Messaging with Kafka

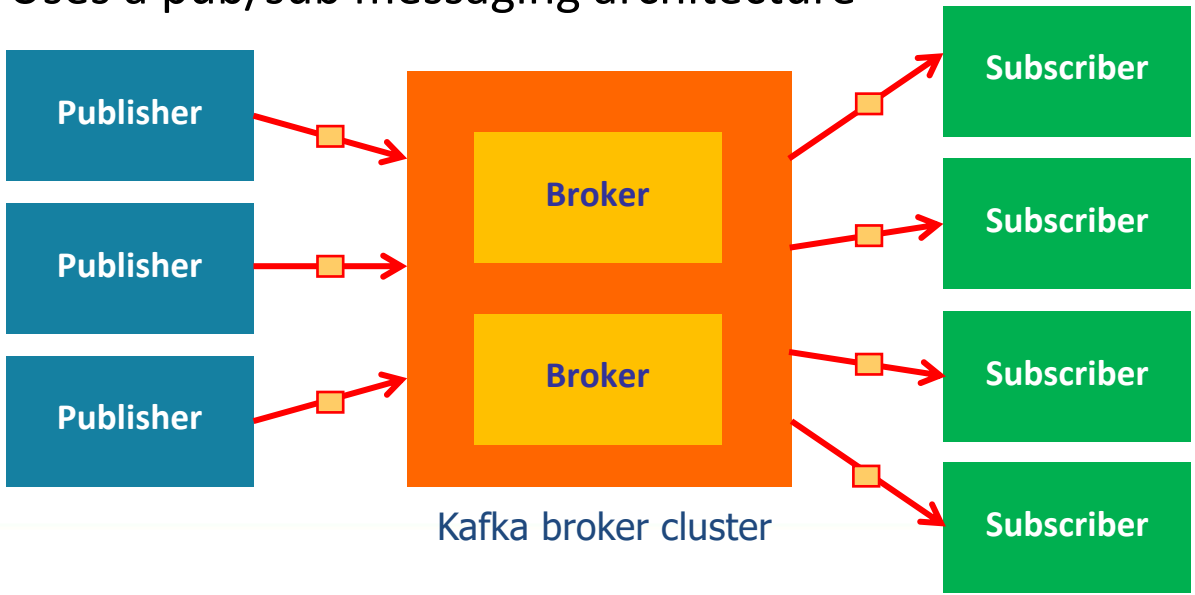
1. Overview of messaging using Kafka
2. Using Kafka in a Spring Boot application

1. Overview of Messaging using Kafka

- What is Kafka?
- Kafka in industry
- Installing Kafka
- A few words about Zookeeper
- Tweak the Kafka and Zookeeper config files
- Starting Zookeeper and Kafka

What is Kafka?

- Apache Kafka is a distributed message broker
  - Kafka runs as a cluster of broker nodes
  - Primary goal is high throughput, idea for cloud-scale architectures
  - Uses a pub/sub messaging architecture



Kafka in Industry

- Kafka is widely used by major players in industry
  - LinkedIn
  - Twitter
  - Netflix
  - Airbnb
  - Goldman Sachs
  - PayPal
  - Coursera
  - Hotels.com
  - Etc.

Installing Kafka

- In Windows…
  - Go to https://kafka.apache.org/downloads.html
  - Download and unzip the latest binary distribution, e.g. `kafka_2.12-2.8.1.tgz`

- In macOS:
  - Install Homebrew (if not already installed)
  - Then run the following command:

```
brew install kafka
```

A Few Words about Zookeeper

- Apache Kafka uses Apache Zookeeper to coordinate cluster info
  - Zookeeper is a distributed hierarchical key-value store
  - Provides a naming service for large distributed systems

- The Kafka download already includes Zookeeper
  - So you don’t need to download Zookeeper separately
  - You just need to run Zookeeper first, then you can run Kafka…

Tweak the Kafka and Zookeeper Config Files

- You must edit the Kafka and Zookeeper configuration files
  - Set the data log directories appropriately for your environment

- Edit the Kafka configuration file, located here:
  - `<Kafka_Home>/config/server.properties`

```
log.dirs=C:/temp/kafka-logs                    server.properties
```

- Edit the Zookeeper configuration file, located here:
  - `<Kafka_Home>/config/zookeeper.properties`

```
dataDir=C:/temp/zookeeper                      zookeeper.properties
```

Starting Zookeeper and Kafka on Windows

- To start Zookeeper on Windows:
  - Open a Command Prompt window
  - In the Kafka installation directory, run the following command:

```
bin\windows\zookeeper-server-start.bat config\zookeeper.properties
```

- To start Kafka on Windows:
  - Open another Command Prompt window
  - In the Kafka installation directory, run the following command:

```
bin\windows\kafka-server-start.bat config\server.properties
```

Starting Zookeeper and Kafka on macOS

- To start Zookeeper and Kafka on macOS:
  - Open a new Terminal window
  - In the Kafka installation directory, run the following commands:

```
zookeeper-server-start /usr/local/etc/kafka/zookeeper.properties &
kafka-server-start /usr/local/etc/kafka/server.properties
```

2. Using Kafka in a Spring Boot Application

- Spring Boot dependency for Kafka
- Configuring application properties
- Sending messages to a topic
- Consuming messages from a topic
- Example REST API to publish messages
- Pinging the REST API

Demo package: demo.kafka.simple

Spring Boot Dependency for Kafka

- To use Kafka in a Spring Boot application, add the following dependency in your pom file:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

pom.xml

Configuring Application Properties

- You can tell Kafka to create topics dynamically
  - This enables your application to send/receive from a topic, without you having to create the topic first

- To enable Kafka to create topics dynamically, define the following application property:

```
spring.kafka.listener.missing-topics-fatal=false
```

application.properties

Sending Messages to a Topic

- To send messages to a topic:
  - Autowire a `KafkaTemplate<K,V>` bean
  - Call `send(topicName,key,value)`
  - Note that Kafka messages are key-value pairs ☺

```
@Service
public class MyPublisher {

    private static final String TOPIC_NAME = "mytopic";

    @Autowired
    private KafkaTemplate<String,String> kafkaTemplate;

    public void sendMessage(String key, String value){
        this.kafkaTemplate.send(TOPIC_NAME, key, value);
    }
}
```

MyPublisher.java

Consuming Messages from a Topic (1 of 2)

- To consume messages from a topic, define a method and annotate it with:
  - `@KafkaListener(topics,groupId)`
- You can define multiple listeners for the same topic
  - One listener in each group will receive the message
- Listener methods receive the message value
  - Can also receive message header metadata, e.g. key, timestamp

Consuming Messages from a Topic (2 of 2)

- Example consumer in our demo app:

```
@Service
public class MyConsumer {

    @KafkaListener(topics = "mytopic", groupId="group1")
    public void group1ConsumerA(
            @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) String key,
            @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
            @Header(KafkaHeaders.RECEIVED_TIMESTAMP) String timestamp,
            String value) {

        // Process the message here…
    }
}
```

MyConsumer.java

Example REST API to Publish Messages

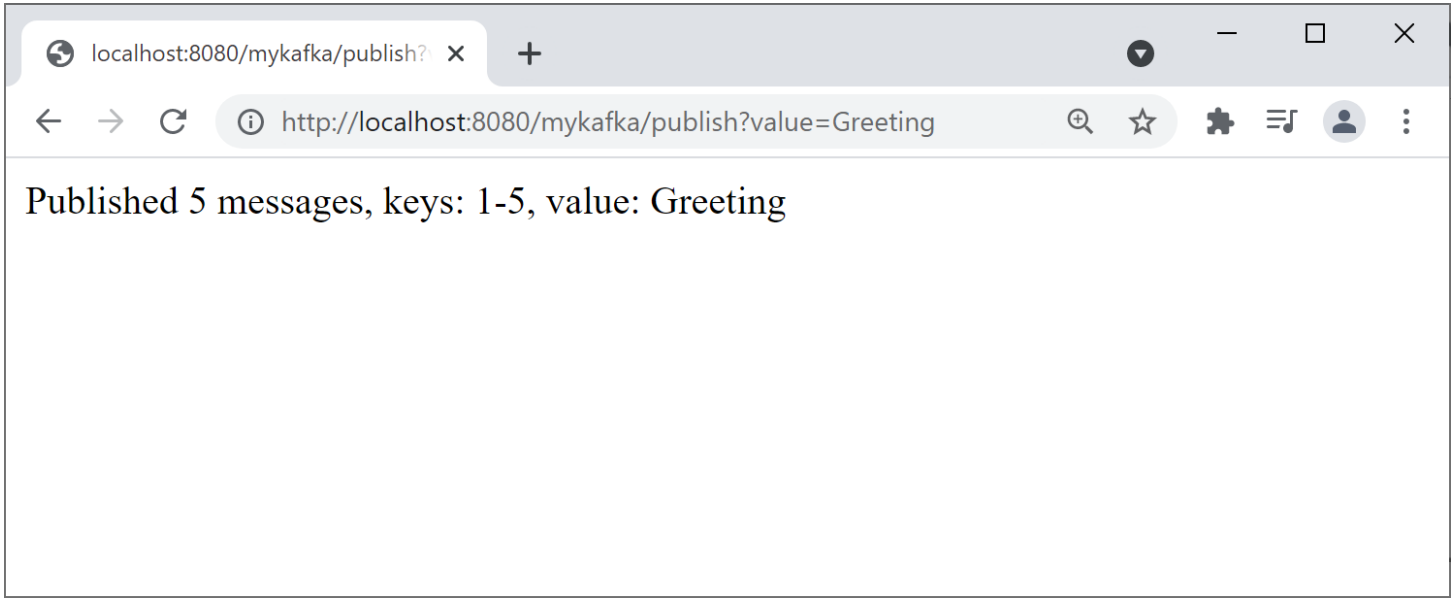
- The demo app has a REST controller
  - Allows users to publish messages to a topic

```
@RestController
@RequestMapping("/mykafka")
@CrossOrigin
public class MyRestController {

    @Autowired
    private MyPublisher publisher;

    @GetMapping("/publish")
    public String publish(@RequestParam("value") String value) {
        for (int i = 1; i <= 5; i++) {
            this.publisher.sendMessage("key" + i, value);
        }
        return "Published 5 messages, keys: 1-5, value: " + value;
    }
}
```

MyRestController.java

Pinging the REST API

- You can ping the REST API as follows:
  - http://localhost:8080/mykafka/publish?value=Greeting



Summary

- Overview of Kafka
- Using Kafka in a Spring Boot application

Exercise

- The simple example we've just shown sends messages with the following data types:
  - Key: `String`
  - Value: `String`
- Kafka lets you send any data types as keys/values
  - You must configure `KafkaTemplate`
  - Specify how to serialize/deserialize objects
  - See: `demo.kafka.customobjects`

Demo package: demo.kafka.customobjects

Containerizing a Spring Boot App

1. Introduction to containerization and Docker
2. Understanding Docker images
3. A closer look at images and containers
4. How to containerize a Spring Boot app

1. Introduction to Containerization and Docker

- What is containerization?
- Containers vs. virtual machines
- Docker editions and platforms
- Downloading and Installing Docker for Windows
- Starting Docker for Windows

What is Containerization?

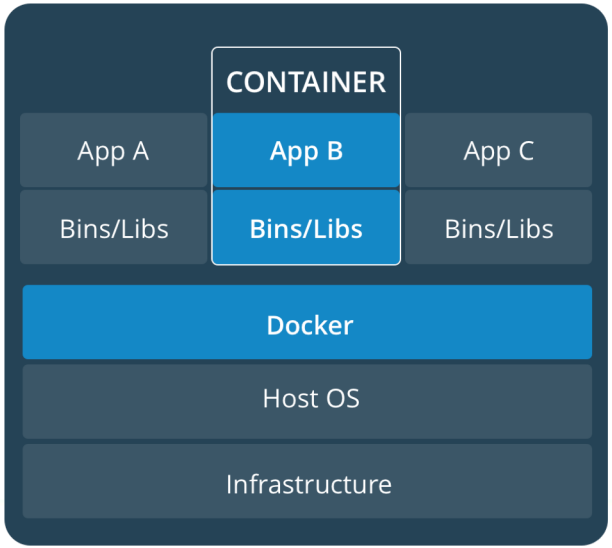
- Containerization is a way of wrapping an application, plus its environment, into a shrink-wrapped container
  - Makes it easy to deploy and run the application, because it runs in a virtualized environment

- Docker is a very popular containerization tool
  - You build an **image** that contains your app, properties, etc.
  - You then run the image - a running image is called a **container**

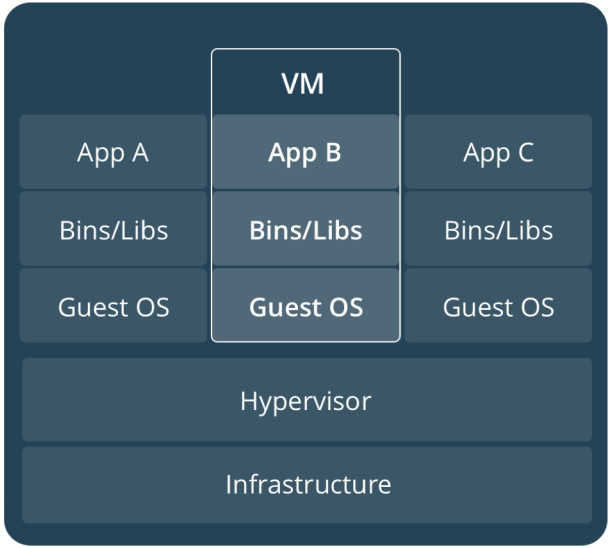
Containers vs. Virtual Machines

- Containers are much more lightweight that VMs
  - Containers run on top of the host OS, e.g. Linux
  - VMs are much bulkier because they actually contain a guest OS

Docker containers



Virtual machines



Docker Editions and Platforms

- Docker comes in two editions
  - Docker Community Edition (CE)
  - Docker Enterprise Edition (EE)

- You can install Docker on various platforms
  - Docker for Linux
  - Docker for Windows
  - Docker for Mac

- We'll be using Docker CE for Windows
  - Requires 64bit Windows 10 Pro/Enterprise/Education
  - For other versions of Windows, use Docker Toolbox instead

Downloading and Installing Docker for Windows

- Browse to:
  - https://hub.docker.com/editions/community/docker-ce-desktop-windows
- Click **Get Docker Desktop**, to download the installer
- When the installer has completely downloaded, run it
- When the installation is complete, click **Close**

Starting Docker for Windows

- To start Docker for Windows:
  - Hit the Windows button, and run **Docker Desktop** as administrator

- Give Docker a few minutes to start, then test it's working like so:
  - Open a Command Prompt window.
  - Run the command **docker version**
  - It should display a message indicating Docker is running

2. Understanding Docker Images

- Overview
- Images vs. containers
- Running a sample image
- Listing images in the local Docker registry

Overview

- A Docker **image** is a black box executable package
  - It includes everything needed to run an application

- E.g. a Docker image for a Java microservice might have:
  - A JVM
  - A web server (e.g. Tomcat)
  - Any additional JARs necessary, e.g. database drivers
  - A JAR containing your REST service

- In this section we're going to see how to download ("pull") and run a pre-built image from Docker Hub

Images vs. Containers

- When you run a Docker image…
  - Docker creates an in-memory instance of the image
  - This in-memory instance is called a **container**
  - You can run many container instances for an image, if you like

Running a Sample Image (1 of 3)

- Docker has a sample pre-built image called `hello-world`
  - You can run it as follows:

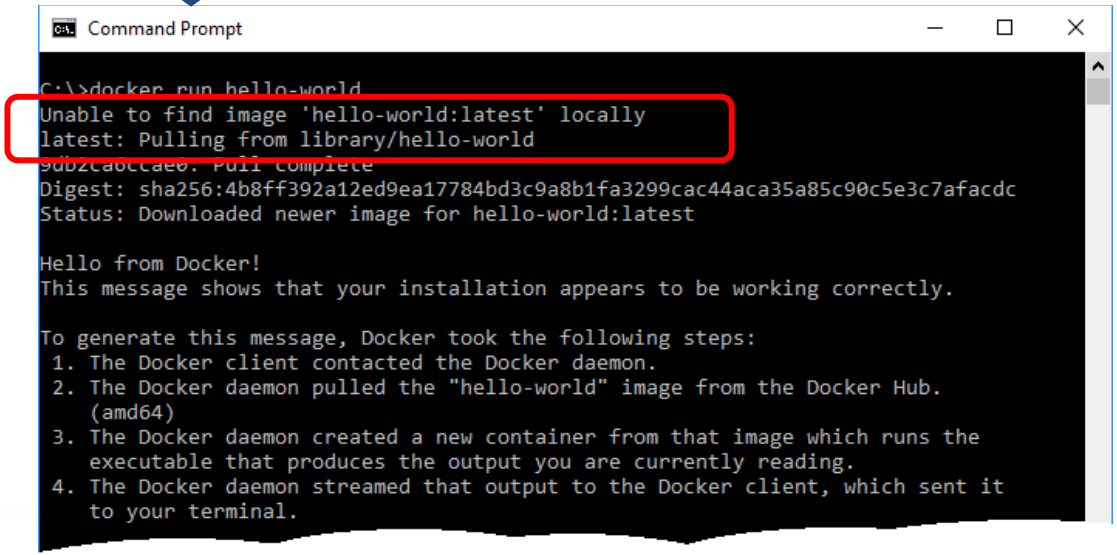
```
docker run hello-world
```

- Docker looks for the image in the local registry
  - The default location for images is `/var/lib/docker`
- If the image isn't in the local registry…
  - Docker pulls it from a global registry (e.g. Docker Hub)
  - Docker stores the downloaded image in the local registry
- Docker then runs the image
  - i.e. it creates a container, a running instance of the image

Running a Sample Image (2 of 3)

- Here's a screenshot of what happens
  - Note in particular, the "pull" request near the top

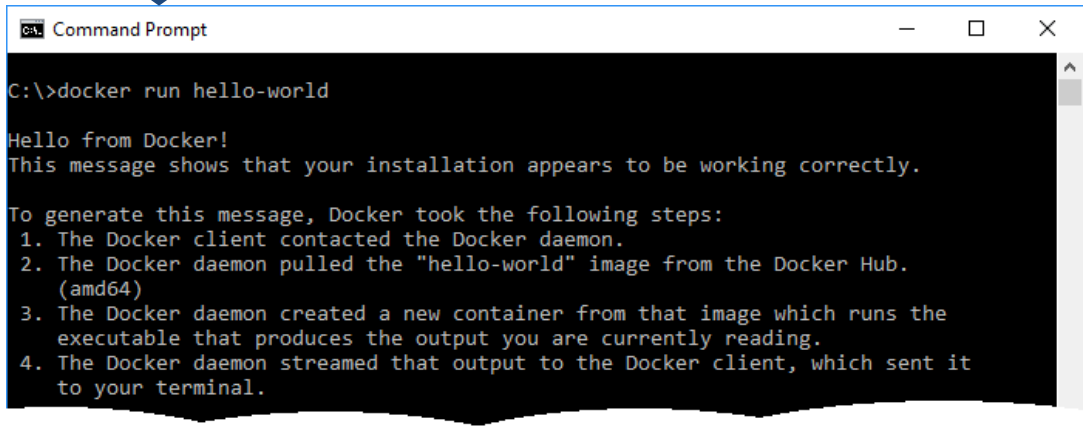
```
docker run hello-world
```



Running a Sample Image (3 of 3)

- Do another Docker run, and see what happens
  - Note there's no "pull" request this time – why not…?

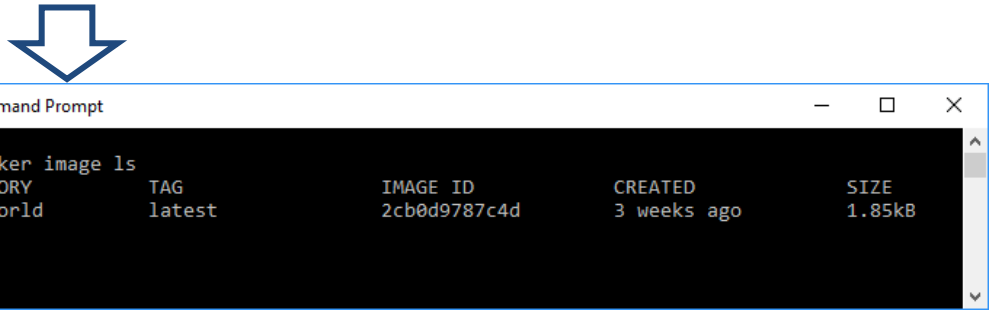
```
docker run hello-world
```



Listing Images in the Local Docker Registry

- You can get a list of all the Docker images in your local Docker registry, as follows:

```
docker image ls
```



3. A Closer Look at Images and Containers

- The power of containerization
- Running multiple containers from an image
- Running containers in detached mode
- Listing containers
- Stopping a container
- Pruning containers and images

The Power of Containerization (1 of 2)

- Docker Hub contains thousands of useful images, providing containerized shrink-wrapped functionality
  - E.g. Tomcat, MySQL, MongoDB, etc.

- E.g. run this command to download and run Tomcat

```
docker run -p 8123:8080 tomcat
```

- This downloads the Tomcat image into your local registry, and creates an instance of the image (i.e. a container)
  - Tomcat runs inside the container
  - Within the container, Tomcat listens on port 8080 by default
  - You can map it to any port on our computer, e.g. 8123 here

The Power of Containerization (2 of 2)

- You can ping Tomcat from your host computer
  - Specify port 8123

```
curl http://localhost:8123
```

- Docker maps the request to port 8080 within the container, which means Tomcat handles the request

Running Multiple Containers from an Image

- You can easily spin up another Tomcat container
  - Tomcat will run on port 8080 within that container
  - You must map it to a different port in your host O/S

```
docker run -p 8246:8080 tomcat
```

- You can ping this instance of Tomcat from your host computer
  - Specify port 8246

```
curl http://localhost:8246
```

Running Containers in Detached Mode

- You can run a container in “detached mode”
  - Specify the `-d` option
  - The container runs in the background

```
docker run -d -p 8369:8080 tomcat
```

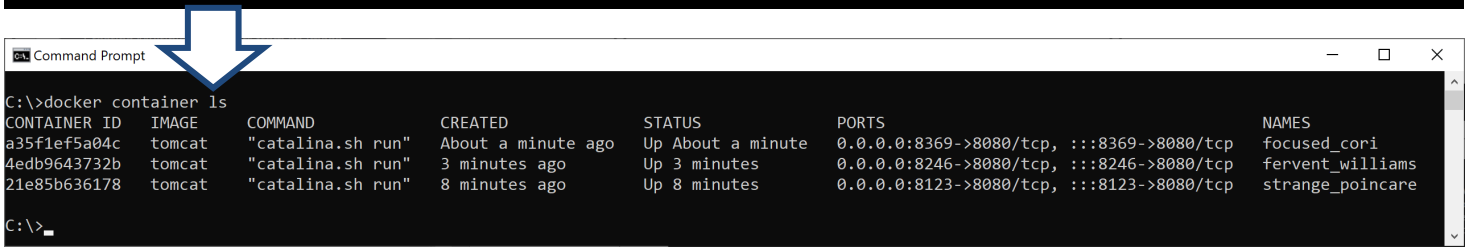
- You can ping this instance of Tomcat from your host computer
  - Specify port 8369

```
curl http://localhost:8369
```

Listing Containers

- You can get a list of all the containers currently running

```
docker container ls
```



- Each container has:
  - A unique container id (abbreviated)
  - The name of the image (of which this container is an instance)
  - The command that is executed within the container
  - Created timestamp and status
  - Port mappings
  - A name for the container (random name by default)

Stopping a Container

- To stop a container, run the following command with the container ID or name you want to stop

```
docker container stop focused_cori
```

- Even after you stop a container, Docker maintains information about that container (e.g. so you can view its logs)
  - You can list all containers (including stopped ones) as follows:

```
docker container ls -a
```

Pruning Containers and Images

- To completely remove all stopped containers:

```
docker container prune
```

- To completely remove all dangling images:

```
docker image prune
```

4. How to Containerize a Spring Boot App

- Overview
- Running the application as normal
- Bundling the application in a JAR
- Defining a Dockerfile
- Understanding the Dockerfile
- Building the image
- Viewing images in the local Docker registry
- Running a container

Demo package: demo.containerization

Overview

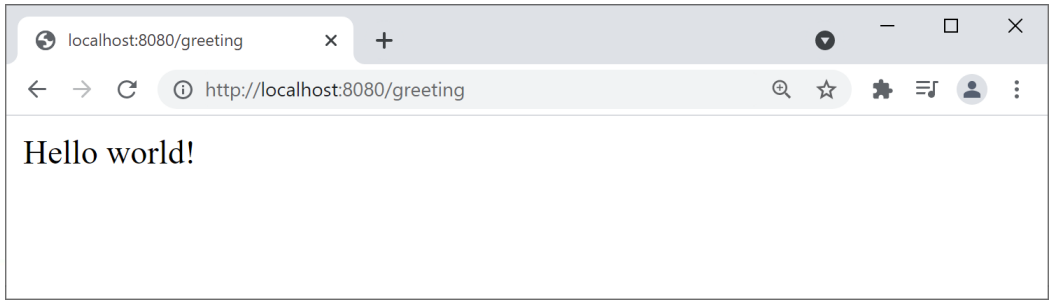
- In this section we'll see how to containerize a Spring Boot app
  - We'll build a Docker image that contains a Spring app
  - Then we'll run a container (i.e. an instance of the Docker image)
  - Our Spring Boot app will run on a JVM inside the container

- See demo project, `demo-15-containerization`
  - Take a look at `Application.java`
  - It's a simple Spring Boot app with a REST service

Running the Application as Normal

- You can run the application as normal
  - Right-click `Application.java`, then Run Application

- This runs the application directly on your host computer
  - The application contains an embedded web server (Tomcat)
  - Tomcat listens on port 8080 on your host computer
  - You can ping it via `http://localhost:8080/greeting`



Bundling the Application in a JAR

- If you want to run a Java app in a Docker container…
  - The first step is to bundle the app into a JAR file

- To bundle the app into a JAR:
  - Open a Terminal window in the project root folder
  - Run the following Maven command

```
.\mvnw package -DskipTests
```

- This creates the JAR file:
  - `target/demo-15-containerization-0.0.1.jar`

Defining a Dockerfile (1 of 2)

- Now we're ready to see how to create a Docker image
  - Remember, a Docker image is a “black box” executable package
  - It includes everything needed to run an application

- In our case, we'll create a Docker image containing:
  - A JVM
  - Our Spring Boot JAR file
  - A command to execute the Spring Boot JAR file on the JVM

Defining a Dockerfile (2 of 2)

- In order to define a Docker image, define a special file named `Dockerfile` (by default)
  - Specifies build instructions, so Docker can build an image

- See this `Dockerfile` in the demo project (root folder)

```
FROM openjdk:11
ARG JAR_FILE
COPY ${JAR_FILE} myapp.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/myapp.jar"]
```

- See following slides for an explanation
  - Also see https://docs.docker.com/engine/reference/builder/

Understanding the Dockerfile (1 of 3)

```
FROM openjdk:11
ARG JAR_FILE
COPY ${JAR_FILE} myapp.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/myapp.jar"]
```

- A Dockerfile starts with a `FROM` instruction
  - Specifies the "base image" from which we are building
  - Our image will be based on OpenJDK version 11
  - OpenJDK is an open-source implementation of Java SE

- When we run this Dockerfile to build our image…
  - Docker will see if we've already downloaded `openjdk:11`
  - If we haven't, Docker will pull it from the Docker Hub

Understanding the Dockerfile (2 of 3)

```
FROM openjdk:11
ARG JAR_FILE
COPY ${JAR_FILE} myapp.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/myapp.jar"]
```

- When you build a Docker image, you can pass arguments into the Docker build command
  - In the Dockerfile, use `ARG` statements to capture these arguments
  - In our example, the `JAR_FILE` arg specifies the name of our JAR

- A Dockerfile specifies files to copy into the Docker image
  - Use `COPY` instructions to copy files into the Docker image
  - In our example, we copy our JAR file into the image
  - Inside the image, the JAR file will be named `myapp.jar`

Understanding the Dockerfile (3 of 3)

```
FROM openjdk:11
ARG JAR_FILE
COPY ${JAR_FILE} myapp.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/myapp.jar"]
```

- The `EXPOSE` instruction:
  - Acts as documentation about port(s) inside the container
  - Indicate this port must be mapped to a port on the host

- The `ENTRYPOINT` instruction:
  - Specifies what to actually execute inside the Docker image
  - In our example, we run our JAR on the JVM in the image

Building the Image

- Use the `docker build` command to build a Docker image
  - Type the following, all on one line
  - It reads and executes the instructions in the Dockerfile

```
docker build -t my-spring-boot-app
             --build-arg JAR_FILE=target/demo-15-containerization-0.0.1.jar
             .
```

- `-t`
  - Specifies the tag name for the image
  - Tells Docker to create an image with this name in the local registry

- `--build-arg`
  - Specifies a value for a build argument
  - Followed by a name=value pair

Viewing Images in the Local Docker Registry

- You can view images in the Docker registry

```
docker image ls
```

Command Prompt

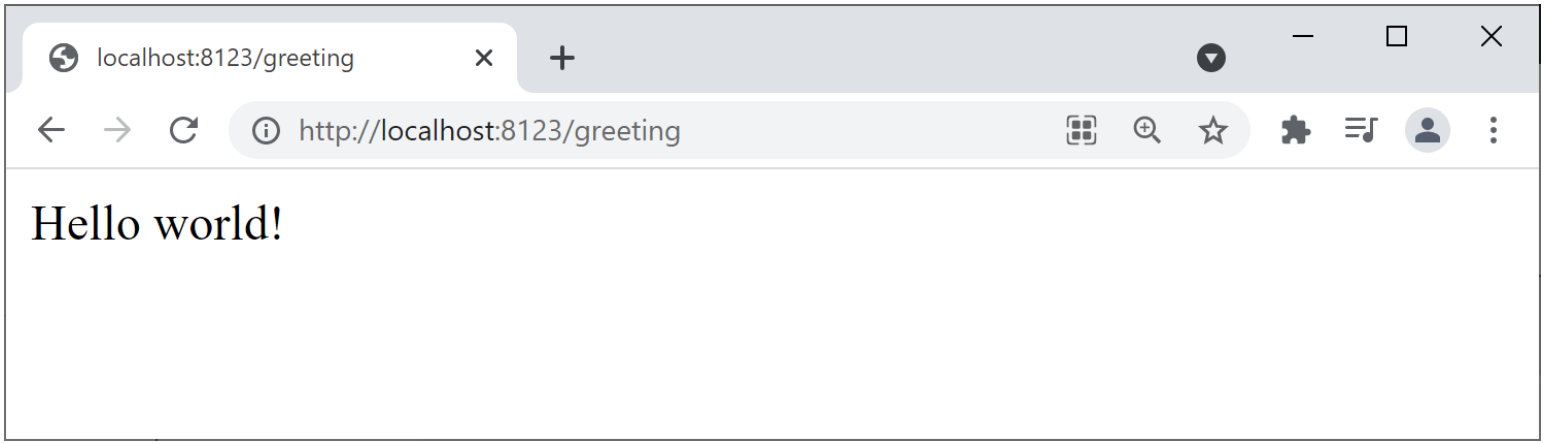
```
C:\>docker image ls
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
my-spring-boot-app  latest    5ef351d00100   46 seconds ago   677MB
hello-world         latest    feb5d9fea6a5   4 days ago       13.3kB
tomcat              latest    bb832de23021   2 weeks ago      680MB

C:\>
```

Running a Container

- You can run a container as normal

```
docker run --name app -d -p 8123:8080 my-spring-boot-app
```

- You can then ping as normal, via the mapped port



Summary

- Introduction to containerization and Docker
- Understanding Docker images
- A closer look at images and containers
- How to containerize a Spring Boot app

Exercise

- Modify your Spring Boot app (e.g. add an `index.html` file) and then rebuild the JAR file:

```
.\mvnw package -DskipTests
```

- Forcibly remove the old container and image:

```
docker container rm -f app
```

```
docker image rm -f my-spring-boot-app
```

- Rebuild the image and run another container:

```
docker build -t my-spring-boot-app --build-arg JAR_FILE=target/demo-15-containerization-0.0.1.jar  .
```

```
docker run --name app -d -p 8123:8080 my-spring-boot-app
```

Spring Cloud Microservices

1. Overview of microservices
2. Microservices application example
3. Implementing circuit breaker behaviour

1. Overview of Microservices

- The challenges
- What are microservices?
- Characteristics of a microservice architecture
- Benefits of the microservices approach
- Microservices and the cloud
- Microservices and Spring

The Challenges

- Globalization and interconnectivity place new demands on organizations and IT departments…
  - Applications need to communicate with many external service providers over the Internet - the age of silo applications is over
  - Customers expect incremental product updates and feature upgrades, rather than complete product releases once a year
  - Architectures must be flexible enough to scale up across multiple servers quickly when volume spikes
  - Availability and resilience in the worldwide market are essential

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.

Characteristics of a Microservice Architecture

- Microservices are a move away from monolithic architectures
  - Functionality is delivered as fine-grained distributed components

- Each microservice is **highly cohesive**
  - Has responsibility for a very specific piece of domain logic
  - Has well-defined boundaries
  - The implementation technology of a microservice is irrelevant

- Microservices are **loosely coupled**
  - Each microservice is deployed independently of other ones
  - Communicate via technology-neutral protocols, e.g. HTTP, JSON

Benefits of the Microservices Approach

- Scalability
  - Microservices can be distributed across multiple servers
  - Easier to scale-out specific services as needed
- Flexibility
  - Microservices offer a finer level of granularity than traditional apps
  - Easier to compose and rearrange to deliver new functionality
- Resilience
  - Microservices are decoupled, so they degrade/fail in isolation
  - Failures can be contained locally, without crashing the whole app

Microservices and the Cloud

- Microservices are ideally suited for deployment on the cloud
  - Easy to deploy individually
  - Typically small in size, so it's OK to start up a large number of the same microservice if demand spikes
  - Increases scalability and resilience

Microservices and Spring

- Spring Boot and Spring Cloud are well suited to microservices
- Spring Boot
  - Focuses on common core development features for creating and packaging REST-oriented microservices
- Spring Cloud
  - Makes it simple to deploy and operate microservices in the cloud (public or private)

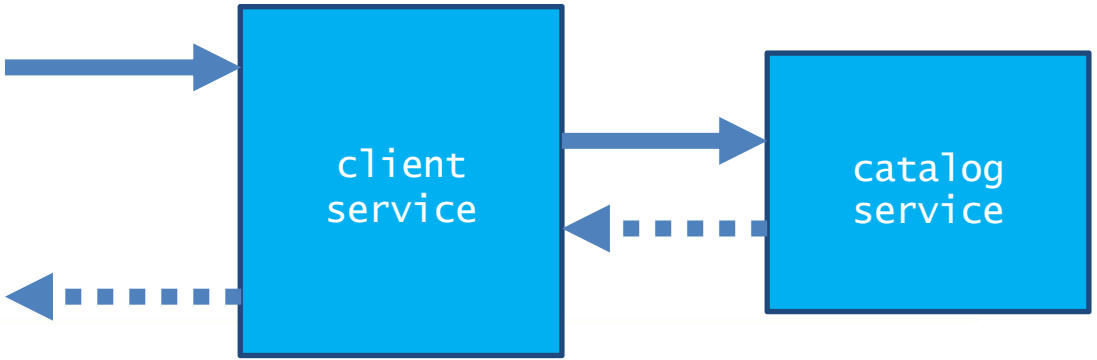
2. Microservices Application Example

- Overview
- Implementing the catalog service
- Implementing the client service

Overview

- In this section we'll show a complete (simple) example of how to create a Spring Cloud microservice application

- There are two Spring Boot applications in the demo:
  - `demo-16-clientservice`
  - `demo-16-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-16-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-16-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. Implementing Circuit Breaker Behaviour

- 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)
  - Netfix 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>2.0.2</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

- Overview of microservices
- Microservices application example
- Implementing circuit breaker behaviour

Authentication using OAuth2

1. Essential concepts
2. Using OAuth2 in Spring Boot

1. Essential Concepts

- Overview of Spring Boot Security
- Spring authentication and authorization
- Overview of OAuth2

Overview of Spring Boot Security

- Spring encapsulates security, offering the following benefits:
  - Portable
    - Portable across web containers (and standalone)
    - No need for platform-specific declarations or role-mappings
  - Comprehensive
    - Supports common web authentication techniques
  - Elegant
    - Security is decoupled from application logic
    - Achieved via Spring AOP and security filters

Spring Authentication and Authorization

- Spring authentication:
  - Establishes a security context
  - Security context contains info about the authenticated principal

- Spring authorization:
  - Examines the security attributes of a secured item
  - Gets principal information from the security context
  - Grants or denies access to the secured item

Overview of OAuth2

- OAuth2 is a client framework
  - Enables access to a user's resources…
  - With the user's consent…
  - Without exposing the user's username/password

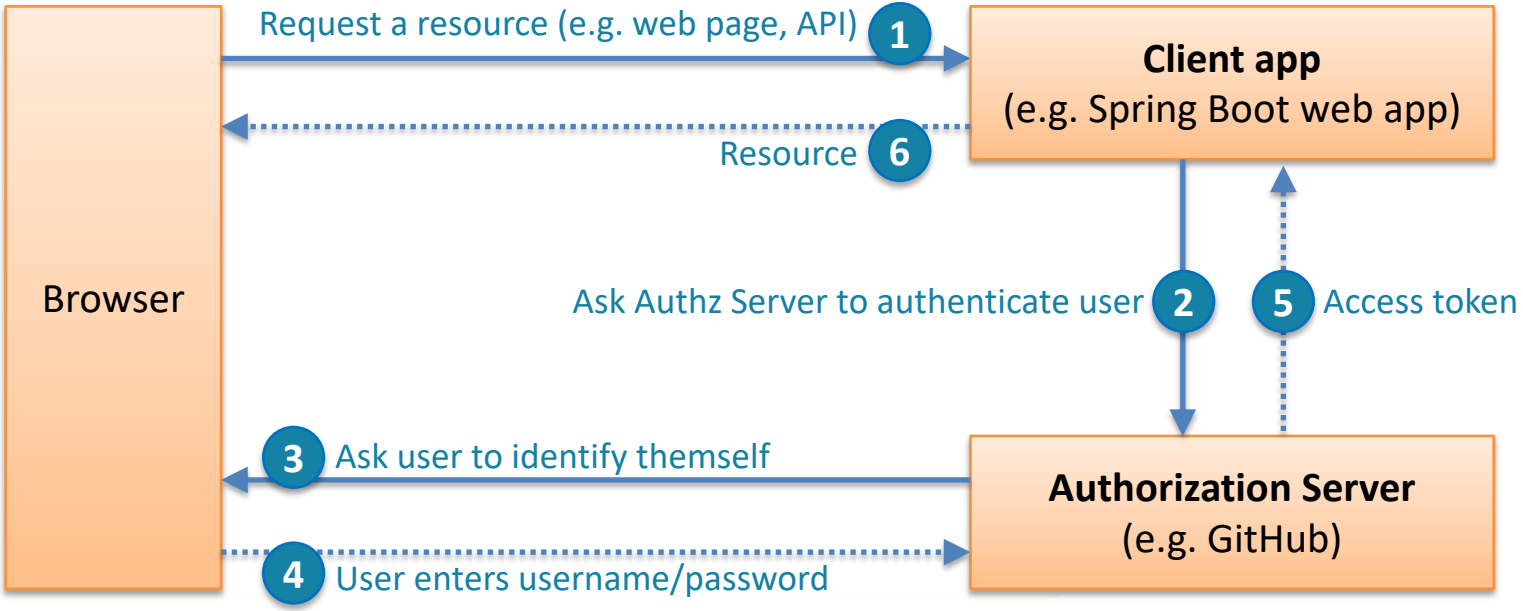
- For example:
  - The user tries to access an endpoint in a Spring Boot web app
  - The web app redirects to a social-media login page, where the user is challenged to authenticate themselves
  - The social media provider returns an "access token" to the web app, which represents the user's authenticated identity

2. Using OAuth2 in Spring Boot

- Overview
- Spring Boot dependency for OAuth2
- Resources in the demo app
- Specifying authentication rules
- Registering your client app with GitHub
- Adding GitHub credentials to your client app
- Running the client app
- Displaying additional info about the user

Overview

- In this section we'll show an example of how to how to use OAuth2 in a Spring Boot client app



Spring Boot Dependency for OAuth2

- Take a look in the demo app
  - `demo-17-oauth2-client`

- Note the pom file includes the OAuth2 dependency
  - This automatically pulls in the Spring Security library too

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

pom.xml

Resources in the Demo App

- The demo app has several resources the user might request
  - `src/main/resources/static/index.html`
  - `src/main/java/mypackage/Controller1.java`
  - `src/main/java/mypackage/Controller2.java`

- By default Spring Boot protects all URLs in your web app
  - i.e. the user must be authenticated to access any URL

Specifying Authentication Rules

- You can specify authentication rules as follows:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
          .authorizeRequests()
              .antMatchers("/", "/controller1").permitAll()
              .antMatchers("/controller2").authenticated()
              .anyRequest().authenticated()
              .and()
          .oauth2Login();
    }
}
```

SecurityConfig.xml

Registering your Client App with GitHub (1 of 3)

- The previous slide specified we want to use OAuth2 to perform authentication
- You must tell OAuth2 how to contact an authorization server in order to do that task
  - E.g. GitHub
- The first step is to register your application with GitHub
  - So GitHub is aware of your application…
  - So GitHub will be willing to authenticate users on its behalf

Registering your Client App with GitHub (2 of 3)

- To register your client app with GitHub:
  - Sign in to https://github.com/settings/developers
  - Click **OAuth apps**, then click **New OAuth App**

- Then specify the following info:
  - Application name **My Cool App**
  - Homepage URL **http://localhost:8080**
  - Authz callback URL **{homeUrl}/login/oauth2/code/{registrationId}**

http://localhost:8080 **github**

(This is where the browser will be redirected after successful authorization)

Registering your Client App with GitHub (3 of 3)

- You will then be able to enter additional info about the app
  - E.g. the owner of the app
  - E.g. a logo for the app
  - E.g. add the app to GitHub Marketplace

- You must also grab the following credentials from GitHub, which you will need to add into your client app (see next slide)
  - **Client ID**
  - **Client secret**

Adding GitHub Credentials to your Client App

- In your Spring Boot app, add the GitHub credentials (from previous slide) to your `application.properties` file

```
spring.security.oauth2.client.registration.github.client-id=<client-id>
spring.security.oauth2.client.registration.github.client-secret=<client-secret>
```

- This is what will happen when a user accesses a web resource:
  - If the web resource is protected…
  - OAuth2 will contact GitHub, passing the client credentials above
  - GitHub will challenge the user to authenticate themself
  - GitHub will return an access token 👍 to your client app

Running the Client App

- Run the client app and try to access the following resources:
  - `/` (no authentication needed)
  - `/controller1` (no authentication needed)
  - `/controller2` (you'll be redirected to GitHub to authenticate)

Displaying Additional Info About the User

- During authentication, GitHub also returned info about the user
  - You can access this info in your client app as follows:

```
@RestController
public class Controller2 {

    @GetMapping(path="/controller2-with-info")
    public String getWithName(@AuthenticationPrincipal OAuth2User principal) {
        return "Hello from Controller2, " +
                principal.getAttribute("name") + ", " +
                principal.getAttribute("company");
    }
    …
}
```

Controller2.java

- To see this in action, ping `/controller2-with-info`

Summary

- Essential concepts
- Using OAuth2 in Spring Boot

Exercise

- We've seen how to use OAuth2 to perform authentication, in the following project:
  - `demo-17-oauth2-client`

- Spring Boot also supports other authentication techniques, e.g. DIY-style forms authentication, see here:
  - `demo-17-diy-security`

Testing Spring Boot Applications

1. The Spring Boot test ecosystem
2. Writing and running tests on beans
3. Mocking beans
4. Additional Spring Boot test techniques

1. The Spring Boot Test Ecosystem

- Overview
- Spring Boot test dependency
- Defining test cases in Spring Boot
- Understanding `@SpringBootTest`
- Specifying Java config classes for tests
- Specifying properties for tests
- Specifying a web environment for tests

Overview

- Spring Boot makes testing easy, in various ways…
- Spring Boot auto-configuration automatically sucks in common test libraries
- Spring Boot automatically makes components available for autowiring into your test cases
- Spring Boot automatically loads application properties, so you can test components with realistic settings

Spring Boot Test Dependency

- When you create a Spring Boot app with Spring Initializr, it has the `spring-boot-starter-test` dependency

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

pom.xml

JUnit 5

Mockito
Mocking framework

Hamcrest
Matcher library

Spring Test and Spring Boot Test utilities

AssertJ
Fluent assertion library

JsonPath
XPath for JSON

Objenesis
Object instantiation library

Defining Test Cases in Spring Boot

- Spring Initializr also generates a simple JUnit test case
  - See the `src/test/java` folder in the demo project

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class ApplicationTests {

    @Test
    void contextLoads() {
    }
    …
}
```

ApplicationTests.java

- We discuss the details on the following slides…

Understanding @SpringBootTest

```
@SpringBootTest
class ApplicationTests {
    …
}
```

- `@SpringBootTest` automatically loads Java config classes, which presumably define the beans you want to test
  - First it loads inner classes annotated with `@Configuration`
  - If none found, it loads your `@SpringBootApplication` class

- `@SpringBootTest` also loads `application.properties`
  - So the beans are initialized properly when you test them

Specifying Java Config Classes for Tests

- `@SpringBootTest` has a `classes` attribute
  - Specifies particular Java config classes you want to load
  - Enables you to control which beans are created for your tests

```
@SpringBootTest(classes={MyJavaConfig1.class, MyJavaConfig2.class})
public class ApplicationTests {
    …
}
```

Specifying Properties for Tests

- `@SpringBootTest` has a `properties` attribute
  - Specifies additional properties you want to use in your tests
  - You specify an array of key=value strings

```
@SpringBootTest(properties={"prop1=value1", "prop2=value2"})
public class ApplicationTests {
    …
}
```

Specifying a Web Environment for Tests

- `@SpringBootTest` has a `webEnvironment` attribute
  - Enables you to configure a web environment for your tests

- To use a mock servlet environment:

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.MOCK)
```

- To use a real server on the port defined by `server.port`:

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.DEFINED_PORT)
```

- To use a real web server on a random port number:

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.RANDOM_PORT)
```

2. Writing and Running Tests on Beans

- Defining a bean to test
- Writing a test for a bean
- Running tests

Defining a Bean to Test

- Here's a simple bean to test

```
@Component
public class BankAccountBean {

    private String name;
    private int balance = 0;

    public void deposit(int amount) {
        balance += amount;
    }

    public void withdraw(int amount) {
        if (amount > balance)
           throw new IllegalArgumentException("Insufficient funds");
        balance -= amount;
    }

    // Plus getters, setters, toString(), etc.
}
```

BankAccountBean.java

Writing a Test for a Bean

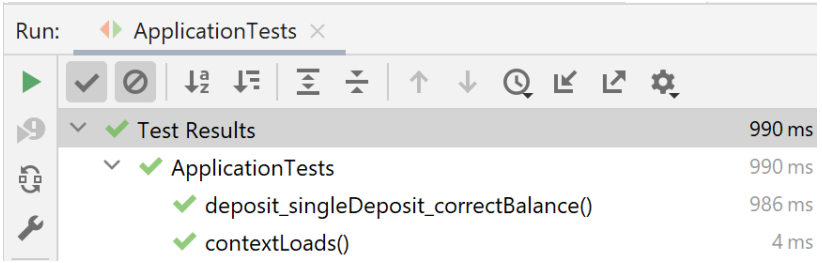
- Here's how we can test the bean
  - Spring loads the application context, as previously discussed
  - So we can autowire the bean into our test case, and then test it

```
import static org.junit.jupiter.api.Assertions.assertEquals;

@SpringBootTest
public class ApplicationTests {

    @Autowired
    BankAccountBean fixture;

    @Test
    public void deposit_singleDeposit_correctBalance() {
        fixture.deposit(100);
        assertEquals(100, fixture.getBalance());
    }
}
```

ApplicationTests.java

Running Tests

- Run tests as normal
  - See if the tests pass or fail



- Also note the info displayed in the console
  - Indicates the Spring application context has been loaded

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.5.4)

2021-09-29 18:38:19.529  INFO 17600 --- [           main] demo.testing.ApplicationTests            : Starting ApplicationTests using Java 11.0.
2021-09-29 18:38:19.533  INFO 17600 --- [           main] demo.testing.ApplicationTests            : No active profile set, falling back to def
2021-09-29 18:38:20.368  INFO 17600 --- [           main] demo.testing.ApplicationTests            : Started ApplicationTests in 1.311 seconds
```

3. Mocking Beans

- Overview
- Java mocking frameworks
- Example bean to test
- Testing the bean using Mockito mocks

Overview

- Object-oriented systems involve lots of interacting objects
  - Unit testing focuses on the behaviour of an isolated object

- We can use a mocking framework to create a "mock" of other objects that we use
  - Specify what methods you expect to be called on a mock object
  - Specify what you want the methods to return

Java Mocking Frameworks

- There are various Java mocking frameworks available:
  - Mockito
  - jMock
  - EasyMock
  - Mock Objects
  - etc…

- The `spring-boot-starter-test` dependency automatically includes the Mockito library
  - To use an alternative mocking framework, add the appropriate dependency to your pom file

Example Bean to Test

- Here's an example bean that we're going to test
  - Note that it has an autowired `BARepository` dependency

```
@Component
public class BAServiceBean {

    private BARepository repo;

    @Autowired
    public BAServiceBean(BARepository repo) {
        this.repo = repo;
    }

    public void depositIntoAccount(int id, int amount) {
        BankAccountBean acc = repo.getById(id);
        acc.deposit(amount);
        repo.update(id, acc);
    }
}
```

```
public interface BARepository {
    …
}
```

BAServiceBean.java

Testing the Bean using Mockito Mocks (1 of 2)

- Spring Boot has a `@MockBean` annotation
  - Tells Mockito to create a mock bean in the application context

```
@SpringBootTest
public class BAServiceBeanTests {

    @MockBean
    private BARepository mockRepo;

    @Autowired
    BAServiceBean service;

    …
}
```

Spring Boot will create a mock instance of `BARepository` in the application context

Spring Boot will inject the `BARepository` mock bean into the `BAServiceBean` bean

BAServiceBeanTest.java

Testing the Bean using Mockito Mocks (2 of 2)

- You can now write tests as follows:

```
@SpringBootTest
public class BAServiceBeanTests {

    …

    @Test
    public void testDeposit() {
        BankAccountBean acc = new BankAccountBean();
        when(mockRepo.getById(anyInt())).thenReturn(acc);

        service.depositIntoAccount(1234, 100);
        assertEquals(acc.getBalance(), 100);

        verify(mockRepo).getById(eq(1234));
        verify(mockRepo).update(eq(1234), refEq(acc));
    }
}
```

BAServiceBeanTest.java

Specify return value for mocked methods

Verify mocked methods were called as expected

4. Additional Spring Boot Test Techniques

- Testing Spring Data repositories
- Testing REST controllers

Testing Spring Data Repositories (1 of 2)

- Earlier in the course we discussed Spring Data repositories
  - Define an interface that extends `CrudRepository`
  - Define query methods, which Spring automatically implements

```
public interface EmployeeRepository extends CrudRepository<Employee, Long> {

    List<Employee> findsByRegion(String region);

    @Query("select e from Employee e where e.dosh >= ?1 and e.dosh <= ?2")
    List<Employee> findInSalaryRange(double from, double to);

    Page<Employee> findByDoshGreaterThan(double sal, Pageable pageable);
}
```

- How can you test these repositories?
  - See next slide…

Testing Spring Data Repositories (2 of 2)

- Spring Boot makes it easy to test Spring Data repositories
  - Define a test class and annotate with `@DataJpaTest`
  - Use a `TestEntityManager` to prepare database state

```
@DataJpaTest  // Configures in-mem db, and does JPA-related config only.
public class EmployeeRepositoryTest {

    @Autowired
    private TestEntityManager em;  // Has some additional test-related APIs.

    @Autowired
    private EmployeeRepository repository;

    @Test
    public void testFindByRegion() {
        em.persist(new Employee(-1, "John Smith", 25000, "London"));
        em.persist(new Employee(-1, "Jane Evans", 30000, "Dublin"));
        List<Employee> emps = repository.findByRegion("London");
        assertEquals(1, emps.size());
    }
}
```

Testing REST Controllers

- Spring Boot makes it easy to test REST controllers
  - In `@SpringBootTest`, set the `webEnvironment` property
  - Inject a `TestRestTemplate`, a test-friendly version of `RestTemplate` that doesn't throw exceptions for server errors

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.RANDOM_PORT)
public class SomeRestControllerTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testGetAllProducts() {

        ResponseEntity<List<Product>> responseEntity = restTemplate.exchange(
                "/full/products", HttpMethod.GET, null,
                new ParameterizedTypeReference<List<Product>>() {});

        List<Product> responseBody = responseEntity.getBody();
        assertEquals(HttpStatus.OK, responseEntity.getStatusCode());
        assertEquals(4, responseBody.size());   // Let's say we expect 4 products.
    }}
```

Summary

- The Spring Boot test ecosystem
- Writing and running tests on beans
- Mocking beans
- Additional Spring Boot test techniques

Exercise

- Run the demo **Spring Data repository** test in this project:
  - `demo-10-spring-data-repositories`
- Run the demo **REST controller** test in this project:
  - `demo-12-full-rest-services`