Spring MVC:

Exception Handling

Spring MVC Framework provides following ways to help us achieving robust exception handling.

1. **Controller Based** – We can define exception handler methods in our controller classes. All we need is to annotate these methods with @ExceptionHandler annotation. This annotation takes Exception class as argument. So if we have defined one of these for Exception class, then all the exceptions thrown by our request handler method will have handled.

These exception handler methods are just like other request handler methods and we can build error response and respond with different error page. We can also send JSON error response, that we will look later on in our example.

If there are multiple exception handler methods defined, then handler method that is closest to the Exception class is used. For example, if we have two handler methods defined for IOException and Exception and our request handler method throws IOException, then handler method for IOException will get executed.

Custom Exception Class

package com.journaldev.spring.exceptions;

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value=HttpStatus.NOT\_FOUND, reason="Employee Not Found") //404

public class EmployeeNotFoundException extends Exception {

    private static final long serialVersionUID = -3332292346834265371L;

    public EmployeeNotFoundException(int id){

        super("EmployeeNotFoundException with id="+id);

    }

}

Notice that we can use @ResponseStatus annotation with exception classes to define the HTTP code that will be sent by our application when this type of exception is thrown by our application and handled by our exception handling implementations.

As you can see that I am setting HTTP status as 404 and we have an error-page defined for this, so our application should use the error page for this type of exception if we are not returning any view.

### Controller Class with Exception Handler Methods

@Controller

public class EmployeeController {

    private static final Logger logger = LoggerFactory.getLogger(EmployeeController.class);

    @RequestMapping(value="/emp/{id}", method=RequestMethod.GET)

    public String getEmployee(@PathVariable("id") int id, Model model) throws Exception{

        //deliberately throwing different types of exception

        if(id==1){

            throw new EmployeeNotFoundException(id);

        }else if(id==2){

            throw new SQLException("SQLException, id="+id);

        }else if(id==3){

            throw new IOException("IOException, id="+id);

        }else if(id==10){

            Employee emp = new Employee();

            emp.setName("Pankaj");

            emp.setId(id);

            model.addAttribute("employee", emp);

            return "home";

        }else {

            throw new Exception("Generic Exception, id="+id);

        }

    }

    @ExceptionHandler(EmployeeNotFoundException.class)

    public ModelAndView handleEmployeeNotFoundException(HttpServletRequest request, Exception ex){

        logger.error("Requested URL="+request.getRequestURL());

        logger.error("Exception Raised="+ex);

        ModelAndView modelAndView = new ModelAndView();

        modelAndView.addObject("exception", ex);

        modelAndView.addObject("url", request.getRequestURL());

        modelAndView.setViewName("error");

        return modelAndView;

    }

}

1. **Global Exception Handler**

Exception Handling is a cross-cutting concern, it should be done for all the pointcuts in our application.

Spring provides@ControllerAdvice annotation that we can use with any class to define our global exception handler.

The handler methods in Global Controller Advice is same as Controller based exception handler methods and used when controller class is not able to handle the exception.

import java.io.IOException;

import java.sql.SQLException;

import javax.servlet.http.HttpServletRequest;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice

public class GlobalExceptionHandler {

    private static final Logger logger = LoggerFactory.getLogger(GlobalExceptionHandler.class);

    @ExceptionHandler(SQLException.class)

    public String handleSQLException(HttpServletRequest request, Exception ex){

        logger.info("SQLException Occured:: URL="+request.getRequestURL());

        return "database\_error";

    }

    @ResponseStatus(value=HttpStatus.NOT\_FOUND, reason="IOException occured")

    @ExceptionHandler(IOException.class)

    public void handleIOException(){

        logger.error("IOException handler executed");

        //returning 404 error code

    }

}

1. **HandlerExceptionResolver implementation**:

 For generic exceptions, most of the times we serve static pages.

Spring Framework provides HandlerExceptionResolver interface that we can implement to create global exception handler.

The reason behind this additional way to define global exception handler is that Spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.

SimpleMappingExceptionResolver is the default implementation class, it allows us to configure exceptionMappings where we can specify which resource to use for a particular exception. We can also override it to create our own global handler with our application specific changes, such as logging of exception messages.

My implementation in Rest API:

Custom Exception:

**public** **class** BadRequestException **extends** RuntimeException {

/\*\* The serialVersionUID. \*/

**private** **static** **final** **long** ***serialVersionUID*** = 2013471991481136493L;

/\*\*

\*

\*/

**public** BadRequestException() {

**super**();

}

/\*\*

\* **@param** message

\* **@param** cause

\* **@param** arg2

\* **@param** arg3

\*/

**public** BadRequestException(String message, Throwable cause, **boolean** arg2, **boolean** arg3) {

**super**(message, cause, arg2, arg3);

}

/\*\*

\* **@param** message

\* **@param** cause

\*/

**public** BadRequestException(String message, Throwable cause) {

**super**(message, cause);

}

/\*\*

\* **@param** message

\*/

**public** BadRequestException(String message) {

**super**(message);

}

/\*\*

\* **@param** cause

\*/

**public** BadRequestException(Throwable cause) {

**super**(cause);

}

}

Rest Controller

/\*\*

\* This method returns all records which have status as dead or partially

\* updated and accept status value as query parameter.

\*

\* **@param** status

\* **@return**

\* **@throws** Exception

\*/

@RequestMapping(value = "/cuidkoidmapping/status/{status}", method = RequestMethod.***GET***, headers = "Accept=application/json", produces = "application/json; charset=utf-8")

**public** @ResponseBody ResponseEntity<CuidKoidMappingsResponse> getCuidKoidMappingRecords(

@PathVariable("status") String status,

@RequestParam(value = "pageNumber", required = **false**) String pageNumber,

@RequestParam(value = "resultsPerPage", required = **false**) String resultsPerPage) **throws** Exception {

*logger*.info("entering CuidKoidMappingRestController-getCuidKoidMappingRecords");

// status validations

// status is empty

**if** (status == **null** || status.isEmpty()) {

*logger*.error("Status is missing in the request.");

**throw** **new** BadRequestException("Status is missing in the request.");

}

// status is not in correct number format.

**int** cuidKoidMappingStatus = 0;

**try** {

cuidKoidMappingStatus = Integer.*parseInt*(status);

} **catch** (NumberFormatException nfe) {

*logger*.error("Status value " + status + " is not correct. " + nfe);

**throw** **new** BadRequestException("Status value " + status + " is not correct.");

}

// validations for pageNumber and resultsPerPage

// check if pageNumber and resultsPerPage both are not present.

**if** ((pageNumber == **null** || resultsPerPage == **null**) && (pageNumber != **null** || resultsPerPage != **null**)) {

*logger*.error("pageNumber and resultsPerPage both are required.");

**throw** **new** BadRequestException("pageNumber and resultsPerPage both are required.");

}

// check for pagination required or not.

**boolean** paginationResultFlag = **false**;

**int** pageNumberInt = 0;

**int** resultsPerPageInt = 0;

**if** (pageNumber != **null** && resultsPerPage != **null**) {

paginationResultFlag = **true**;

**try** {

pageNumberInt = Integer.*parseInt*(pageNumber);

resultsPerPageInt = Integer.*parseInt*(resultsPerPage);

} **catch** (NumberFormatException nfe) {

*logger*.error("Either pageNumber: " + pageNumber + " or reusultsPerPage: " + resultsPerPage

+ " is not correct. " + nfe);

**throw** **new** BadRequestException("Either pageNumber: " + pageNumber + "or reusultsPerPage: "

+ resultsPerPage + " is not correct. ");

}

}

// status doesn't belong to KOID\_STATUS\_DEAD or

// KOID\_STATUS\_PARTIAL\_UPDATE

**if** (cuidKoidMappingStatus != PasswordSyncConstants.***KOID\_STATUS\_DEAD***

&& cuidKoidMappingStatus != PasswordSyncConstants.***KOID\_STATUS\_PARTIAL\_UPDATE***) {

*logger*.error("Status value " + cuidKoidMappingStatus + " is not correct. ");

**throw** **new** BadRequestException("Status value " + cuidKoidMappingStatus + " is not correct.");

}

List<CuidKoidMapping> cuidKoidMappingList = **null**;

CuidKoidMappingsResponse cuidKoidMappingsResonse = **new** CuidKoidMappingsResponse();

**try** {

cuidKoidMappingsResonse

.setTotalCount(String.*valueOf*(cuidKoidMappingDAO.getCountByStatus(cuidKoidMappingStatus)));

**if** (paginationResultFlag) {

cuidKoidMappingList = cuidKoidMappingDAO.getRecordsByStatus(cuidKoidMappingStatus, pageNumberInt,

resultsPerPageInt);

} **else** {

cuidKoidMappingList = cuidKoidMappingDAO.getRecordsByStatus(cuidKoidMappingStatus);

}

cuidKoidMappingsResonse.setCuidKoidMappingList(cuidKoidMappingList);

} **catch** (Exception e) {

*logger*.error(e);

**throw** **new** InternalServerException(e);

}

*logger*.info("completed CuidKoidMappingRestController-getCuidKoidMappingRecords");

**return** **new** ResponseEntity<CuidKoidMappingsResponse>(cuidKoidMappingsResonse, HttpStatus.***OK***);

}

Global Exception Handler

@ControllerAdvice

**public** **class** ControllerExceptionHandler {

/\*\*

\* This handles the ResourceNotFoundException and convert it into

\* HttpStatus.NOT\_FOUND with custom error message.

\*

\* **@param** ex

\* **@param** request

\* **@return**

\*/

@ResponseStatus(HttpStatus.***NOT\_FOUND***)

@ExceptionHandler(ResourceNotFoundException.**class**)

**public** ResponseEntity<ErrorInfo> handleResourceNotFoundException(RuntimeException ex, WebRequest request) {

**return** **new** ResponseEntity<ErrorInfo>(**new** ErrorInfo(HttpStatus.***NOT\_FOUND***.toString(), ex.getMessage()),

HttpStatus.***NOT\_FOUND***);

}

/\*\*

\* This handles the BadRequestException and convert it into

\* HttpStatus.BAD\_REQUEST with custom error message.

\*

\* **@param** ex

\* **@param** request

\* **@return**

\*/

@ResponseStatus(HttpStatus.***BAD\_REQUEST***)

@ExceptionHandler(BadRequestException.**class**)

**public** ResponseEntity<ErrorInfo> handleBadRequestException(RuntimeException ex, WebRequest request) {

**return** **new** ResponseEntity<ErrorInfo>(**new** ErrorInfo(HttpStatus.***BAD\_REQUEST***.toString(), ex.getMessage()),

HttpStatus.***BAD\_REQUEST***);

}

/\*\*

\* This handles the InternalServerException and convert it into

\* HttpStatus.INTERNAL\_SERVER\_ERROR with custom error message.

\*

\* **@param** ex

\* **@param** request

\* **@return**

\*/

@ResponseStatus(HttpStatus.***INTERNAL\_SERVER\_ERROR***)

@ExceptionHandler(InternalServerException.**class**)

**public** ResponseEntity<ErrorInfo> handleInternalServerException(RuntimeException ex, WebRequest request) {

**return** **new** ResponseEntity<ErrorInfo>(

**new** ErrorInfo(HttpStatus.***INTERNAL\_SERVER\_ERROR***.toString(), ex.getMessage()),

HttpStatus.***INTERNAL\_SERVER\_ERROR***);

}

Spring MVC with Annotation:



## **@Controller Annotation**

We can add “component-scan” in spring-context and provide the base-package.

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

<mvc:annotation-driven />

Then add @Controller annotation to controllers.

The dispatcher will start from the base-package and scan for beans that are annotated with @Controller annotation and look for @RequestMapping.

@Controller annotation just tells the container that this bean is a designated controller class

## **@RequestMapping Annotation**

@RequestMapping annotation is used to map a particular HTTP request method (GET/POST) to a specific class/method in controller which will handle the respective request.

@RequestMapping annotation can be applied both at class and method level. In class level we can map the URL of the request and in method we can map the url as well as HTTP request method (GET/POST).

We can use wildcard characters like \* for path pattern matching.

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

@Controller

@RequestMapping("/hi")

public class HelloWorldController {

@RequestMapping(method = RequestMethod.GET)

public String hello() {

return "hello";

}

@RequestMapping(method = RequestMethod.POST)

public String hi(@RequestParam("name") String name, Model model) {

String message = "Hi " + name + "!";

model.addAttribute("message", message);

return "hi";

}

}

**Multi-action Controller**

In a multi-action controller urls are mapped at method level since the controller services multiple urls. In below example two urls are serviced by the controller and they are mapped to separate methods.

package com.javapapers.spring.mvc;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

@Controller

public class HelloWorldController {

@RequestMapping("/")

public String hello() {

return "hello";

}

@RequestMapping(value = "/hi", method = RequestMethod.GET)

public String hi(@RequestParam("name") String name, Model model) {

String message = "Hi " + name + "!";

model.addAttribute("message", message);

return "hi";

}

}

## **@RequestParam Annotation**

To bind the request parameter a variable in method scope this @RequestParam annotation is used.

In below code “name” is bound to the request parameter.

public String hi(@RequestParam("name") String name, Model model) {

String message = "Hi " + name + "!";

model.addAttribute("message", message);

return "hi";

}

* @RequestParam can also be used for query parameter/query string

@RequestMapping(value = "/cuidkoidmapping/status/{status}", method = RequestMethod.***GET***, headers = "Accept=application/json", produces = "application/json; charset=utf-8")

**public** @ResponseBody ResponseEntity<CuidKoidMappingsResponse> getCuidKoidMappingRecords(

@PathVariable("status") String status,

@RequestParam(value = "pageNumber", required = **false**) String pageNumber,

@RequestParam(value = "resultsPerPage", required = **false**) String resultsPerPage) **throws** Exception {}

## **@ModelAttribute Annotation**

An annotated method parameter can be mapped to an attribute in a model using @Modelttribute in controller. It can also be used to provide reference data for the model when used at method level.

## **@SessionAttributes, @CookieValue, @RequestHeader Annotation**

We can declare session attributes used by a method handler using @SessionAttributes annotation in controller.

Similarly @CookieValue annotation is used to bind a method parameter to a HTTP cookie. In below example, a cookie with key “username” value will be bound to method variable name.

@RequestMapping("/hi")

public void userInfo(@CookieValue("username") String name) {

//...

}

Very similar to cookie binding, @RequestHeader is used to bind a header value to a method parameter.  
Assume we have the following header value, and the following annotation in controller will bind the host variable to the value.  
Host: localhost:8080

@RequestMapping("/hi")

public void hostInfo(@RequestHeader("Host") String host) {

//...

}

## Spring Annotations: Contents:

|  |  |
| --- | --- |
| **Annotation** | **Package Detail/Import statement** |
| [@Service](http://www.techferry.com/articles/spring-annotations.html#Service) | import org.springframework.stereotype.Service; |
| [@Repository](http://www.techferry.com/articles/spring-annotations.html#Repository) | import org.springframework.stereotype.Repository; |
| [@Component](http://www.techferry.com/articles/spring-annotations.html#Component) | import org.springframework.stereotype.Component; |
| [@Autowired](http://www.techferry.com/articles/spring-annotations.html#Autowired) | import org.springframework.beans.factory.annotation.Autowired; |
| [@Transactional](http://www.techferry.com/articles/spring-annotations.html#Transactional) | import org.springframework.transaction.annotation.Transactional; |
| [@Scope](http://www.techferry.com/articles/spring-annotations.html#Scope) | import org.springframework.context.annotation.Scope; |
| [Spring MVC Annotations](http://www.techferry.com/articles/spring-annotations.html#MVC) | |
| [@Controller](http://www.techferry.com/articles/spring-annotations.html#Controller) | import org.springframework.stereotype.Controller; |
| [@RequestMapping](http://www.techferry.com/articles/spring-annotations.html#RequestMapping) | import org.springframework.web.bind.annotation.RequestMapping; |
| [@PathVariable](http://www.techferry.com/articles/spring-annotations.html#PathVariable) | import org.springframework.web.bind.annotation.PathVariable; |
| [@RequestParam](http://www.techferry.com/articles/spring-annotations.html#RequestParam) | import org.springframework.web.bind.annotation.RequestParam; |
| [@ModelAttribute](http://www.techferry.com/articles/spring-annotations.html#ModelAttribute) | import org.springframework.web.bind.annotation.ModelAttribute; |
| [@SessionAttributes](http://www.techferry.com/articles/spring-annotations.html#SessionAttributes) | import org.springframework.web.bind.annotation.SessionAttributes; |
| [Spring Security Annotations](http://www.techferry.com/articles/spring-annotations.html#SpringSecurity) | |
| [@PreAuthorize](http://www.techferry.com/articles/spring-annotations.html#PreAuthorize) | import org.springframework.security.access.prepost.PreAuthorize; |

For spring to process annotations, add the following lines in your application-context.xml file.

<context:annotation-config />

<context:component-scan base-package="...specify your package name..." />

|  |  |
| --- | --- |
| Spring Annotation Tip | Spring supports both Annotation based and XML based configurations. You can even mix them together. Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches. |

### @Service

Annotate all your service classes with @Service. All your business logic should be in Service classes.

@Service

public class CompanyServiceImpl implements CompanyService {

...

}

### @Repository

Annotate all your DAO classes with @Repository. All your database access logic should be in DAO classes.

@Repository

public class CompanyDAOImpl implements CompanyDAO {

...

}

### @Component

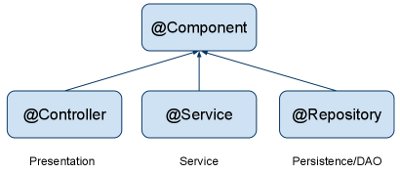
Annotate your other components (for example REST resource classes) with @Component.

@Component

public class ContactResource {

...

}

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.   
  


### @Autowired

Let Spring auto-wire other beans into your classes using @Autowired annotation. 

@Service

public class CompanyServiceImpl implements CompanyService {

@Autowired

private CompanyDAO companyDAO;

...

}

|  |  |
| --- | --- |
| Spring Annotation Tip | Spring beans can be wired by name or by type.   * @Autowire by default is a type driven injection. @Qualifier spring annotation can be used to further fine-tune autowiring. * @Resource (javax.annotation.Resource) annotation can be used for wiring by name.   Beans that are themselves defined as a collection or map type cannot be injected through @Autowired, because type matching is not properly applicable to them. Use @Resource for such beans, referring to the specific collection or map bean by unique name. |

### @Transactional

Configure your transactions with @Transactional spring annotation.

@Service

public class CompanyServiceImpl implements CompanyService {

@Autowired

private CompanyDAO companyDAO;

@Transactional

public Company findByName(String name) {

Company company = companyDAO.findByName(name);

return company;

}

...

}

|  |  |
| --- | --- |
| Spring Annotation Tip | To activate processing of Spring's @Transactional annotation, use the <tx:annotation-driven/> element in your spring's configuration file. |

The default @Transactional settings are as follows:

* Propagation setting is PROPAGATION\_REQUIRED.
* Isolation level is ISOLATION\_DEFAULT.
* Transaction is read/write.
* Transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.
* Any RuntimeException triggers rollback, and any checked Exception does not.

These default settings can be changed using various properties of the @Transactional spring annotation. 

|  |  |
| --- | --- |
| Spring Annotation Tip | Specifying the @Transactional annotation on the bean class means that it applies to all applicable business methods of the class. Specifying the annotation on a method applies it to that method only. If the annotation is applied at both the class and the method level, the method value overrides if the two disagree. |

### @Scope

As with Spring-managed components in general, the default and most common scope for autodetected components is singleton. To change this default behavior, use @Scope spring annotation.

@Component

@Scope("request")

public class ContactResource {

...

}

Similarly, you can annotate your component with @Scope("prototype") for beans with prototype scopes.

|  |  |
| --- | --- |
| Spring Annotation Tip | Please note that the dependencies are resolved at instantiation time. For prototype scope, it does NOT create a new instance at runtime more than once. It is only during instantiation that each bean is injected with a separate instance of prototype bean. |

## Spring MVC Annotations

### @Controller

Annotate your controller classes with @Controller.

@Controller

public class CompanyController {

...

}

### @RequestMapping

You use the @RequestMapping spring annotation to map URLs onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping.

@Controller

@RequestMapping("/company")

public class CompanyController {

@Autowired

private CompanyService companyService;

...

}

### @PathVariable

You can use the @PathVariable spring annotation on a method argument to bind it to the value of a URI template variable. In our example below, a request path of /company/techferry will bind companyName variable with 'techferry' value.

@Controller

@RequestMapping("/company")

public class CompanyController {

@Autowired

private CompanyService companyService;

@RequestMapping("{companyName}")

public String getCompany(Map<String, Object> map,

@PathVariable String companyName) {

Company company = companyService.findByName(companyName);

map.put("company", company);

return "company";

}

...

}

### @RequestParam

You can bind request parameters to method variables using spring annotation @RequestParam.

@Controller

@RequestMapping("/company")

public class CompanyController {

@Autowired

private CompanyService companyService;

@RequestMapping("/companyList")

public String listCompanies(Map<String, Object> map,

@RequestParam int pageNum) {

map.put("pageNum", pageNum);

map.put("companyList", companyService.listCompanies(pageNum));

return "companyList";

}

...

}

Similarly, you can use spring annotation @RequestHeader to bind request headers.

### @ModelAttribute

An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, the argument's fields should be populated from all request parameters that have matching names. This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

@Controller

@RequestMapping("/company")

public class CompanyController {

@Autowired

private CompanyService companyService;

@RequestMapping("/add")

public String saveNewCompany(@ModelAttribute Company company) {

companyService.add(company);

return "redirect:" + company.getName();

}

...

}

### @SessionAttributes

@SessionAttributes spring annotation declares session attributes. This will typically list the names of model attributes which should be transparently stored in the session, serving as form-backing beans between subsequent requests.

@Controller

@RequestMapping("/company")

@SessionAttributes("company")

public class CompanyController {

@Autowired

private CompanyService companyService;

...

}

@SessionAttribute works as follows:

* @SessionAttribute is initialized when you put the corresponding attribute into model (either explicitly or using @ModelAttribute-annotated method).
* @SessionAttribute is updated by the data from HTTP parameters when controller method with the corresponding model attribute in its signature is invoked.
* @SessionAttributes are cleared when you call setComplete() on SessionStatus object passed into controller method as an argument.

The following listing illustrate these concepts. It is also an example for pre-populating Model objects.

@Controller

@RequestMapping("/owners/{ownerId}/pets/{petId}/edit")

@SessionAttributes("pet")

public class EditPetForm {

@ModelAttribute("types")

public Collection<PetType> populatePetTypes() {

return this.clinic.getPetTypes();

}

@RequestMapping(method = RequestMethod.POST)

public String processSubmit(@ModelAttribute("pet") Pet pet,

BindingResult result, SessionStatus status) {

new PetValidator().validate(pet, result);

if (result.hasErrors()) {

return "petForm";

}else {

this.clinic.storePet(pet);

status.setComplete();

return "redirect:owner.do?ownerId="

+ pet.getOwner().getId();

}

}

}

## Spring Security Annotations

### @PreAuthorize

Using Spring Security @PreAuthorize annotation, you can authorize or deny a functionality. In our example below, only a user with Admin role has the access to delete a contact.

@Transactional

@PreAuthorize("hasRole('ROLE\_ADMIN')")

public void removeContact(Integer id) {

contactDAO.removeContact(id);

}

<web-app>

<servlet>

<servlet-name>golfing</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>golfing</servlet-name>

<url-pattern>/golfing/\*</url-pattern>

</servlet-mapping>

</web-app>

@Controller

**public** **class** HelloWorldController {

@RequestMapping("/helloWorld")

**public** ModelAndView helloWorld() {

ModelAndView mav = **new** ModelAndView();

mav.setViewName("helloWorld");

mav.addObject("message", "Hello World!");

**return** mav;

}

}

<?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:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="org.springframework.samples.petclinic.web"/>

// ...

</beans>

@Controller

@RequestMapping("/appointments")

**public** **class** AppointmentsController {

**private** **final** AppointmentBook appointmentBook;

@Autowired

**public** AppointmentsController(AppointmentBook appointmentBook) {

**this**.appointmentBook = appointmentBook;

}

@RequestMapping(method = RequestMethod.GET)

**public** Map<String, Appointment> get() {

**return** appointmentBook.getAppointmentsForToday();

}

@RequestMapping(value="/{day}", method = RequestMethod.GET)

**public** Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {

**return** appointmentBook.getAppointmentsForDay(day);

}

@RequestMapping(value="/new", method = RequestMethod.GET)

**public** AppointmentForm getNewForm() {

**return** **new** AppointmentForm();

}

@RequestMapping(method = RequestMethod.POST)

**public** String add(@Valid AppointmentForm appointment, BindingResult result) {

**if** (result.hasErrors()) {

**return** "appointments/new";

}

appointmentBook.addAppointment(appointment);

**return** "redirect:/appointments";

}

}

@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)

**public** String findOwner(@PathVariable String ownerId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

model.addAttribute("owner", owner);

**return** "displayOwner";

}

@Controller

@RequestMapping("/owners/{ownerId}")

**public** **class** RelativePathUriTemplateController {

@RequestMapping(value = "/pets/{petId}", params="myParam=myValue")

**public** **void** findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {

*// implementation omitted*

}

}

@Controller

@RequestMapping("/owners/{ownerId}")

**public** **class** RelativePathUriTemplateController {

@RequestMapping(value = "/pets", method = RequestMethod.POST, headers="content-type=text/\*")

**public** **void** addPet(Pet pet, @PathVariable String ownerId) {

*// implementation omitted*

}

}

#### Supported handler method arguments and return types

Handler methods that are annotated with @RequestMapping can have very flexible signatures. Most of them can be used in arbitrary order (see below for more details).

* Request or response objects (Servlet API). Choose any specific request or response type, for example ServletRequest or HttpServletRequest.
* Session object (Servlet API): of type HttpSession. An argument of this type enforces the presence of a corresponding session. As a consequence, such an argument is never null.

|  |  |
| --- | --- |
| [Note] | **Note** |
| Session access may not be thread-safe, in particular in a Servlet environment. Consider setting theAnnotationMethodHandlerAdapter's "synchronizeOnSession" flag to "true" if multiple requests are allowed to access a session concurrently. |

* org.springframework.web.context.request.WebRequest or org.springframework.web.context.request.NativeWebRequest. Allows for generic request parameter access as well as request/session attribute access, without ties to the native Servlet/Portlet API.
* java.util.Locale for the current request locale, determined by the most specific locale resolver available, in effect, the configured LocaleResolver in a Servlet environment.
* java.io.InputStream / java.io.Reader for access to the request's content. This value is the raw InputStream/Reader as exposed by the Servlet API.
* java.io.OutputStream / java.io.Writer for generating the response's content. This value is the raw OutputStream/Writer as exposed by the Servlet API.
* java.security.Principal containing the currently authenticated user.
* @PathVariable annotated parameters for access to URI template variables. See [Section 15.3.2.1, “URI Templates”](http://docs.spring.io/spring-framework/docs/3.0.x/reference/mvc.html#mvc-ann-requestmapping-uri-templates).
* @RequestParam annotated parameters for access to specific Servlet request parameters. Parameter values are converted to the declared method argument type. See [Section 15.3.2.4, “Binding request parameters to method parameters with @RequestParam”](http://docs.spring.io/spring-framework/docs/3.0.x/reference/mvc.html#mvc-ann-requestparam).
* @RequestHeader annotated parameters for access to specific Servlet request HTTP headers. Parameter values are converted to the declared method argument type.
* @RequestBody annotated parameters for access to the HTTP request body. Parameter values are converted to the declared method argument type using HttpMessageConverters. See [Section 15.3.2.5, “Mapping the request body with the @RequestBody annotation”](http://docs.spring.io/spring-framework/docs/3.0.x/reference/mvc.html#mvc-ann-requestbody).
* HttpEntity<?> parameters for access to the Servlet request HTTP headers and contents. The request stream will be converted to the entity body using HttpMessageConverters. See [Section 15.3.2.7, “Using HttpEntity<?>”](http://docs.spring.io/spring-framework/docs/3.0.x/reference/mvc.html#mvc-ann-httpentity).
* java.util.Map / org.springframework.ui.Model / org.springframework.ui.ModelMap for enriching the implicit model that is exposed to the web view.
* Command or form objects to bind parameters to: as bean properties or fields, with customizable type conversion, depending on @InitBindermethods and/or the HandlerAdapter configuration. See the webBindingInitializer property on AnnotationMethodHandlerAdapter. Such command objects along with their validation results will be exposed as model attributes by default, using the non-qualified command class name in property notation. For example, "orderAddress" for type "mypackage.OrderAddress". Specify a parameter-level ModelAttribute annotation for declaring a specific model attribute name.
* org.springframework.validation.Errors / org.springframework.validation.BindingResult validation results for a preceding command or form object (the immediately preceding method argument).
* org.springframework.web.bind.support.SessionStatus status handle for marking form processing as complete, which triggers the cleanup of session attributes that have been indicated by the @SessionAttributes annotation at the handler type level.

The Errors or BindingResult parameters have to follow the model object that is being bound immediately as the method signature might have more that one model object and Spring will create a separate BindingResult instance for each of them so the following sample won't work:

**Example 15.1. Invalid ordering of BindingResult and @ModelAttribute**

@RequestMapping(method = RequestMethod.POST)

**public** String processSubmit(@ModelAttribute("pet") Pet pet,

Model model, BindingResult result) { … }

Note, that there is a Model parameter in between Pet and BindingResult. To get this working you have to reorder the parameters as follows:

@RequestMapping(method = RequestMethod.POST)

**public** String processSubmit(@ModelAttribute("pet") Pet pet,

BindingResult result, Model model) { … }

**ModelAndView:**

The model presents a placeholder to hold the information you want to display on the view. It could be a string, which is in your above example, or it could be an object containing bunch of properties.

**Example 1**

If you have...

return new ModelAndView("welcomePage","WelcomeMessage","Welcome!");

... then in your jsp, to display the message, you will do:-

Hello Stranger! ${WelcomeMessage} // displays Hello Stranger! Welcome!

**Example 2**

If you have...

MyBean bean = new MyBean();

bean.setName("Mike!");

bean.setMessage("Meow!");

return new ModelAndView("welcomePage","model",bean);

... then in your jsp, you can do:-

Hello ${model.name}! {model.message} // displays Hello Mike! Meow!

**Student.java** file:

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

**StudentController.java** file:

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.ui.ModelMap;

@Controller

public class StudentController {

@RequestMapping(value = "/student", method = RequestMethod.GET)

public ModelAndView student() {

return new ModelAndView("student", "command", new Student());

}

@RequestMapping(value = "/addStudent", method = RequestMethod.POST)

public String addStudent(@ModelAttribute("SpringWeb")Student student,

ModelMap model) {

model.addAttribute("name", student.getName());

model.addAttribute("age", student.getAge());

model.addAttribute("id", student.getId());

return "result";

}

}

* the first service method **student()**, we have passed a blank **Student**object in the ModelAndView object with name "command" because the spring framework expects an object with name "command" if you are using <form:form> tags in your JSP file. So when **student()** method is called it returns **student.jsp** view.

Spring MVC interceptor:

spring MVC’s handler interceptor is like a good friend and will help in time of need. Spring’s handler interceptor as rightly named, intercepts a request,

* just before the controller or
* just after the controller or
* just before the response sent to view

Spring’s interceptor can be configured for all the requests (for any URI’s requested) or for a group of URI’s (may be for a set of modules, etc.). Just remember controller and handler are the same.

* In real scenario, Spring MVC handler interceptors are used for authentication, logging, to add a common message to all response.
* For the pages displayed we want to remove all bold tags from the response, it is possible using Spring interceptor.

## **Important Points about Spring Interceptor**

* [***HandlerInterceptor***](http://docs.spring.io/spring/docs/3.2.x/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html) – an interface, which must be implemented by the Spring interceptor classes, has the following three methods.
* ***preHandle(…)*** – called just before the controller
* ***postHandle(…)*** – called immediately after the controller
* ***afterCompletion(…)*** – called just before sending response to view
* ***HandlerInterceptorAdaptor*** – an implementation class of***HandlerInterceptor*** interface provided by Spring as a convenient class. By extending this we can override only the necessary methods out of the three.
* Interceptor classes must be declared in spring context xml configuration file within the tag **<mvc:interceptors>**
* Interceptor can be configured to execute in two ways, execute for all requests and map to specific url requests.
* ORDER: All global interceptors gets executed first and then the mapped interceptor. Among them, the same order in which the interceptor are declared, the execution is also done.
* If true is returned, the execution chain continues and for false, the execution stops for that request with that interceptor.
* Two interceptors declared. **GreetingInterceptor** is configured to be called for all the request. **AnimalInterceptor**is configured to be called only for the page /AnimalList (in path mapping we can use spring expression to choose a set of pages also).

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Component;

import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

@Component

public class GreetingInterceptor extends HandlerInterceptorAdapter {

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

System.out.println("GreetingInterceptor: REQUEST Intercepted for URI: "

+ request.getRequestURI());

request.setAttribute("greeting", "Happy Diwali!");

return true;

}

}

<?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/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config />

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

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/view/" />

<property name="suffix" value=".jsp" />

<property name="order" value="1" />

</bean>

<mvc:interceptors>

<bean class="com.javapapers.spring.mvc.interceptor.GreetingInterceptor" />

<mvc:interceptor>

<mvc:mapping path="/AnimalList" />

<bean class="com.javapapers.spring.mvc.interceptor.AnimalInterceptor" />

</mvc:interceptor>

</mvc:interceptors>

</beans>

Filter vs Interceptor:

1. Filter is at very high level. It is generic.  
2. Handler interceptor has got options, once intercepted it has access to the controller (handler) object on which the intercept has happened. We can do request specific operations.  
3. Interceptor can be configured based on handler mappings. We can intercept based on group of URLs and exclude URLs among them.  
4. Filter is configured in web.xml and interceptor is configured in spring context.  
5. Filter is more powerful, the request and response object that is passed on to the next level can be changed.  
6. Interceptor allows handler pre and post processing.