* Rest vs Soap
  + Representational State Transfer
  + Richardson Maturity Model
    - The steps towards becoming Restful
    - Level 0 - You have one endpoint that processes every single request in your web service, we are using http
    - Level 1 - We make resources - we have many different endpoints for different individual resources ShoppingCart, Item, User, Order
    - Level 2 - use different HTTP verbs to define different actions for interacting with the resources
      * Some HTTP verbs have different data safety levels
      * Safe - Get, Option - no matter what will never change the data in the system
      * Idempotent - Put, Patch, Delete - Change data, but multiple copies of the same request change data to the same state
      * Dangerous - Post - Every time we send post we make something new
    - Level 3 - Introduce HATEOS - Hypertext as the engine of application state
      * When interacting with a resource, it should also tell you the next endpoints and verbs for doing the process
    - <https://martinfowler.com/articles/richardsonMaturityModel.html>
  + 6 principles of Rest
    - Stateless
      * Why? We need to be able to do horizontal scaling. Once we have multiple copies of the server, the client should be able to be directed to any copy at any time.
      * If the server has state, this means the state must be replicated on every copy, which means pretty soon, the only thing the server has memory for is just state
      * We make the client send in any relevant state with every request
      * JWT
    - Uniform Interface
      * Everyone everywhere using everything interacts with the api in the same way
      * The interface should be internally consistent
      * We should be HATEOS
    - Client-Server
      * Client and server should be able to grow and change independent of each other
      * Client only knows resource endpoints and uses those for everything
    - Layered System
      * When a client sends a request, that request can be processed by many different servers, running different technologies and the client will be none the wiser
    - Cacheable
      * Whenever possible allow resources to be cacheable and declare that they are cacheable
      * It means we can save the data somewhere as the api assures us that the data won’t change frequently
    - (optional) Code on Demand
      * Mostly we return data in a format like JSON
      * We can also return executable code for a client to use to render something
  + What is soap?
    - Simple Object Access Protocol
    - Competitor to REST
    - We could use with any transfer protocol, http, ftp, smtp,
    - We always had to build an XML Envelope that described our request and how it functioned
    - The server made an XML WSDL- Web Service Definition Language
      * Defines every endpoint and what you can do at those endpoints + how to access
    - For a while SOAP was better because soap transmissions could be encrypted
    - But then HTTPs comes around and http can be encrypted as well, and nowadays soap is pretty dead
* Microservices
  + What is this design pattern
    - Idea where we break up our Monolithic Server into smaller discrete servers that each handle a particular part of our service as a whole
  + Benefits
    - Loosely Couple Servers
    - Independent Scaling of Servers - most cost efficient
    - No single point of failure - great for users
    - Easier to try new technologies and new techniques
    - Every service has its own DB - speed up DB access times
  + Downsides
    - No longer have referential integrity across our microservices
    - No longer have acid transactions across our microservices
    - More Complexity
  + What are some of the necessary pieces
    - Gateway
    - ServiceRegistry
    - Intercommunication
    - CircuitBreaking
* Netflix OSS stack
  + They made all of these tools for their internal use, and released those tools and made them open source
  + Nowadays they don’t use this stack a lot
  + Who manages it now?
    - Spring Cloud
  + How do we utilize it
    - In Spring boot, all we need to do is download the dependency, add an annotation and some properties to the application.yaml
* Eureka
  + Annotation and setup
    - Dependencies - Eureka Server ( spring cloud )
    - @EnableEurekaServer
    - @EnableDiscoveryClient
  + Discovery
    - Process where services can search for the eureka client and register themselves with it
  + Heartbeat
    - Every service will send a heartbeat message to eureka telling eureka it is alive
    - We can configure that message and specify how often we send it and how long eureka will wait before calling us dead
* Actuators
  + These are prebuilt endpoints for spring boot applications, that allow for metadata on the server and interactions with the server itself
  + Ie: Info, restart, shutdown, beans, health
  + You should always have these
* Zuul
  + Annotation and setup
    - @EnableZuulProxy
    - @EnableDiscoveryClient
  + Path Proxy
    - In yaml set up path pattern matchers to service ids
    - Strip prefix - true if we added a prefix to the path from the client for requests, otherwise false
  + Ribbon
    - Loadbalancer that we normally put on zuul
    - Default balancing strategy is round robin, so everyone gets one request before anyone gets a second
* FeignClient
  + Annotations and setup
    - @EnableFeignClient on main
    - @FeignClient above an interface - name of the service it is interacting with
  + How to send requests to other services
    - In the interface make methods that match controller methods in that other service
    - Feign builds those implementations
* Hysterix
  + Annotations and setup
    - @EnableCircuitBreaker
    - Add feign.hysterix.enabled true in yaml
    - In @FeignClient fallback to the class
  + How does it interact with FeignClient
    - Feign client uses it as a fallback if the request that it send fails for some reason
  + Why is it important to have
    - It allows our services to return responses independent of each other, so that way the user always gets whatever data is available even if somethings are crashed
    - Makes sure we remove any single point of failure
* Spring Cloud Config
  + Annotations and setup
    - Config Server
    - @EnableConfigServer
    - Set up a git repo that has all of the different services yaml configurations
  + Why is this useful
    - It keeps all config in one location for easy management
    - It offers remote live service config reload
      * I can change the cloud config repo files and it will trigger a reload of properties for all of the services that rely on it
* BASE
  + What happens to our database consistency and transactions
    - In microservices, they are gone. So we make a new form of transaction
  + How should they look now?
    - Basic Availability
      * We should always have as much data as possible be available to the user, but there’s no guarantee that is all the data
    - Soft State
      * All of the databases combined, might be changing data and that is allowed
      * SQL all data ends in a hard state as soon as a transaction
      * State might always be changing, no guarantees about it being saved to disk
    - Eventual Consistency
      * The database is always moving towards being internally consistent
      * It might never reach there, but if we stopped changing anything. We would get there.
  + How to implement that?
    - Its kind of tricky but we can achieve it using a variety of messaging queues
    - Have a pub sub queue send notifications to a polling queue
    - This way we handle the problem of a service sending more than one request on info change
    - We also handle receiving notifications if a service is crashed
* Messaging Queues
  + Amazon SNS
    - Simple notification service
    - Pub sub model
  + Amazon SQS
    - Simple queue service
    - Polling model
  + How to use these to achieve Eventual Consistency
    - We have a pub sub send notifications to a polling queue and allow a spring to poll messages from that queue
  + DeadLetterQueue
    - If we try to process a message x amount of times and fail, then move that message to a special queue and try the next message.