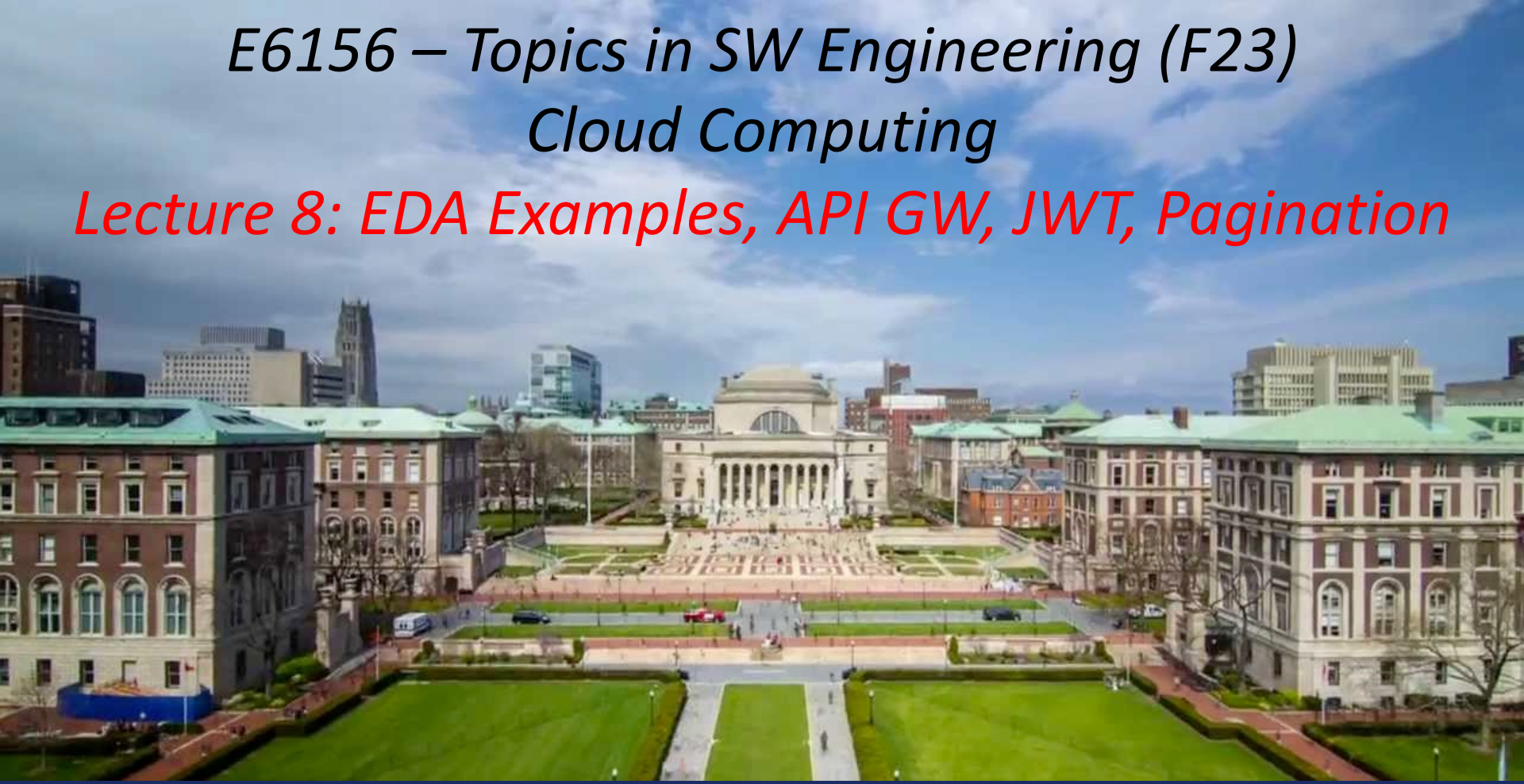
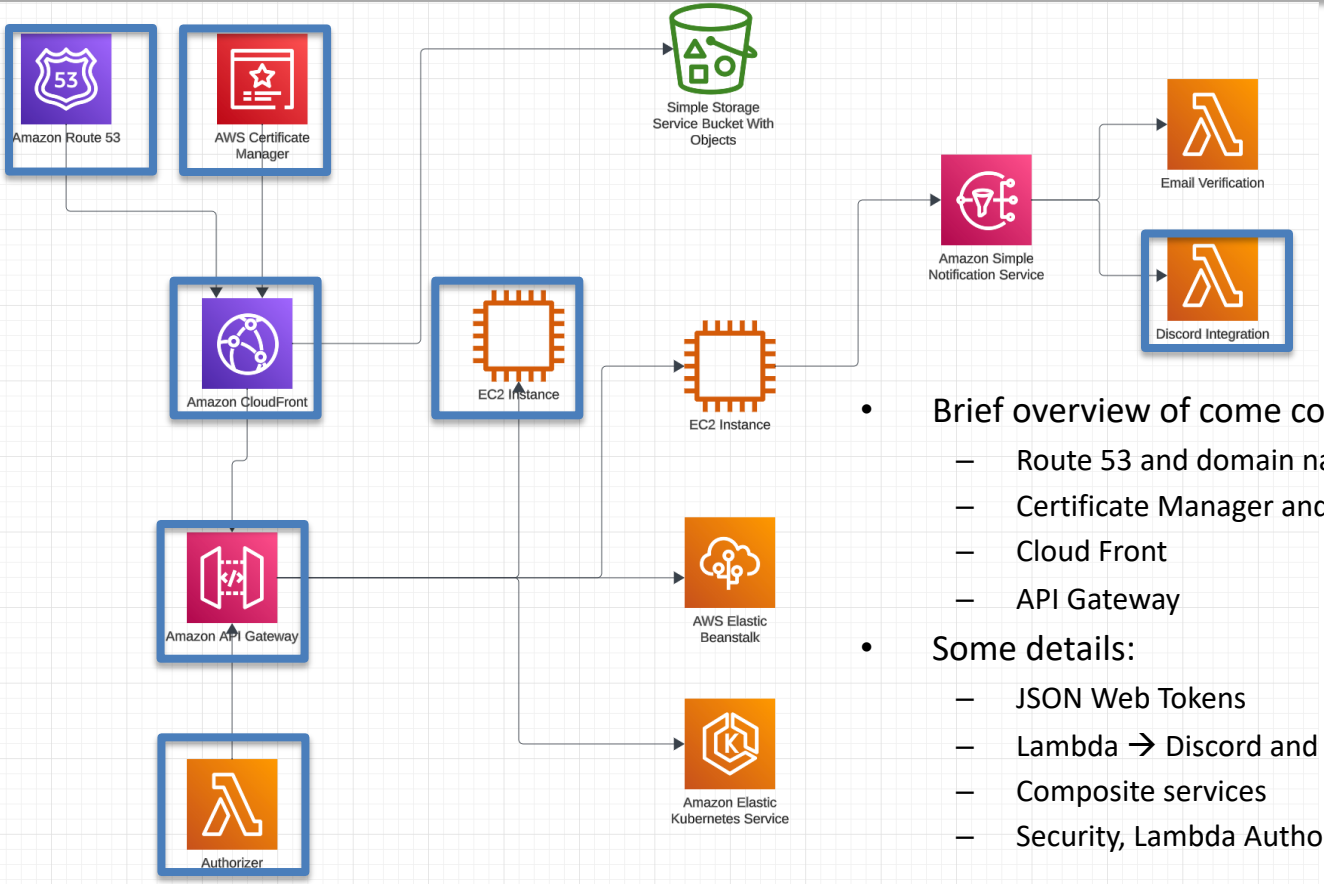
E6156 – Topics in SW Engineering (F23)
Cloud Computing
Lecture 8: EDA Examples, API GW, JWT, Pagination



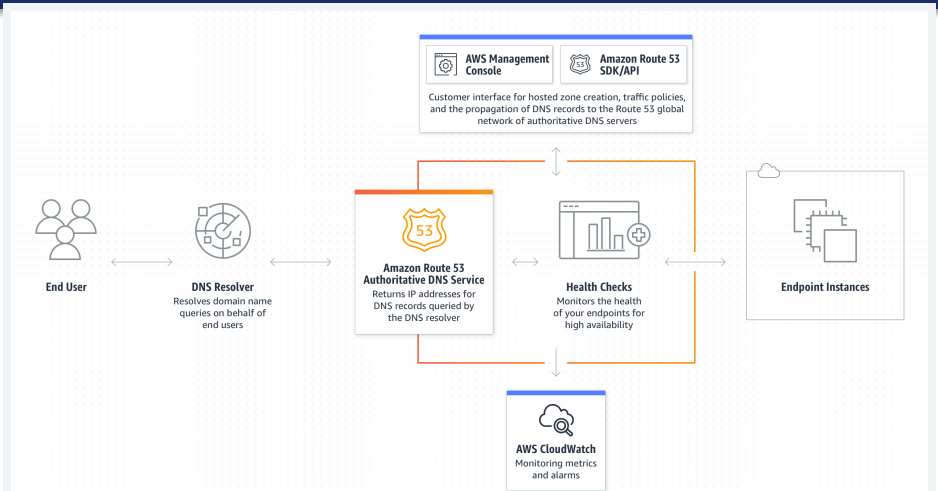
Some Aspects of My Solution/Project

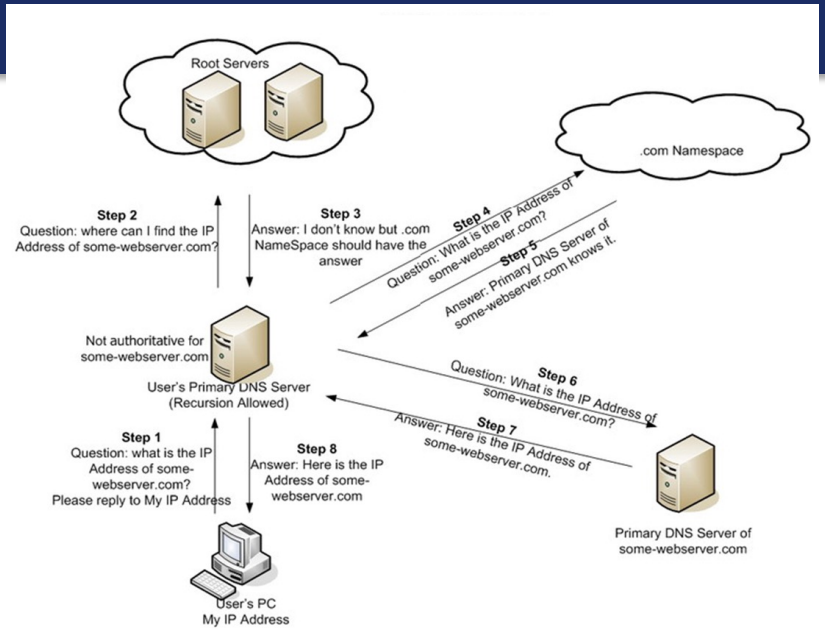
Partial Architecture Diagram



- Brief overview of come concepts:
  - Route 53 and domain name servers
  - Certificate Manager and HTTPs
  - Cloud Front
  - API Gateway
- Some details:
  - JSON Web Tokens
  - Lambda → Discord and Webhooks
  - Composite services
  - Security, Lambda Authorizer

DNS and Route 53





- Internet connectivity relies on IP addresses.
  - V4: 164.86.3.127
  - 2001:db8:3333:4444:CCCC:DDDD:EEEE:FFFF
- DNS transforms "names" like www.columbia.edu to IP addresses
- May implement more advanced functions like load balancing, … …
- Most large environments have a private IP address space, with some of the nodes also having a public IP address.
  - This is due to the limitations on the number of IP addresses publicly available.
  - There are network address translation gateways at the perimeter.

HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP). It uses encryption for secure communication over a computer network, and is widely used on the Internet.[1][2] In HTTPS, the communication protocol is encrypted using Transport Layer Security (TLS) or, formerly, Secure Sockets Layer (SSL). The protocol is therefore also referred to as HTTP over TLS,[3] or HTTP over SSL.

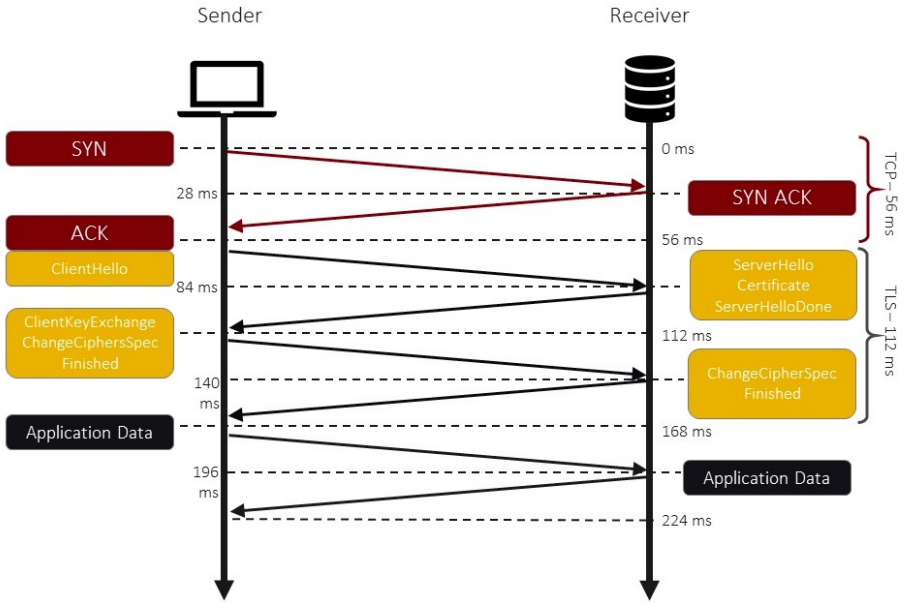
… …

The principal motivations for HTTPS are authentication of the accessed website and protection of the privacy and integrity of the exchanged data while it is in transit. It protects against man-in-the-middle attacks, …

The authentication aspect of HTTPS requires a trusted third party to sign server-side digital certificates.

https://en.wikipedia.org/wiki/HTTPS

https://anushadasari.medium.com/the-https-protocol-explained-under-the-hood-c7bd9f9aaa7b



Certificates

Web browsers know how to trust HTTPS websites based on certificate authorities that come pre-installed in their software. Certificate authorities are in this way being trusted by web browser creators to provide valid certificates. Therefore, a user should trust an HTTPS connection to a website if and only if all of the following are true:

- The user trusts that their device, hosting the browser and the method to get the browser itself, is not compromised (i.e. there is no supply chain attack).
- The user trusts that the browser software correctly implements HTTPS with correctly pre-installed certificate authorities.
- The user trusts the certificate authority to vouch only for legitimate websites (i.e. the certificate authority is not compromised and there is no mis-issuance of certificates).
- The website provides a valid certificate, which means it was signed by a trusted authority.
- The certificate correctly identifies the website (e.g., when the browser visits "https://example.com", the received certificate is properly for "example.com" and not some other entity).
- The user trusts that the protocol's encryption layer (SSL/TLS) is sufficiently secure against eavesdroppers

Certificates and PKI

A public key infrastructure (PKI) is a set of roles, policies, hardware, software and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption.

… …

In cryptography, a PKI is an arrangement that binds public keys with respective identities of entities (like people and organizations).[1] The binding is established through a process of registration and issuance of certificates at and by a certificate authority (CA)

PKI provides "trust services" - in plain terms trusting the actions or outputs of entities, be they people or computers. Trust service objectives respect one or more of the following capabilities: Confidentiality, Integrity and Authenticity (CIA).

- Confidentiality: Assurance that no entity can maliciously or unwittingly view a payload in clear text. Data is encrypted to make it secret, such that even if it was read, it appears as gibberish. Perhaps the most common use of PKI for confidentiality purposes is in the context of Transport Layer Security (TLS). TLS is a capability underpinning the security of data in transit, i.e. during transmission. A classic example of TLS for confidentiality is when using an internet browser to log on to a service hosted on an internet based web site by entering a password.
- Integrity: Assurance that if an entity changed (tampered) with transmitted data in the slightest way, it would be obvious it happened as its integrity would have been compromised. Often it is not of utmost importance to prevent the integrity being compromised (tamper proof), however, it is of utmost importance that if integrity is compromised there is clear evidence of it having done so (tamper evident).
- Authenticity: Assurance that every entity has certainty of what it is connecting to, or can evidence its legitimacy when connecting to a protected service. The former is termed server-side authentication - typically used when authenticating to a web server using a password. The latter is termed client-side authentication - sometimes used when authenticating using a smart card (hosting a digital certificate and private key).

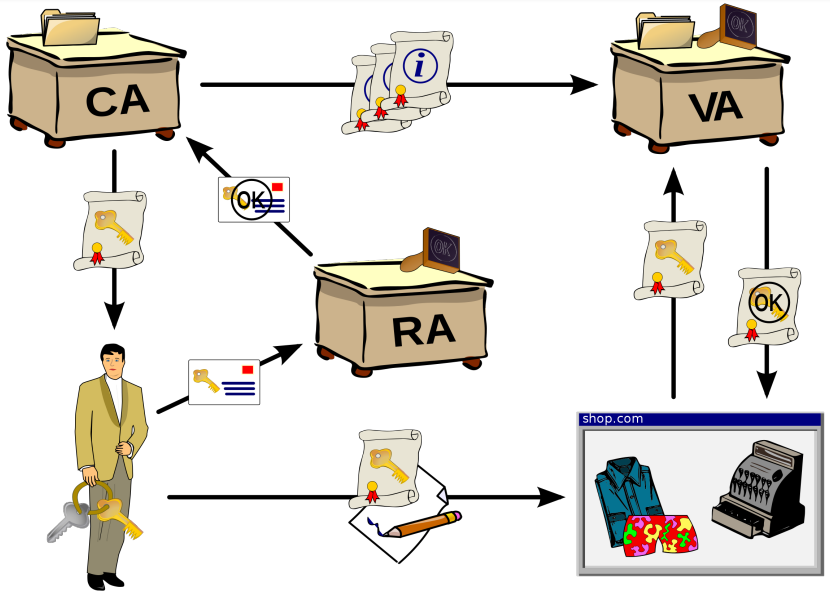
PKI

A PKI consists of:

- A certificate authority (CA) that stores, issues and signs the digital certificates;
- A registration authority (RA) which verifies the identity of entities requesting their digital certificates to be stored at the CA;
- A central directory—i.e., a secure location in which keys are stored and indexed;
- A certificate management system managing things like the access to stored certificates or the delivery of the certificates to be issued;
- A certificate policy stating the PKI's requirements concerning its procedures. Its purpose is to allow outsiders to analyze the PKI's trustworthiness.

In my environment:

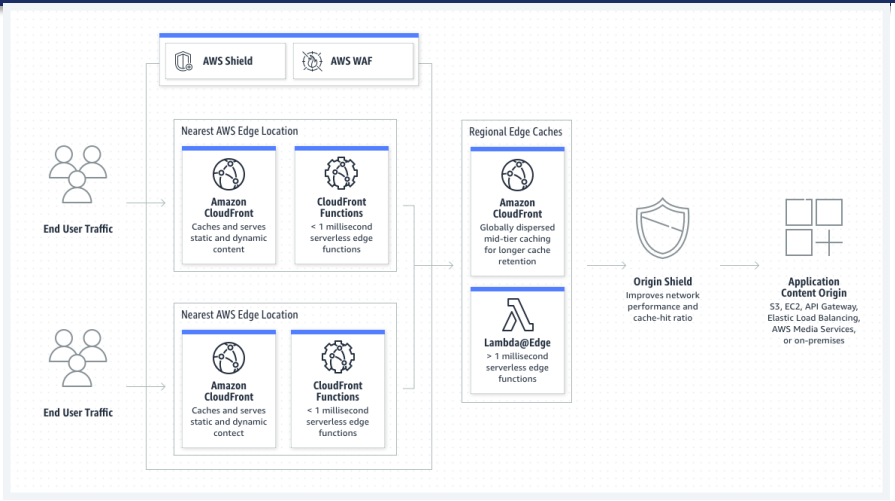
- AWS Certificate Authority is the CA
- Route 53 is the Registration Authority
- Embedded technology in browsers verifies server side certificates.

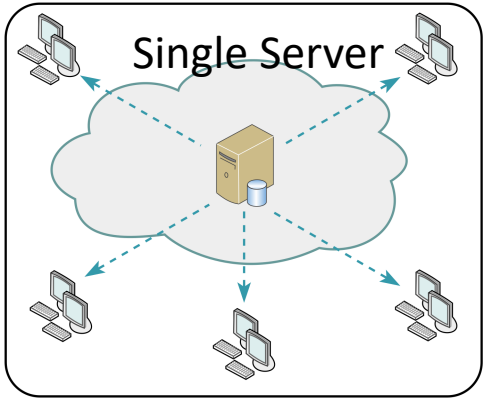


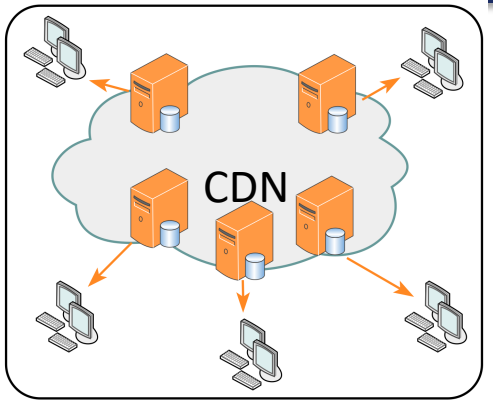
To show:

- Certificate in browser.
- Route 53
- Certificate Manager
- Certificate in CloudFront

Cloud Front







"A content delivery network, or content distribution network (CDN), is a geographically distributed network of proxy servers and their data centers. The goal is to provide high availability and performance by distributing the service spatially relative to end users."
https://en.wikipedia.org/wiki/Content_delivery_network

They key concepts are: Origin Server, Edge Server, Cache Policy

To show: CloudFront, HTTP cache headers.

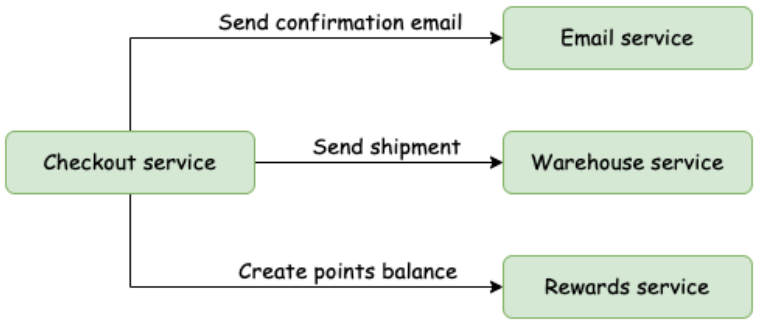
Composition, Orchestration, Choreography
Event Driven Architecture Examples

Some Concepts

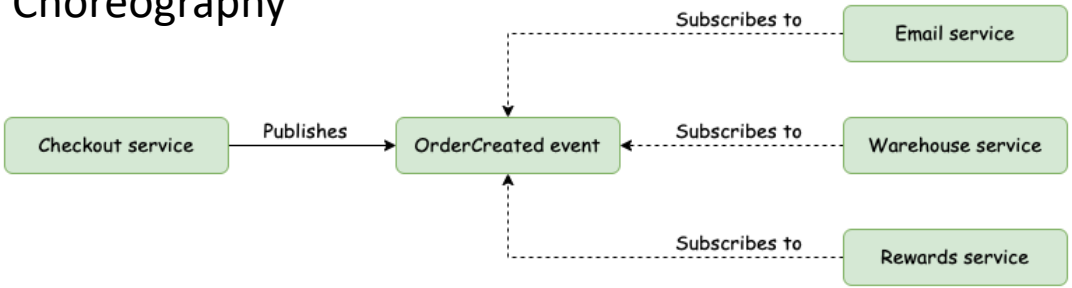
- There are two concepts but without precise, universally agreed meanings.
  - Orchestration: “On the other hand, the orchestrator sends a command message to downstream microservices, telling it what operation to perform. After this microservice has performed the operation, it sends a reply message to the orchestrator. The orchestrator then processes the message and determines which next step to perform.”
  - Choreography” With choreography, you inform each part of the system of its job and let it work out the details, like dancers all finding their way and reacting to others around them in a ballet.”
- There are many good overviews of the concepts”
  - https://ninetailed.io/blog/microservices-orchestration/
  - https://camunda.com/blog/2023/02/orchestration-vs-choreography/
  - https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/orchestration-choreography.html
  - https://betterprogramming.pub/microservice-orchestration-vs-choreography-5595b602fe3b

Orchestration and Choreography

Orchestration

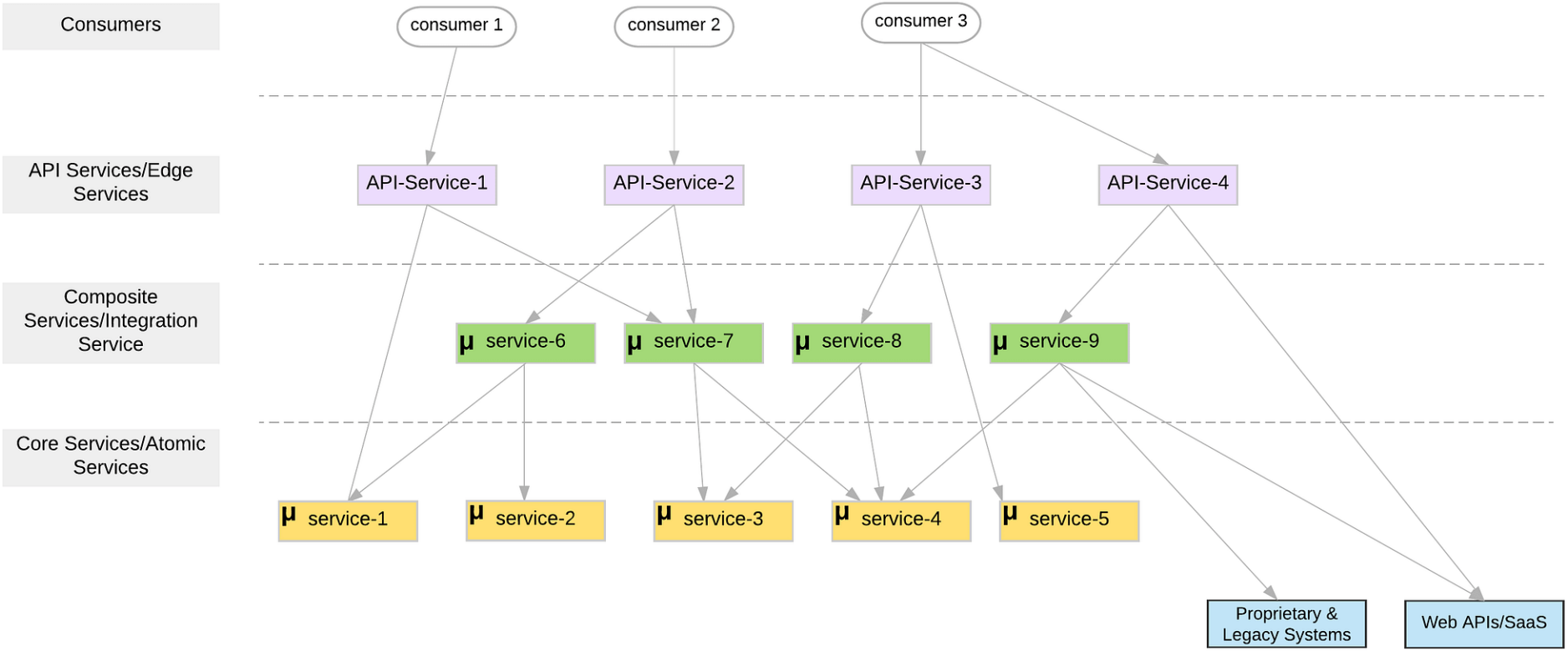


Choreography

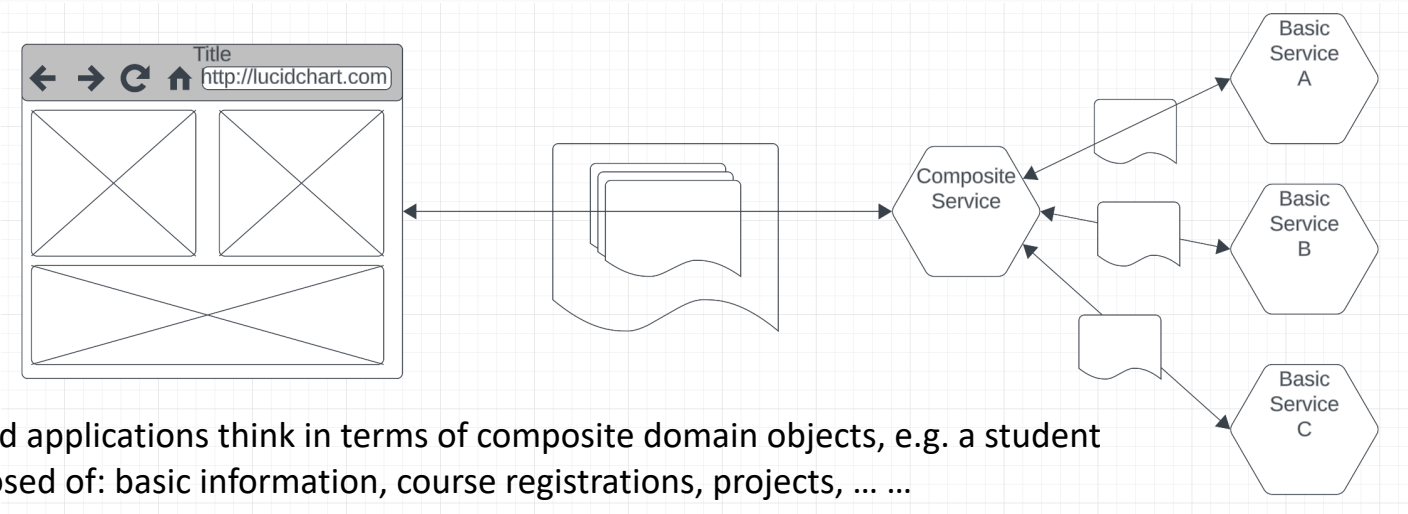


- Orchestration:
  - A N+1 service implements the composition logic.
  - Sends messages/events to other services to coordinate actions.
- Choreography:
  - Each service is independent and knows its function.
  - Subscribes to and published events.

Microservice Layers

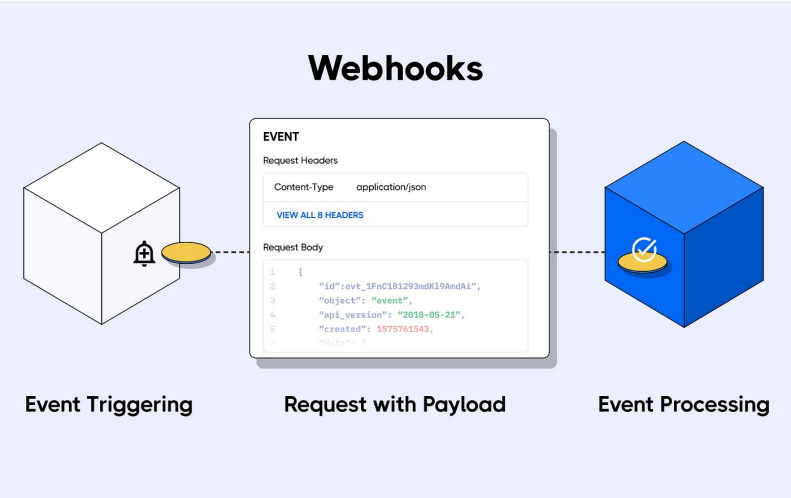


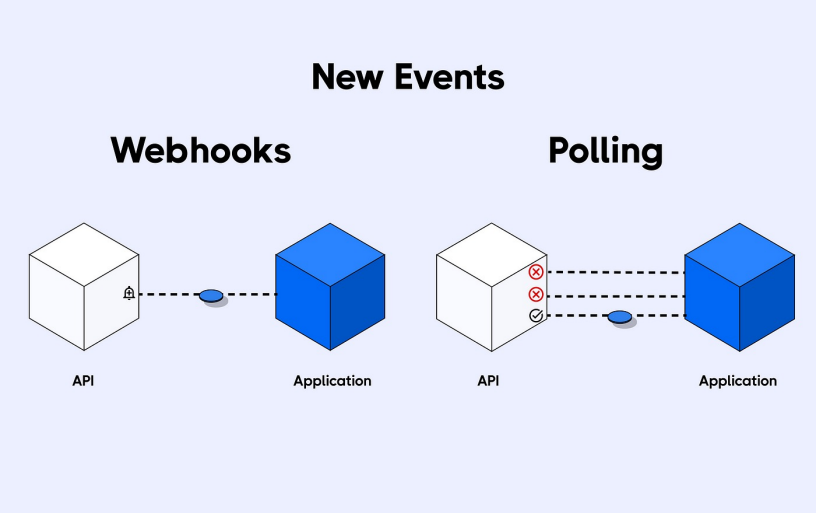
Composite Service



- Users and applications think in terms of composite domain objects, e.g. a student is composed of: basic information, course registrations, projects, … …
- UIs and other clients want to “retrieve the whole thing” and “update the whole thing.”
- Implementing composition in the browser is a poor design.
  - In general, a lot of business logic in the browser/client is a poor idea. Prone to errors and fragile when things change.
  - Something has to handle the operations on the composite
    - Constraints between the objects.
    - Read or update failures of parts of the composite
    - Etc.
- So, our next task is fleshing out implementation of the basic services functions and simple composite.

Webhooks

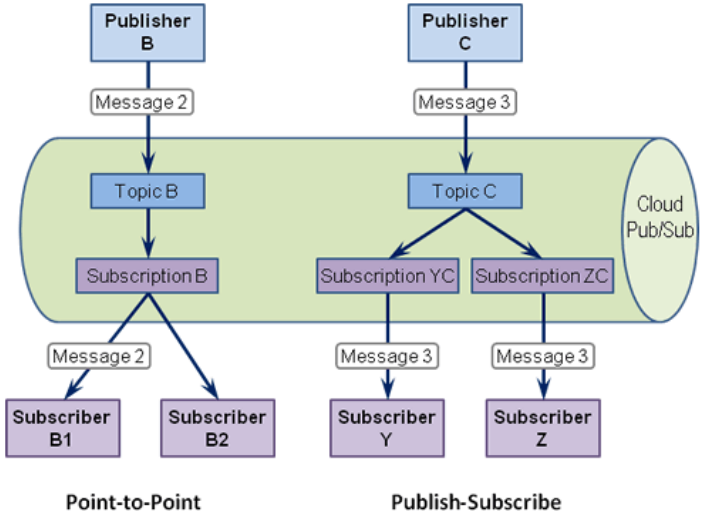


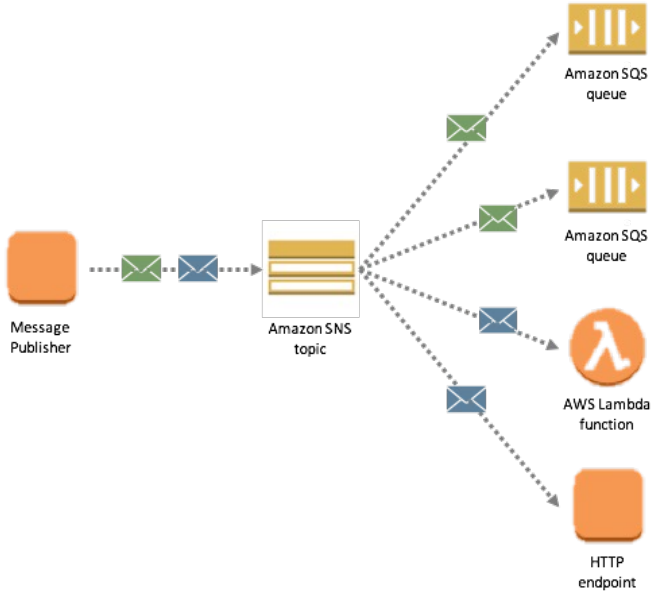


https://medium.com/hookdeck/webhooks-explained-what-are-webhooks-and-how-do-they-work-90c613c055a4

- Webhooks are a form of subscription.
- Used to connect web/REST applications using HTTP.
- Often used to integrate with user collaboration and user automation systems (Slack, Discord, …) with applications.

Pub/Sub

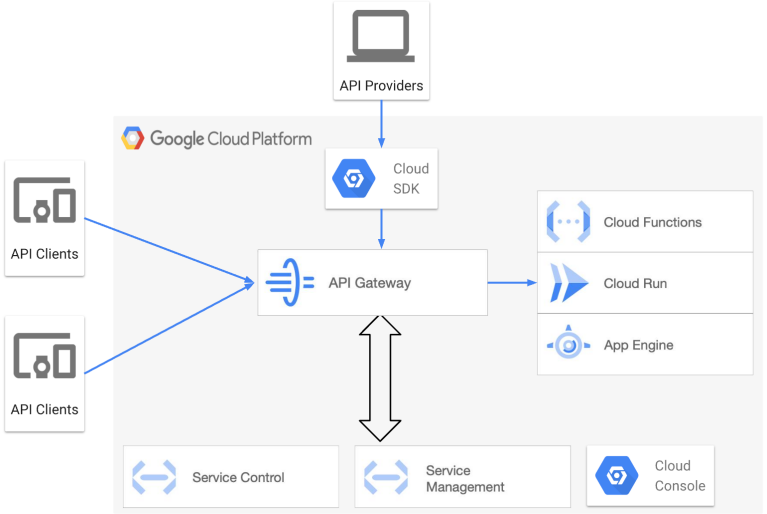


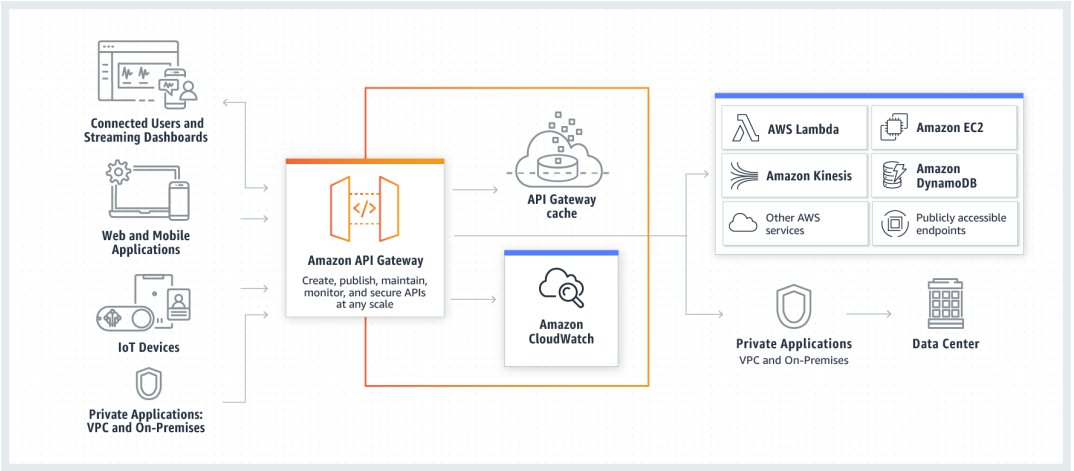


- Reshow the topic, subscription, API keys, publication, webhook for the Discord example.

API Gateway
JWT
Authorization

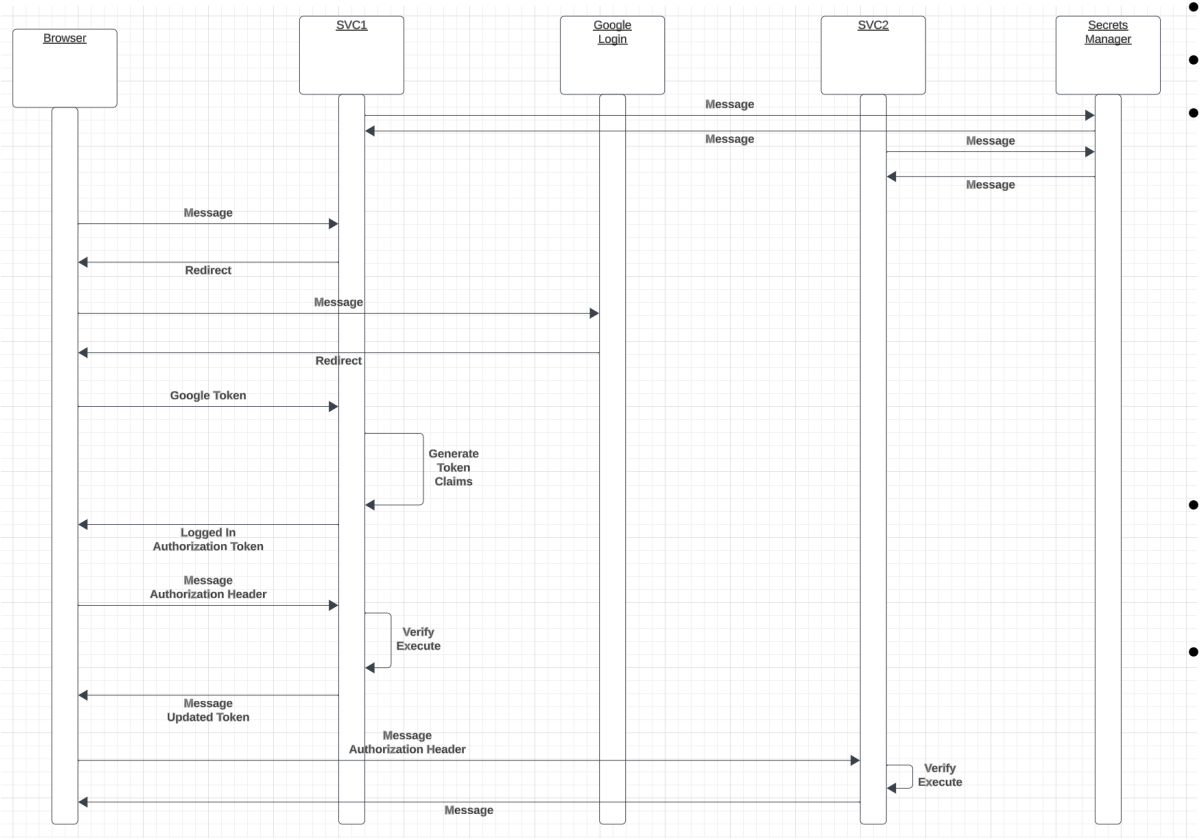
API Gateway





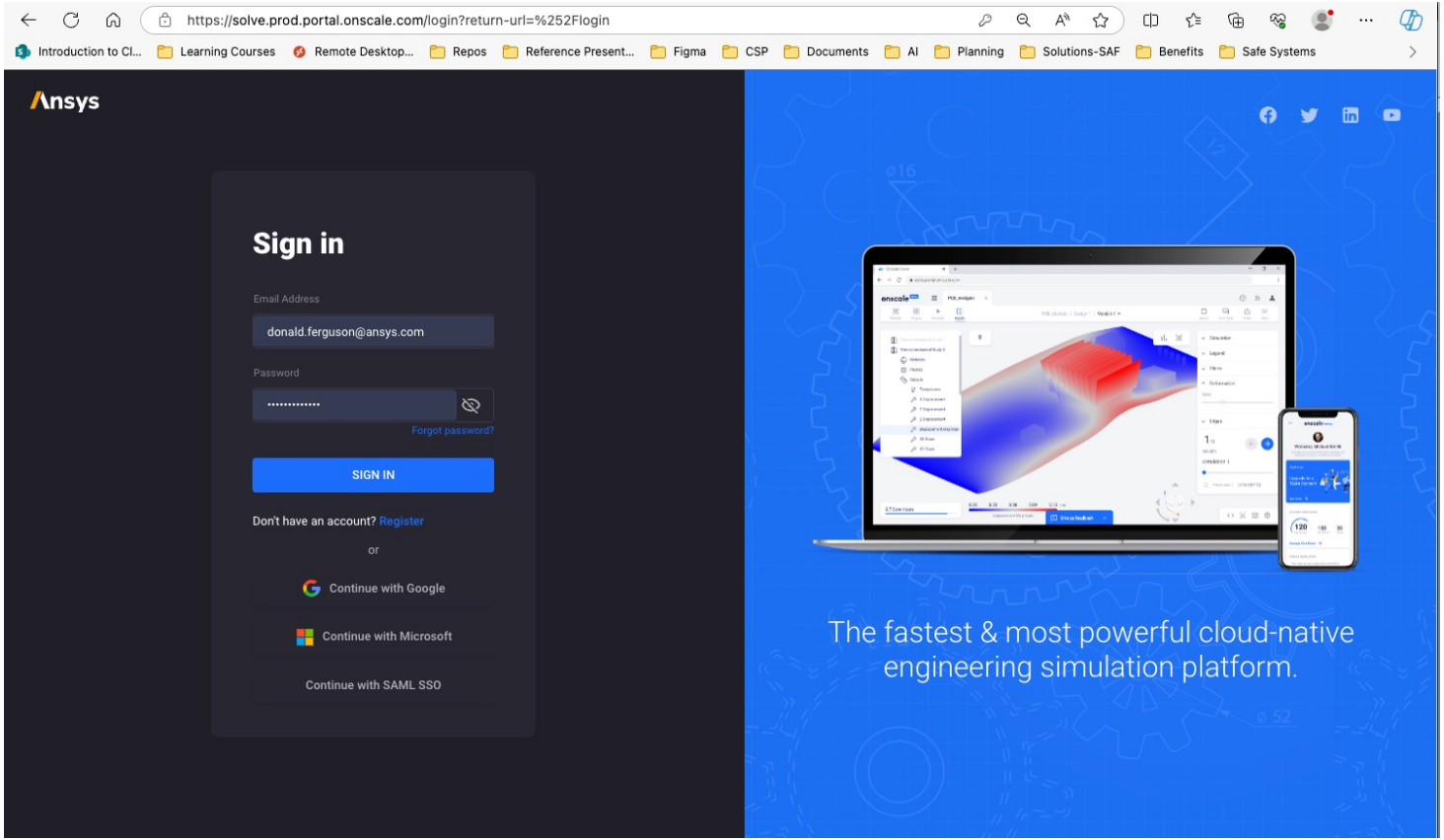
- API Management and API Gateways are fundamental concepts.
- Vastly simplifies changing microservice architectures, monitoring, enforcing security, throttling, versioning, … …
- We will do some simple stuff with a gateway, starting with security.

Login and Authorization Tokens

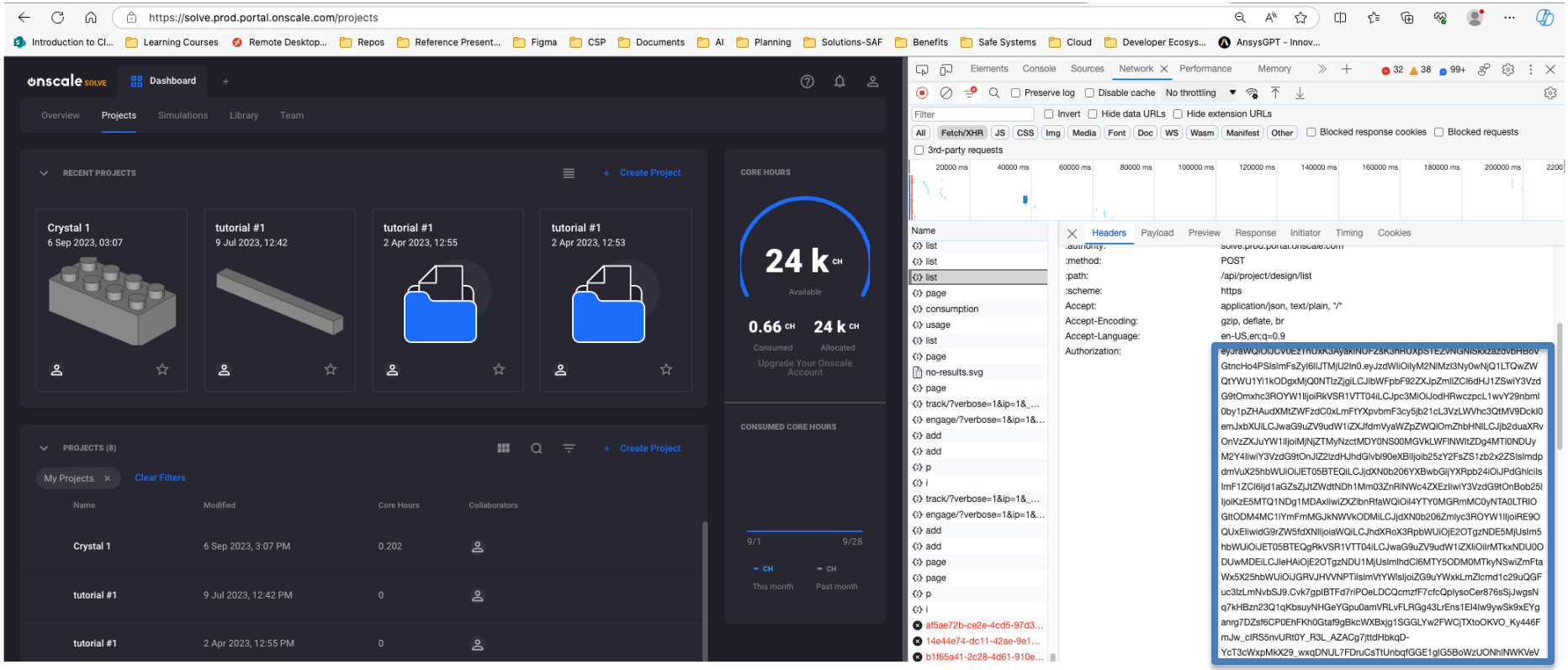


- We have seen Google Login.
- Forms the basics of authentication.
- The server applications:
  - ”Look up” user info in their database and systems.
  - Determine what operations the user can perform.
  - Generate a JWT Authorization containing signed claims.
  - Returns the token in an authorization header.
- The client
  - “Saves” the token.
  - Attached to subsequent calls.
- Servers can then authorize based on
  - Verifying the signature.
  - Checking that the client has a claim authorizing the operation.

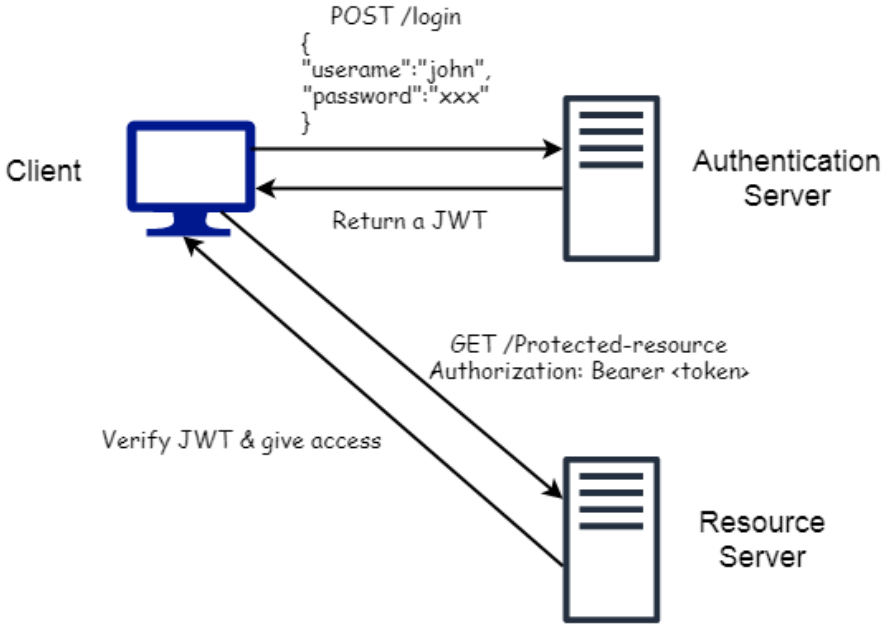
Real Example



Real Example



JWT



When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

- **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

- **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

https://medium.com/swlh/all-you-need-to-know-about-json-web-token-jwt-8a5d6131157f

REST is Stateless (https://restfulapi.net/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**.

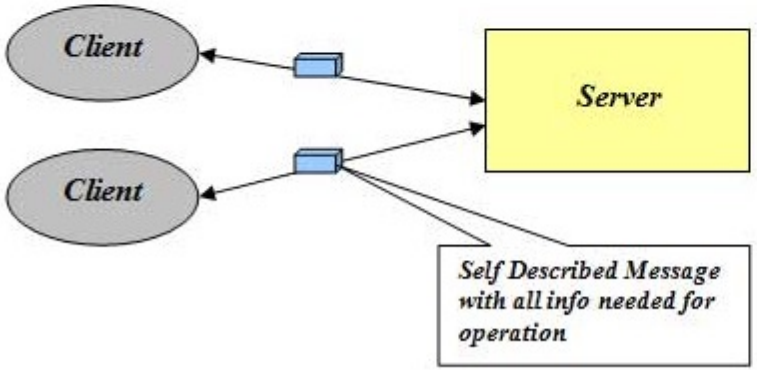
1. Statelessness helps in **scaling the APIs to millions of concurrent users** by deploying it to multiple servers. Any server can handle any request because there is no session related dependency.
2. Being stateless makes REST APIs **less complex** – by removing all server-side state synchronization 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.

Stateless

http://mrbool.com/rest-architectural-elements-and-

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.



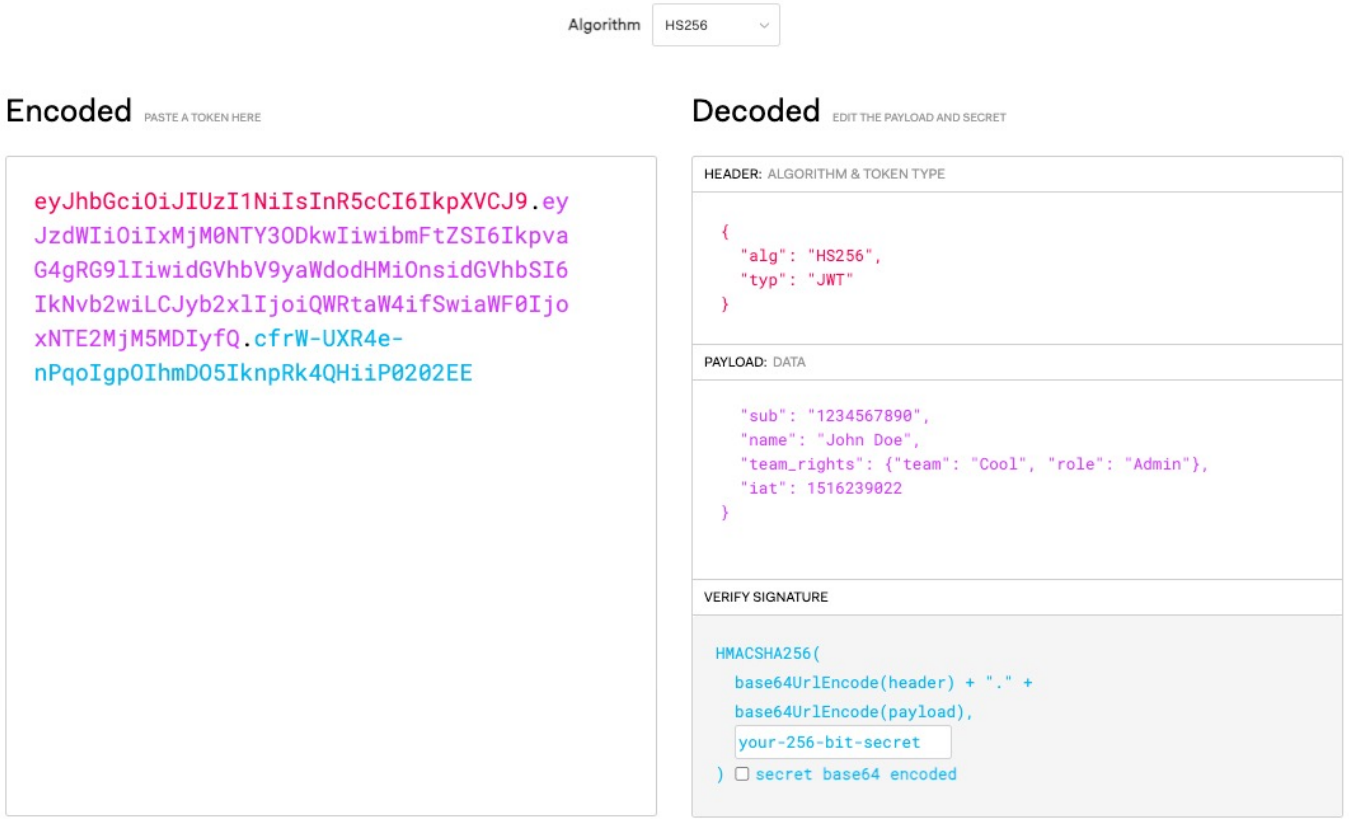
Many platforms provided server-side functions to save session/conversation/application state.

- Caching results of previous DB queries.
- Current navigation position in a complex, multi-page application.
- etc.

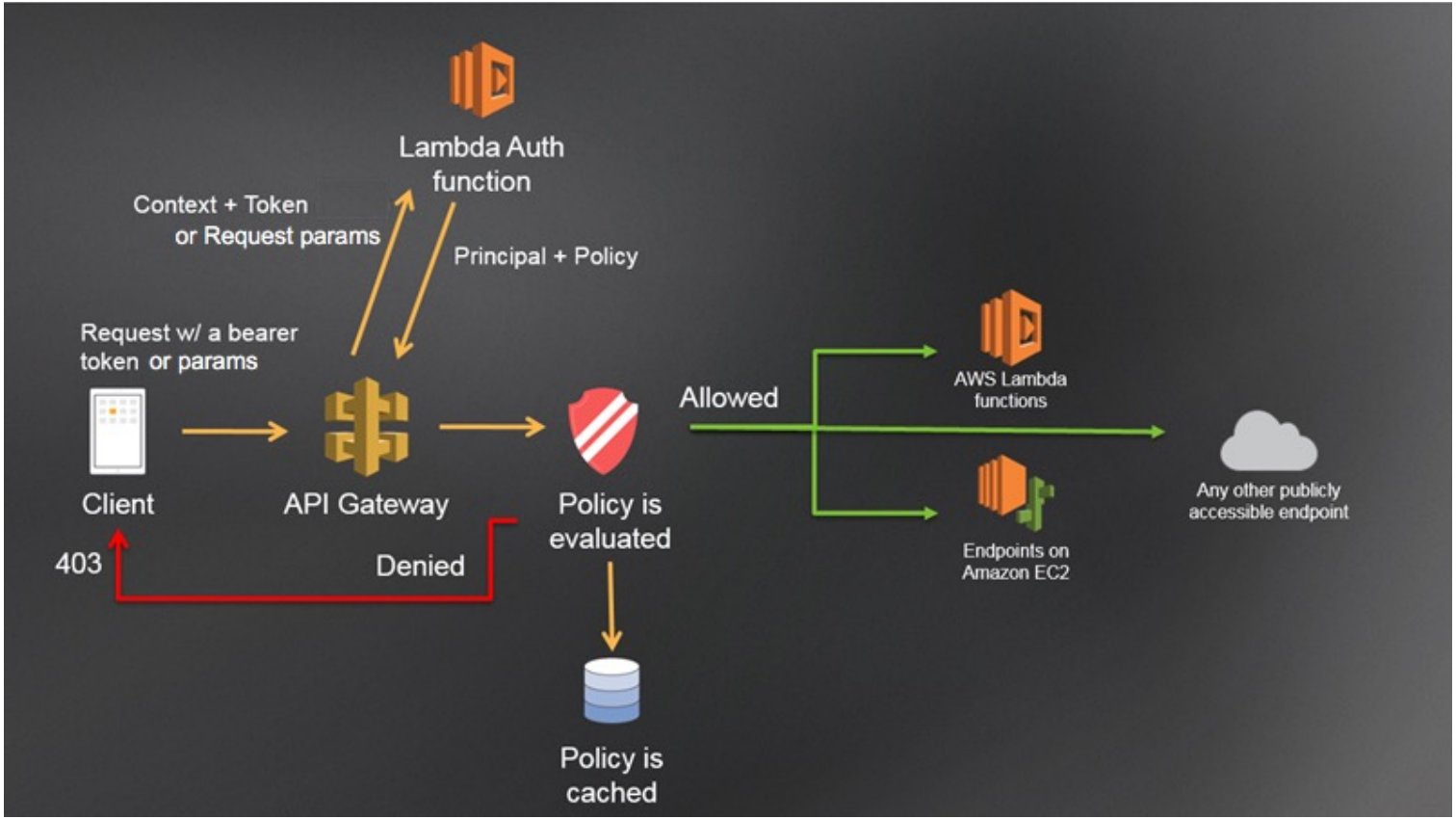
REST keeps the information in various headers or cookies. The browser re-provides the information on each request.

This is another use for JWT.

JWT.io

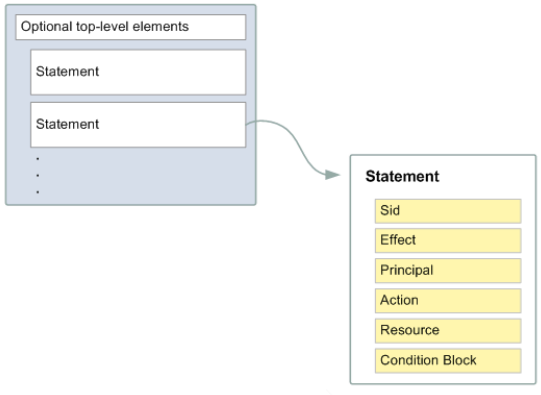


Lambda Authorizer



Example and Concepts

- Show /Users/donaldferguson/Dropbox/0-F23-Projects/courseworks-lambda/authorizer/example_authorizer.py
- The basic concepts in an authorization decision are:
  - Who is the user/identity?
  - What is the "ID" of the thing being touched?
  - What is the operation being performed?
- This explains the structure of AWS policy documents (https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html).
- Policies get attached to users, roles, groups, … …




```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1658851085578",
            "Action": [
                "s3:DeleteBucket",
                "s3:GetObject"
            ],
            "Effect": "Deny",
            "Resource": "arn:aws:s3:::iam-policy-bucket1"
        }
    ]
}
```

Pagination

Pagination

- Bad things may happen when a GET returns a huge result set.
- A *page* is a window on the large result set.
- *limit* is the no. of entries in the result set.
  - Client may specific a limit.
  - Server may have a default and/or maximum limit it allows.
- There are many design patterns for implementing pagination.
- Simplest is mirroring the SQL model:
  - *limit=xxx&offset=yyy*
  - The links section is the response should contain:
    - prev
    - next
    - current
    - May contain *first* and *last.*


```
{
  "data": [
    {
      "UNI": "dff9",
      "last_name": "Ferguson",
      "first_name": "Donald",
      "links": [
        {"rel":  "self", "href": "/api/students/dff9"},
        {"rel": "sections", "href": "/api/enrollments?uni=dff9"}
      ]
    },
    {
      "UNI": "ff11",
      "last_name": "Ferguson",
      "first_name": "Frodo",
      "links": [
        {"rel":  "self", "href": "/api/students/ff11"},
        {"rel": "sections", "href": "/api/enrollments?uni=fb11"}
      ]
    }
  ],
  "links": [
    {"rel": "current", "href": "/api/students?last_name=Ferguson&limit=2,offset=2"},
    {"rel": "prev", "href": "/api/students?last_name=Ferguson&limit=2,offset=0"},
    {"rel": "next", "href": "/api/students?last_name=Ferguson&limit=2,offset=4"}
  ]
}
```

Next Sprint –
Writeup will come this weekend
Start using some of the recent concepts.