**Technical Specifications:**

1. **Java(1.7)**
2. **Spring(Spring IOC, Spring MVC, Spring Securities)**
3. **SpringBoot**
4. **Restful webservices.**
5. **Hibernate(ORM framework)**
6. **MySQL**
7. **Maven(Build tool)**
8. **Git (Version Control)**

**Alliedacademies Flow Guide**

Alliedacademies is developed under multi-layer architecture. First one is **Spring Web MVC** and second is **RESTful** architecture. These designes developed based on Annotations, didn’t use any xml files for configurations, where all dependencies are configured in POM.xml file.

**1.Spring Web MVC**

Spring MVC is designed around the front controller pattern where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components.

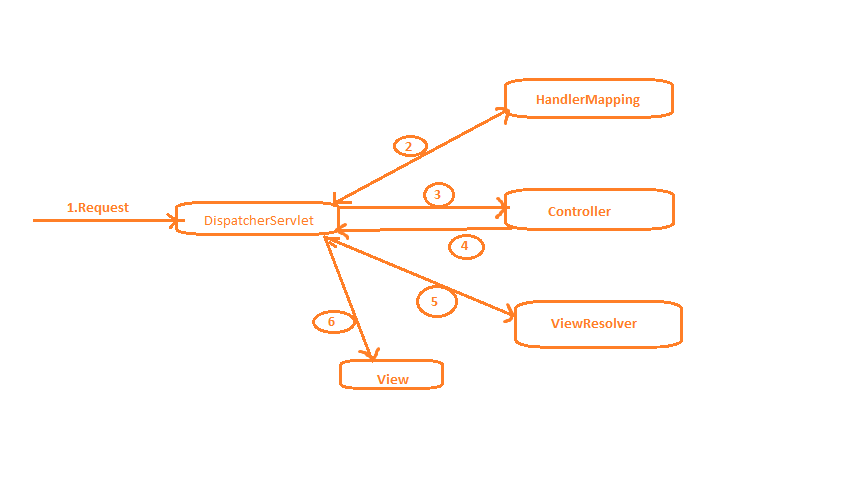
The Spring Web MVC framework provides Model-View-Controller architecture and ready components that can be used to develop flexible and loosely coupled webapplications.

DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for uploading files. The default handler is based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods.

All Conference pages developed under Spring MVC architecture.

Whenever we rise the request from the browser, Dispatcherservlet first trapes the request and takes the help of HandlerMapping class help in order to call the particular controller.

**Spring MVC Flow Diagram**



**MVC is a design pattern which provides a solution to layer an application by separating Business(Model), Presentation(View) and ControlFlow(Controller). The Model contains the business logic and the Controller is responsible for the redirection and the interaction between View component and Model. The View component contains the presentation part of the application.**

**1) First request will be received by DispatcherServlet.**

**2) DispatcherServlet will take the help of HandlerMapping and get to know the Controller class name associated with the given request.**

**3) So request transfer to the Controller, and then controller will process the request by executing appropriate method.**

**4) Controller returns ModelandView object back to the DispatcherServlet.**

**5) Now DispatcherServlet send the model object to the ViewResolver to get the actual view page.**

**6) Finally DispatcherServlet will pass the Model object to the View page to display the result.**

**Now we can check one request related to WEB, how this flow will happened.**

**MVC Flow in Conference Web Pages**

**For this we need to check 3 files where you can find url’s and controllers that are mapped to the particular url.**

1. **Constants.java**
2. **PatternService.java**
3. **WebUIService.java**

**All the url’s related to web requests configured in Constants.java.**

**All the requests related to web handled by the controller which is in PatternService.java and forward those requests to Service layer for further processing.**

**For example consider below url:**

**URL:** http://copd.alliedacademies.com/about

URL: http://copd.alliedacademies.com/call-for-abstracts

URL: http://copd.alliedacademies.com/organizing-committee

For all, sub-domain is same but the url related to requests is different. Identify the url key word first.

Then search the same key word in **PatternService.java.**

**T**ake the first request: **http://copd.alliedacademies.com/about**

You can find “about ” in **Constants.java.**

**i.e, public static final String CONFERENCE\_ABOUT\_URL= "about";**

**Then copy String name : CONFERENCE\_ABOUT\_URL**

**Search the same (CONFERENCE\_ABOUT\_URL) in PatternService.java**

**Then you can find the particular method that is mapped with that request.**

**case Constants.CONFERENCE\_ABOUT\_URL:**

**return webUIService.aboutConferenceScreen(modelAndView, conferenceId,request);**

**You can check the implementation in WebUIService.java**

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

@Transactional

**public** ModelAndView aboutConferenceScreen(ModelAndView modelAndView,

@PathVariable(value = "id") **long** conferenceId,HttpServletRequest request)

{

//Business logic to add the data to modelAndView object.

}

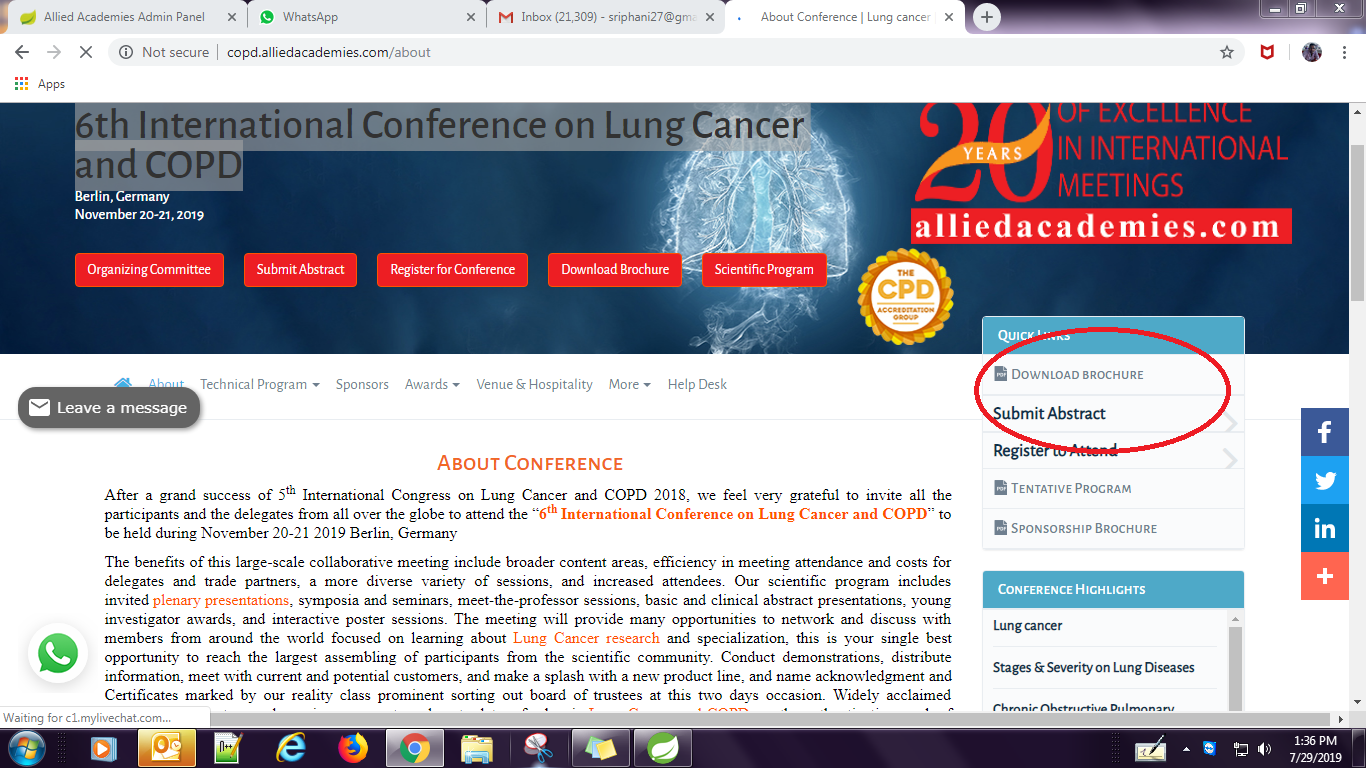
1. **In First step one attribute will be added to “modelAndView” related to view page information.**

modelAndView = conferenceHandler.setViewName(modelAndView, conferenceId, Constants.CONFERENCE\_ABOUT\_PAGE);

1. **In Second step some attributes will be added to “modelAndView” related to “QuickLinks”**

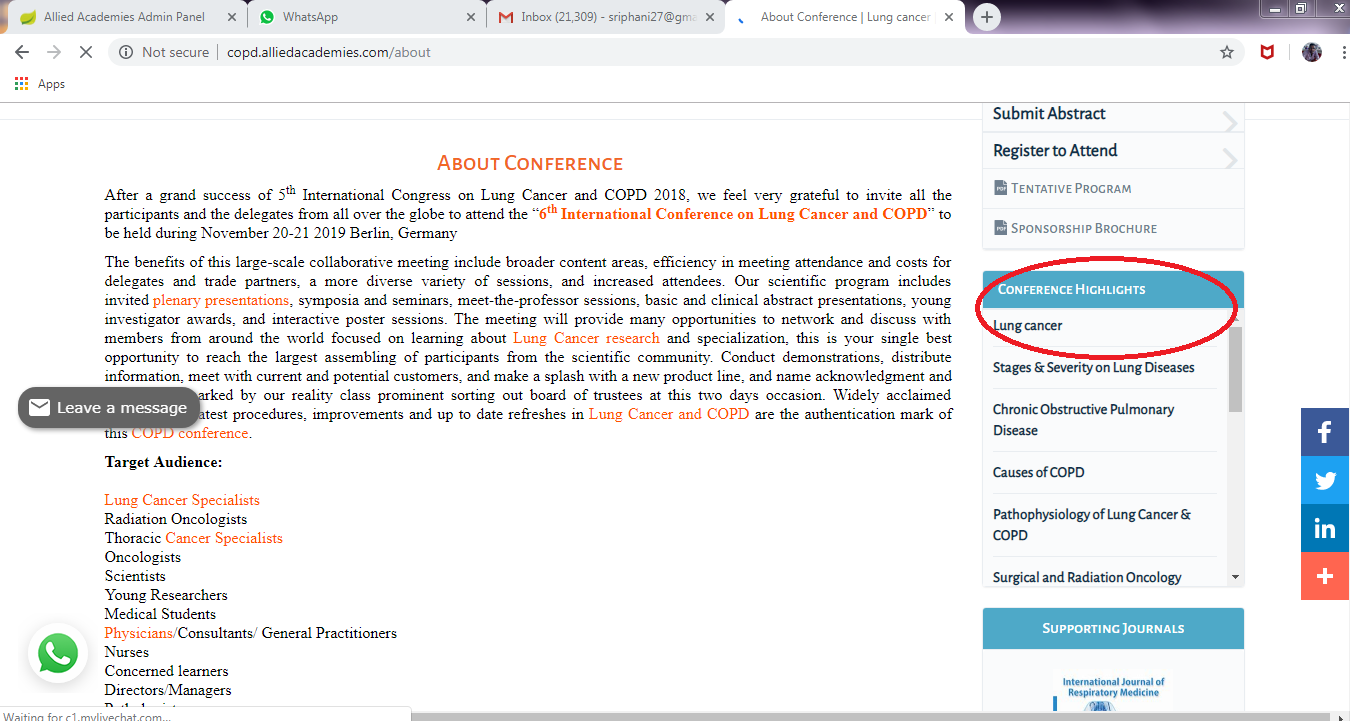
**modelAndView.addObject("isConferenceBrochureDownloadEnabled",**

**conferenceMiscellaneousHandler.isConferenceBrochureDownloadEnabled(conferenceId));**



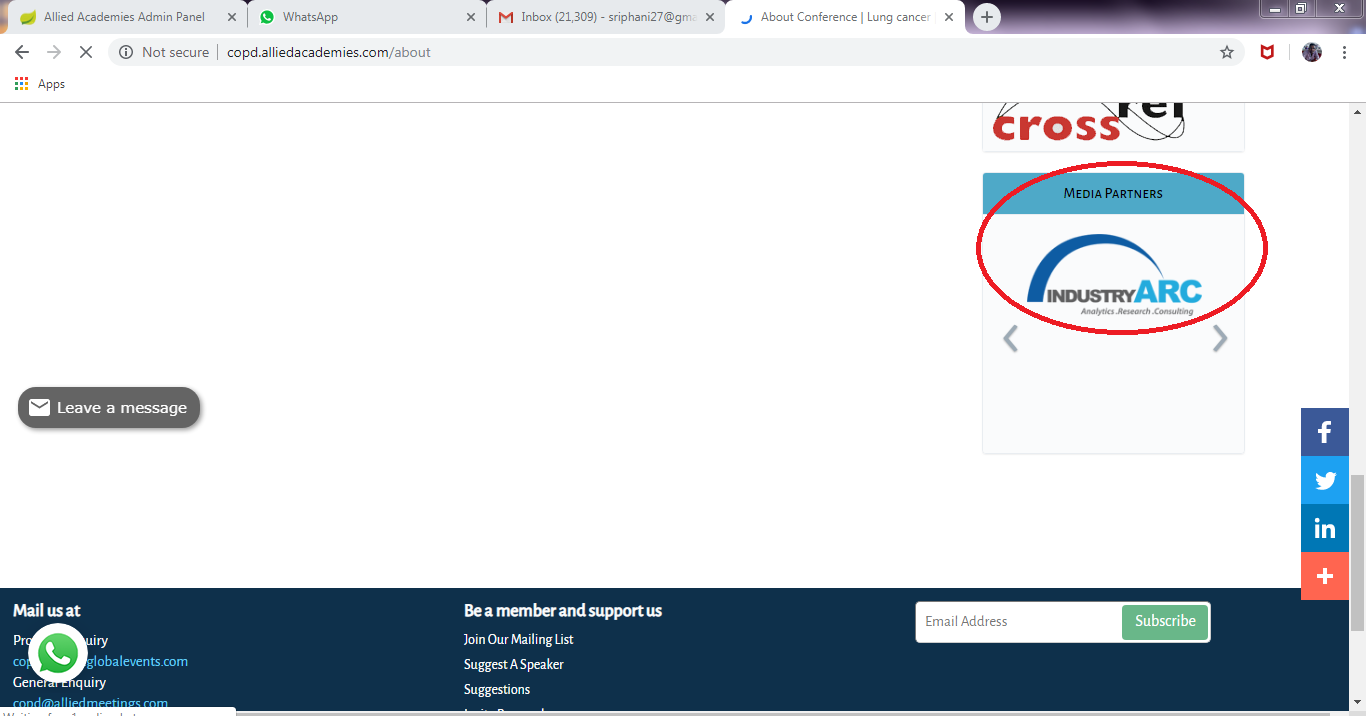
1. **In Third step another attribute will be added to “modelAndView” object related to “Conference Highlights”**

**modelAndView.addObject("conferenceHighlights", conferenceHandler.getConferenceHighlights(conferenceId));**



1. **Media Partners attribute will be added to “modelAndView” object.**

**modelAndView.addObject("conferenceMediaPartners", conferenceHandler.getMediaPartners(ConferencePartner.TYPE\_MEDIA\_PARTNERS, conferenceId, 1, 10).getList());**



**All the remaining attributes like SimilarConferences, RelatedConferences, ConferenceAssociations, ConferencePressRelease will be added in the same way.**

**Admin’s related flow guide**

**We have different admins provided in order for data updation. Whatever the data we are updating through admins those data will be available in Web.**

**All these admins are developed under RESTful services.**

**It uses different methods in order for data related operations.**

**HTTP GET**

**HTTP POST**

**HTTP PUT**

**HTTP DELETE**

**HTTP PATCH**

**HTTP GET**

**Use GET requests to retrieve resource representation/information only – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be safe methods.**

**HTTP POST**

**Use POST APIs to create new subordinate resources.** **Talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources.**

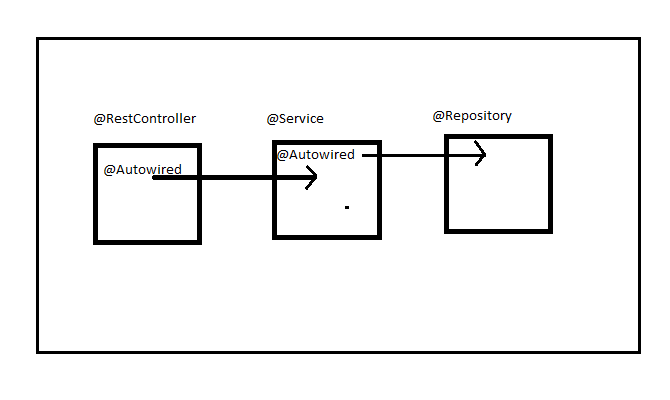
**The following are the admin related urls:**

<http://alliedacademies.com/management/login.html>

<http://alliedacademies.com/cnfadmin/login.html>

<http://alliedacademies.com/sysadmin/login.html>

In order to understand the flow related to admins refer the below diagram:



Once particular call is received at RestController, it will communicate Service layer in order to execute the business logic. Here Service layer related object will be injected in RestController by using @Autowired annotation.

In Service layer , if it needs any data from DB it will communicate Repository layer and this class object will be injected in Service layer by using @Autowired annotation.

We can check the flow from below steps:

Take any request from admin’s;

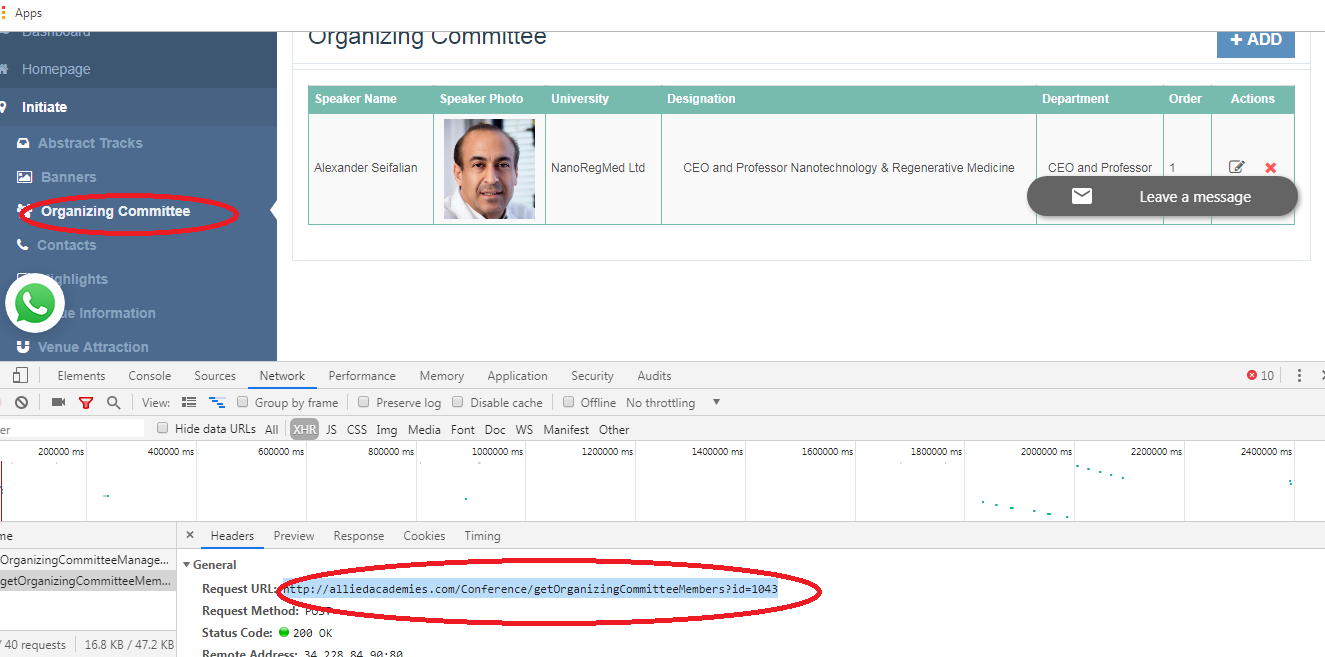
Ex : <http://alliedacademies.com/Conference/getOrganizingCommitteeMembers?id=1043>

[\*\*\*\*\*\*\*\*\*\*/**Conference**/getOrganizingCommitteeMembers?id=1043](http://alliedacademies.com/Conference/getOrganizingCommitteeMembers?id=1043)

Here “Conference” represents particular controller and “getOrganizingCommitteeMembers”

Represents particular method under controller. Where “id=1043” is the parameter that is read inside the method.

For this please check the below screen shot: and find the url that is raised.



URL: <http://alliedacademies.com/Conference/getOrganizingCommitteeMembers?id=1043>

First we need to identify the controller that is mapped with “Conference” , then we can understand the remaining flow.

Refer “**ConferenceService.java**”

@RestController

@RequestMapping(value = "Conference")

**public** **class** ConferenceService **extends** BaseService {

@Autowired

ConferenceMiscellaneousHandler conferenceMiscellaneousHandler;

}

Then we can identify particular method that is mapped with “getOrganizingCommitteeMembers”

@RequestMapping(value = "getOrganizingCommitteeMembers", method = RequestMethod.***POST***)

**public** List<ConferenceOrganizingCommittee> getOrganizingCommitteeMembers(

@RequestParam("id") **long** conferenceId) **throws** BusinessException

{

**return** conferenceHandler.getOrganizingCommitteeMembers(conferenceId);

}

Here “conferenceHandler” class object will be injected by @Autowired annotation .

In next step: check the below call and find its implementation.

conferenceHandler.getOrganizingCommitteeMembers(conferenceId);

In this call it will communicate with Service layer in order to execute the business logic.

Refer: “**ConferenceHandler.java**”

**public** List<ConferenceOrganizingCommittee> getOrganizingCommitteeMembers(

**long** conferenceId) **throws** ObjectNotFoundException

{

List<ConferenceOrganizingCommittee> cnfOrgCmtList = cnfOrgCmtDAO

.getOrganizingCommitteesByConferenceId(conferenceId);

}

This call is invoked from Controller layer and communicated with Service layer. In this layer we are injecting “**ConferenceOrganizingCommitteeDAO**” object by using @Autowired annotation.

@Autowired

ConferenceOrganizingCommitteeDAO cnfOrgCmtDAO;

By using this reference we can communicate with Repository layer. And check the below call how service layer communicate with repository layer.

List<ConferenceOrganizingCommittee> cnfOrgCmtList = cnfOrgCmtDAO

.getOrganizingCommitteesByConferenceId(conferenceId);

In this call it will communicate Repository layer in order for list of organizing committee members related to particular conferenceid.

Refer: “**ConferenceOrganizingCommitteeDAOImpl.java**”

@Repository("conferenceOrganizingCommitteeDAO")

**public** **class** ConferenceOrganizingCommitteeDAOImpl **extends** AbstractDao **implements**

ConferenceOrganizingCommitteeDAO

{

@Override

**public** List<ConferenceOrganizingCommittee> getOrganizingCommitteesByConferenceId(

**long** conferenceId)

{

Criteria criteria = createCustomCriteria(ConferenceOrganizingCommittee.**class**);

criteria.add(Restrictions.*eq*(Constants.***LABEL\_CONFERENCE\_ID***,conferenceId));

criteria.addOrder(Order.*asc*(ConferenceOrganizingCommittee.***LABEL\_ORDER***));

**return** criteria.list();

} }

In Repository layer it will connected to DB and fetch the details regarding organizing committee members.

\*\*\* Criteria criteria = createCustomCriteria(ConferenceOrganizingCommittee.**class**);

In this call criteria will be applied on “**cnforganizingcommittee**” table.

\*\*\*criteria.add(Restrictions.*eq*(Constants.***LABEL\_CONFERENCE\_ID***,conferenceId));

In this call restrictions will be applied for members.

\*\*\*criteria.addOrder(Order.*asc*(ConferenceOrganizingCommittee.***LABEL\_ORDER***));

Added the ascending order for returned members.

\*\*\* return criteria.list();

Finally it will return the list of OrganizingCommittee members for particular conference.

We can check the remaining calls in the same manner.