E6156 – Topics in SW Engineering: Microservices and Cloud Applications

Lecture 6: Web API, Pub/Sub, Messaging

Donald F. Ferguson, dff@cs.columbia.edu

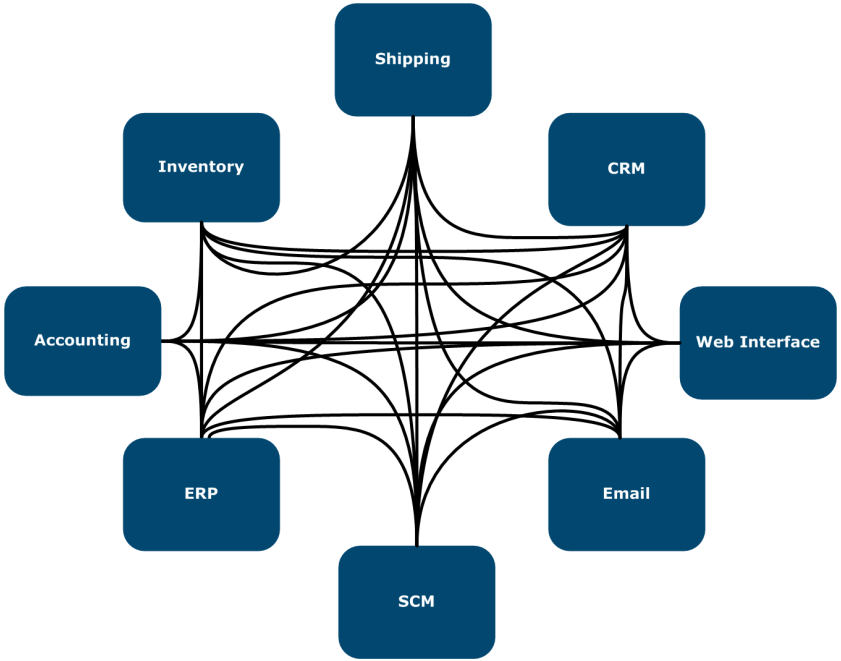
Introduction

Contents

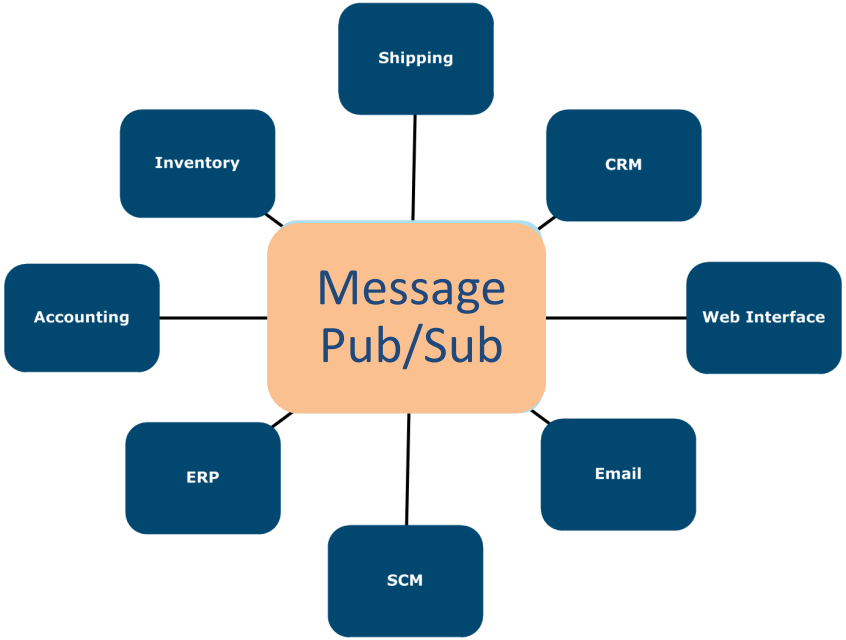
- Whiteboard discussion of:
  - Base project topics to date.
  - What we should try to accomplish in foundation to enable the simple application you are building.
- Web APIs:
  - Profile and address verification example:
    - End-to-end flow.
    - Client side
  - Stripe: Credit card payment overview (if you want to simulate it).
- Asynchronous microservice assembly:
  - Almost all programming you have done in university is tightly coupled *call-return.*
  - An extremely, if not most common approach, to coupling systems is loosely-coupled:
    - Publish/Subscribe
    - Message driven queuing/message driven processing
  - We will get a feel for pub/sub by including in email verification.

Integration/Assembly Patterns

Classic Approach



Pub/Sub or Message Hub

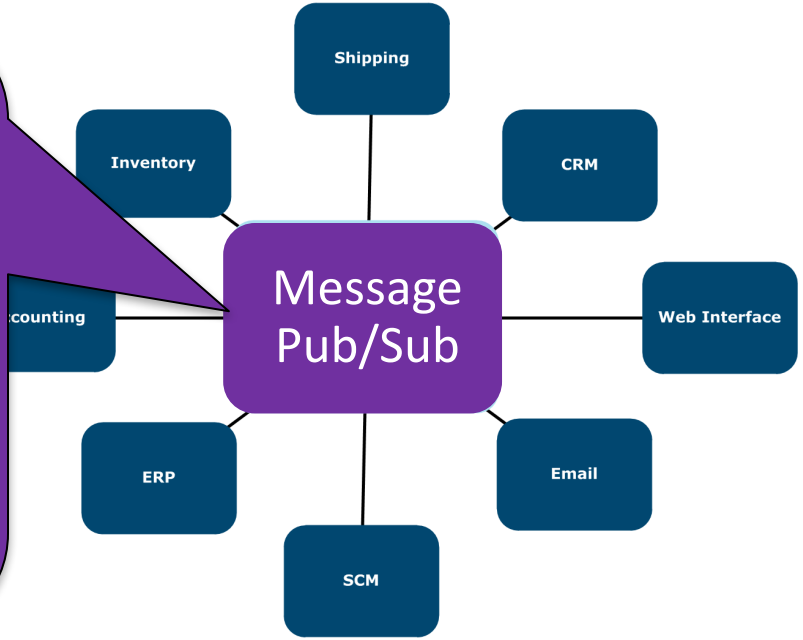


Integration/Assembly Patterns

Classic Approach

- The logical model is a “hub” supporting
  - Topics
  - Queues
- The physical model is a robust, multi-system, highly available and high performance infrastructure

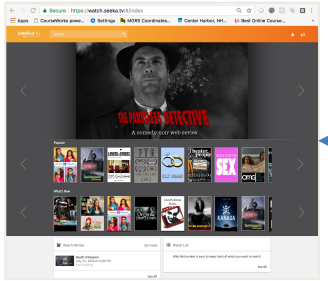
Pub/Sub or Message Hub



Web APIs
API Economy

Concept

Extensive Use of Web APIs



stripe Payment

MailChimp

brightcove

Auth0
The new way to solve Identity

Seeka.TV
AWS Application

Uploadcare

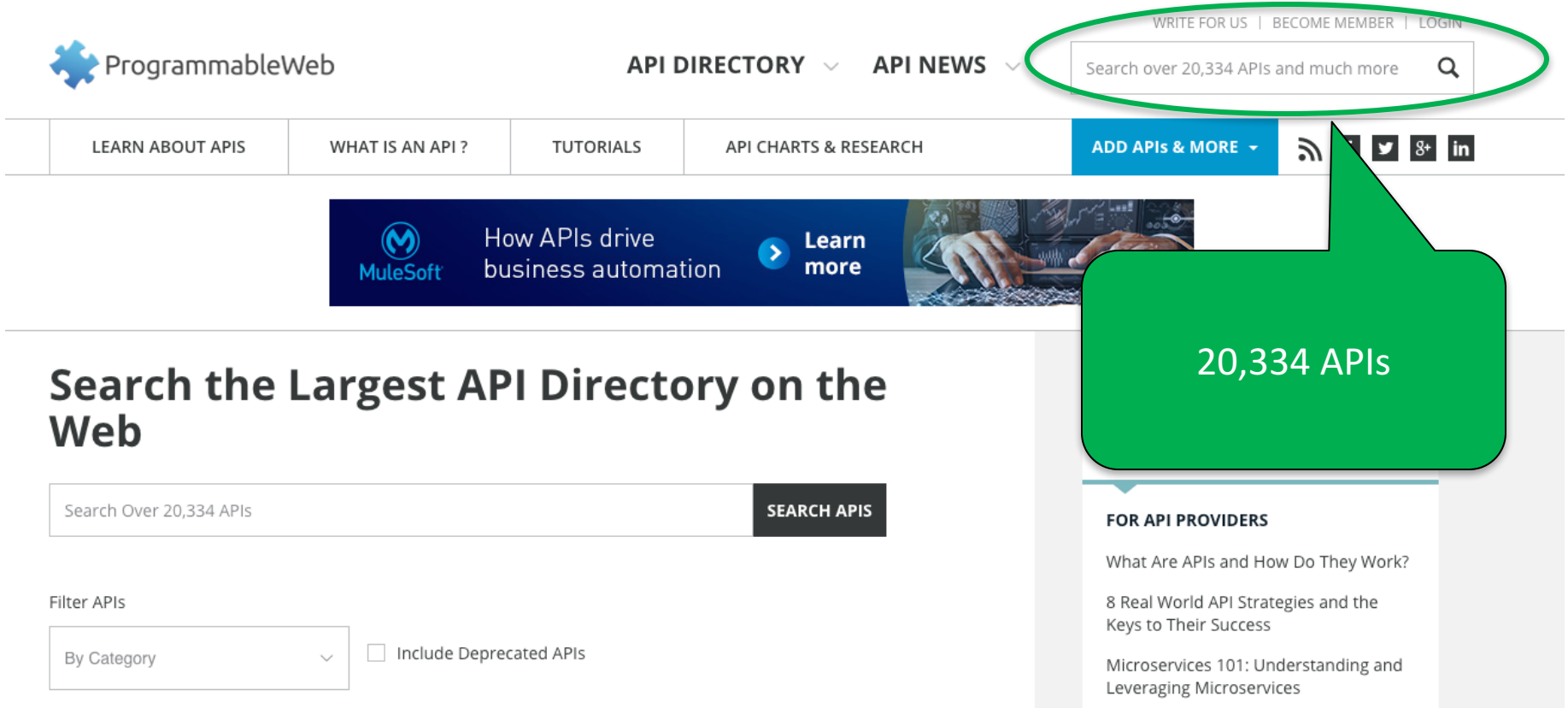
GRAPHENEDB

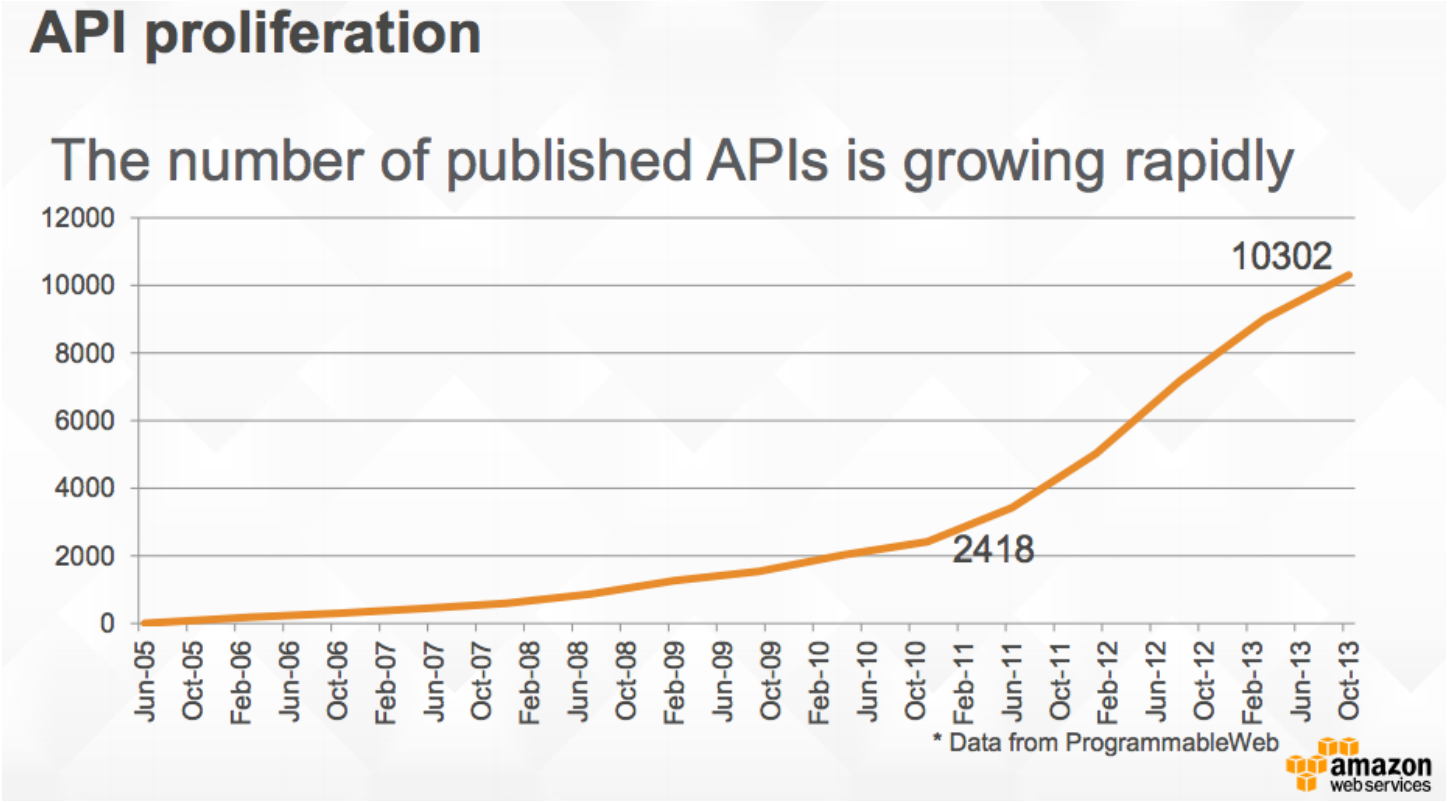
twitter

YouTube

facebook

Programmable Web – An Example




API proliferation
The number of published APIs is growing rapidly
12000
10000
8000
6000
4000
2000
0
10302
2418
Jun-05
Oct-05
Feb-06
Jun-06
Oct-06
Feb-07
Jun-07
Oct-07
Feb-08
Jun-08
Oct-08
Feb-09
Jun-09
Oct-09
Feb-10
Jun-10
Oct-10
Feb-11
Jun-11
Oct-11
Feb-12
Jun-12
Oct-12
Feb-13
Jun-13
Oct-13
* Data from ProgrammableWeb
amazon web services

Goals of API Design

Nevertheless, many API programs are growing out of previous SOA initiatives. Web services focused on internal or partner integrations are being opened up to developers—both within and outside the enterprise. During this process, it is important for API designers to remember that an API program has drivers and requirements quite different from the ones that initially led enterprises to open their IT assets via Web services.

With this in mind, the broad goals of API design in general can be defined as:

- Enabling self-service for app developers and app users alike
- Reducing barriers to accessing valuable enterprise resources
- Prioritizing the needs and preferences of client app developers
- Encouraging collaboration between and among internal and external resources
- Addressing the security and scaling issues of exposing IT assets to the open market

Above all, API design must be focused on maximizing the business value of the interface. In part two, we will take a closer look at how APIs add value to the business.

API Economy (http://www.gartner.com/smarterwithgartner/welcome-to-the-api-economy/)

Smarter With Gartner

> The API economy is an enabler for turning a business or organization into a platform.

We live in an API economy, a set of business models and channels based on secure access of functionality and exchange of data. APIs make it easier to integrate and connect people, places, systems, data, things and algorithms, create new user experiences, share data and information, authenticate people and things, enable transactions and algorithms, leverage third-party algorithms, and create new product/services and business models.

"The API economy is an enabler for turning a business or organization into a platform." said Kristin R. Moyer, vice president and distinguished analyst at Gartner. "Platforms multiply value creation because they enable business ecosystems inside and outside of the enterprise to consummate matches among users and facilitate the creation and/or exchange of goods, services and social currency so that all participants are able to capture value."

Uber, for instance, is an example of a business built on a platform because it leverages Google Maps through an API to enable its entire business model of matching drivers who have a vehicle with passengers who need a ride. Walgreens offers an API for its in-store photo printing services that enables others to offer photo apps on its platform. It moves from being a photo printer to being a photo platform.

2017 Is Quickly Becoming The Year Of The API Economy

https://www.forbes.com/sites/louiscolumbus/2017/01/29/2017-is-quickly-becoming-the-year-of-the-api-economy/#4b3806fe6a41

Louis Columbus, CONTRIBUTOR
FULL BIO
Opinions expressed by Forbes Contributors are their own.

This year more CIOs will have their bonuses tied to how many new business models they help create with existing and planned IT platforms than ever before. This trend will accelerate over the next three years. CIOs and IT staffs need to start thinking about how they can become business strategists first, technicians and enablers of IT second. CIOs must create and launch new business models faster to keep their companies competitive. APIs are the fuel helping to make this happen.

The Urgency To Create New Business Models Is Driving API Proliferation

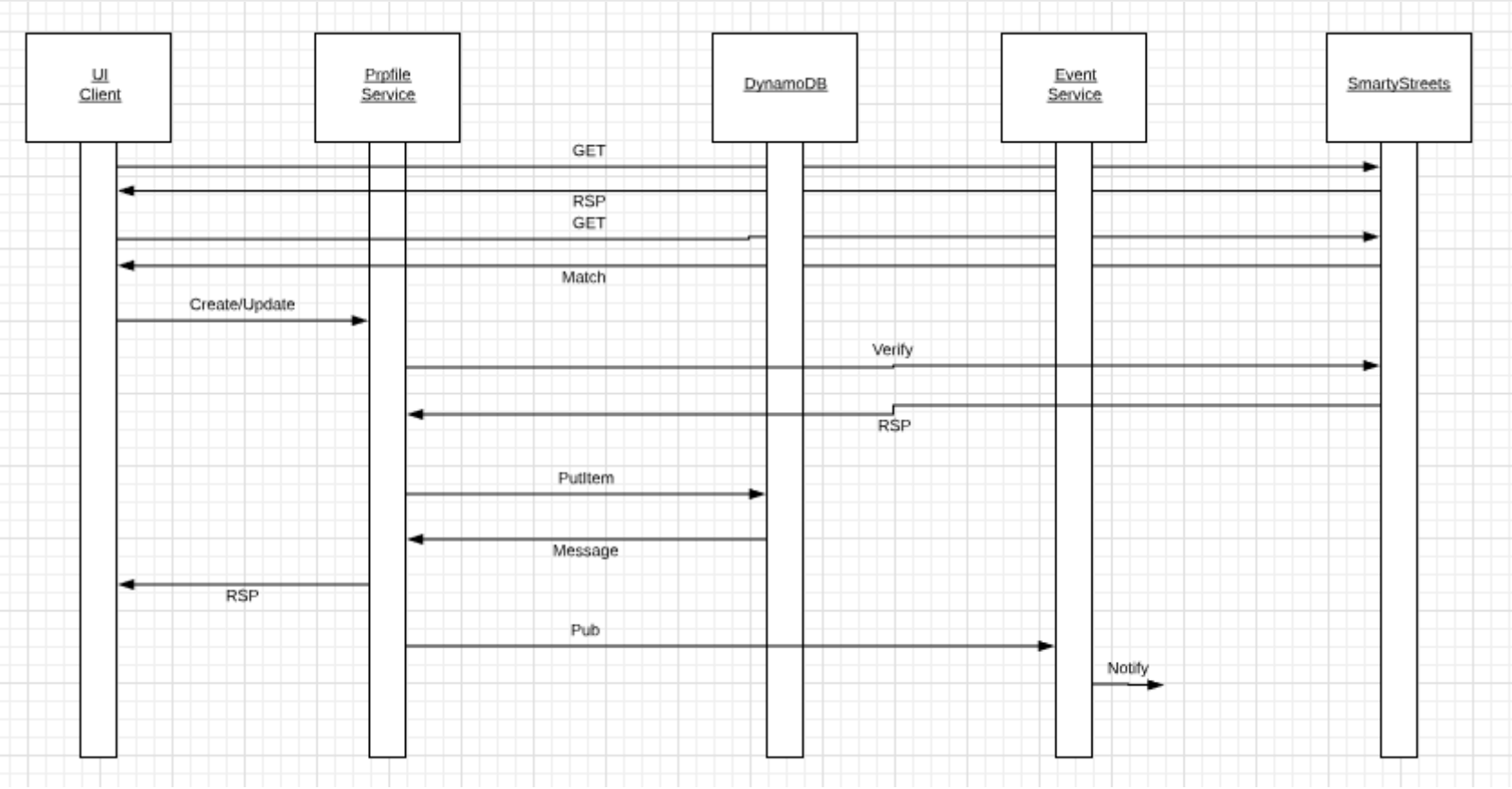
APIs (Application Programmer Interfaces) are the components that enable diverse platforms, apps, and systems to connect and share data with each other. Think of APIs as a set of software modules, tools, and protocols that enable two or more platforms, systems and most commonly, applications to communicate with each other and initiate tasks or processes. APIs are essential for defining and customizing Graphical User Interfaces (GUIs) too. Cloud platform providers all have extensive APIs defined and work in close collaboration with development partners to fine-tune app performance using them. Amazon Web Services, Facebook, Google, Marketo, Salesforce, SAP Hybris, Twitter and thousands of other companies have APIs available. As of today, the Programmable Web lists 16,590 APIs in its database.

APIs

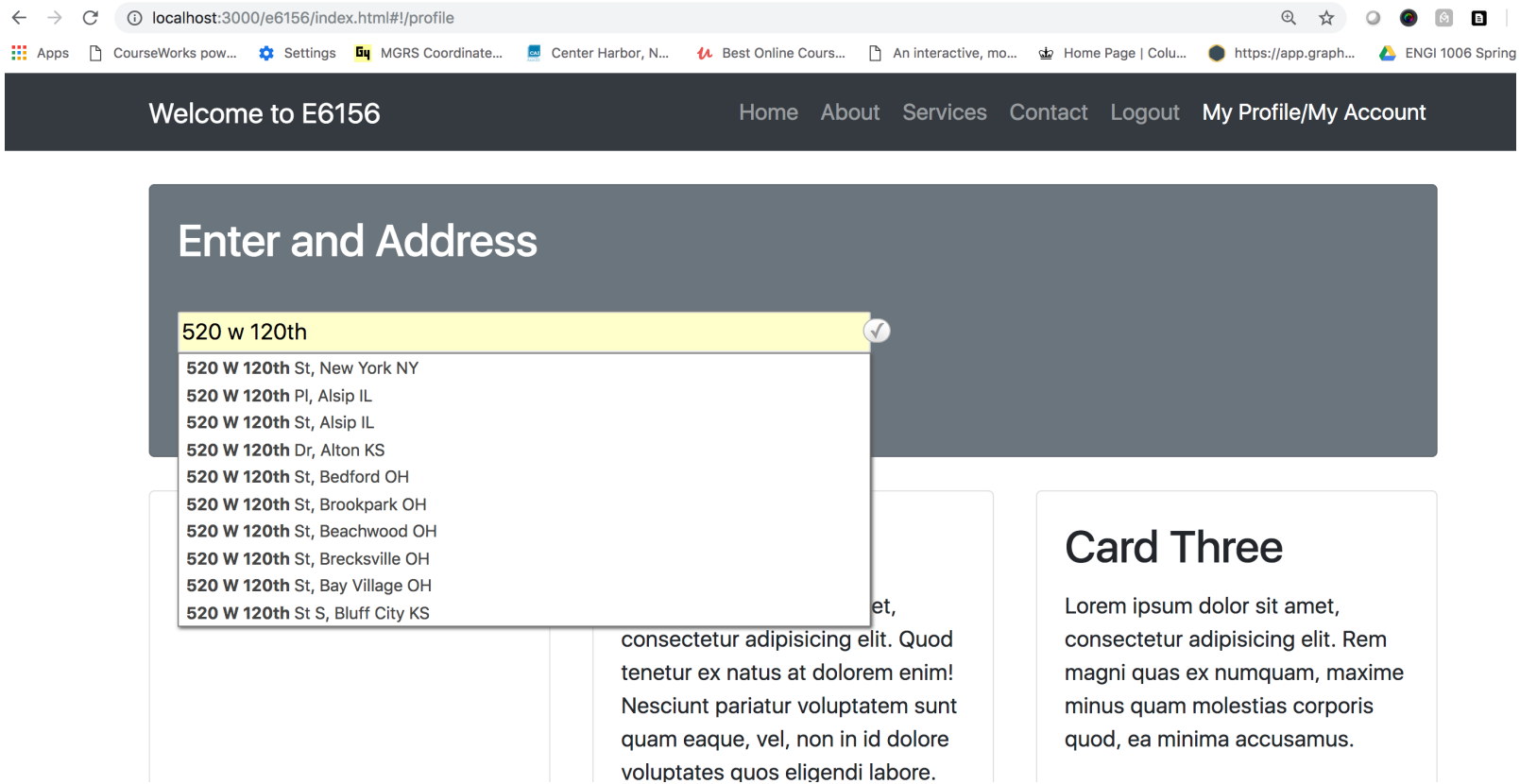
- A large part of developing microservices and cloud applications is determining
  - Which APIs do I want to publish to 3rd parties or unanticpated calling applications.
  - Which 3rd party/cloud APIs do I want to call instead of writing the equivalent function?
  - Only implement a microservice if:
    - You cannot find a web API that meets your needs.
    - Your implementation will be differentiated and core to your business model.
- This is why I keep harping on the fact that you should not assume your UI is the only thing calling your APIs.
- We will do one example in this course, SmartyStreets, and I can show one other, Stripe.

SmartyStreets

SmartyStreets Sequence



SmartyStreets Demo(s) – UI



UI – HTML/JavaScript

```
<!-- Call to Action Well -->
<div class="card text-white bg-secondary my-4">
    <div class="card-body">
        <h2>Enter and Address</h2>
        <form>
            <br>
                <input type="text" id="newaddress" name="newaddress" placeholder="{{placeholder}}"  style="...">
            <br>
        </form>
        <p></p>
        <div class="dropdown">
            <button class="btn  btn-primary dropdown-toggle"
                    type="button" id="dropdownMenuButton" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false" style="..." >
                Choose the type of address
            </button>
            <div class="dropdown-menu" aria-labelledby="dropdownMenuButton">
                <a ng-click="addressKind($index)" ng-repeat="k in addressKinds" class="dropdown-item">{{k}}</a>
            </div>
        </div>
    </div>
</div>
```

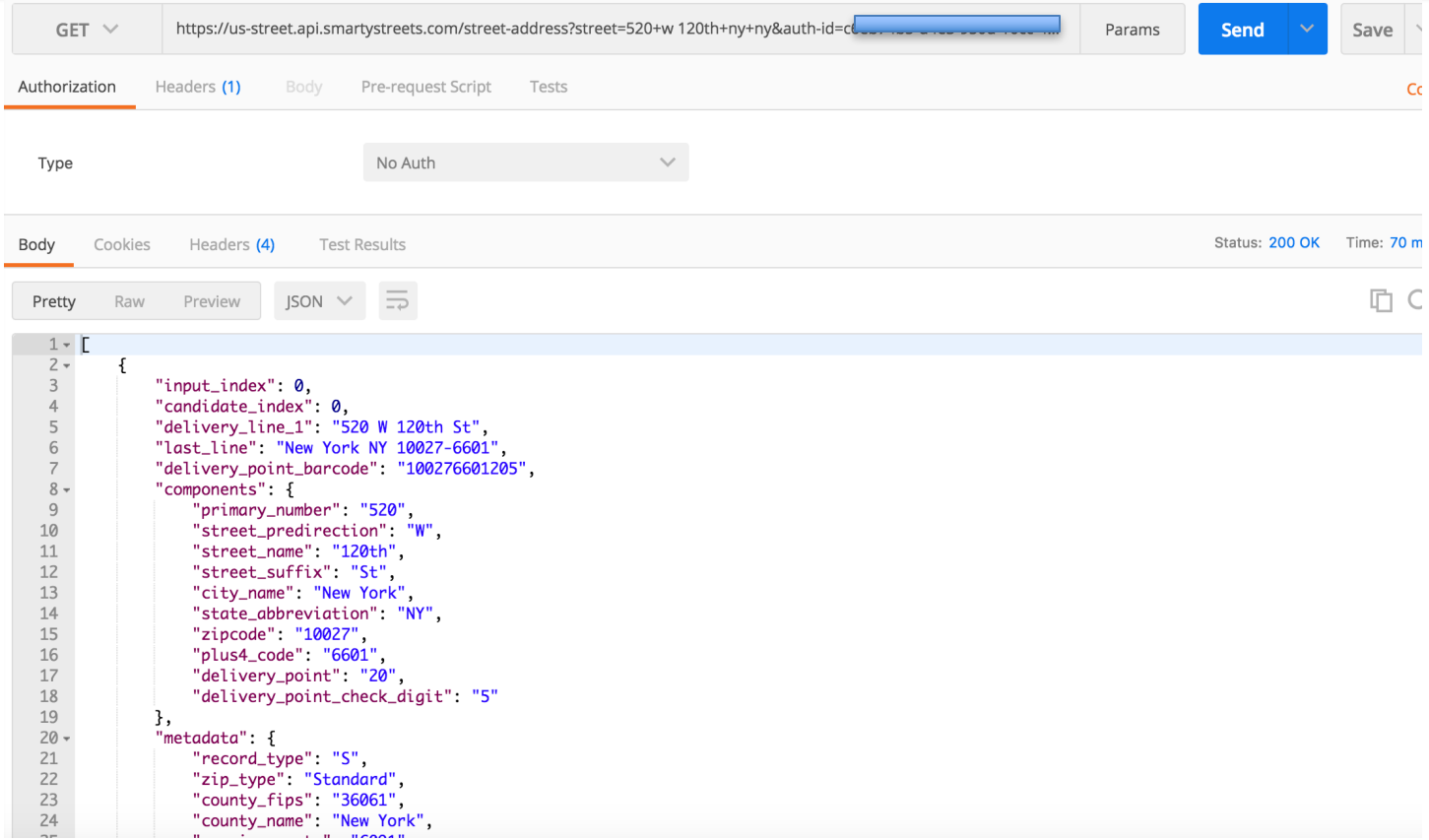
UI JavaScript

Instead of ”Boo Yah”

- POST to profile endpoint
- Address info.
- Address type.

```
CustomerApp.controller("ProfileController", function($scope, $http, $location, $window) {

    console.log("Profile controller loaded.")

    let s3 = jQuery.LiveAddress({
        key: "18981749384552786",
        waitForStreet: true,
        debug: true,
        target: "US",
        placeholder: "Enter address",
        addresses: [{
            freeform: '#newaddress'
        }]
    });

    s3.on("AddressAccepted", function(event, data, previousHandler)
    {
        console.log("Boo Yah!")

    });

    $scope.placeholder = "enter an address and select a choice."

    $scope.addressKinds = ['Home', 'Work', 'Other']

    $scope.addressKind = function(idx) {
        console.log("Address kink = " + $scope.addressKinds[idx]);
    };

});
```

SmartyStreets – Microservice Verification Call

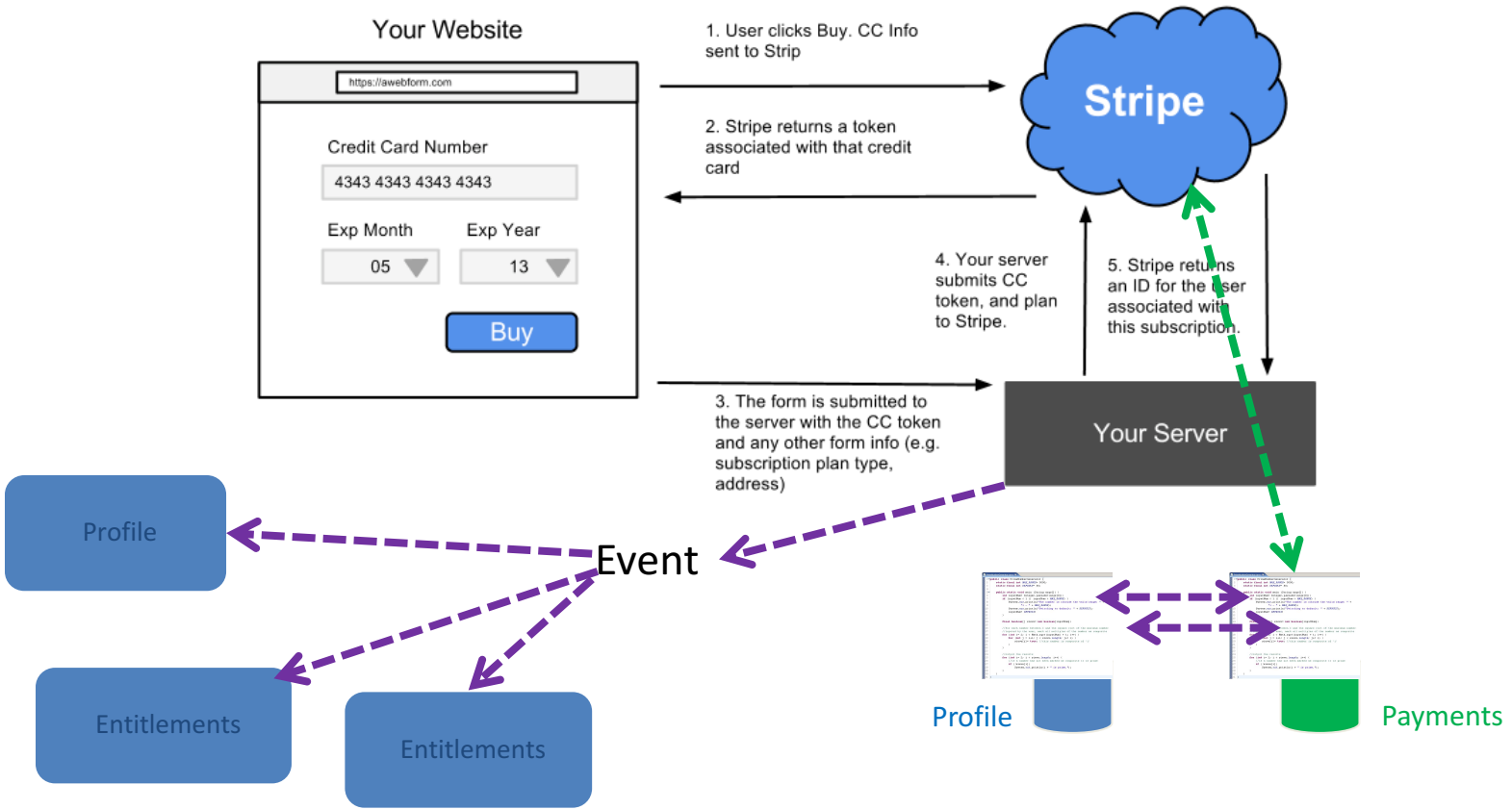


Example Code

```
auth_id = "c6eb7[redacted]cc-445f305eaf32"
auth_token = "wY[redacted]YM"
base_url = "https://us-street.api.smartystreets.com/street-address"

def check_address(street_no, street_name, town, state):

    get_url = base_url + "?street="
    added = False
    if street_no:
        added = True
        get_url +=  str(street_no)
    if street_name:
        if added:
            get_url += "+"
        get_url += street_name
    if town:
        if added:
            get_url += "+"
        get_url += town
    if state:
        if added:
            get_url += "+"
        get_url += state

    if added:
        get_url += "&auth-id="+auth_id+"&auth-token="+auth_token

        print("URL = ", get_url)
        r = requests.get(get_url)
        return r
```

Stripe

Stripe



Publish/Subscribe

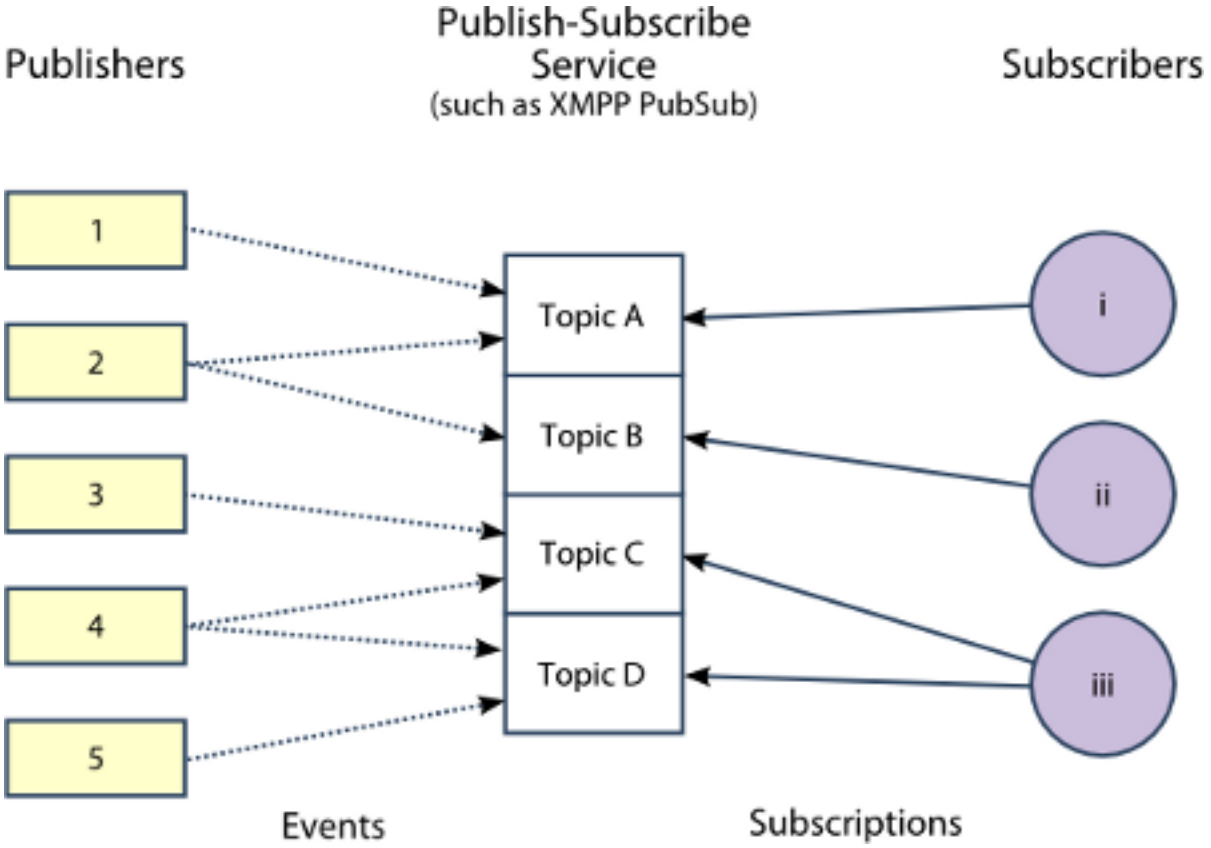
Publish/Subscribe (Wikipedia)

"In software architecture, **publish–subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes 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."

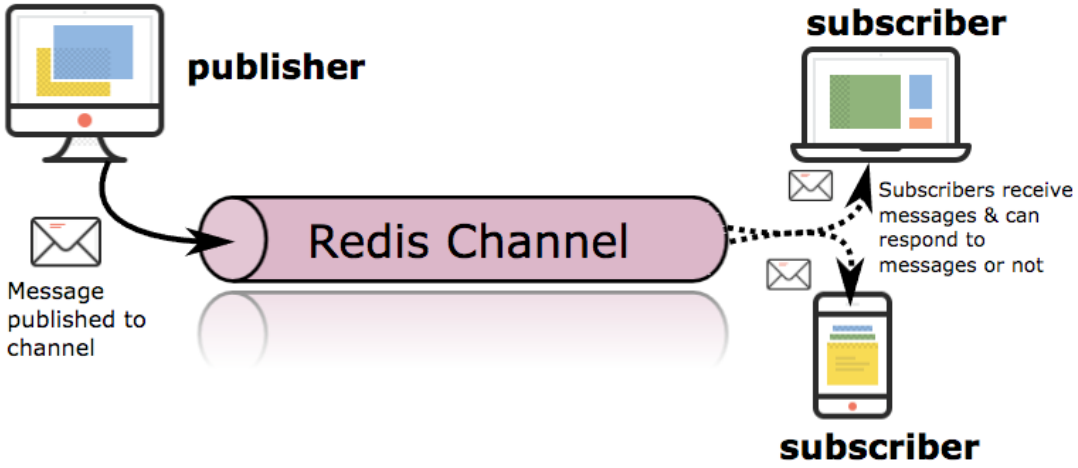
Publish/Subscribe



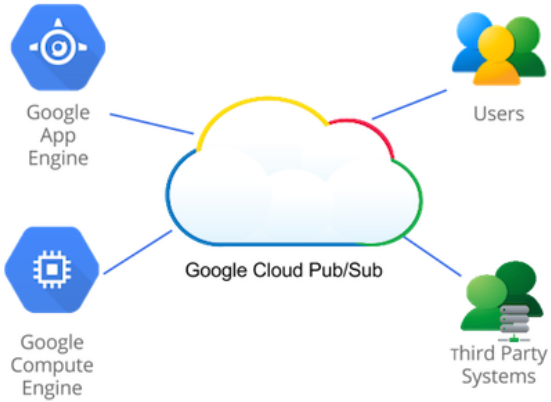
Redis Example

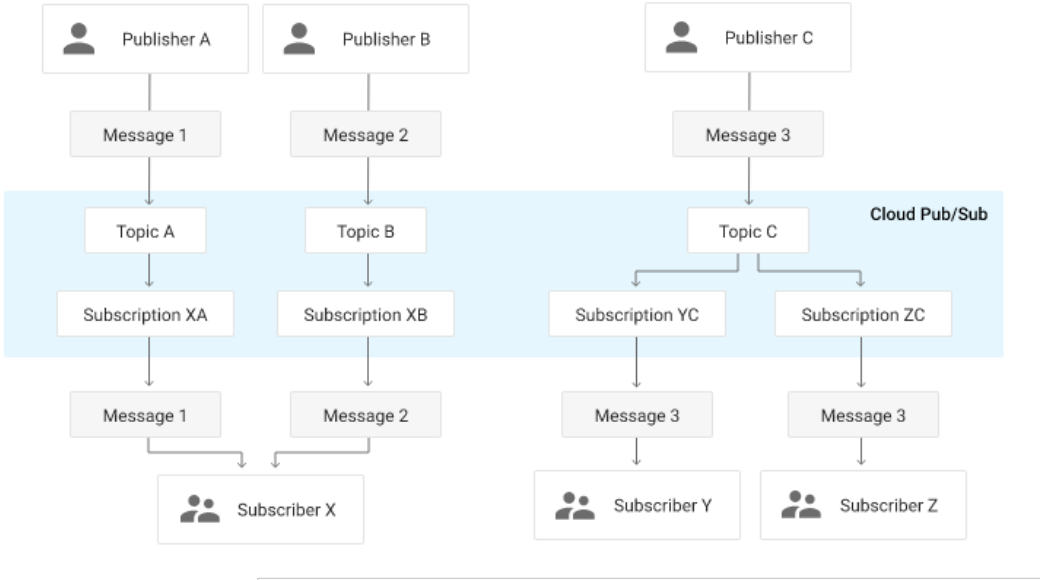
Pub/Sub Messaging Pattern

Redis Pub/Sub uses a message passing system that message senders - called **publishers** - post a message to a channel that the message receivers - called **subscribers** - can respond to messages without either the publishers or subscribers knowing any details about each other.



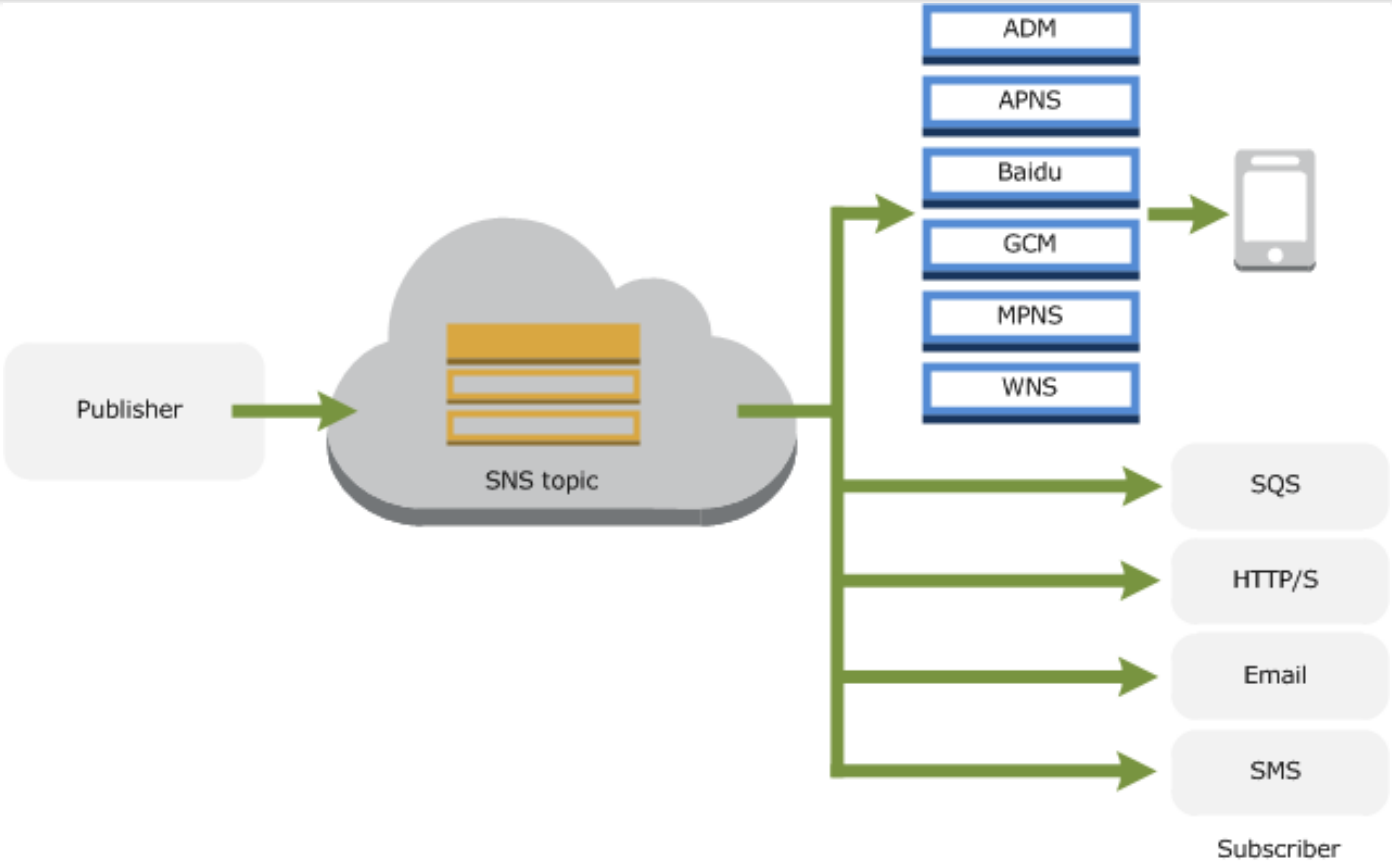
Google Cloud Pub/Sub





https://cloud.google.com/pubsub/overview

SNS



Publish/Subscribe

Message Queueing (Wikipedia)

“Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. Message queues have implicit or explicit limits on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue.

… …

There are often numerous options as to the exact semantics of message passing, including:

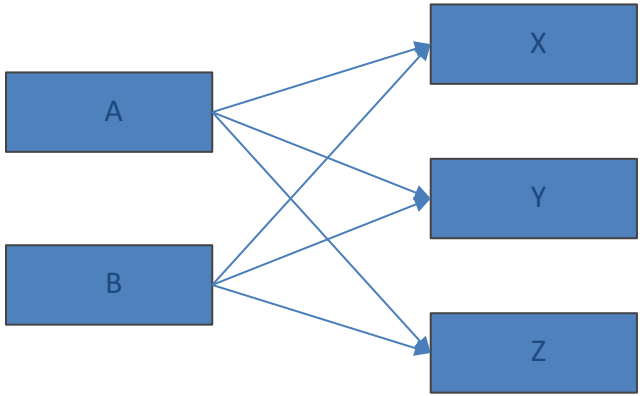
- Durability - messages may be kept in memory, written to disk, or even committed to a DBMS if the need for reliability indicates a more resource-intensive solution.
- Security policies - which applications should have access to these messages?
- Message purging policies - queues or messages may have a "time to live"
- Message filtering - some systems support filtering data so that a subscriber may only see messages matching some pre-specified criteria of interest
- Delivery policies - do we need to guarantee that a message is delivered at least once, or no more than once?
- Routing policies - in a system with many queue servers, what servers should receive a message or a queue's messages?
- Batching policies - should messages be delivered immediately? Or should the system wait a bit and try to deliver many messages at once?
- Queuing criteria - when should a message be considered "enqueued"? When one queue has it? Or when it has been forwarded to at least one remote queue? Or to all queues?
- Receipt notification - A publisher may need to know when some or all subscribers have received a message.

Message Queue Service Pattern

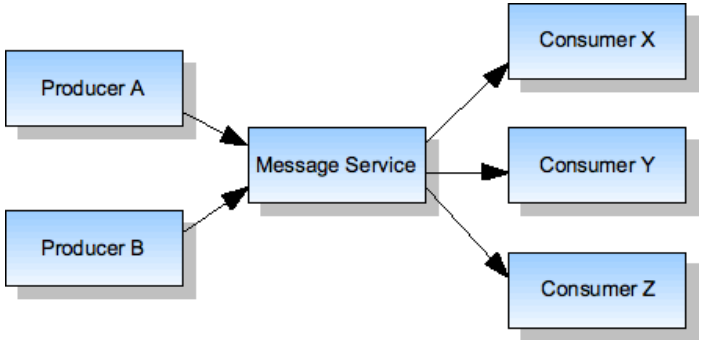
Point-to-Point communication between modules becomes fragile at scale (no. of messages, no. of particpipants)

- Adding modules P and Q requires finding and configuring all senders.
- I want to send M to any one of X1, X2 and X3, but only one.
- I want to make sure that someone processes M but do not want to hold the transaction until I get a response (I may not even need the response).
- My destinations are not “up” at the same times I am.

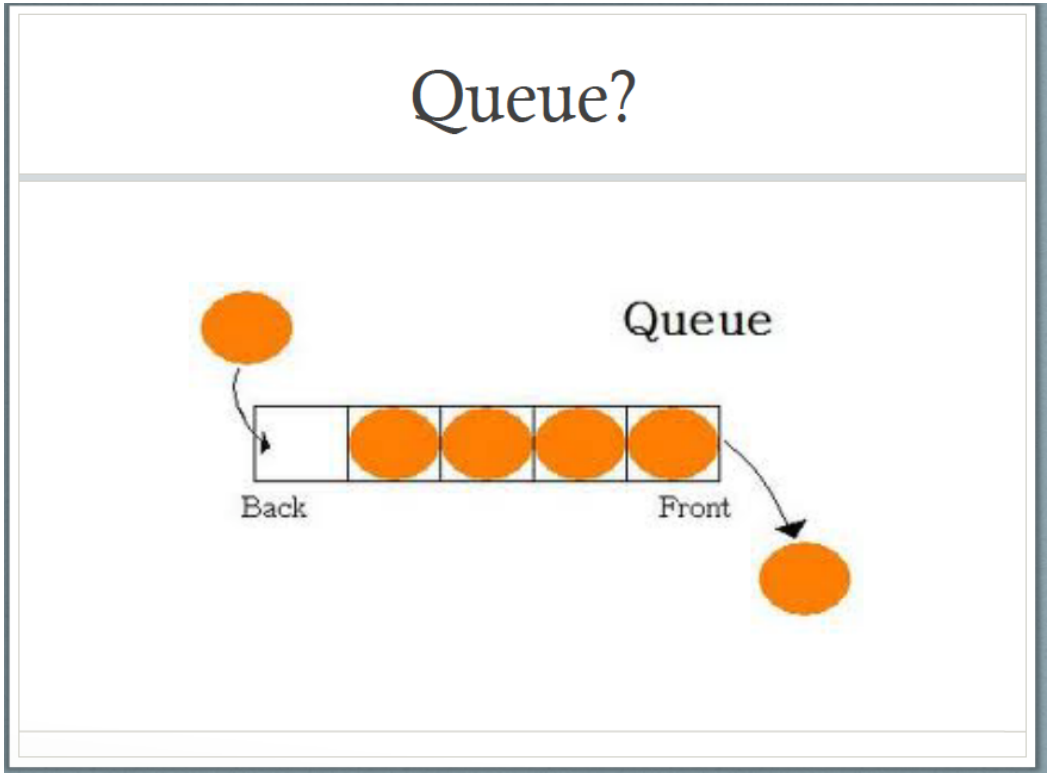
Anti-Pattern



Best Practice Pattern



SQS



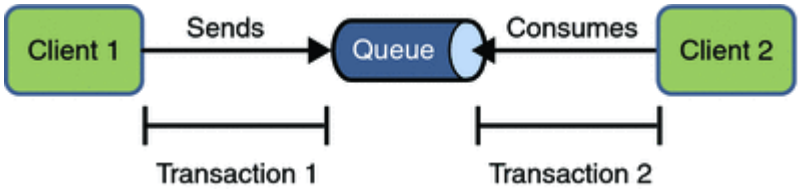
Queue Operations

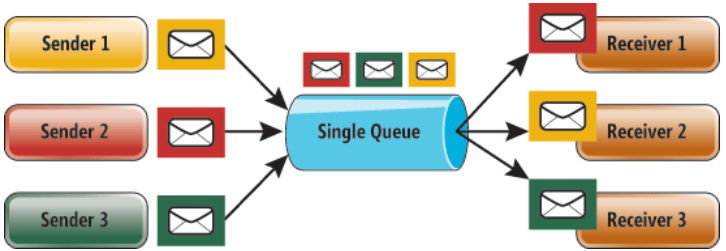
- Create Q
- Delete Q
- Send Message
- Receive Message
- Delete Message

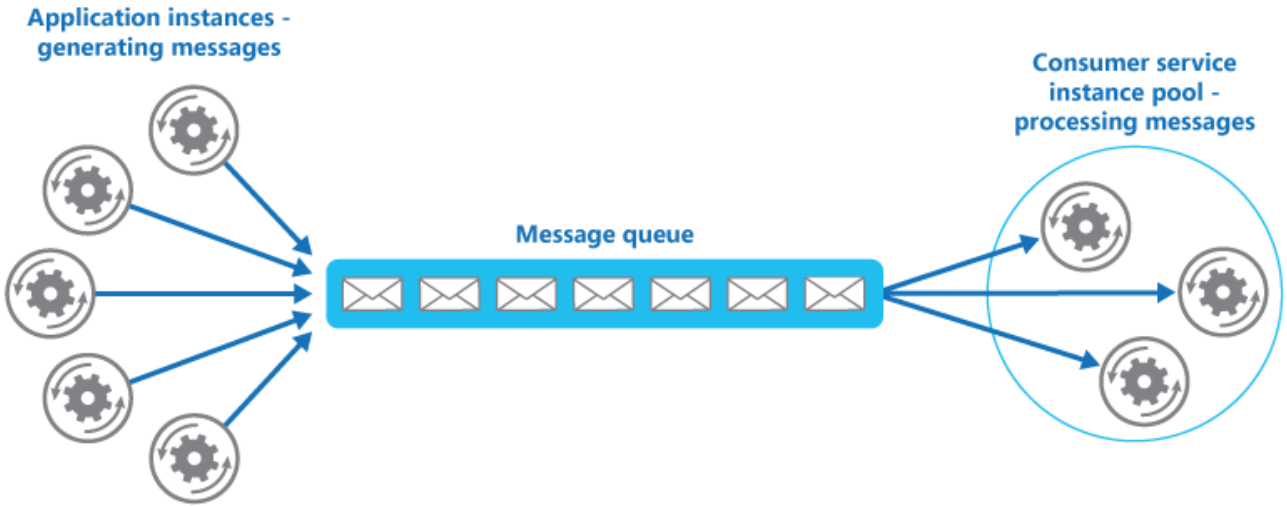
FIFO Semantics

- Strict, limited scalability.
- Best effort, highly scalable.
- Transactional

Message Exchange Patterns







Why Would I Use a Queue?

http://www.javacodegeeks.com/2013/06/working-with-amazon-simple-queue-service-using-java.html

- Flow control
  - My server may be able to process 100 reqs/second
  - There are thousands of clients
  - The aggregate rate could be 10 req/s or 300 req/s
  - Without a queue, clients would get connect failures under load
- The request is going to take a long time
  - Image reformatting and tagging
  - I cannot have my code “wait” for a response
  - And I do not want to write a lot of “Is it done yet?” calls
- I do not know (or care) which one implements the logic
  - I have 5 – 10 image processors for GIFs
  - I have 7 – 10 processors for .wav files
  - I do not want to know which ones are active, when and processing what.
  - I just want to put the “thing in a queue” knowing that someone will eventually pick it up.
- Think “email” for API calls

Amazon SQS – Example