well done, corrections in red

Review of Team4 by Team9

Design

- Violation of the MVC pattern

The project follows almost everywhere the MVC pattern, one thing that might be called a violation of this pattern is, that the ProfilePictureViewController is Autowired to a Dao and addresses therefore the DB directly.
Another violation is the ExceptionServiceImpl. As I understand, it does the job of a controller (no DB access, adds an Object to a model) and the controller that passes a model to it could add the message by itself. Same thing with addProfilePictureInfoToModel in the ProfilePictureServiceImpl.

- Usage of helper objects between view and model

It is quite nice that almost for every that "does something" a separate form (with well defiened constraints) is used. And that, with the various included jsps, the responsibility of the page restricts itself to handling those forms.

- Rich OO domain model

Not much to say here, the domain model seems to be just what is needed and to provide good access to all fields.

- Clear responsibilities

The responsibilities are very clear defined. For example it is nice, that the responsibilities concerning the profile are split up into show and edit informations, and additionaly into the more complex show-image and edit-image responsibilities.

- Sound invariants

No invariants spotted. But many checks and custom exception seem to garantee valid states.

- Overall code organization & reuse, e.g. views

For each service, an interface is defined but none is implemented more than once. This seems to be unnecessary, but follows the proposed style from the skeleton project. In the views, you couldn't reuse your code more. This makes them very clean and easy to read.

Coding Style

- Consistency
  The codebase is consistent. Similar classes are grouped into corresponding packages. Class and method naming is consistent and according to java coding style with camelCase, lowercase starting names in methods etc...

- Intention-revealing names
  Method names in the controller classes are very intention revealing, methods starting with 'display' create a view which simply displays a page most often after a get request of the user, 'update' updates the state after post requests. Exceptions also reveal in what situations they might be thrown, e.g. UserAlreadyExistsException, NoResultFoundException.. Tests are named intention revealing and separated into unit tests or integration tests.
- Do not repeat yourself
  Every service implements a unique interface. In this case I think that the interface might have been left out because every service has it's own interface, so for every service class an extra interface is created which doesn't add any functionality. Regarding jsp pages includes were used to factor out common parts like the header and footer which are the same for many pages.
- Exceptions, testing null values
  Team4 has extensively used Error classes with intention revealing names in appropriate places and documented in which cases they are thrown. is that good or bad? would normal exceptions be better?
- Encapsulation
  Encapsulation was used according to the Spring framework. Lecture, Message, User encapsulate the data which corresponds to the model for the real world. Different form classes are used for encapsulating the data which a user can put in by the view. Behaviour for the interaction of view and model is done properly in Controller classes like intended by the framework.
- Assertions, contracts, invariant checks
  Assertions most often assert whether the input paramters are not null. Contracts are clearly defined in method names and state whether a parameter mustn't be null or what exception can be thrown and in which case a method gets thrown. Invariants haven't been used but I also did not see a particular case in which this would have been necessary.
- Utility methods
  Only utility class used was the MultipartFileMocker class which was created in the ch.ututor.utils package and is a helper class for mocking images for tests.

Documentation

- Understandable
  Documentation is understandable for example considering following exerpts:

```
/**
     * @return  If the user's already tutor, the method returns a ModelAndView
     *          with an AddLectureForm to add a new lecture. Otherwise a
ModelAndView
     *          to become tutor is returned.
     */
    @RequestMapping( value = {"/user/add-lecture"}, method = RequestMethod.GET )
    public ModelAndView displayAddLectureForm() {
```

```
   /**
     *  Updates the profile data based on the information entered in the
profileEditForm
```

```
     *
     *  @return ModelAndView of the own profile if the update has succeeded.
     *          Otherwise a ModelAndView with a profile edit form containing error
or exception messages.
     */
    @RequestMapping( value = {"/user/profile/edit"}, method = RequestMethod.POST )
    public ModelAndView updateProfileData(  @Valid ProfileEditForm profileEditForm,
                                            BindingResult result,
                                            RedirectAttributes redirectAttributes )
{
```

Existing documentation is concise and describes what what will be returned and in which case. * Intention-revealing The srs is well written and shows all major use cases.

- Describes responsibilities
  Class documentation is missing but responsibilities are described in the method documentation which exists for most important methods. Maybe in the case of the MessageCenterServiceImpl and ExceptionServiceImpl class documentation would have been helpful to understand the codebase a little bit faster. In all other cases documentation for methods was helpful. *Match a consistent domain vocabulary domain vocabulary?

Testing

- Clear and distinct test cases
  Test cases are both clear and distinct. Unit tests do not overlapp and it is clear what item is tested. Differentiation between unit and integration tests.
- Number/coverage of test cases
  All controller and services are tested except LoginController (which doesn't need to be tested). The classes are tested extensively including Exceptions. Very good code coverage overall.
- Easy to understand the case that is tested
  The names of the test methods are well choosen and describtive. A nice addition (but not necessary) would be a link to the tested class. Something like this:`/*`
  `* Test of class {@link IndexController}`
  `*/`
- Well crafted set of test data
  Use of a data seeder is a good method for setting up the test data. The data set is comprehensive and you have some nice test users :)
- Readability
  The tests are well formatted with a few missteps in line length (e.g. ProfileEditControllerTest). However, I don't see why you didn't use the standard file structure: `main`
  `java ...`
  `tests`
  `java`
  `ch`
  `ututor`
  `controller  <- controller tests; name the integration tests`

```
ControllerNameIT (or something like that)
                    service     <- service tests
```

I think that would be a better structure than the one you have now. You could still use test suits if you want to. Use assertArrayEquals instead assertEquals when comparing arrays for equality (best practise).

Example class from package controller

```
package ch.ututor.controller.service.implementation;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import ch.ututor.controller.exceptions.custom.NoResultFoundException;
import ch.ututor.controller.service.SearchService;
import ch.ututor.model.TutorLecture;
import ch.ututor.model.dao.TutorLectureDao;

@Service
public class SearchServiceImpl implements SearchService {

    @Autowired   private TutorLectureDao tutorLectureDao;

    /**
     *  @param query    mustn't be null
     *
     *  @throws         NoResultFoundException if no lectures are found for the
search term
     */
    public List searchByLecture( String query ) throws NoResultFoundException {
        assert ( query != null );

        List lectures = tutorLectureDao.findByLectureNameLikeOrderByLectureName(
'%' + query + '%' );

        if( lectures.size() == 0 ) {
            throw new NoResultFoundException( "No lectures found." );
        }

        return lectures;
    }
}
```

The SearchServiceImpl class has a well defined responsibility to find a list of lectures by a given search parameter. In this case the responsibilities are clear defined.

clear defined, but too many responsibilities? should logic be moved?