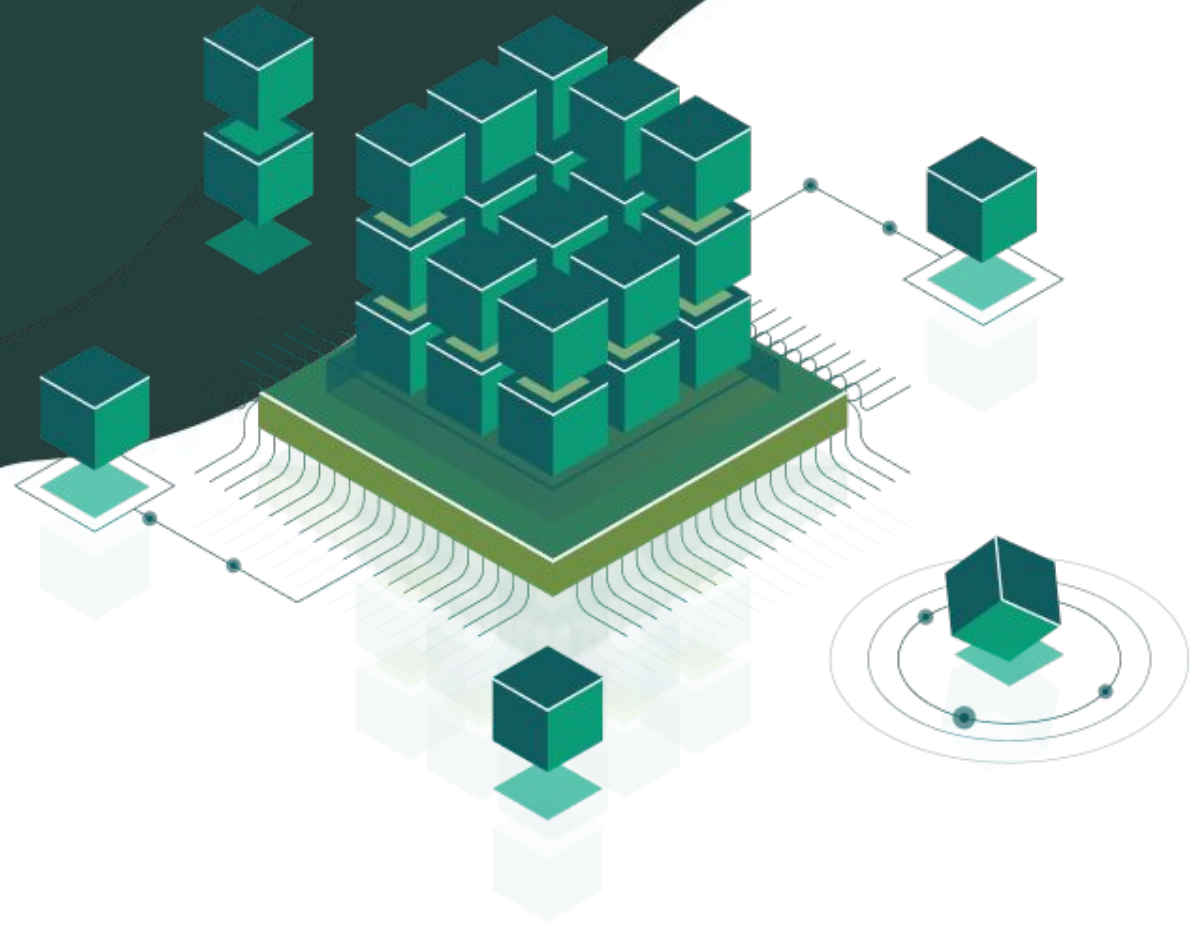


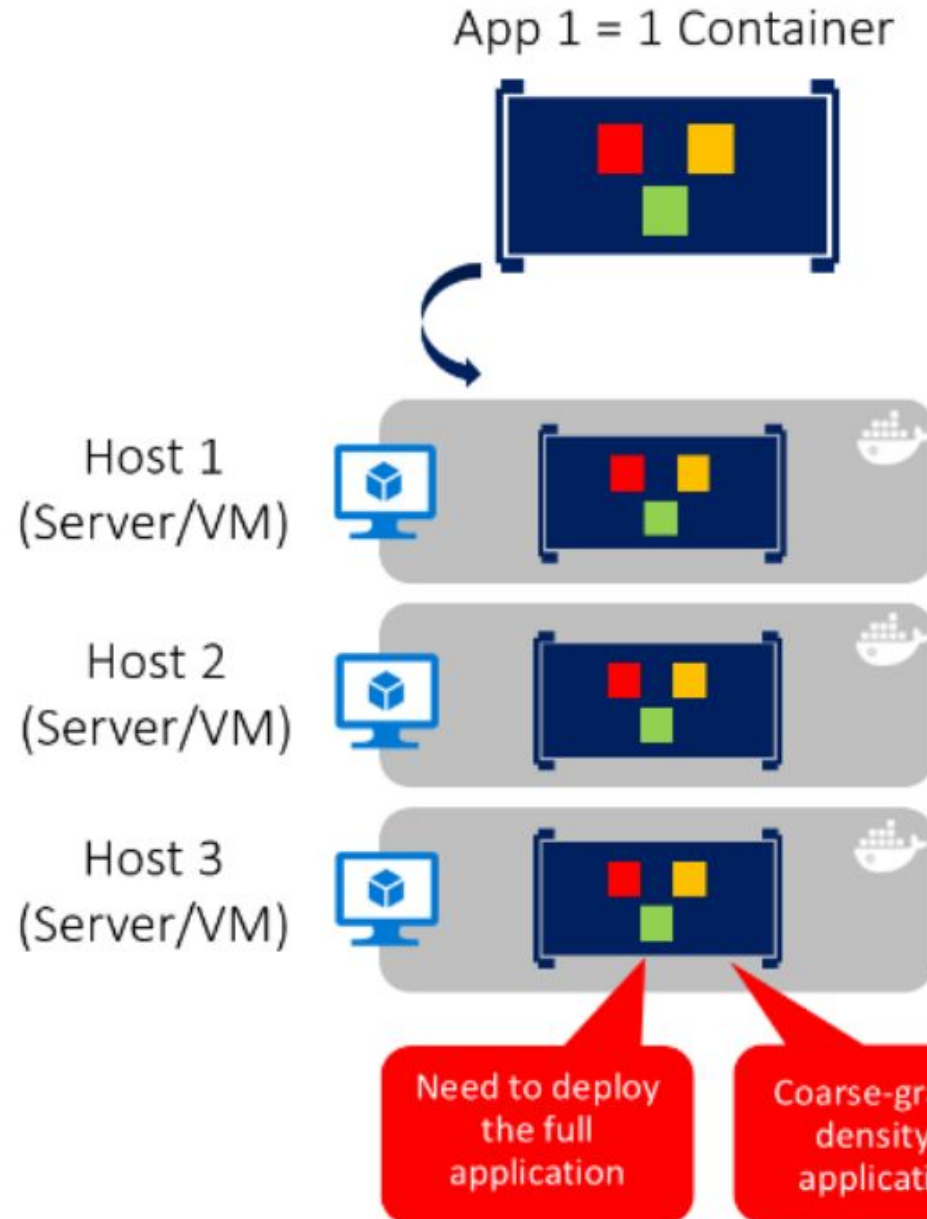
Microservices. Health checks. Proxy server. Logs.
Api-gateway. Api-orchestration.



Speaker:

Roman Tikhiiy

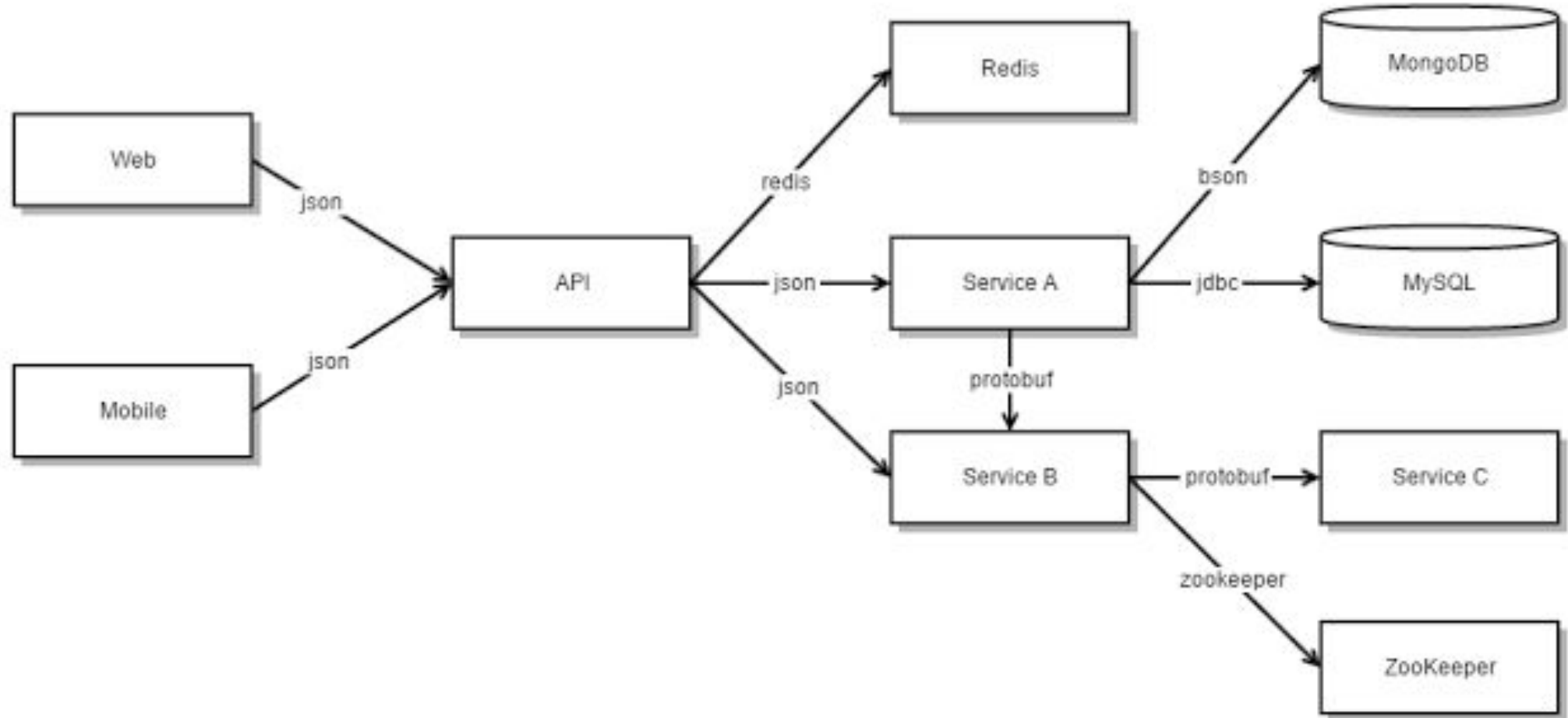
Monolithic Containerized application



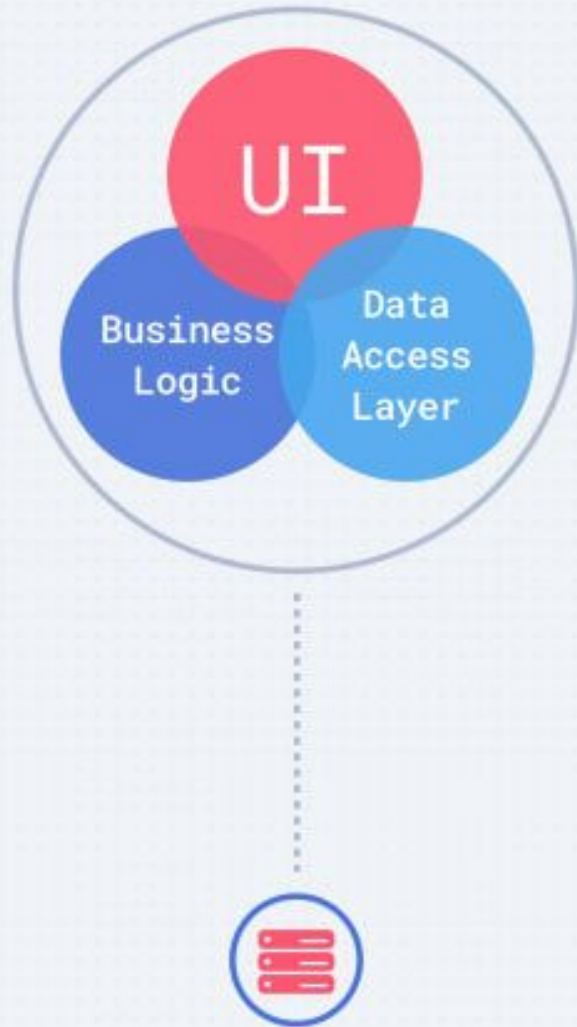
A monolithic application has most of its functionality within a single process/container that is componentized with internal layers or libraries.

Scales out by cloning the app/container on multiple servers/VMs

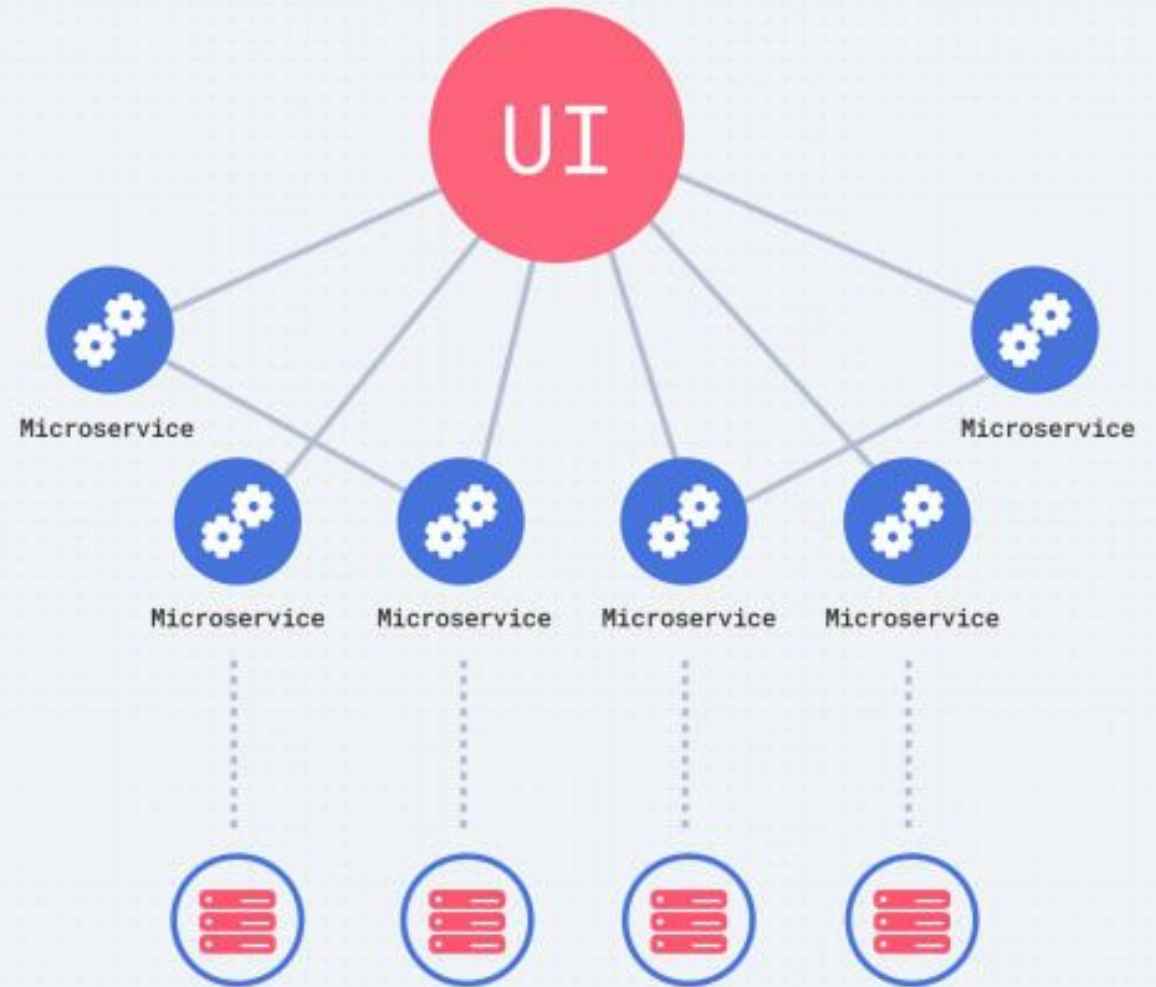
Microservices



Monolithic Architecture



Microservices Architecture



Deployment

Monolith

- one point of failure during deployment and, if everything goes wrong, you could break your entire project.

Microservices

- more complexity
- if something goes wrong, you will only break one small microservice, which is less problematic than the entire project
- easier to rollback one small microservice

Microservices reliability

Breaking one microservice affects only one part and causes issues for the clients that use it, but no one else. If, for example, you're building a banking app and the microservice responsible for money withdrawal is down, this is definitely less serious than the whole app being forced to stop.

Scalability

Monolith

- inefficient way of using resources
- impossible to scale it horizontally, leaving only vertical scaling possible for your monolith app

Microservices

- easier
- allow you to scale only that parts that require more resources

Use monolith architecture if you

- have a small team.
- build the MVP version of a new product.
- did not get millions in investments to hire DevOps or spend extra time on complex architecture.
- don't see performance bottlenecks for some key functionality.

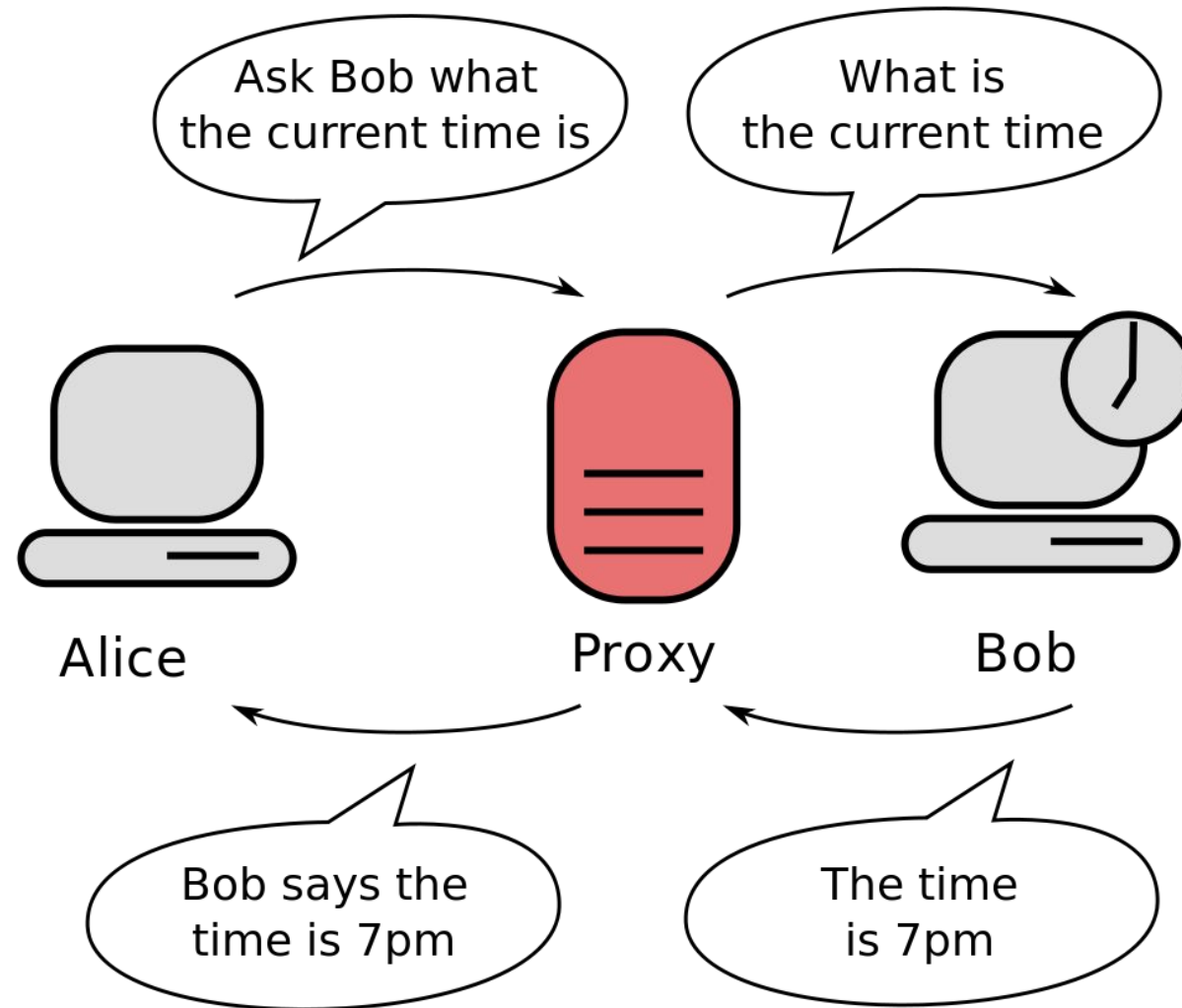
Use microservices architecture if you

- don't have a tight deadline; microservices require you to research and architecture planning to ensure it works.
- have a team with knowledge of different languages.
- worry a lot about the scalability and reliability of your product.
- potentially have a few development departments(maybe even in different countries/time zones).
- have an existing monolith app and see problems with parts of your application that could be split across multiple microservices.

Microservices Best Practices

- Independent Teams – Create an environment where teams can get more done without having to coordinate with other teams.
- Focus on automation – Automate everything.
- Built for resilience – Ensures failure doesn't impact too much.
- Simplify the maintenance – Have proper guidelines and documentation for each service.
- Provide flexibility – Give teams the freedom to do what's right for their services.

Proxy server

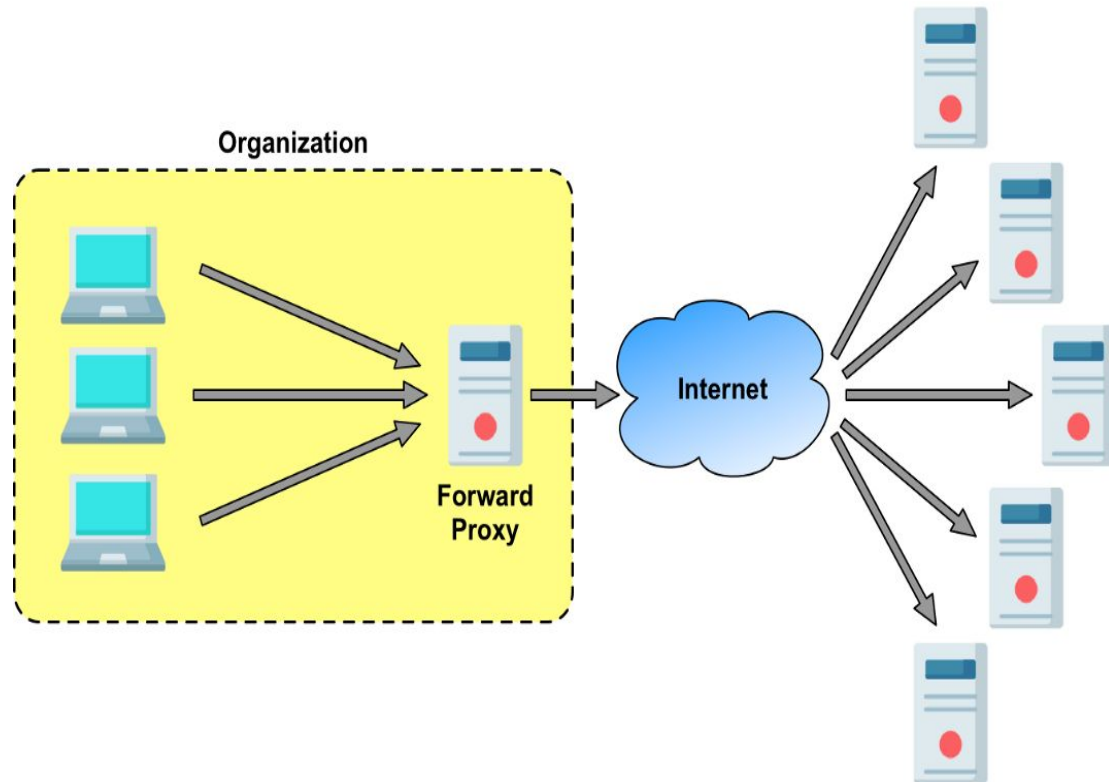


Use of proxy servers

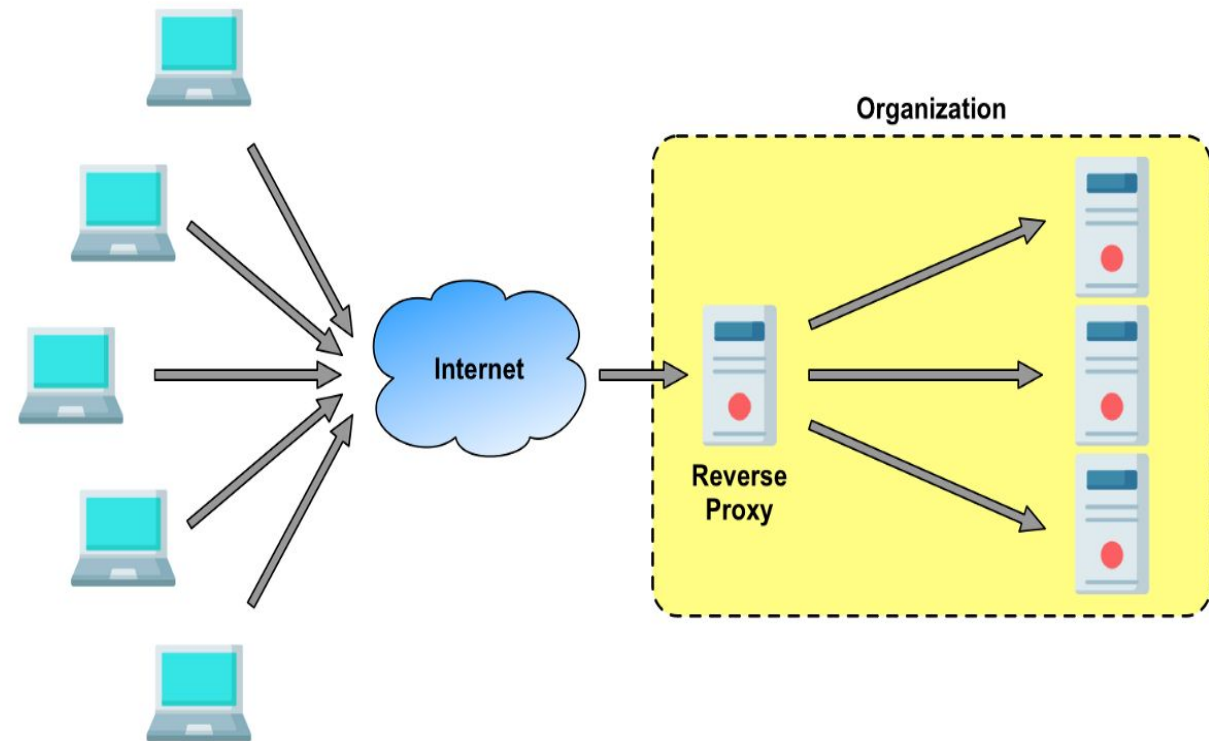
- increase performance using caching mechanism.
- zipping data.
- security.
- logging.
- access control.

Use of proxy servers

Forward Proxy



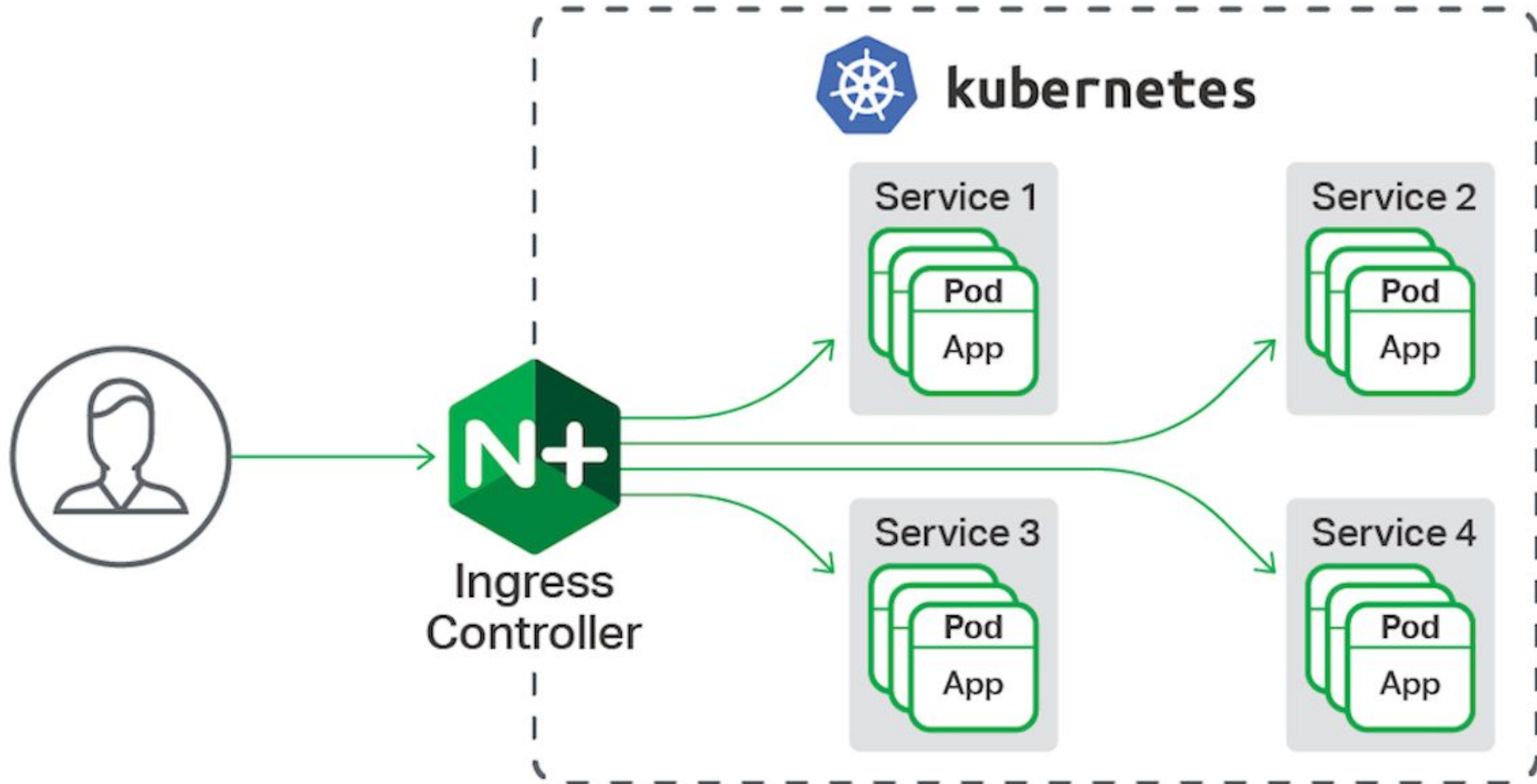
Reverse Proxy



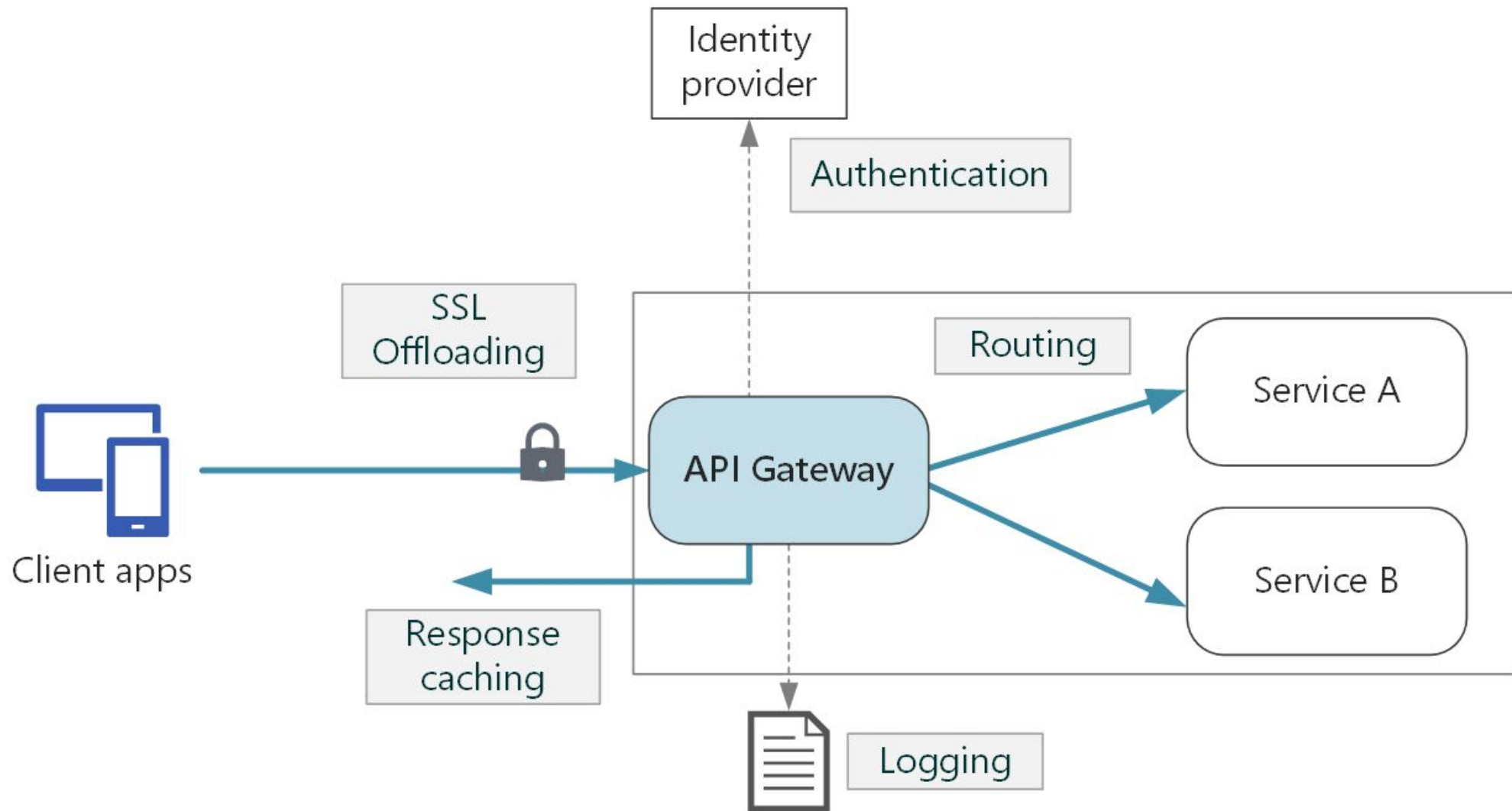


- easy to configure
- fast and reliable
- load balancing support
- high performance caching

Nginx

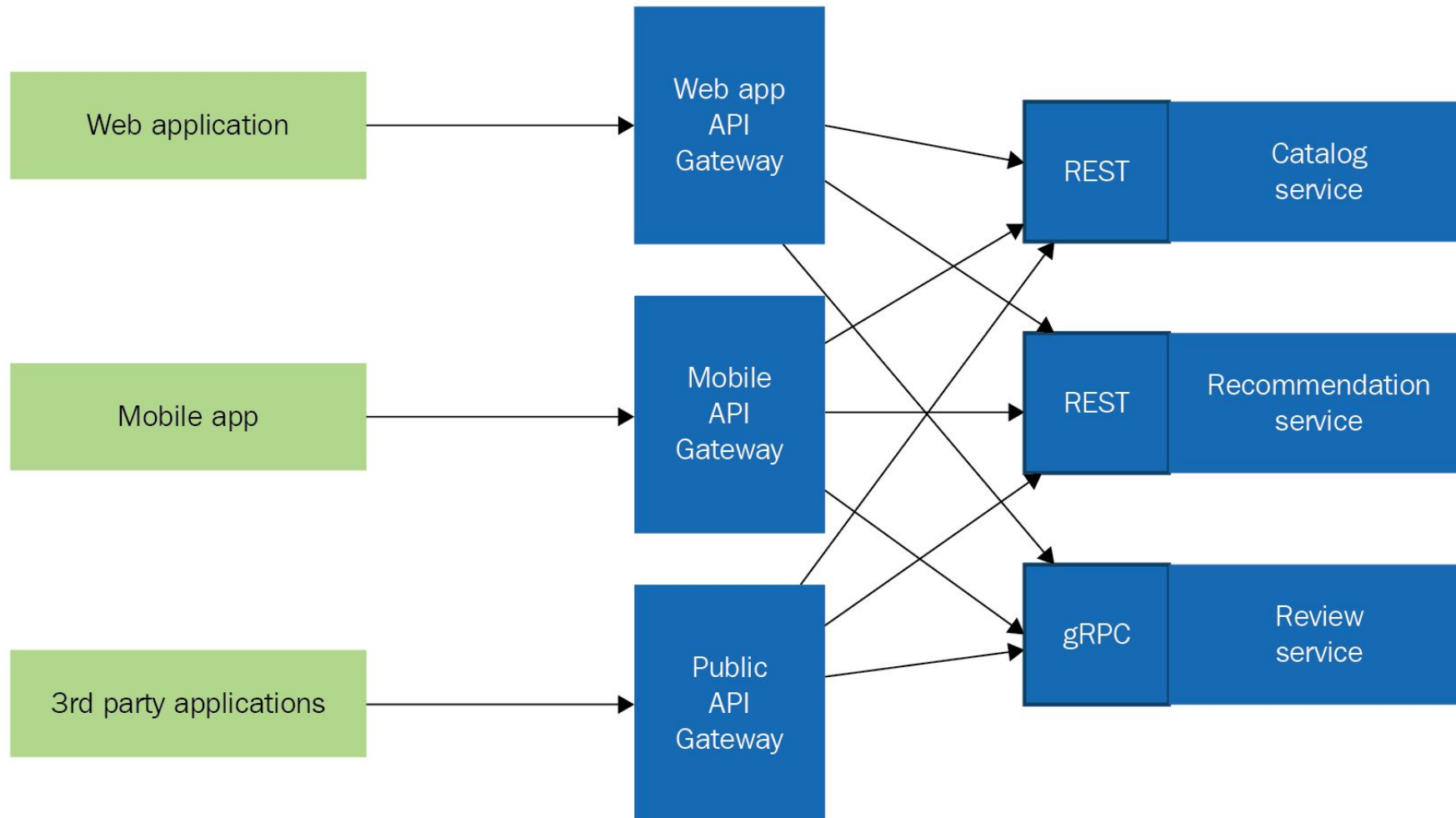


Api gateway



Api gateway

Variation: Backends for frontends



Benefits of api gateway

- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips. For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is essential for mobile applications.
- Simplifies the client by moving logic for calling multiple services from the client to API gateway
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally
- Response aggregate

Drawbacks

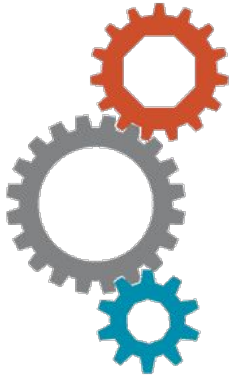
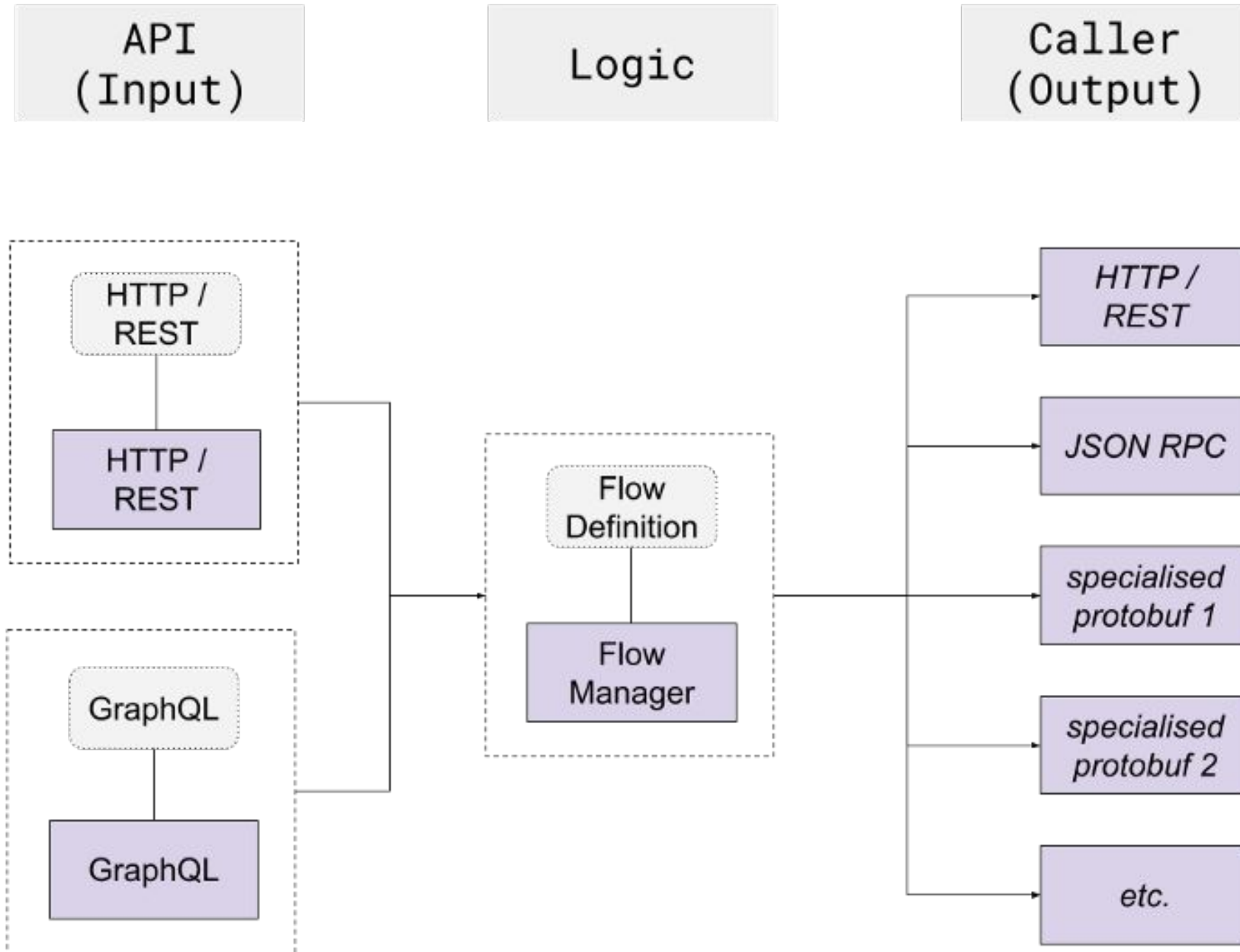
- Increased complexity - the API gateway is yet another moving part that must be developed, deployed and managed
- Increased response time due to the additional network hop through the API gateway - however, for most applications the cost of an extra roundtrip is insignificant.

Api orchestration

Building blocks

- **Identity** - The system doing the orchestration needs to be able to handle multiple identities to call the different backends. In some cases where the API is not open, there is a need to verify the caller by checking API keys, OAuth or SAML tokens.
- **Routing** - Depending on inputs and requests, the calls will need to get routed to the backends and also the responses will need to get routed to the initiator.
- **Caching** - Not only in complex systems but in general, API calls trigger to compute power, which is expensive. Caching helps reduce the load, keeping a copy of the last response in memory, serving it way quicker than going all the way to the backend. Caching is not possible for all API calls but for things that are quite static or very heavy, caching can be a good help.
- **Payload transformation** - Incoming calls will look very different from what backend APIs will require, so the payload needs to be able to be decomposed and transformed. The backend response sometimes needs transformation too. The consumer usually expects a composite or merged response. This is also called API mashup.

Api orchestration



Health checks

Problem: “How to detect that a running instance is unable to handle requests?”



Health checks handler performs various checks, such as:

- the status of the connections to the infrastructure services used by the service instance
- the status of the host, e.g. disk space
- application specific logic

Endpoint

`http://host:port/health`

`http://host:port/hc`

Use of health checks

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, a container orchestrator may respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might react to an unhealthy app by routing traffic away from the failing instance to a healthy instance.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

Elementary health check

```
public class BasicStartup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHealthChecks();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseHealthChecks("/health");

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                "Navigate to /health to see the health status.");
        });
    }
}
```

Custom health check signature

```
public class ExampleHealthCheck : IHealthCheck
{
    public ExampleHealthCheck()
    {
        // Use dependency injection (DI) to supply any required services to the
        // health check.
    }

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        // Execute health check logic here. This example sets a dummy
        // variable to true.
        var healthCheckResultHealthy = true;

        if (healthCheckResultHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("The check indicates a healthy result."));
        }

        return Task.FromResult(
            HealthCheckResult.Unhealthy("The check indicates an unhealthy result."));
    }
}
```

Register your custom health check

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddCheck<ExampleHealthCheck>("example_health_check");
}
```

Check your database server

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddSqlServer(Configuration["ConnectionStrings:DefaultConnection"]);
}
```

Logs

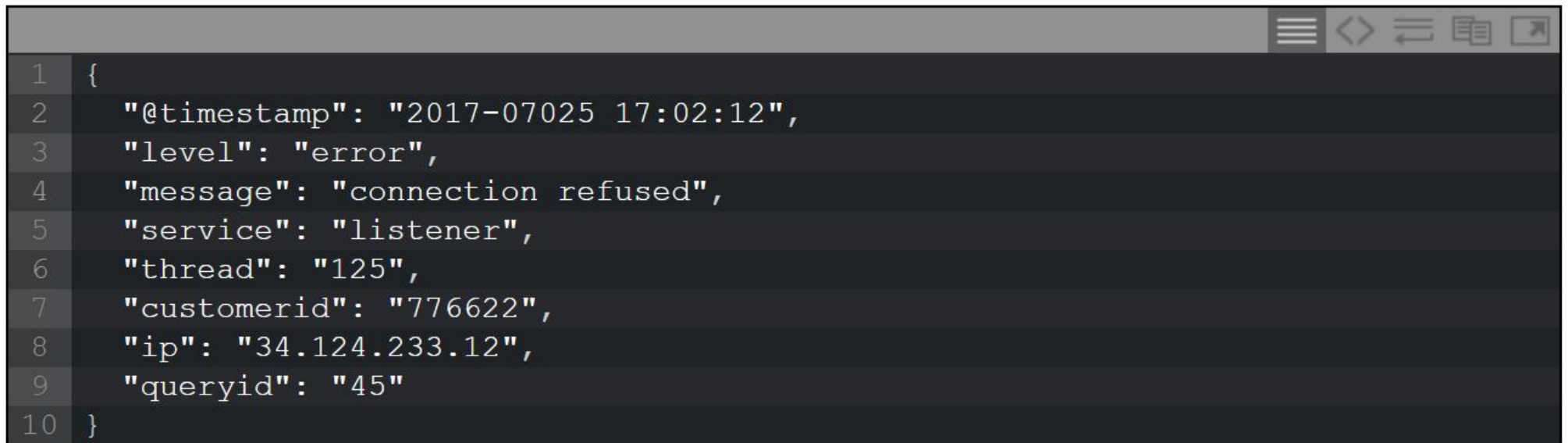
Best practices:

- Use a standard and easily configurable logging framework
- Use a standard structured format like JSON
- Don't let logging block your application
- Create a standard schema for your fields
- Automatically retry if sending fails
- Don't let local storage use all your memory or disk space
- Encrypt data in transit

Standardize your logs

General log message json format.

Example:

A code editor window with a dark theme and a light gray title bar. The title bar contains standard window controls (minimize, maximize, close) on the right. The editor area shows a JSON object with several fields. Line numbers 1 through 10 are visible on the left side of the editor.

```
1 {  
2   "@timestamp": "2017-07025 17:02:12",  
3   "level": "error",  
4   "message": "connection refused",  
5   "service": "listener",  
6   "thread": "125",  
7   "customerid": "776622",  
8   "ip": "34.124.233.12",  
9   "queryid": "45"  
10 }
```

Log levels

```
namespace Microsoft.Extensions.Logging
{
    ...public enum LogLevel
    {
        ...Trace = 0,
        ...Debug = 1,
        ...Information = 2,
        ...Warning = 3,
        ...Error = 4,
        ...Critical = 5,
        ...None = 6
    }
}
```


Most popular logging providers

- Log4net
- NLog
- Serilog

```
loggerFactory.AddNLog();  
or  
loggerFactory.AddSerilog();
```

```
using NLog;  
using System;  
  
namespace LoggingDemo.Nlog  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            LogManager.LoadConfiguration("nlog.config");  
            var log = LogManager.GetCurrentClassLogger();  
            log.Debug("Starting up");  
            log.Debug("Shutting down");  
            Console.ReadLine();  
        }  
    }  
}
```

```
2018-08-18 13:27:10.5022 | DEBUG | LoggingDemo.Nlog.Program | Starting up  
2018-08-18 13:27:10.5623 | DEBUG | LoggingDemo.Nlog.Program | Shutting down
```



Thank you!