Microservices

UA.DETI.IES - 2019/20

Resources & Credits

- Microservices Best Practices for Java, M. Hofmann, E. Schnabel, K. Stanley, IBM
  - *chapters 2, 3, 4 & 8*

- Spring Microservices in Action, John Carnell, Manning Publications

Overview

- Application architecture patterns are changing in the era of **cloud computing**.
- A convergence of factors has led to the concept of "cloud native" applications:
  - The general availability of cloud computing platforms
  - Advancements in virtualization technologies
  - The emergence of agile and DevOps practices as organizations looked to streamline and shorten their release cycles
- **Microservices** are cloud native applications composed of smaller, independent, self-contained pieces.

Cloud native applications

- Cloud computing environments are dynamic, with on-demand allocation and release of resources from a virtualized, shared pool.
  - This elastic environment enables more flexible scaling options, especially compared to the up-front resource allocation typically used by traditional on premises data centers.
- Cloud native systems properties:
  - Applications or services (microservices) are loosely coupled with explicitly described dependencies.
  - Applications or processes are run in software containers as isolated units.
  - Processes are managed by using central orchestration processes to improve resource utilization and reduce maintenance costs.

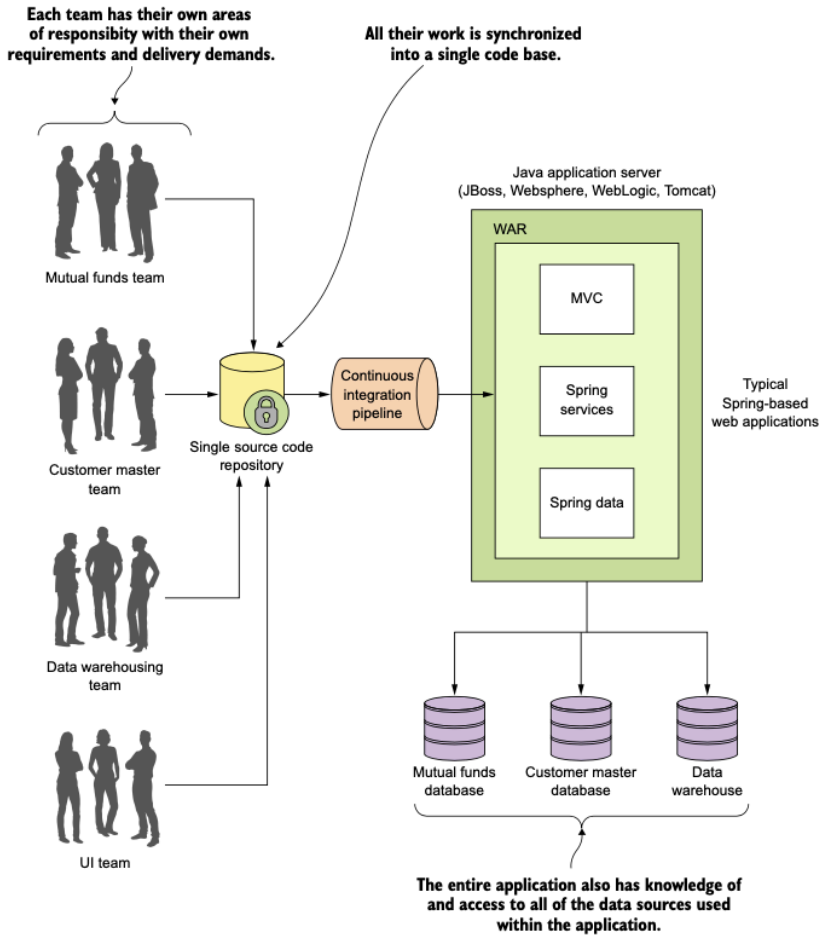
Microservices Twelve Factors

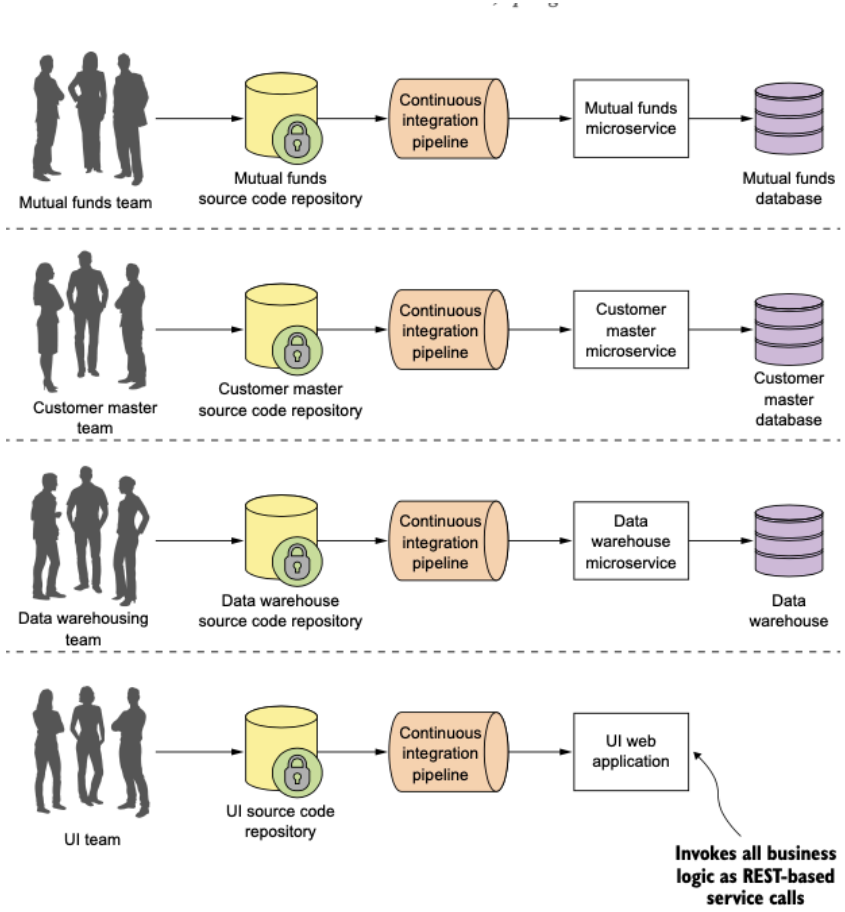
- The 12-factor application methodology was drafted by developers at Heroku.
- The characteristics mentioned in the 12 factors are not specific to cloud provider, platform, or language.
- The factors represent a set of guidelines or best practices for portable, resilient applications that will thrive in cloud environments (specifically software as a service applications).

Microservices Twelve Factors

1. There should be a one-to-one association between a versioned **codebase** (for example, an IT repository) and a deployed service. The same codebase is used for many deployments.
2. Services should explicitly declare all **dependencies**, and should not rely on the presence of system-level tools or libraries.
3. **Configuration** that varies between deployment environments should be stored in the environment (specifically in environment variables).
4. All **backing services** are treated as attached resources, which are managed (attached and detached) by the execution environment.
5. The delivery pipeline should have strictly separate stages: **Build, release, and run**.
6. Applications should be deployed as one or more stateless **processes**. Persisted data should be stored in an appropriate backing service.
7. Self-contained services should make themselves available to other services by listening on a specified **port**.
8. **Concurrency** is achieved by scaling individual processes (horizontal scaling).
9. Processes must be **disposable**: Fast startup and graceful shutdown behaviors lead to a more robust and resilient system.
10. **Dev/Prod parity**: All environments, from local development to production, should be as similar as possible.
11. Applications should produce **logs** as event streams, and trust the execution environment to aggregate streams.
12. If **admin** tasks are needed, they should be kept in source control and packaged alongside the application to ensure that it is run with the same environment as the application.

Monolithic vs. Microservices

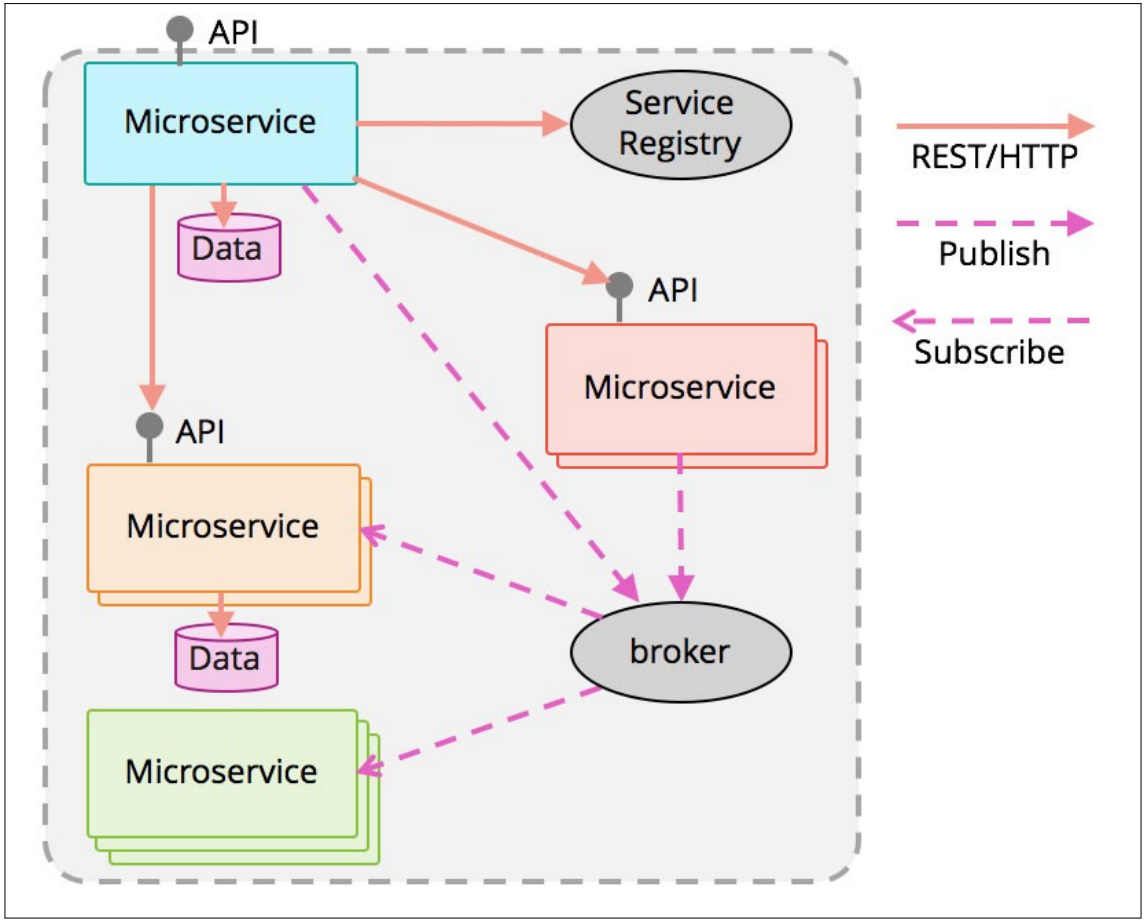




Microservices

- Although there is no standard definition for microservices, most reference Martin Fowler's seminal paper.
  - "Microservices: A new architectural term"
- The paper explains that microservices are used to compose complex applications by using:
  - small,
  - independent (autonomous),
  - replaceable processes that
  - communicate by using lightweight APIs that do not depend on language.
- Each microservice is a 12-factor application, with replaceable backing services providing a message broker, service registry, and independent data stores.

Microservices



The meaning of “small”

- Many descriptions make parallels between the roles of individual microservices and chained commands on the UNIX command line:

```
$ ls | grep ‘service’ | sort -r
```

  - These UNIX commands each do different things.
  - You can use the commands without awareness of what language the commands are written in, or how large their codebases are.
- The use of the word “small”, as applied to a microservice, essentially means that it is focused in purpose.
  - The microservice should do one thing, and do that one thing well.

Independence and autonomy

- Each service must be capable of being started, stopped, or replaced at any time without tight coupling to other services.
  - Many other characteristics of microservice architectures follow from this single factor.
- If deploying changes requires simultaneous or time sensitive updates to multiple services, you are doing it wrong.
- There are a few possible outcomes:
  - You need to revise your versioning strategy to preserve the independent lifecycle of services.
  - The current service boundaries are not drawn properly, and services should be refactored into properly independent pieces.

Polyglot

- Polyglot is a frequently cited benefit of microservice-based architectures.
  - Being able to choose the appropriate language or data store to meet the needs of each service can be powerful, and can bring significant efficiency.
- Polyglot applications are only possible with language-agnostic protocols.
  - Representational State Transfer (REST) architecture patterns define guidelines for creating uniform interfaces that separate the on-the-wire data representation from the implementation of the service.
  - RESTful architectures require the request state to be maintained by the client, allowing the server to be stateless.

Creating Microservices in Java

- How you identify and create the microservices?
  - with specific emphasis on how identified candidates are converted into RESTful APIs,
  - and then implemented in Java?

1. Java platforms
2. Versioned dependencies
3. Identifying services
4. Creating REST APIs

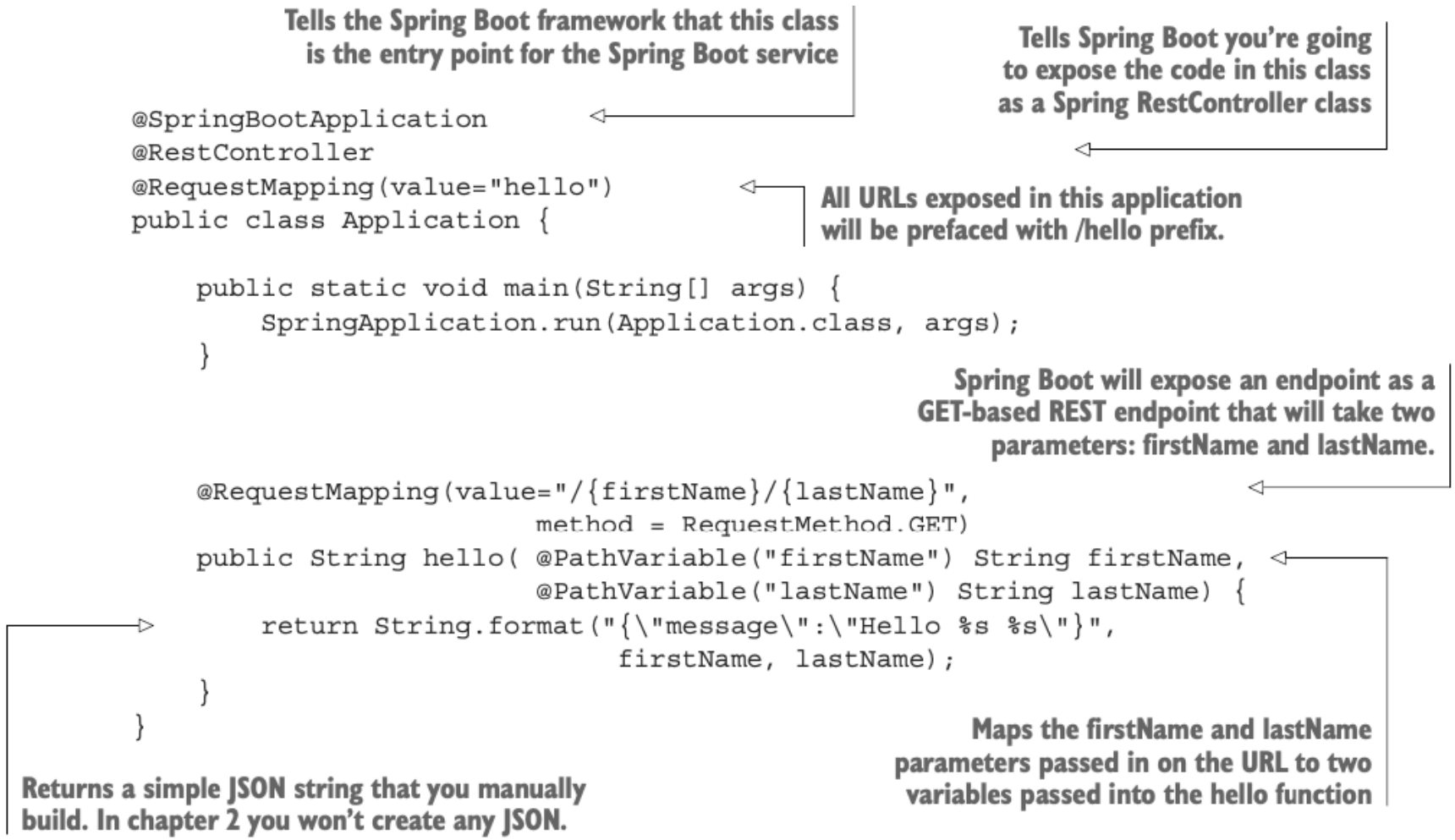
Java platforms

- Embrace new language features like lambdas, parallel operations, and streams.
- Using annotations can simplify the codebase by eliminating boilerplate
  - but can reach a subjective point where there is "too much magic".
- You can create perfectly functional microservices using nothing but low-level Java libraries.
  - However, it is generally recommended to have something provide a reasonable base level of support for common, cross-cutting concerns. It allows the codebase for a service to focus on satisfying functional requirements.

Spring Boot and Spring Cloud

- Spring Boot emphasizes convention over configuration
  - Uses a combination of annotations and classpath discovery to enable additional functions.
- Spring Cloud is a collection of integrations between third-party cloud technologies and the Spring programming model.
  - Some are integrations between the spring programming model and third-party technologies.
  - In fewer cases, Spring Cloud technologies are usable outside of the Spring programming model, like Spring Cloud Connectors Core.

Spring Boot

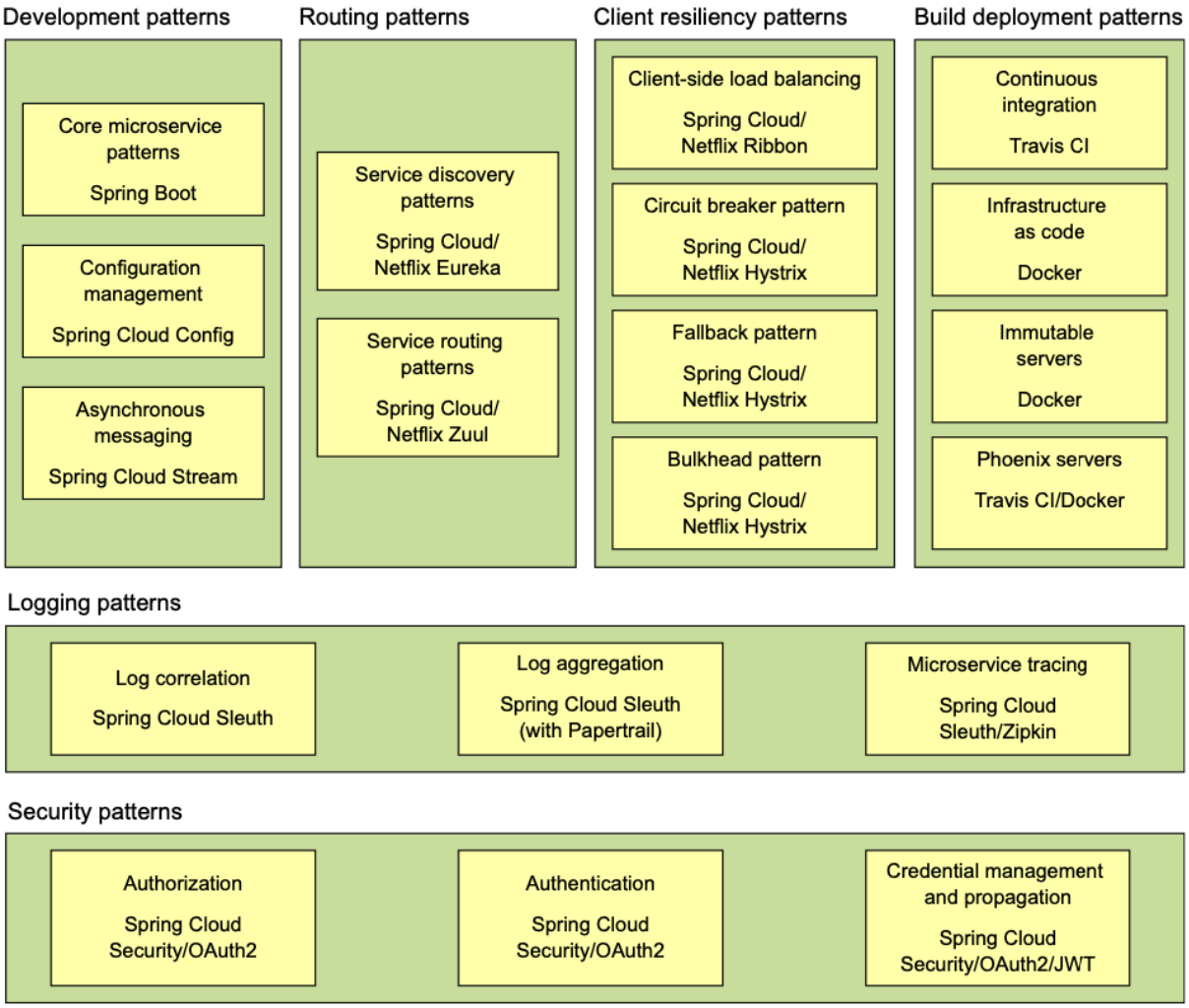



```
@SpringBootApplication
@RestController
@RequestMapping(value="hello")
public class Application {

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

    @RequestMapping(value="/{firstName}/{lastName}",
                    method = RequestMethod.GET)
    public String hello( @PathVariable("firstName") String firstName,
                         @PathVariable("lastName") String lastName) {
        return String.format("{\"message\":\"Hello %s %s\"}",
                             firstName, lastName);
    }
}
```

Spring Cloud

- We may develop microservices using Spring Boot.
  - These were all stand alone applications. But suppose we now have to connect the various applications and build a distributed system.
- Spring Cloud deals with:
  - Complexity associated with distributed systems
    - Network issues, Latency overhead, Bandwidth issues, security issues.
  - Service Discovery
    - Service discovery tools manage how processes and services in a cluster can find and talk to one another.
  - Redundancy
    - Redundancy issues in distributed systems.
  - Load balancing
  - Performance issues
  - Deployment complexities
    - Requirement of DevOps skills

Spring Cloud



Spring Cloud:

Example

```
package com.thoughtmechanix.simpleservice;

//Removed other imports for conciseness
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import  org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

@SpringBootApplication
@RestController
@RequestMapping(value="hello")
@EnableCircuitBreaker
@EnableEurekaClient
public class Application {

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

    @HystrixCommand(threadPoolKey = "helloThreadPool")
public String helloRemoteServiceCall(String firstName,
                                     String lastName){
    ResponseEntity<String> restExchange =
       restTemplate.exchange(
          "http://logical-service-id/name/
          [ca]{firstName}/{lastName}",
           HttpMethod.GET,
           null, String.class, firstName, lastName);

       return restExchange.getBody();


    }

@RequestMapping(value="/{firstName}/{lastName}",
     method = RequestMethod.GET)
    public String hello( @PathVariable("firstName") String firstName,
                         @PathVariable("lastName") String lastName) {
       return helloRemoteServiceCall(firstName, lastName)
}
}
```

Enables the service to use the Hystrix and Ribbon libraries

Tells the service that it should register itself with a Eureka service discovery agent and that service calls are to use service discovery to "lookup" the location of remote services

Wrappers calls to the helloRemoteServiceCall method with a Hystrix circuit breaker

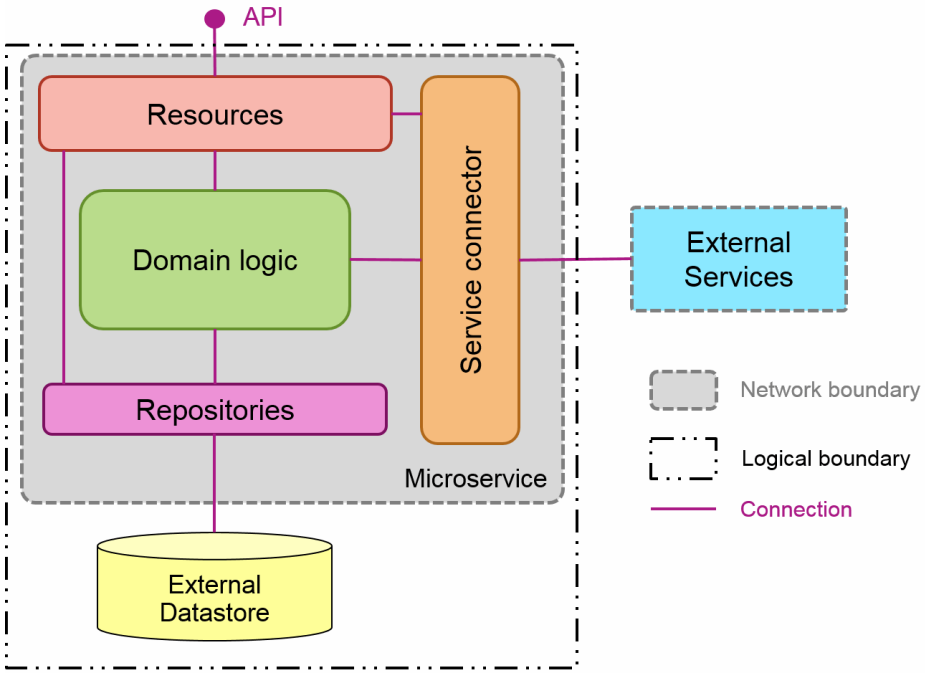
Uses a decorated RestTemplate class to take a "logical" service ID and Eureka under the covers to look up the physical location of the service

Versioned dependencies

- Build tools like Apache Maven or Gradle provide easy mechanisms for defining and managing dependency versions.
  - Explicitly declaring the version ensures that the artifact runs the same in production as it did in the testing environment.
- Several tools are available to make creating Java applications easier, e.g.:
  - WebSphere Liberty App Accelerator http://wasdev.net/accelerate
  - Spring Initializr http://start.spring.io/
  - Wildfly Swarm Project Generator http://wildfly-swarm.io/generator/

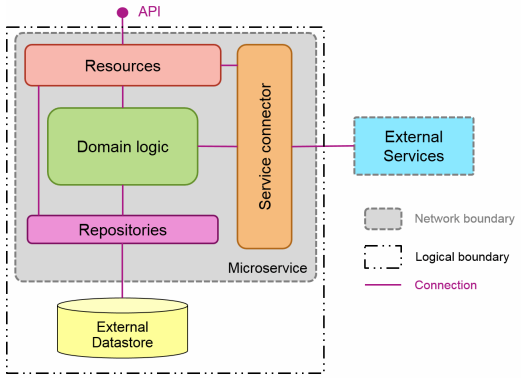
Microservice internal structure

- Structure code inside a service to have a clear separation of concerns to facilitate testing.
- Calls to external services should be separate from the domain logic, which should also be separate from logic specific to marshalling data to/from a backing data service.



Microservice internal structure

- This architecture should follow a few rules when creating classes, each performing one of these tasks:
  - Perform domain logic
  - Expose resources
  - Make external calls to other services
  - Make external calls to a data store



- These are general recommendations for code structure that do not have to be followed strictly.
  - The important characteristic is to reduce the risk of making changes.

Documenting APIs

- The Open API Initiative (OAI) is a consortium focused on standardizing RESTful APIs descriptions.
  - The OpenAPI specification is based on Swagger, which defines the structure and format of metadata to create a representation of a RESTful API.
  - https://github.com/OAI/OpenAPI-Specification
- This definition is usually expressed in a single, portable file - name.json or name.yaml.
- The swagger definition can further be used to generate client or server stubs.

Example

https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/petstore.yaml

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: A paged array of pets
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pets"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
    post: ...
```

REST APIs: Use the correct HTTP verb

- **POST** operations may be used to Create(C) resources.
  - The distinguishing characteristic of a POST operation is that it is not idempotent.
    - For example, if a POST request is used to create resources, and it is started multiple times, a new, unique resource should be created as a result of each invocation.
- **GET** operations must be both idempotent and nullipotent.
  - They should cause no side effects and should only be used to Retrieve(R) information.
- **PUT** operations can be used to Update(U) resources.
  - PUT operations usually include a complete copy of the resource to be updated, making the operation idempotent.
- **PATCH** operations allow partial Update(U) of resources.
  - They might or might not be idempotent depending on how the delta is specified and then applied to the resource.
- **DELETE** operations are used to Delete(D) resources.
  - Delete operations are idempotent, as a resource can only be deleted once.
    - However, the return code varies, as the first operation succeeds (200), while subsequent invocations do not find the resource (204).

Machine-friendly, descriptive results

- The **HTTP status code** should be relevant and useful.
  - Use a 200 (OK) when everything is fine. When there is no response data, use a 204 (NO CONTENT) instead.
  - Beyond that technique, a 201 (CREATED) should be used for POST requests that result in the creation of a resource, whether there is a response body or not.
  - Use a 409 (CONFLICT) when concurrent changes conflict, or a 400 (BAD REQUEST) when parameters are malformed.
    - For more information, see the following website: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
- Consider also what data is being returned in the responses to make communication efficient.
  - The created/modified resource is usually included in the response, because it eliminates the need for the caller to make an extra GET request to fetch the created resource.

Resource URIs and versioning

- In general, resources should be nouns, not verbs, and endpoints should be plural.
- This technique results in a clear structure for create, retrieve, update, and delete operations:
  - POST /accounts — Create a new item
  - GET /accounts — Retrieve a list of items
  - GET /accounts/16 — Retrieve a specific item
  - PUT /accounts/16 — Update a specific item
  - PATCH /accounts/16 — Update a specific item
  - DELETE /accounts/16 — Delete a specific item
- Relationships are modeled by nesting URIs, for example, something like
  - /accounts/16/credentials
    for managing credentials associated with an account.

Versioning

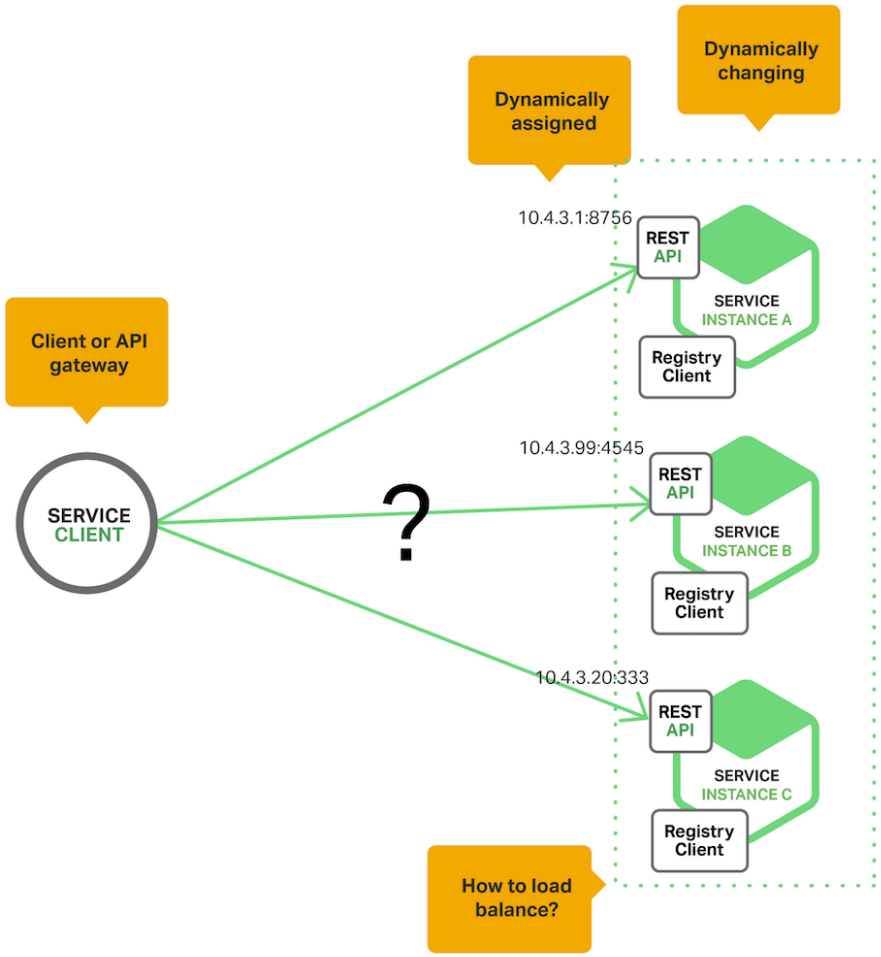
- One of the major benefits of microservices is the ability to allow services to evolve independently.
  - Given that microservices call other services, that independence comes with a giant caveat. You cannot cause breaking changes in your API.
- If you do need to make breaking API changes for an existing service, there are generally three ways to handled versioning a REST resource:
  - Put the version in the URI
  - Use a custom request header
  - Put the version in the HTTP Accept header and rely on content negotiation

Put the version in the URI

- The version should apply to the application, as a whole, for example:
  - /api/v1/accounts instead of /api/accounts/v1.
- Advantages
  - It is easy to understand.
  - It is easy to achieve when building the services.
  - It is compatible with API browsing tools like Swagger and command-line tools like curl.
- Disadvantages
  - In a strict interpretation of the HTML standard, a URL should represent the entity and if the entity being represented didn't change, the URL should not change.
  - Another concern is that putting versions in the URI requires consumers to update their URI references.

Location services

- Microservices are designed to be easily scaled.
  - This scaling is done horizontally by scaling individual services.
- With multiple instances of microservices, we need a method for locating services and load balancing over the different instances of the service you are calling.



Service registry

- There are four reasons why a microservice might communicate with a service registry:
- **Registration**
  - After a service has been successfully deployed, the microservice must be registered with the service registry.
- **Heartbeats**
  - The microservice should send regular heartbeats to the registry to show that it is ready to receive requests.
- **Service discovery**
  - To communicate with another service, a microservice must call the service registry to get a list of available instances
- **De-registration**
  - When a service is down, it must be removed from the list of available services in the service registry.

Third-party registration # self-registration

- The registration of the microservice with the service registry can be done by the microservice or by a third party.
- Using self-registration pulls the registration and heartbeat logic into the microservice itself.
- Using a third party requires said party to inspect the microservice to determine the current state and relay that information to the service registry.
  - The advantage of using a third party is that the registration and heartbeat logic is kept separate from the business logic.
  - The disadvantage is that there is an extra piece of software to deploy and the microservice must be exposed to a health endpoint for the third party to poll.
- Many service registry solutions provide a convenience library form registration, reducing the complexity of the code required. Some service registry solutions are available:
  - Consul, https://www.consul.io/
  - Eureka, https://github.com/Netflix/eureka
  - ZooKeeper, https://zookeeper.apache.org

Microservice communication

- **Synchronous** communication is a request that requires a response, whether that be immediately or after some amount of time.
- **Asynchronous** communication is a message where no response is required.
  - A strong case can be made for using asynchronous events or messaging in highly distributed systems that are built from independent parts.
- For either invocation style, services should be documented to ensure that the system is comprehensible and enable future consumers.
- Event subscribers and API consumers should tolerate unrecognized fields, as they might be new.
- Services should also expect and handle bad data. Assume that everything will fail at some point.

Synchronous messaging (REST)

- Synchronous APIs should be used where clear request/response semantics apply,
  - or where one service needs to trigger a specific behavior in another service.
- These are usually RESTful operations that pass JSON formatted data, but other protocols and data formats are possible.
  - In a Java based microservice, having the applications pass JSON data is the best choice.
  - There are libraries to parse JSON to Java objects, and JSON is becoming widely adopted, making it a good choice to make your microservice easily consumable.

Asynchronous messaging (Events)

- Asynchronous messaging is used for decoupled coordination.
  - An asynchronous event can only be used if the creator of the event does not need a response.
- Use a reactive style for events.
  - A service should only publish events about its own state or activity.
  - Other services can then subscribe to those events and react accordingly.
- To coordinate the messaging we can use any one of the available message brokers.
  - Some examples that integrate with Java are the AMQP broker with the RabbitMQ Java client, Apache Kafka, and MQTT.

Internal messaging

- Events can be used within a single microservice.
- Example: a user placing an order on an online store.
  - After the user entered the order details, s/he clicks the Submit button.
  - The user expects a confirmation that the order has gone through, so the button triggers a synchronous request that returns a response.
  - Internally, the application starts processing the order.
  - The service that received the order request sends out asynchronous events that various other services are subscribed to (e.g. inventory updates, shipping events, and payment events).
  - Some services trigger other synchronous or asynchronous requests in response to the events.
    - For example, the inventory service uses a synchronous request to update the inventory database.

Fault tolerance

- **Timeouts**
  - When making a request to another microservice, whether asynchronously or not, the request must have a timeout.
- **Circuit breakers**
  - A circuit breaker is designed to avoid repeating timeouts. If the count of failures reaches a certain level, it prevents further calls from occurring.
- **Bulkheads**
  - We need to make sure that a failure in one part of your application doesn't take down the whole thing.
  - The simplest implementation of the bulkhead pattern is to provide fallbacks. They allow the application to continue functioning when non-vital services are down.
    - For example, in an online retail store there might be a service that provides recommendations for the user. If the recommendation service goes down, the user should still be able to search for items and place orders.
- **Consuming APIs**
- **Producing APIs**

Consuming APIs

- As the consumer of an API, we need to validate the response to check that it contains the information.
  - If we are receiving JSON, we also need to parse the JSON data before performing any Java transformations.
- When doing validation or JSON parsing, we must:
  - Only validate the request against the variables or attributes that we need
    - Do not validate against variables just because they are provided.
  - Accept unknown attributes
    - Do not issue an exception if we receive an unexpected variable.
- In this way, microservices are more resilient against changes.

Producing APIs

- When providing an API to external clients, do these two things when accepting requests and returning responses:
  - Accept unknown attributes as part of the request
    - If a service has called your API with unnecessary attributes, discard those values. Returning an error in this scenario just causes unnecessary failures.
  - Only return attributes that are relevant for the API being invoked
    - Leave as much room as possible for the implementation of a service to change over time.
- By following these two rules we may also changing the API in the future and adapting to changing requirements from consumers - Robustness Principle.

Deployment pipelines and tools

- Deployment artifacts often run on different systems
  - many systems must be configured to make the services work.
- Human intervention should be reduced to a minimum.
  - The code should be in a Source Code Management (SCM) tool like Subversion or GIT.
  - It should be built with a tool that includes dependency management like Maven or Gradle.
  - Choose one of the repository management tools (such as Nexus or Artifactory) to store versioned binary files of your shared libraries.
  - Build your services using one of the CI-Tools like Jenkins or Bamboo.
- After the automated build of the microservice testing must be done.
  - Scripting frameworks/infrastructures like Chef, Puppet, Ansible, and Salt can handle this situation.

Packaging options

- After building the microservice we need to decide on the packaging format to deploy the application.
  - A recurring goal is the creation of immutable artifacts that can be run without change across different deployment environments.
- The following packaging options are available, among others:
  - JAR/WAR file deployment on preinstalled middleware
  - Executable JAR file
  - Containers
  - Every microservice on its own virtualized server

JAR/WAR deployment

- This variant has been the default way for deploying Java EE applications.
  - the application gets hosted in different application servers in different staging environments (such as quality assurance, performance test, and user acceptance test).
- Problems:
  - Frequent deployments in a shorter time period (day, hour) can be hard to achieve, since some application deployments need a restart of the application server.
  - If different applications are hosted together on the same application server, the load from one application can interfere with the performance of the other application.
  - High variations in the load of an application cannot easily be adopted in the infrastructure.
- If this pattern is used with microservices, the relationship should be one-to-one between the service and the preinstalled middleware server.
  - This configuration is common, for example, in PaaS environments, where you push only the WAR to a freshly built, predefined server.

Executable JAR file

- One way to avoid depending on the deployment environment to provide dependencies is to pack everything that is needed to run the service inside an executable JAR file.
  - These fat JAR files eliminates problems introduced by dependency mismatches between development and test or production deployment environments.
  - These self-contained JAR files are generally easy to handle, and help create a consistent, reproducible environment for local development and test in addition to target deployment environments.

Containerization

- The previous strategies are all running on an operating system that must be installed and configured to work with the executable JAR file.
- A strategy to avoid the problems resulting from inconsistently configured operating systems is to bring more of this operating system with you.
  - This strategy leads to the situation that a microservice runs on an operating system with the same configuration on every stage.
- Containers are lightweight virtualization artifacts that are hosted on an existing operating system.
  - Docker provides an infrastructure capability to build a complete file system to run the application.

Docker containers

- Docker images are immutable artifacts.
  - When applications are built into Docker images, those images can then be used to start containers in different deployment environments, ensuring that the same stack is used across all environments.
- Docker containers makes the exact packaging format irrelevant because the image configuration determines how a container is started.
  - Configuration values for the microservice hosted inside a container can be provided either by using environment variables or mounted volumes.
- Data stores and all other backends needed by the microservice should not be hosted in the same container as the microservice itself.
  - These backends should be treated as services, which is an essential characteristic of a microservice.
  - Data stores, for example, can also be managed as Docker containers.

Every microservice on its own server

- Another alternative for packaging is to use the same virtualized server as the basis for every stage.
  - Create a server template with the operating system and other tools that are needed.
  - Use this template as the basis for all servers that host microservices, which is another form of immutable server.
- To manage these servers and server templates, we can use tools such as Vagrant, VMware, and VirtualBox.
- This configuration leads mainly to two variants:
  - One microservice packaged on a single virtualized server.
  - Multiple microservice packages on a single server

Preferred practices on packaging

- Virtualization or cloud services are the preferred basis for infrastructure because dynamic provisioning can help a system of microservices deal with changing capacity requirements.
- System level configuration should either be as similar as possible across all stages, or should be isolated from the application by using technology like containers.
- An immutable artifact should be produced that is promoted unchanged between stages.
- Immutable artifacts for microservices should be stand alone and self-contained. This item could be an executable JAR file or a Docker container.
- Each immutable artifact should contain only one microservice application and its supporting run time and libraries.
- Shared libraries should be built as part of your application, rather than being shared or used as a dependency provided by the hosting environment.

Summary

- Microservices…
  - Creation
  - Location
  - Communication
  - Deployment

- Further reading
  - https://spring.io/projects/spring-cloud
  - https://www.baeldung.com/spring-cloud-configuration
  - https://www.baeldung.com/spring-cloud-netflix-eureka