Getting started

Checkout the `background-jobs-rabbit-start` tag.

```
git checkout background-jobs-rabbit-start
```

Take some time to review the given solution to the previous lab.

During this lab we are going to introduce RabbitMQ and use it to trigger the job. Our app will have a controller action that will help us publish the message onto a queue. We will also implement a consumer that will perform the job when receiving the message.

Setting up RabbitMQ

You should already have RabbitMQ from a previous lab. Find the `rabbitmq-server` then run it in a terminal.

Visit the queue admin interface (http://localhost:15672/#/queues) and create a queue named `my-rabbit-queue`.

Setting up the RabbitMQ connection in Spring Boot

Add the following `dependencyManagement` configuration above the `dependencies` tag of your `pom.xml`.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Camden.SR3</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Then add the following dependencies:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-java-dsl</artifactId>
</dependency>
```

Add the following to the top level of your `application.yml`.

```
rabbitmq:
  uri: amqp://guest:guest@localhost
  queue: my-rabbit-queue
```

Finally, create the following `RabbitConfig` class.

```
@Configuration
public class RabbitConfig {

    @Value("${rabbitmq.uri}") String rabbitMqUri;

    @Bean
    public ConnectionFactory connectionFactory() {
        CachingConnectionFactory factory = new CachingConnectionFactor
y();
        factory.setUri(rabbitMqUri);
        return factory;
    }
}
```

Triggering the message

Create a controller action that we can invoke via HTTP to publish the message onto the queue.

```
@RestController
public class RabbitMessageController {

    private final RabbitTemplate rabbitTemplate;
    private final String queue;

    public RabbitMessageController(ConnectionFactory connectionFactory
, @Value("${rabbitmq.queue}") String queue) {
        this.rabbitTemplate = new RabbitTemplate(connectionFactory);
        this.queue = queue;
    }

    @PostMapping("/rabbit")
    public Map<String, String> publishMessage() {
        rabbitTemplate.convertAndSend(queue, "This text message will t
rigger the consumer");

        Map<String, String> response = new HashMap<>();
        response.put("response", "This is an unrelated JSON response")
;
        return response;
    }
}
```

Consuming the message

Replace `AlbumsUpdateScheduler` with `AlbumsUpdateMessageConsumer` using the following implementation:

```java
package org.superbiz.moviefun.albums;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.messaging.Message;
import org.springframework.stereotype.Component;

@Component
public class AlbumsUpdateMessageConsumer {

    private final AlbumsUpdater albumsUpdater;
    private final Logger logger = LoggerFactory.getLogger(getClass());

    public AlbumsUpdateMessageConsumer(AlbumsUpdater albumsUpdater) {
        this.albumsUpdater = albumsUpdater;
    }


    public void consume(Message<?> message) {
        try {
            logger.debug("Starting albums update");
            albumsUpdater.update();
            logger.debug("Finished albums update");

        } catch (Throwable e) {
            logger.error("Error while updating albums", e);
        }
    }
}
```

Now add the following to the `RabbitConfig` class.

```
@Value("${rabbitmq.queue}") String rabbitMqQueue;

@Bean
public IntegrationFlow amqpInbound(ConnectionFactory connectionFactory
, AlbumsUpdateMessageConsumer consumer) {
    return IntegrationFlows
        .from(Amqp.inboundAdapter(connectionFactory, rabbitMqQueue))
        .handle(consumer::consume)
        .get();
}
```

Test locally

- Run the app
- Hit the `/rabbit` endpoint with curl

  ```
  curl -X POST http://localhost:8080/rabbit -d ""
  ```

  ```
  {"response": "This is an unrelated JSON response"}
  ```

- Check the logs for `Starting albums update` and `Finished albums update`

Deploying to Cloud Foundry

- Create a Rabbit MQ service and bind it to the app

  ```
  cf create-service p.rabbitmq standard rabbit
  cf bind-service moviefun rabbit
  ```

- Create the queue
  - Find the `Dashboard` url

    ```
    cf service rabbit
    ```

```
...
Dashboard: https://pivotal-rabbitmq.run...
...
```

    - Visit the url provided on the `Dashboard` line.
    - Create the `my-rabbit-queue` queue as we did locally.

- Build and push the app

  ```
  mvn clean package -DskipTests -Dmaven.test.skip=true
  cf push moviefun -p target/moviefun.war
  ```

- Tail the logs in a separate terminal

  ```
  cf logs moviefun
  ```

- Trigger the job using the `/rabbit` endpoint, for example:

  ```
  curl -X POST http://moviefun-pretypographical-bombshell.cfapps-01.
  haas-66.pez.pivotal.io/rabbit -d ""
  ```

  ```
  {"response":"This is an unrelated JSON response"}
  ```

The logs should show that *only one* instance consumed the message.

Assignment

Once you are done with the lab and the application is deployed and working on PCF, you can submit the assignment using the `submitReplatformingBackgroundJobsWithAmqp` gradle task. It requires you to provide the `movieFunUrl` project property. For example:

```
cd ~/workspace/assignment-submission
./gradlew submitReplatformingBackgroundJobsWithAmqp -PmovieFunUrl=http
://my-movie-fun.cfapps.io
```

(https://pivotal.io)

course version: 1.5.3