Services Oriented Architecture

What is SOA?

- SOA – Service Oriented Architecture
- Application/software architecture
  - Organizational structure for software systems – SOA organize around services
- Services are provided by application components
  - Accessed over the network using communication protocol
  - Form of distributed computing
- Services may communicate directly with each other
  - Directly invoking the service
- Services may communicate thru a bus
  - No knowledge of the details of other services except their logical name
  - Some message bus provides very sophisticated routing capability which includes orchestration
- Services also use queues for communication

Example of SOA

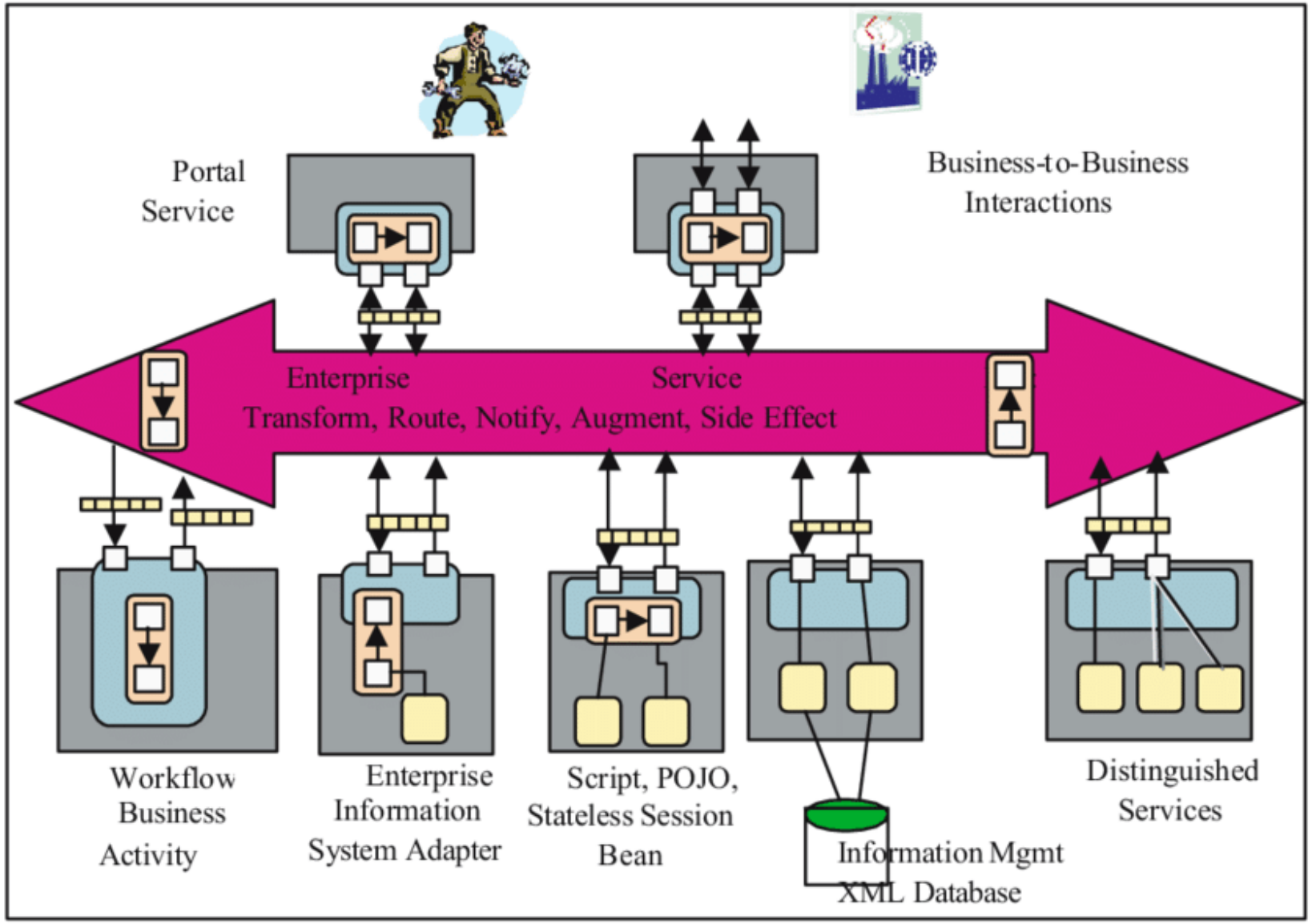


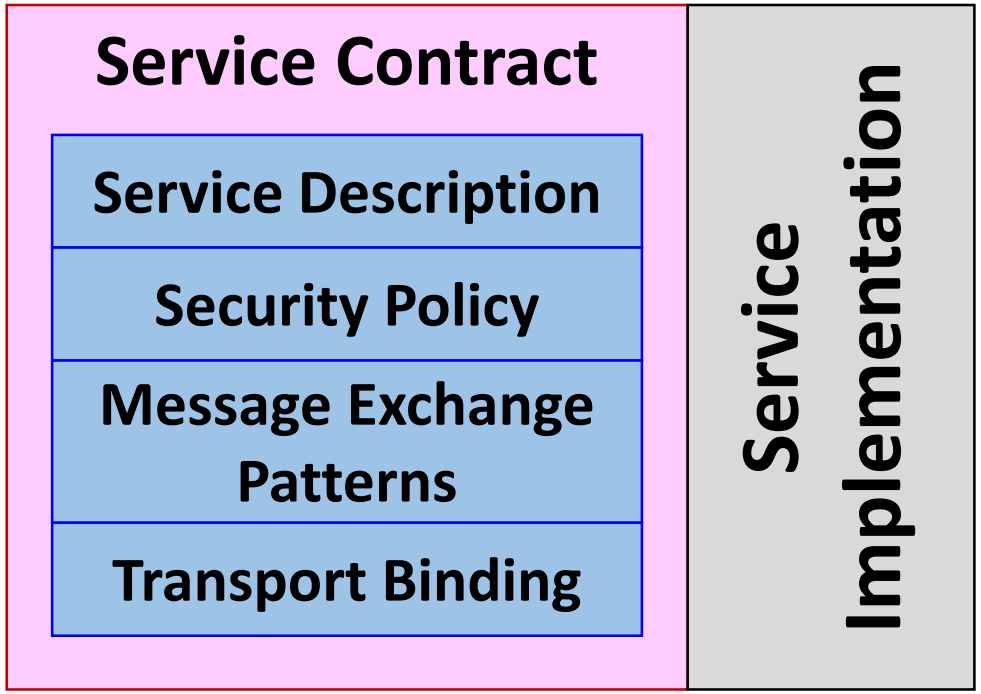
Image from https://www.researchgate.net/figure/A-general-service-oriented-architecture_fig1_221050852

Service Technologies and Implementations

- CORBA – Common Object Request Broker Architecture – 1990s
  - Quibble – technically CORBA exposes an object but think of object as a service
- COM/DCOM – Component Object Model – 1990s
  - Similar to CORBA
- RMI – Remote Method Invocation – late 1990
- SOAP – Simple Object Access Protocol – early 2000
  - XML based, works over may different transport eg. HTTP, SMTP
- RESTful – HTTP based
- gRPC – return to RPC based

What is a Service?

- Self describing discoverable element that performs a specific function
  - Logically represents a business activity eg. Customer order
  - Is self-contained and independent
- The implementation is hidden from the client
- Service contract describes a service; consists of
  - Service description
  - Security policy
  - Message exchange pattern
  - Transport binding



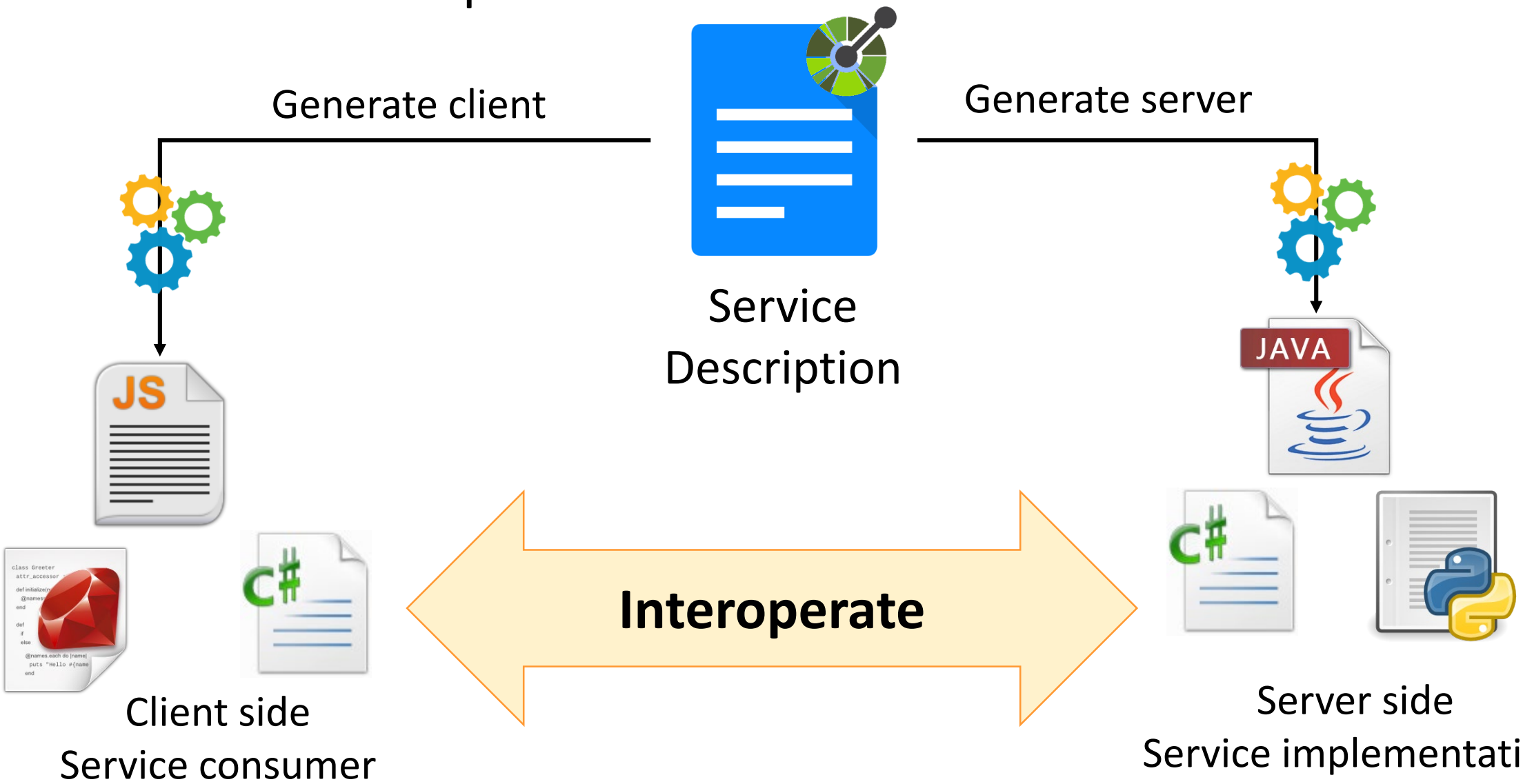
Service Description

- A service's metadata - describes the services
  - Data
  - Parameters, types, result, error
  - Binding information
  - Security policy
- Service description part of many distributed services framework
  - Eg. OpenAPI, IDL, WSDL
- Like a contract to the service's consumers
  - Service description can be used to generated the client, the server or both

```
paths:
  /customers:
    get:
      description: A list of customers
      operationId: getCustomers
      parameters:
      - name: limit
        in: query
        schema:
          type: number
         default: 20
      - name: offset
        in: query
        schema:
          type: number
     responses:
      200:
        content:
          application/json:
            schema:
              $ref: "customer-schema.json#/Customer"
```

Example of a service description in OpenAPI

Service Description



Service Description

- RESTful web services are described with
  - JSON schema - description of data type
  - OpenAPI - the service including request, response, parameters, etc
- Accessible by public
  - Eg. under `/schemas`
  - Can be use by 3rd party clients to generate libraries to call the service

Security Policy

- Describes what are the security requirements in order for a client to consume the service
- Includes
  - Application level security
    - Role based security
    - Login
  - Message level security – data at rest
    - Data integrity
    - Data confidentiality
    - Non repudiation
  - Transport layer security – data in transit

Authentication and Authorization

Authentication

- Verifies who you say you are
- Authentication required to consume the service
- Method
  - User name/password
  - Certificates
  - Challenge response
  - Biometric

Authorization

- Decides if you can perform an operation on a resource
- Authorization to access specific resources
- Purpose
  - Access control - determines if a request is allowed access or perform certain operations on a resource
  - Track and control resource usage

Securing RESTful Web Service

- OpenAPI supports the following authorization scheme
  - HTTP authentication
    - Uses `Authenticate` HTTP header
    - Including `Basic`, `Bearer`, etc.
    - See https://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml#authschemes for other schemes
  - OAuth 2.0
  - OpenID Connect
  - API Keys
    - Application specific keys
    - Passed either in HTTP headers or as query parameters

HTTP Basic Authentication

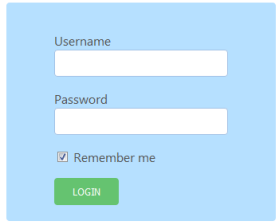
`WWW-Authenticate` indicates the type of authentication required in order for the client to gain access to the resource

```
GET /api/v1/customers
```

```
401 Unauthorized
WWW-Authenticate: Basic realm="acme"
```

Security realm

Authentication scheme; others include `Bearer`, `OAuth`, `vapid`

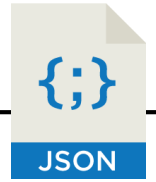


Authorization contains credentials for the scheme

```
GET /api/v1/customers
Authorization: Basic ZnJlZDp5YWJhZGFiYWRvbw==
```

```
btoa('fred:yabadabadoo')
```

```
200 OK
Content-Type: application/json
```



Example Authentication

```
app.get('/api/vi/customers', (req, resp) => {
   const auth = req.header('Authorization');
   if (!!auth) {
      resp.set('WWW-Authenticate', 'Basic realm="acme"');
      return resp.status(401).end();
   }
   const authInfo = atob(auth.split(' ')[1]).split(':');

   const conn = //get MySQL connection
   conn.connect()
   conn.query('select * from user where login = ? and password = sha1(?)',
      [ authInfo[0], authInfo[1] ],
      (err, result) => {
         conn.end();
         if (result.length <= 0)
            return resp.status(403).end();
         //continue with process
      }
   )
});
```

Check if request has the requisite Authorization header to make this request

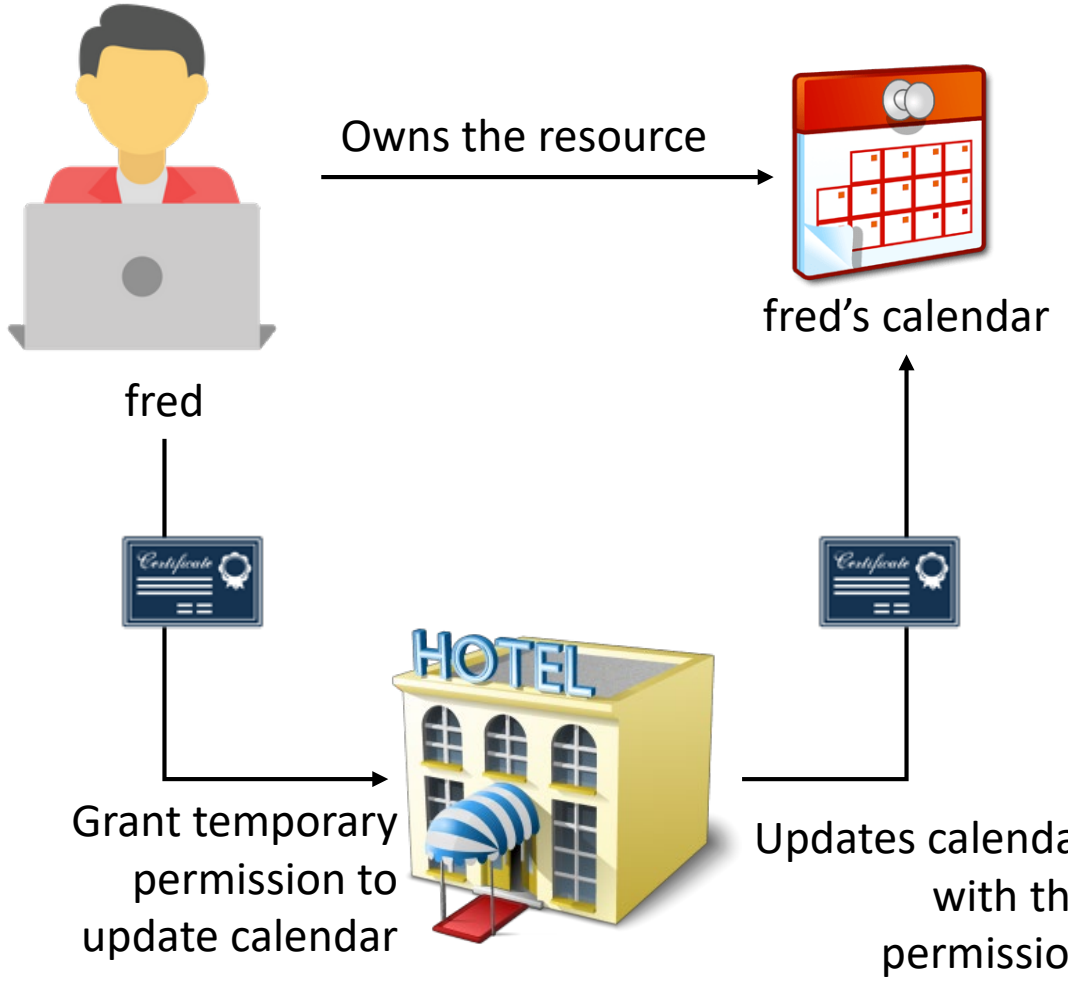
Split and unbase64 `Basic ZnJlZDpmcmVk` to username and password

Check the user/password against the database. Note the password is stored as a hash, so need to hash to hash the password

If no record is found, then return a 403 status

OAuth 2 and OpenID Connect

- An authorization mechanism to allow a 3rd temporary access your resources without revealing the password
  - Eg. contacts, calendar, drive
- User can revoke the access anytime
- OpenID connects allows a 3rd party to access the user's profile
  - Eg. email, real name, city, etc from social media
  - Using the OpenID provider as a source of authentication for the user
- Scope specifies what actions are permissible on the resource



OAuth 2 Flow

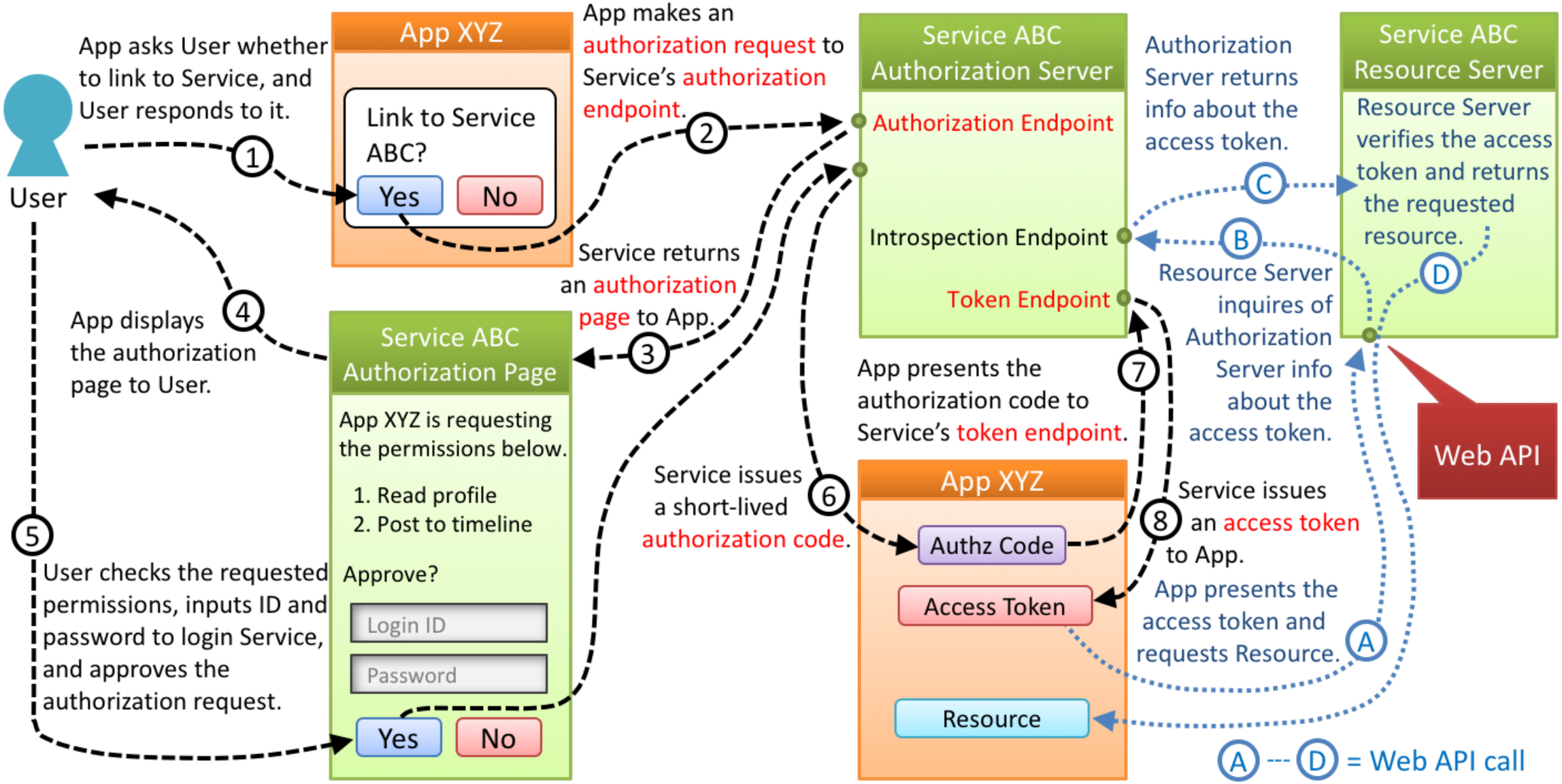


Image from https://medium.com/@darutk/diagrams-and-movies-of-all-the-oauth-2-0-flows-194f3c3ade85

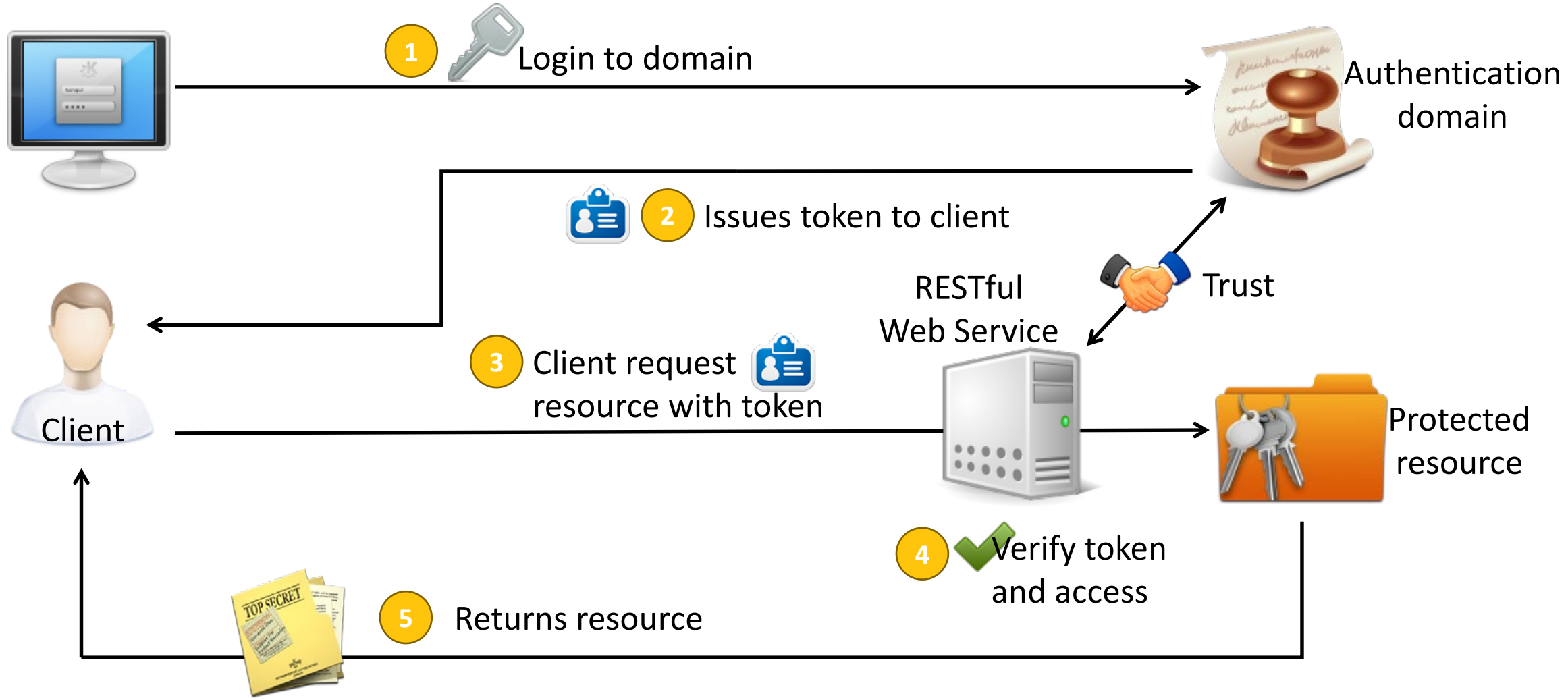
API Key Based Authentication

- Clients are issued cryptographic tokens after they have successfully authenticated with an identity provider
  - Tokens can be pre-generated
  - Dynamically generated whenever a user authenticates - short-lived tokens
- Tokens are stateless
  - Holds all information required to authenticate the token bearer
  - Does not require other external resources
- Tokens/API keys are the most common way to authorize access to RESTful resources
- JSON Web Token (JWT) is a popular standard
  - Bearer token viz access will be granted to the holder (bearer) of the token

Typical Information in a Token

- Token consist of the following
  - Issuer
  - Validity
  - Subject
  - Assertions about the subject - eg. email, job grade, address, etc.
- Immutable - cannot be changed once issued
  - But can be exchanged for a different type of token
  - Usually digitally signed
- Trust in the token issuer, implicitly trust the token
  - Like NRIC

Token Based Security



JSON Web Token

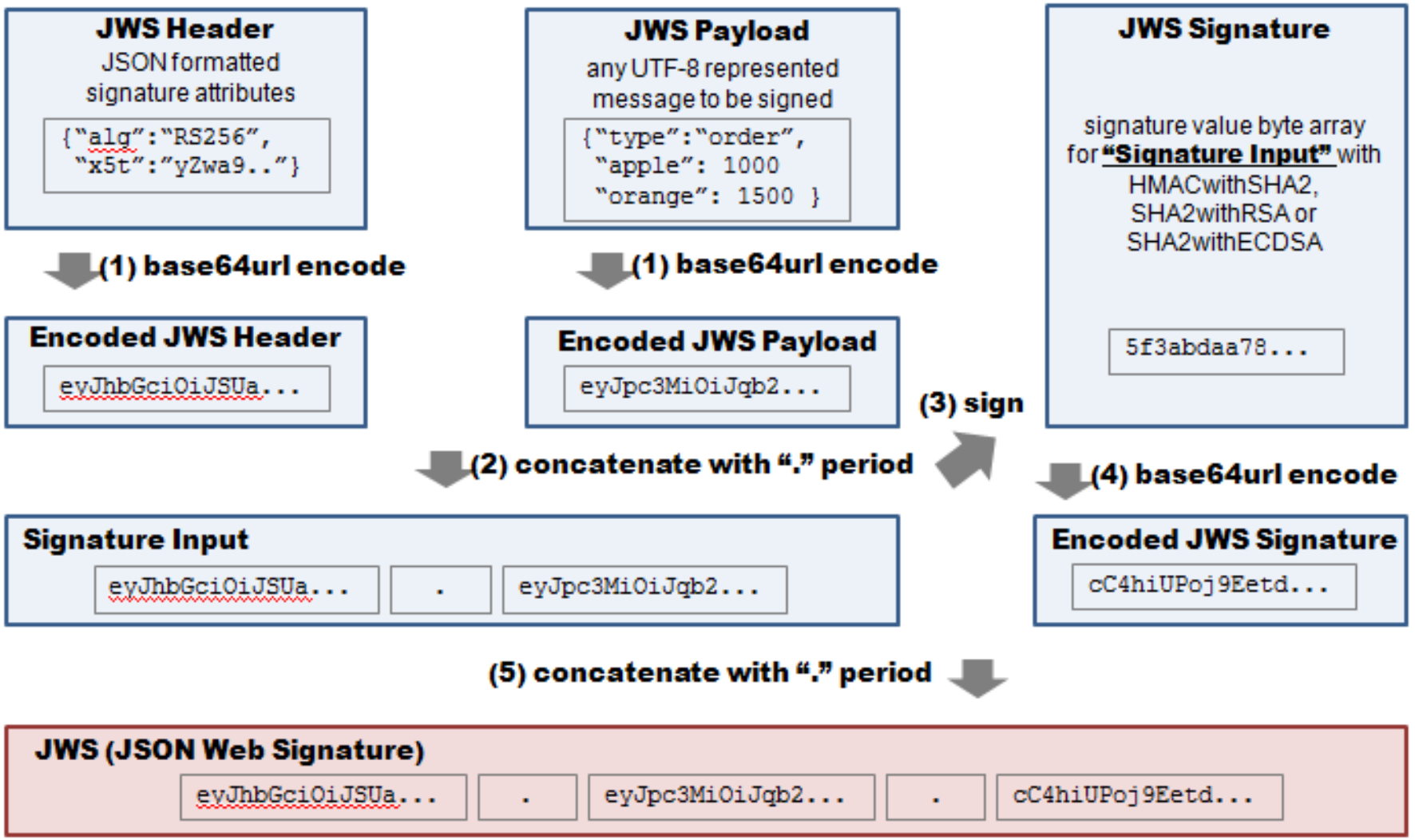


Image from http://kjur.github.io/jsjws/jws_generate.png

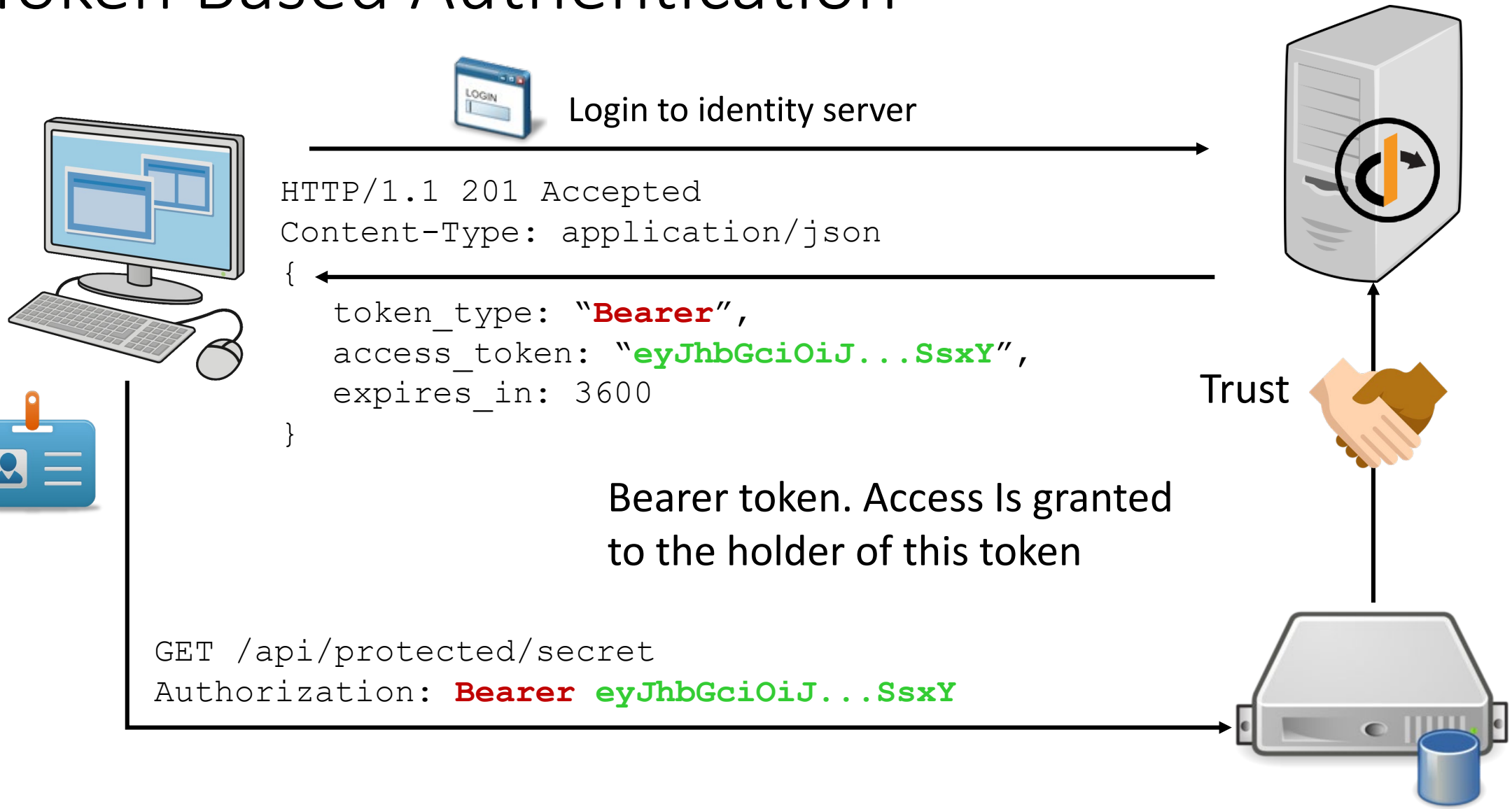
Bearer Token

- Tokens that are passed use by the client (bearer) to access protected resource
- Passed to the server on every HTTP request
  - Typically in HTTP header

```
Authorization: Bearer <token>
```

Use `Bearer` keyword to indicate that the token is a bearer token

Token Based Authentication



Example OpenAPI Bearer Token

```
openapi: "3.0.2"
...
component:
   securitySchemes:
      appToken:
         type: http
         scheme: Bearer
         bearerFormat: JWT
```

Define a security scheme to be used by the application

HTTP Authentication using Bearer scheme

```
paths:
   /api/customer/{cid}
      get:
         parameters:
         - name: cid
           in: path
           schema:
              type: number
         security:
         - appToken: []
         response:
            200:
               description: OK
               content:
                  application/json
                  ...
            401:
               description: Unauthorized
```

Use the `appToken` security scheme for this resource

Possible responses including the unauthorized response

Example Requesting a Token

```
import { JwtModule } from '@auth0/angular-jwt';
@Module({
   imports: [
      HttpClient,
      JwtModule.forRoot({
         config: {
            tokenGetter: () => sessionStorage.getItem('access_token')
         }
      })
   ]
})
export class AppModule { }
```

Configure `JwtModule` on how to retrieve the token

session Storage

Authenticate against the identity provider. If it is successful, store the return token in `sessionStorage` under

```
authenticate(user: string, password: string) {
   this.http.post('/login', { user: 'fred', password: 'yabadabadd' })
      .then((token) => {
         sessionStorage.setItem('access_token', token.access_token);
      })
}
```

Example Generating a JWT Token

```
const jwt = require('jsonwebtoken');
app.post('/login', (req, resp) => {
   const user = req.body.user;
   const password = req.body.password;
   if (!valid(req.body.user, req.body.password))
      return resp.status(401).end();

   const token = jwt.sign({
      sub: req.body.user,
      iss: 'the boss',
      iat: (new Date()).getTime(),
      exp: '1h',
      data: { /* valid claims */ }
   }, 'top secret');
   resp.status(200).type('application/json');
   resp.json({
      token_type: 'Bearer',
      expires_in: 3600,
      access_token: token
   })
})
```

Returns unauthorized status if authentication fails

Sign the token with the secret key

Standard attributes

Application specific claims

Return the token to the client

Express

Example Authenticating

Verify request has a JWT bearer token

```
app.get('/api/customer/:cid',
   (req, resp, next) => {
      const auth = req.get('Authorization');
      if (!(auth && auth.startsWith('Bearer ')) {
         resp.status(403).json({error: 'Not authenticated'}); return;
      }
      const token = auth.substring('Bearer '.length);
      try {
         req.jwt = jwt.verify(token, 'top secret');
         next();
      } catch (e) {
         resp.status(403).json({error: e}); return;
      }
   },
   (req, resp) = > {
      req.jwt.data /* Use the data to control access control */
   }
)
```

Check if the request has the `Authorization` header and that it is a `Bearer` type authorization

Verify and decode the token. Add the decoded token to the request object so that it is available in subsequent middleware

Express

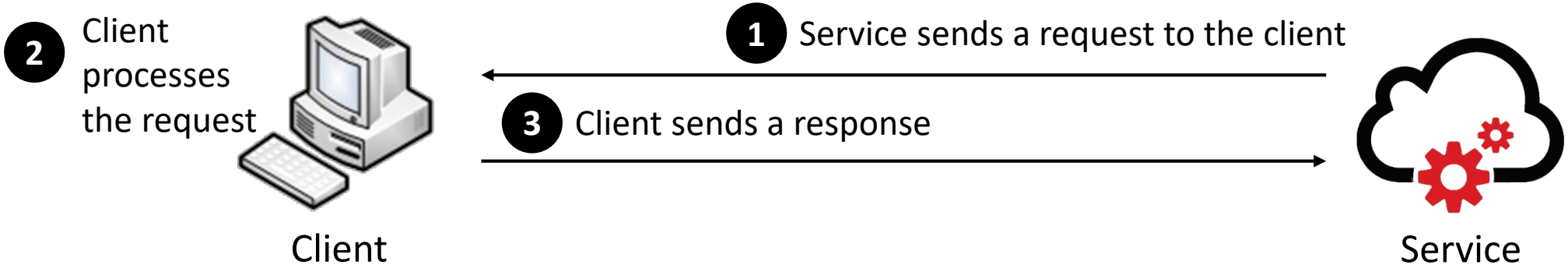
Message Exchange Patterns

- Describes how messages are exchanged between 2 communicating parties
  - Eg. going to buy a hamburger vs having one delivered to you
- HTTP as a protocol is a unidirectional request-response protocol
  - Client always initiates the message exchange
  - Not possible for the server to sent unsolicited messages
- Message Exchange Pattern refers to message exchange at the service/application level
  - Not at the protocol level but still uses HTTP

Message Exchange Patterns

- Request/Response - a request arrives at the service, the server processes the request and sends a response back to the client
- AJAX communications
- Communications between service may be stateless or stateful
  - Eg. HTTP uses session id in the form of cookie to track the state of the communications

Message Exchange Patterns



- Solicit Response - reverse of Request/Response pattern
- Use case - receiving an out-of-band order status confirmation
  - Eg. Service sends the order status confirmation to client, client responds with a status received confirmation
- Employs a webhook for callback, if using HTTP
  - Webhook can be predetermined or ‘well-known’
  - Dynamically register by client on demand

Example Checkout

Client

Internet

Shopping Service

Intranet

Order Service

POST /api/checkout

200 OK

POST /api/order
...
{ "callbackUrl": "https://.../status",
... }

202 Accepted

Callback when order has been created

POST /api/status
...
{ "orderId": "abc123", ... }

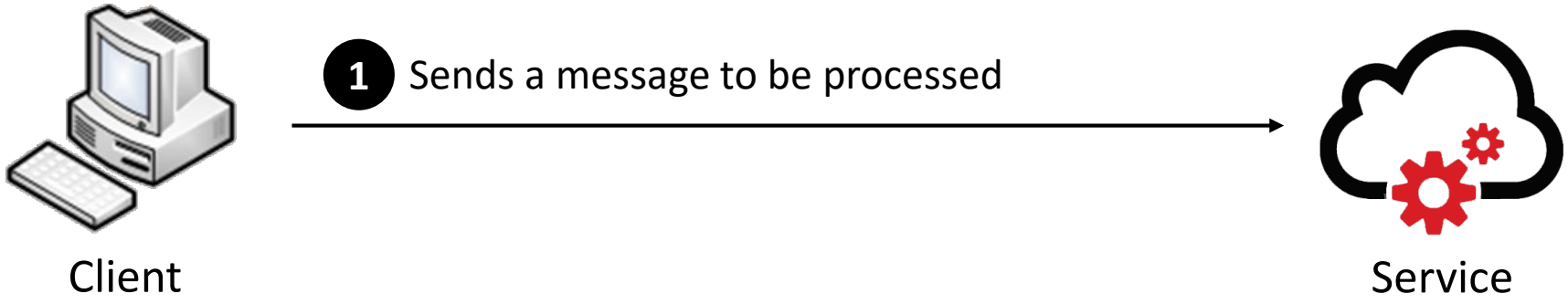
200 OK

Example Dynamic Callback Registration

```
openapi: '3.0.2'
...
paths:
  /api/order
    post:
      callbacks: ...
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                callbackUrl:
                  type: string
                  format: uri
      responses:
        '202':
          description: checkout
```

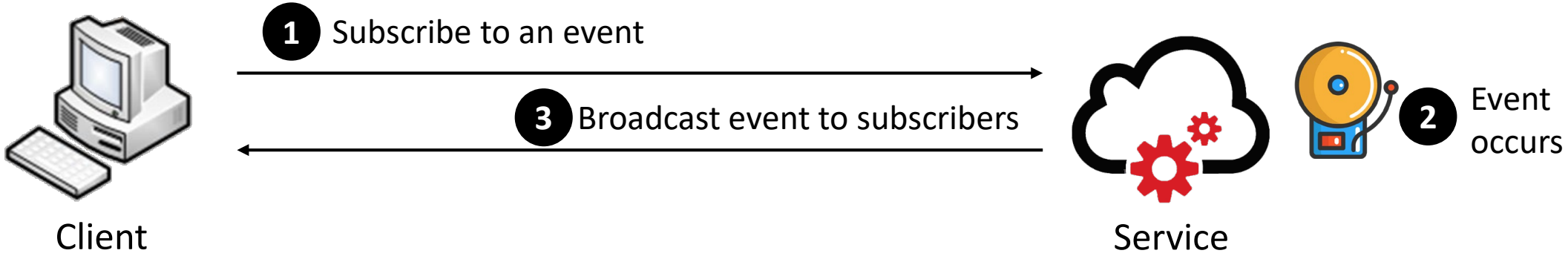
```
callbacks:
  orderConfirmed:
    '{$request.body#/callbackUrl}':
      post:
        requestBody:
          required: true
          content:
            application/json:
              schema:
                type: object
                properties:
                  orderId:
                    type: string
                ...
        responses:
          '200':
            description: accepted
          '400':
            description: bad order
```

Message Exchange Patterns



- One-Way - a message is sent to the service; the service produces no response other than received
  - Typical status code is 202 Accepted
  - Fire and forget
- Quality of a one-way service is that they are reliable
  - No way for a client to know if an operation failed after successfully submitted
  - Client assume the service is reliable eg. like post box
- Use case - thumbnailing service
  - Thumbnail service

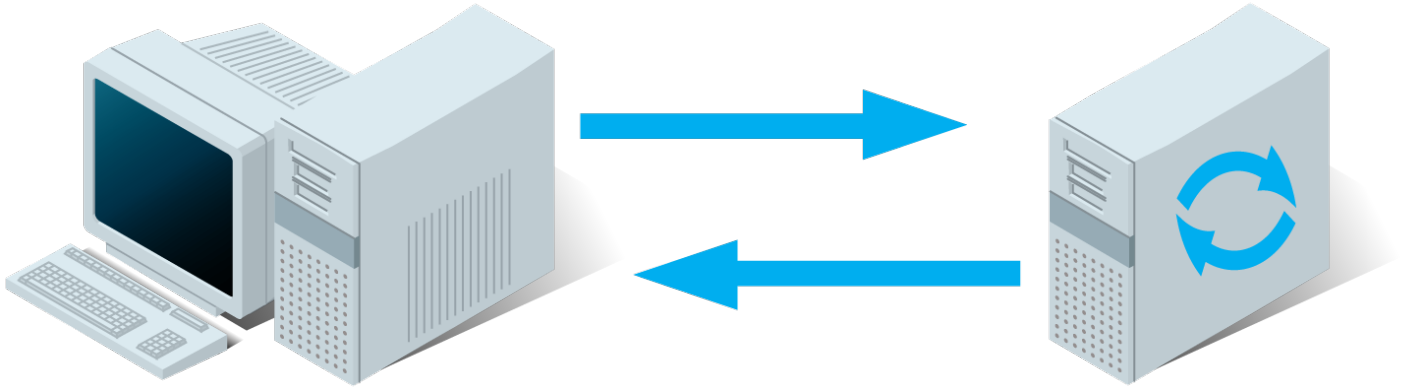
Message Exchange Patterns



- Notification - the service sends message/events to the client; client does not need to send a respond back to the service
- Use cases - publish/subscribe or general broadcast
  - Client receives subscription for latest news

Server Notifications

- HTTP is a request/response protocol
  - The browser must first make a request before the server can respond

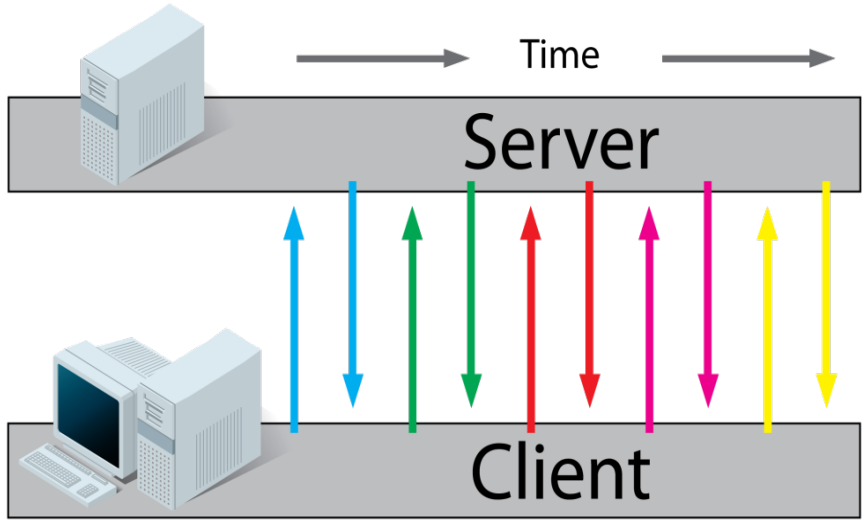


- There are lots of use cases where the server needs to notify clients/browsers that an important event has happened
  - Eg. Breaking news, stock price alerts, Facebook updates, etc.

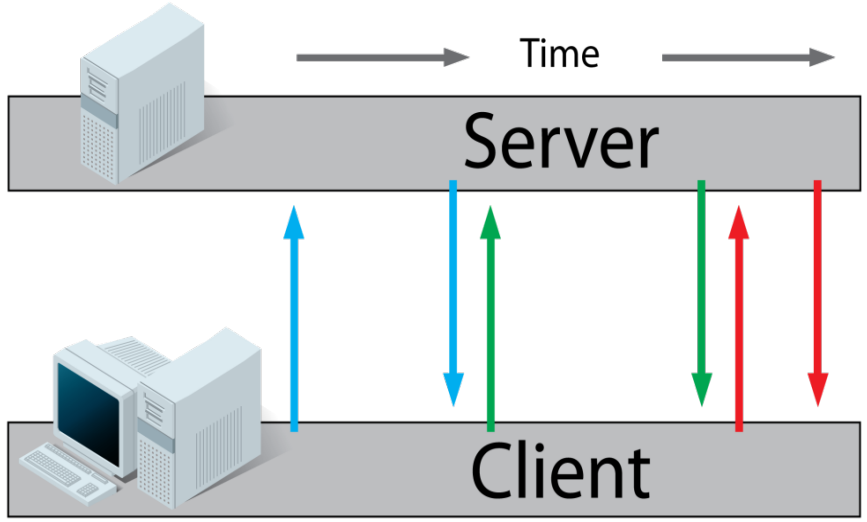
Images from: http://stackoverflow.com/questions/11077857/what-are-long-polling-websockets-server-sent-events-sse-and-comet

AJAX Polling and Long Poll

- AJAX polling is the technique where the client sends AJAX request at regular interval to the server
  - If there are any data, the server will return the data
  - Otherwise the server will return a no-op and closes the connection
- Long poll is a variation of AJAX polling
  - AKA Comet
  - Behaves like AJAX polling if the server has data
  - But if the server does not have any data, the server will hold on to the connection until
    - New data is available
    - Passes a certain duration (timeout) where the server will send a no-op and closes the connection



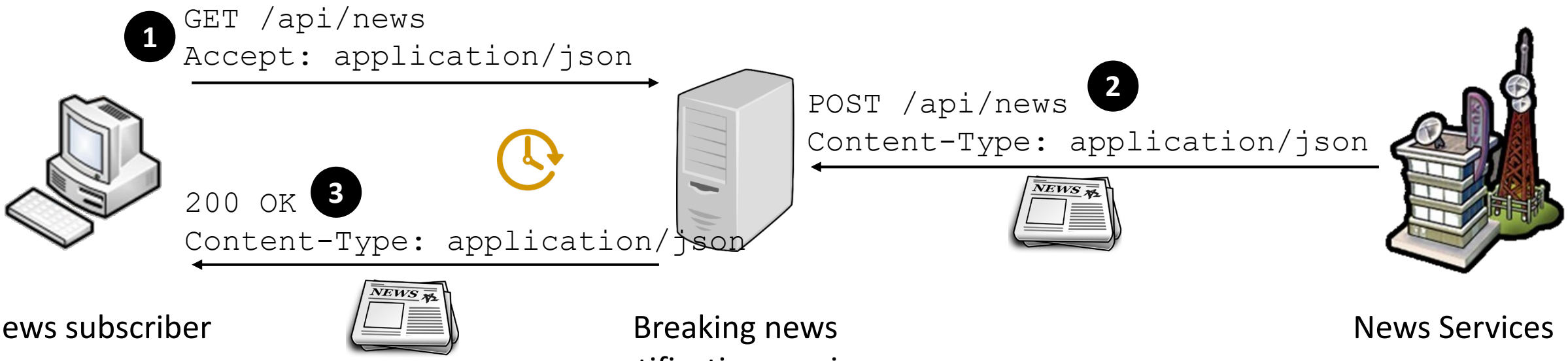
AJAX Polling



AJAX Long Poll/Comet

Images from: http://stackoverflow.com/questions/11077857/what-are-long-polling-websockets-server-sent-events-sse-and-comet

Example Long Poll



1. Client subscribes to news service
2. News services publishes news to the notification service
3. Use Comet to push content back to subscribers

Example Long Poll

```
getNews() {
   this.http.get<News>(`/api/news`)
      .toPromise()
      .then((n: News) => {
         //Do something with news
         ...
         this.getNews();
      })
}
```

Call `/api/news` again to wait for the next update

```
200 OK
Content-Type: application/json

{ "headline": "....", ... }
```

Express

Create longpoll for Express application

```
const longpoll = require('express-longpoll')(app);

longpoll.create('/api/news');
```

Callback used by news services to publish news

```
GET /api/news
Accept: application/json
```

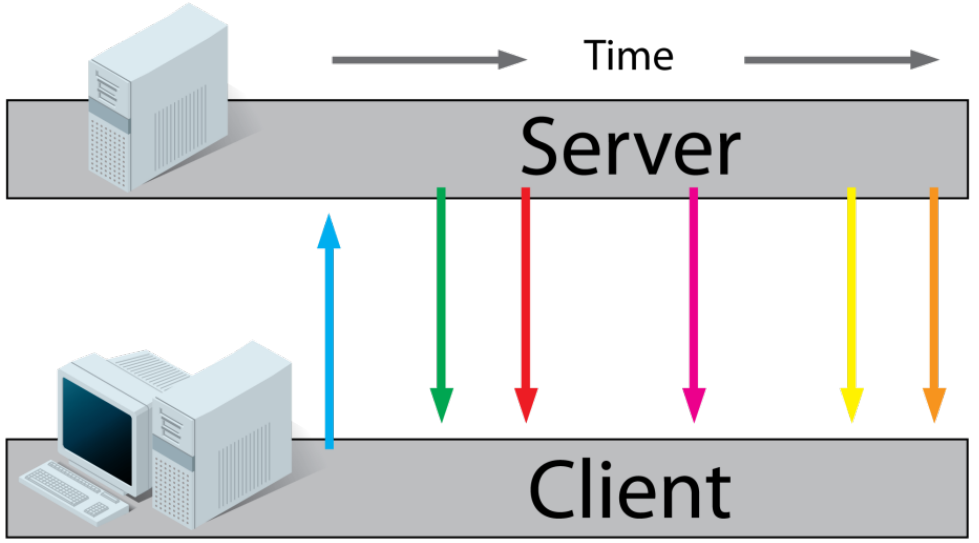
```
app.post('/api/news', (req, resp) => {
   longpoll.publish('/api/news', req.body);
   resp.end();
});
```

Publishes the news back to all subscribers

News

Server Sent Event

- Server sent event is a technique that allows the server to send notification/data to a browser
- This is known as the push model
  - AJAX is pull model
- Under normal situation after the server has sent the HTML page back to the browser, the server will close the connect
- In SSE, the server keeps the connection open
  - Continue to use the connection to send data to the browser



- The data that the server sends to the client are datagram
  - Viz. Each piece of information that the server sends is self contained
  - Eg. cannot stream the data
- The connection is unidirectional
  - The client cannot use this connection to send data back to the server
  - If the client wants to send data to the server, it must open a new connection eg using AJAX

Server Sent Event Protocol

- SSE is standard HTTP
  - Uses a `GET` method
  - `Accept` header is set to `text/event-stream` MIME type
- The server will then keep the connection open
  - Data packets are separated by a blank line (`\n`)
  - Data in packets are prefixed with `data:`

Server Sent Event Example

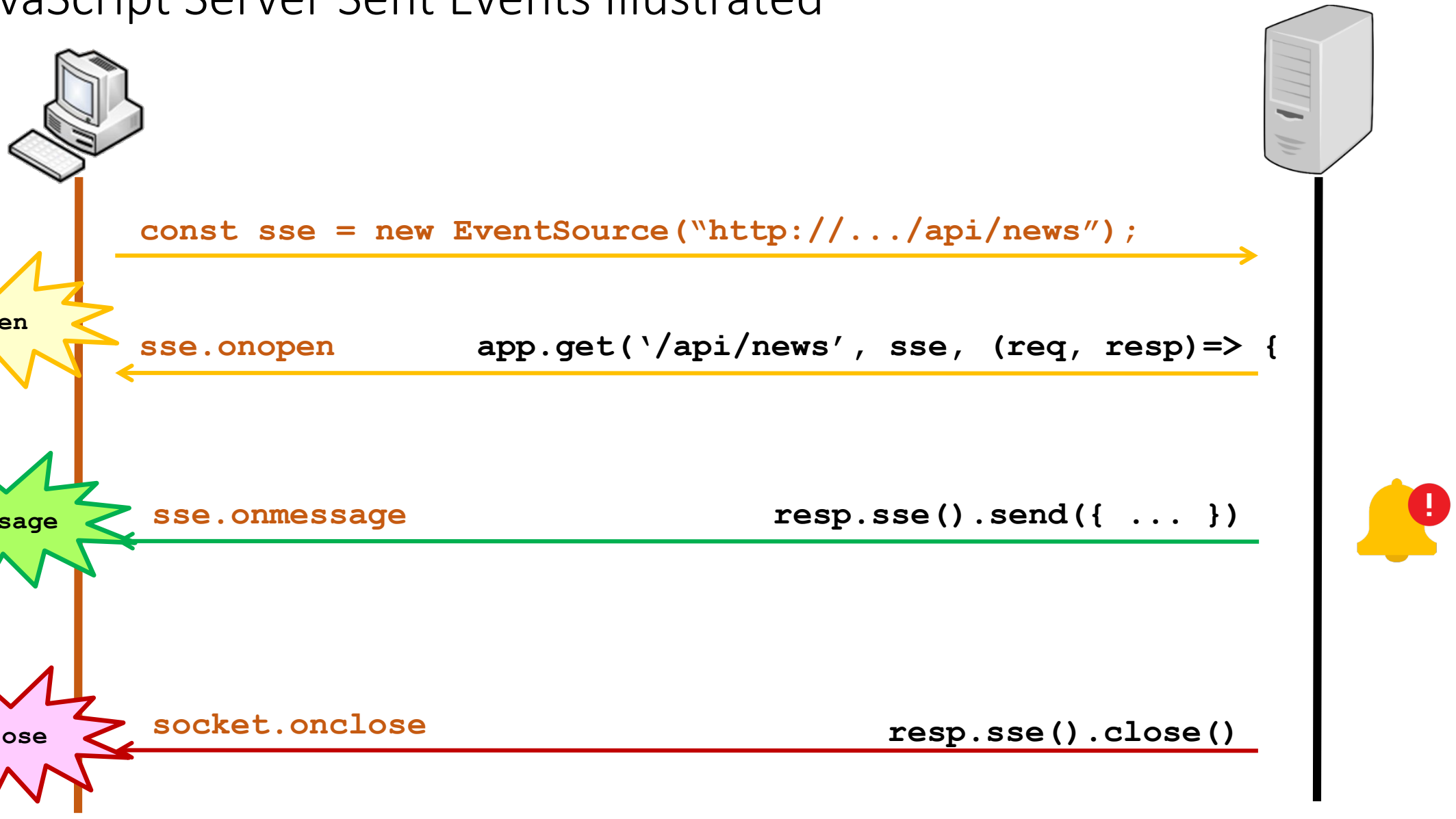
```
GET /api/news HTTP/1.1
Host: localhost:8080
Accept: text/event-stream
```

```
HTTP/1.1 200 OK
Content-Type: text/event-stream
\r\n
data: {"headline":"Rumor has it there's a cunning plan
to bring back Blackadder for fifth season" ...}\n
\n
data: {"headline":"The world's largest undersea
restaurant", ...}\n
...
\n
```

Datum

Datum

JavaScript Server Sent Events Illustrated



Example Server Sent Event

```
getNews(): Observable<News> {
   return (Observable<News>.create(
      (obs) => {
         const sse = new EventSource('http://.../api/news');
          sse.onopen = () => {
            /* do something when connection is opened */
          }
          sse.onmessage = (event) => {
            obs.next(<News>JSON.parse(event.data))
          }
      }
   ));
}

getNews()
   .subscribe(news => {
      /* do something with news */
   })
```

Endpoint must support CORS if it is cross domain

Listen to the open event

Parse the data and use RX to publish the data to subscribers

Subscriber

Example Server Sent Event

Express

```
const events = require('events')
const expSSE = require('express-sse-middleware');
const builder = expSSE['builder'];
const sseMW = expSSE['default']

const sub = [];

app.use(cors());

app.get('/api/news', sseMW, (req, resp) => {
   sub.push(resp.sse());
})

app.post('/api/news', (req, resp) => {
   const data = builder.builder().data(news).build();
   sub.forEach(sse => sse.send(data));
   resp.end();
})
```

SSE middleware setup the SSE connection

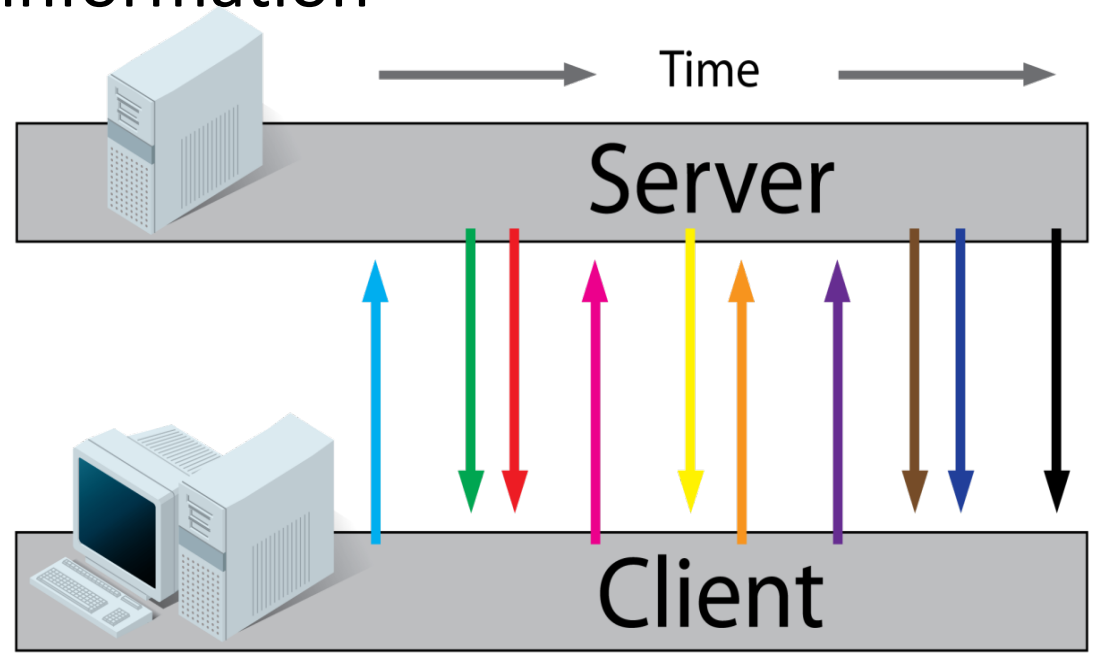
Retrieve the SSE connection and save it to a array

Push the event when the news source publishes a new headline

WebSocket

- Bi-directional socket connection between client and service
- Stateful in the sense that a socket connection connects on client to one service
- Connection is persistent until explicitly closed by either client or service
- Has lower overhead and faster than the other methods

- Typical use case includes performing near real-time pushes for news, stock information



Images from: http://stackoverflow.com/questions/11077857/what-are-long-polling-websockets-server-sent-events-sse-and-comet

WebSocket

- WebSocket is considered as part of the HTML5 platform
  - Is defined in the communication part of the HTML5 specification
- WebSocket connections are established by upgrading from a HTTP GET request
- Once a connection has been established, the client and server can communicate in full-duplex
  - Data are send between the client and server in frames
  - Data can be text or binary

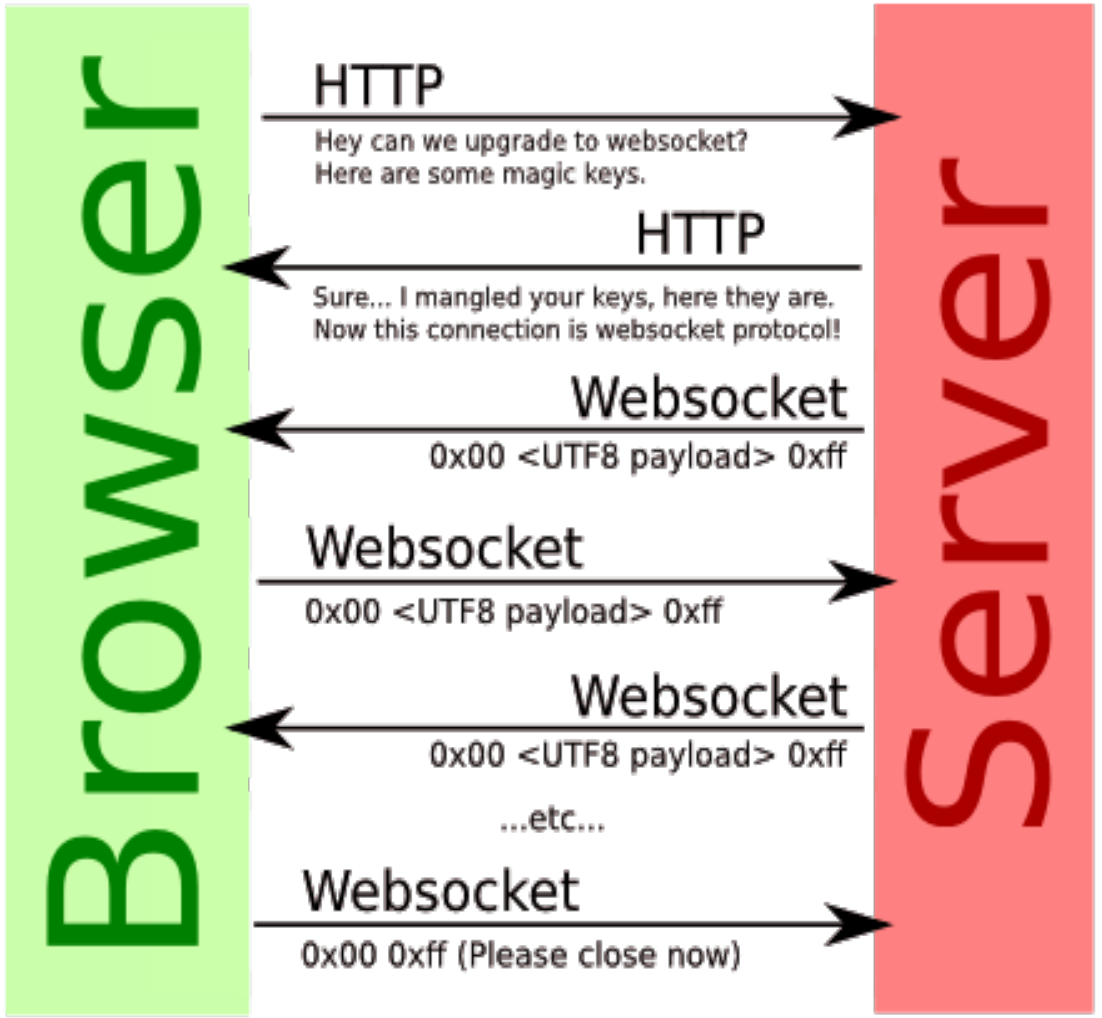
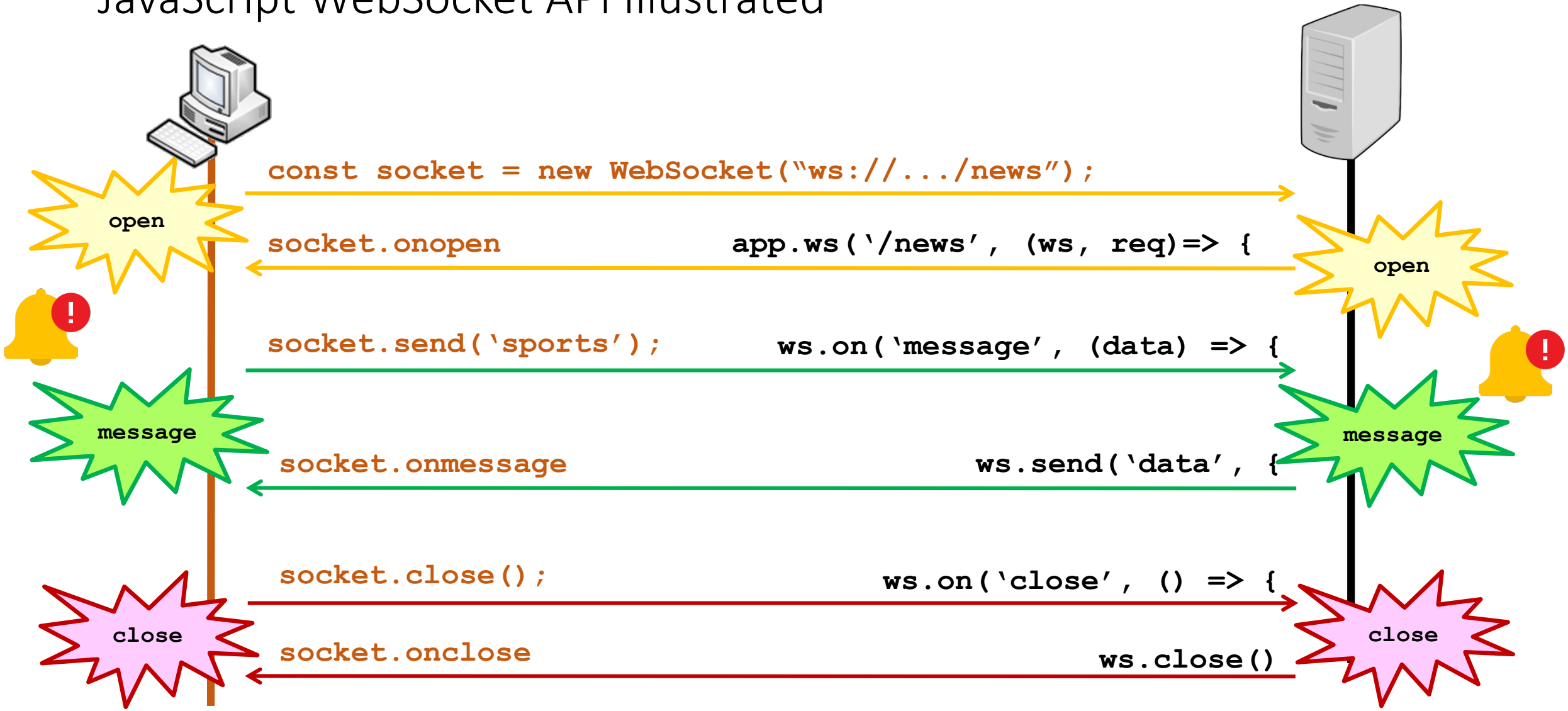


Image from http://uwe.stevelacey.net/atwd/assignment2/images/xwebsocket-lifecycle.png.pagespeed.ic.weBP3HlFSb.png

JavaScript WebSocket API Illustrated



Example Web Socket

```
getNews(): Observable<News> {
   return (Observable<News>.create(
      (obs) => {
         const ws = new WebSocket('ws://.../api/news');
          ws.onopen = () => {
            /* do something when connection is opened */
          }
          ws.onmessage = (event) => {
            obs.next(<News>JSON.parse(event.data))
          }
      }
   ));
}

getNews()
   .subscribe(news => {
      /* do something with news */
   })
```

Note the `ws` scheme instead of `http` scheme

Endpoint must support CORS if it is cross domain

Listen to the open event

Parse the data and use RX to publish the data to subscribers

Subscriber

Very similar to Server Sent Events

Example Web Socket

Express

```
const express = require('express');

const app = express();
const appWS = require('express-ws')(app);

app.ws('/api/news', (ws, req) => {
  const topic = req.params.topic

  ws.on('message', (payload) => {
    const data = JSON.parse(payload);

  });

})
```

Add WebSocket Express middleware to an Express application

Use `ws` to listen to WebSocket routes

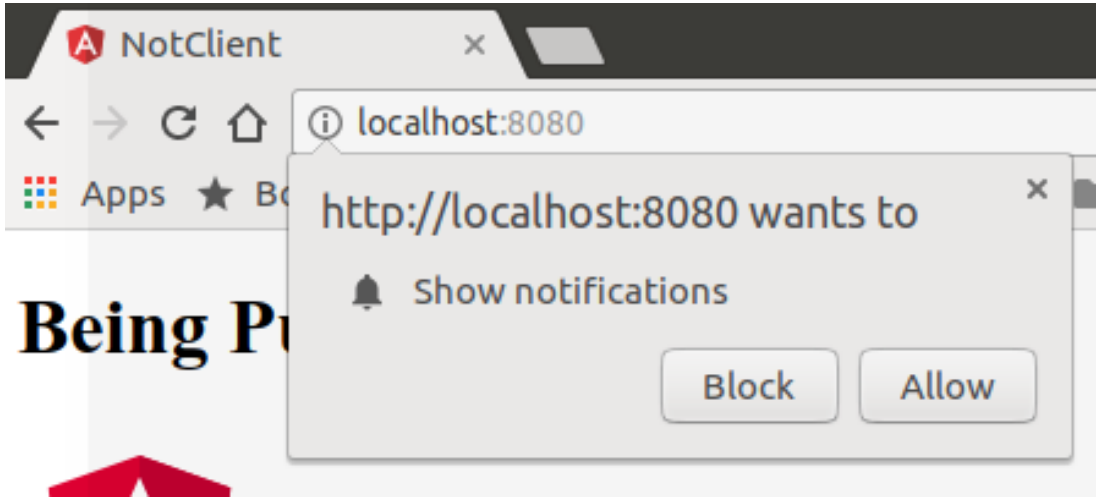
The WebSocket connection is passed to the middleware. This is the active connection between the server and the client

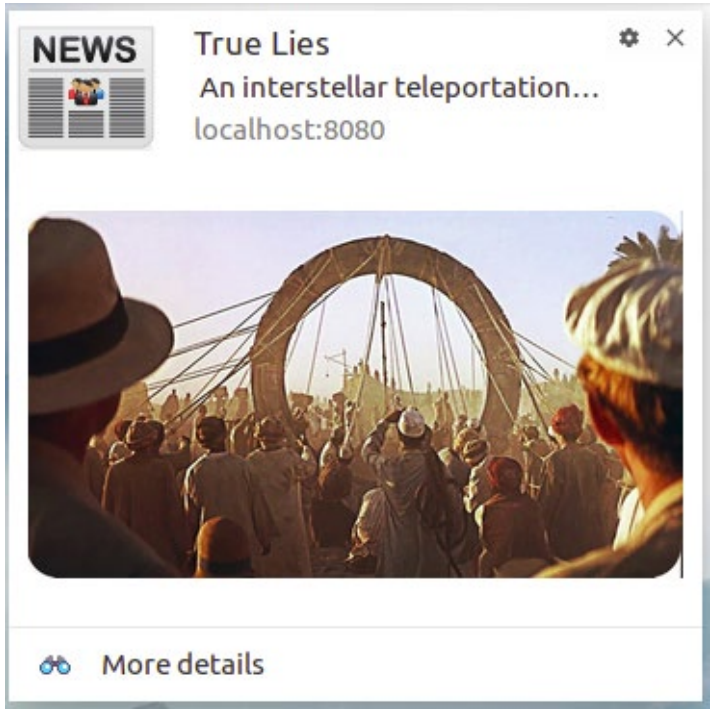
Add a listener to the message event. Listener will be called whenever client sends

Data are passed as string. If using JSON as transfer format, then parse string to JSON

Push Notification

- Comet, server sent events require that the client to be online in order for the server to push data to them
- Offline notification allow the user to be notified even though the application is not running
  - Eg. Social media notification, messaging notification
- Notification can be used as a trigger to launch the application
  - Eg. Notification to chat





Subscription

VAPID key pair

Public key 

Private key 

Push Service 

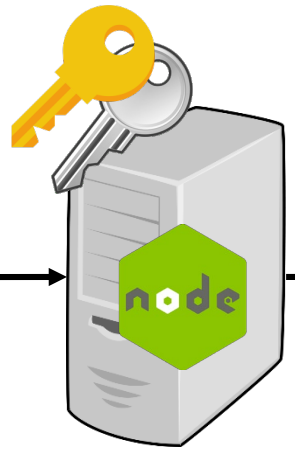
2 Subscription endpoint is tied to the public key

1 Subscribe with public key 

Users

Application

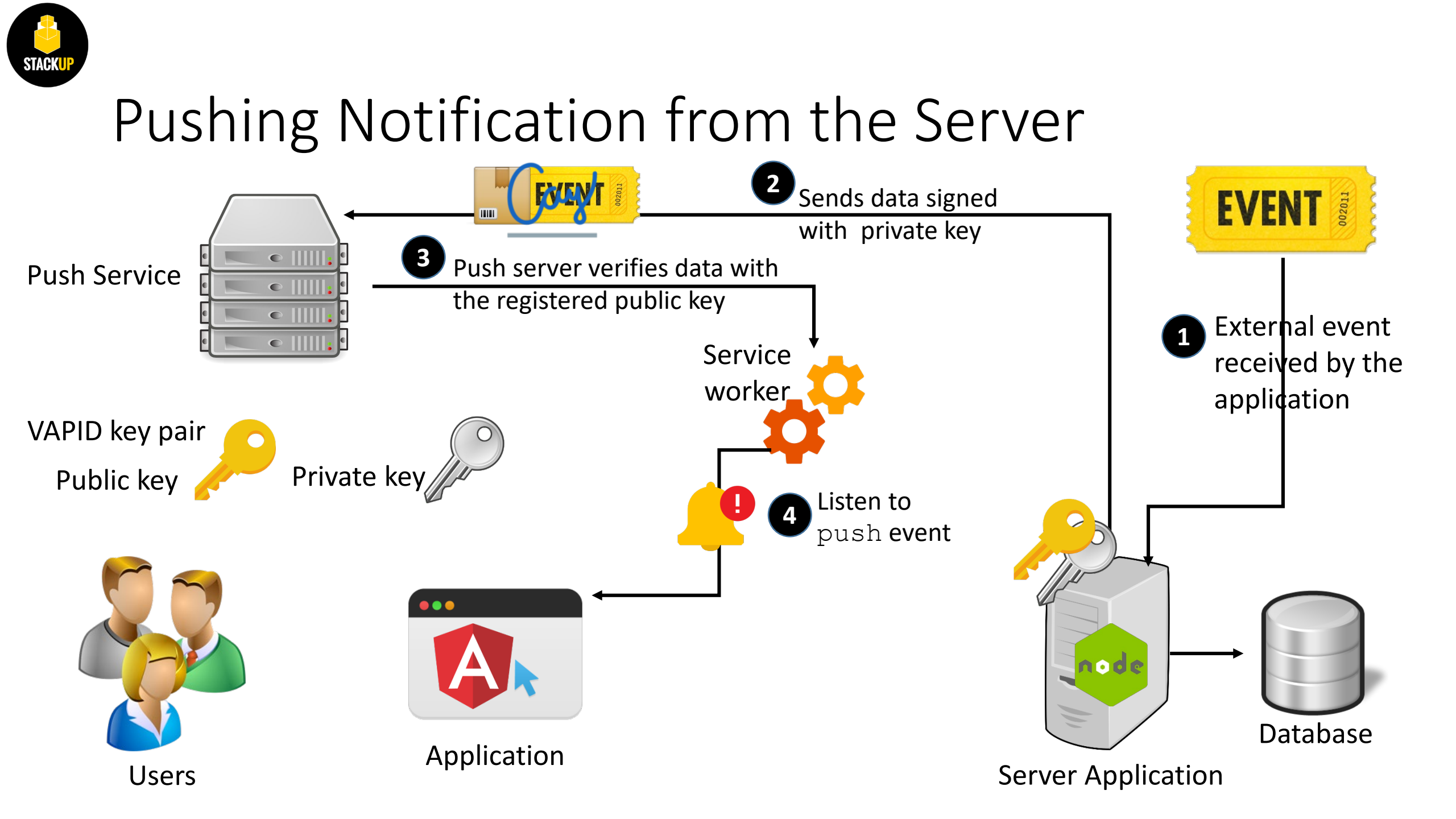
3 Send subscription to application



Server Application

Database

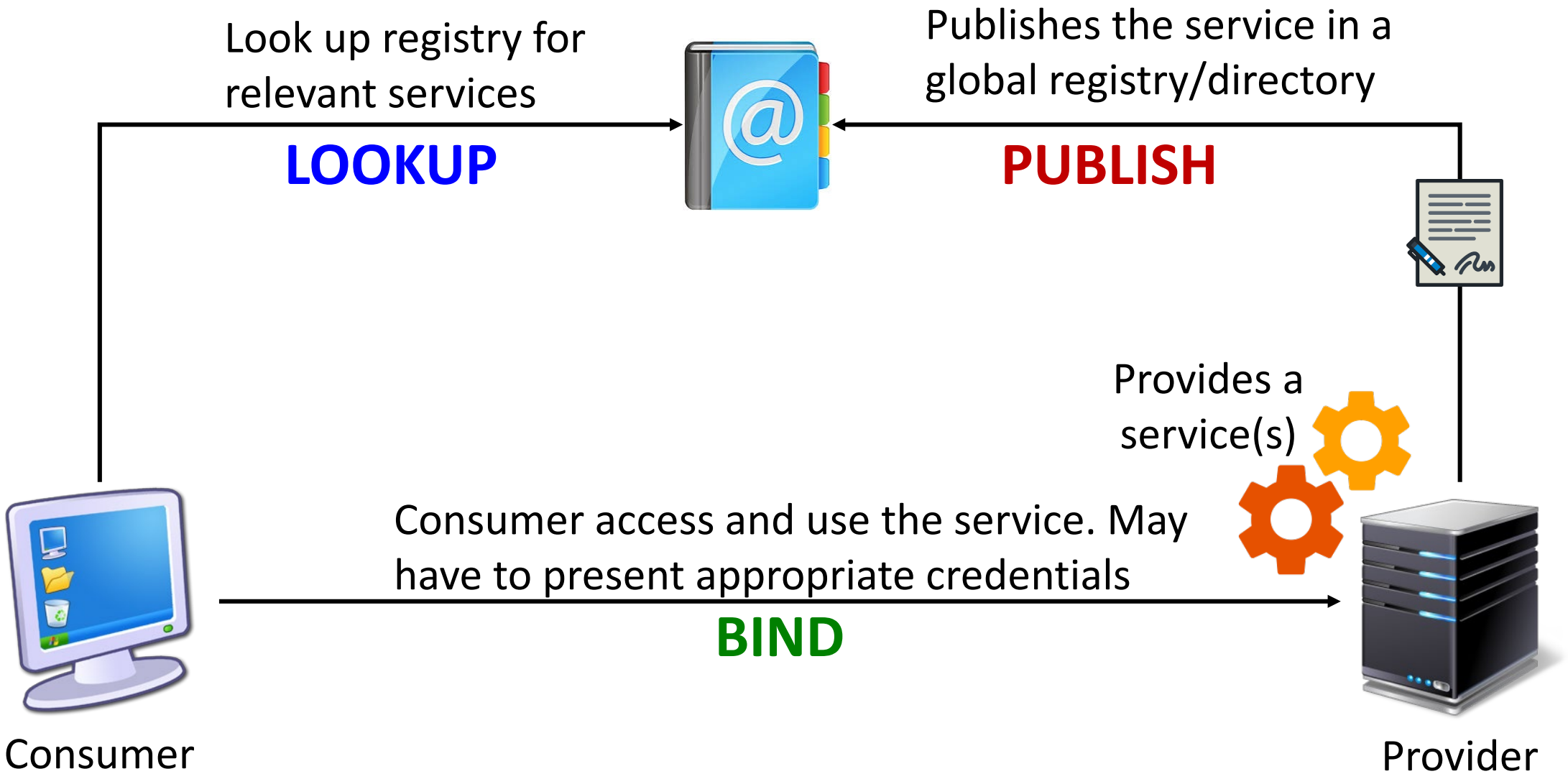
Pushing Notification from the Server



Notifications

Comet	Server Sent Events	Web Socket	Web Push
HTTP	HTTP	Web Socket	HTTP
Text payload, unlimited size	Text payload, unlimited size	Text or binary payload, unlimited size	Text or binary payload, 4K bytes limit
Request/response One request, one response	Publish and subscribe One request, multiple responses	Connection oriented Multiple request, multiple responses	Publish and subscribe Multiple responses
Uni-direction, simplex	Uni-direction, simplex	Bi-direction, duplex	Uni-direction, simplex
Datagram	Datagram	Datagram and stream	Datagram
Online	Online	Online	Online or offline

Service Discovery



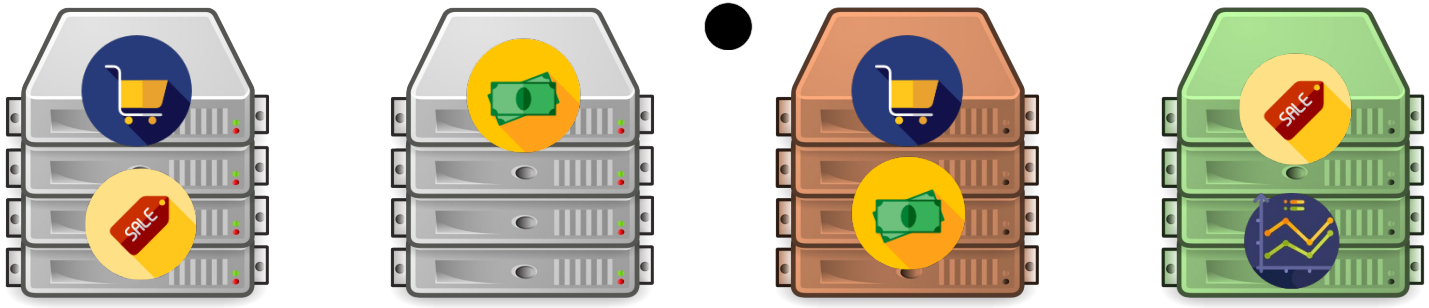
Why Do We Need Service Discovery?

Image from https://carnivalsatheart.com/wp-content/uploads/2014/07/IMG_2973.jpg

Service Discovery

Client needs to know where services are located and how to contact them

Multiple micro services are deployed across the cluster/data centre for resiliency, performance and scaling

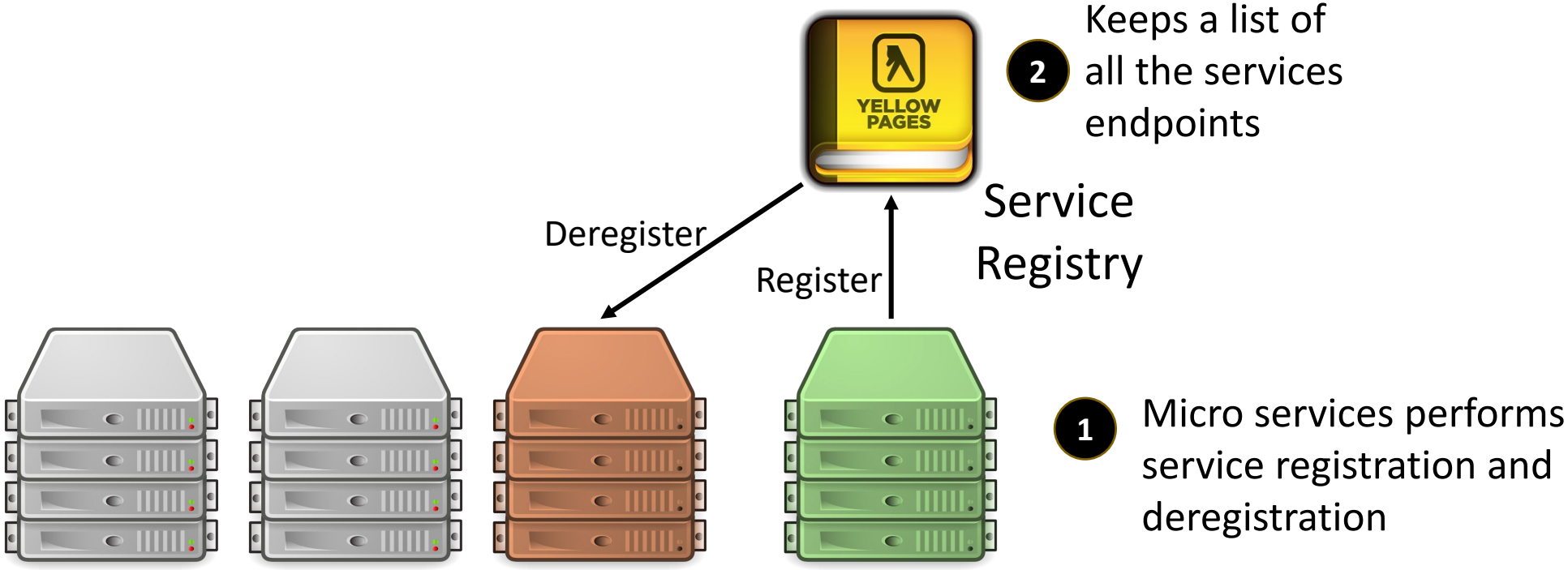


Service Discovery

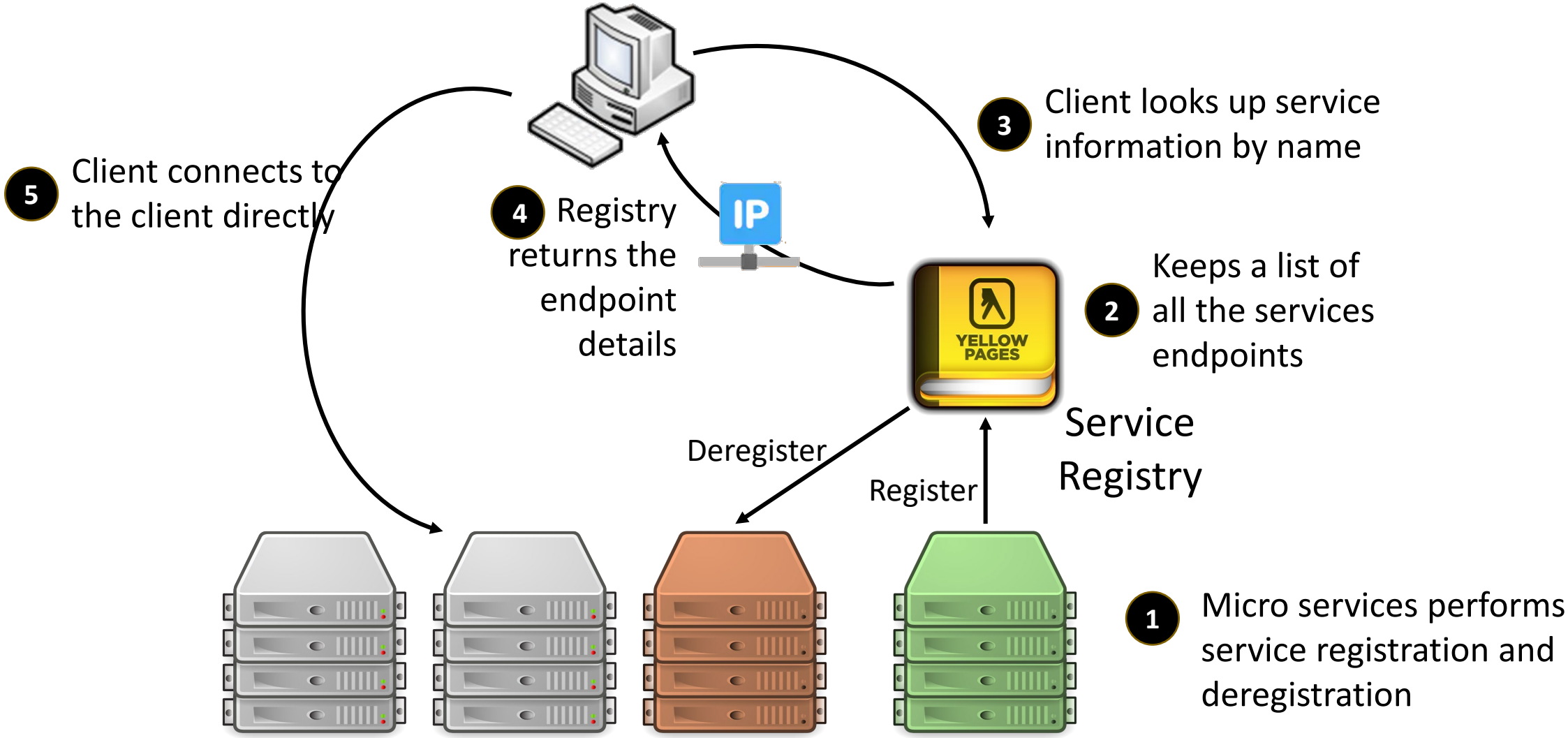
- Services are ephemeral
  - Need a way for clients to located services dynamically
  - Without hardcoding the IP address and port number
- Hide the topology of the service
  - How the services are organized
- Returns an instances that meets certain criteria to the client
  - Build in load balancer
  - Eg. the instance with the lightest load or geographically closest

- Provide health checks on the instances to ensure that they are healthy
  - Deregister the unhealthy instances automatically
- Failure detection
  - Perform health check a service by periodically invoke a given URL eg. `GET /health`, or
  - Service periodically sends heartbeat to the gateway

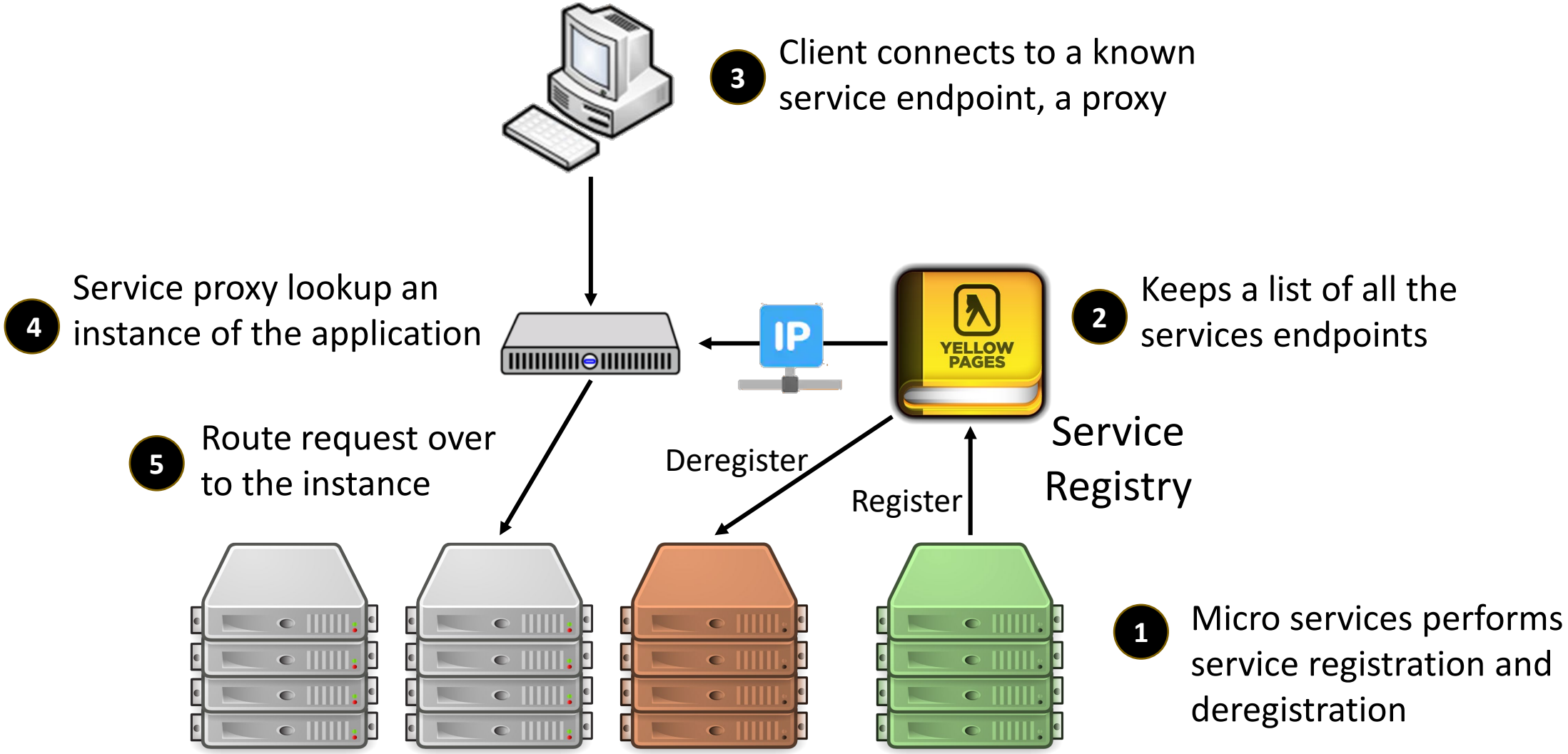
What is Service Discovery?



Client Side Service Discovery



Server Side Service Discovery



Service Discovery

Client Side

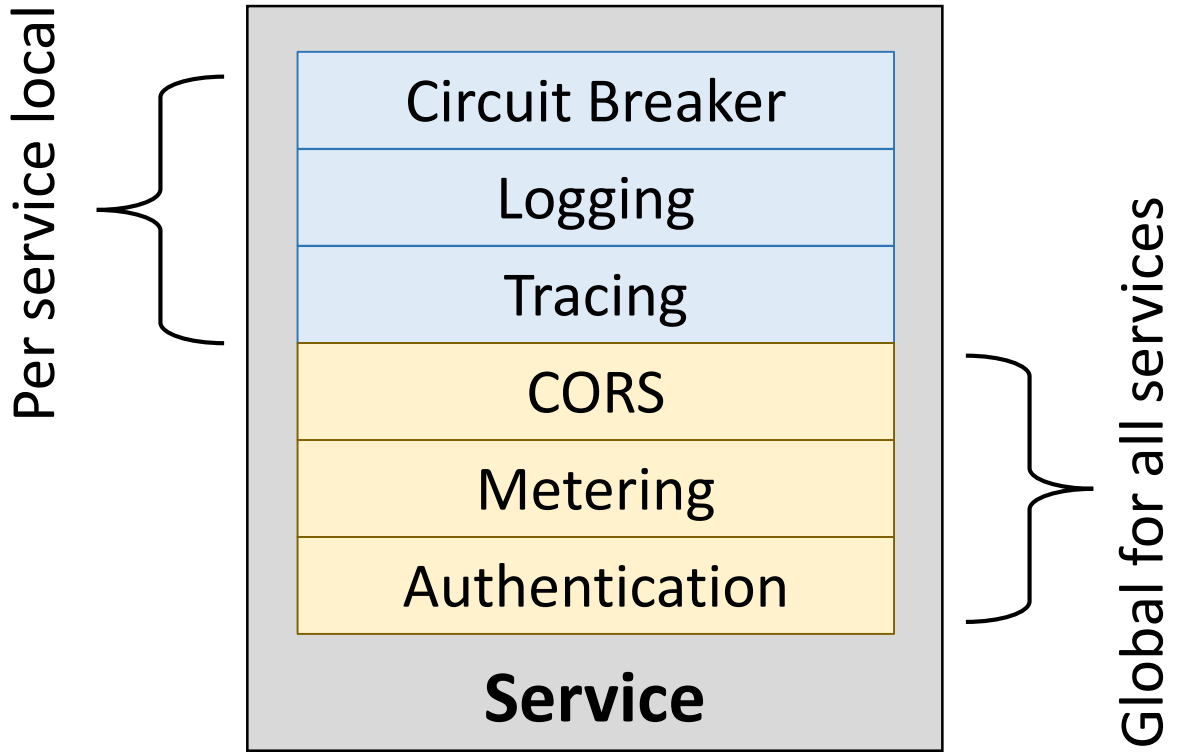
- Client gets a different endpoints
  - Client knows about the network
- Secure and harden each node
- No single point of failure
  - Ignoring the service registry
- Generally only function as a service registry
- Example: Consul

Server Side

- One fixed endpoint
  - Network topology is not exposed to the client
- Only have to harden the proxy
- Proxy is the single point of failure
- Proxy can be very sophisticated
  - API Gateway
- Example: Kong, Nginx, Traefik, Express-gateway

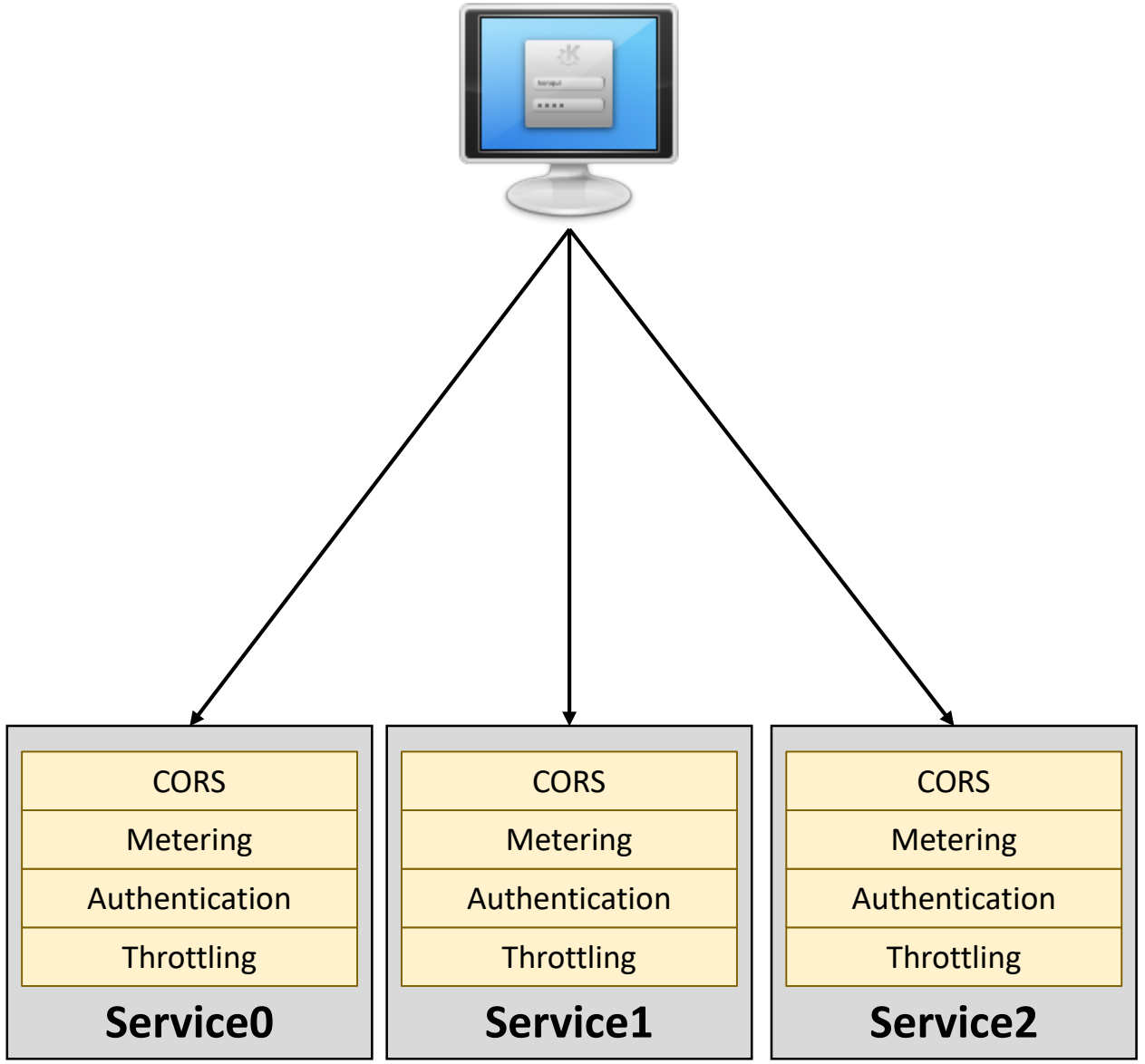
Typical Application Concerns

- Common features across all services
  - Fault management
  - Logging
  - Request tracing
  - Authentication
- Should be same across all services and not implemented by individual service
- Transparent to the service



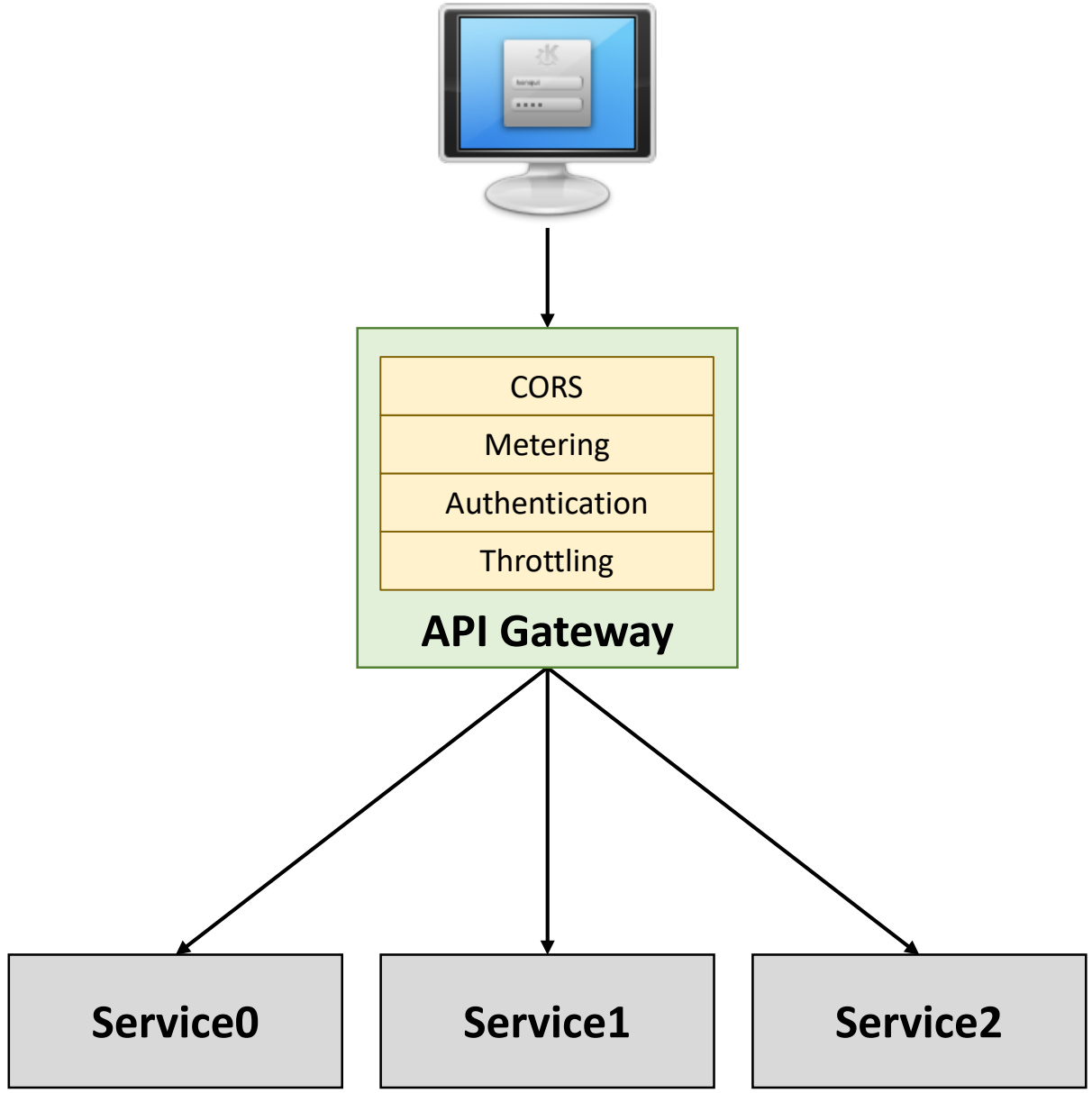
Service Challenges

- Enterprise level concerns replicated across the services
  - Eg. metering, authentication and authorization, CORS, throttling, fault injection, etc.
- Domain wide routing of traffic
  - Different versions, A/B testing, canary deployment, service failure, etc
- Service management
  - Monitoring access, tracing, fault reporting, load balancing, etc.
- Others
  - Caching responses, aggregating responses, request transformation, etc.

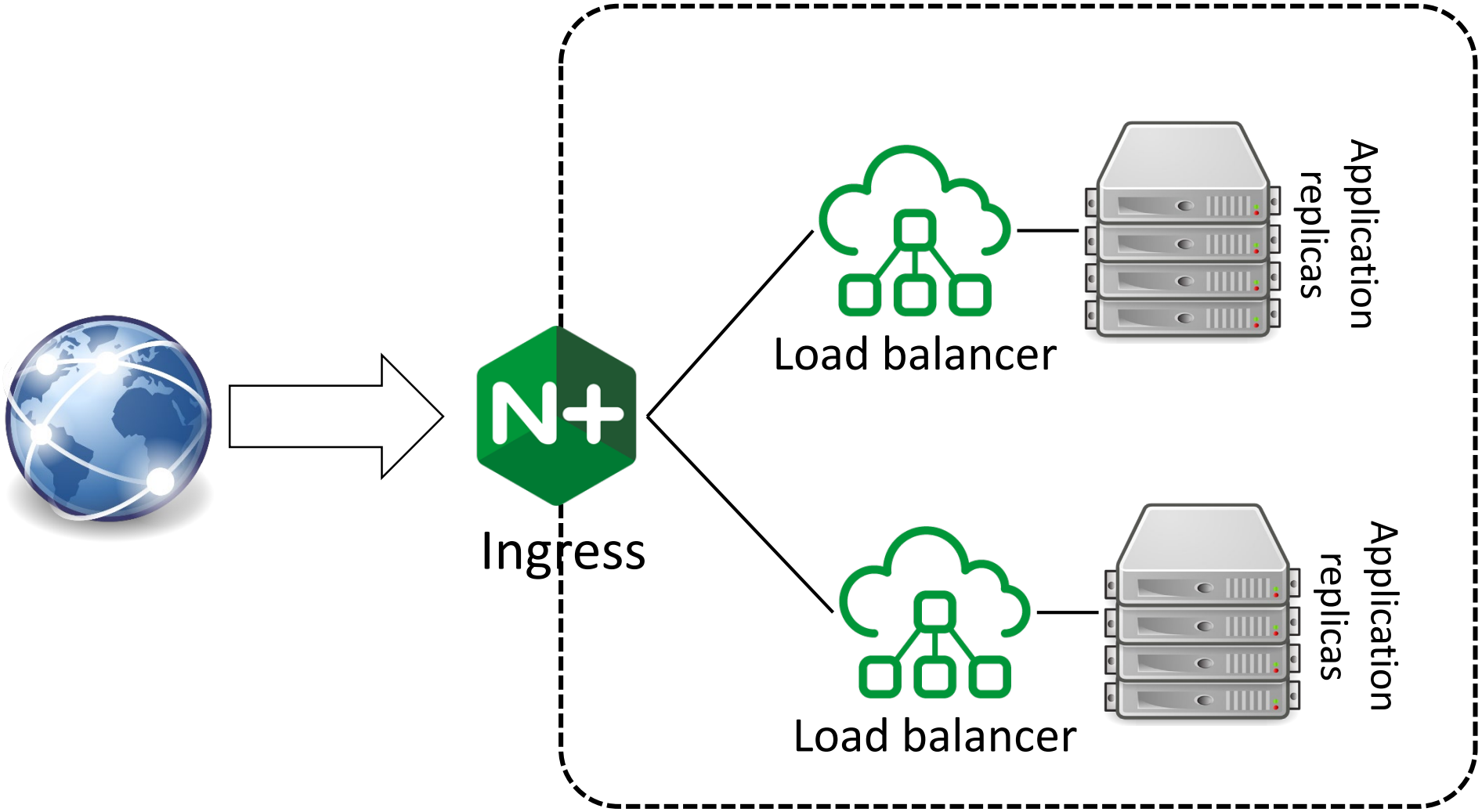


API Gateway

- Single point of entry into the service mesh for clients
  - North/south traffic
- Perform load balancing, weighted routing across services
  - East/west traffic
- Aggregate cross cutting concerns
- Management and monitoring



Gateway Example



Example of Nginx Configuration

```
http {
    upstream api {
        least_conn;
        server 192.168.0.10:3000;
        server 192.168.0.11:3000;
    }
    limit_req_zone $request_uri zone=apizone:20m rate=10r/s;
    server {
        server_name api.acme.com;
        limit_req zone=apizone;
        if ($request_method ~* "(GET|POST)" {
            add_header "Access-Control-Allow-Origin" *;
        }
        location / {
            proxy_pass https://api;
        }
    }
}
```

Upstream server block routed based on load

Limit the request rate by URI to 10/sec

Add CORS header to response. No preflight support

Forwarding all request to `api` upstream

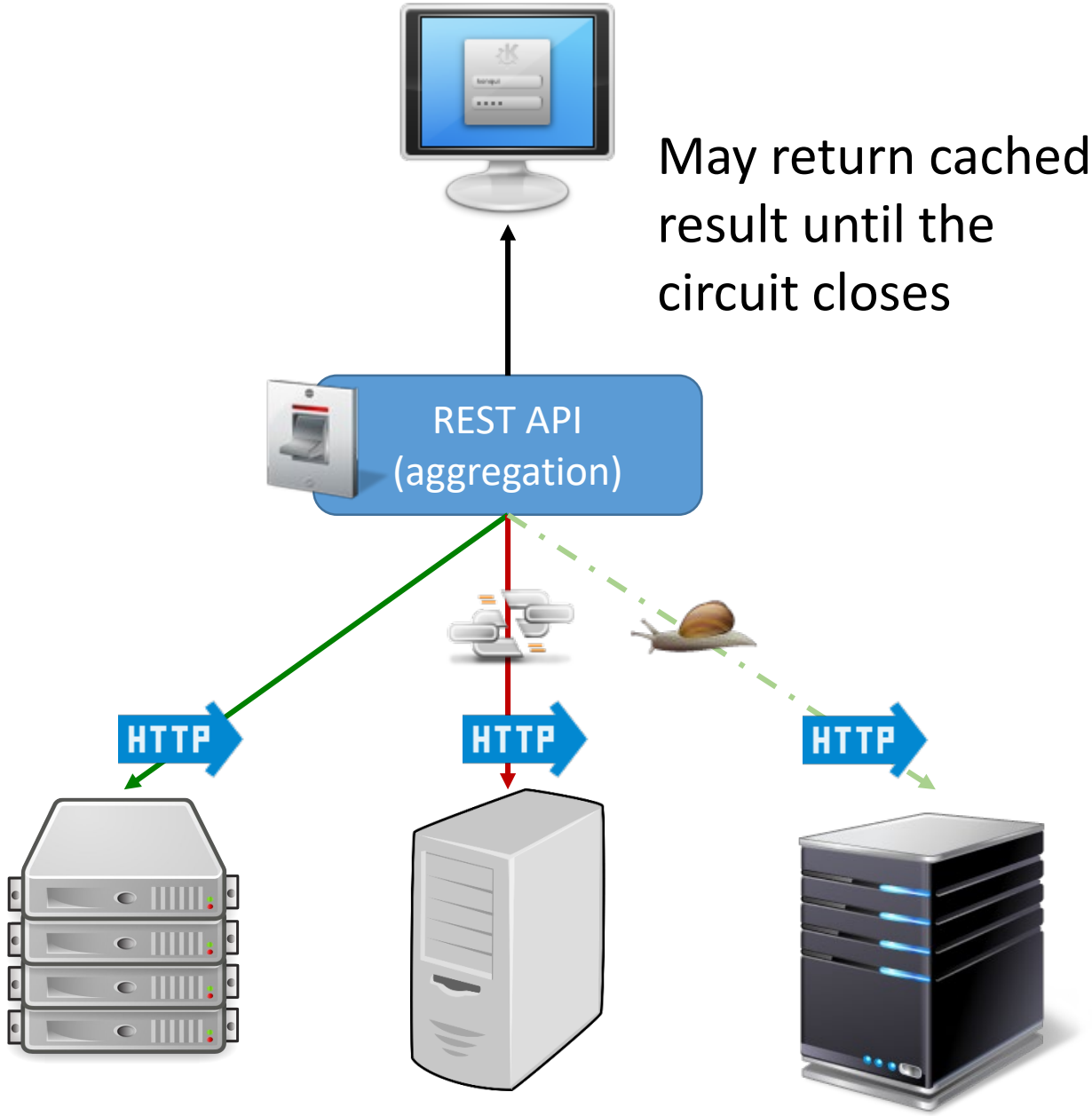
Appendix

Microservice Support Infrastructure

- Requires management of the services, example
  - Request tracing
  - Handling failures
  - Service discovery
  - Mutual authentication between services
- Otherwise lots of functionalities will have to be build into the micorservice
- Lots of libraries and platforms to support deployment of microservices

Circuit Breaker

- Design pattern for detecting error
  - Provide a default action if the error reoccurs
- Microservices - prevent continually invoking the failed service
  - Do not allocate resources to the invocation
  - Faster response back to the caller instead of waiting for a timeout



Circuit Breaker

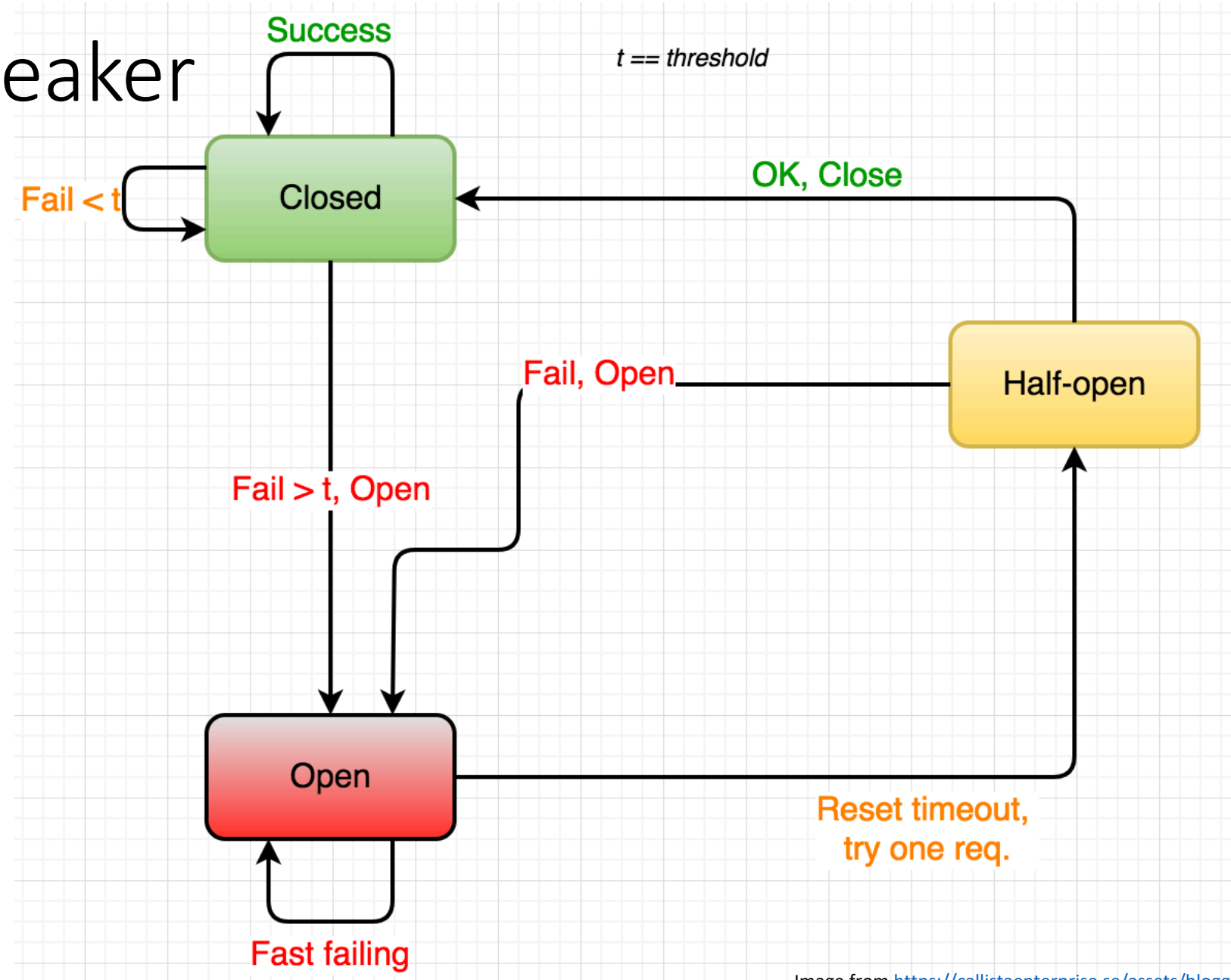


Image from https://callistaenterprise.se/assets/blogg/goblog/part11-cb-std.png

Example Circuit Breaker

Express

Configurations for the circuit breaker

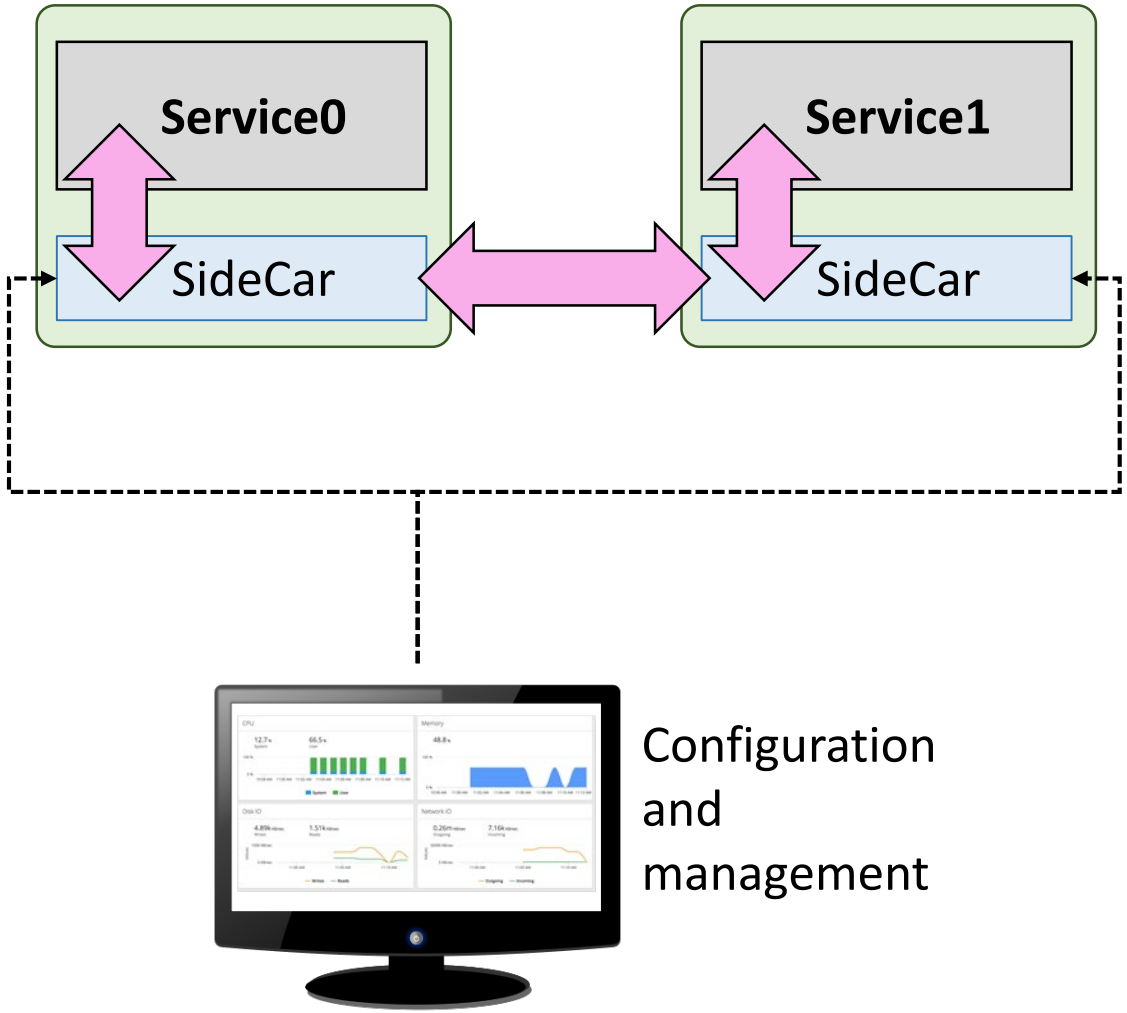
```
const circuitBreaker = require('opossum');
const request = require('request-promise');
const opt = {
   timeout: 5000, errorThresholdPercentage: 50, resetTimeout: 30000 };

const getWeather = circuitBreaker(
   function(name) {
      return request.get('http://api.openweathermap.org/...')
   }, opt);

getWeather.fallback(function() { resp.status(503).end(); })

app.get('/weather/:city', (req, resp) => {

   getWeather.fire(req.params.city)
      .then(result => resp.status(200).json(result))
      .catch(error => resp.status(400).json(error));
})
```

Wrap remote call in a circuit breaker

Provide a function to handle open circuits

'Normal' result

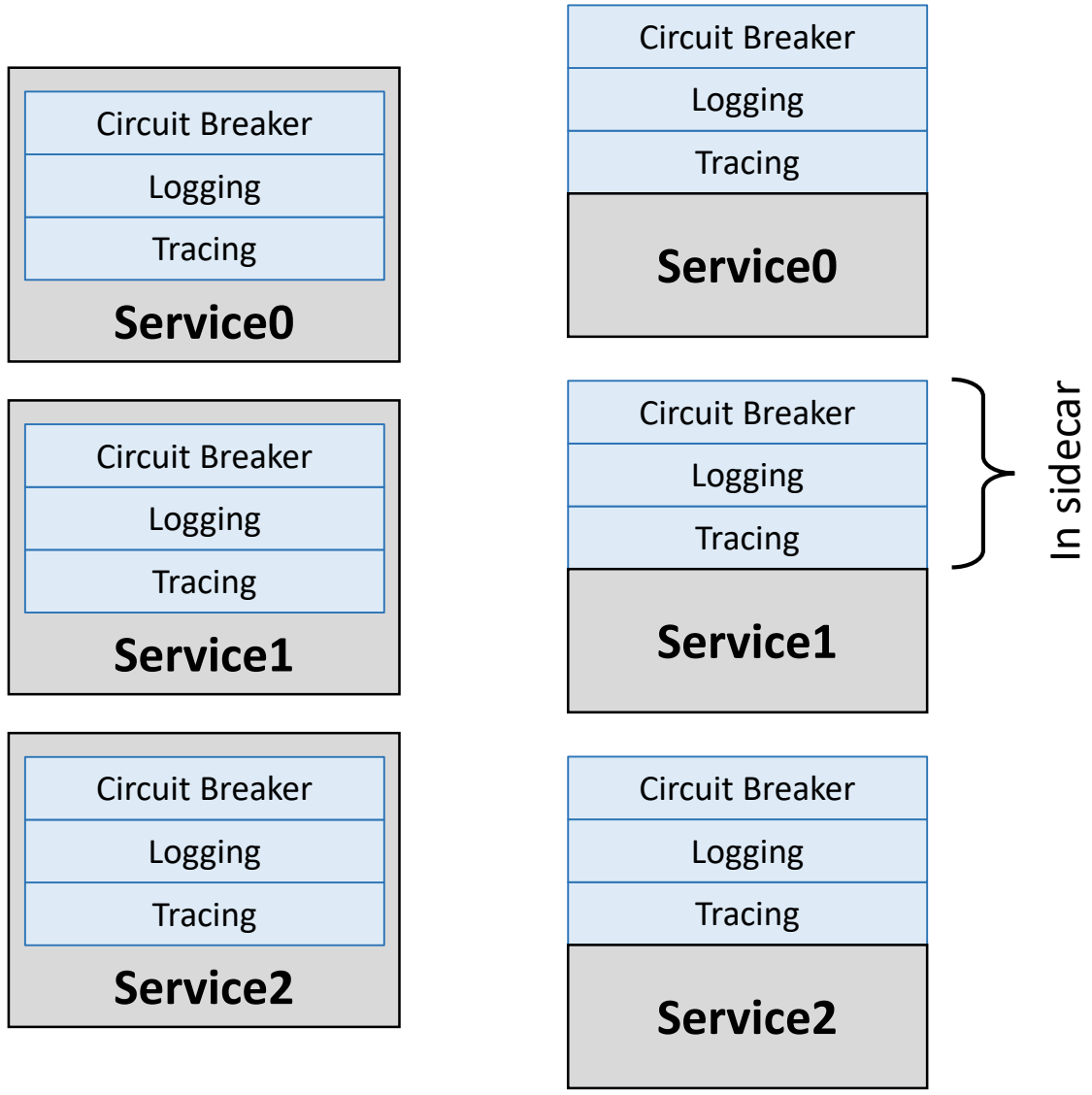
Service SideCar



- Abstract common services so that they can be shared
- All traffic entering and exiting a service must be proxied through the side car
- Perform these common services on request entering and exiting a service
- Managed and configured centrally
- Different strategies depending on how services are deployed
  - Consul - separate process
  - Istio - containers

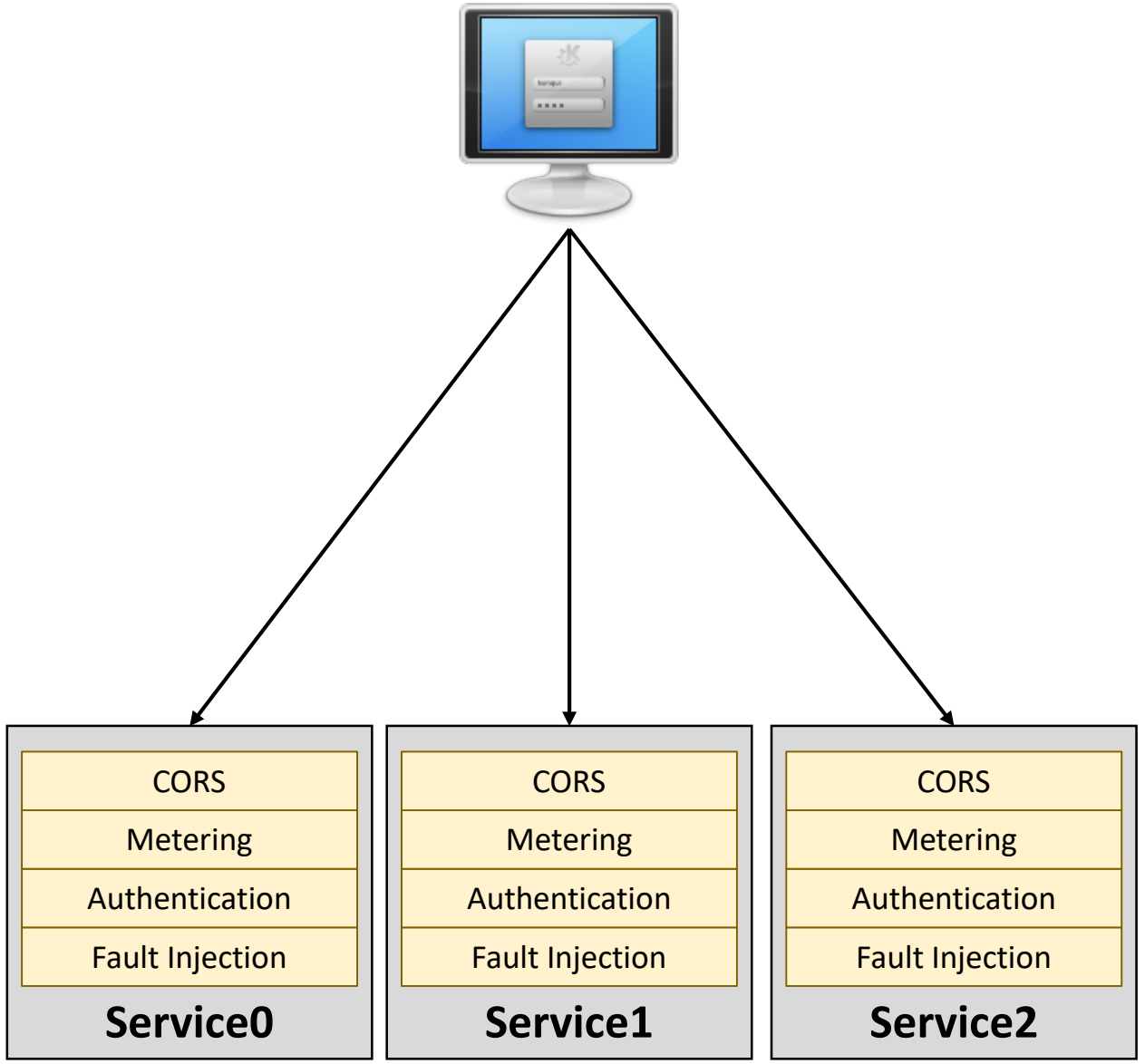
SideCar Pattern

- Common related functionalities
  - Circuit breaking
  - Logging to a central log
  - Open tracing for request leaving the service
- Should be same across all services and not implemented by individual service
  - Transparent to the service
- Externalize these service features
  - To be shared across all services



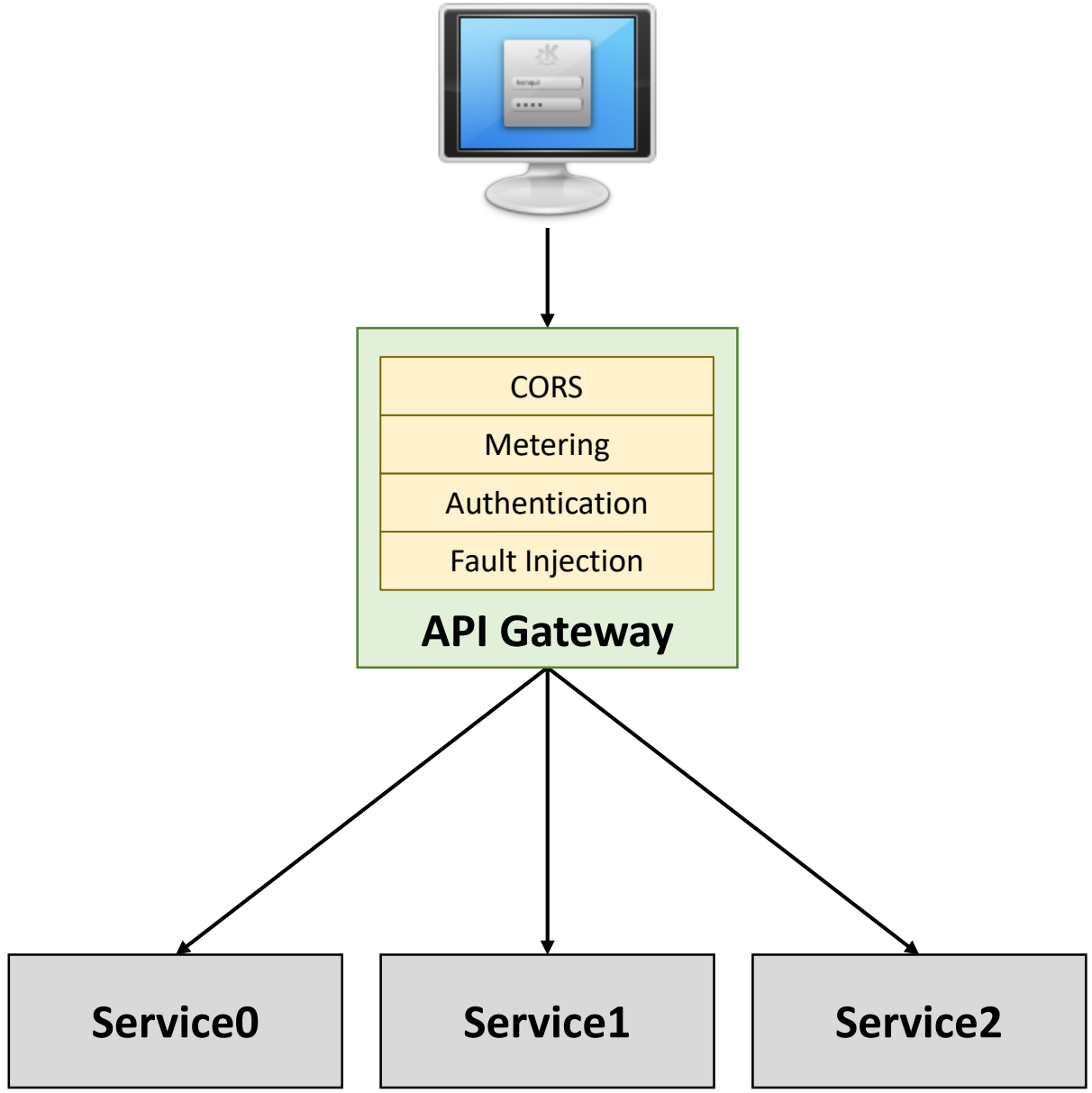
Service Challenges

- Enterprise level concerns replicated across the services
  - Eg. metering, authentication and authorization, CORS, fault injection, etc.
- Domain wide routing of traffic
  - Different versions, A/B testing, canary deployment, service failure, etc
- Service management
  - Monitoring access, tracing, fault reporting, load balancing, etc.
- Others
  - Caching responses, aggregating responses, request transformation, etc.



API Gateway

- Single point of entry into the service mesh for clients
  - North/south traffic
- Perform load balancing, weighted routing across services
  - East/west traffic
- Aggregate cross cutting concerns
- Management and monitoring



Throttling the Request

- Limit the rate of calls made to your REST API
  - Especially public facing ones
  - Prevent abuse
- Returns 429 status
  - Client is making too many calls
  - Reach the number of invocation
- Gateway will insert HTTP headers to provide hints to the client
  - `X-RateLimit-Limit` – requests per time window, per provider
  - `X-RateLimit-Remaining` – number of request left for the time window
  - `After-Retry` – retry again after the given time and date

Scaling Out

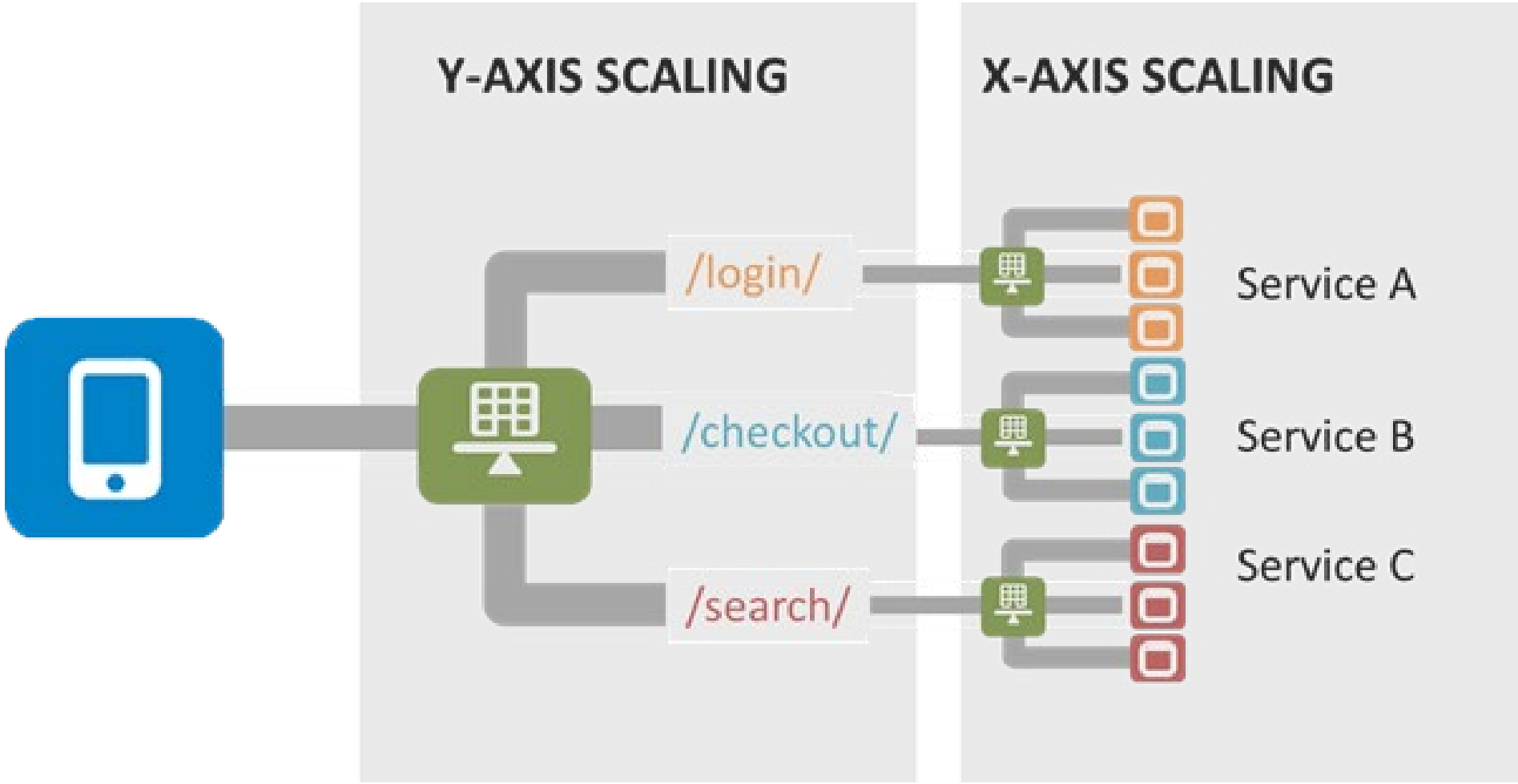


Image from https://devcentral.f5.com/Portals/0/Users/038/38/38/x_and_y_axis_scaling_pattern_thumb.png

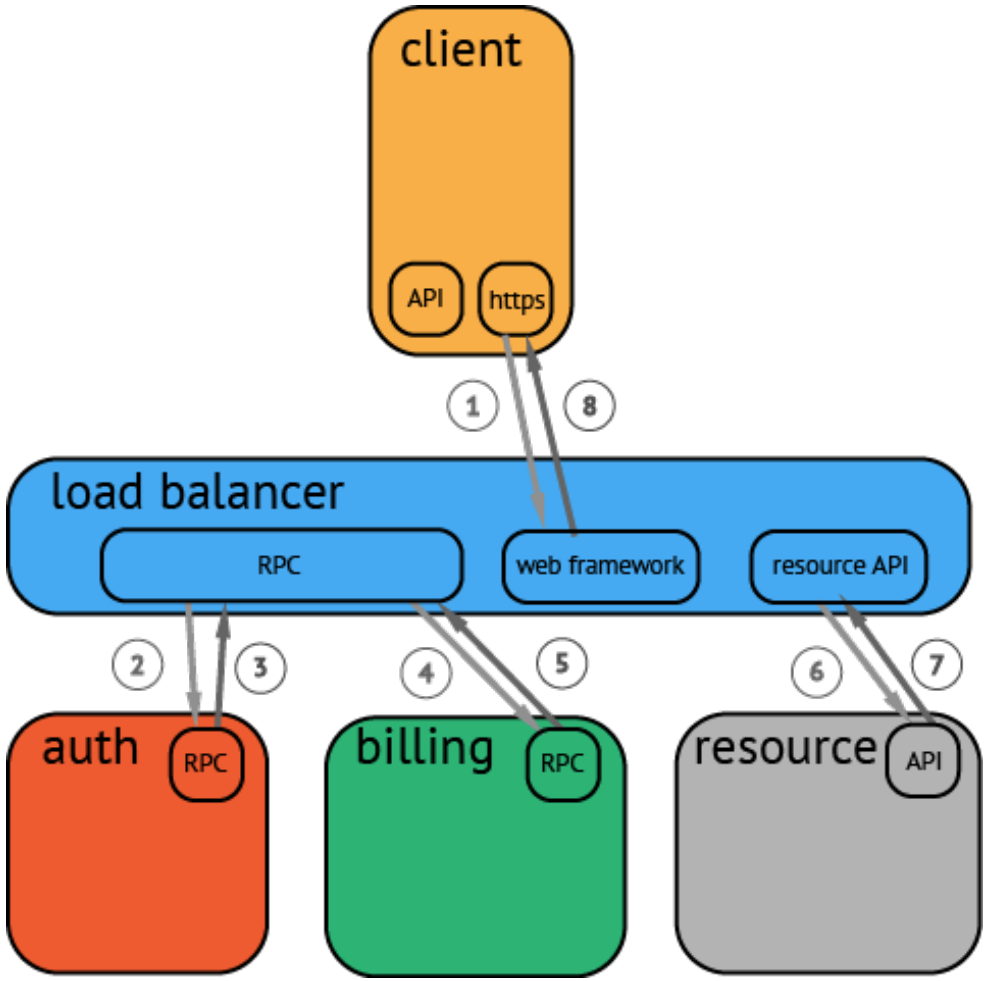
API Gateway

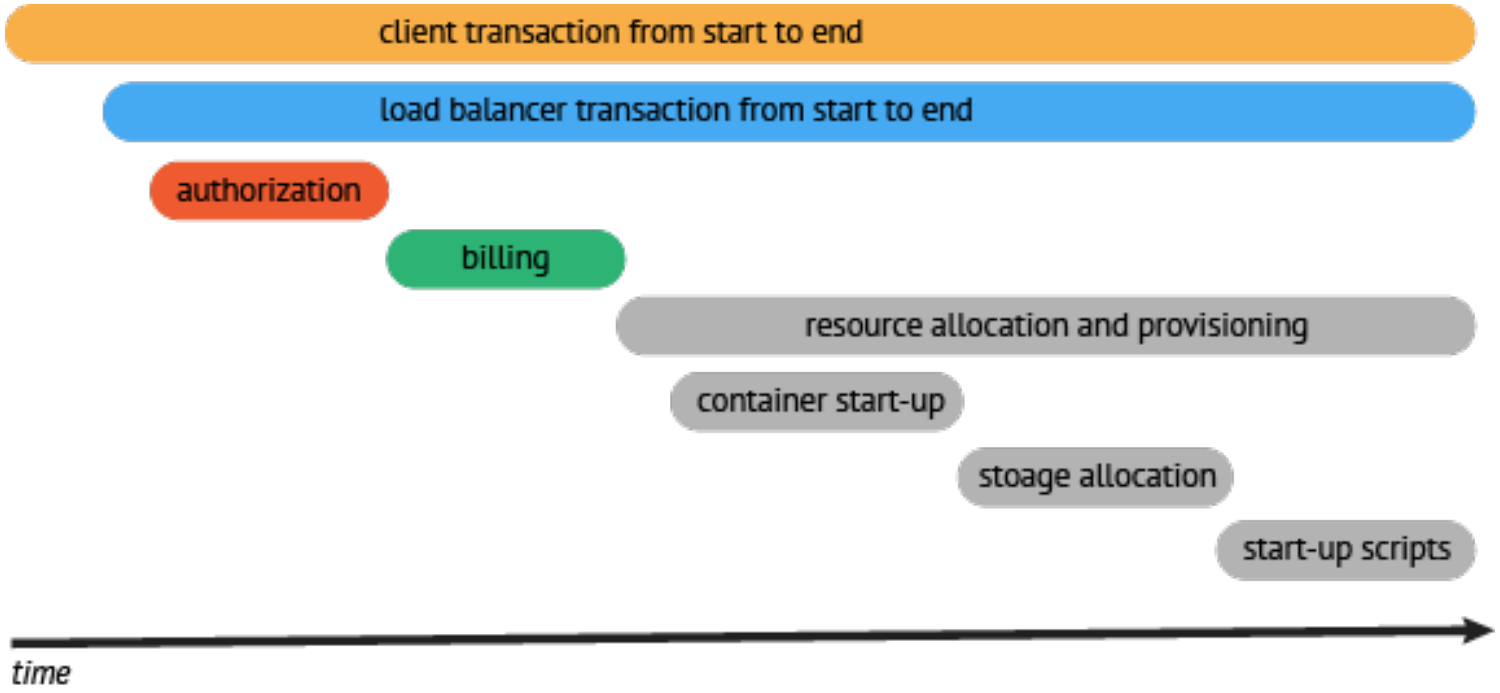
- Single point of failure
- Ingest traffic to other services including non HTTP based services
- Lots of other features especially from cloud providers
  - Request/response transformation
  - Security integration
  - Invoke serverless functions
  - Metering and rate limiting
  - Criteria based routing eg. HTTP header, host, resource names

Tracing

- Add tracing to microservices
- Tracing starts when a request is made and does not stop until a response is returned
  - Implemented at API gateway and at sidecars
- Observe how a request is serviced
  - What are the microservices that are servicing the request
  - What is the time spend in each of these service
  - If an error occurs, where does it occur
- OpenTracing is a standard for distributed tracing

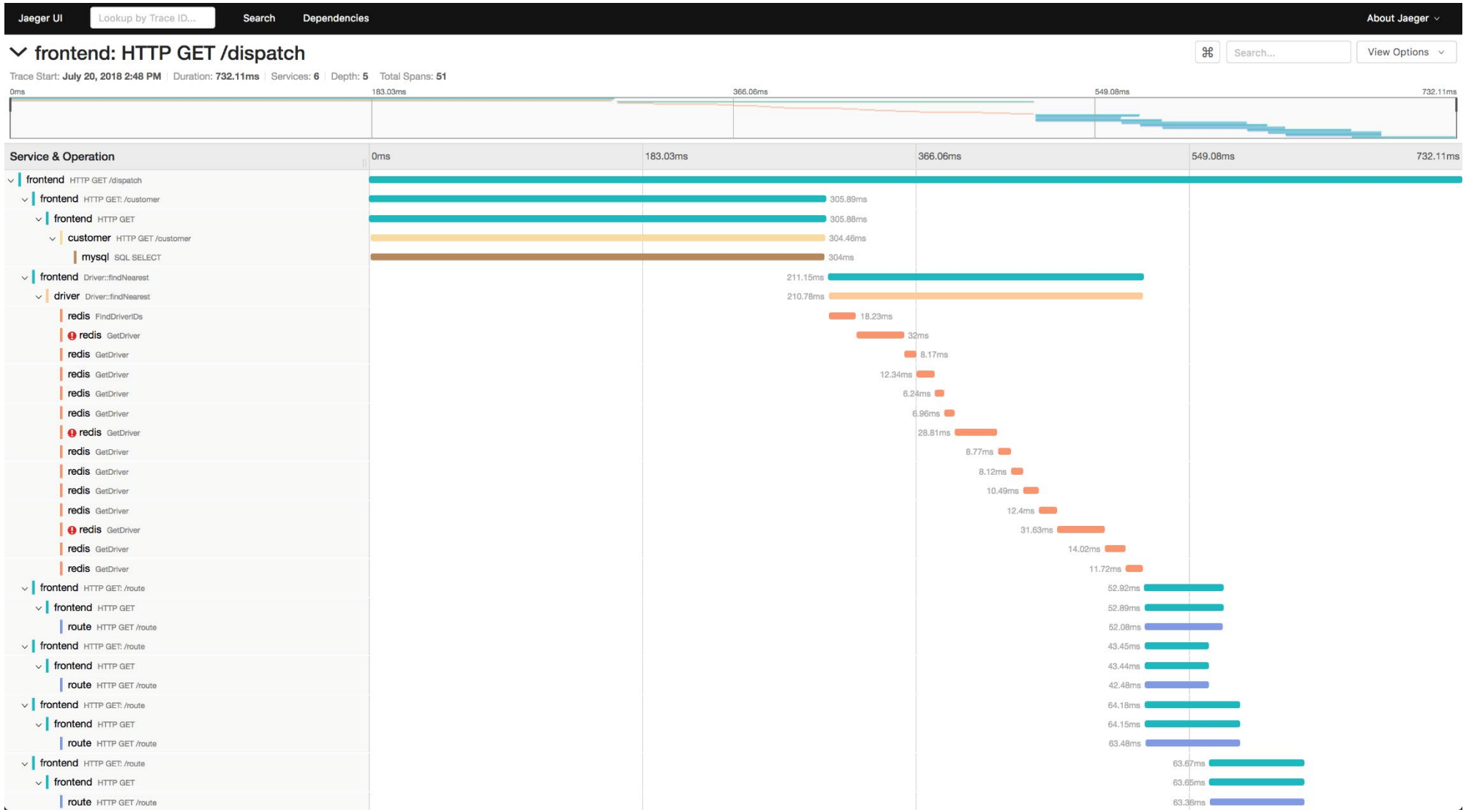
OpenTracing Illustrated





Images from https://opentracing.io/docs/best-practices/instrumenting-your-application/

Example A Trace from Jaeger



List of Modules

- mysql - https://www.npmjs.com/package/mysql
- Angular JWT Module - https://www.npmjs.com/package/@auth0/angular-jwt
- JSONWebToken - https://github.com/auth0/node-jsonwebtoken
- express-longpoll - https://www.npmjs.com/package/express-longpoll
- express-sse-middleware - https://github.com/taqm/express-sse-middleware
- express-ws - https://www.npmjs.com/package/express-ws
- opossum - https://github.com/nodeshift/opossum
- consul - https://github.com/silas/node-consul

Service Mesh

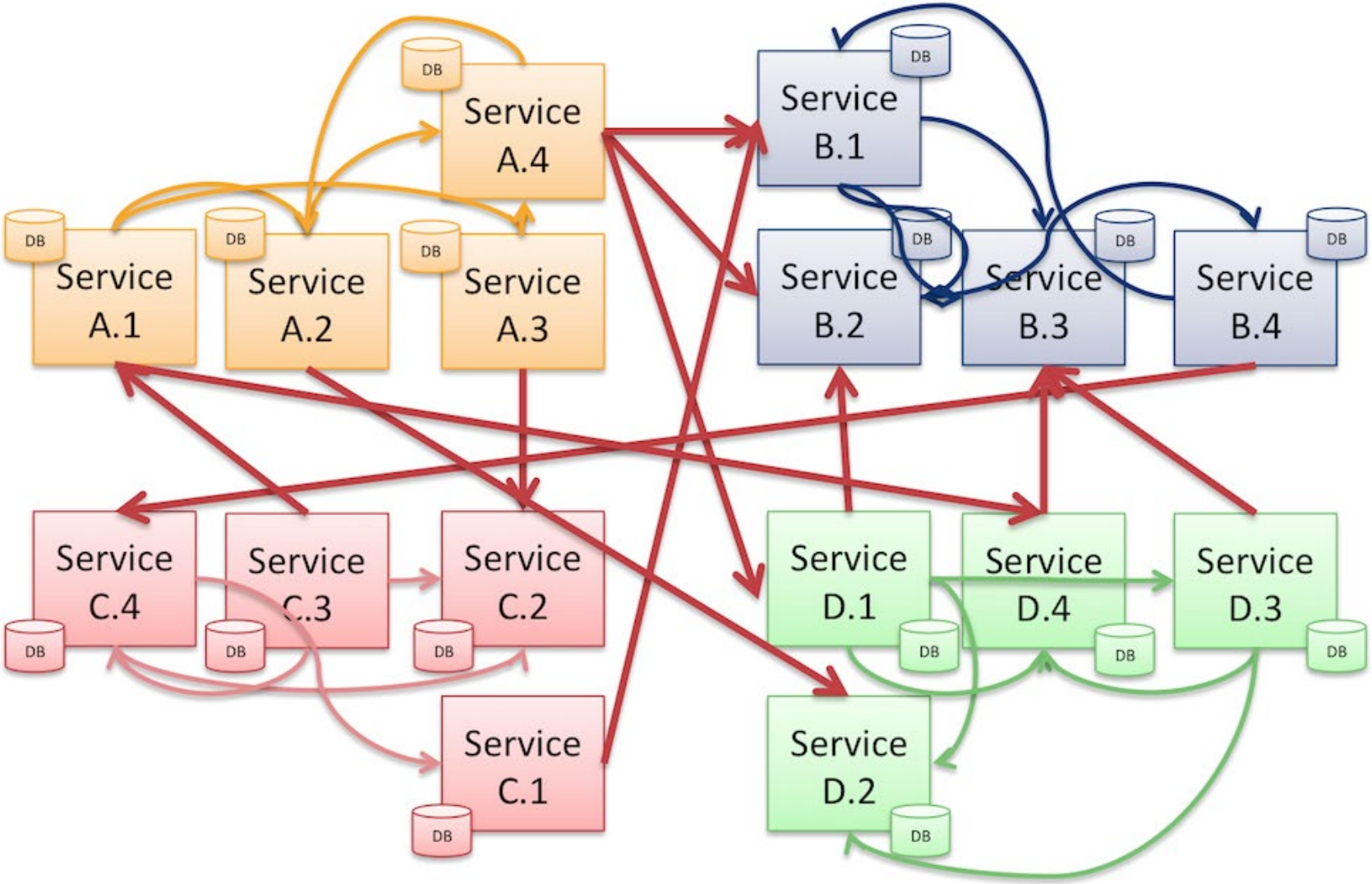


Image from https://www.tigerteam.dk/2014/microservices-its-not-only-the-size-that-matters-its-also-how-you-use-them-part-4/

Example Service Registration

```
const ID = require('uuid').v4();

const consul = require('consul')({ promisify: true });
const app = express();
...

app.listen(PORT, () => {
  consul.agent.service.register({
      name: 'myservice', port: PORT, id: ID,
      check: { ttl: '10s', deregistercriticalserviceafter: '60s' }
  }).then(() => {
      (function a() {
         setTimeout(() => {
            consul.agent.check.pass({ id: `service:${ID}` })
               .then(() => a())
               .catch(error => { ... });
         }, 10 * 1000)
      })();
  })
})
```

Register the service

Service name is registered as `myservice.service.consul` in Consul's internal DNS

Service details

Note the service identification format

Heartbeat not notify Consul that the service is healthy

Send heartbeat every 10 seconds

Alternative to heartbeat is for Consul to poll the service via a http GET

Example Deregistration

Express

Signal generated from CTRL-C or when a container is stopped

Deregister service with Consul when service is shutting down

```
process.on('SIGINT', () => {
   consul.agent.service.deregister({ id: ID })
      .then(() => process.exit());
});
```

If you do not deregister, the service will be listed in Consul until TTL runs out or Consul determines that the service is no longer healthy

Example Service Lookup - DNS

```
dig @localhost -p 8600 myservice.service.consul

;; QUESTION SECTION:
;myservice.service.consul.          IN   A

;; ANSWER SECTION:
myservice.service.consul. 0    IN   A    127.0.0.1


dig @localhost -p 8600 myservice.service.consul SRV

;; QUESTION SECTION:
;web.service.consul. IN SRV

;; ANSWER SECTION: myservice.service.consul. 0 IN SRV 1 1 3000 pc.node.dc1.consul.
;; ADDITIONAL SECTION: pc.node.dc1.consul. 0 IN A 127.0.0.1
```

Example Service Lookup - JavaScript

```
const consul = require('consul'){{ promisify: true });

consul.catalog.node.list()
   .then(result => result[0])
   .then(node => Promise.all([
      node.Address, consul.agent.service.list(node.Node)]))
   .then(result => {
      const ip = result[0]
      const svc = result[1]
      for (let i in svc)
         console.info(`service: ${svc[i].Service}, ip: ${ip}, port: ${svc[i].Port}`)
   })
   .catch(error => {
      console.error('error: ', error);
   })
```

List all the available nodes

Select a node

List the service from the selected node

Node's IP address

The service's name, IP address and port number