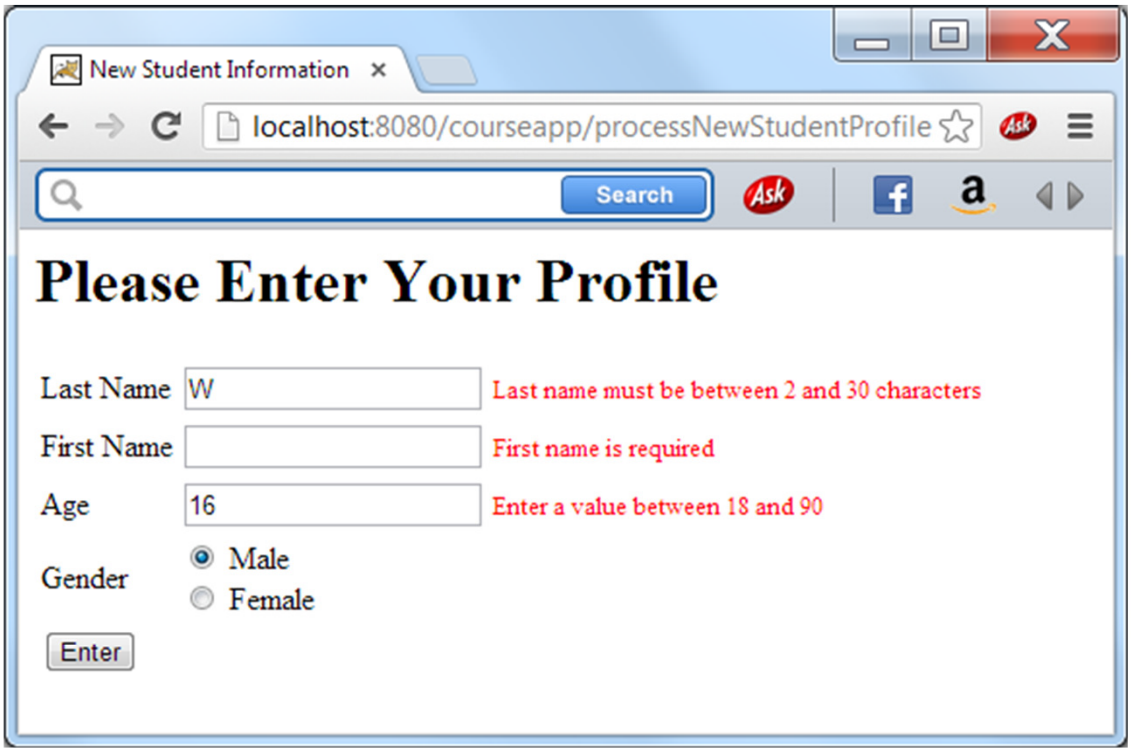
Form Validation

Traditional form processing can be a lot of work

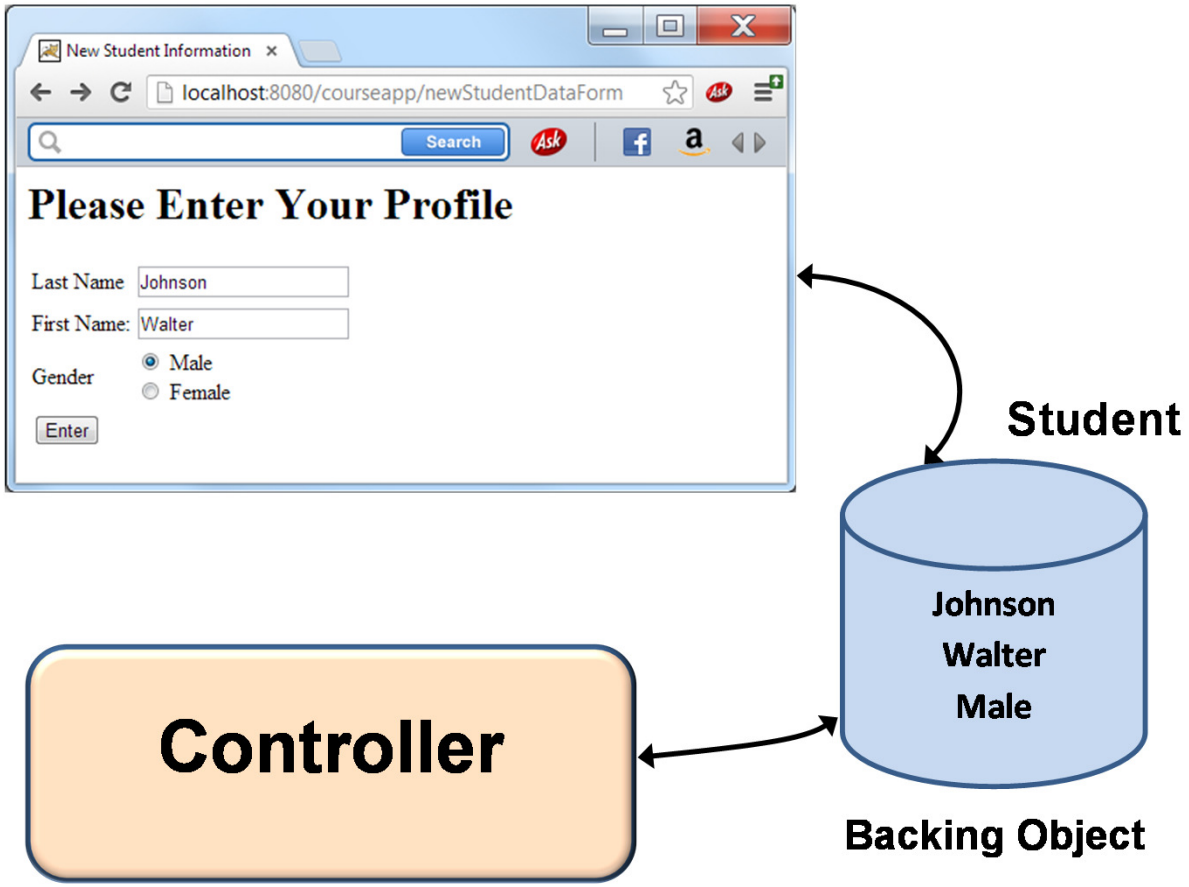
- The form processor must check each field and if there are errors display the form again with old values and error messages
- Often called representing the form



Backing Objects

Backing Object

- The object that stores the form data
- When the form is initially presented, it can retrieve the data from the backing object
- When the form is submitted, data is placed in the backing object
- When the form is re-presented, the data is retrieved from the backing object



Spring MVC: A better Form

Spring provides special .jsp tags for better form handling

- Instead of using the standard HTML form tags, use tags from the Spring MVC tag library

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

- commandName is the class of the Backing object
- path is the name of a field in the Backing object
- Spring produces a standard HTML form for you

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<form:form action="./processNewStudentProfile" method="POST"
           commandName="student">
  <table>
    <tr>
        <td><form:label path="lastName">Last Name</form:label></td>
       <td><form:input path="lastName" /></td>
    </tr>
    <tr>
       <td><form:label path="firstName">Last Name</form:label></td>
       <td><form:input path="firstName" />
       </td>
    </tr>
    </table>
  </form:form>
</body>
</html>
```

Spring MVC: A better Form

HTML form attributes that become optional

- action
  - Spring will use URL of the original GET request if this attribute is not present
- method
  - Default value is POST since that is how most forms are handled (if method attribute is not present)

Note that action and method are **not** present in form below:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<form:form commandName="student">

</form:form>
```

```
@RequestMapping(value = "/newStudentDataForm",
                method = RequestMethod.GET)
public ModelAndView presentStudentDataForm() {
  ...
  return modelView;
}

 @RequestMapping(value = "/newStudentDataForm",
                 method = RequestMethod.POST)
public ModelAndView processStudentDataForm() {
  ...
  return modelView;
}
```

Spring MVC: A better Form

- For Using MVC Controllers and Spring Forms

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <!-- Enables the Spring MVC @Controller programming model -->
  <mvc:annotation-driven />
  <mvc:resources mapping="/resources/**" location="/resources/" />

  <context:component-scan base-package="com.nvz.courseapp.controllers" />

</beans>
```

- Make sure to limit component scanning to only the controllers in the web context
  - Otherwise components will be duplicated

Using a Backing Object

- Create the Backing object and pass to the initial presentation page
  - Fields in the Backing object are matched to **path** names in the Spring form

```
@Controller
public class StudentController {
  @Autowired
  StudentService studentService;

  @RequestMapping(value = "/newStudentDataForm", method = RequestMethod.GET)
  public ModelAndView newStudentDataForm() {
    ModelAndView modelView;

    modelView = new ModelAndView("studentDataForm");
    /* Create the Backing object and pass for presentation */
    modelView.addObject("student", new Student());
    return modelView;
  }
}
```

Use the class name of your backing object as the model attribute name and the commandName

```
<form:form action="./processNewStudentProfile"
                                    method="POST" commandName="student">
```

Obtaining a BindingResult

- When a Spring form is submitted, Spring will trap exceptions and perform specified validations on the form data
  - Add **@Valid** in front of Backing Object
  - Errors are reported in the BindingResult
  - If errors have been found, you can re-present the form

```
@Controller
public class StudentController {
  @Autowired
  StudentService studentService;

  @RequestMapping(value = "/processNewStudentProfile", method = RequestMethod.POST)
  public ModelAndView processNewStudentForm(
     @Valid Student student, BindingResult result)
  {
  ModelAndView modelView;

  if (result.hasErrors()) {
    /* Do re-presentation with error messages */
    modelView = new ModelAndView("studentDataForm", "student", student);
    return modelView;
  }

  /* No validation errors, add new student profile */
  studentService.addNewStudent(student);
  modelView = new ModelAndView("newStudentProfileSuccess");
  return modelView;
  }
}
```

What About Error Messages?

- Use the Spring form **errors** tag so provide error messages
  - path is the name of the field in the Backing object being validated
  - cssClass is the name of the CSS class used to format the error message

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<form:form action="./processNewStudentProfile" method="POST" commandName="student">
  <table>
    <tr>
        <td><form:label path="lastName">Last Name</form:label></td>
       <td><form:input path="lastName" /></td>
           <form:errors path="firstName" cssClass="error"/>
    </tr>
    <tr>
       <td><form:label path="firstName">Last Name</form:label></td>
       <td><form:input path="firstName" />
            <form:errors path="firstName" cssClass="error"/>
       </td>
    </tr>
    </table>
  </form:form>
</body>
</html>
```

Trapping Conversion Errors

- If a non-numeric value is entered for the age, we get a conversion error
  - We are now trapping it instead of getting a server error
  - The error message is not very user friendly

- We'll use the Message Bundle approach to specify better messages

Message Bundle

- A .properties file containing text that is to be displayed
  - Placing text in the file means you can change the text without modifying Java code
  - Allows for internationalization (text automatically changed based on the Locale)

- Consists of key / value pairs
  - Message Label (key)
  - Message Text (value)

message.properties

```
title.addStudent=New Student Profile Form
student.firstName=First Name
student.lastName=Last Name
form.submit=Enter
```

Enabling Message Bundles

- Add the XML below so that Spring will create a bean called messageSource
  - You can then refer to messages found in the file

root-context.xml

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="messages"/>
</bean>
```

- Add the JSTL tag library **fmt** to your .jsp file
  - Use the message tag with the message key

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<h1><fmt:message key="studentDataForm.title" /></h1>
```

Externalizing Messages

```
studentDataForm.title=Please Enter Your Profile
firstnameLabel=First Name
lastnameLabel=Last Name
enterBtn=Enter
```

```
<h1><fmt:message key="studentDataForm.title" /></h1>

  <form:form action="./processNewStudentProfile"
             method="POST" commandName="newStudent">
  <table>
    <tr>
    <td><fmt:message key="lastnameLabel" /></td>
        <td><form:input path="lastName" />
            <form:errors path="lastName" cssClass="error"/>
        </td>
    </tr>
   <tr>
    <td><input type="submit" value="<fmt:message key="enterBtn" />">
    </td>
    </tr>
    </table>
  </form:form>
```

Internationalization

There can be different message files for different languages

The file is chosen based on the current Locale

messages.properties

```
studentDataForm.title=Please Enter Your Profile
firstnameLabel=First Name
lastnameLabel=Last Name
enterBtn=Enter
```

Each language has a corresponding _lang. For example, Spanish is _es

messages_es.properties

```
studentDataForm.title=Por favor, Introduzca Su Perfil
firstnameLabel=Nombre
lastnameLabel=Apellido
enterBtn=Entrar
```

A Better Conversion Error Message

- Now we can specify our own conversion error messages.
  - To trap an invalid integer, use key: typeMisMatch.int in the message file

messages.properties

```
studentDataForm.title=Please Enter Your Profile
firstnameLabel=First Name
lastnameLabel=Last Name

typeMismatch.int=Expecting an integer value
```

Declarative Validation

- JSR-303
  - Standards specification for validation of Java beans using annotations
  - Annotations are placed on domain classes
  - Hibernate Validator is an (extended) implementation of JSR 303
    - `javax.validation.constraints` (JSR 303)
    - `org.hibernate.validator.constraints` (extensions)
  - **@Valid** (in your controller) triggers validation

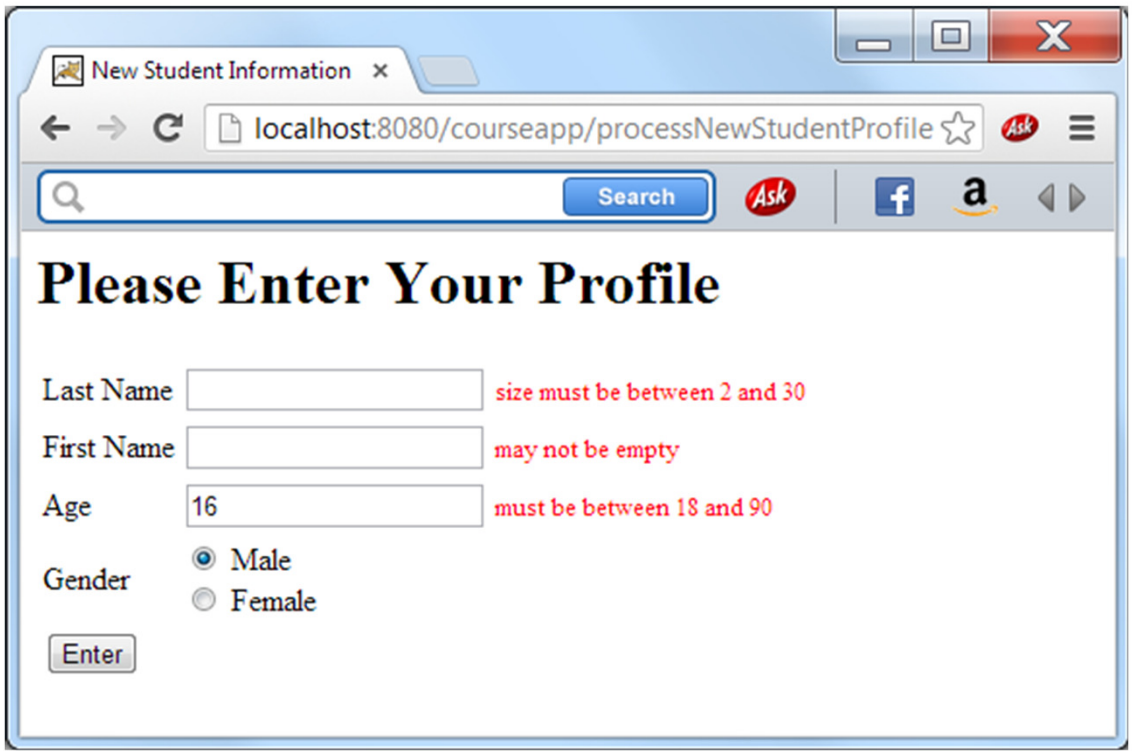
```
public class Student {
  private long id;
  @Size(min=2, max=30)
  private String lastName;
  @NotEmpty
  private String firstName;
  @Range(min=18, max=90)
  int age;
  private Gender gender;


}
```

Declarative Validation

```
public class Student {
  private long id;
  @Size(min=2, max=30)
  private String lastName;
  @NotEmpty
  private String firstName;
  @Range(min=18,max=90)
  int age;
  private Gender gender;


}
```



Declarative Validation

- Make sure you use **@Valid** in front of your Bean to invoke Bean validation
- The BindingResult parameter must come directly after your Bean parameter to work properly

```
@RequestMapping(value = "/processNewStudentProfile",
                method = RequestMethod.POST)
  public ModelAndView processNewStudentForm(
       @Valid Student student, BindingResult result)
  {
    ModelAndView modelView;

    if (result.hasErrors()) {
      modelView = new ModelAndView("studentDataForm",
                     "student", student);
      return modelView;
    }
    ...
}
```

Declarative Validation

- Partial List of Constraints
  - @Size
  - @NotEmpty
  - @NotNull
  - @Min
  - @Max
  - @Digits
  - @Range
  - @Email
  - @Pattern
  - @Past

- Error Message Keys Defined as:
  <ConstraintName>.<CommandName>.<FieldName>
  - Examples
    NotEmpty.student.firstName
    Size.student.lastName

Changing Default Error Messages

- Add your custom messages to messages.properties
  - You can use placeholders such as {0} to specify values found in the annotations

messages.properties

```
NotEmpty={0} is required.
NotEmpty.lastName=Last name must be provided
Size.lastName=Last name must be between {2} and {1} characters
Range=Enter a value between {2} and {1}
```

Programmatic Validation

- For non-standard validation requirements you can write your own logic with a Validator class
  - Can write a validator for a single type of field (such as phone number)
  - Can write a single validator to verify all fields of a class
- You can mix declarative validation and programmatic validation
- Validator interface
  - `public boolean supports(Class<?> arg)`
  - **public void** validate(Object tstObj, Errors errors)
- ValidatorUtils class
  - Utilities to check for empty or white space

Programmatic Validation

```
public class AccountNameValidator implements Validator {

  public boolean supports(Class<?> targetClass) {
    return String.class.equals(targetClass);
  }

  public void validate(Object tstObj, Errors errors) {
    String tstName = (String) tstObj;
    int nameLen;
    char tstChar, prevChar;
    int i;

     ValidationUtils.rejectIfEmptyOrWhitespace(errors,
            "acctName", "Required");

     tstChar = tstName.charAt(0);
     if (!Character.isLetter(tstChar)) {
        errors.rejectValue("acctName", "NonLetterStart");
        return;
     }
  }
}
```

- Call rejectValue() to add an error to the error list
  - Provide field name and constraint name
  - Message keys for above errors:
    Required.acctName
    NonLetterStart.acctName

Validating a Field

```
public ModelAndView storeNewStudentData(@Valid UserLoginInfo loginInfo,
                 BindingResult result, HttpSession session)
{
    String acctName, password;
    ModelAndView mv;
    Student student, existingStudent;
    AccountNameValidator acctNameValidator = new AccountNameValidator();

    student = (Student) session.getAttribute("student");

    acctName = loginInfo.getAcctName();
    existingStudent = studService.findStudentByAcctName(acctName);

    if (existingStudent != null) {
      result.rejectValue("acctName", "AcctNameExists");
    } else {
      acctNameValidator.validate(acctName, result);
    }

    if (result.hasErrors()) {
      mv = new ModelAndView("/studentNewAcctForm");
      mv.addObject("userLoginInfo", loginInfo);
      return mv;
    }

    mv = new ModelAndView("student/studentHome");

    acctName = loginInfo.getAcctName();
    password = loginInfo.getPassword();
    student.setAcctName(acctName);
    student.setPassword(password);
      ...
}
```

Validator for a Bean

- A Bean Validator can check multiple fields – in this case account name and password

```
public class AccountInfoValidator implements Validator {

  public boolean supports(Class<?> targetClass) {
    return targetClass.equals(UserLoginInfo.class);
  }

  public void validate(Object tstObj, Errors errors) {
    UserLoginInfo login = (UserLoginInfo) tstObj;
    String acctName, password;
    int nameLen;
    char tstChar, prevChar;
    int i;

    acctName = login.getAcctName();
    tstChar = acctName.charAt(0);
    if (!Character.isLetter(tstChar)) {
      errors.rejectValue("acctName", "NonLetterStart");
      return;
    }

    password = login.getPassword();
    if (password.length() > 20) {
      errors.rejectValue("password", "LengthExceeded");
      return;
    }
  }
}
```

Validating for a Bean

- Remember to use @Valid

```
public ModelAndView storeNewStudentData(@Valid UserLoginInfo loginInfo,
            BindingResult result, HttpSession session)
{
      String acctName, password;
      ModelAndView mv;
      Student student, existingStudent;
      AccountInfoValidator acctInfoValidator = new AccountInfoValidator();

      student = (Student) session.getAttribute("student");
      acctName = loginInfo.getAcctName();

      existingStudent = studService.findStudentByAcctName(acctName);
      if (existingStudent != null) {
         result.rejectValue("acctName", "AcctNameExists");
      } else {
         acctInfoValidator.validate(loginInfo, result);
      }

      if (result.hasErrors()) {
         mv = new ModelAndView("/studentNewAcctForm");
         mv.addObject("userLoginInfo", loginInfo);
         return mv;
      }

      mv = new ModelAndView("student/studentHome");

      acctName = loginInfo.getAcctName();
      password = loginInfo.getPassword();
      student.setAcctName(acctName);
      student.setPassword(password);
   ...
}
```

Interceptors

- Intercept http requests
  - Useful for security, logging, etc.
  - Can intercept prior to controller call, after the controller call, and/or after the view has been displayed

- Implement **HandlerInterceptor**
  - Can also extend **HandlerAdaptor**

```
public class AuthenticatedInterceptor implements HandlerInterceptor
{

  public void afterCompletion(HttpServletRequest request,
      HttpServletResponse response, Object handler, Exception arg3)
      throws Exception
  {

  }

  public void postHandle(HttpServletRequest request,
      HttpServletResponse response, Object handler,
      ModelAndView arg3) throws Exception
  {

  }

  public boolean preHandle(HttpServletRequest request,
              HttpServletResponse response,
      Object handler) throws Exception
  {
    return false;
  }
}
```

Example Interceptor

- Authentication Example
  - After successful login, a student object is placed in the Session
  - If student is not in the session, there has not been a successful login
  - Don't allow users to access URLs in the student directory without first logging in
    - Send user back to the home page

```
public class AuthenticationInterceptor extends HandlerInterceptorAdapter {

  public boolean preHandle(HttpServletRequest request,
          HttpServletResponse response, Object handler)
          throws Exception
  {
    String context = request.getContextPath();
    Student student;
    HttpSession session = request.getSession();

    student = (Student) session.getAttribute("student") ;
    if (student == null) {
      response.sendRedirect(context + "/home");
      return false;
    }

    return true;
  }
}
```

Registering Interceptors

- Register your interceptor in the web context
  - Specify the URL patterns that will be intercepted
  - Use one `<mvc:interceptor>` tag for each interceptor
    - Tags go in the `<mvc:interceptors>` list

```
<mvc:interceptors>
   <mvc:interceptor>
       <mapping path="/student/**"/>
       <bean
        class="com.nvz.courseapp.interceptors.AuthenticationInterceptor" />
    </mvc:interceptor>

    <mvc:interceptor>
       <mapping path="/**"/>
       <bean class="com.nvz.courseapp.interceptors.LoggingInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>
```

Example Interceptor II

- Authentication Example
  - An alternative approach would be to look at the URL being requested
  - If the URL is not a login/logout page look to see if the student is in the session

```
public class AuthenticationInterceptor extends HandlerInterceptorAdapter {
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws Exception
    {
        HttpSession session;
        Student student;
        String uri = request.getRequestURI();

        if (!uri.endsWith("login.html") && !uri.endsWith("logout.html")) {
            session = request.getSession();
            student = (User) session.getAttribute("student");
            if (student == null) {
                response.sendRedirect("login.html");
                return false;
            }
        }

        return true;
    }
```

Exception Handling

- Send Users to a special page when a specified exception occurs
  - Add SimpleMappingExceptionResolver to your web context
  - List Exception names and View that should be rendered

```
<bean
  class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
      <props>
        <prop key="com.nvz.courseapp.exceptions.UnexpectedScenarioException"
              value="CourseAppExceptionPage"/>
        <prop key="java.lang.Exception" value="oops"/>
      </props>
  </property>
</bean>
```

- Views have access to the exception
  - Use ${exception}

```
<html>

<h1>${exception.getMessage}</h1>

</html>
```