# Additional REST Service Techniques

1. Consuming REST services
2. Microservice architecture
3. Circuit breakers

# 1. Consuming REST Services

- Overview
- Key methods in `RestTemplate`
- Example
- Key classes in the REST client application
- Aside: Consuming a REST service via `WebClient`
- Aside: Consuming a REST service from HTML

# Overview

- Spring enables you to implement client code to consume REST services
  - Via the `RestTemplate` class

- Include the following POM dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

pom.xml

# Key Methods in `RestTemplate`

- Here are some of the key methods in `RestTemplate`:

```
ResponseEntity<T> getForEntity(String, Class<T>, Object...)
```

```
ResponseEntity<T> postForEntity(String, Object, Class<T>, Object...)
```

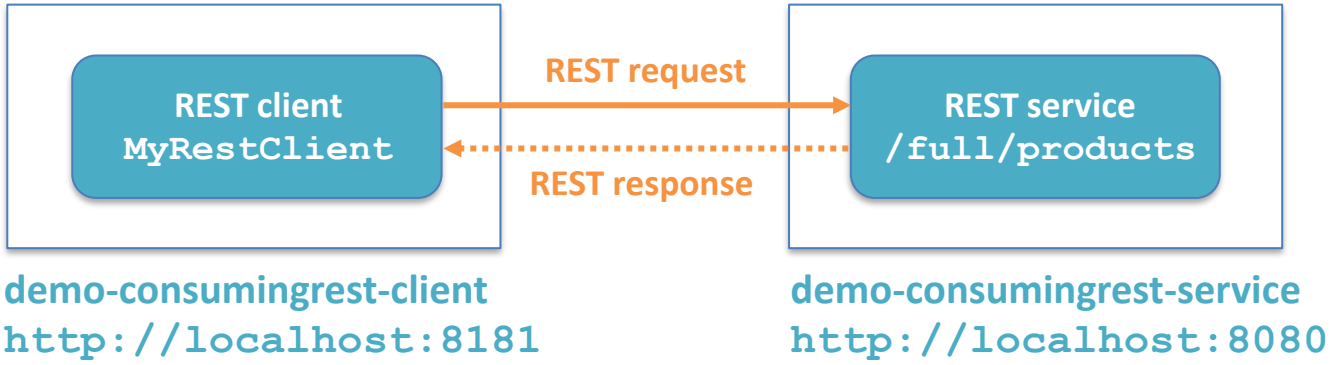
```
void put(String, Object, Object...)
```

```
void delete(String, Object...)
```

```
ResponseEntity<T> exchange(String, HttpMethod, object, Class<T>)
```

# Example

- Let's see an example of how to consume REST endpoints:



# Key Classes in the REST Client Application

- `MyRestClient`
  - Calls REST service endpoints, by using `RestTemplate`
- `Product`
  - `Product` objects passed to/from REST service
  - Serialized/deserialized by `RestTemplate`

# Aside: Consuming a REST Service via `WebClient`

- We've seen how to use `RestTemplate`
  - This is synchronous (the client blocks until the response is in)
- An alternative approach is to use `WebClient`
  - Can be synchronous or asynchronous
  - Well suited to calling reactive REST services (WebFlux)
  - Requires this dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

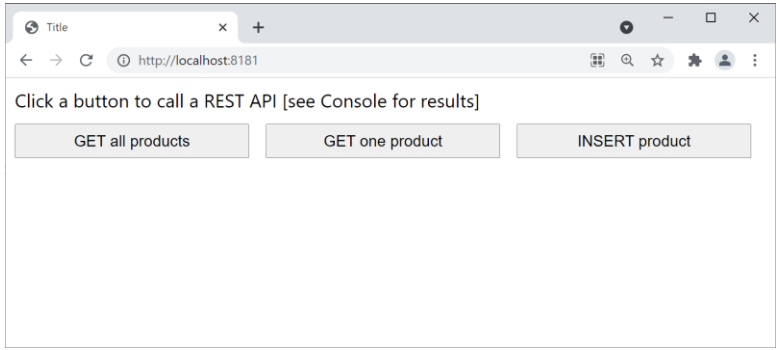
pom.xml

- See `MyRestClientViaWebClient.java`

# Aside: Consuming a REST service from HTML

- We've also implemented a simple HTML page to show how to consume a REST service from a web UI
  - project: `demo-consumingrest-client`
  - Folder: `src/main/resources/static`

- Open a browser and browse to `http://localhost:8181`



# 2. Microservice Architecture

- What are microservices
- Microservice application example
- Implementing the catalog service
- Implementing the client service

# What are Microservices?

- According to Wiki:

**Microservices** is a specialisation of an implementation approach for service-oriented architectures (SOA) used to build flexible, independently deployable software systems.

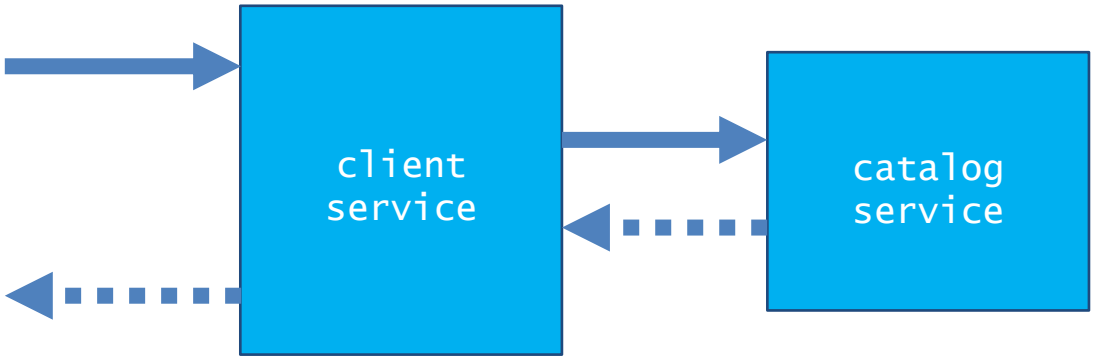
Services in a **microservice architecture (MSA)** are processes that communicate with each other over a network in order to fulfil a goal. These services use technology-agnostic protocols.

The microservices approach is a first realisation of SOA that followed the introduction of DevOps and is becoming more popular for building continuously deployed systems.

# Microservice Application Example

- Let's see a complete (simple) example of how to create a microservice application

- There are two Spring Boot applications in the demo:
  - `demo-msa-clientservice`
  - `demo-msa-catalogservice`



# Implementing the Catalog Service (1 of 2)

- The "catalog" service is a Spring Boot application with a REST service that returns catalog info
  - See `demo-msa-catalogservice`
  - The `server.port` property is `8081`

- Take a look at the endpoints in `CatalogController`:
  - `/catalog`
  - `/catalog/{index}`

# Implementing the Catalog Service (2 of 2)

- Run the catalog app and ping the following URLs…

**http://localhost:8081/catalog**



```
[
    "Bugatti Divo",
    "Lear Jet",
    "Socks from M&S"
]
```

**http://localhost:8081/catalog/0**

```
Bugatti Divo
```

# Implementing the Client Service (1 of 3)

- The "client" service is another Spring Boot application with a REST service
  - See `demo-msa-clientservice`
  - The `server.port` property is `8080`

- Take a look at the endpoint in `ClientController`:
  - `/client/{index}`

# Implementing the Client Service (2 of 3)

- The "client" service invokes the "catalog" service
  - Using a Spring `RestTemplate`

HTTP request

```
@RestController
public class ClientController {

   @GetMapping("/client/{index}")
   public String getItem(@PathVariable int index){

      URI catalogUrl = URI.create("http://localhost:8081/catalog/" + index);
      RestTemplate restTemplate = new RestTemplate();

      String result = restTemplate.getForObject(catalogUrl, String.class);
      return String.format("[%s] Item %d %s", LocalTime.now(), index, result);
   }
}
```

Catalog service

# Implementing the Client Service (3 of 3)

- Run the client app and ping the following URL…
  - `http://localhost:8080/client/0`

# 3. Circuit Breakers

- Overview
- Circuit breakers in Spring Cloud
- Spring Cloud circuit breaker dependency
- Spring Cloud circuit breaker example
- Seeing a circuit breaker in action

# Overview

- In a microservice application, services call other services
  - E.g. ServiceA calls ServiceB, ServiceB calls ServiceC, etc.

- If any service is down, you get a ripple effect of failures
  - E.g. if ServiceC is down…
  - Then ServiceB will fail (because it depends on ServiceC)
  - Then ServiceA will fail (because it depends on ServiceB), etc.

- To avoid the ripple effect of failures, use a **circuit breaker**
  - Specify a fallback method that can be called, if a service fails

# Circuit Breakers in Spring Cloud

- Spring Cloud provides a circuit breaker API
  - Via the `CircuitBreakerFactory` class

- `CircuitBreakerFactory` is an abstraction over various circuit breaker implementations, including:
  - Resilience4J (we'll use this)
  - Netflix Hystrix
  - Sentinel
  - Spring Retry

# Spring Cloud Circuit Breaker Dependency

- To use the Resilience4J circuit breaker implementation, add the following dependency to the pom file in your (client) project :

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
    <version>3.0.0</version>
</dependency>
```

**pom.xml (client project)**

- Once you've added this dependency, Spring Boot autoconfig will automatically create a Resilience4J bean
  - This bean is exposed via `CircuitBreakerFactory`
  - See next slide for an example of how to use a circuit breaker…

# Spring Cloud Circuit Breaker Example

HTTP request

```
@RestController
public class ClientWithFallbackController {

   @Autowired
   private CircuitBreakerFactory factory;

   @GetMapping("/clientWithFallback/{index}")
   public String getItem(@PathVariable int index){

      URI catalogUrl = URI.create("http://localhost:8081/catalog/" + index);
      RestTemplate restTemplate = new RestTemplate();

      CircuitBreaker circuitBreaker = factory.create("circuitbreaker");
      String result = circuitBreaker.run(
               ()  -> restTemplate.getForObject(catalogUrl, String.class),
              err -> getFallback(index));
      return String.format("[%s] Item %d %s", LocalTime.now(), index, result);
   }

   public String getFallback(int i) { return "FALLBACK-ITEM-" + i;}
}
```

Catalog service

# Seeing a Circuit Breaker in Action

- To see the effect of the circuit breaker, follow these steps:
  - Stop the catalog service
  - Then ping the following client endpoints…

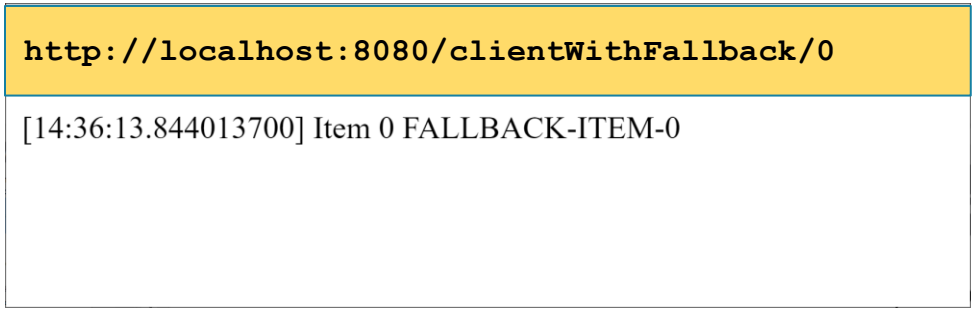
**http://localhost:8080/client/0**

**Whitelabel Error Page**

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Sep 29 14:34:07 BST 2021
There was an unexpected error (type=Internal Server Error, status=500).

**http://localhost:8080/clientWithFallback/0**



[14:36:13.844013700] Item 0 FALLBACK-ITEM-0

# Summary

- Consuming REST services
- Microservice architecture
- Circuit breakers