ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Лекция 11

Пла

н

- Редактирование работника
- Удаление работника
- Spring MVC + Hibernate + AOP
- REST API

Редактирование работника

Для редактировании работника нам необходим тот же список полей, что и при создании. Можно создать новую форму и новый метод, но это не слишком удачное решение. Например, если мы заходим добавить поле, нам потребуется менять две формы, а не одну.

Поэтому лучшим вариантом будет повторно использовать view employee-info и метод save.

Редактирование работника

Но сначала добавим кнопки для редактирования работников.

```
<tr>
    <th>Name</th>
    <th>Surname</th>
    <th>Department</th>
    <th>Salary</th>
    <td>Operations</td>
</tr>
```

```
<c:forEach var="employee" items="${allEmployees}">

    <c:url var="updateButton" value="/updateInfo">
        <c:param name="empId" value="${employee.id}"/>
    </c:url>

    <tr>
        <td>${employee.name}</td>
        <td>${employee.surname}</td>
        <td>${employee.department}</td>
        <td>${employee.salary}</td>
        <td>
            <button type="button"
                    onclick="window.location.href = '${updateButton}'">Update</button>
        </td>
    </tr>
</c:forEach>
```

Редактирование работника

Поскольку мы будем работать с id работника, нужно добавить метод, который вернет нам работника из базы по id.

DAO:

```java
public interface EmployeeDAO {
    public List<Employee> getAllEmployees();

    public void saveEmployee(Employee employee);

    public Employee getEmployee(int id);
}
```

```java
@Repository
public class EmployeeDAOImplementation implements EmployeeDAO {
    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public List<Employee> getAllEmployees() {...}

    @Override
    public void saveEmployee(Employee employee) {...}

    @Override
    public Employee getEmployee(int id) {
        Session session = sessionFactory.getCurrentSession();

        return session.get(Employee.class, id);
    }
}
```

Редактирование работника

Service:

```java
public interface EmployeeService {
    public List<Employee> getAllEmployees();

    public void saveEmployee(Employee employee);

    public Employee getEmployee(int id);
}
```

```java
@Service
public class EmployeeServiceImplementation  implements EmployeeService{

    @Autowired
    private EmployeeDAO employeeDAO;

    @Override
    @Transactional
    public List<Employee> getAllEmployees() { return employeeDAO.getAllEmployees(); }

    @Override
    @Transactional
    public void saveEmployee(Employee employee) { employeeDAO.saveEmployee(employee); }

    @Override
    @Transactional
    public Employee getEmployee(int id) { return employeeDAO.getEmployee(id); }
}
```

:

Редактирование работника

Метод

```
@RequestMapping("/updateInfo")
public String updateEmployee(@RequestParam("empId") int id, Model model){

    Employee employee = employeeService.getEmployee(id);
    model.addAttribute( attributeName: "employee", employee);

    return "employee-info";
}
```

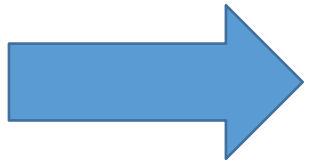
Для сравнения метод отображения формы создания нового работника:

```
@RequestMapping("/addEmployee")
public String addEmployee(Model model) {

    Employee employee = new Employee();
    model.addAttribute( attributeName: "employee", employee);

    return "employee-info";
}
```

Редактирование работника

Но это еще не все. На данном этапе форма не содержит данных об id работника

```
<body>
<form:form action="saveEmployee" modelAttribute="employee">
    Name <form:input path="name" />
    <br><br>
    Surname <form:input path="surname" />
    <br><br>
    Salary <form:input path="salary" />
    <br><br>
    Department <form:input path="department" />
    <br><br>
    <button>OK</button>
</form:form>
</body>
```



```
<body>
<form:form action="saveEmployee" modelAttribute="employee">
    <form:hidden path="id" />
    Name <form:input path="name" />
    <br><br>
    Surname <form:input path="surname" />
    <br><br>
    Salary <form:input path="salary" />
    <br><br>
    Department <form:input path="department" />
    <br><br>
    <button>OK</button>
</form:form>
</body>
```

Редактирование работника

Не стоит забывать, что с помощью той же формы мы создаем нового работника. Что будет у него в поле id?

Id этого работника будет 0 (значение int по умолчанию).

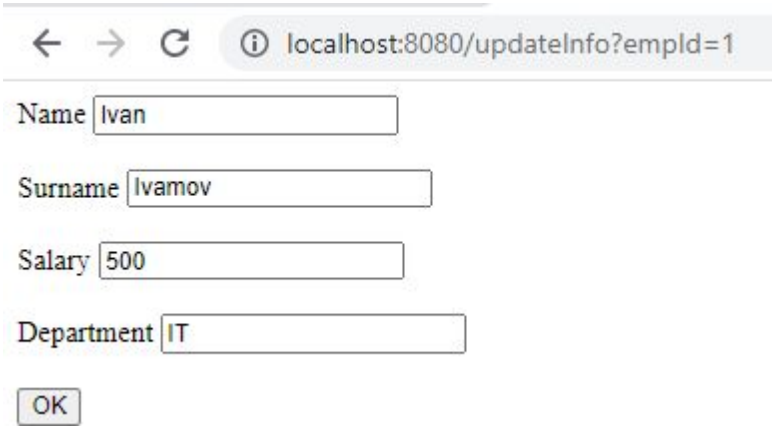
Редактирование работника

Также необходимо изменить метод в DAO. Поскольку метод save только добавляет новых работников в базу, нам необходим метод update. Одним из вариантов будет написание if, но лучше использовать возможности hibernate и применить метод **saveOrUpdate**:

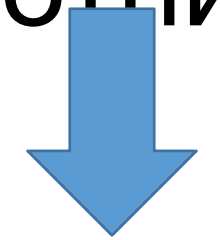
```
@Override
public void saveEmployee(Employee employee) {
    Session session = sessionFactory.getCurrentSession();
    session.saveOrUpdate(employee);
}
```

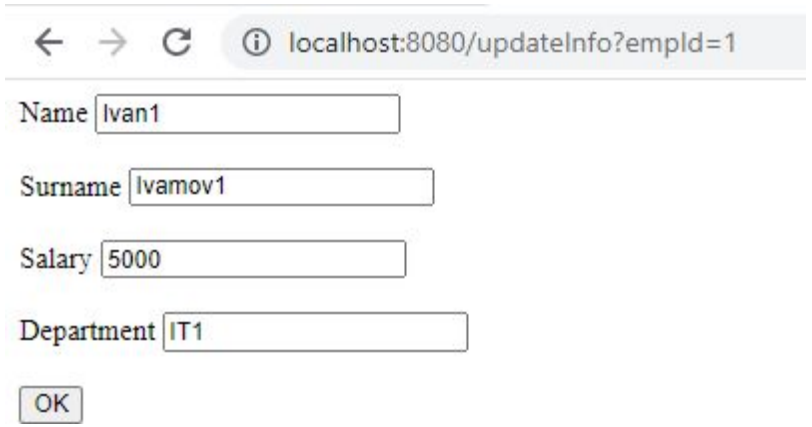
Редактирование работника

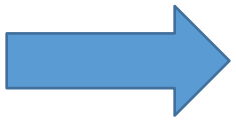
Теперь можно запускать:



Редактирование работника

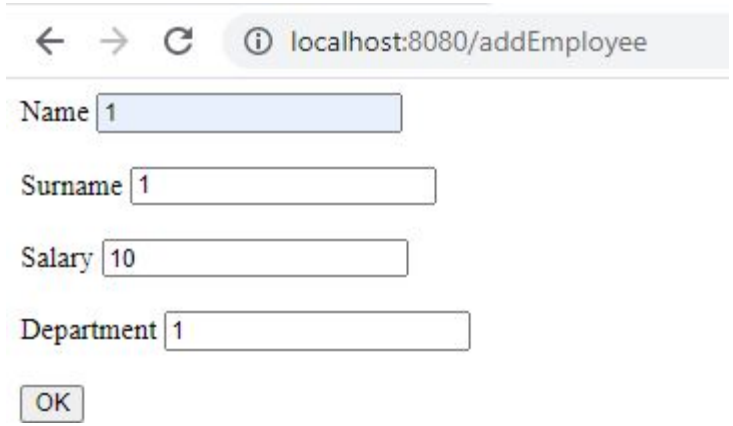






Редактирование работника

Также проверим добавление:



Удаление работника

Создадим кнопку

```
<c:forEach var="employee" items="${allEmployees}">

    <c:url var="updateButton" value="/updateInfo">
        <c:param name="empId" value="${employee.id}"/>
    </c:url>

    <c:url var="deleteButton" value="/deleteEmployee">
        <c:param name="empId" value="${employee.id}"/>
    </c:url>

    <tr>
        <td>${employee.name}</td>
        <td>${employee.surname}</td>
        <td>${employee.department}</td>
        <td>${employee.salary}</td>
        <td>
            <button type="button"
                    onclick="window.location.href = '${updateButton}'">Update</button>
            <button type="button"
                    onclick="window.location.href = '${deleteButton}'">Delete</button>
        </td>
    </tr>
</c:forEach>
```

Удаление работника

Метод в контроллере будет называться

```
@RequestMapping("/deleteEmployee")
public String deleteEmployee(@RequestParam("empId") int id){
    return "redirect:/";
}
```

После выполнения удаления нам необходимо вернуть список работников, поэтому возвращаем redirect на функцию, которая в свою очередь возвращает страницу со списком.

Удаление работника

Для выполнения удаления необходимо добавить метод удаляющий работника из базы.

DAO:

```
public interface EmployeeDAO {
    public List<Employee> getAllEmployees();

    public void saveEmployee(Employee employee);

    public Employee getEmployee(int id);

    public void deleteEmployee(int id);
}
```

```
@Override
public void deleteEmployee(int id) {
    Session session = sessionFactory.getCurrentSession();
    Query<Employee> query = session.createQuery( s: "delete from Employee " +
            "where id =:employeeId");
    query.setParameter( s: "employeeId", id);
    query.executeUpdate();
}
```

Удаление работника

Service:

```java
public interface EmployeeService {
    public List<Employee> getAllEmployees();

    public void saveEmployee(Employee employee);

    public Employee getEmployee(int id);

    public void deleteEmployee(int id);
}
```

```java
@Service
public class EmployeeServiceImplementation  implements EmployeeService{

    @Autowired
    private EmployeeDAO employeeDAO;

    @Override
    @Transactional
    public List<Employee> getAllEmployees() { return employeeDAO.getAllEmployees(); }

    @Override
    @Transactional
    public void saveEmployee(Employee employee) { employeeDAO.saveEmployee(employee); }

    @Override
    @Transactional
    public Employee getEmployee(int id) { return employeeDAO.getEmployee(id); }

    @Override
    @Transactional
    public void deleteEmployee(int id) { employeeDAO.deleteEmployee(id); }
}
```

Удаление работника

Теперь можно вызвать метод сервиса в

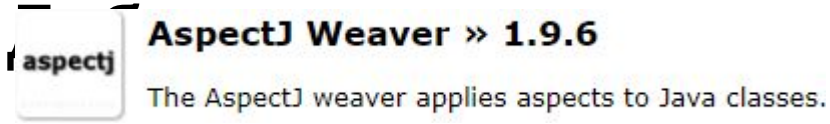
```
@RequestMapping("/deleteEmployee")
public String deleteEmployee(@RequestParam("empId") int id){

    employeeService.deleteEmployee(id);

    return "redirect:/";
}
```

Удаление работника

Результат.

Spring MVC + Hibernate + AOP

Прежде всего необходимо выполнить конфигурацию AOP.



pom:

```
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.9.6</version>
</dependency>
```


```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.6</version>
</dependency>
```


Spring MVC + Hibernate + AOP

Далее добавим конфигурацию в

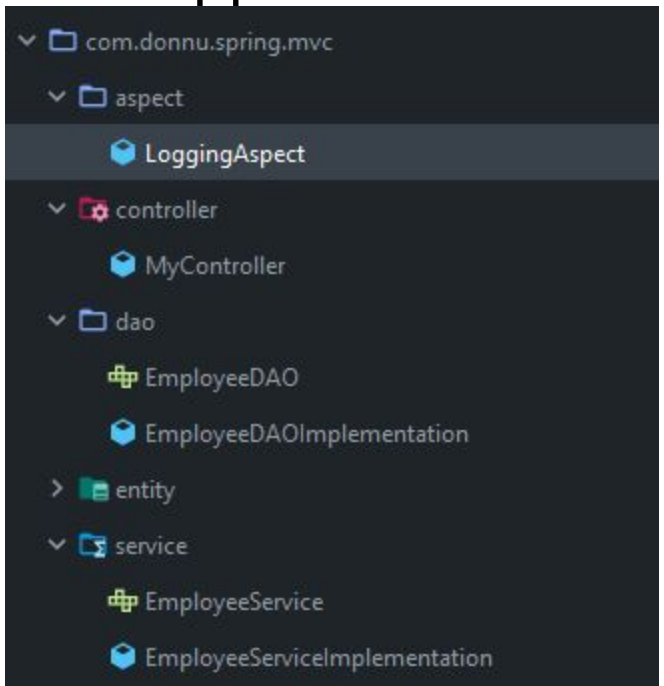
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
```

```
<context:component-scan base-package="com.donnu.spring.mvc" />

<mvc:annotation-driven/>

<aop:aspectj-autoproxy/>
```

```
 xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">
```

Spring MVC + Hibernate + AOP

После конфигурации можно переходить к созданию



.

```java
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LoggingAspect {

    @Around("execution(* com.donnu.spring.mvc.dao.*.*(..))")
    public Object aroundAllRepositoryMethodsAdvice(
            ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        MethodSignature methodSignature =
                (MethodSignature) proceedingJoinPoint.getSignature();
        String methodName = methodSignature.getName();

        System.out.println("Begin of " + methodName);
        Object targetMethodResult = proceedingJoinPoint.proceed();
        System.out.println("End of " + methodName);

        return targetMethodResult;
    }
}
```

Spring MVC + Hibernate + AOP

Запускае


```
Output
Begin of getAllEmployees
Hibernate: select employee0_.id as id1_0_,
End of getAllEmployees
```

Spring MVC + Hibernate + AOP

Добавляем работника:


```
Begin of saveEmployee
Hibernate: insert into employees (department, name, salary, surname) values (?, ?, ?, ?)
End of saveEmployee
Begin of getAllEmployees
Hibernate: select employee0_.id as id1_0_, employee0_.department as departme2_0_, employee0_.name
End of getAllEmployees
```

REST API

REST API — это прикладной программный интерфейс (API), который использует HTTP-запросы для получения, извлечения, размещения и удаления данных. Аббревиатура REST в контексте API расшифровывается как «передача состояния представления» (Representational State Transfer).

REST API

Начнем с того, что такое API. Для веб-сайта это код, который позволяет двум программам взаимодействовать друг с другом. API предлагает разработчикам правильный способ написания программы, запрашивающей услуги у операционной системы или другого приложения. Проще говоря, это своего рода стандарт, который позволяет программам и приложениям понимать друг друга; это язык интернета, который необходим для работы практически всех сайтов и приложений.

REST API

Еще одна распространенная сфера применения — облачные технологии, где **REST API** нужен для предоставления и организации доступа к веб-службам. Технология позволяет пользователям гибко подключаться к облачным сервисам, управлять ими и взаимодействовать в распределенной среде.

REST API

API REST (или **RESTful**) основан на передаче состояния представлений, архитектурном стиле и подходе к коммуникациям, часто используемым при разработке веб-служб. Некоторые веб-мастера рекомендуют использовать вместо этой технологии SOAP, которая считается более надежной. Но она проигрывает REST API по скорости (последняя использует меньшую пропускную способность, что делает ее более подходящей для эффективного использования интернета).

REST API

Принципы REST API определены в диссертации его создателя Роя Филдинга. Основные из них:

- единый интерфейс;
- разграничение клиента и сервера;
- нет сохранения состояния;
- кэширование всегда разрешено;
- многоуровневая система;
- код предоставляется по запросу.

REST API

Единый интерфейс

Ресурсы должны быть однозначно идентифицированы посредством одного URL-адреса и только с помощью базовых методов сетевого протокола (DELETE, PUT, GET, HTTP).

POST -- create

GET -- read

PUT -- update/replace

DELETE -- delete

PATCH -- partial update/modify

REST API

Клиент-сервер

Должно быть четкое разграничение между клиентом и сервером:

- пользовательский интерфейс и вопросы сбора запросов — на стороне клиента.
- доступ к данным, управление рабочей нагрузкой и безопасность — на стороне сервера.

REST API

Сохранение состояния

Все клиент-серверные операции должны быть без сохранения состояния. Любое необходимое управление состоянием должно осуществляться на клиенте, а не на сервере.

Кэширование

Все ресурсы должны разрешать кэширование, если явно не указано, что оно невозможно.

Многоуровневая система

REST API допускает архитектуру, которая состоит из нескольких уровней серверов.

REST API

Запрос кода

В большинстве случаев сервер отправляет обратно статические представления ресурсов в формате XML или JSON. Однако при необходимости серверы могут отправлять исполняемый код непосредственно клиенту.