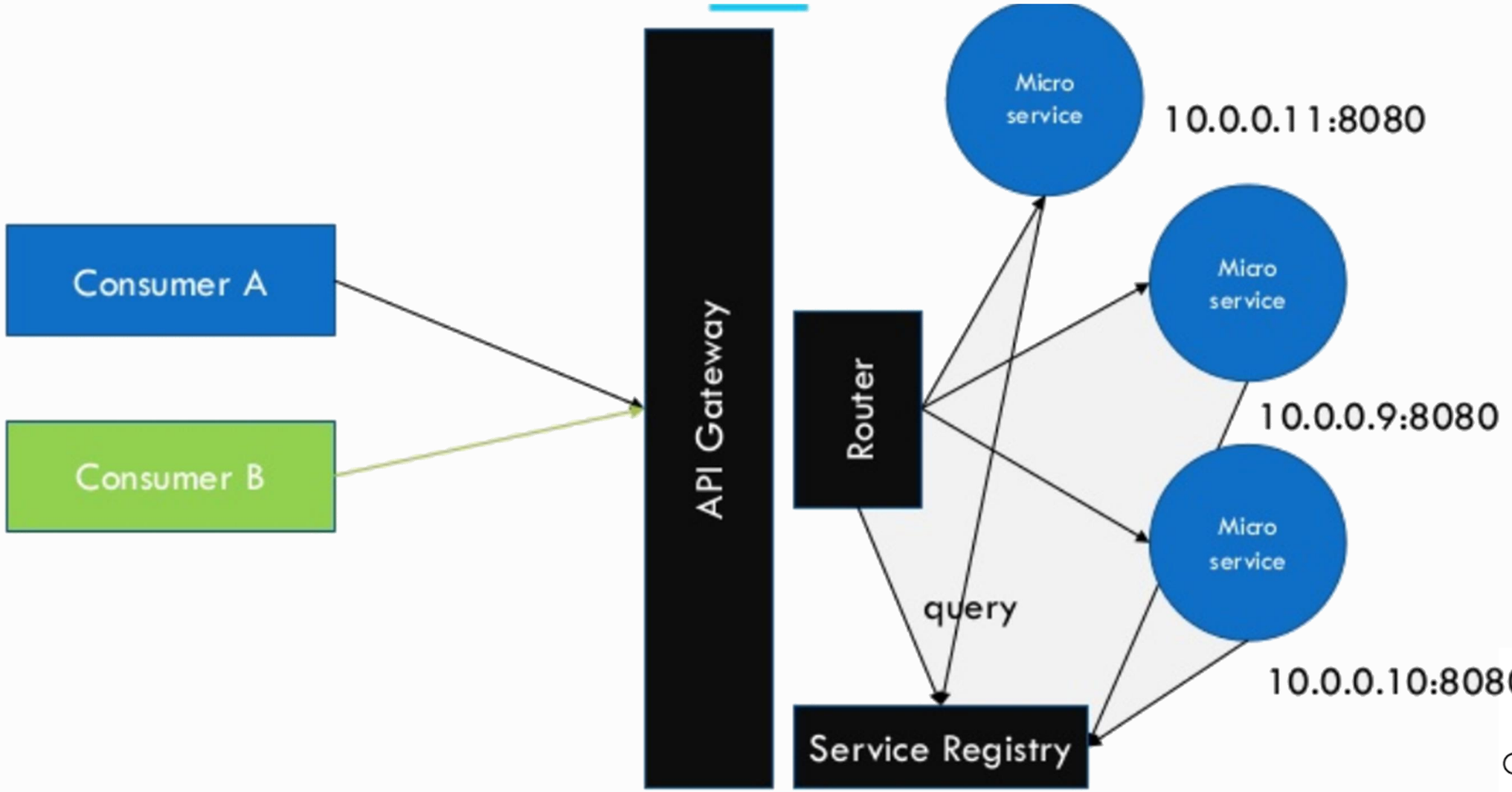
# Multiple Endpoint Location



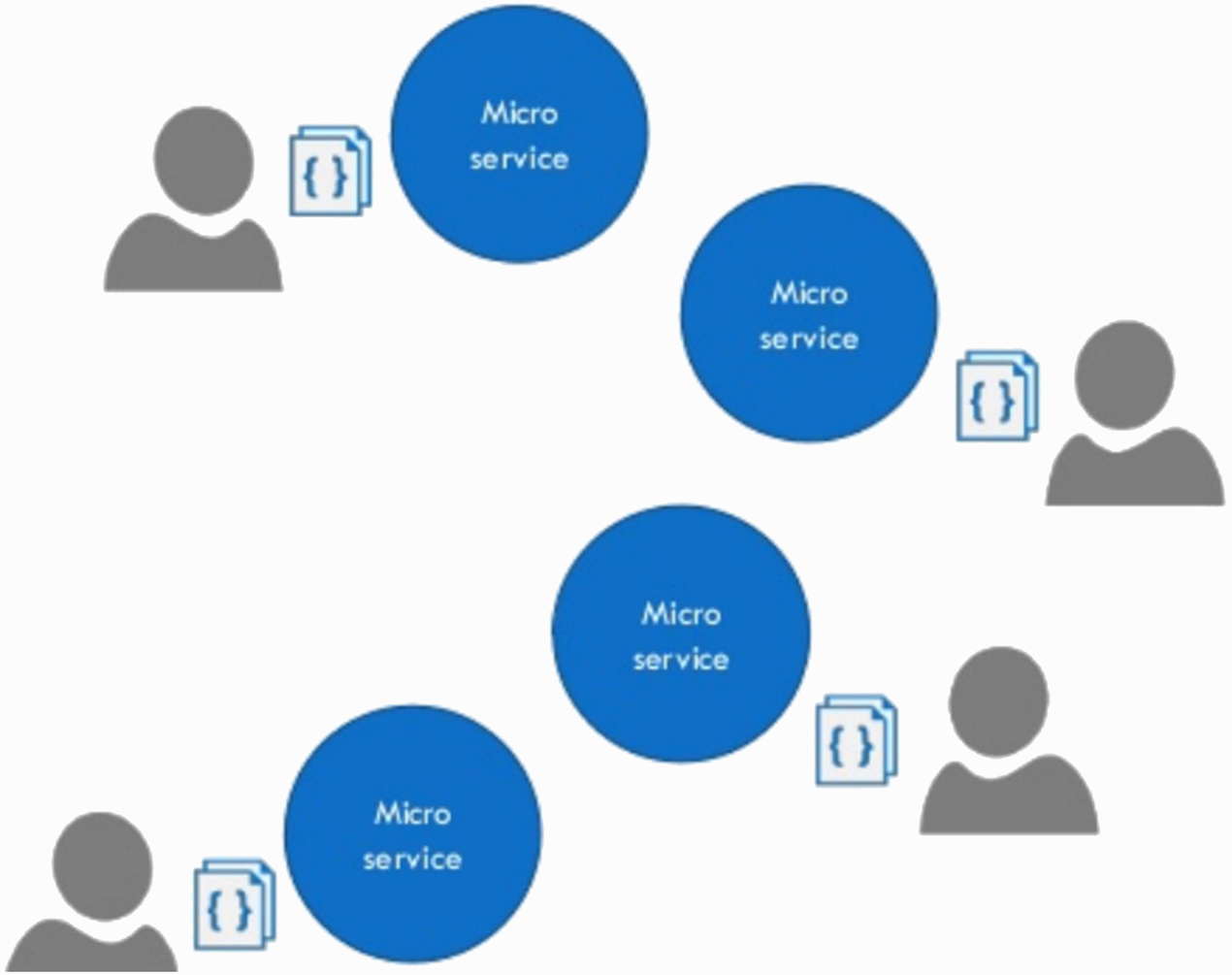
# Service Discovery



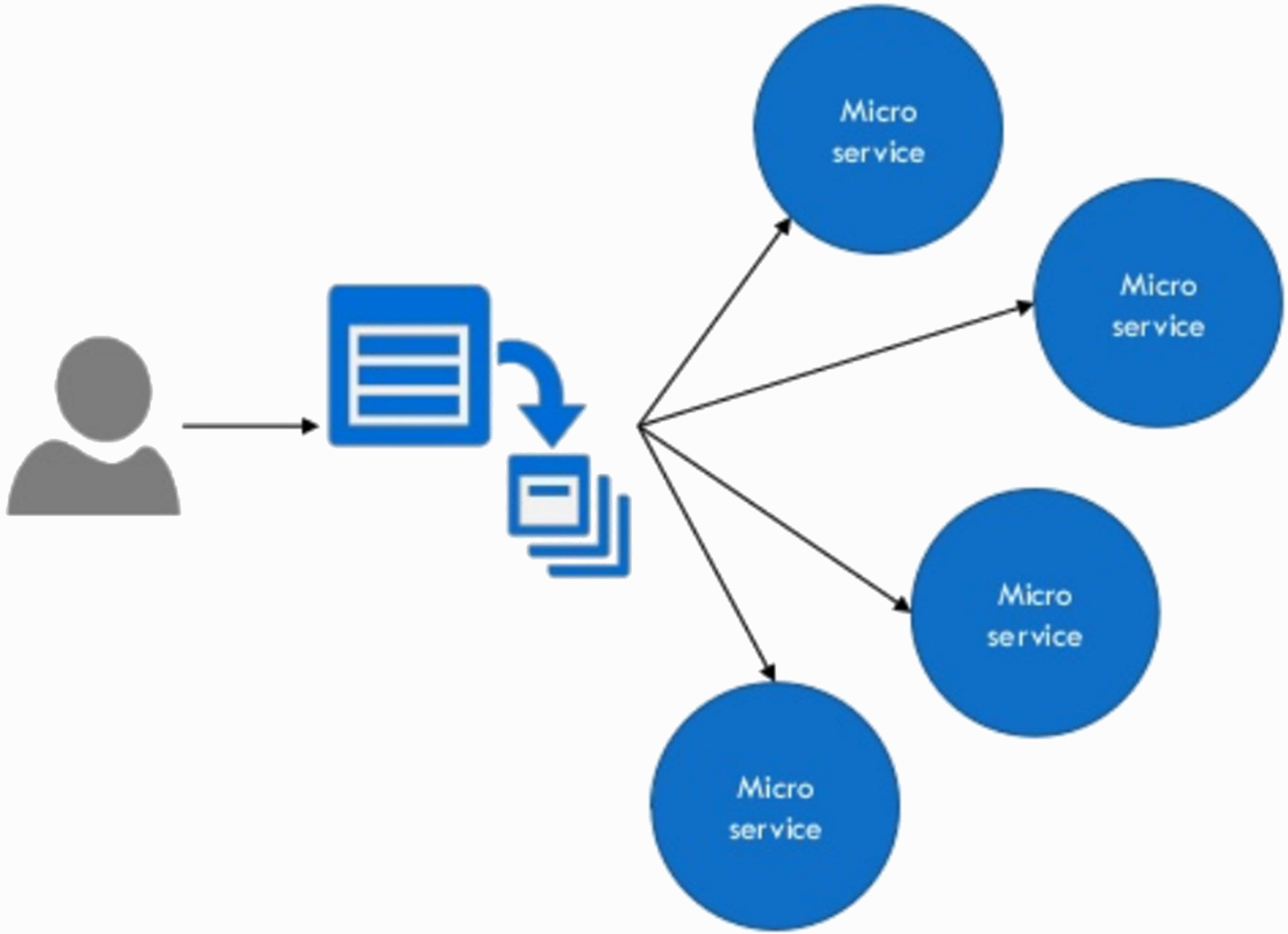
# Service Discovery



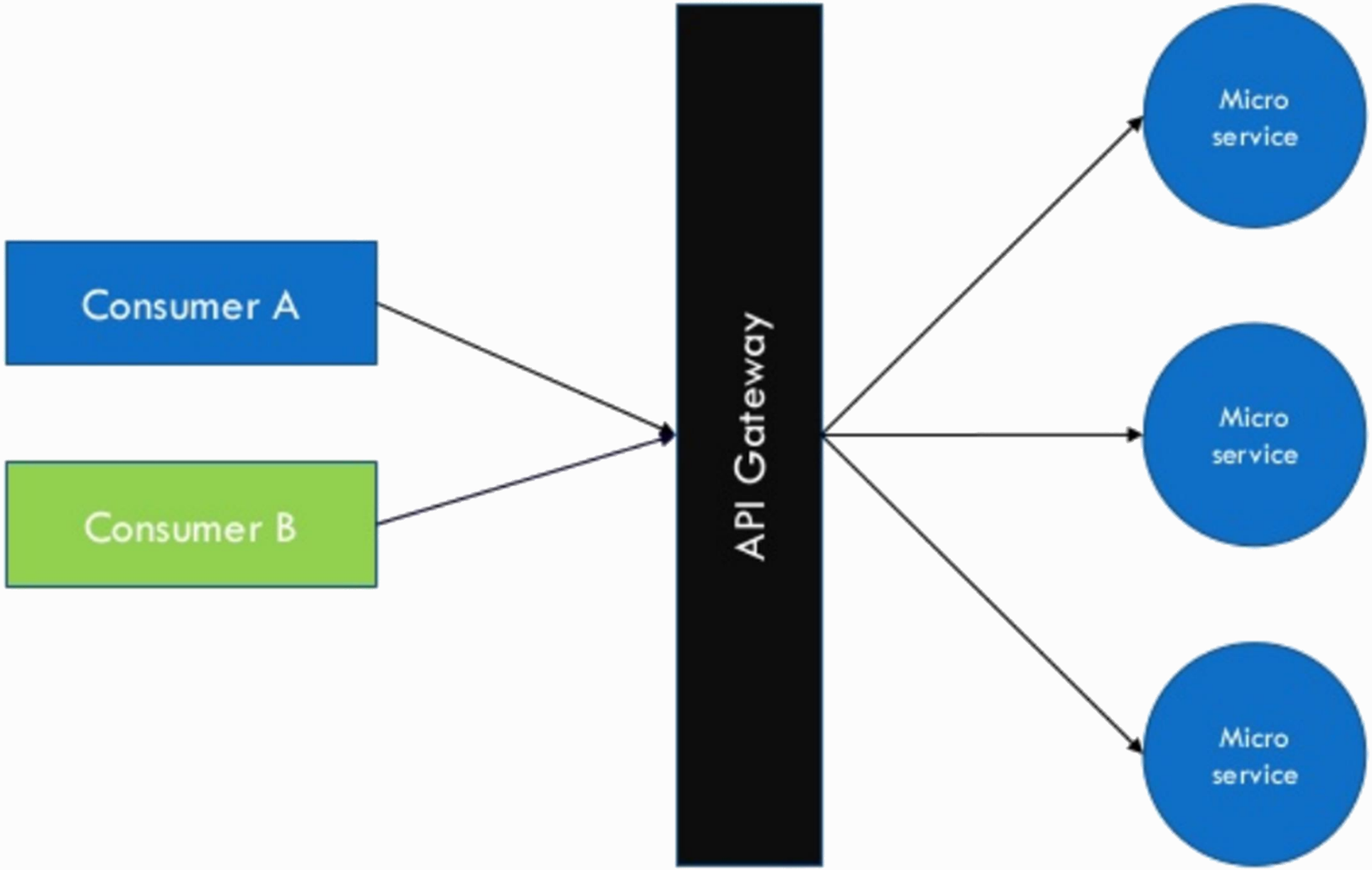
# Multiple Configurations



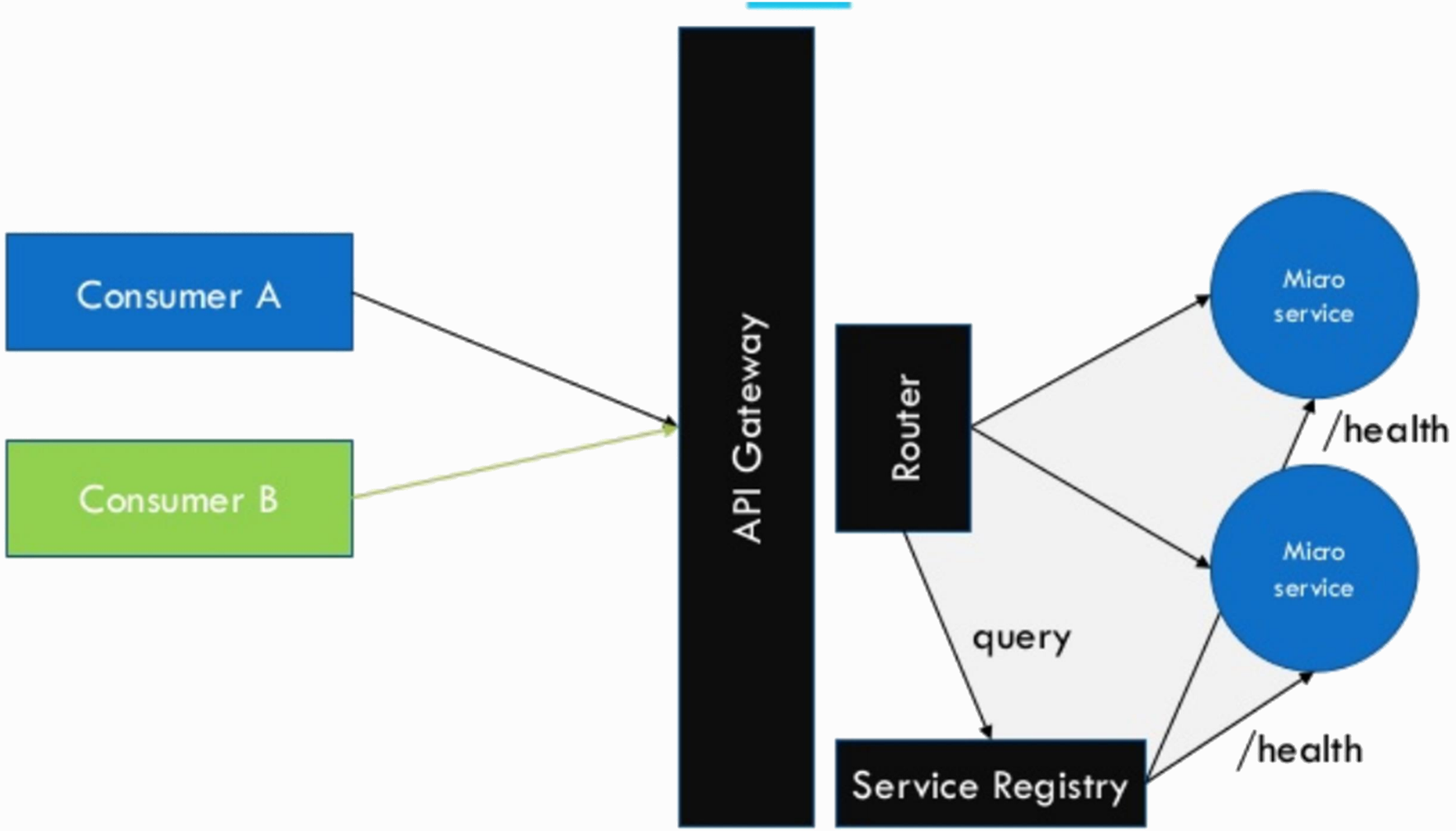
# Centralized Configuration



# Retry Pattern



# Health Check API



# GRACIAS

Certificatic®
Beyond Certification