

Spring Boot Microservices to Cloud Beginner to Guru

including <https://www.coursera.org/learn/spring-cloud-overview/home/week/1>

Traditional Monolith

- A single application (executable)with one code base and build system and single executional program (ie war or ear).
- Disadvantage can become very big can have many many classes and packages
- Traits
 - code is stored together and release are one big version.
 - Typically will use one database
 - Scaling is an all or nothing solution – If One component needs to increase in scalue the whole component needs to scale
 - Usually when scaling create multiple instances of the application clustering and database clustering
- Benefits
 - Development is easy everything is in one project
 - Only one app to deploy and test
- Problems
 - As the business requirements grow, so does the complexity which can lead to spaghetti code – Difficult to modify
 - When it comes to scaling the application must be redeployed (modifications made to code) and must change the internals
 - Technology lock in – Monolith becomes tightly coupled coupled to the technology stack
 - Difficult to introduce new technologies
 - CI/CD difficult
 - Every time a change is made the single application must redeployed

Traditional Monolith

- With a monolith may need to bring up the entire application
- Put all the functionality into one process
- Are Monoliths Bad
 - For small program they are good (not so complex)
 - Maturity of the team

What Microservices Are

- Small targeted service – contains one element of functionality
- Each has its own repository
- Isolated from other services
 - Should not be bundled with other services when deployed
- Loosely Coupled
 - Interacting with other services should be done in a technology agnostic manner
 - Rest Services HTTP/JSON, HTTP/SOAP, XMLRPC
- Microservice Architecture
 - Applications are composed using individual microservices
 - Each service will have its own database
 - Each service is deployable and replaceable and upgradable and scalable
 - Scaling of individual services is now possible
 - CI/CD becomes easier since services are smaller and less complex to deploy

What Microservices Are

- Benefits
 - Easy to understand and develop – Services are smaller and more targeted
 - Software quality – service are more targeted and have a limited scope
 - Scalability – Scale different services at different frequencies
 - Reliability – Software bugs are isolated
 - Technology Flexibility – Each service can have its own language and technology stack
- Disadvantages
 - Integration can be difficult. Have many applications to test
 - Deployments are more complex. Rather than one application there are many
 - Operational cost with each service – Each service is a small application
 - Needs own repo, own deployment process own database, etc
 - Additional hard resources – Additional services need additional hardware to run on.
 - Example multiple JVMs which increases hardware command

What Microservices Are

- How big should microservices be
 - A microservice can be as small as a single API Endpoint
 - Ex Get Orders
 - A microservice can be several or even dozens of API Endpoints
 - Should be able to be supported by 0 – 12 people
 - Scalability
 - The higher the scalability, the more specialized the service should be (more targeted service)

What is the Cloud

- Cloud Based Architectures
 - Microservices are a key aspect to a Cloud Base Architecture
 - Focus on abstraction redundancy and avoidance of single point failure
 - Example file is copied to multiple servers in multiple data centers before save is confirmed.
- Microservices in Cloud Based Architectures
 - Multiple instance of microservices are deployed in a cloud environment
 - If a service is terminated another running instance can assume the workload
- Don't forget Netflix's tool "Chaos Monkey"
 - Terminate process at random to assure there are no single points of failure
 - Achieve Scalability and Reliability
- Deployment Tools
 - AWS ECS/EKS Kubernetes, Docker Swarm Openshift
 - We will use Spring Cloud
- Small Independent services that are designed to allow them to fail without impacting the application.

Adopting Microservices

- Adopting Microservices
 - Often applications start a monoliths
 - Due to culture or cost of building a microservice
 - Monoliths are well established inside corporate
- Microservices grew out of the high demand for scalability
- Decomposing is the process of taking a larger monolithic application and breaking into microservices
 - Strategies
 - By Business Capability – Order Service
 - By Domain Object
 - By action verbs – Payment Service
 - By nouns – Customer service
 - Use the Single Responsibility Principle – A class should have one reason to change (very specific to what they do)
 - Do one thing and do it very well

Adopting Microservices

- Development Teams
 - Small teams should be responsible for specific microservices
 - Lends itself to business functions
 - An account team working on accounting related services
 - Often you will see a lot of overlap of business domain services
 -

Architecture

- Overall View
 - Gateway LoadBalancer
 - Hitting a single point which routes traffic
 - Will have queues to talk to other queues
 - Some will have a database to communicate with
- Gateway
 - Endpoint that is exposed to other services
 - Can be internet for public API
 - More likely to be internal
 - Abstracts implementation of services
 - Client Calls URL is unaware of routing taking place to running instance
 - Individual clients don't know where the service is running
 - Acts as roughly a proxy for network service
 - Can also act a load balancer to distribute the load between the load balancers

Architecture

- Services Instances
 - Expect to be running N Number of services
 - Exact number depends on reliability and load requirements
 - Minimum might be 3 for high availability
 - Some tools allow you to dynamically scale based on load or anticipated load
 - Netflix will scale up and down with load
- Database Tier
 - Typically one database per microservice
 - Highly scalable services will often have on transactional database and one or more read databases
 - Have a mix of NOSQL and SQL and other database
 - Don't have a lot of load my use a shared database
-

Architecture

- Messaging
 - A common pattern is to expose an API Endpoint via a RESTful API
 - Dependent microservices are often message based
 - Messages follow an event or command pattern
 - Messages allow for decoupling and scalability
 - Messaging can be used to define work flow
 - New Order, Validate Order, Charge Credit Card, Allocate Inventory, Ship Order
- Downstream Services
 - Often an action on a microservice will invoke action on multiple down stream services
 - A search on Amazon will invoke over 100 services to return the search results
 - Example – Placing a new order might invoke the following : Validate Order, Pay Credit Card, Allocate Inventory and Ship Order
 -

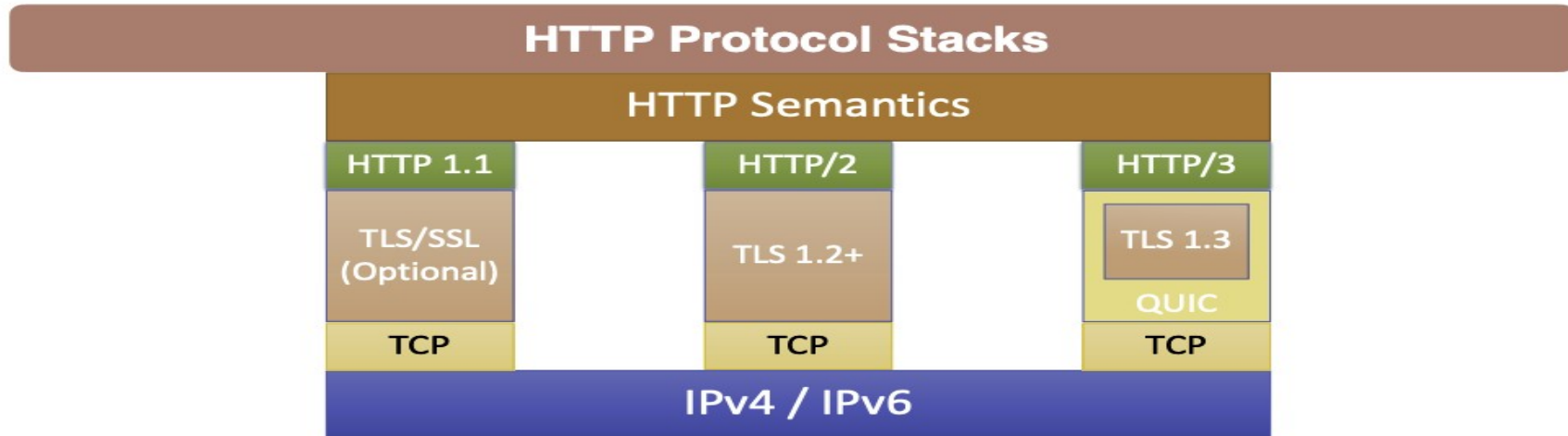
12 Factor Applications

- Best Practices for Development and Deployment of Microservices
 - Codebase – Once Codebase tracked in revision control many deploys
 - Dependencies – Explicitly declare and isolate dependencies in other words Track the versions of the libraries
 - Use Maven
 - Config – Store Config in the environment
 - The build artifact should be considered an immutable environment (don't bake in username/password)
 - Backing Services – Treat backing services as attached resources
 - When deployed the resource are identified by using the configuration and getting information about the environment
 - Build a artifact, release the artifact, run the artifact – Strictly separate build and run stages (
 - Build a docker image which is the artifact and immutable to configure itself for a specific environment
 - Processes – Execute the app as one or more stateless processes
 - Port Binding – Export services via port binding
 - Concurrency – Scale out via the process model (horizontal scalability)
 - Disposability – Maximize robustness with fast startup and graceful shutdown (build artifacts)
 - Can have multiple containers with different versions running
 - Dev/prod parity – Keep development staging and production as similar as possible
 - Logs – Treat logs as event streams (need to create a view across microservices)
 - Admin Processes – Run admin/management tasks as one off process (one off processes)

HTTP Protocol

Http Request Methods

- First part of Spring Course of HTTP Request Methods.
 - HTTP2 and 3 are going to be secure protocols
- Second Part spring Course of HTTP Request Methods
 - Safe Methods are Get, Head, Options, Trace
 - Idempotent (repetitions have no further effect) – Safe Methods _ Put, Delete
 - Post is non idempotent



Http Request Methods

METHOD	Request Body	Response Body	Safe	Idempotent	Cachable
GET	No	Yes	Yes	Yes	Yes
HEAD	No	No	Yes	Yes	Yes
POST	Yes	Yes	No	No	Yes
PUT	Yes	Yes	No	Yes	No
DELETE	No	Yes	No	Yes	No
CONNECT	Yes	Yes	No	No	No
OPTIONS	Optional	Yes	Yes	Yes	No
TRACE	No	Yes	Yes	Yes	No
PATCH	Yes	Yes	No	No	Yes

200 Okay, 201 Created, 204 Accepted
301 Moved Permanently – Telling the client to update its records
400 Bad Request, 401 Not Authorized, 404 not found
500 Internal Server, 503 Service Unavailable

Beginner's Guide to Rest

- REST (Re)presentational (S)tate (T)ransfer
 - Representation State Transfer
 - State Transfer – Typically via HTTP
- Restful Terminology
 - Verbs HTTP Methods (Get, Put, Post, Delete)
 - Message : The payload of the action (JSON/XML)
 - (U)niform (R)esource (I)dentifier – A unique string identifying a resource
 - (U)niform (R)esource (L)ocator – A URI with network information – The url to get the resource
- Idempotent → Exercise the operation multiple times without changing the resource
 - Example refresh web page
- Stateless – Service does not maintain any client State
- HATEOES – Hypermedia as the engine of application state
 - A rest client should be able to use discover all the available actions and resources it needs. As access proceeds the server responds with text that includes hyperlinks to other actions that are currently available

Beginner's Guide to Rest

- HTTP – Get
 - Read only
 - Idempotent and safe operation
- HTTP – Put
 - Use to insert (if not found) or update (if found)
 - Multiple puts will not change the result (ex. Saving a file multiple times)
 - Not a safe operation – does change state of resource, but idempotent
- HTTP – Post
 - Use to create a new object
 - Non-Idempotent – Multiple posts is expected to create multiple object
 - Not Safe Operation – does change state of resource
 - Only Non-Idempotent, Non-Safe HTTP Verb
- HTTP – Delete
 - Use to delete an object (resource)
 - Idempotent – Multiple Delete(s) will have the same effect/behavior and not safe operation does change state of resources

Richardson Maturity Model

- A model used to describe the maturity of restful services since there is no formal specification of soap
- (R)ichardson (M)aturity (M)odel
- Model
 - Level 0 – Swamp of (P)lain (O)ld (X)ML – Very old way of doing things
 - Creating a One Off to your own project
 - Uses implement protocol as a transport protocol
 - Typically uses on URI and one kind of method
 - Examples RCP, SOAP, XML-RPC
 - Using an xml to post a document to a specific URI to do a specific action
 - Level1 – Resource
 - Using multiple URIs to identify multiple resource
 - Examples (<http://www.example.com/product/1234>), (<http://www.example.com/45678>)
 - Still use a single method (ie get)
 - Setting up the concept of resources
 - Breaks large service into distinct URI(s) tied to specific resources

Richardson Maturity Model

- Level 2 – HTTP Verbs
 - Http verbs are used with URIs for desired actions
 - Get, Put, Delete
 - Taking the URI and using the verbs to do different action against a specific resource
 - This level is the most common
- Level 3 – Hypermedia
 - Representation now contains URI which may be useful to consumers
 - Helps client developers explore resources
 - No clear standard as of 2019
 - Spring provides an implementation of HATEOS
 - If you save a resource it will have information with URIs about how interact with it.
 - Provides discoverability making the API more self documenting

Spring Framework and RESTFul Services

- The Spring Framework has very robust support for creating and consuming RESTFul Services
- Spring Framework has 3 distinct libraries for creating RESTFul services
- Spring Framework has 2 distinct libraries for consuming RESTFul services
- Spring MVC
 - The oldest and most commonly used library for creating RESTful web services
 - Based on servlet api, based on blocking which makes it non reactive
- Spring WebFlux
 - Uses project Reactor to provide reactive web services
 - non blocking
 - Follows the configuration model of MVC using annotations to configure the controller
- Functional WebFlux aka WebFlux.fn
 - A functional programming model used to define endpoints
 - Alternative to annotation base configuration
 - Designed to rapidly and simply define microservice endpoint

Spring Framework and RESTFul Services

- Spring RestTemplate
 - RestTemplate is Spring's primary library for consuming RESTFul web services
 - Very mature and has been part of Spring for a long time
 - Highly configurable, but in maintenance mode
 - deprecated since it is a blocking calling
- Spring WebClient
 - The reactive web client
 - Default uses Reactor Netty (a non blocking HTTP Client Library)
- Most common is SpringMVC and SpringTemplate
- Marshaling – Convert Java POJO to JSON or XML / Uninstalling – Convert JSON to XML Uninstalling
- Jackson is the library for Marshaling/Uninstalling
 - Decouples the front end technology
- Single Page Applications – Often combined with Rest APIs for rich user web applications
- design paradigm that relies on asynchronous programming logic to handle real-time updates to otherwise static content

HTTP Get with Spring MVC

- Creating the Spring Boot Project
 - Match artifact ids to the repository
 - Using start.spring.io
 - Using Web , JPA, Actuator, DevTools
- Import it into IntelliJ
 - Don't forget the Annotation Setting and Lombok plugin
- We will be working in packages below guru.springframework.springframework.msscbrewery
- Setup DataModel
 - Web.model
 - Beer.dto // Works with web layer and the entity (database) Beer
-

HTTP Get with Spring MVC

- MVC Controller
 - Example
 - `@RequestMapping("/api/v1/beer")` // Setups a base url for the entire class
 - `@RestController` // A helper annotation extends the controller annotation and adds `@ResponseBody`
 - Public Class {
 - `@GetMapping("/{beerId}")` // Can match up the parameter name beerId
 - `public ResponseEntity<BeerDto> getBeer(@PATHVARIABLE UUID beerId) {` // Can use PATHVariable explicitly
 - `return new ResponseEntity<Beer>(BeerDto.builder().build(), HttpStatus.OK);`
 - }
- Beer Service
 - Logic inside the controller should be minimal. The service should handle the business logic
 - Code to the interface to make dependency injection useful.
 - Inside the controller `private final BeerService beerService;`
- `@GetMapping` – extends `@RequestMapping`, but limits the type of the request mapping to GET
 - `@GetMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.GET)`.
- For constructor Injection no autowiring is needed

HTTP Get with Spring MVC

- In Service Directory
 - @Service
 - In BeerService
 - @Override
 - Public BeerDto getBeerById(uuid beerId) {
 - return BeerDto.builder().id(UUID.randomUUID())
 - .bearName("Galaxy Cat")
 - .beerStyle("Pale Ale");
 - .build()
 - }
- UUID → a 128 bit number
 - Advantage Does not depend on a central server, but probability that the UUID will be duplicated is very close to 0

Introduction to Postman

Axis TCPMon

- In postman use the Get Verb and provide <http://localhost:8080/api/v1/beer/foo>
- www.uuidgenerator.net
- Resend Get Verb and provide <http://localhost:8080/api/v1/beer/<uuid>>
- Additional Thinks heaers, body , different paraemters
- Axis TCPMon
 - A plugin in IntelliJ
 - Can see what is going across the wire for the client/server
 - To the Admin Console add the port number
 - Using port 8081 run the url above and you can see the HTML and you can see the request and response
 - The last 0 is the terminator

HTTP Post with SpringMVC

- In the BearController enter the following:
 - `@PostMapping` – Creates an entity
 - `@ResponseBody`
 - ```
Public ResponseEntity handlePost(@RequestBody BeerDto beerDto) {
 beerDto.builder().id(randomUUID()).build()
 • HttpHeaders headers = new HttpHeaders();
 • headers.add ("Location", "/api/v1/beer/savedDTO.getId().toString()) // Location is mandated by the Rest Standard
 • return new ResponseEntity(headers, HttpStatus.CREATED)
}
```
- Location Http Attribute
  - load a different web page ( URL Redirection ) usually status code 300
  - The UUID fo the newly created object.
- `@RequestBody` → maps the HttpRequest body to a transfer or domain object, enabling automatic deserialization of the inbound HttpRequest body onto a Java object.
- `@ResponseBody` → tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

# Spring Developer Tools

- Added to project via artifact 'spring-boot-devtools'
- Developers tools are automatically disabled when running a packaged application ( ie java -jar )
  - allows it to be disabled when deployed to production
- By default not include in repackage archives
- Developer tools features
  - Automatic Restart
    - Triggers a restart of the Spring Context when class change
    - Use two classloaders. One for the application and one for the project jar dependencies
      - The application contains the stuff that changes and the dependencies usually don't change
      - Since you only reload the application it make reloading very fast
    - Restarts are very fast since only your project classes are being loaded
      - IntelliJ – Build / Make Project will trigger automatic restart
      - There is an advance setting to override it.
  - Template Caching
    - By default templates are cache for performance
    - But caching will require a container restart to refresh the cache
    - Developers tools will disable template caching so the restart is not required to see changes

# Spring Developer Tools

- Live Reload
  - Automatically trigger a browser refresh when resources are changed
  - Spring Boot Developer Tools includes a LiveReload server
  - Browser plugins are available for a free download at [livereload.com](https://livereload.com)
- Sometime you have to do a restart.

# HTTP Put with Spring MVC

- Updating an existing entity
- The id resource should be considered read only by the client
- Example
  - In the Beer Controller
  - `@PutMapping("/{beerId}")`
  - ```
public ResponseEntity handleUpdate(@PathVariable("beerId") UUID beerId, BeerDto beerDto) {  
    • beerService.updateBeer(beerId, beerDto);  
    • return new ResponseEntity(HttpStatus.NO_CONTENT)           // 204 – Request was successful, no content in body by design  
}
```
-

HTTP Delete With Spring MVC

- Example
 - In the BeerController
 - `@DeleteMapping("/{beerId}")`
 - `@ResponseStatus(HttpStatus.NO_CONTENT)` *// Used when no changes to the HTTP HEADER needs to be made and when
// returning no other information*
 - ```
public void deleteBeer(UUID beer) {
 • beerService.deleteById(@PathVariable("beerId") UUID beerId)
}
```
  -
- With lombok you have the `@Slf4j` – Inject a logger and can get logging capabilities

# API Versioning

- Example `/api/v1/beer` where v1 is the version
  - should be towards the beginning of the URL before the operations
- Allow evolution of the API without breaking existing API Consumer
- Lifespan
  - v1 – first Release
  - v2 – second release notify consumers v1 is deprecated
  - v3 – remove v1, notify consumers v2 is deprecated
- Semantic Versioning
  - <http://semver.org>
  - Version Major.minor.patch
    - Major – version for major incompatible API Changes – aka breaking changes
      - ex. Adding a required property
    - Minor – New Functionality – backwards compatible changes
      - ex. Adding a non required property
    - Patch – backwards compatible bug fixes
  - API URLs typically only use major version
    - Can optional use minor and patch `/v1` or `/v1.1`

# API Versioning

- Non-Breaking changes may be performed under Minor or Patch Versions
  - Examples
    - New Optional Parameter
    - New Response Fields
    - New Service
    - Bug Fixes – behavior will not change to API itself
- Breaking Changes should be done under a MAJOR version
  - Examples
    - New Required parameter
    - Removal of existing parameter
    - Removal of response value
    - Parameter name change or type
    - Description of service



# API Versioning Example

- Lets say we had style to make the BeerStyle go from a String to an enum
  - This would be a breaking change
  - When you change a property add a v2 directory and put the new POJO's in that class
  - In the v2 directory is where you would change the beerStyle to beerStyle2 ( adds a version 2 ) to the package
  - In the controller add a v2 and copy the BeerController into that directory as BeerControllerV2 and return the enumeration
  - Also change the BeerServiceV2
  - Use @Deprecated to show that the class is getting for removal for the programming

# API Versioning Example

- `lyndseypadget/semflow`
- Use features branch which correlates to API Version
- Setup the Master Branch that branches Release branch then Feature Branch
-

# LC Beer Service Initial Creation

## LC Beer Service – DataModel Beer Controller

### Junit

- libraries – Actuator, Spring Boot Admin, String Boot client, Actuator Doc, Lombok
- IntelliJ – Enable Auto Import ???
- Jackson API
  - `java.xml.bin:jaxb-api, com.sun.xml.bind:jaxl-core, com.sun.xml.bind:jaxb-impl`
- Don't forget you can add a README.md project that Github will display
- Try IntelliJ Git Commit Directory
- For Data/Time it is best to use UTC. It does not need the timezones which mean you don't need to convert the data for each calculation only printing
- Homework : Find out what `org.springframework.data.domain.Pageable` is
- Junit 5
  - We only want to use Junit
    - Search for <dependency> Tag
    - Search for spring-boot-start-test
    - Add <exclusions><exclusion><groupId>junit</groupId><artifactId>junit</artifactId></groupId></exclusion></exclusions>
    - Need to add `org.junit.jupiter:junit-jupiter-api` / `org.junit.jupiter:junit-jupiter-engine` / `org.mockito-junit-jupiter`
    - `@SpringBootTest` – What does it do.

# LC Beer Service

- @WebMvcTest – Can pass in a list of controller
- Inside the class
  - @Autowired MockMvc mockMvc
  - Inside @Test void getBeerById
    - mockMvc.perform("api/v1/version", UUID.randomUUID().toString()).accept(MediaType.APPLICATION\_JSON).andExpect(status.isOk())
- What is the @Mapper

# HTTP Get with Spring Rest Template

- Maven Config
  - Dependencies
    - spring-boot-start-web, spring-boot-dev-tools, Lombok maven surefire plugin version 3
      - maven enforcer plugin – Enforces particular versions of software
      - Question : Research maven enforcer plugin
      - What does the repository tag do
- Host Name Prop
  - In application sfg.brewery.apihost = <http://localhost:8080>
  - RestTemplate Builder– Want to inject that incase we want to add security, or http client library
  - Example
    - @Component
    - @ConfigurationProperties("sfg.brewery", ignoreUnknownFields = False) // prefix for the apihost
    - public class BreweryClient {
      - public final String BEER\_PATH\_V1 = "/api/v1/beer"
      - private String apiHost // Source in from the application.properties : name in properties file is sfg.brewery.apiHost
      - private final RestTemplate
      - public BreweryClient { this.restTemplate = restTemplateBuilder.build() }
      - public void setApiHost(String apiHost) { this.apiHost = apiHost; }
      - public BreweryClient(RestTemplateBuilder restTemplateBuilder) { this.restTemplate = restTemplateBuilder.build() }
      - public BeerDto getBeerByld(UUID uuid) { return restTemplate.getForObject(apiHost + BEER\_PATH\_V1 + uuid.toString(), BeerDto.class); }
      - }

# HTTP Get with Spring Rest Template

- Get Hold of a Rest Template
  - Inject the Rest Template, but SpringBoot has a preconfigured RestTemplate
  - Best Practice : Get a holder of builder and create a RestTemplate from the builder
    - Use the Standard Environment so you will pick anything up globally
    - RestTemplateBuilder is a Spring API Class

- Testing

- 
- class BreweryClient {
  - @AutoWired
  - BreweryClient client;
  - @Test
  - void getBeerById() {
    - BeerDto dto = client.getBeerById(UUID.randomUUID());
    - assertNotNull(dto);
  - }
  -
- }

# HTTP Post

# HTTP Put

# HTTP Delete

- Example
  - In BreweryClient
  - `public URI saveNewBeer(Beer beerTo) {`
    - `return restTemplate.postForLocation(apihost + BEER_PATH_V1 + , beerDto);`
    - `}`
  - Create an unit test
  - `postForLocation` for be looking at the response header for a location
- HTTP Put
  - `public void updateBeer(UUID uuid, BeerDTO beerDTO) {`
    - `restTemplate.put(api_host + BEER_PATH_V1 + "/" + uuid, beerDTO);`
  - `}`
- HTTP Delete
  - `public void deleteBeer(UUID uuid) {`
    - `restTemplate.delete( apihost + BEER_PATH_V1 + "/" + uuid ); }`
- Errors – Need to handle responses with bad response codes
- For the client make sure you work with interface so you can work with the `RestTemplate` or mock

# HTTP Clients

- Communication Layers → how the client is communicating with the server
  - TCP – Transmission Control Protocol Transport Layer
    - How data is moved in packets between client and server
    - Server listens on port
    - Data is divided up into packets and then reassembled
  - IP – Internet Protocol – Internet Layer
    - Specification of how packets are move between hosts – just one packet
- Network communication in Java is done via java.io
  - These are low java libraries used to communicate with the host operating system
  - TCP/IP connection are made via sockets which are light weight, but there is a cost to establish
    - Thread are more costly than a socket connection
  - Modern OSs can support 100s of thousands of sockets, but only  $\sim 10^4$  threads
- Blocking and I/O
  - Blocking IO – Thread sleep while IO completes ( Pre Java 1.4 ) One thread per connection ( cost to put thread to sleep and wake up)
  - Non Blocking IO which allows for I/O without blocking the reads ( Sets of sockets now can be used by a thread )
    - central to reactive programming
      - non blocking essential to reactive programming – Allowing the processing thread to stay active without getting put to sleep
  - Java 7 added NIO.2 with asynchronous I/O → Networking task completely in background by the OS



# HTTP Clients

- HTTP Client Performance
  - Not uncommon for microservices to have many client connections
  - Non Blocking clients typically benchmark much higher than blocking clients ( could be 4 to 5 time more )
  - Connection pooling used to avoid cost of thread creation and establishment of connections
    - Non Blocking and connection pooling can have a significant difference in the performance of your application
- Blocking Clients
  - JDK
  - Apache HTTP Client
  - Jersey
  - OkHttp – may be changing version 4 ( non blocking ) under development
- Non Blocking
  - Apache Async Client
  - Jersey Async HTTP Client
  - Netty – Used by reactive Spring
  -

# HTTP Clients

- HTTP/2
  - HTTP/2 is more performant than http 1.1
  - HTTP/2 used the TCP Layer much more efficiently
    - Multiplex Streams
    - Binary Protocols / Compression
    - Reduced Latency
    - Faster Encryption
  - To the Rest API Developer functionality the same
    - Both server and client need to support HTTP/2
- HTTP/2 Clients
  - Java 9+      Jetty      Netty
  - OkHttp
  - Vert.x
  - Apache 5.x is currently in Beta of 2019

# Apache HTTP Client Configuration

- Configure Spring Boot
  - Add dependency org.apache.httpcomponents: httpasyncclient
- A restTemplate customizer to customize a restTemplate which contains a customize(RestTemplate restTemplate) function
- The class that customizes the rest template
  - @FunctionalInterface
  - package org.springframework.boot.web.client;
  - import org.springframework.web.client.RestTemplate
  - public interface RestTemplateCustomizer {
    - /\*\*
    - \* Callback to customize a Rest Template instance
    - \* @param restTemplate the template to customize
    - \*/
    - void customize(RestTemplate restTemplate) {
      - restTemplate.setRequestFactory( clientHttpRequestFactory())\_
    - }
- To the POM add the dependency : org.apache.httpcomponents:httpasyncclient.4.14.
- The customize

# Apache HTTP Configuration

- Should have one only one, but showing ClientHttpRequestFactory, but show two different factories
- Generally Setup a ConnectionManager, RequestConfiguration then setup the client which gets injected into the RequestFactory
  - The factory will be set on the RestTemplate
- ```
public class BlockingTemplateCustomizer implements RestTemplateCustomizer {  
  
    // Sets up all the apache configuration  
  
    public ClientHttpRequestFactory clientHttpRequestFactory() {  
        • // Creation and configuration for the pooling manager  
        • PoolingHttpClientConnectionManager connectManager = new PoolingHttpClientConnectionManager();  
        • connectionManager.setMaxTotal(100);  
        • connectionManager.setDefaultMaxPerRoute(26)  
        • // Configure the client  
        • RequestConfig requestConfig = RequestConfig.custom().setConnectTimeout(3000).setSocketTimeout(3000).build();  
        • ClosableHttpClient httpClients.custom().setConnectionManager(connectionManager).setKeepAliveStrategy(new DefaultKeepAliveStrategy()).setDefaultRequestConfig(requestConfig).build()  
        • // Inject the client into the Request Factory  
        • return new HttpComponentsClientHttpRequestFactory(httpClient) }  
  
    private ClientHttpRequestFactory clientHttpRequestFactory() {  
        • restTemplate.setRequestFactory( this.clientHttpRequestFactory()); // Sets up the Apache  
    }  
}
```
- Drill to and look HttpRequestFactory – Many different clients → BufferingClientHttpRequestFactory, ClientHttpRequestFactory
 - We are implementing ClientHttpRequestFactory

Apache Client Request Logging

- Need the previous section on Apache Client Configuration
- Can setup a Request Interceptor and actually inspect that request and implement your own logic.
 - Both Apache and Nutty have support
- Show what is going across the wire
 - In application.properties set the logging.level.org.apache.http = debug
 - look for something like org.apache.http.headers or org.apache.wire
 - Be careful about Personal Information

LC JPA Entries

SpringDataRepositories

- The primary key is a UUID in the class Beer with annotation @Entity
 - @Id
 - @GeneratedValue(generator="UUID")
 - @GenericGenerator(name="UUID" strategy="org.hibernate.id.UUIDGenerator"_)
 - @Column(length = 36, columnDefinition="varchar(36) not nullable = true, unique = true")
- @Column – tells the database how to create the column
- @Version private Long version // optimistic logging property
- @CreateTimeStamp , @UpdateTimeStamp // Hibernate specific from JPA
- Spring Data Repositories
 - PageAndSortingRepository
 - NoRepositoryBean – ???
- BootStrap Class to initialize Beer Objects
 - Create a directory call BootStrap
 -

LC – Bootstrap Data

```
- public class BeerLoader implement CommandLineRunner {  
    • private final BeerRepository beerRepository  
    • public BeerLoader(BeerRepository beerRepository) {  
        - this.beerRepository = beerRepository; }  
    • @Override  
    • public void run( String.. args) throws Exception {  
        - loadBeerObjects()  
    • }  
    • private void loadBeerObjects () {  
        - // if ( nothing is initialized ) then create a set of beer object which are saved to the beerRepository  
- }
```

Spring MVC Validation

- History
 - JSR 303 – Introduced Java Bean Validation
 - Set of annotations used to validate Java Bean Properties
 - Before that it was rather painful
 - JSR 349 – Java Bean Validation 1.1 provided method level validation to validate input parameters and dependency injection for bean validation components
 - JSR 380 – Bean Validation 2
 - Used in Spring Boot 2++
 - Add Java 8 languages features
 - Add 11 new build in validation features (Hibernate 6.0 (Implementation of Bean 2.0))
 - See Spring 5 – Data Validation JSR 303
 - Hibernate Validator is a popular implementation of the API
- Build in Constraint Definitions of Spring Bean Validation
 - @Null, @NotNull, @Asserttrue, @AssertFalse, @Min, @Max. @DecimalMin, @DecimalMax, @Negativie, @NetativeOrZero, @Positive, @PositiveOrZero
 - @size – Check if between a min or max – can be a collection or size
 - @Digit – Check howmany digits there are and fraction – How many after the decimal point

Spring MVC Validation

- @Past, @PastOrPresent, @Future, @FutureOrPresent
- @Pattern
- @NotEmpty
- @NotBlank
- @Email
- Hibernate Validator
 - @ScriptAssert Class level annotation, checks class against script – When do you use it (google it)
 - @CreditCardNumber
 - @Currency
 - @DurationMax, @DurationMin –
 - @EAN
 - @ISBN
 - @Length
 - @CodePointLength
 - @Range // Acts a combination of Min and Max
 - @SafeHTML // Verify the string has no malicious characters
 - @UniqueElements
 - @URL

Bean Validation :

At 2:23 show how to drill into source

- spring-boot-starter-validation

- Example

- @Data
- @NoArgConstructor
- @AllArgConstructor
- @Builder
- public class BeerDto {
 - @Null // Client can not set it to null
 - private UUID id
 - @NotBlank
 - private String beerName
 - @NotBlank
 - private String beerStyle
 - @Positive
 - private Long upc
 - }

Bean Validation

- Inside the controller
 - `public ResponseEntity handlePost(@Valid @RequestBody BeerDto)`
 - `public ResponseEntity handlePost(@Valid @RequestBody BeerDto)`
 - Get an Http Error Code for 400

Bean Error Validation

Spring Boot Method Validation

- Example in the controller
 - // When ConstraintViolationError was thrown.
 - @ExceptionHandler({ConstraintViolationException.class}) // How Bean Validation throws exceptions
 - public ResponseEntity<List> validationErrorHandler({ConstraintViolationException cl}) {
 - List<String> errors = new ArrayList<> (cl.getConstraintViolations().size());
 - cl.getConstraintViolations().forEach(constraintViolation -> (
 - errors.add(constraintViolation.getPropertyPath() + " : " + constraintViolation.getMessage());
 - });
 - return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST)
 - }
- Spring Boot Method Validation
 - @Valid - Spring Boot attempts to validate @ConfigurationProperties

Spring MVC Controller Advice

LC – Bean Validation Validation and Error Handling

- Create a java class MVCEExceptionHandler
- add @ControllerAdvice to the top of class – Allows you to handle exceptions across all applications
 - Allow all classes that throw this exception to use the method mention below.
 - good for removing duplicate code
- add the ValidationExceptionHandler from a previous
- How to handle a bind exception – Bind a socket to a local address and port
 - @ExceptionHandler(BindException.class)
 - public ResponseEntity<>(BindException be) {
 - return new ResponseEntity(be.getAllErrors(), HttpStatus.BAD_REQUEST);
 - }
- Spring LC - Bean Validation and Validation and ErrorHandling
 - For properties that the client should not have to set (primaryKey, version, createDate, LastModifi9ed give them the annotation null so they must null
 -

Overview of Project Lombok

- How lombok works
 - Hooks in via Annotation process API
 - AST (raw source code) is passed to Lombok for code generation before the Java Compiler continues
 - target/classes can view compile class files
 - IntelliJ will decompile to show source code
- Project Lombok Features
 - `val` – local variables declared final
 - `var` mutable local variable
 - `@NonNull` Null Check – throw NPE if parameter is null
 - `@Cleanup` Will call close on resource in finally block
 - `@NoArgs Constructor` – Will cause compiler error if there are final fields
 - `@RequiredArgsConstructor` – throw a Null Pointer Exception if any `@NonNull` fields are null
 - `@Value` – immutable variant of `@Data`
 - `@SneakyThrows` – Throws checked exceptions without declaring method's throw clause
 - `@Synchronized` – A safer implementation of java's synchronized

Overview of Project Lombok

Lombok Examples

- `@Getter(lazy = true)` will calculate the value the first time and then cache
- `@Slf4j` will allow you to swap out different logging frameworks – change source code
- Lombok Examples
 - Can use `val/var` ex. `val httpHeaders = new HttpHeaders();`
 -

Overview of Mapstruct

- Map Struct is a code generator for mapping between Java Bean Types (DTO to Entities)
 - There may be some data that you don't want to show the user
 - The consumers of the data are going to have different needs.
- Follows convention over configuration
- The developer declares an interface, MapStruct generates the implementation
 - Properties of the same name/type will get mapped automatically
 - Different property names are configured via annotation properties
 - Different types can be referenced with additional mapper implementations
- If present, MapStruct will use builders, will use lombok builders
- Mapper can reference other mappers – Order Mapper can use order line mapper
- Can be configured to annotate generated mappers as Spring Components
 - Useful for Dependency injection with Spring
- Can use default methods on interfaces
- Complex mapping can be done via abstract class – ex. a web service call to get information for a DTO

Map Struct Configuration

- Add maven dependency org.mapstruct:mapstruct > 1.3 versio
- can inherit springboot version of libraries from parent example lombok use \${lombok.version}
- Since we are using MapStruct and Lombok together we need to maven compiler processor about the annotation path
- To work with both Lombok and Map Struct
 - Need a build plugins plugin – Telling the maven build system about the two annotation process so the Java Compile can use them
 - org.apache.maven.plugins.maven-compiler-plugin > 3.8 version
 - configuration
 - Add An Annotation path
 - <plugin><groupId>org.apache.maven.plugins</groupId><artifactId>maven-compiler-plugin</artifactId><version>3.80</version><Configuration>
 - <annotationProcessorPaths>
 - <path>
 - <groupId>org.mapstruct</groupId>
 - <artifactId>mapstruct-processor</artifactId>
 - <versionId>1.3.0.Final</version>
 - </path>
 - <path>
 - <groupId>org.project.lombok</groupId>
 - <artifactId>lombok</artifactId>
 - <version>\${lombok.version}</version>
 - </path>
 - </annotationProcessingPaths>
 - <compilerArgs>
 - <compilerArg>-Amapstruct.defaultComponentModel=spring</compilerArg> // MapStruct any generated class annotate with @Component
 - </compilerArgs>

Example of Using Project Lombok and MapStrut

- Example – Usually done int the Persistence Layer
 - @Data
 - @NoArgsConstructor
 - @AllArgsConstructor
 - @Builder
 - public class BeerDtoV2 {
 - private UUID id;
 - private String beerName;
 - private BeerStyleEnum beerStyle;
 - private Long upc
 - @Null
 - private offsetDateTime // Need a different mapper for these to – Will be called Date Time Mapper
 - @Null
 - private offsetDateTime
 - }
 - We have the same class in 2 different packages guru.springframework.mscbrewery.domain and guru.springframework.mscbrewery.web.model.v2

Example of Using Project Lombok and MapStrut

- Add directory mappers
 - @Mapper
 - public interface BeerMapper {
 - BeerDto beerToBeerDto(Beer beer);
 - Beer beerDtoToBeer(BeerDto dto);
 - }
 - mvn compile and in the target directory we will get a BeerMapperImpl
 - BeerMappaerImpl contains methods such as beerToBeerDTO, beerDtoToBeer
 - There is an @Generated Annotation with value, date and comments – good if you generate code
 - MapStrut will convert the String in the database to a enum in the java code.

Example of Using MapStruct

- Example - Used by map struct to convert a Timestamp to an OffsetDateTime both ways (custom code)
- @Component
- public class DateMapper {
- public OffsetDateTime asOffsetDateTime(Timestamp ts){
- if (ts != null){
- return OffsetDateTime.of(ts.toLocalDateTime().getYear(), ts.toLocalDateTime().getMonthValue(),
- ts.toLocalDateTime().getDayOfMonth(), ts.toLocalDateTime().getHour(), ts.toLocalDateTime().getMinute(),
- ts.toLocalDateTime().getSecond(), ts.toLocalDateTime().getNano(), ZoneOffset.UTC);
- } else {
- return null; } }
- }
- public Timestamp asTimestamp(OffsetDateTime offsetDateTime){
- if(offsetDateTime != null) {
- return Timestamp.valueOf(offsetDateTime.atZoneSameInstant(ZoneOffset.UTC).toLocalDateTime());
- } else {
- return null; } }}

Data Conversion using MapStruct

- Still using the same DTO from before Beer

- To Beer we add

```
- private Timestamp createdAt;  
  
- private Timestamp lastUpdateDate
```

- To BeerDTO

```
- private OffsetDateTime createdAt  
  
- private OffsetDateTime lastUpdateDate
```

- Problem we need to do a Type conversion

```
- Add a new Java class to the mappers called DateMapper
```

```
- public class DateMapper {  
    • public class OffsetDateTime  
        - if ( ts != null ) {  
            • return OffsetDateTime.of(ts.toLocalDateTime().getYear(), ts.toLocalDateTime(), ts.toLocalDateTime().getMonthValue(),  
            • ts.toLocalDateTime().getDayOfMonth(), ts.toLocalDateTime().getHour(), ts.toLocalDateTime().getMinute(),  
            • ts.toLocalDateTime().getSecond(), ts.toLocalDateTime().getName(), ZoneOffset.UTC);  
        - } else {  
            • return null;  
        - }  
    }  
}
```

```
- public Timestamp asTimestamp(OffsetDateTime offsetDateTime) {  
    • if ( offsetDateTime != null ) {  
        - return Timestamp.valueOf(offsetDateTime.atZoneSameInstant(ZoneOffset.UTC).toLocalDateTime());  
    • } else {  
        - return null;  
    • }  
}
```

```
- Above BeerMapper add @Mapper(uses = {DateMapper.class})
```

Adding CI Builds with Circle CI

- Google Circle Ci
- Need to add project and setup project
 - On your machine need to create .circleci into the project and add an yml
 - commit to github
 - Ci build badge - tell you when it will fail
- Circle CI cache dependencies
- run maven integration test : compile, test , integration
- using webhooks once you comit it does it right away

Spring Rest Docs

- What it is
 - A tool for generating API Documentation
 - Spring REST Docs hooks into controller test to separate documentation snippets
 - The test will generate small documents of information The snippets are then assembled into final documentation via Asciidoctor
 - Test Clients supported
 - SprintMVC Test
 - WebTestClient (WebFlux)
 - REST Assured
 - Test Frameworks supports
 - Junit 4 and 5
 - Spock
 - Test NG with additional configuration required
 -

Spring Rest Docs

- Default Generated Snippets
 - curl-request
 - http-request
 - http-response
 - httpie-request
 - request-body
 - response-body
- What will happen
 - Test(s) will generate a collection of snippets that will be passed through AsciiDoctor to create the final documentation.
 - A few options
 - Can optionally use Markdown rather than AsciiDoctor
 - Maven and Gradle maybe used for the build process
 - You can package the documentation to be service as static content via Spring Boot
 - Extensive Options for customizing Spring Boot

Spring Rest Docs

- Third Party Extensions
 - restdocs-wiremock – Auto generate WireMock Stubs
 - restdocsext-jersey – Enable use of Rest Docs with Jersey's Test Framework
 - spring-auto-restdocs – Use reflection to automatically document request and responses parameters
 - restdocs-api-spec – Generate OpenAPI 2 and OpenAPI 3 specifications
-

Spring Docs Maven Configuration

- Dependency
 - Maven Coordinates – Give use the testing capabilities (inherit the version from the
 - group id : org.springframework.restdocs
 - artifact spring-restdocs-mockmv
 - test
 - Curated dependency (version managed by Spring Boot)
- In build, plugins, plugin
 - group id : org.asciidoctor
 - artifact: asciidoctor-maven-plugins
 - version 1.5.3
 - execution
 - phase: prepare package (just before maven packages the project)
 - goals: process-asciidoc
 - configuration
 - backend is html and doctype is book
 - dependency: (group) org.springframework.restdocs, (artifact) spring-restdocs-asciidoctor, (version) \${spring.restdocs.version}

Spring Docs Maven Configuration

- Ascii Doctor
 - Create a new directory called asciidoc which will be looked for in the ascii doctor plugin – Past in a template which will be converted into an HTML Document
 - In the directory create a file index.adoc
 - index.adoc is the template
 - Example index::(snippets)/orders url-request.adoc
 - This snippets will get replaced by the code
- Sprng Rest Documentation is documented

Spring Mock MVC Configuration

- Need a Controller test
- Example
 - `@RestDocumentationExtension.class(@RestDocumentationExtension.class)` // Need for Spring MVC Rest Doc
 - `@AutoConfigureRestDocs` // Auto configure documentation – Need for Spring MVC Rest Docs
 - `@MockMvcTest(BeerController.class)`
 - `@ComponentScan(basePackages = "guru.springframeowkr.sfgrestdocsexample.web.mappers")`
 - class BeenControllerTest {
 - `@Autowired`
 - `MockMvc mockMvc;` // Adding an `@MockMvc` from Spring Boot for MVC Rest Docs
 -
 - `@Autowired`
 - `ObjectMapper objectMapper;`
 -
 - `@MockBean`
 - `BeerRepository beerRepository;`
 - `@Test...`
 - }

Documenting Path Parameters

- Use Case – ex get require an input parameter
- First Part contains an issue – make sure to don't use the webServlet perform()
- Spring Rest Doc are hooked in the mockMvc
- for the mockMvc.perform() at the end add a andDo(document("v1/beer", pathParameters(where v1/beer is the endpoint
 - parameterWithName("beerId").description("UUID of desired beer to get"))));
 - the description is documentation for the endpoint
 - the mockMvc.get(get("/api/v1/beer/{beerId}", UUID.randomUUID().toString()).accept(MediaType.APPLICATION_JSON) // instead of having /api/v1/beer/3

Documentation Query Parameters

Documenting Response

Documenting Requests

- `mockmvc.perform(...).param("iscold", "yes").andDo(document(requestParameters(parameterWithName("iscold").description("Is Beer Query Param"))))`
- Note description will contain all document from mockMvc such as pathParameters, requestParameters
- Documenting Response
 - In side the document add
 - `responseFields(
 - fieldWithPath("id").description("Id of Beer") // Has to document all the properties`
 - Getting a lot of documentation mixed in with test and get code duplication
- Documenting Request
 - Now lets look at saveBeer which takes in a BeerDTO
 - `.andDo("v1/beer",
 - requestFields(
 - fieldWithPath("beerName"), description("Name of Beer:)) // Need to document all the properties
 - fieldWithPath("id").ignored() // This is uued server by persisted layer user does not need to know about`

Documentation Constraints

- Add @Size(min=3, max=100)
- Need to setup a custom template
 - In resources org.springframework.restdocs.templates
 - create a new file called request-fields snippet --request-fiels.snippet – don't indent just like github it messes up the code.
 - |===
 - |Path|Type|Description|Constraints
 - •
 - |{{ #fields}}
 - |{{path}}
 - |{{type}}
 - |{{description}}
 - |{{constraints}}
 - |{{/fields}}
 - • |===
 - use this inner class to get validation values
 - private static class ConstrainedFields // Will replace the fieldWithPath with fields.withPath – the one found here
 - private final ConstraintDescriptions constraintDescriptions;
 - ConstrainedFields(Class<?> input) { this.constraintDescriptions = new ConstraintDescriptions(input); }
 - private FieldDescriptor withPath(String path) { return fieldWithPath(path).attributes("constraints").values(StringUtils.collectionToDelimitedString(this.constraintDescriptions.descriptionsForProperty(path), ".")); } }

Serving Docs with Spring Boot

- maven configure
 - <plugin>
 - <artifactId>maven-resources-plugin</artifactId>
 - <version>2.7</version>
 - <execute> id copy resources and configuration for the output directory

Documentation Constraints

- In the test call – `ConstraintDescriptions fields = new ConstaintDescriptions(BeerDto.class)`
- `adoc` is field generated from the template

URI Customization

Documentation Generation

- `curl 'http://localhost:8080/api/v1/beer/<uuid>' -H 'Accept: application/json'` – This is actually the documentation
- `@AutoConfigureRestDocs` can have variables set `uriScheme="https"` `uriHost="dev.springframework.guru"` `uriPort = 90`
 - `mvn` package will regenerate the docs
- Documentation Generation
 - For the document function the identifier must `/v1/beer` should be replaced with `/v1/beer-get`
- All the generated code is under `target/generated-snippets`
- `ascii doc` assembles all the files with `*.doc`

Maven BOM Setting Common Properties

- In the pom file created from the previous slide
 - Can add the tag licenses, license
 - Can add the organization
 - Can add the developers
 - Have a properties
 - `project.build.sourceEncodings` - UTF-88
 - `project.reporting.outputEncoding` - UTF-8
 - `java.version` – 11
 - `maven-compiler.source` -- `${java.version}`
 - `maven-compiler.target` – `${java.version}`
 -

Maven BOM Dependency Management

- dependencyManagement Tag
 - set up dependencies that can be inherited from this pom downstream
 - They are not part of the project, not a transitive dependency
 - If you use this dependency this version is used.
 - Example
 - `<dependencyManagement>`
 - `<dependencies>`
 - `</dependencies>`
 - `</dependencyManagement>`
 - Allow us to centralize the specific dependencies. Each project can or cannot include them
 - Can handle `<scope>`

Enterprise Dependency Management

- Spring Parent Bom (bill of materials)
 - Spring is already providing us a Parent POM which provides us a set of curated proeprties
 - Also common set of properties is deleted
 - Differences from BOM
 - Does not set common dependencies
 - Does not set common plugins
 - Does not set common plugin configure
- Spring Parent POM inherits from the BOM inherits from Service POM
- BOM Configuration
 - Set common maven properties
 - set common maven plugins and configuration
 - Set dependency versions
 - Set common dependencies
 - set common build profiles
 - set just about any inheritable property which is common

Maven BOM Creation

- repository mssc-brewery-bom – Contains a pom file to setup our own specific properties
- Create a new project : maven with no archetype
- group gruug/springframework
 - make the github name and artifact id the same :mssc-brewery-com
- From another project being used
 - copy .mvn, mvnw, mvnw.cmd, README.md
- In the pom.xml
 - Change the packaging to pom

Maven BOM Common Plugins

Maven Enforcer Build Plugin

- spring boot maven plugin
- maven clean plugin – can do an automatic clean
- maven compiler plugin
- Maven Enforcer Plugin – rules ot fail the build on
 - maven version `<version>[3.6.0)</version>` `> 3.6.0`
 - java version
 - `requireReleaseDependencies` – Request no Snapshots when release --- Child Tags onlyWhenRelease and Message

IntelliJ Workspace Tips and Tricks

- Setup a workspace for all the projects in an application.
 - Create a directory
 - From IntelliJ create an empty project : end it with -ws
 - clone all you projects into the directory
 - IntelliJ : File → New → Module from Existing Sources and import using maven
 -

Overview of Datasource Configuration with Spring Boot

- Data source Configuration tells Java how to connect to a JDBC compliant Data Source
- JDBC is the standard for connecting to Relational Databases
- Spring Boot Embedded Databases
 - If embedded datais on classpath Spring will auto-configure embedded data source if not provided in config
 - H2, HSQL, Derby are supported
 - H2
 - generally preferred for local development
 - support comparability modes for most major databases
 - Has a GUI to browse the database
 - Configured automatically if DevTools is on the Classpath
- SpringBoot Datasources
 - For permanent datasource you are responsible for providing data source configuration details
 - When you provide data source connection details, an embedded database might not be started
 - Type managed via Sprint Boot Configuration Options
 - application.properties , application-<profile-name>.properties
 - environmental parameters
 - Spring Cloud Config

Configure H2 MySQL Compatibility Mode

MySQL Requirements and Setup

- properties
 - `spring.datasource.url=jdbc:h2:mem:testdb; MODE=MYSQL` // Sets it MySQL Compatibility Mode
 - url must be the same as the url found in the H2 Console
 - `spring.h2.console.enabled=true`
- MySQL Requirements and Setup
 - When asked about authentication make sure version five is installed
 - Connect on localhost:3306
 - GUI –
 - MySQL Workdbench
 - Dbeaver – supports mongoDB too
 - Sequel Pro
 - Squirrel SQL
- MySQL Setup
 - Very poor practice – root is the administrator account for mysql – Best practice is to use an account with minimal authority in the database
 - Ideal is a service account with CRUD authorities only
 - In this example we will setup schema owner account – CRUD + DDL operations due to Hibernate is managing schema
 - Each microservice will have own user and database schema – simulating unique DB per service – Each account will only see its schema

MySQL Beer Service Configuration

- From the mssc-beer-service in resources there will a script to create the schema and add the data to the database
- SQL Statements – MySQL
 - drop user if exists 'beer_service'@`%` – The user would `beer_serice@Ip` where IP is a wildcard
 - create user if not existis `beer_servcie`@`%` identified with mysql_native_password by 'password' // identified with import > Mysql 8
 - create database if not exists beerservice CHARACTERSET utf8mb4 COLLATE uft8mb4_unicode_ci
 - create user if not exists `beer_service`@`%`
 - grant select, insert, update, delete, create, drop, references, index, alter, execute, create view, show view, create routinee, alter routine, event, trigger on `beerservcie`. * to `beer_service`@`%`
 - flush privileges;

Configure Beer Service for Local Mysql

- application-localmysql.properties
 - spring.datasource.username=beer_service
 - spring.datasource.password=password
 - spring.datasource.url=jdbc:mysql://127.0.0.1:3306/beerservice?useUnicode=true&characterEncoding=UTF 8 & serverTimezone=UTC
 - spring.jps.database=mysql
 - spring.jpa.hibernate.ddl-auto=create
 - Several different modes for hibernate managing the schema
 - create Create the schema and destroy previous model
 - create-drop Create and then destroy the schema at on shutdown destroy it
 - none Disable DDL Handling
 - update Update the schema if necessary
 - validate Validate the schema, make no changes to the database // Important for production
 - spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect // Just the name of the database
 - spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

Correcting Hibernate Errors With Mysql

- Getting the error Username: null SSL certificate subject DN: unavailable
 - Fix @SpringBootApplication(exclude = ArtemisAutoConfiguration.class)
 - problem is JMS is not running
- You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use
 - Was working with H2
 - some accept an empty varchar and other database need a size.
- Assignment – Database Initialization

Data Source Connection Pooling

- Establish a Database Connection is an expensive operation
 - Call out to Database Server to get authenticated
 - Database server needs to authenticate
 - Establish a connection
 - Establish a session ie allocate memory and resources
- Database Optimizations
 - Prepared Statements : SQL with placeholders for variables
 - Saves server from having to parse and optimize execution plan
 - avoid SQL Injection attacks
 - Optimizations within a single datasource connection
 - cache prepared statements – if reused then it will return the cache values
 - User server side prepared statements – Look globally and cache statements
 - statement batching – A series of SQL Statements at once
 - Using JPAA you get a lot of this for free

Data Source Connection Pooling

- Spring Boot 1.x used Tomcat
- Spring Boot 2.x used HikariCP
 - light weight
 - very high performance
- hackers guide to connection pooling
 - Every RDMS will accept a max number of connections – Each connection has a cost
 - If running multiple instances of your microservice keep number of pool connections lower
 - If fewer number of microservice, can go to higher of connections per instance
 - MySQL default to a limit of 151 connection – Can be adjusted to much higher depending on the hardware running MySQL
 - statement caching is good, but does consume memory on the server
 - disabling autocommit can help improve performance
 - If doing heavy inserts/updates – can improve performance
-

HikariCP Configuration for MySQL

- properties
 - `spring.datasource.hikari.maximum-pool-size = 5`
 - `logging.level.org.hibernate.SQL=DEBUG`
 - `logging.level.com.zaxxer.hikari.HikarConfig=DEBUG`
 - `logging.level.org.hibernate.type.descriptor.sql.BasicBinder = true`
- The documentation has a set of recommended values

Introduction to JMS

- java messaging service
- A Java API to send a message to another application
- Requires an underlying implementation to be provided
- highly scalable and allows loosely couple applications using asynchronous messaging
- Examples of JMS Implementations
 - Apache Active MQ
 - Rabbit MQ
 - Jboss Messaging
- Why Use JMS over rest
 - JMS is a true messaging service
 - Asynchronous – send and forget
 - Greater throughput – The HTTP protocol is slow and JMS protocols are very performant
 - Flexibility - Deliver to one or many consumers
 - Security
 - Reliability – Can guarantee message delivery

Introduction to JMS

- Type of Messages
 - Point to Point
 - Message is queued and delivered to one consumer
 - Can have multiple consumers, but the message is delivered only once
 - Consumers connect to a queue
- Publish / Subscribe
 - Message is delivered to one or more subscribers
 - Subscribers will subscribe to a topic then receive a copy of all the messages sent to the topic
- Key Terms
 - JMS Provider JMS Implementation
 - JMS Client Application which send or receives messages from the JMS provider
 - JMS Producer or Publisher JMS Client which sends messages
 - JMS Consumer or Subscriber JMS Client which receives messages
 - JMS Message
 - JMS Queue Queue for point to point messages, often FIFO
 - JMS Topic Similar to a queue, but for publish and Subscribe

Introduction to JMS

- JMS Message
 - Header contains metadata about the message
 - Properties
 - Application From Java Application send message
 - Provider Used by the JMS provider and are implementation specific
 - Standard Properties Defined by the JMS API and must be supported by the provider
 - Payload – The message itself
- JMS Header Properties
 - JMS CorrelationID String value typically a UUID. Set by application often used to trace a message through multiple consumers
 - JMSExpires Long – zero does not expire, else time when message will be and be removed
 - JMSMessageID A string typically set the JMS Provider
 - JMSPriority Priority to the message
 - JMSTimestamp time when message was sent

Introduction to JMS

- JMS Header Properties
 - JMSType String – The type of message
 - JMSReplyToQueue or topic which sender is expecting replies
 - JMSRedelivery Boolean – Has message been redelivered
 - JMS DeliveryMode Integer, set by JMS Provider for delivery mode
 - Persistent (Default) JMS Provider should make best effort to deliver message
 - Non Persistent Occasional lost message is acceptable
- JMS Message Properties
 - JSMXuserId User Id sending message. Set by JMS Provider
 - JSMXAppId Id of the application sending the message. Set by JMS Provider
 - JMSXDeliveryCount Number of delivery attempts. Set by JMS Provider
 - JMSXGroupId The message group which the is part of/ Set by JMS Client
 - JMSXGroupSeq Sequence Number of message in group Set by JMS Client
 - JMSXProducerTXID Transaction ID when message was produced Set by JMS Producer
 - JMSXConsumerTXID Transaction ID when message was consumed Set by JMS Provider
 - JMSXRcvTimestamp Timestamp when message was delivered by consumer Set by JMS Provider
 - JMSXSate State of the JMS Message Set by JMS Provider

Introduction to JMS

- JMS Custom Properties
 - Properties are set as key /value pairs
 - Values must be : String, boolean, byte , double, float, int, short, long, Object
- JMS Provider Properties
 - JMS Client can also set JMS Provider specific properties
 - name is JMS+<provider name>
 - JMS provider specific properties to utilize features specific to the JMS Provider
- JMS Message Types
 - Message Just a message with no payload used to notify about events
 - Bytes Message payload is array of bytes
 - Text Message Message is store as String (Often JSON or XML)
 - Stream Message sequence of Java Primitives
 - Map Message Name / Value Pairs
 - Object Message Serialized Object
- Message can be consumed by any technology
- Best Practice – Use Text Message with JSON or XML

Message Converter Configuration

- Setting up at test message with a JSON Payload
- The message Convert make it easy to convert between the JSON and the JMS Message
- Create a new class JMS Config with annotations @Configuration
 - @Bean
 - public MessageConverter messageConverter() {
 - MappingJackson2MessageConverter converter = new MappingJackson2MessageConvert();
 - converter.setTargetType(MessageType.TEXT) // Should the message be bytes or text.
 - converter.setTypeIdPropertyName("_type"); // A property name such as the classname
 - return converter;
-

The Message Object

Embedded Server Configuration

Task Configuration

- Create the POJO that will be the payload of the message
 - should implement Serializable if sent as a JAVA_OBJECT
- Embedded Server Configuration
 - In the main method
 - `// Minimum Configuration`
 - `ActiveMQServer server = ActiveMqServers.newActiveMQServer(new ConfigurationImpl()`
 - `.setPersistenceEnabled(false)`
 - `setJournalDirectory("target/data/journal")`
 - `.setSecurityEnabled(false)`
 - `.addAcceptorConfiguration(name="invm", "vm://0");`
 - `server.start()`
- Task Configuration
 - Enable Configuration so we can send out a message at a fixed rate
 - Create a java Class called TaskConfig with annotations `@EnableScheduling`, `@EnableAsync`, `@Configuration`
 - In the Class have a constructor with `@Bean` return new `SimpleAsyncTaskExecutor()` `// TaskExecutor – part of the Spring Framework`

Create Beer Event Object

- Tie an event to Java Object
- Create a Beer Event
 - @Data
 - @RequiredArgsConstructor
 - @Builder
 - public class BeerEvent implements Serializable
 - private final BeerDto beerDto
 - }
 - public BrewBeerEvent extends BeerEvent {
 - public BrewBeerEvent(BeerDto beerDto) {
 - super(beerDto)
 - }
 - }
 - public NewInventoryEvent extends InventoryEvent {
 - public NewInventoryEvent (BeerDto beerDto) { super(beerDto) } }

Create Brewing Service

- @Slf4j
- @Service
- Need a taskConfig which was written be see @EnableSync, @EnableScheduling
- ```
public class BrewingService {
 - private final BeerRepository beerRepository;
 - private final BeerInventoryService beerInventoryService;
 - private JMSTemplate jmsTemplate;
 - private final beerMapper beerMapper
 -
 - public void checkForLowInventory() {
 • List<Beer> beers = beerRepository.findAll();
 • beers.forEach(beer. → {
 - Integer invQOH = beerInventoryService.getOnhandInventory(beer.getId());
 - if (beer.getMinOnHand() >= invQOH)
 • jmsTemplate.convertAndSend("brewing-request", new BrewBeerEvent(beerToBeerDto(beer)));
 }
 }
}
```

# Initial Project and Maven Dependencies

- Initial Project and Maven Dependencies
  - Create a new project
    - Spring Web Starter
    - Spring ActiveMQ Artemis
  - The spring start for Artemis will only give you the ability to talk to the sever
    - We will be talking to an embed server need org.apache.activemq:artemis-server, or.apahce.activemq-artemis-jms-server
    -

# Sending JMS Message

- In the Jms String add the queue name
  - `public String MY-QUEUE = "my-hello-world"`
- `@RequiredArgsConstructor @Component public class HelloSender {`
  - `private final JmsTemplate jmsTemplate;`
  - 
  - `@Scheduled(fixedRate = 2000 )`
  - `public void sendMessage() {`
    - `System.out.println("I am sending a message")`
    - `HelloWorldMessage message = HelloWorldMessage.builder().id(UUID.randomUUID()).message("Hello World").build();`
    - `jmsTemplate.convertAndSend(JmsConfig.MY_QUEUE, message);`
  - `}`
- `}`

# Receiving JMS Messages

- @Component
- Create a class called HelloMessageListener
  - // @Payload    Deserialize the Component
  - // @Headers    Get the message Headers
  - 
  - @JmsListener(destination =JMSSConfig.MY\_QUEUE)
  - public void listen( @Payload HelloWorldMessage helloWorldMessage, @HeaderMessageHeaders headers, Message message) {
    - System.out.println(helloWorldMessage);
  - }
  - If an exception is thrown JMS will try to deliver the message if configured correctly It will set the redelivered property to true and increase the DeliveryCount by 1
    - It will get redelivered

# JMS Send and Receive a Reply

- `@RequiredArgsConstructor @Component public class HelloSender {`
  - `private final JmsTemplate jmsTemplate;`
- - `@Scheduled(fixedRate = 2000 )`
  - `public void sendMessage() {`
    - `System.out.println("I am sending a message")`
    - `HelloWorldMessage message = HelloWorldMessage.builder().id(UUID.randomUUID()).message("Hello World").build();`
    - `Message jmsTemplate.convertAndSend(JmsConfig.MY_SEND_RECV_QUEUE, new MessageCreateor() {`
      - `@Override`
      - `public Message createMessage(Session session) throws JMSException {`
        - `Message helloMessage = session.createTextMessage(objectMapper.writeValueAsString);`
        - `Message helloMessage.setStringProperty("_type", "guru.springframework.sfgims.model.HelloWorld); // how to deserialize it`
        - `return helloMessage`
    - `}`
    - `System.out.println(receivedMsg.getBody(String.class));`
  - `}`
- `}`

# JMS Send and Receive

- In another class add
  - `@JMSListener( destination = JmsConfig.MY_SEND_RCV_QUEUE)`
  - `public void listenForHello( @Payload HelloWorldMessage, @Headers headers, Message message) {`
  - `HelloWorldMessage payloadMsg = HelloWorldMessage.builder().id(UUID.randomUUID()).message("World!!!").build()`
    - `jmsTemplate.convertAndSend((Destination) message.getJMSReplyTo(), payloadMsg)`
  - `}`

# Running Active MQ Docker Using Local Activemq Broker with SpringBoot

- `vromero/activemq-artemis-docker`
- `docker --rm` – remove upon termination
- `username/password` – artemis `/simetraehcapa`
- Using Local ActiveMQ Broker with Spring Boot
  - If you have `artemis-jms-server` and `artemis` server Spring Boot will setup up JMS automatically
  - in `application.properties` add `spring.artemis.user`, `spring.artemis.password`

# JMS and Spring Message Data Types

- There are two types of message
  - `javax.jms`
  - `org.springframework.messaging` // Better for portability, but requires a typecase
-



# Spring Cloud

- Spring Cloud provides tools for developers to develop to build common patterns in Microservice
  - Functionality
    - Load Balanced Service
    - Discovery
    - Service Scaling
    - Service Registration
    - Deployment
- Many of these features are included in Spring Boot which extends Spring Cloud – ex actuators
- Service Discovery
  - Writing some code that invokes a service that has a REST API your code needs to know the service instance location
    - A DNS record that maps the URL to IP Address and port
  - In microservices this becomes problematic as service instances are dynamically assigned network locations upon registration and upon auto-scaling ( copying ) as client traffic rises
- Client Discovery
  - The client is responsible for determining the network locations of available server instances
  - The client queries the service registry, a database of available service instances, Each registry has the DNS/IP Address entry specifics
  - The client uses a load balancing algorithm to select one of the available service instances

# Spring Cloud

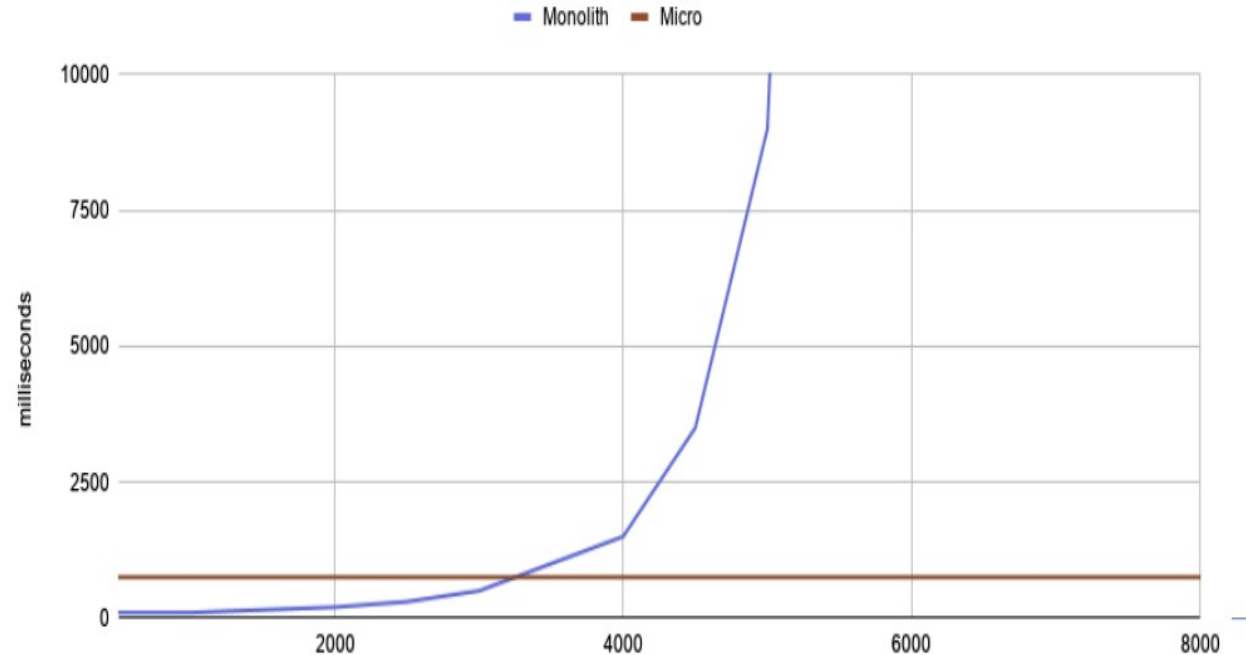
- In a distributed environment calls to remote resources and services can fail
  - due to slow network connections, timeouts or resource being unavailable
  - It might be pointless
    - Prevents an application from repeatedly trying to execute an operation that is likely to fail
    - Allows the application to continue without waiting for the fault to be fixed by calling the fallback operation
    - Can detect whether the fault has been resolved If the problem appears to have been fixed the application can try to invoke the original operation again.
  -

# Spring Cloud Gateway

## API Gateway Pattern

- The API Gateway Pattern
  - All the clients of the microservice go through it
  - single endpoint for your application
- Simplest form of an API Gateway is a load balancer
- microservice – linear scalability
  - Take a hit on performance, but don't get the hockey stick on the right
- Responsibilities
  - Routing / Dynamic Routing
  - Security
  - Rate Limiting
  - Monitoring / Logging
  - Blue / Green Deployments – Goes out to 10% until the new microservice is good.
  - Caching
  - Monolith Strangling – flexibility and a lot of creativity that you can apply to how you handle client request coming in.

Monolith vs Microservices - Request vs Response Time



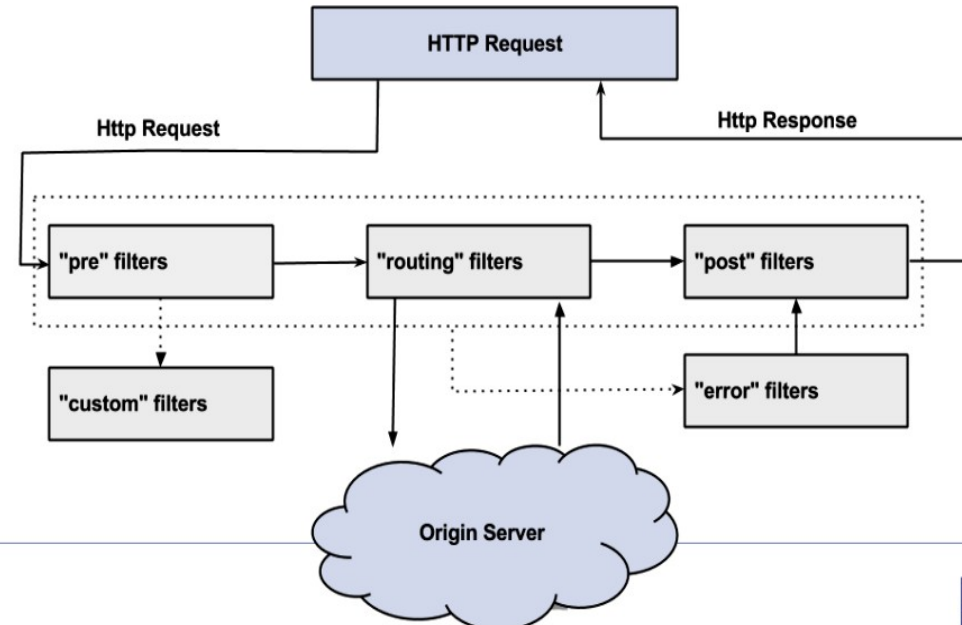
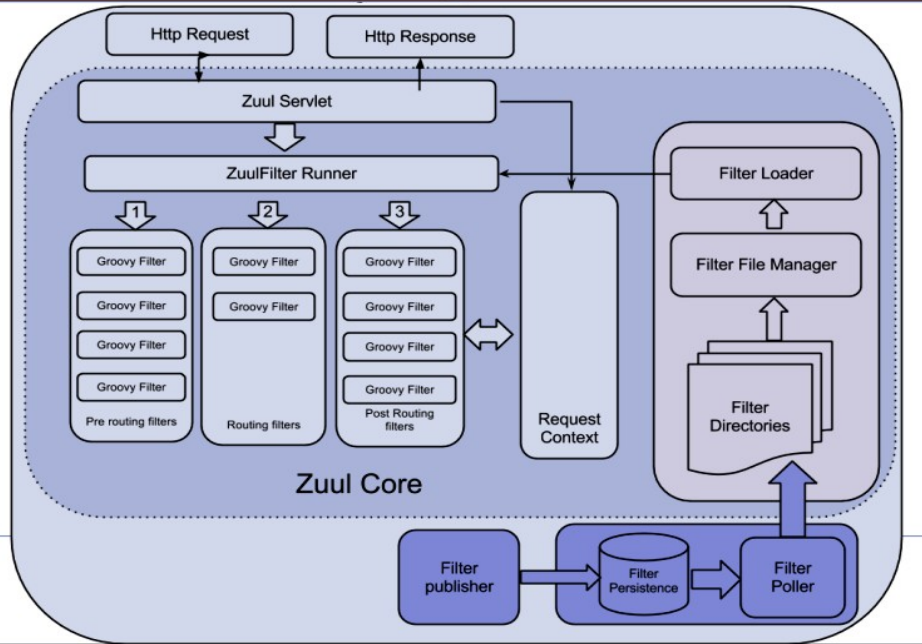
# Spring Cloud Gateway

## API Gateway Pattern

- Type of Gateways
  - Appliance / Hardware – Example F5
  - SAAS – AWS Elastic Load Balancer
  - Web Servers – configured as proxies
  - Developer Orientated – Zuul or Spring Cloud Gateway
  - These type can be combined.
- Single Point – Can apply Security, Logging, Monitoring

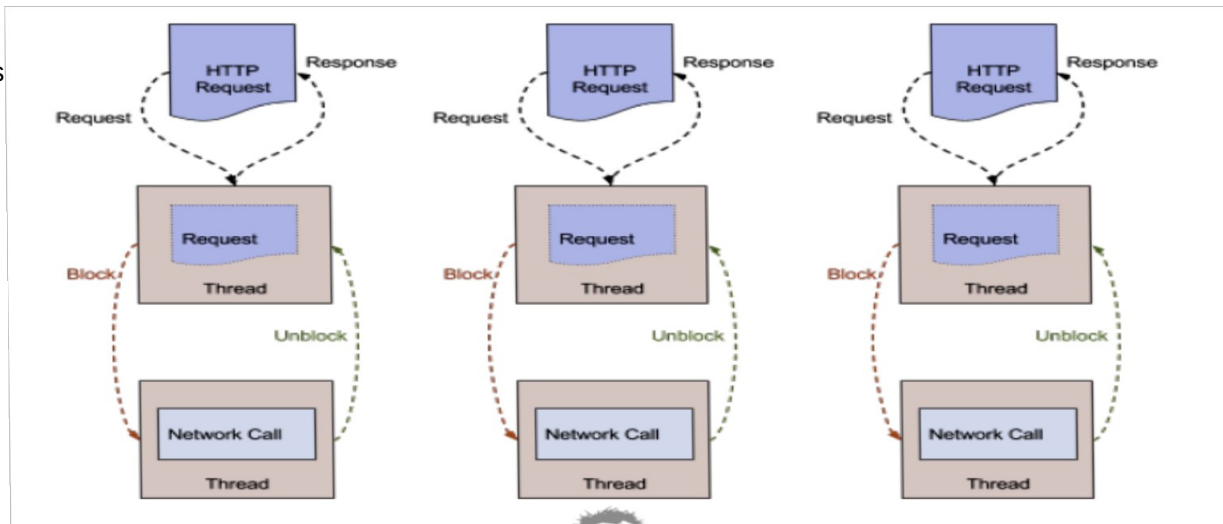
# Developer Orientated Gateways

- Zuul
  - Edge service in the cloud
  - 1000 client types and 50,000 requests per second
- Architecture of Zuul ( provides filter to get thing in and out )



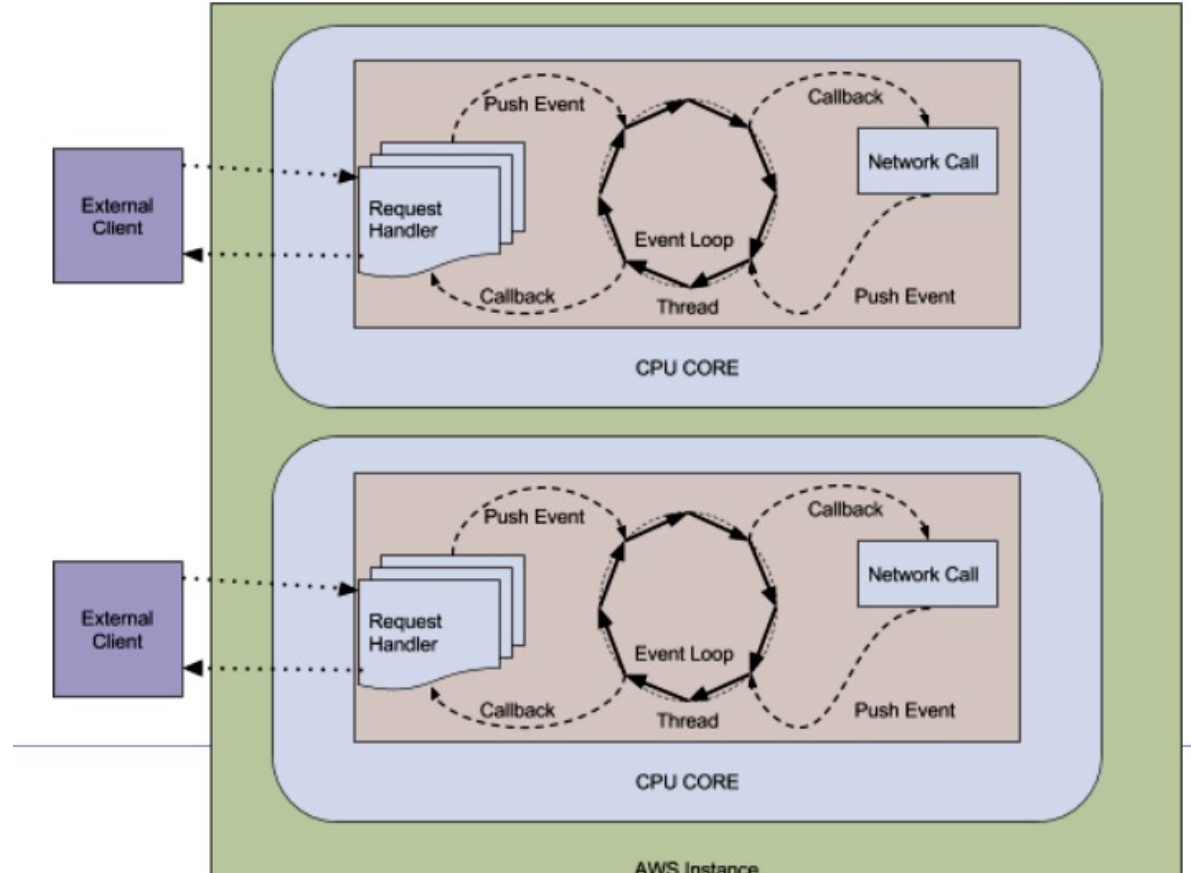
# Developer Oriented API Gateways

- Used Java HTTP Servlet
  - Blocking – Inefficient
  - Did not support HTTP2
- Zuul 2
  - Non Blocking much more efficient
  - Support for HTTP2
- In a Blocking environment you have multiple thread. The thread blocks waiting for a network call responds Scalable , but inefficient since the thread is waing



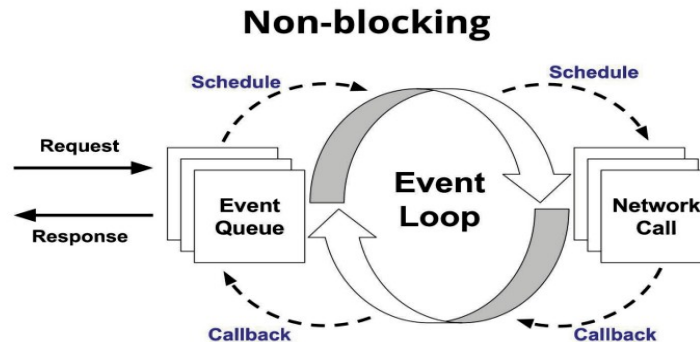
# Developer Oriented API Gateways

- Non Blocking – Have an event loop waiting for work → and it will push an event then rather than waiting tags it for a callback
- Spring Cloud Gateway
  - Java 8 / Spring Framework 5 / Spring Boot 2
  - Non blocking Http support netty
  - Dynamic routing
  - Route Mapping on HTTP Request Attributes
  - filters for HTTP Request and Response

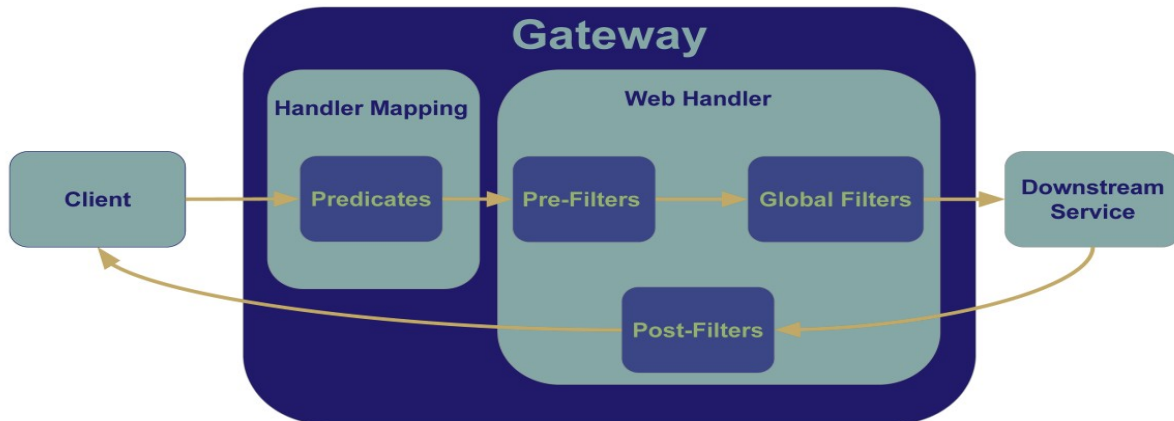


# Developer Oriented API Gateways

- Reactive Gateway
  - Same concept as Zuul two
- Gateway Flow
  - similar to Zuul
- Depending on your architecture you might not use a gateway, but you might be deploying AWS Elastic load balancer



## Gateway Flow





# Spring Cloud Gateway Service Creation

- Add Spring Cloud Gateway
- under resources set `spring.application.name` and server port
- The Spring Gateway will use all the reactive so the netty server will be used
  - Netty is an asynchronous event driven network application framework
  - Makes our gateway much more efficient in how it handles resources

# Spring Cloud Gateway Service Creation

- Gateways are effectively a proxy.
- Setup a google proxy example
  - Access google through our gateway
  - Will use the Spring Boot Parent Pom : Spring Cloud Routing
- Example
  - @Configuration
  - @profile("google") – Add to the running of the program
  - ```
public class GoogleConfig {  
    • // When a request comes in with the path /googlesearch then go to the gateway  
    • // Google told the browser that there is a redirect so it does go through the server once it been cached  
    • // For the first time will print a log message with Mapping Exchange Get to Route  
    • @Bean  
    • public RouteLocator google(RouteLocator builder) {  
        - return builder.routes()  
        - //route( r → r.path("/googlesearch") // When google search go to the URI  
        - .uri("https://google.com"))  
        -  
        - // A Second Example  
        - .route( r → r.path("/googleSearch2").filters(f → f.rewritePath("googlesearch2(<segment>?<.*>","${segment}"))  
            • // Rewrite the path for google search – generates a 301  
        -  
        - .id("google")  
        - .build()  
    }  
}
```

Service Registration with Eureka

- Introduction to Eureka
 - Netflix Eureka is a Service Discovery and registration Service
 - When a microservice instance starts it register itself with the Eureka Service
 - Provides host name / IP, port and service name
 - Know as service registration
 - Service Discovery is the process of discovering the available service instance
- Flow
 - The service will register with Eureka
 - Another service will lookup the service or query
 - Using the data from Eureka you can start utilizing it.
- Service Registration
 - Spring Boot provides a Spring Boot Starter Eureka Server
 - Spring Boot also provides a spring starter for Eureka Clients
 - both will self configure for localhost operations
 - configure the service name in `application.properties` – value to lookup the service in Eureka

Service Registration with Eureka

- Service Discovery
 - Spring Cloud Open Feign allow for easy service discovery between the microservices
 - Works in connection with Eureka and Ribbon
 - Spring Cloud Gateway can be configured to lookup service in Eureka
 - Works in conjunction with Ribbon to load balance requests
- Eureka Service Creation
 - @EnableEurekaServer – need spring-cloud-netflix-starter-eureka-server
 - setting
 - server.port
 - eureka.client.register-with-eureka = false
 - eureka.client.fetch-registry = false // Designed to run as cluster so this will allow to fetch other registries
 - logging.level.com.netflix.eureka = OFF
 - logging.level.com.netflix.discovery = OFF
 - @EnableEurekaClient
 - Uses the spring.application.name as the service name
 - Create a class @Profile("local-discovery") @EnableDiscoveryClient @Configuration public class LocalDiscovery {}
 - // If local-discovery not set does register
 - spring.application.name
 - eureka.client.service.defaultZone - http://<ip address and port>eureka
 - can use \${spring.app.name} - \${random.int}

Service Discovery with Eureka

- Open Feign Client – FeignClient – Creates REST API Clients in a declarative way
 - spring-cloud-starter-netflix-openfeign
 - `@EnableFeignClients` on the SpringBean Does not work on a Configuration class, but must be put on class with main from SpringBoot
 - Replace RestTemplate `@Profile(!local-discovery)`
 - Example
 - `@FeignClient(name="inventory-service")` // name is the Eureka Service Name
 - `public class InventoryServiceFeignClient {`
 - `// INVENTORY_PATH is the URL`
 - `@RequestMapping(method=RequestMethod.GET, value = BeerInventoryServiceRestTemplateImp.INVENTORY_PATH)`
 - `ResponseEntity<BeerInventoryDto> getOnhandInventory(@PathVariable UUID beer)`
 - `}`
 - `// Create a new service`
 - `@Service @Slf4j @RequiredArgsConstructor @Profile("local-discovery`
 - `public class BeerInventoryServiceFeign implements BeerInventoryService {`
 - `ResponseEntity<List<BeerInventoryDto>> responseEntity = inventoryServiceFeignClient.getOnhandInventory(beerId);`
 - `Integer onHand = Objects.requireNonNull(responseEntity.getBody()).stream().mapToInt(BeerInventoryDto::getQuantityOnHand).sum();`
 - `return onHand`

Configure Gateway for Service Discovery

- For Eureka Gateway does not need to register itself. `eureka.client.register = false`
- To the RouteLocator
 - to the uri add `lb: //beer-service`

Circuit Breaker Pattern

- Circuit Break Pattern Overview
 - Algorithm
 - If service is unavailable or throwing unrecoverable errors
 - specify an alternative action
- Spring cloud circuit breaker is a project which provides abstractions across several circuit break implementations
 - Your code is not tied to a specific implementation
- Supported
 - Netflix Hysterix Currently in Maintenance Mode
 - Resilience 4J Recommended
 - Sentinel
 - Spring Retry
 - Spring Retry
- Spring Cloud Gateway Circuit Breakers
 - Supports Cloud Gateway Netflix Hystrix and Resilience4j
 - Gateway filters are used on top of Spring Cloud Circuit Breaker APIs

Using Hystrix Circuit Breaker with Feign Client

- Note that *.properties get picked up if the local file is active.
 - profile local_discovery get application_local_discovery.properties
- properties
 - feign.hystrix.enabled = true
- Create a new interface InventoryFailoverFeignClient
 - @FeignClient added to the top of the interface with attributes (name of service ex. inventory service and fallback)
 - @RequestMapping(method = RequestMethod.GET, value="/inventory-failover")
 - ResponseEntity<List<BeerInventoryDto>> getOnhandInventory();
- Create InventoryServiceFeignClientFailover implements InventoryServiceFeignClient – just another Feign client
 - @RequiredArgsConstructor
 - @Component
 - private InventoryFailOverFeignClient failoverFeignClient;
 - getOnhandInventory function which return ResponseEntity<List<BeerInventoryDto>> – failoverFeignClient.getOnhandInventory();
- On the InventoryServiceFeignClient we have add fallback to InventoryServiceFeignClient
 - to the Feign client annotation --add fallback = failoverClient
- Could go through the Gateway to.

Overview of Spring Cloud

- Spring Cloud Config provides externalized configuration for distributed environments
- Provides a restful style for API for Spring Services to lookup configuration values
- Spring Boot application on startup obtain values from Spring Cloud Config Server
- Properties can be global and application specific
- Properties can be stored by Spring Profiles
- Easily encrypt and decrypt properties
- provides tooling for unencrypt and encrypt
- Provides a number of options for property storage
 - Git
 - File system
 - JDBC, Redis
 - AWS S3
 - HashiCorps Value

Spring Cloud Config Client

Create a Spring Cloud Config Server

- Spring Cloud Config Client Will look for a URL property
 - default is <http://localhost:8888>
- If using discovery client, client will look for service called config server
 - Can be configured to fail fast – if config server cannot be reached
- Can configure to fail with exception if config server cannot be reached
- Configuration Resources
 - served as /application/profile/label
 - .application spring.application.name
 - .profile .profile
 - .label spring.cloud.config.label
- Create Spring cloud Config Server
 - msvc-beer-service – port 8080
 - msvc-beer-inventory-service – 8082
 - msvc-inventory-8083
 - msvc-beer-order-service – 8081
- Spring cloud
 - Gateway 9090
 - Eureka 8761 which is default for Eureka
 - config-server 8888

Create a Spring Cloud Config Server

- Spring boot initialization for a Config Server
 - Spring Boot Starters for : Config, Eureka Discovery Client
 - `@EnableCofnigServer`
 - application properties

Spring Cloud Config Server

- To the file with SpringBootApplication annotation
- Properties
 - server.port
 - spring.application.name // Used for setting up with EUREKA
 - spring.cloud.config.server.git.uri // Where do we get the information from get it from the url line
 - spring.cloud.config.server.git.clone_on_start // default false. set to true either do in on startup or initial request
 - logging.level.org.springframework.cloud = debug
 - logging.level.org.springframework.web = debug
 - spring.cloud.config.server.git.search-paths={application} // Look in the folder of the application name for the properties/values
- Can trouble shoot just like a Spring MVC Application
- On Postman with Eureka running try : <http://localhost:8888/anyapp/default> and <http://localhost:8888/anyapp/someactive>
-

Sever Side Application Configuration

- Configuration Endpoints available endpoints – different options to support properties
 - `/{{application}}/{{profile}}/{{label}}` create beer-service directory as the application-local (where local is a profile)
 - `/{{application}}-{{profile}}.yaml`
 - `{{label}}/{{application}}-{{profile}}.yaml`
 - `/{{application}}-{{profile}}.properties`
 - `/{{label}}/{{application}}-profile.properties`
- if no profile then /application/default then you get properties
- Create a msce-brewery-config-repo=
 - Create a directory for the brewery beer service where the properties for that
 - referencing the actual service
 - In the folder add copy the application-localmysql.properties and rename to application-local.prooperties
 - do a commit and push
- Add a new property `spring.cloud.config.server.search-paths={{application}}` – Where to look
- <http://localhost:8888/beer-service/default>
- <http://localhost:888/beer-service/local> When the file is local it is picked up
-

Spring Cloud Config Client Configuration

- When we deploy into a cloud type environment our service can wake up and know the location of the eureka server and then find the Spring Cloud Config
- Need spring-cloud-starter-config Spring Boot Dependency
- Spring Cloud uses a bootstrap startup to find the environment and then go through all the normal stuff
 - In the beer service create a bootstrap-local-discovery properties with 2 properties
 - `spring.cloud.discovery.enabled=true`
 - `spring.cloud.config.discovery.service-id` must match the name in msssc-config-server
- When getting data from the my local database make sure you have `spring.cloud.discovery.enabled=false` so we don't have the overhead of the cloud.
- Used profiles to allow how to have different configurations

Introduction To Distributed Tracing

- Services could span many different data center/companies and parts of huge chains
- Distributed Tracing is used in two aspects : Performance / Logging / Trouble Shooting
- Spring Cloud Sleuth
 - Distributed Tracing Tool for Spring Cloud
 - Uses a open source distributed tracing library called Brave
- Conceptually
 - Headers on HTTP Request or message are enhanced with trace data
 - Logging is enhanced with Trace Data
 - Trace Data can be reported to ZipKin
- Terminology – Used by Dapper (Distributed tracing system created by Google)
 - Span – Basic unit of work. Typically a send and receive of a message
 - Trace – A set of spans for a transaction
 - cs / sr Client Sent / Server Received aka the request
 - ss/ cr Server Sent / Client Received – aka the response

Introduction To Distributed Tracing

- Zipkin is project use to report distributed tracing metrics
- Information can be reported to Zipkin via webservies via HTTP
 - Optimally metrics can be provide via Kafka or Rabbit
- A Spring MVC Project
 - Recommended to use binary distribution or Docker image
 - Building your own is not supported
 - Uses in memory database for development
 - Cassandra or Elasticsearch should be used for production instead of the In Memory
- Quick Start via curl
 - `curl -sSL http://zipkin.io/aquickstart.sh | bash -s`
 - `java -jar zipkin.jar`
- Quick Start via docker
 - `docker run -d -p 9411:9411 openzipkin/zipkin`
- View traces in UI at: http://your_host:9411/zipkin
- `org.springframework.cloud:spring-cloud-starter-zipkin` – get `org.springframework.cloud:spring-cloud-start-sleuth` for free, but should be add to the dependencies to follow the 12 step
- Property : `spring.zipkin.baseUrl` is used to configure Zipken Server – It is the URL
 - For each service

Introduction To Distributed Tracing

- Zipkin is project use to report distributed tracing metrics
- Information can be reported to Zipkin via webservies via HTTP
 - Optimally metrics can be provide via Kafka or Rabbit
- A Spring MVC Project
 - Recommended to use binary distribution or Docker image
 - Building your own is not supported
 - Uses in memory database for development
 - Cassandra or Elasticsearch should be used for production instead of the In Memory
- Quick Start via curl
 - `curl -sSL http://zipkin.io/aquickstart.sh | bash -s`
 - `java -jar zipkin.jar`
- Quick Start via docker
 - `docker run -d -p 9411:9411 openzipkin/zipkin`
- View traces in UI at: http://your_host:9411/zipkin
- `org.springframework.cloud:spring-cloud-starter-zipkin` – get `org.springframework.cloud:spring-cloud-start-sleuth` for free, but should be add to the dependencies to follow the 12 step
- Property : `spring.zipkin.baseUrl` is used to configure Zipken Server – It is the URL
 - For each service

Intro to Distributed Tracing

- Logging output get the following
 - DEBUG [Appname, TraceId, SpanId, Exportable]
 - exportable – Should span be exported to zipkin or not
 - Generated by Sleuth
 - Trace id is tied to a particular request
 - Span id is unique id per microservice
- Microservices typically use consolidated logging
 - Should be available in JSON to make it easier
 - Spring uses logback

Zipkin Server

Setup Spring Cloud Sleuth

- zipkin.io
- Setup Spring Cloud Sleuth
 - For the Beer Service – Add the dependency spring-cloud-starter-zipkin. It will be added in if spring-cloud-starter-sleuth is added
 - In the gateway properties : spring.zipkin.baseUrl = <http://localhost:9411>
 - For the Gateway add the above dependency as well – zipkin
 - In the gateway properties : spring.zipkin.baseUrl = <http://localhost:9411>
 - Check that the command line has debug with zipkin traces so you can look at the trace ids
 - For IntelliJ Maven use Reimport if it did not pick up the changes to the pom
- Logging Config for JSON
 - Add net.logstash.logback:logstash-logback-encoder
 - An open source library that will grab log file that will be moved to consolidate logging server
 - Setup a custom configuration for logstash
 - logback-spring.xml – See the next page.

Logging Config for JSON

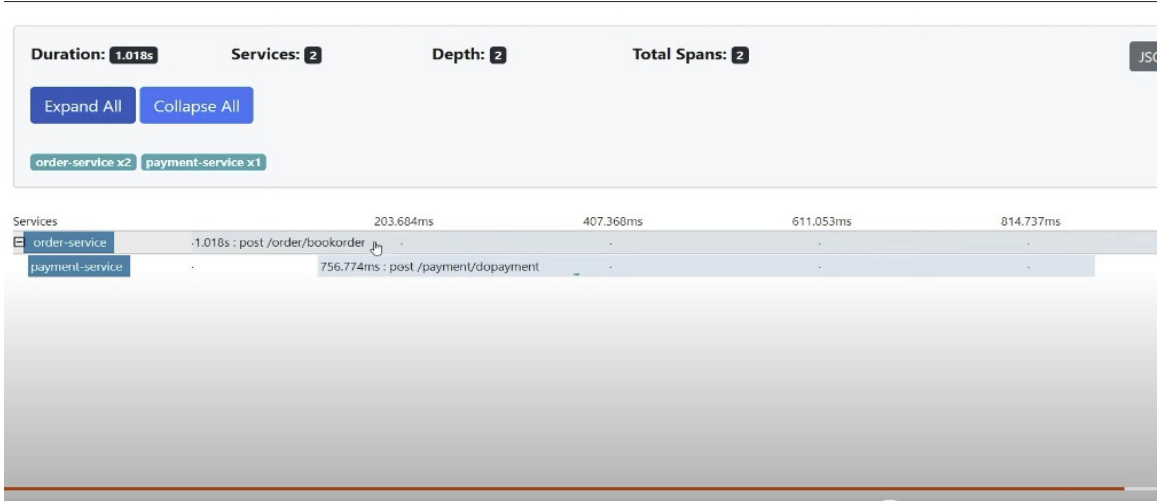
Refactor Zipkin Server

- mscs-brewery-config-repo we are adding
 - Add the variables to this project
 - In mssc-brewery-config-repo add `spring.zipkin.baseURL`, `spring.zipkin.enabled=true` // Will override in application.properties
- In each of the services setting `spring.zipkin.enabled = false`

The screenshot shows the Zipkin web interface in a browser window at `localhost:9411/zipkin/`. The interface has a dark header with navigation links: "Investigate system behavior", "Find a trace", "View Saved Trace", and "Dependencies". There is a "Try Lens UI" button and a "Go to trace" input field with a "Search" button. Below the header is a search form with the following fields:

- Service Name:** A dropdown menu with "all" selected. Below it is a search input field with a magnifying glass icon and a list of suggestions, including "all" and "d cluster=foo and cache.miss".
- Span Name:** A dropdown menu with "Span Name" selected.
- Lookback:** A dropdown menu with "1 hour" selected.
- Duration (μ s) \geq :** A text input field with "Ex: 100ms or 5s" as a hint.
- Limit:** A text input field with "10" entered.
- Sort:** A dropdown menu with "Longest First" selected.

At the bottom of the form is a blue "Find Traces" button and a help icon (question mark in a circle). Below the form is a light blue banner with the text: "Please select the criteria for your trace lookup."



Security 101

- Analyzing risk vector then implementing mitigating actions
- Mitigating action
 - OS Patching to prevent known exploits
 - Password Length and complexity
 - Firewalls – Only expose necessary ports to internet
 - prevent direct sign on to super accounts
 - Use OS Level security features to restrict Access
 - Read access to email data files should be highly restricted
 - Other types of security – Physical Security, Personal Security – Only people who need to access the server should have access, Segregation of Duties – people should have limited access for those roles (Department managers should not have super user accounts)
 - Phishing
 - 2 Factor Authentication help prevent
 - password expiration policies
 - Network Security
 - An organization will have more specialized resources.
 - Use and update anti-virus software
 - Need to know
 - Assign a unique id to each person with computer access
 - Implement logging and log management
 - Conduct vulnerability scans and penetration test
 - Services account should have minimal access

Security

- CSRF
 - Example use change the address on a hacker
 - Cause the victim to carry out an action unintentionally
 - such as funds transfer
 - Fix: CSRF Token with relevant requests

Property Encryption / Decryption

- At Rest Encryption
- Java Cryptography Extension – JCE – part of Java 11
- Spring Cloud Configuration will store encrypted properties as
 - {cipher}<your encrypted value here>
- When a Spring Cloud Config client request an encrypted property the value is decrypted and presented to the client in the request
 - Setup some sort of HTTP tunnel to the client
- Must be a symmetric key in property 'encrypt.key' should prefer setting this as environment variable
- asymmetric (public /private keys)_ are supported.
- Spring cloud config provides endpoint for property encryption / decryption
 - POST /encrypt – will encrypt body of post
 - POST /decrypt – will decrypt body of post
-

Property Encryption / Decryption

Encrypt Beer Service Passwords

- <http://localhost:8888/encrypt> – Sent to Spring Cloud configuration
 - Example Post Data is MyPssword – Error encryption algorithm is not strong enough
 - The private key should not be in the properties key, but an environment variables
 - add encrypt.key to boot.properties and not it will be able to encrypt
- SpringBoot has a 2 phase startup
 - SpringBoot.proeprites
 - application.properties
- <http://localhost:8888/encrypt>
- Encrypt Beer Service Passwords
 - create application-local.properties
 - In configuration server
 - add the encrypted value you received from the Spring Cloud
 - spring.dtasoruce.passsword={cipher}<string>
 -

Secure Spring Cloud Server

- To the mssc-config-server add
 - `org.springframework.boot:spring-starter-security`
 - set basic security with random password
 - To make non random add to application properties : `spring.security.user.name`, `spring.security.user.password`
 - `spring.security.user.name`, `spring.security.user.password` – causes http authentication
 - `spring.cloud.config.fail-fast = true`
 - `// Start up one of the other applications` – The client will fail right away
- Setup the client
 - Add `spring.cloud.config.username` / `spring.cloud.config.password`, `spring.cloud.config.failfast`
 - Cannot talk to the Spring Cloud Server the application will fail.

Use Spring Security to Secure Eureka Server

- For mssc-brewery-eureka
 - need spring-boot-starter-security
- In application properties
 - spring.security.user.name
 - spring.security.user.password
 - generates a random password
- Create a package called config
 - @Configuration
 - public class securityConfig extends WebSecurityConfigurerAdapter
 - protected void configure(HttpSecurity http) throws Exception {
 - http.csrf().disable().anyRequest().authenticated().and().httpBasic() // disable – Disable CSRF - Cross Site Request Forgery
- In mssc config server add the configuration for the client to talk to each other.
 - add the following eureka.client.service-url.defaultZone=http://netflix:eureka@localhost:8761/eureka
 - where netflix is the user and eureka is the password

Consolidate Logging with Elk Stack – Overview

- Elk stands for
 - E – Elastic Search
 - JSON Search Engine based on Lucene
 - L – Logstash
 - Allows you to collect data from multiple sources, Transform, Send
 - K – Kibana
 - Data Visualization for Elastic Search
 - Can Query Data and Act as an Dashboard
 - Create charts, graphs and alerts
- Filebeat is a log shipper
 - Move logs to a destination
 - Often destination is a logstash server
 - Logstash is used for further transformation before sending to ElasticSearch
 - Filebeat has the capability to do some transformations – possible to skip logstash and write directly to ElasticSearch
 - Convert JSON logs to JSON object for Elastic search

Add elastic Search View Logs in Kibana

- Kibana home is <ip>/app/kibana#home/
- Click on the discovery icon
 - Set up an index which is your filebeat
 - @timestamp
- Click on Discovery again and can see the log messages
 - trace.trace_id = <trace id>
 - Can see JMS Requests and web Requests

Logging Configuration Update

Security 101

- Security Audit Framework / Certifications
 - PCS-DSS – Payment Card industry data security standard
 - SOX – Sarbanes-Oxley
 - HIPAA
 - SSAE-16
- Common Terminology
 - PII
 - Encryption at rest – Sensitive data needs to be encrypted when store
 - Encrypt in flight
 - Segregation of Duties
 - Process and Control – Be able to document compliance

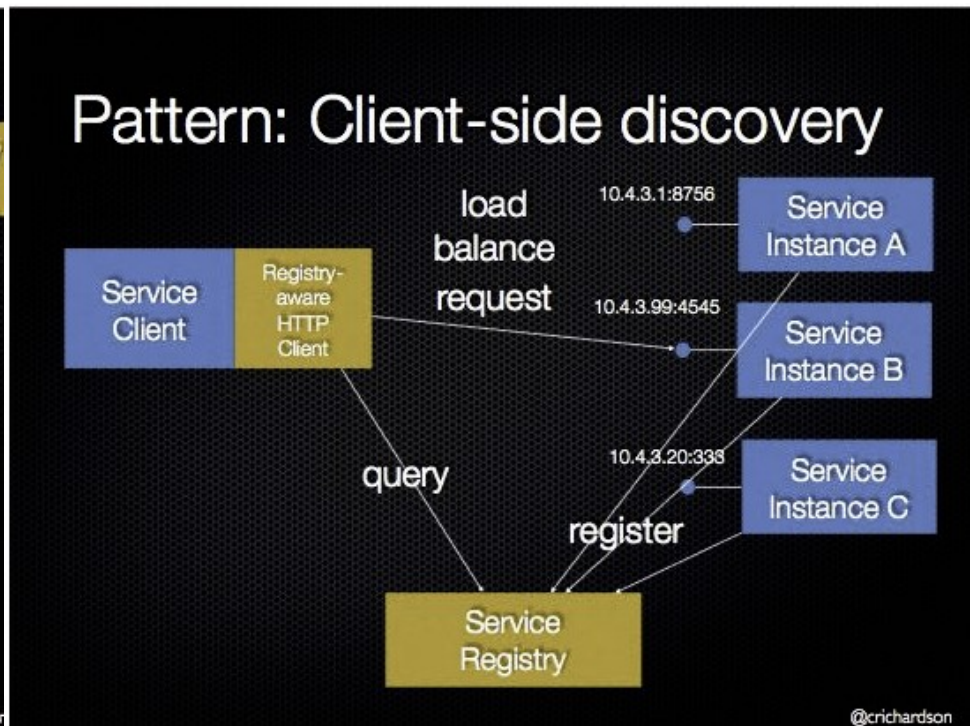
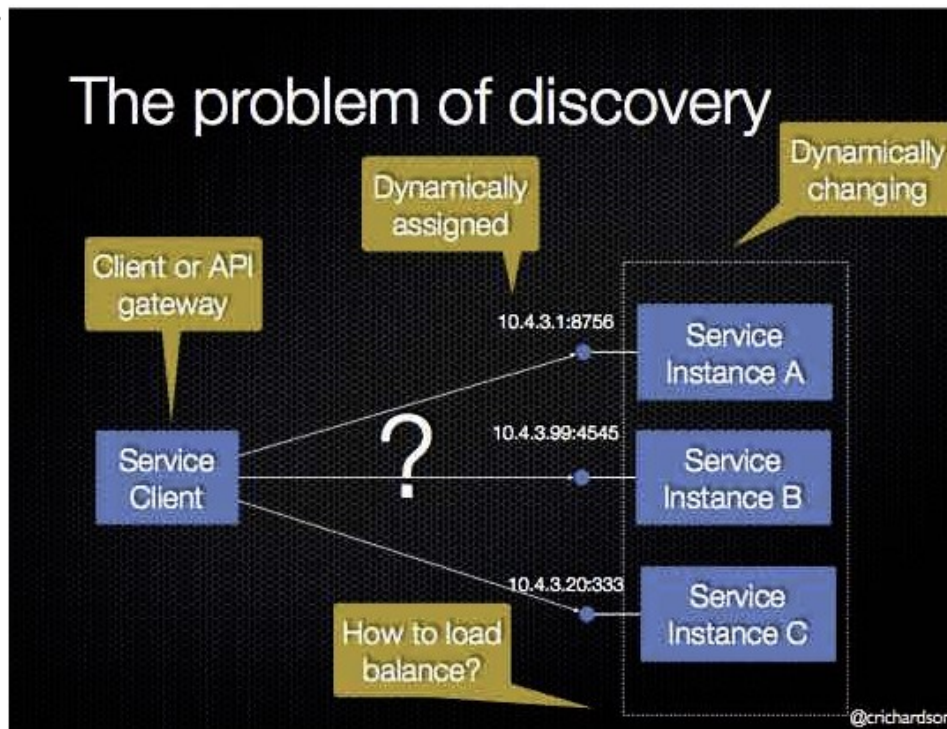
-

Pattern Service Registry

- Client of a service use either client side or server side discovery to determine location of a service instance to which to send requests
- Problem : How do clients of a service (client side discovery) and/or routers (Server side discovery) know about the available instances of a service
- Forces
 - Each instance of a service exposes a remote API at a particular location (host and port)
 - The number of services and their locations changes dynamically
- Solution : Implement a service registry
 - A databases of services
 - Good location for a health check API
- Examples – Eureka, Apache Zookeeper, Consul, Etcd
- Disadvantages
 - Another infrastructure that must be setup/configured/managed
 - Client should cache data provided by the service registry since it could fail or become out of data
 - Service Registry must be highly available

Client Side Discovery

- Problem – How does the client of a service – the API gateway or another service discover the location of a service instance

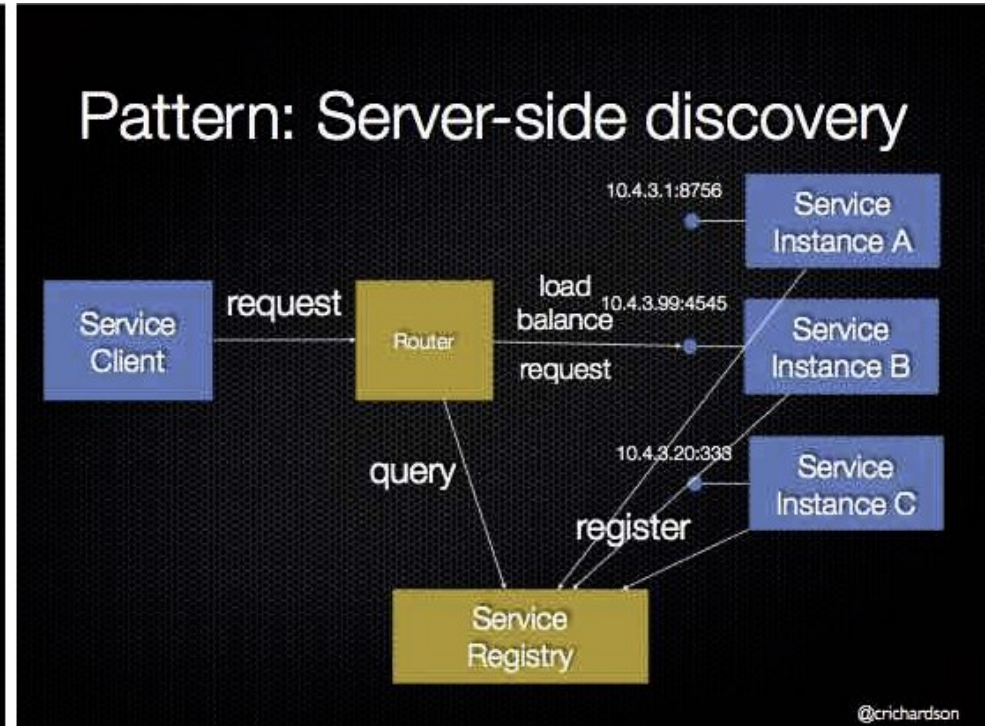
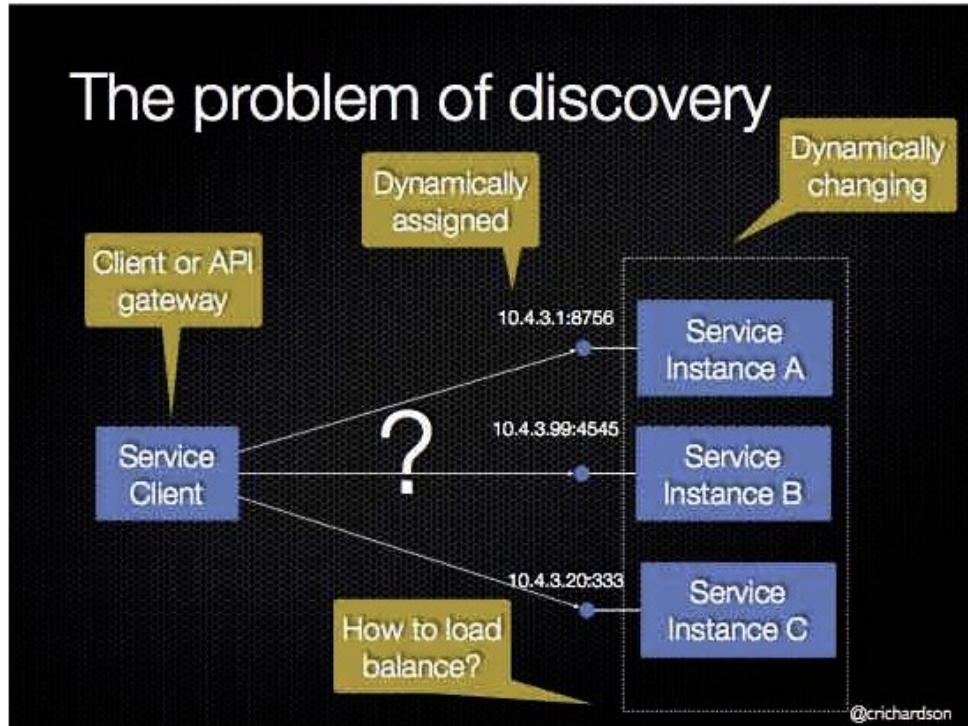


Client Side Discovery

- Fewer moving parts than server side discovery
- Disadvantages
 - Couples the client to the service registry
 - Need to implement client side service discovery logic for each programming language/framework used by your application
- Example – Netflix Prana

Server Side Discovery

- Problem – How does the client of a service discover the location of a service instance



Server Side Discovery

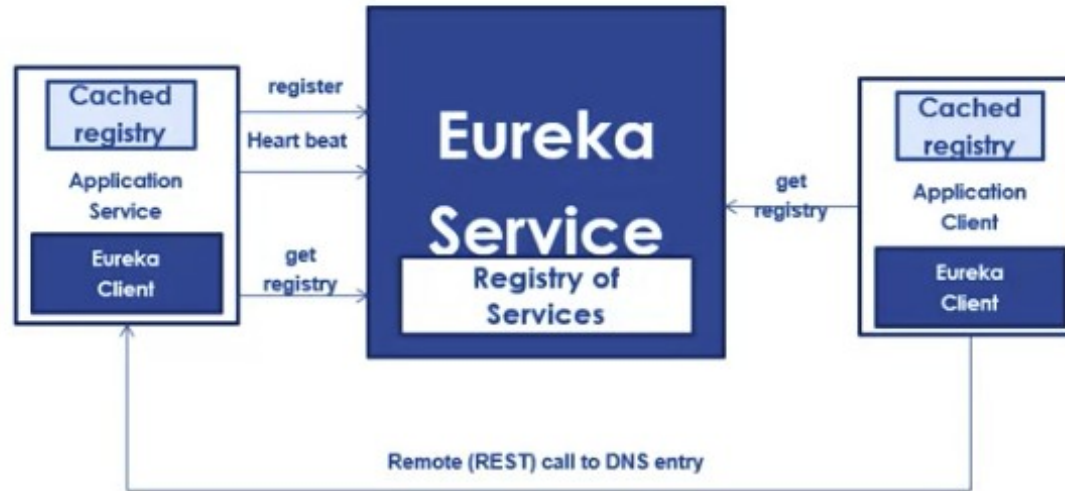
- The client does not deal with discovery but makes a request to the router
- Some cloud environments provide this functionality – AWS Elastic Load Balancer
- Draw Backs
 - Unless provided by the cloud environment it is another piece of the infrastructure create/maintained/monitored
 - Must support all following protocols
 - More network hops are required vs Client Side Discovery

Service Registries

- Netflix OSS is part of the Spring Cloud of projects
- Provides
 - Service Registration
 - Service to service Communication through load Balanced Service Discovery
 - Service Registration Removal
- Eureka
 - A Rest based service for locating services for the purpose of load balancing
 - Such service contains a registry of services that have registered themselves as being available to the Eureka Service
 - Continued heartbeat interaction from registered service is still active – If failed then remove from the registry
 - The Registry has the DNS/IP Address entry for the registered services and are retrieve using a service id
 - Eureka Client – simplifies interactions with the Eureka Server
- Eureka Server Registration
 - Eureka clients fetch (copy) the registry information from the Eureka Server and caches it locally with themselves
 - Sends a heartbeat to the master registry (keeps it in the registry) so it can pull the master registry from the Eureka Server
 - service to service communication – lookup up the service to get the DNS Entry and then call RestTemplate
 - If a service is scaled the registry has multiple services for each service id
- The code only knows the service name

Eureka Service Registration

Eureka Architecture



Service Discovery

- Load Balancing
 - Want to maximize availability of our service via scaling
 - Access to distributed services can be controlled via load balancing to optimize distribution of workloads across multiple services
 - Client Side Load Balancing
 - Delivers a list of server IP Address to the client and the client randomly selects the IP Address
 - Ribbon – Provides client side load balancers
 - When Ribbon is attached to your service the service will select from the local registry of service any entry f[u] the desired service and place them in a subset.
 - Use Round robin algorithm to direct the request to each service in the extracted list of services (subset)
 - Eureka with Ribbon
 - Ribbon client are typically created and configured for each of the registered services
 - When Eureka and Ribbon are used together (both on the classpath)
 - RibbonServerList – The subset or list of potential services from the local registry
 - Ribbon uses the Local Eureka Service Engine to get the list of acceptable service
 - DynamicServerListLoadBalancer is used to delegate the selected service instance
 -

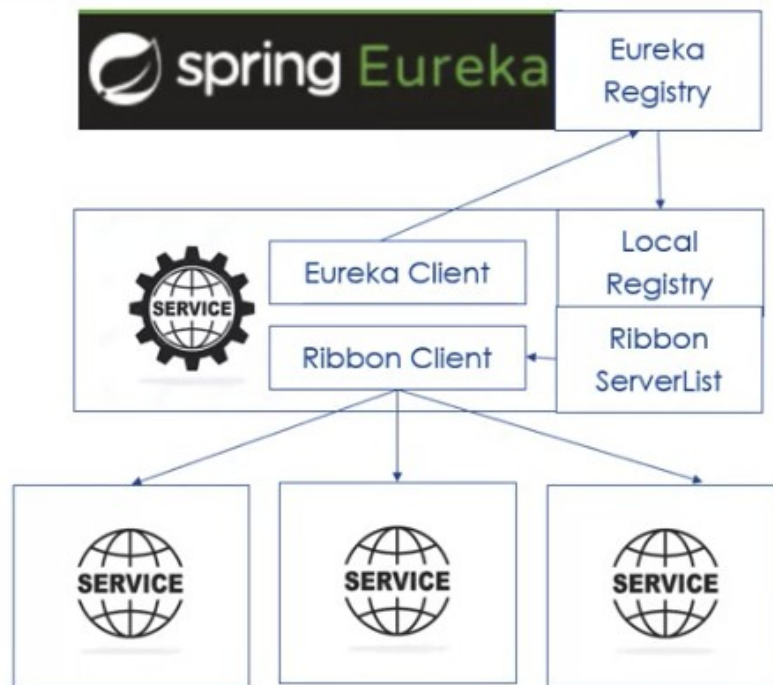
Service Discovery

- Each service is identified by its service id and any associated server instance in the Ribbon Server List.
 - Ribbon then delegates the request in round robin fashion to each service in RibbonServerList as requests are received by the Eureka Client

```
@SpringBootApplication
@EnableDiscoveryClient
.RibbonClient(name = "capitolServ")
@RestController("/")
public class Application_Service_Client {
    @Bean
    @LoadBalanced
    RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

1. name = "capitolServ" is the name found in the properties file : spring.application.name

2. restTemplate is both a Spring Component and the id is found in the configuration file



Service Discovery

- Configuration
 - Match the service name to the actual service id in the created local registry
 - This is service discovery since you do not know the IP Address or hostname of the service
 - As new instance are added to the registry the RibbonServerList get bigger and if an instance goes out of service it gets slower.
 - Remove when the system remove them or the Eureka heartbeat fails

```
@RibbonClient(name = "capitolServ")
```

```
capitolServ:  
  ribbon:  
    DeploymentContextBasedVipAddresses: capitol-service  
    NIWSServerListClassName: com.netflix.niws.loadbalancer.DiscoveryEnabledNIWSServerList
```

Service Discovery

Client Side Load Balancing

- Use the load balancing template
 - The url uses the service id and not the ip address of the service: <http://capitolServ> (instead of ip address and port)
 - Ribbon is using the locally cached Eureka list of services to secure a list of CapitolServ entries

```
@GetMapping(value="/cityInfo/{code}")
public String get(@PathVariable("code") String id) throws Exception {
    String url = "http://capitolServ/capitol/{id}";
    return this.restTemplate().getForObject(url, String.class, id);
}
```

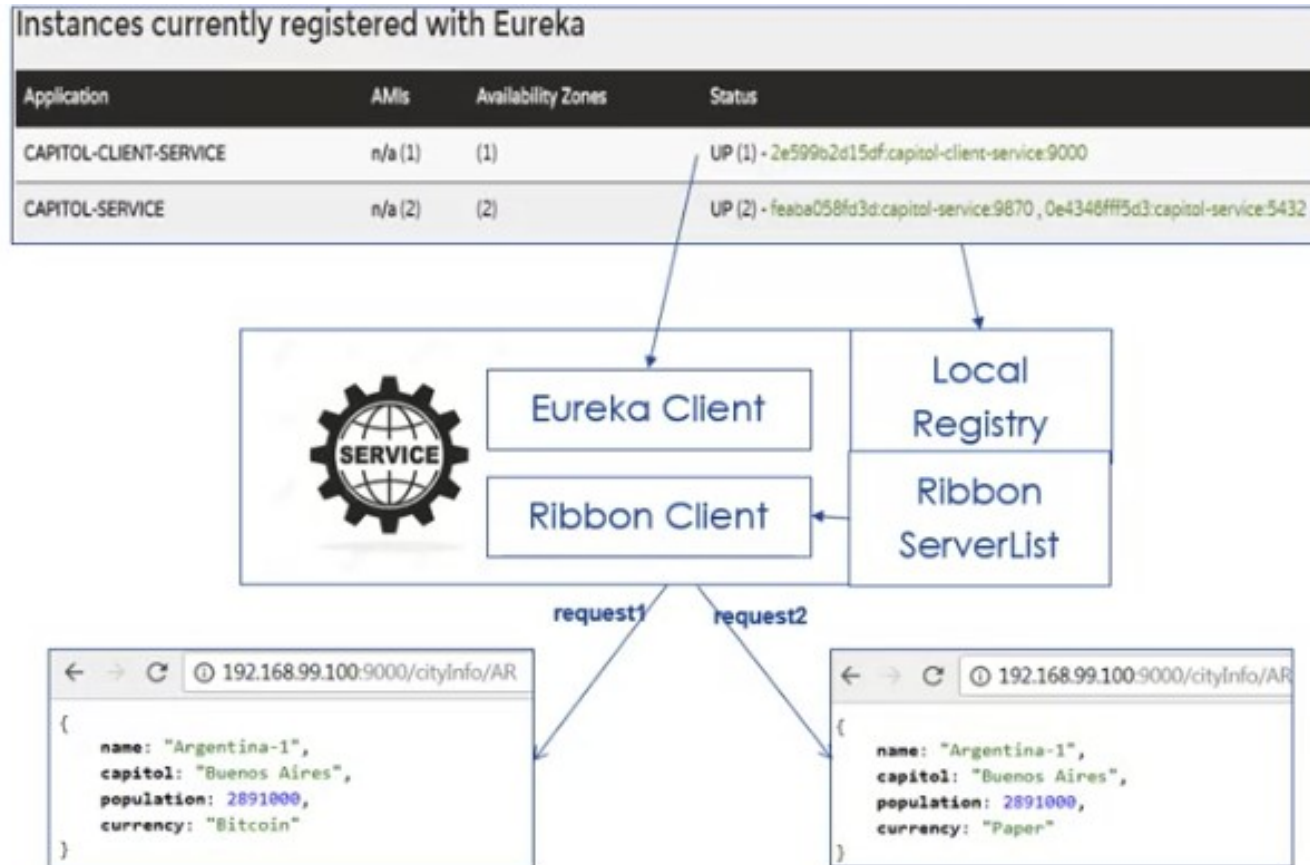
Client Side Load Balancing

If we start two instance of the capitol-service as below both will be registered with a running instance of the Eureka Service

The locally cached registry will receive two capitol service entries

Maximizing throughput and services throughput and services

Client Side Load Balancing



Client Side Load Balancing

- Problem – The tutorial is using ribbon
- Information comes from Baeldung – The only idea that I can find is that is easier to use
- Algorithms for load balancing
 - Random selection: Choosing an instance randomly
 - Round-robin: Choosing an instance in the same order each time
 - Least connections: Choosing the instance with the fewest current connections
 - Weighted metric: Using a weighted metric to choose the best instance (for example, CPU or memory usage)
 - IP hash: Using the hash of the client IP to map to an instance
-

Hystrix

- It isolates the points of access between the services --stops cascading failures across them and provides fallback options
- We could add the spring cloud and have Hystrix on our classpath
- Two Key annotations
 - `@enabledCircuitBreaker` trigger all the code necessary
 - `@hystrix(fallbackMethod="MethodName")` If the method fails we want to call a fallback method
- The implementation is a proxy – So if an exception is thrown it is usually a timeout or the service is not available
 - The proxy will verify if the service is available and if so will call the service else call the fallback method.

```
@EnableCircuitBreaker
public class Service01 {
    public static void main(String[] args) {
        SpringApplication.run(Service01.class, args);
    }
}
```

```
@Component
public class MountainDAO {
    @HystrixCommand(fallbackMethod = "defaultMtn")
    public Mountain get(Long key){
        //we could call an unavailable service here
        throw new RuntimeException("oh dear");
    }
    public Mountain defaultMtn(Long key){ //same signature
        return summits.get(4L);
    }
}
```



Feign

- Feign will add in creating REST client through alternative RestTemp to lookup your services
- Feign must be added to the classpath
- `@EnableFeignClients` annotation to the Application Class which
 - Provide the default configurations for the resolution of the Rest Service via Feign Implementation

- Feign clients

- Create an interface and add the interface to the `@Feign Client`
- A proxy implementation of the interface will be created at runtime
 - find the endpoint from the Eureka Registry
 - `@GetMapping` will identify the endpoint we need
 - The proxy will delegate the real service instance through a ribbon load balancer
- Inject the interface where we want to use it
- In the Rest Controller we delegate to the proxy (`CityClient`)
 - The proxy encapsulate the remote call via Ribbon to make the rest call itself.

```
@FeignClient(name = "capitolServ")
interface CityClient {
    @GetMapping(value = "/capitol/{code}")
    public String get(@PathVariable("code") String id) throws JSONException;
}
```

```
@Named
private CityClient client;
```

```
@RequestMapping("/{code}")
public String hello(@PathVariable("code") String id) throws
JSONException {
    return client.get(id);
}
```


Feign – Complete Feign Client

```
@EnableFeignClients
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class Client {
    @Named
    private CityClient client;

    public static void main(String[] args) {
        SpringApplication.run(Client.class, args);
    }

    @GetMapping("/{code}")
    public String hello(@PathVariable("code") String id) throws JSONException {
        return client.get(id);
    }

    @FeignClient(name = "capitolServ")
    interface CityClient {
        @GetMapping(value = "/capitol/{code}")
        public String get(@PathVariable("code") String id) throws JSONException;
    }
}
```



The diagram illustrates the Feign client setup. A box labeled "Feign Proxy" points to the `@Named` annotation on the `client` field. Another box labeled "Call Service" points to the `client.get(id)` call in the `hello` method.

Feign – Complete Feign Client

Spring Properties

```
spring:
  application:
    name: cityFeign
# Discovery Server Access
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
    instance:
      metadataMap:
        instanceId: ${spring.application.name}:${spring.application.instance_id:${random.value}}
```

HTTP Server

```
server:
  port: 9999 # HTTP (Tomcat) port
capitolServ:
ribbon:
  DeploymentContextBasedVipAddresses: capitol-service
  NIWSServerListClassName: com.netflix.niws.loadbalancer.DiscoveryEnabledNIWSServerList
```

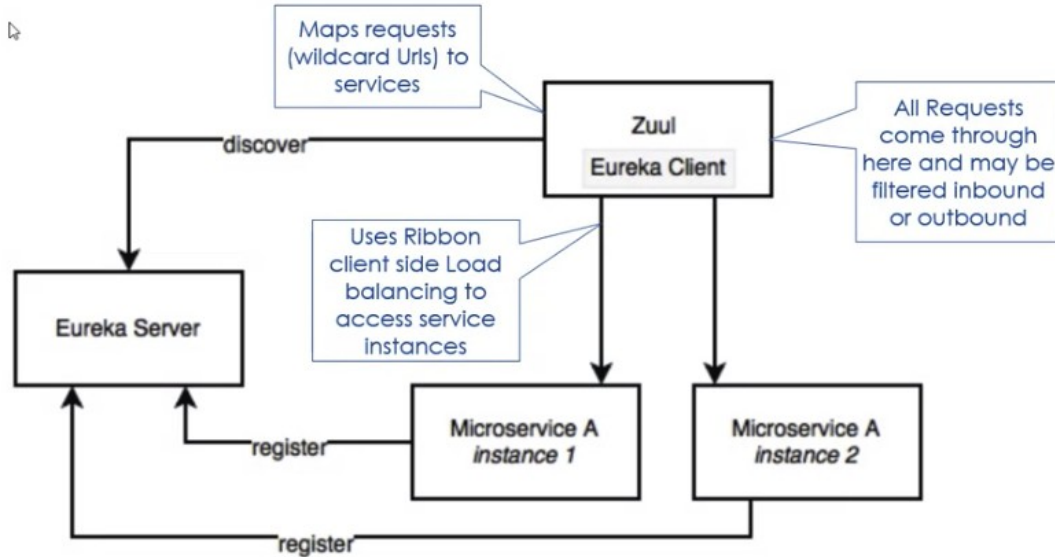


Label to
identify
service

Zuul

- Zuul is the front door for all request from various devices and websites to the backend of a Netflix Streaming operation
 - All Request will go through that service
- A series of fitters that are capable of performing a range of actions during the routine of HTTP Requests and responses to desired services

Zuul as a Gateway or Edge Service



Zuul

- There are several standard filter type that correspond to the typical lifecycle of a request
 - Pre Filters execute before the routing of the origin
 - Routing Filters handle routing the request to a destination
 - HTTP request is build and sent using Apache HttpClient or Netflix Ribbon
 - Changing the service being routed to
 - Post Filter execute after the request has been routed to the service on the return route
 - Error Filter execute when an error happens during one of the phases
- We have defined a prefilter to add login action during the routing of http request and response
 - Zuul filters store request and state information in the RequestContext
 - Using that to get to the HttpServletRequest

Zuul

```
@Named
public class SimpleFilter extends ZuulFilter {
    private static Logger log = LoggerFactory.getLogger(SimpleFilter.class);
    public String filterType() {
        return "pre";
    }
    public int filterOrder() {
        return 1;
    }
    public boolean shouldFilter() {
        return true;
    }
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        log.info(String.format("%s request to %s", request.getMethod(), request.getRequestURL().toString()));
        return null;
    }
}
```

filterType() returns a String that stands for the type of the filter---
i.e., pre

filterOrder() gives the order in which this filter will be executed,
relative to other filters

shouldFilter() returns Boolean that contains the logic that
determines if to execute this filter

run() contains the functionality of the filter

Zuul

- Spring Cloud Netflix uses an embeded Zuul proxy
 - @EnableZuulProxy – Turns the application into a proxy that forwards relevant call to other services.
 - It will use a set of routes defined in YAML to match incoming request to a destination service
 - Encapsulated in the Proxy that uses a load balanced RestTemplate to make the HTTP Request to the desired service
 - For a Eureka Registered Service we are matching URLs to service names to actual service ids
 - We have defined where to find the service with familiar Eureka Ribbon Configuration for the Zuul Service to crate a RibbonServerList

```
@EnableZuulProxy
@SpringBootApplication
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

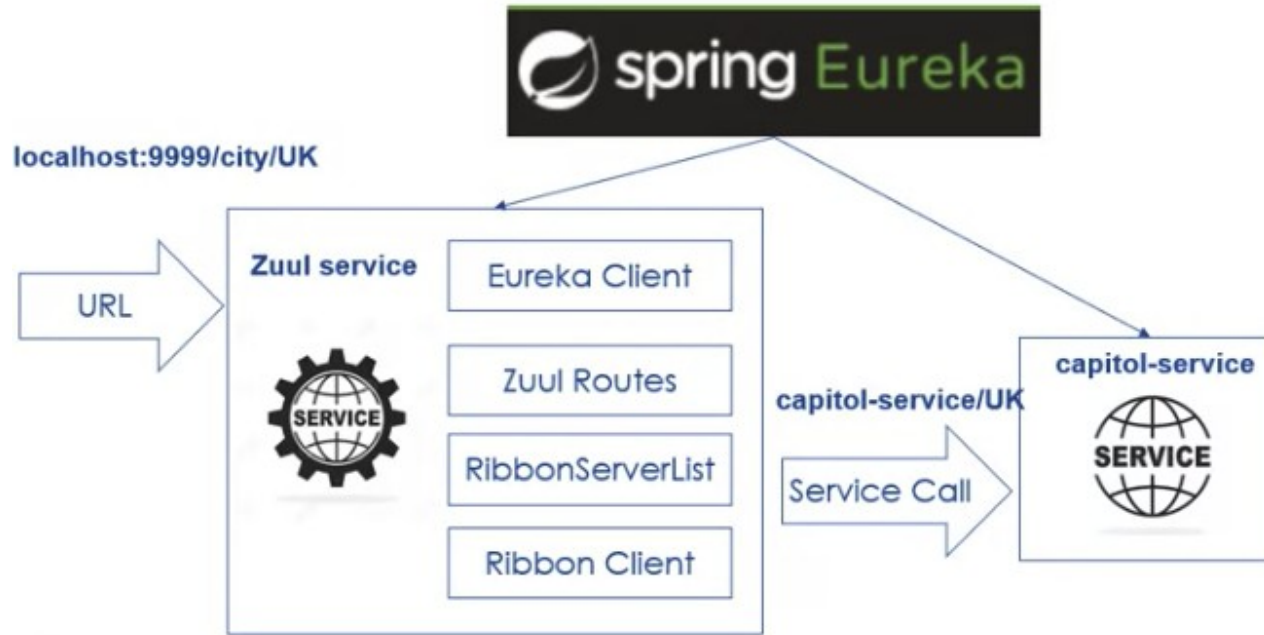
```
zuul:
  routes:
    echo:
      path: /city/**
      serviceId: capitolServ
  capitolServ:
    ribbon:
      DeploymentContextBasedVipAddresses: capitol-service
      NIWSServerListClassName:
com.netflix.niws.loadbalancer.DiscoveryEnabledNIWSServerList
```

Url- mapped to service label

service label mapping to actual service id

Zuul

Making a request to our Edge service it map /cityUK to capital-service OK



Zuul

```
spring:
  application:
    name: gateway
server:
  port: 8080
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
  instance:
  metadataMap:
    instanceId: ${spring.application.name}:${spring.application.instance_id:
${random.value}}
zuul:
  routes:
    test :
      path: /**
      serviceId: anotherservice

anotherservice:
  ribbon:
    DeploymentContextBasedVipAddresses: myservice
    NIWSServerListClassName: com.netflix.niws.loadbalancer.DiscoveryEnabledNIWSServerList
```

Deconstructing the Monolith – Introduction