Tutorial 7 – Web Application With Spring Boot (1e)

❖ Contents:

- Build data filter with multiple fields (using Criteria API) and pagination for web app with Spring Boot, Thymeleaf & JavaScript.

❖ Part 1 - Filtering with multiple fields:

1. Download & import the provided solution project of **Tutorial 5** to continue coding.

2. In your `repository` package, create a class named `EmployeeDao` and annotate it with the `@Repository` annotation. We need an `EntityManager` instance so we're auto-wiring it. Then create a method to execute a custom query. This method receives the request parameters (company, sort, gender). We'll add the drop-down menu for selecting genders later on the web UI.

```
@Repository
public class EmployeeDao {
    @Autowired
    EntityManager em;
    @Autowired
    CompanyRepository companyRepository;

    // data access methods go here
    public List<Employee> applyFilters(Long comId, int gender, int sortMode) {
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
        Root<Employee> employee = cq.from(Employee.class);
        // determining sort column
        Path<Object> sortColumn = employee.get("id"); // default sort column
        if (sortMode == 2 || sortMode == 3) {
            sortColumn = employee.get("name");
        }
        // determining sort order
        Order sortOrder = cb.desc(sortColumn); // default sort order
        if (sortMode == 1 || sortMode == 2) {
            sortOrder = cb.asc(sortColumn);
        }
        // filter criteria
        List<Predicate> predicates = new ArrayList<>();
        // filtering by company
        Optional<Company> comp = companyRepository.findById(comId);
        if (comp.isPresent()) {
            predicates.add(cb.equal(employee.get("company"), comp.get()));
        }
        // filtering by gender (0 means not filtered by gender)
        if (gender == 1) {
```

```
            predicates.add(cb.equal(employee.get("male"), false));
        } else if (gender == 2) {
            predicates.add(cb.equal(employee.get("male"), true));
        }
        // add a list of predicates as query criteria
        if (!predicates.isEmpty()) {
            cq.where(predicates.toArray(new Predicate[0]));
        }
        // sort
        cq.orderBy(sortOrder);
        return em.createQuery(cq).getResultList();
    }
}
```

Note that all the logics to determine sort order and sort column are moved here from the `getAllEmployee()` controller method. You could put all the logics in `EmployeeDao` in `EmployeeController`, but we're supposed to put data access logic in repository classes.

Therefore, the `getAllEmployee()` method will be changed into:

```
@RequestMapping(value = "/employee/list") // homepage redirects here
public String getAllEmployee(
        @RequestParam(value = "company", required = false, defaultValue = "0") Long
comId,
        @RequestParam(value = "gender", required = false, defaultValue = "0") int
gender,
        @RequestParam(value = "sort", required = false, defaultValue = "0") int
sortMode,
        Model model) {
    model.addAttribute("comId", comId);
    model.addAttribute("gender", gender);
    model.addAttribute("sortMode", sortMode);
    List<Employee> employees = employeeDao.applyFilters(comId, gender, sortMode);
    model.addAttribute("employees", employees);
    model.addAttribute("companies", companyRepository.findAll());
    return "employeeList";
}
```

Please note the addition of the `gender` query parameter.

3. In the `employeeList.html` template, add a drop-down menu for the gender filter:

```
Gender:
<select id="filterGender">
    <option value="0">All genders</option>
    <option th:value="1" th:text="Female"/>
    <option th:value="2" th:text="Male"/>
</select>
```

Note that this menu has 3 options, one default option (`value="0"`) means “not filtered by gender”. If you look back at the `applyFilters()` method in `EmployeeDao`, you shall notice that there is a mapping of: `0 = not filtered, 1 = female, 2 = male`.

As we’ve passed the `gender` variable to the template model in the controller method, we can display its value inside the JS code of the template so that we have current the gender value in the JS code:

```
<title>Employee List</title>
<script>
    window.addEventListener("load", function () {
        let comId = [[${comId}]];
        let gender = [[${gender}]]; // add this variable
        let sortMode = [[${sortMode}]];
```

Next, add the code to handle the `change` event of the `filterGender` menu:

```
const genderFilter = document.getElementById("filterGender");
genderFilter.value = gender;
genderFilter.addEventListener("change", function () {
    gender = genderFilter.value;
    filterRedirect();
});
```

Finally, modify the `applyFilter()` function to include the `gender` parameter:

```
function filterRedirect() {
    let url = "/employee/list?company=" + comId + "&gender=" + gender + "&sort=" + sortMode;
    window.location.href = url; // redirect
}
```

❖ Part 2 - Pagination:

1. Add a `page` parameter to the `getAllEmployee()` controller method and define a variable to specify the page size:

```
public String getAllEmployee(
        @RequestParam(value = "company", required = false, defaultValue = "0") Long comId,
        @RequestParam(value = "gender", required = false, defaultValue = "0") int gender,
        @RequestParam(value = "sort", required = false, defaultValue = "0") int sortMode,
        @RequestParam(value = "page", required = false, defaultValue = "0") int page,
        Model model) {
    final int pageSize = 1;
    // ...
}
```

(** Note that you can always change the value of the `pageSize` variable in this method to make your page bigger or smaller)

2. Rename the `applyFilters()` method in the `EmployeeDao` class into `filterAndSortEmployees()` to better reflect the behavior of this method. Change the method's return type into `Page<Employee>` to return a page of the query result instead of the entire result. Add a `Pageable` parameter to the `filterAndSortEmployees()` method for specifying which page should be retrieved.

```
public Page<Employee> filterAndSortEmployees(
        Long comId,
        int gender,
        int sortMode,
        Pageable pageable) {
```

(*) Some explanation about using `Pageable`:

Database tables often contain a large number of rows (thousands to millions and more). The number of data that can be displayed in a single web page is limitted. For example, you can fit maybe 25, 50, 100 emails in an Inbox web page, but the total number of emails in your account might easily reach thousands. Therefore, we need to show data in multiple pages, thus the concept of pagination.

Your data is a long list, and the data you display in a particular page is a sublist of that list. In order to get the list of results for a certain page, you need to provide the starting position (*offset*) and the ending position (derived from the number of results you want to display in a page, or the *page size*). To make it simple, Hibernate helps you calculate pagination information using `Pageable` and `PageRequest`:

```
Pageable firstPageWithTwoElements = PageRequest.of(0, 2);
Pageable secondPageWithFiveElements = PageRequest.of(1, 5);
```

3. In your `getAllEmployee()` controller method, create a `Pageable` object from the `page` and `pageSize` values and pass it into the `filterAndSortEmployees()` method like so:

```
Page<Employee> employees = employeeDao.filterAndSortEmployees(
        comId, gender, sortMode,
        PageRequest.of(page, pageSize)
);
```

Now, the `filterAndSortEmployees()` method knows which page of the result you want to get. Its job is to fetch and return the correct page of result from database. To get the *offset* value, we can use `pageable.getOffset()` and to get *page size* we can use `pageable.getPageSize()`.

4. Modify method to use `HibernateCriteriaBuilder` and `JpaCriteriaQuery` instead of `CriteriaBuilder` and `CriteriaQuery`. The reason for this is that `JpaCriteriaQuery` has the `createCountQuery()` method to create a query that counts the number of results of the original query. This is useful because when performing pagination, we need to know the *total number of records* in order to determine the *total number of pages*.

```
HibernateCriteriaBuilder cb =
        em.unwrap(Session.class).getCriteriaBuilder();
JpaCriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
```

5. Before returning the query result, use the following code to apply the `Pageable` object to the query result. The offset and the page size are similar to the two parameters of the `LIMIT` clause in a MySQL/MariaDB query (e.g. `LIMIT 5, 6` means offset is `5` and page size is `6`, or "*get 6 rows after skipping the first 5 rows*"). In the following code, please notice the place where we have to provide a function for determining the total row count of the entire result. We use a Lambda expression to express this function:

```
// sort
cq.orderBy(sortOrder);
// finalize & return results
TypedQuery<Employee> tq = em.createQuery(cq);
tq.setFirstResult((int) pageable.getOffset());
tq.setMaxResults(pageable.getPageSize());
return PageableExecutionUtils.getPage(
        tq.getResultList(),
        pageable,
        () -> em.createQuery(cq.createCountQuery())
                .getSingleResult() // counting
);
```

6. Don't forget to modify the `getAllEmployee()` method to use the new `filterAndSortEmployees()` method to get a `Page` of employees. Also pass the current `page` and the total number of `pages` into the template:

```
Page<Employee> employees = employeeDao.filterAndSortEmployees(
        comId, gender, sortMode,
        PageRequest.of(page, pageSize)
);
model.addAttribute("page", page);
model.addAttribute("pages", employees.getTotalPages());
model.addAttribute("employees", employees.get());
```

7. Last but not least, update the `employeeList.html` template to integrate the page parameter into the UI and show a pagination bar. First, add a page variable into the JavaScript code:

```
let comId = [[${comId}]];
let gender = [[${gender}]];
let sortMode = [[${sortMode}]];
let page = [[${page}]];
```

Secondly, edit the `filterRedirect()` function to include the `page` parameter:

```
function filterRedirect() {
    let url = "/employee/list?company=" + comId + "&gender=" + gender;
    url = url + "&sort=" + sortMode + "&page=" + page;
    window.location.href = url; // redirect
}
```

Thirdly, add the pagination links:

```
<div class="pagination" id="pagibar">
    <a th:data="${i}"
       th:each="i: ${#numbers.sequence(0, pages - 1)}"
       href="#"
       th:text="${i + 1}" />
</div>
```

Finally, add JS code to handle the `click` event on those pagination links:

```
const pageLinks = document.querySelectorAll("#pagibar > a");
pageLinks.forEach(link => {
    link.addEventListener("click", (e) => {
        e.preventDefault();
        page = parseInt(link.getAttribute("data"));
        filterRedirect();
    });
});
```

❖ Run, debug & submmit your project

Once finished, compress your project into a `.zip` file and submit the file to the tutorial's submission box.