Look-aside Cache Pattern Example

This repo contains provides an example application demonstrating the use of Tanzu GemFire as a look-aside cache. With whatever your chosen IDE tool or editor is you will import an existing gradle project. Follow similar steps as Lab 1 to import the project if you're using an IDE.

The application uses Spring Boot for Apache Geode to cache data from the Bikewise.org public REST API. Look-aside caching is enabled with just a few annotations. When serving cached data, the application response time is dramatically improved.

How to get the app running on the local Env.

We are going to work now with an application that will use the look-aside features of gemfire caching. A standard spring-mvc app employs the use of stereotypes. These include:

- Domain Model
- Java DSL for Configuration
- Services
- View Controllers
- Spring MVC

The web MVC framework we are using is Thyme Leaf but will not be covered in this workshop. We will only look at the java code involved in the integration of Spring Boot Data Gemfire. Let's start with the domain.

Domain Object

Our domain is examining reports on incidents that effect bikes within a grouping nearby and associated with a zipcode. Our BikeIncident is structure to capture a description of an incident. Copy and paste this code into the *BikeIncident* class in the domain package.

```
private String type;
 private String title;
 private String description;
 private String address;

 public String getType() {
     return type;
 }

 public String getTitle() {
     return title;
 }

 public String getDescription() {
     return description;
 }

 public void setDescription(String description) {
     this.description = description;
 }

 public String getAddress() {
     return address;
 }
```

Java DSL for configuration

We are going to add a few configurations to enable Spring Boot's AutoConfiguration that we learned about in Lab 2. First we annotate our *LookAsideCacheApplicationConfig* class with the @Configuration so that Spring Boot will process our configurations. We then tell it to @EnableCachingDefinedRegions, which we will identify in our Service coming up next. We then annotate it with @EnableClusterAware to take advantage of SBDG's autoconfigfruation so that we can develop from our IDE with a near-cache, with a local cache and by the end of this lab you will see how to deploy and connect to your gemfire-cluster inside of kubernetes. Copy and paste this code into *LookAsideCacheApplicationConfig* class within the *config* package. [Note: we will use the @EnableStatics to show the difference in the speed of a cache access vs the time taken to invoke the restful API].

```
@Configuration
@EnableCachingDefinedRegions
@EnableStatistics
@EnableClusterAware
```

BikeIncidentService

Next we will add a service that takes the zipcode captured by our simple UI and calls out to a restful service to retrieve a report of bike incidents for a given zipcode and surrounding area.

```
    private final RestTemplate restTemplate;

    @Value("${bikewise.api.url}")
    private String API_URL;

    public BikeIncidentService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @Cacheable("BikeIncidentsByZip")
    public List<BikeIncident> getBikeIncidents(String zipCode) throws IOException {

        String jsonIncidents = restTemplate.getForObject(API_URL + zipCode, String.class);

        return convertJsonToBikeIncidents(jsonIncidents);
    }

    private List<BikeIncident> convertJsonToBikeIncidents(String jsonIncidents) throws IOException {
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);

        JsonNode bikeIncidentsAsJsonNode = objectMapper.readTree(jsonIncidents);

        List<BikeIncident> bikeIncidents = objectMapper.
               convertValue(bikeIncidentsAsJsonNode.get("incidents"),
                new TypeReference<List<BikeIncident>>(){});

        return bikeIncidents;
    }
```

The important annotation to notice in this service is the *@Cacheable* that will enable the capturing of a list of bike incidents into a *BikeIncidentsByZip* region that we'll be able to query from within Gemfire.

View Controller

Next let's implement our View Controller.

Add the following code to the class *ViewController* in the controllers package:

```
   @Autowired
    BikeIncidentService bikeIncidentService;

    private List<String> responseTimes = new ArrayList<>();

    @GetMapping("/")
    public String homePage(Model model) {
        model.addAttribute("zipCode");
        return "home";
    }

    @PostMapping("/")
    public String requestIncidents(
            @RequestParam String zipCode,
            Model model) throws IOException {

        long timeStampBeforeQuery = System.currentTimeMillis();
        List<BikeIncident> bikeIncidents =
                bikeIncidentService.getBikeIncidents(zipCode);
        long timeElapsed = System.currentTimeMillis() - timeStampBeforeQuery;
        recordNewDataRequest(timeElapsed, zipCode);
        populateModelWithSearchResults(model, bikeIncidents, zipCode);

        return "home";
    }

    private void recordNewDataRequest(long timeElapsed, String zipCode) {
        responseTimes.add( "Zip Code: " + zipCode + ", Response Time: " + timeElapsed + " ms");
    }

    private void populateModelWithSearchResults(Model model, List<BikeIncident> bikeIncidents,
            String zipCode) {
        model.addAttribute("zipCode", zipCode);
        model.addAttribute("responseTimes", responseTimes);
        model.addAttribute("bikeIncidents", bikeIncidents);
    }
```

There is nothing specific to Gemfire that is in this controller. We have autowired in the *BikeIncidentService* and invoke the service with the zipcode parameter that is input into our search page.

Testing your application functionality locally

You should now be able to run your BikeIncident app locally with a local cache. Simply start the spring boot application *LookAsideCacheApplication* from your IDE or commandline.

```
gfsh>run --file=configuration/src/main/resources/geode/bin/start-simple-cluster.gfsh
```

Now run some queries on zip codes and do the following query:

```
gfsh>query --query="SELECT address, description FROM /BikeIncidentsByZip"
```

Section Two - deploying our application to Kubernetes

The fist thing we need to do is change our default profile from "default" to kubernetes, allowing us to retrieve our gemfire-cluster locators and servers to connect to.

Open application.properties and uncomment the spring.profiles.active=kubernetes. This will pick up the additional properties we need for the gemfire cluster when we deploy our image.

If you are still in your IDE stop the spring boot process we just executed and exit gfsh in the following way:

```
gfsh>shutdown --include-locators=true
```

1. Build the image

Starting with Spring Boot 2.3, you can now run a comand and your build tool (maven or gradle) will build the Docker file for you. For this application if you run:

```
./gradlew bootBuildImage
```

This creates an image for you on your local docker daemon.

2. Load the image

We now need to load the image into the kind environment.

```
kind load docker-image docker.io/library/look-aside-cache:0.0.1-SNAPSHOT
```

3. Create a deployment

kubectl create deployment [my deployment name I make up] --image=[your look-aside-cache-image-name here]. - this should create you a deployment, replicaset, and pod using the image you build in step 1.

```
kubectl create deployment look-aside-cache --image=docker.io/library/look-aside-cache:0.0.1-SNAPSHOT
```

4. Expose the deployment

You now need to expose the deployment so that you can access your website: kubectl expose deployment/deployment-name --type="NodePort" --port 8080

```
kubectl expose deployment look-aside-cache --type="NodePort" --port 8080
```

You will now need the port exposed through the new service by doing the following:

```
kubectl get services
```

and the output should look like the following:

```
NAME               TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes         ClusterIP   10.96.0.1        <none>        443/TCP          4d22h
look-aside-cache   NodePort    10.108.118.222   <none>        8080:*30615*/TCP   17s
```

You will need your NodePort that is highlighed in bold.

5. Find the internal ip address

Once the deployment has been exposed you will need to find your internal ip address and then add NodePort you created in the previous step.

```
kubectl get nodes -o wide
```

and the output should look something similar to:

```
kubectl get nodes -o wide
NAME                 STATUS   ROLES    AGE     VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE       KERNEL-VI
kind-control-plane   Ready    master   4d22h   v1.18.2   172.18.0.2    <none>        Ubuntu 19.10   5.4.0-10
```