E6156 – Topics in SW Engineering: Cloud Computing

Lecture 6: Middleare, APIGW, Lambda, SSO, Events

Go To CourseWorks – Attendance Exam

- X12983P

Contents

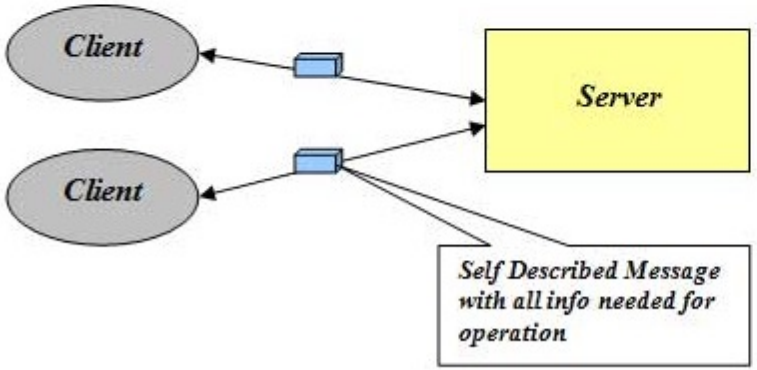
Agenda

- Stateless, and sessions.
  - Reprising “stateless”
  - Session and other examples
- Middleware
  - Concepts: APIs, handlers
  - Three examples: security, notification, logging
- Event Driven Processing
  - Concepts and motivation
  - Applied to your projects: email verification, webhooks
- Function-as-a-Service, AWS Lambda functions
- HW2 – Sprint 2 Objectives

Stateless, Sessions Example

Stateless

The notion of statelessness is defined from the perspective of the server. The constraint says that the server should not remember the state of the application. As a consequence, the client should send all information necessary for execution along with each request, because the server cannot reuse information from previous requests as it didn't memorize them. All info needed is in message.



Statelessness

2. Application State vs Resource State

It is important to understand the between the application state and the resource state. Both are completely different things.

Application state is server-side data that servers store to identify incoming client requests, their previous interaction details, and current context information.

Resource state is the current state of a resource on a server at any point in time – and it has nothing to do with the interaction between client and server. It is what we get as a response from the server as the API response. We refer to it as resource representation.

> REST statelessness means being free from the application state.

3. Advantages of Stateless APIs

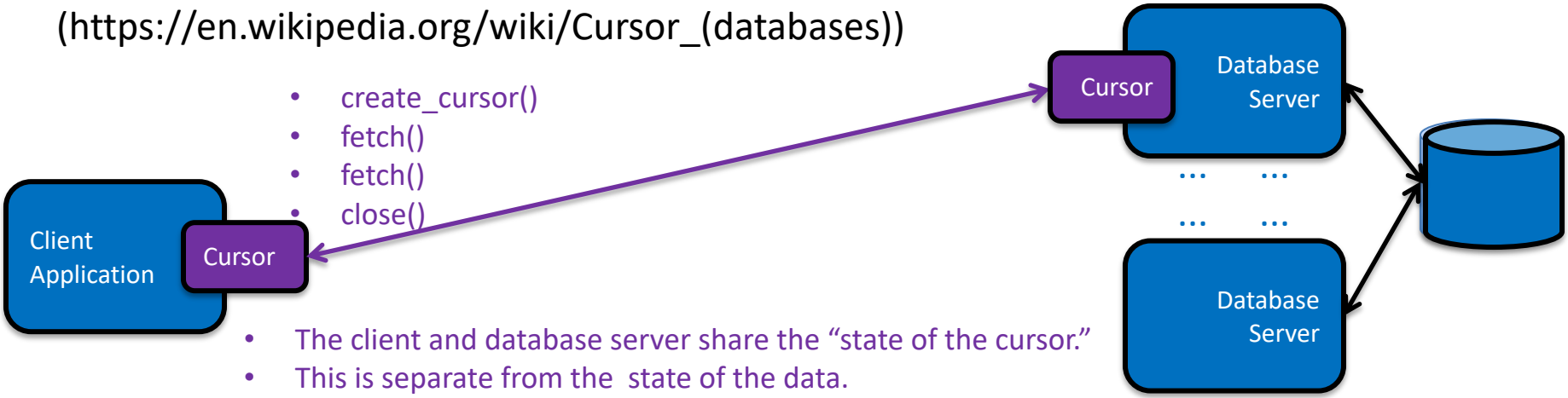
There are some very noticeable advantages of having **REST APIs stateless**.

1. Statelessness helps in **scaling the APIs to millions of concurrent users** by deploying it to multiple servers. Any server can handle any request because there is no session related dependency.
2. Being stateless makes REST APIs **less complex** – by removing all server-side state synchronization logic.
3. A stateless API is also **easy to cache** as well. Specific softwares can decide whether or not to cache the result of an HTTP request just by looking at that one request. There's no nagging uncertainty that state from a previous request might affect the cacheability of this one. It **improves the performance** of applications.
4. The server never loses track of "where" each client is in the application because the client sends all necessary information with each request.

- This concept can be confusing with experience with other approach to session state.
- Some application frameworks did/do keep session state on the server.
- Application servers and frameworks have various support for "session state," e.g. https://flask-session.readthedocs.io/en/latest/

Simple Example from Databases

- "In computer science, a database cursor is a mechanism that enables traversal over the records in a database. Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records. The database cursor characteristic of traversal makes cursors akin to the programming language concept of iterator."
(https://en.wikipedia.org/wiki/Cursor_(databases))



- The client and database server share the "state of the cursor."
- This is separate from the state of the data.
- This hinders scalability and fault tolerance if there are multiple database servers in a cluster.

Some Code

```
def t_cursor():

    print("\nUsing cursors ...")
    sql = "select * from users";

    for i in range(0,5):
        if i == 0:
            res = with_cursor(done=False, sql=sql)
        elif i == 4:
            res = with_cursor(done=True)
        else:
            res = with_cursor()

        print("Res = ", res)
```

```
def t_cursor():

    print("\nUsing cursors ...")
    sql = "select * from users";

    for i in range(0,5):
        if i == 0:
            res = with_cursor(done=False, sql=sql)
        elif i == 4:
            res = with_cursor(done=True)
        else:
            res = with_cursor()

        print("Res = ", res)
```

```
def t_without_cursor():

    print("\nNot using cursors.")

    sql = "select * from users";

    for i in range(0,5):
        res = without_cursor(sql=sql, offset=i)
        print("Res = ", res)
```

```
def without_cursor(sql, offset):

    cursor = conn.cursor()
    tmp_sql = sql + " limit 1" + " offset " + str(offset)
    res = cursor.execute(tmp_sql)
    res = cursor.fetchone()
    cursor.close()
    return res
```

An Execution

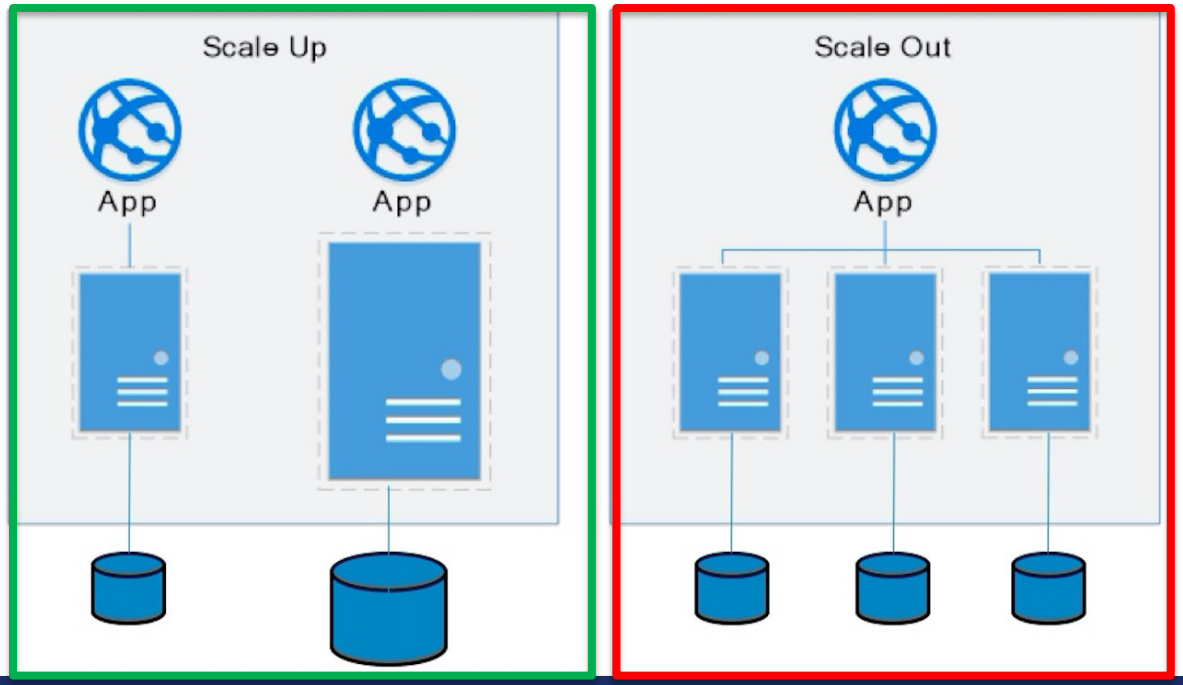
```
Using cursors ...


Creating a cursor!


Res =  {'id': 1, 'first_name': 'Hall', 'last_name': 'Dellatorre', 'email': 'hdellatorre0@xing.com', 'address_id': 510}
Res =  {'id': 2, 'first_name': 'Bradan', 'last_name': 'Iredell', 'email': 'biredell1@answers.com', 'address_id': 502}
Res =  {'id': 3, 'first_name': 'Cassy', 'last_name': 'Tolotti', 'email': 'ctolotti2@ted.com', 'address_id': 838}
Res =  {'id': 4, 'first_name': 'Lilla', 'last_name': 'Indruch', 'email': 'lindruch3@columbia.edu', 'address_id': 10}


Closing a cursor!


Res =  None

Not using cursors.
Res =  {'id': 1, 'first_name': 'Hall', 'last_name': 'Dellatorre', 'email': 'hdellatorre0@xing.com', 'address_id': 510}
Res =  {'id': 2, 'first_name': 'Bradan', 'last_name': 'Iredell', 'email': 'biredell1@answers.com', 'address_id': 502}
Res =  {'id': 3, 'first_name': 'Cassy', 'last_name': 'Tolotti', 'email': 'ctolotti2@ted.com', 'address_id': 838}
Res =  {'id': 4, 'first_name': 'Lilla', 'last_name': 'Indruch', 'email': 'lindruch3@columbia.edu', 'address_id': 10}
Res =  {'id': 5, 'first_name': 'Tiffany', 'last_name': 'Mecozzi', 'email': 'tmecozzi4@nifty.com', 'address_id': 643}
```

Approaches to Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system.

Replace system with a bigger machine,
e.g. more memory, CPU, … …

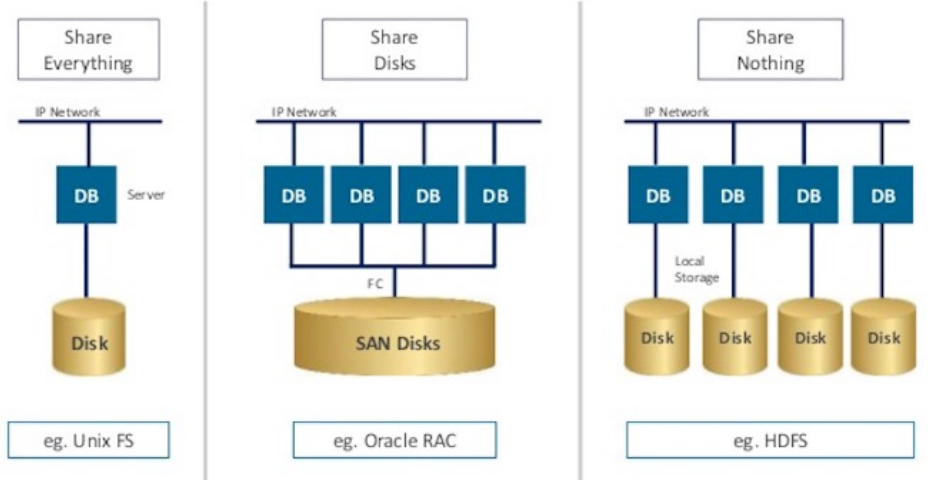
Add another system.



- Scale-up:
  - Less incremental.
  - More disruptive.
  - More expensive for extremely large systems.
  - Does not improve availability
- Scale-out:
  - Incremental cost.
  - Data replication enables availability.
  - Does not work well for functions like JOIN, referential integrity, … …

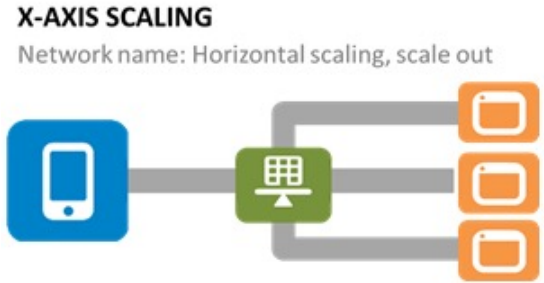
Disk Architecture for Scale-Out

- Share disks:
  - Is basically scale-up for data/disks. You can use NAS, SAN and RAID.
  - Isolation/Integrity requires distributed locking to control access from multiple database servers.
- Share nothing:
  - Is basically scale-out for disks.
  - Data is partitioned into *shards* based on a function f() applied to a key.
  - Can improve availability, at the code consistency, with data replication.
  - There is a router that sends requests to the proper shard based on the function.



Stateless

- There are similar design issue for microservice applications.
- There is conversation state, e.g.
  - Did the client create a shopping cart?
  - Did the client choose a preferred store?
  - etc.



HTTP sessions is an industry standard feature that allows Web servers to maintain user identity and to store user-specific data during multiple request/response interactions between a client application and a Web application. HTTP sessions preserves:

- Information about the session itself (session identifier, creation time, time last accessed, etc.)
- Contextual information about the user (client login state, for example, plus whatever else the Web application needs to save)

https://docs.progress.com/bundle/pas-for-openedge-administration-117/page/Overview-of-HTTP-sessions.html

- Prior technologies maintained the conversation on the server, e.g. HttpSession. (https://docs.progress.com/bundle/pas-for-openedge-administration-117/page/Overview-of-HTTP-sessions.html)
- Stateless with respect to conversations has much better scalability and availability.
- Show in demo (demo-flask). Mention Cookie is default but headers also work if the client application understand and uses the header. (Also, show header in Postman)

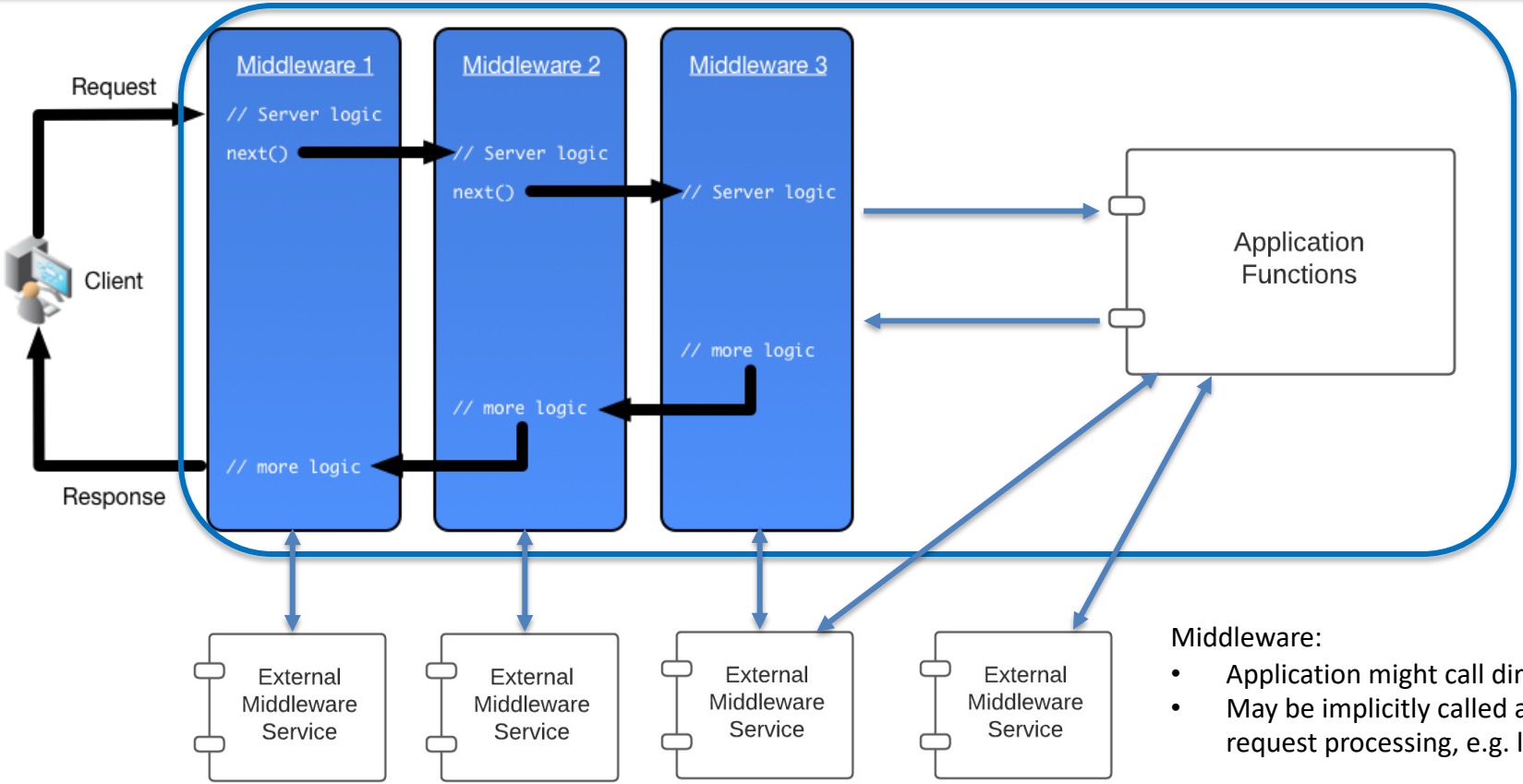
Middleware

What is middleware?

Middleware is software that lies between an operating system and the applications running on it. Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications. It's sometimes called plumbing, as it connects two applications together so data and databases can be easily passed between the "pipe." Using middleware allows users to perform such requests as submitting forms on a web browser, or allowing the web server to return dynamic web pages based on a user's profile.

Common middleware examples include database middleware, application server middleware, message-oriented middleware, web middleware, and transaction-processing monitors. Each program typically provides messaging services so that different applications can communicate using messaging frameworks like simple object access protocol (SOAP), web services, representational state transfer (REST), and JavaScript object notation (JSON). While all middleware performs communication functions, the type a company chooses to use will depend on what service is being used and what type of information needs to be communicated. This can include security authentication, transaction management, message queues, applications servers, web servers, and directories. Middleware can also be used for distributed processing with actions occurring in real time rather than sending data back and forth.

Middleware – Conceptual Model



Middleware:

- Application might call directly, e.g. DB
- May be implicitly called as part of request processing, e.g. logging.

Middleware and Flask

```
# These methods get called before/after. You can check security information for all requests.
@application.before_request
def before_decorator():
    a_ok = sec.check_authetication(request)
    print("a_ok = ", a_ok)
    if a_ok[0] != 200:
        handle_error(a_ok[0], a_ok[1], a_ok[2])


@application.after_request
def after_decorator(rsp):
    print("... In after decorator ...")
    notify.notify(request, rsp)
    return rsp
```

- White list of URLs
- If not on white list:
  - Verify token
  - Verify authorization

- List of filters of the form:
  - URL
  - HTTP Method
  - SNS topic to use
- Send an event to SNS if matches filter

```
@application.route("/api/users", methods=["GET", "POST"])
def users_get():...
```

Show some demo code.

So, How Will We Use This?

- Add before_request and after_request to microservices.
  (Note: There are other approaches).
- Implement two middleware interceptors:
  - Security
    - Configured with a JSON file.
    - File lists which {path, method} not being logged on.
    - While IDs can perform which methods on which paths.
    - Automates driving login redirect (401 – Unauthorized)
    - Checks authorization (403– Forbidden)
  - Notification:
    - Configured with JSON in environment variables.
    - File lists which operations emit a before and/or after event to an SNS topic.
- We will use notification to drive some other things like webhooks.

- I will demo middleware and webhook today.
- Your project will use middleware and SNS.

Event Drive Processing
Publish/Subscribe

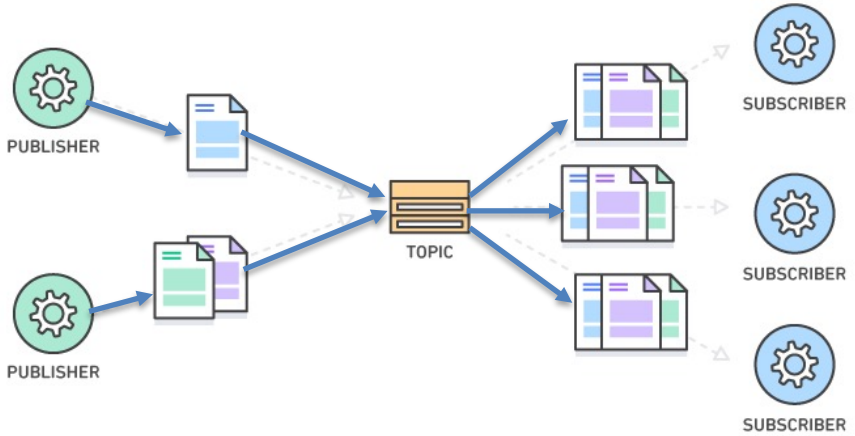
Pub/Sub – Event Driven Processing

"In software architecture, **publish–subscribe** is a messaging pattern where senders of messages do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize pub classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in o and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

Publish–subscribe is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system. Most messaging systems support both the pub/sub and message queue models in their API, e.g. Java Message Service (JMS).

This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the publisher and the structure of the published data."

https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

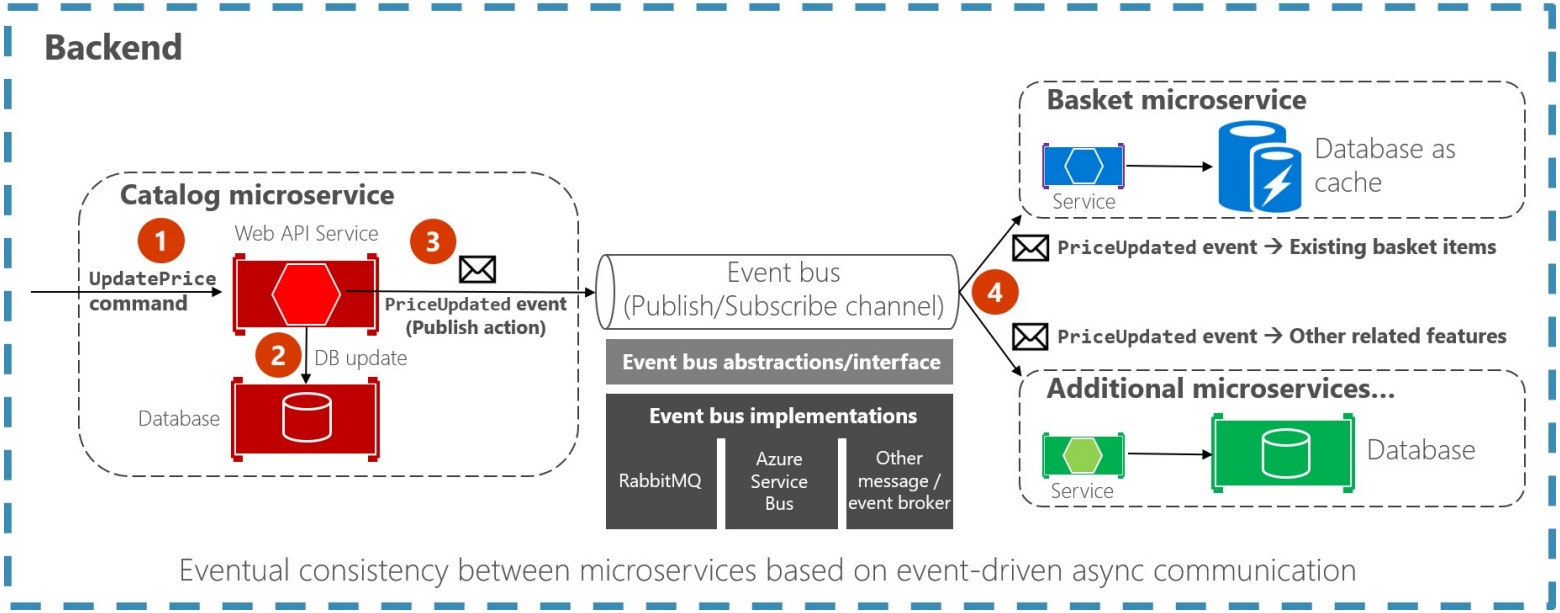


For microservices:

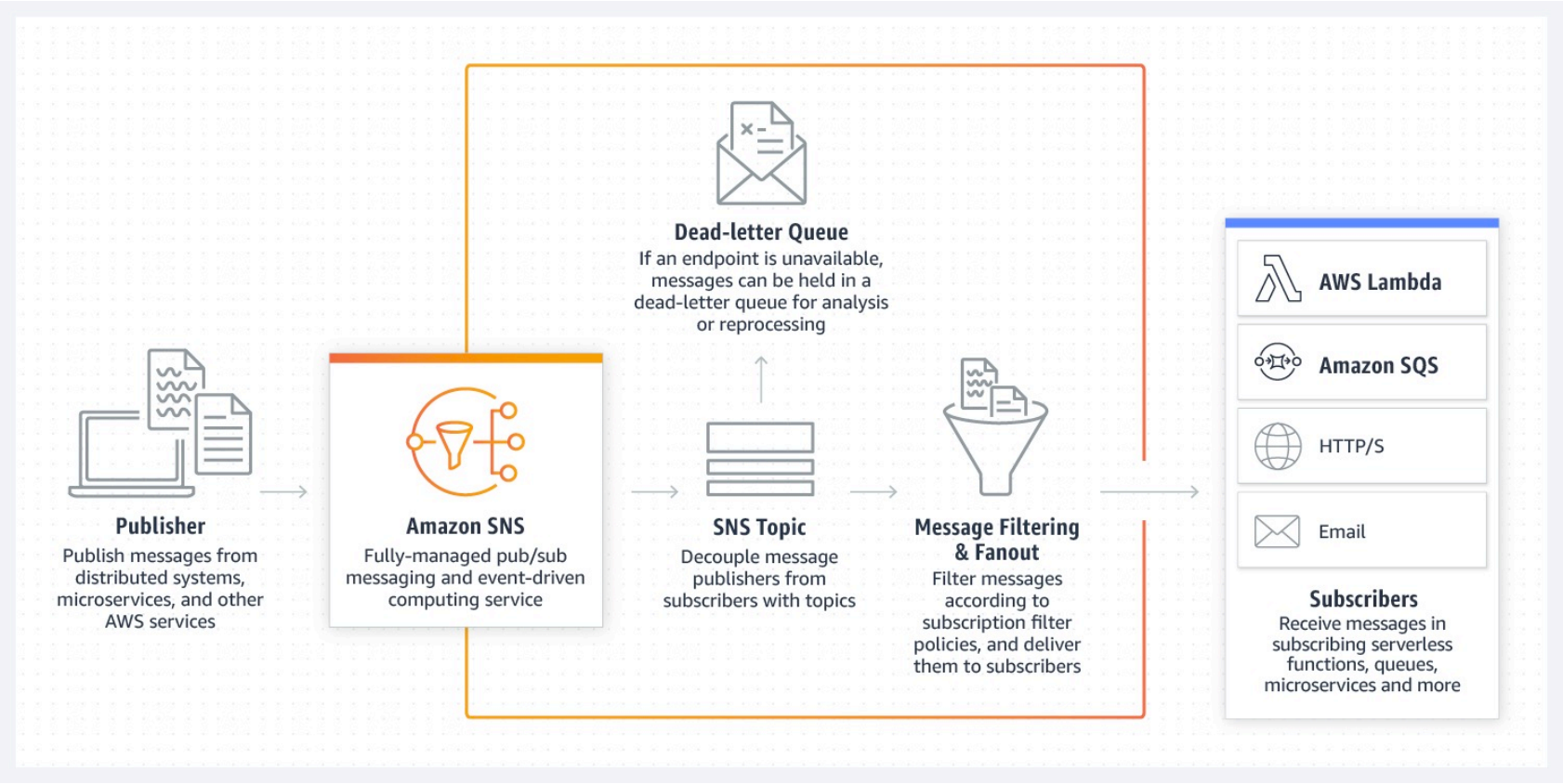
- It support loose coupling for application evolution and agility.
- I can add and evolve services without modifying the the base services.
- For example,
  - If creating a customer service **calls** "send an email."
  - I have to modify the original code to add support for SMS, webhook, etc.
  - With EDA, I just add or modify subscriptions.

Concept

Implementing asynchronous event-driven communication with an event bus

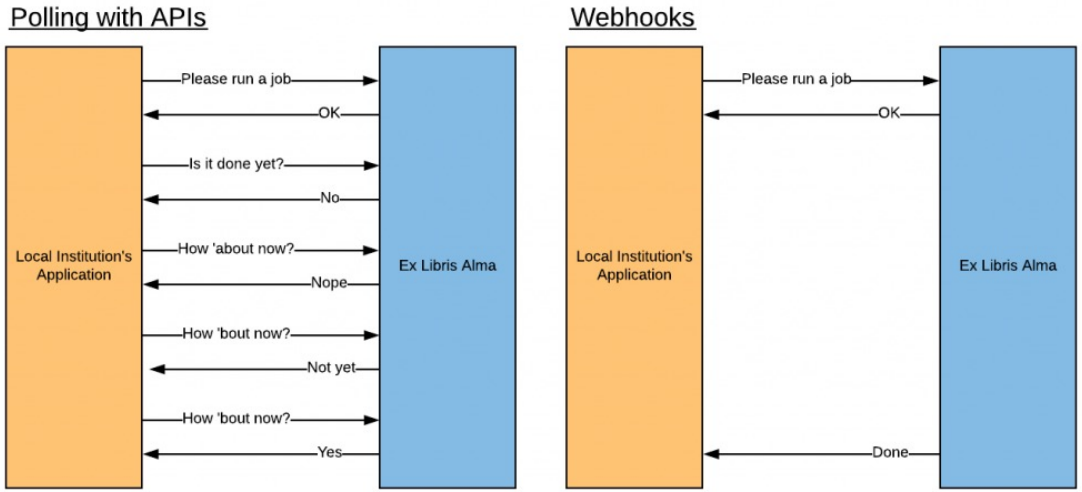


https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications



Webhooks

- “A webhook in web development is a method of augmenting or altering the behavior of a web page or web application with custom callbacks. These callbacks may be maintained, modified, and managed by third-party users and developers who may not necessarily be affiliated with the originating website or application. The term "webhook" was coined by Jeff Lindsay in 2007 from the computer programming term hook.”



Function-as-a-Service

Serverless and Function-as-a-Service

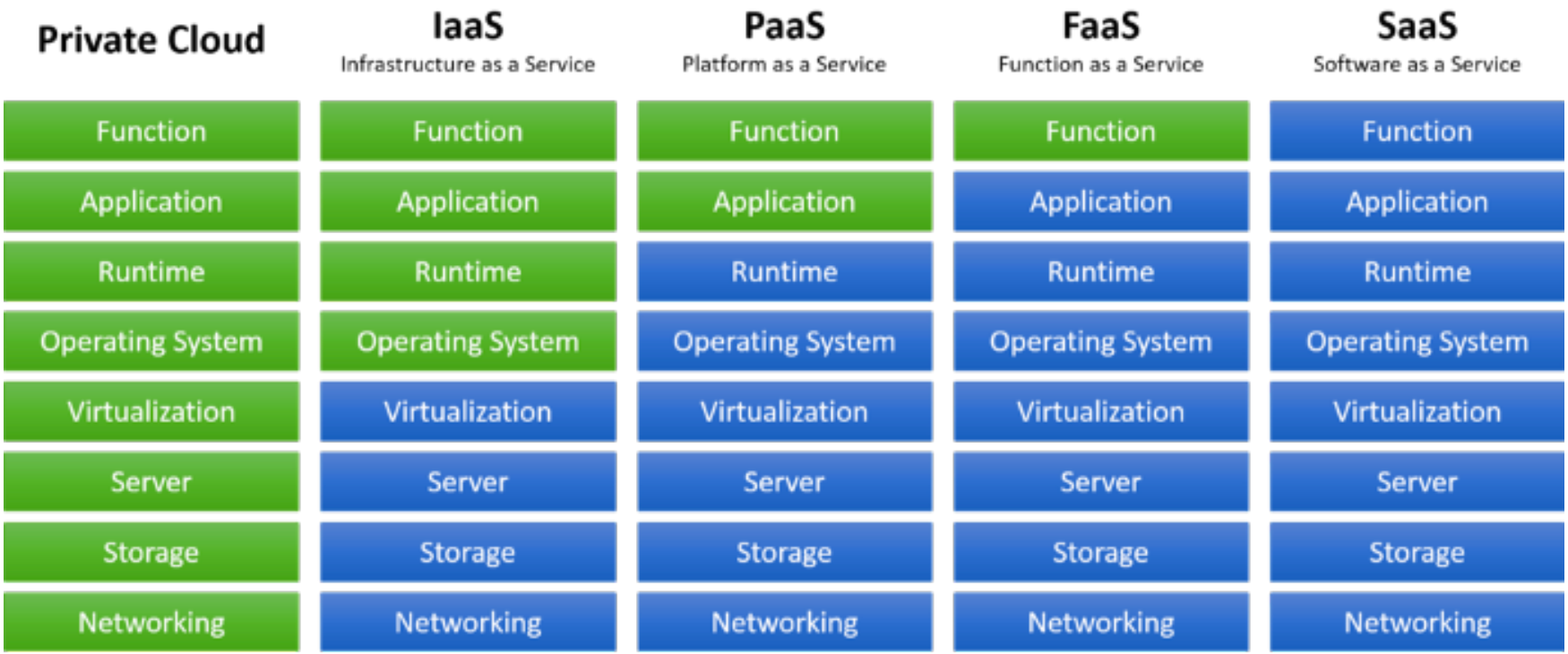
- **Serverless computing** is a cloud computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of machine resources. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity.[1] It can be a form of utility computing.

  Serverless computing can simplify the process of deploying code into production. Scaling, capacity planning and maintenance operations may be hidden from the developer or operator. Serverless code can be used in conjunction with code deployed in traditional styles, such as microservices. Alternatively, applications can be written to be purely serverless and use no provisioned servers at all.[2] This should not be confused with computing or networking models that do not require an actual server to function, such as peer-to-peer (P2P)."

- **Function as a service** (**FaaS**) is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.[1] Building an application following this model is one way of achieving a "serverless" architecture, and is typically used when building microservices applications."

- That was baffling:
  - IaaS, CaaS – You control the SW stack and the cloud provides (virtual) HW.
  - PaaS – The cloud hides the lower layer SW and provides an application container (e.g. Flask) with "Your code goes here." You are aware of container.
  - FaaS – The cloud provides the container, and you implement functions (corresponding to routes in REST).

Serverless and Function-as-a-Service



https://medium.com/@tanmayct/serverless-architecture-function-as-a-service-19e127b8c990

What is Serverless Good/Not Good For ... ...

Serverless is **good** for

short-running
stateless
event-driven

- Microservices
- Mobile Backends
- Bots, ML Inferencing
- IoT
- Modest Stream Processing
- Service integration

Serverless is **not good** for

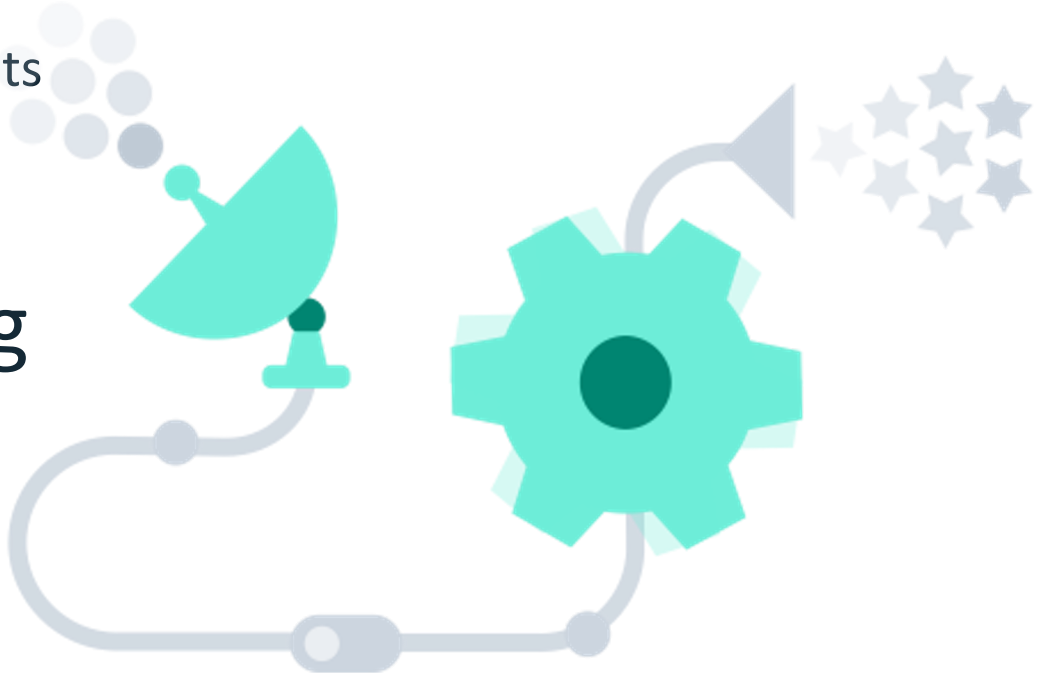
long-running
stateful
number crunching

- Databases
- Deep Learning Training
- Heavy-Duty Stream Analytics
- Numerical Simulation
- Video Streaming

What triggers code execution?

Runs code **in response** to events

Event-programming model



(Some) Current Platforms for Serverless

AWS
Lambda

OpenLambda

Azure
Functions

Kubernetes

IBM Cloud
Functions

Red-Hat

Google
Functions

Let’s Build a Lambda Function

- We are going to do it manually to understand the process/steps.

- There are several more complete approaches:
  - Pipelines
  - Serverless Framework (https://www.serverless.com/framework/docs/providers/aws/guide/intro)
  - AWS Toolkit for PyCharm (https://aws.amazon.com/pycharm/)

- We will explore some of these … …

API Gateway

API Management

API management is the process of creating and publishing web application programming interfaces (APIs), enforcing their usage policies, controlling access, nurturing the subscriber community, collecting and analyzing usage statistics, and reporting on performance. API Management components provide mechanisms and tools to support developer and subscriber community.[1]

Components [edit]

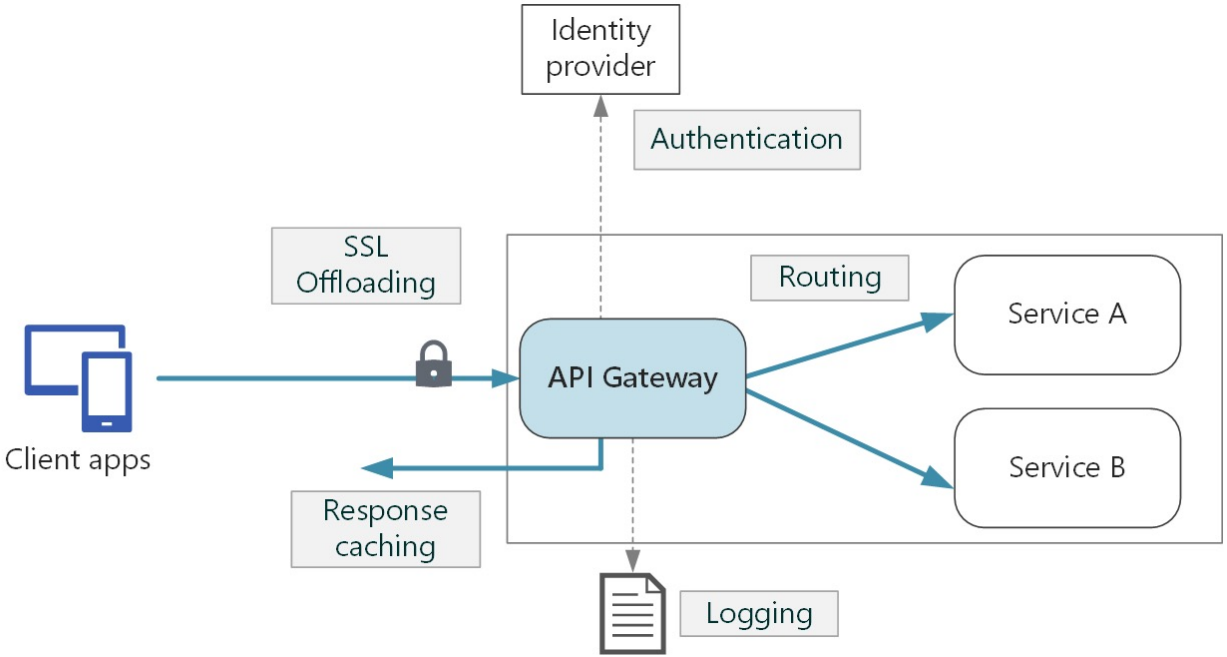
While solutions vary, components that provide the following functionality are typically found in API management products:

- **Gateway**: a server that acts as an API front-end, receives API requests, enforces throttling and security policies, passes requests to the back-end service and then passes the response back to the requester.[2] A gateway often includes a transformation engine to orchestrate and modify the requests and responses on the fly. A gateway can also provide functionality such as collecting analytics data and providing caching. The gateway can provide functionality to support authentication, authorization, security, audit and regulatory compliance.[3]
  It can be implemented by a Reverse proxy.
- **Publishing tools**: a collection of tools that API providers use to define APIs, for instance using the OpenAPI or RAML specifications, generate API documentation, govern API usage through access and usage policies for APIs, test and debug the execution of API, including security testing and automated generation of tests and test suites, deploy APIs into production, staging, and quality assurance environments, and coordinate the overall API lifecycle.
- **Developer portal/API store**: community site, typically branded by an API provider, that can encapsulate for API users in a single convenient source information and functionality including documentation, tutorials, sample code, software development kits, an interactive API console and sandbox to trial APIs, the ability to subscribe to the APIs and manage subscription keys such as OAuth2 Client ID and Client Secret, and obtain support from the API provider and user and community.
- **Reporting and analytics**: functionality to monitor API usage and load (overall hits, completed transactions, number of data objects returned, amount of compute time and other internal resources consumed, volume of data transferred). This can include real-time monitoring of the API with alerts being raised directly or via a higher-level network management system, for instance, if the load on an API has become too great, as well as functionality to analyze historical data, such as transaction logs, to detect usage trends. Functionality can also be provided to create synthetic transactions that can be used to test the performance and behavior of API endpoints. The information gathered by the reporting and analytics functionality can be used by the API provider to optimize the API offering within an organization's overall continuous improvement process and for defining software Service-Level Agreements for APIs.
- **Monetization**: functionality to support charging for access to commercial APIs. This functionality can include support for setting up pricing rules, based on usage, load and functionality, issuing invoices and collecting payments including multiple types of credit card payments.

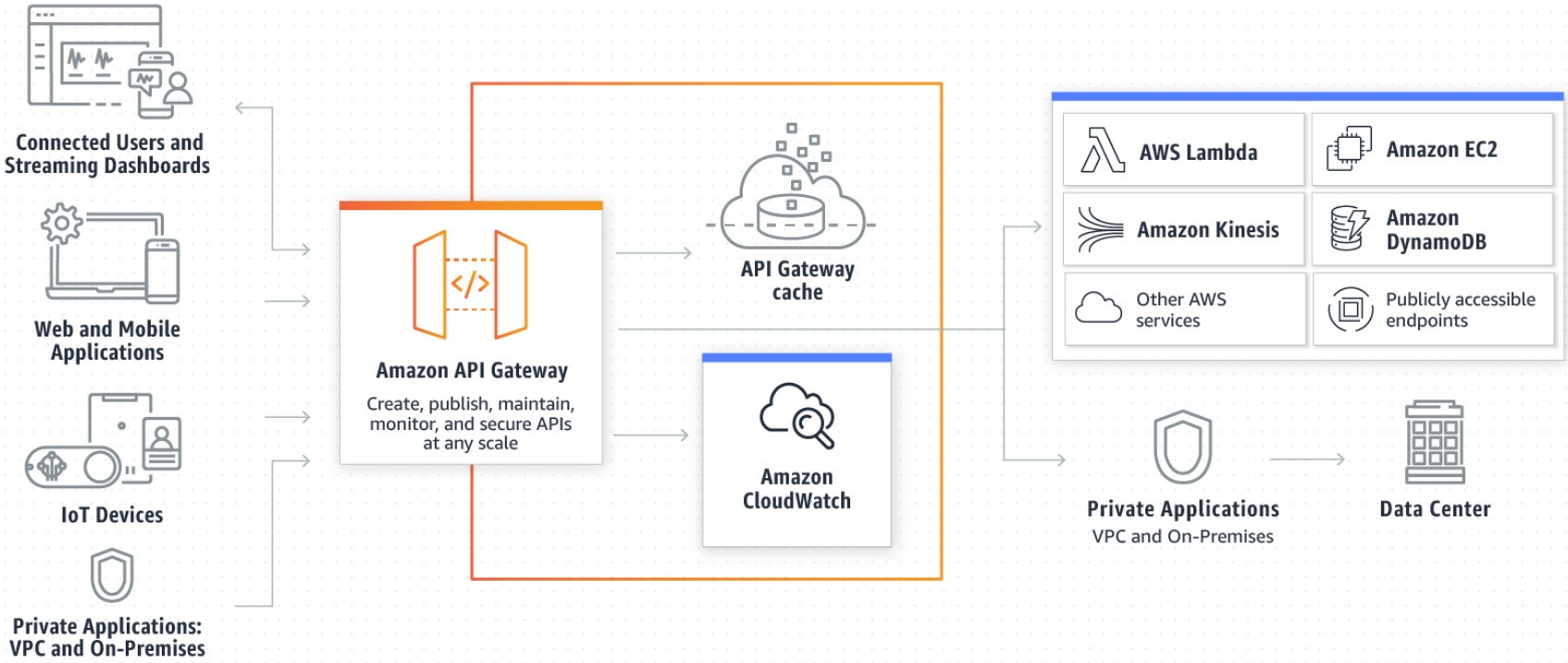
API Gateway (https://docs.microsoft.com/en-us/azure/architecture/microservices/design/gateway)

Some functions:

- SSL termination
- Authentication
- IP allow/block list
- Client rate limiting (throttling)
- Logging and monitoring
- Response caching
- Web application firewall
- GZIP compression
- Servicing static content

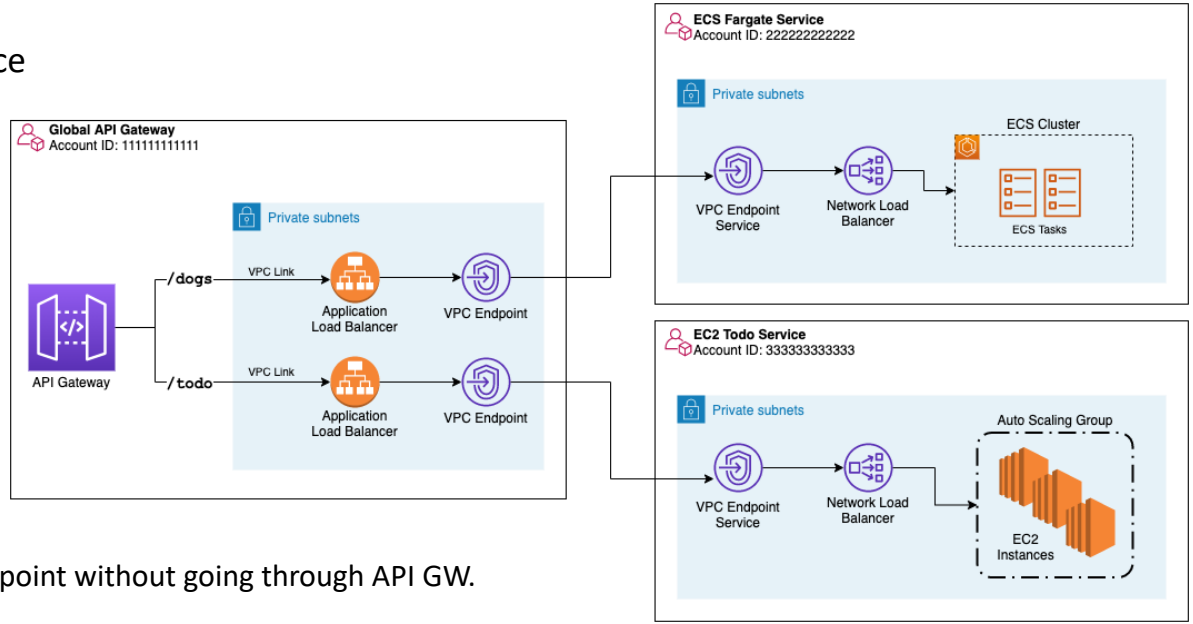


AWS API Gateway



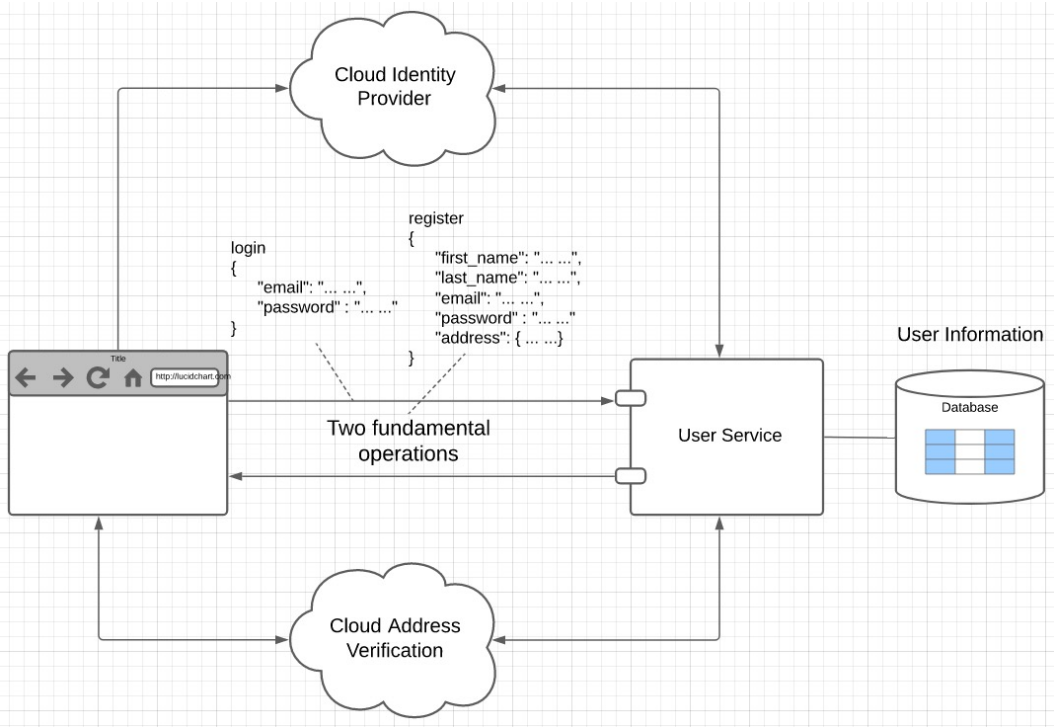
Initial Setup of an API Gateway

- Show tests of API GW
  - Lambda function
  - HTTP to Elastic Beanstalk instance
  - API Key
  - Deploy and stages
- API GW
  - Single point of entry.
  - Encapsulate things like EC2, Elastic Beanstalk, ...
  - Putting
    - Inside VPC
    - Behind ELB
    - Prevent access to the REST endpoint without going through API GW.
- The configuration can get complex.
  We will not worry about it and keep ELB and EC2 endpoints “public.”



Motivating Scenario

User Registration, Login and Profile



Two major challenges:

- User management:
  - Securing the user/password database.
  - Implementing password reset.
  - Verifying email ownership to avoid spamming by registration.
  - Two factor
  - … …
- Addresses:
  - People are very bad at entering address information.
  - "620 w 120th St, NY NY"
  - "620 West 120th Street, NYC, NY"
- Implementing these functions is tedious and non-core to your business. ➔ *Use cloud services.*

Cloud Identity
(Open ID, OAuth2, ...)
Reminder

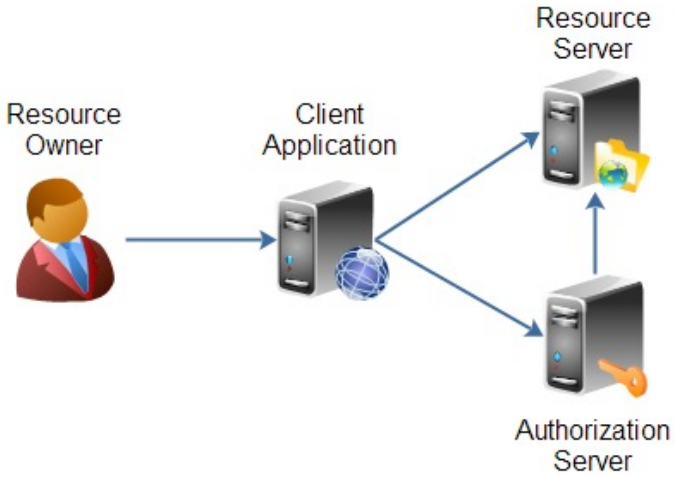
Overview (http://tutorials.jenkov.com/oauth2/index.html)

- Resource Owner
  - Controls *access* to the "data."
  - Facebook **user**, LinkedIn **user**, …
- Resource Server
  - The website that holds/manages info.
  - Facebook, LinkedIn, ...
  - And provides access API.
- Client Application
  - "The product you implemented."
  - Wants to read/update
    - "On your behalf"
    - The data the data that the Resource Server maintains
- Authorization Server
  - Grants/rejects authorization
  - Based on Resource Owner decisions.
  - Usually (logically) the same as Resource Server.

OAuth 2.0 defines the following roles of users and applications:

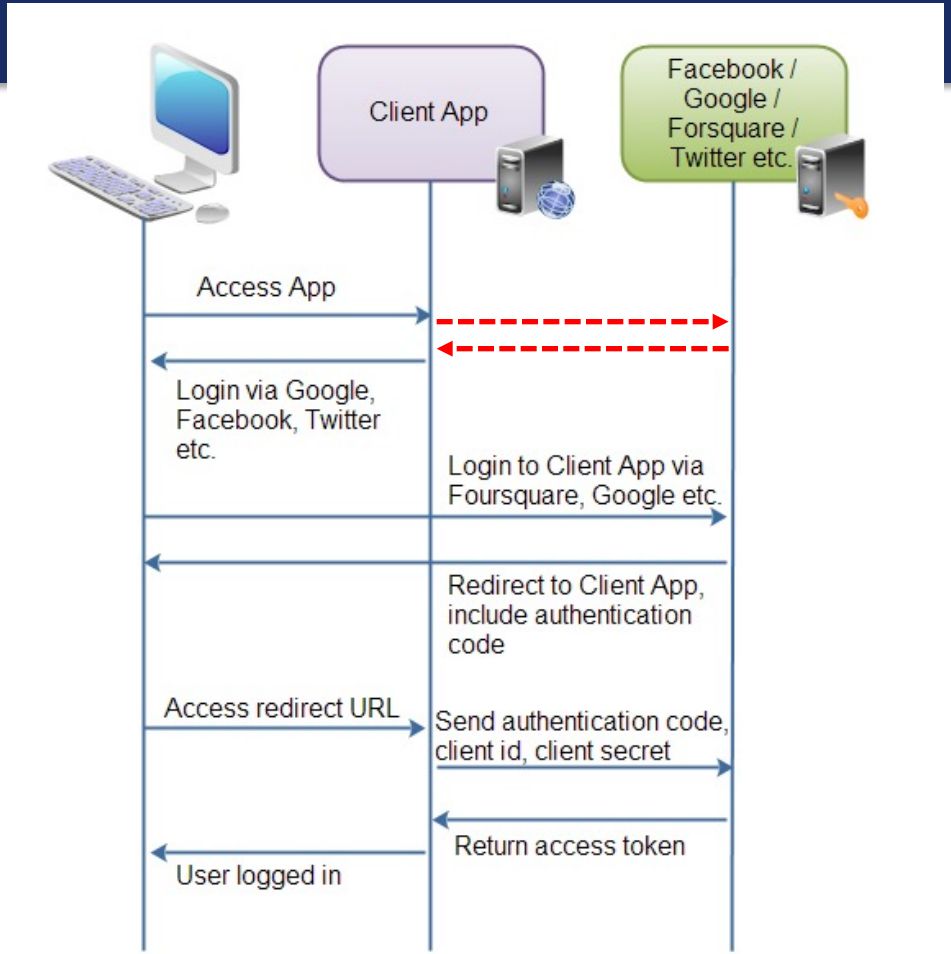
- Resource Owner
- Resource Server
- Client Application
- Authorization Server

These roles are illustrated in this diagram:

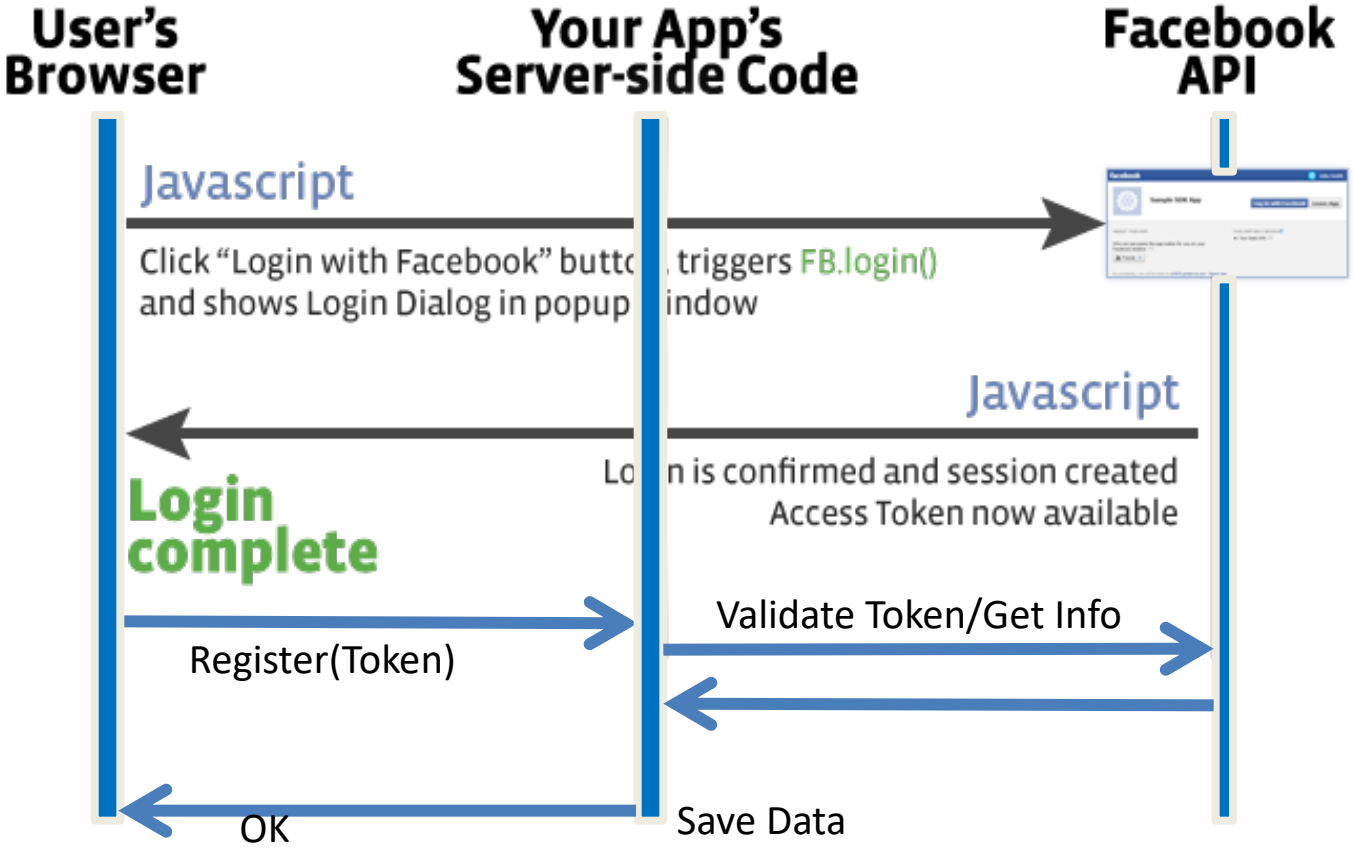


Roles/Flows

- User Clicks “Logon with XXX”
  - Redirect user to XXX
  - With Client App application ID.
  - And permissions app requests.
  - Code MAY have to call Resource Server to get a token to include in redirect URL.
- Browser redirected to XXX
  - Logon on to XXX prompt.
  - Followed by “do you grant permission to …”
  - Clicking button drives a redirect back to Client App. URL contains a temporary token.
- User/Browser
  - Redirected to Client App URL with token.
  - Client App calls XXX API
  - Obtains access token.
  - Returns to User.
- Client App can use access token on API calls.

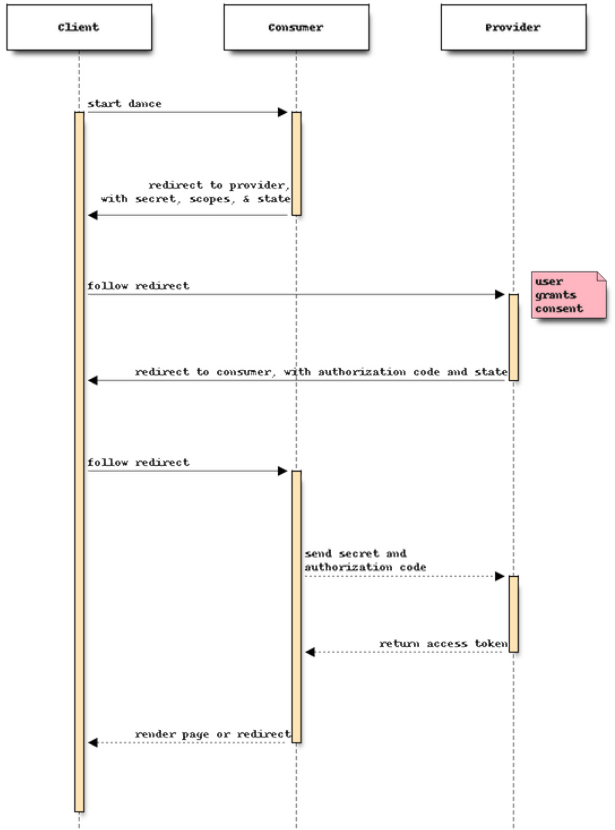


Facebook



Walkthrough: Middleware and Security

- Explain why I have to go incognito.
- Show:
  - Project code ... ...
  - GCP application registration page.
  - Token in the header (Postman)
- I normally do these things explicitly in my code.
  - I used Facebook in the past.
  - I decided to use Google because I expect people using my team management application to have Lionmail/Google accounts.
  - For expediency, I used Flask-Dance (https://flask-dance.readthedocs.io/en/latest/)
  - I understand the magic in general, but ...
  - I have not yet "learned the flask-dance magic" and how it uses the protocol under the hood.
- The net is:
  - OAuth2 is about *authorization,* but ...
  - Getting the basic profile proves *authentication*
  - And, I can use to simplify the registration process.



HW2, Sprint 2

HW2, Sprint 2

Backup

- Backup

REST Method

- When your handler receives a REST operation, there are the following elements. Not all methods have all elements:
  - Path
  - Path parameters
  - Query string parameters
  - Headers
  - Body
- The REST handler's response has the following elements:
  - Status code
  - Body
  - Headers
- The route handler maps between the REST representation of the elements and the language/runtime representation.

- Flask provides access through the
  - Request object.
  - Response object.
- Walkthrough

REST Paths https://restful-api-design.readthedocs.io/en/latest/

- There are many patterns. I typically follow something like:

URL Structure

Each collection and resource in the API has its own URL. URLs should never be constructed by an API client. Instead, the client should only follow links that are generated by the API itself.

The recommended convention for URLs is to use alternate collection / resource path segments, relative to the API entry point. This is best described by example. The table below uses the ":name" URL variable style from Rail's "Routes" implementation.

URL	Description
/api	The API entry point
/api/:coll	A top-level collection named "coll"
/api/:coll/:id	The resource "id" inside collection "coll"
/api/:coll/:id/:subcoll	Sub-collection "subcoll" under resource "id"
/api/:coll/:id/:subcoll/:subid	The resource "subid" inside "subcoll"

Even though sub-collections may be arbitrarily nested, in my experience, you want to keep the depth limited to 2, if possible. Longer URLs are more difficult to work with when using simple command-line tools like curl.

Method	Scope	Semantics
GET	collection	Retrieve all resources in a collection
GET	resource	Retrieve a single resource
HEAD	collection	Retrieve all resources in a collection (header only)
HEAD	resource	Retrieve a single resource (header only)
POST	collection	Create a new resource in a collection
PUT	resource	Update a resource
PATCH	resource	Update a resource
DELETE	resource	Delete a resource
OPTIONS	any	Return available HTTP methods and other options

- We will not be too rigorous about status codes, but some common examples are:
  - 200: OK (success)
  - 201: CREATED (for POST)
  - 404: NOT FOUND
  - 500: INTERNAL SERVICE ERROR
  - 418: I'M A TEAPOT

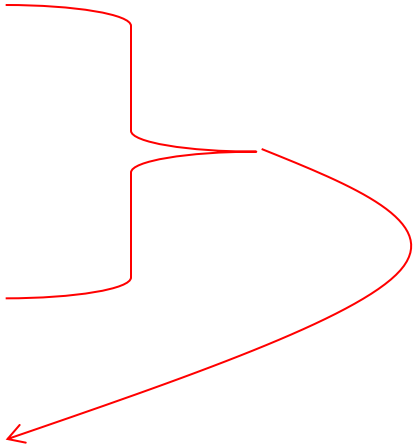
Success Response Codes

Operation	HTTP Request	HTTP Response Codes Supported
READ	GET	200 - OK with message body 204 - OK no message body 206 - OK with partial message body
CREATE	POST	201 - Resource created (Operation Complete) 202 - Resource accepted (Operation Pending)
UPDATE	PUT	202 - Accepted (Operation Pending) 204 - Success (Operation Complete)
DELETE	DELETE	202 - Accepted (Operation Pending) 204 - Success (Operation Complete)

202 means

Your request went asynch.

The HTTP header Link is where to poll for rsp.

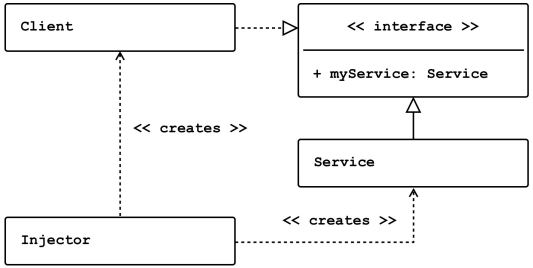


Examples of Link Headers in HTTP response:

```
Link: <http://api/jobs/j1>;rel=monitor;title="update profile"
Link: <http://api/reports/r1>;rel=summary;title="access report"
```

Dependency Injection

- Concepts (https://en.wikipedia.org/wiki/Dependency_injection):
  - In software engineering, dependency injection is a technique in which an object receives other objects that it depends on, called dependencies.
  - The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.

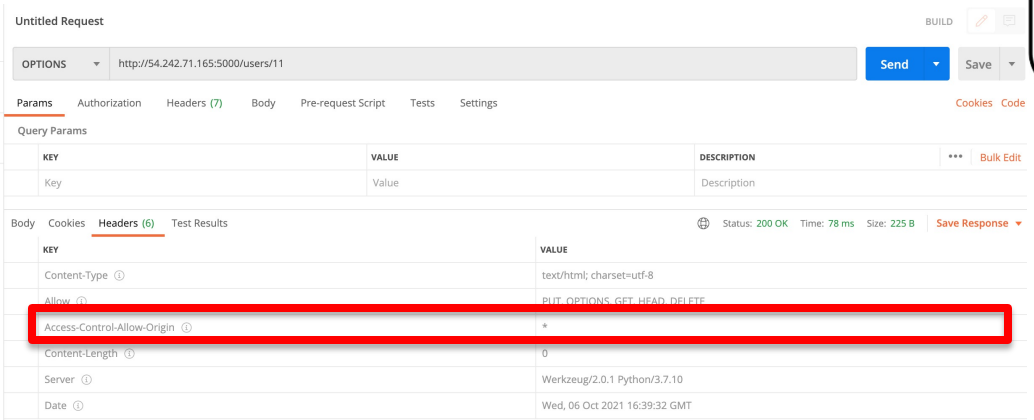


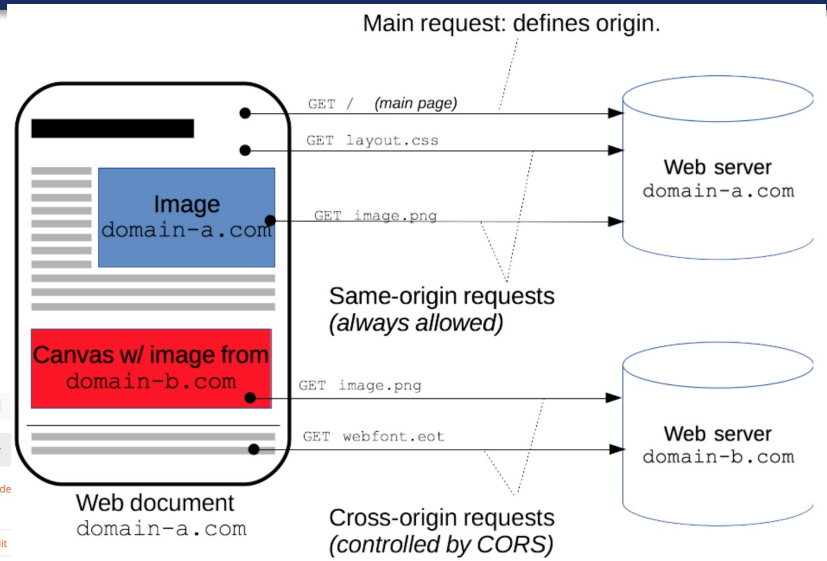

```
class UserResource(BaseApplicationResource):

    def __init__(self, config_info):
        super().__init__(config_info)
```

- There is no single approach. There are several frameworks, tools, etc.
- Like many other things, it is easy to get carried away and become dogmatic.
- I follow a single approach:
  - A Context class converts environment and other configuration and provides to application.
  - The top-level application injects a config_info object into services.

CORS https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

- "For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, XMLHttpRequest and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers."






```
app = Flask(__name__)
CORS(app)
```

REST Principles
HATEOAS

REST Principles

- REST Principles:
  - Uniform interface
  - Client–server
  - Stateless
  - Cacheable
  - Layered system
  - Code on demand (optional)

HATEOAS: Hypertext as the Engine of Application State -- The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

- Many of these are straightforward. We will explore some of the concepts in various lectures.

- Today, we cover “statelessness.”

Pagination and Field Selection

Pagination

Most endpoints that returns a list of entities will need to have some sort of pagination.

Without pagination, a simple search could return millions or even billions of hits causing extraneous network traffic.

Paging requires an implied ordering. By default this may be the item's unique identifier, but can be other ordered fields such as a created date.

Offset Pagination

This is the simplest form of paging. Limit/Offset became popular with apps using SQL databases which already have LIMIT and OFFSET as part of the SQL SELECT Syntax. Very little business logic is required to implement Limit/Offset paging.

Limit/Offset Paging would look like `GET /items?limit=20&offset=100`. This query would return the 20 rows starting with the 100th row.

Example

(Assume the query is ordered by created date descending)

1. Client makes request for most recent items: `GET /items?limit=20`
2. On scroll/next page, client makes second request `GET /items?limit=20&offset=20`
3. On scroll/next page, client makes third request `GET /items?limit=20&offset=40`

Retrieving specific fields: use a fields query parameter

Your API should allow end users to retrieve specific fields, instead of retrieving the full object. To do this, use a query parameter named fields. For example, to retrieve only the first_name and last_name attributes of the users:

```
GET /users?fields=first_name,last_name
```

https://www.moesif.com/blog/technical/api-design/REST-API-Design-Filtering-Sorting-and-Pagination/