Design and Develop Service Tier

We will make use of the database tier created in the previous lab and implement our service tier to communicate with MySql database.

Learning Outcomes

After completing the lab, you will be able to:

1. Develop the service-tier consisting of a single microservice using TDD approach.
2. Implement Repository pattern
3. Deploy the service tier
4. Understand native K8s service discovery by using labels and selectors to discover database service deployed in the previous lab.

Before starting the lab, checkout the `service-tier-start` task.

```
   git status
   # Ensure the source code is checked in to github
   # You can take a back up of your codebase, to keep your depl
oyment files from being overwritten
   # Checkout into a feature branch
   git checkout service-tier-start -b service
   # You are on branch service
```

In case you get an error when you cherry-pick/check-out, open `intellij`, right-click on the `project`, select `git → resolve-conflicts → View changes and merge them based on the differences`

Develop the pages microservice

1. Few test classes were added to the test package. `IPagesRepository` interface is provided to implement the repository pattern. Observe these changes in intellij. The source code will not compile at this stage.

2. The first step is to get all the test cases passing.
3. Create a class `src/org/dell/kube/pages/Page` with the below fields

   ```
     public Long id;
     public String businessName;
     public Long categoryId;
     public String address;
     public String contactNumber;
   ```

   Generate getters,setters & constructor/s as expected by the test classes
4. Create a repository class `MySqlPageRepository` which implements `IPageRepository` as expected by the test class `MySqlPageRepositoryTest` . Annotate the class with `@Repository`
5. Create `PageController` class and implement methods as expected by the test class `PageApiTest` . Inject the dependency of type `MySqlPageRepository` in the constructor of the controller class.

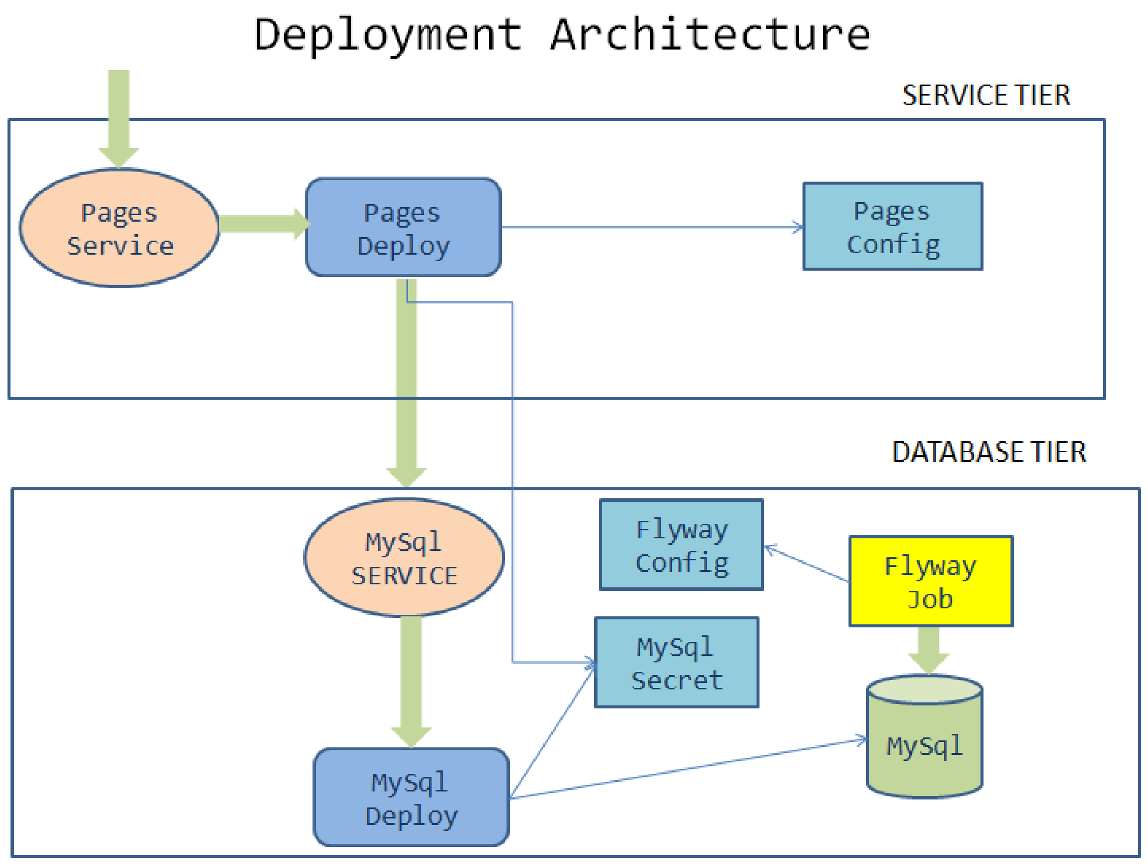
Building locally

1. Ensure that you have created the database pages locally, which was done in the previous lab. Otherwise follow the previous lab instructions to create the database and run the flyway migrations for the pages table to be created.
2. Provide the spring datasource url in `application.properties` for both development and testing.

   ```
   spring.datasource.url=jdbc:mysql://localhost:3306/pages?useSSL=false
   spring.datasource.username=pages_user
   spring.datasource.password=password
   ```

   We are using the same database for both development and testing for demonstration purposes. However, we recommend using separate database for development, testing and production in real projects.
3. Ensure that your application builds and all the test cases pass.

Connecting service tier with database tier



Kubernetize

1. The pages service needs to connect to the mysql database. The communication happens through `mysql service` that we created in the previous lab. However, the pages deployment should be aware of the `spring datasource` related properties.

2. Update the pages deployment by adding environment variables to the container

```
 SPRING_DATASOURCE_URL   ->  jdbc:mysql://pages-mysql/page
s
 SPRING_DATASOURCE_USERNAME  -> "root"
 SPRING_DATASOURCE_PASSWORD  -> value from secret name mys
ql-pass with key name password. This secret is already cre
ated in the previous lab.
```

With the above environment variables, spring boot can auto-configure the things necessary for database connectivity.

The updated env section looks like the below snippet.

```
        env:
          - name: PAGE_CONTENT
            valueFrom:
              configMapKeyRef:
                name: pages-config-map
                key: PAGE_CONTENT
          - name: SPRING_DATASOURCE_URL
            value: jdbc:mysql://pages-mysql/pages
          - name: SPRING_DATASOURCE_USERNAME
            value: "root"
          - name: SPRING_DATASOURCE_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mysql-pass
                key: password
```

3. Delete the pages deployment. This is not recommended in production, however we want to ensure that we have a fresh deployment for local testing. `kubectl delete deploy pages`
4. Build the docker image & push it to docker hub.

   `docker build -t [docker-username]/pages:service`

   `docker push [docker-username]/pages:service`
5. Update the pages deployment with the appropriate namespace, image & tag name.
6. Add label &/or selector `tier=service` (in addition to app=pages) in pages deployment yaml file at 3 places:

   `metadata.labels`

   `spec.selector.matchLabels`

   `spec.template.metadata.labels`
7. Add label &/or selector `tier=service` (in addition to app=pages) in pages service yaml file at 2 places:

   `metadata.labels`

   `spec.selector`
8. Adding the labels as above not only allows the `pages service` to serve the incoming requests but also helps the `pages-service` to discover `pages-mysql` service using native K8s support, without adding any additional code. This enables the `pages deployment` to communicate with `mysql deployment`

Testing locally on minikube

1. Switch the kubectl context to minikube - `kubectl config use-context minikube`

   Set the kubectl context namespace to your namespace - `kubectl config set-context --current --namespace [student-name]`

2. Since the pages service is updated with new labels, updating the immutable propery of the service is not allowed. Delete the service

   ```
   kubectl delete svc pages
   ```

3. Create the service

   ```
   kubectl apply -f deployment/pages-service.yaml
   ```

4. Create pages deployment.

   ```
   kubectl apply -f deployment/pages-deployment.yaml
   ```

5. Test the pages application by performing CRUD operations using curl/postman. Refer Pages Curl Guide for testing.
6. Update the docker image tag in `pipeline.yaml` and `pages-deployment.yaml` and verify they are same

Deploy to the production cluster

1. Ensure that the tag names are the same in deployment and pipeline.
2. In the `pipeline.yml` file update the branch to pick up the commit from the feature branch. Replace it with the branch name `service`
3. Commit your changes and push them to github. The pipeline will deploy your new image to the production cluster.
4. Pushing the souce code to github

   ```
   git status
   git add .
   git commit -m "service-tier"
   git push -u origin service
   ```

5. Ci/CD pipeline will trigger the build and deployment. Wait for it to succeed.
6. Switch the `kubectl` context to `production cluster` pointing to your namespace
7. Port forward to connect to pages service running inside K8s from the local machine

   ```
   kubectl port-forward svc/pages 8080:8080
   ```

8. Test the pages application by performing CRUD operations using curl/postman. Refer Pages Curl Guide for testing.

Task Accomplished

We completed integrating the service-tier with the database-tier & successfully deployed a two-tier cloud native application to K8s cluster.