**Advice**: The implementation of the cross-cutting concern is called advice. **JoinPoint**: is a specific point (method) in code – Where the advice will be applied. **Target:** While executing an advice, the object on which the joinpoint is located is called the target. **Pointcut**: is a collection of points

**Aspect**: combination of advice and pointcut. What (advice) should execute where (pointcut)

**Weaving:** seen at execution time. Execution weaves back and forth between advice and the actual method. **@Around**: @Around has to choose when (and if) it calls the real method. Receives the parameters to pass to the real method. Receives the return from the real method

@Aspect @Component public class LogAspect { private static final Logger logger = LogManager.getLogger(LogAspect.class.getName());

@Before("execution(\* cs544.spring40.aop.terms.CustomerService.\*(..))")

public void logBefore(JoinPoint joinpoint) { logger.warn("Method: " + joinpoint.getSignature().getName()); }

@After("execution(\* cs544.spring40.aop.terms.CustomerService.\*(..))")

public void logAfter(JoinPoint joinpoint)

{ logger.warn("Just execed: " + joinpoint.getSignature().getName()); }

@AfterReturning(pointcut="execution(\* cs544.spring41.aop.advices.CustomerService.getName(..))", returning="ret") public void afterRet(JoinPoint jp, String ret) {

System.out.println(jp.getSignature().getName() + " returned: " + ret); }

@AfterThrowing(pointcut="execution(\* cs544.spring41.aop.advices.CustomerService.getAge(..))", throwing="ex") public void afterThrow(JoinPoint jp, MyException ex) { }

@Around("execution(\* cs544.spring41.aop.advices.CustomerService.getName(..))")

public Object around(ProceedingJoinPoint pjp) {

String m = pjp.getSignature().getName(); System.out.println("Before " + m);

Object ret = null; try { ret = pjp.proceed(); }

catch (Throwable e) { e.printStackTrace(); }

System.out.println("After " + m + " returned " + ret); return ret; }

**JoinPoint:** Every advice method can optionally receive as its first argument a JoinPoint object

The JoinPoint object contains info about the method (point) that will be (or was) joined for this call

JoinPoint is an optional argument for @Before, @After, @AfterReturning and @AfterThrowing

**ProceedingJoinPoint:** ProceedingJoinPoint extends JoinPoint for use inside an @Around advice adding the following overloaded method: Object proceed () Object proceed(Object[] args). Proceed without args causes the real method to be called, giving us (the advice) the return as an Object. ProceedingJoinPoint is not optional for @Around – Cannot function without it – Also has to return Object – And declare throws throwable (or catch it)

**PointCut Expression Language:** execution (modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?).

@**RequestMapping** can be used to map an incoming HTTP request to a method

@**PathVariable** can be used to bind the variable to an input parameter.

URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI.

**@RequestParam:** Method param name has to match request param name. It is optional parameter.

public String getParam (@RequestParam int id...) GET /cars?id=1.

@Configuration @EnableWebMvc @ComponentScan("cs544") public class WebConfig implements WebMvcConfigurer{@Bean public ViewResolver viewResolver() { InternalResourceViewResolver bean = new InternalResourceViewResolver(); bean.setViewClass(JstlView.class); bean.setPrefix("/WEB-INF/view/"); bean.setSuffix(".jsp"); return bean; }

@**ModelAttribute:** Critical for form data – Especially if you want to show an empty form.

@GetMapping(value="/addCar") public String get(@ModelAttribute("car") Car car) { return "addCar"; } =>Places an empty Car Object in the Model with key “car”

**DefaultRequestToViewNameTranslator**

@Controller can specify @**SessionAttributes.** – Intended for the duration of the controller. Lists the names of model attributes that should be stored in the session (instead of request). Once added to the model it will be available to subsequent requests.

**Flash Attributes**: Flash attributes are a way for a request to store attributes intended for use in the next (single) request – Stored in the session for a very short time – Removed right away after first use. To set: redirectAttrs.addFlashAttribute("message", msg);

To check: model.containsAttribute("message").

**Spring Data** offers a flexible abstraction for working with data access frameworks. is an umbrella project that provides you with easy to use data access technologies for all kinds of relational and non-relational DBs. Purpose is to unify and ease the access to different kinds of persistence stores.

**JpaRepository:** findAll(), findById(id), save(S entity), delete(ID id), delete(T entity), deleteAll(), exists(ID id), count(),saveAll(Iterable<s> entities), findAllById(Iterable ids)

public interface UserRepository extends JpaRepository{ @Query("Select u from User u where u.emailAddress = ?1") User findByEmailAddrss(String emailAddress);

2 things needed to debug **Spring Security**: – Enable in settings (web. .debug(true)) – Enable debug output on logger-<Logger name=”org.springframework.security level = “debug>

**Method Level security(@Secured,jsr-250 annotation,** @PreAuthorize and @PostAuthorize)

<dependency><groupId>org.springframework.security</groupId><artifactId>spring-security-config </artifactId></dependency>

@Configuration @EnableWebMvc @EnableGlobalMethodSecurity( securedEnabled = true, jsr250Enabled = true, prePostEnabled = true) @ComponentScan("cs544") public class WebConfig implements WebMvcConfigurer{ }

@Service @Transactional public class ContactService { @Resource private ContactDao contactDao; @Secured({ "ROLE\_USER", "ROLE\_ADMIN" }) public Contact get(Long id) { return contactDao.getOne(id); } @RolesAllowed({ "ROLE\_USER", "ROLE\_ADMIN" }) public Contact get(Long id) { return contactDao.getOne(id); }@PreAuthorize("hasRole('USER') or hasRole('ADMIN')") public void add(Contact contact) { contactDao.save(contact); }

@**Valid:** (Any non-primitive) Go into the object and validate it. @**Size**(min=,max=): String / Collection. Check size is >= min and <= max, column length set to max

@**NotEmpty:** String / Collection. Not null or empty.

**Messaging**: Messaging is a loosely coupled asynchronous and reliable communication between applications. **point to point** Message is sent from one application to another application via a queue. Here there is only one consumer but we can have multiple producers. **Publish – subscribe:** Message(publish) is sent from producer(publisher) to multiple consumers(subscribers) via topic. Here there is only one producer. The publisher is not required to know information about the subscribers. It is easily deployable, dynamic and flexible. **RabbitMQ**: RabbitMQ is a popular message-oriented middleware server using the AMQP protocol. **Direct** **Exchanges:** Routes messages with a routing key equal to the routing key declared by the binding queue.

**Fanout Exchanges**: Routes messages to all bound queues indiscriminately. If a routing key is provided, it will simply be ignored. **Topic** **Exchanges**: Routes messages to queues whose routing key matches all, or a portion of a routing key. **Queue**: where messages are stored on the broker. **Spring Boot** makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". Create a full application in one executable JAR. Create stand-alone Spring applications. Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files). Provide opinionated 'starter' dependencies to simplify your build configuration. Automatically configure Spring and 3rd party libraries whenever possible. Provide production-ready features such as metrics, health checks and externalized configuration. Absolutely no code generation and no requirement for XML configuration.

**Autoconfiguration**: @Import to include @Configuration files – @ImportResource can include XML config files.

**Opinionated View**: The official documentation says Spring Boot takes “an opinionated view of the Spring platform and third party libraries so you can get started with minimal fuss”.

@Configuration @ComponentScan @EnableAutoConfiguration @EnableWebSecurity public class Application {public static void main(String[] args) {SpringApplication.run(Application.class, args);

**For Client Side: Rest Template:**  @Service  
public class PersonServiceProxy implements PersonService { @Autowired private RestTemplate restTemplate; private final String personUrl = "http://localhost:8080/person/{id}";

private final String pplUrl = "http://localhost:8080/person/"; @Override public Person get(Long id) {  
 return restTemplate.getForObject(personUrl, Person.class, id);} @Override  
 public List<Person> getAll() {ResponseEntity<List<Person>> response = restTemplate.exchange(pplUrl, HttpMethod.*GET*, null, new ParameterizedTypeReference <List<Person>>() { }); return response.getBody();} @Override public Long add(Person p) {  
 URI uri = restTemplate.postForLocation(pplUrl, p); if (uri == null) {return null; }  
 Matcher m = Pattern.*compile*(".\*/person/(\\d+)").matcher(uri.getPath()); m.matches();  
 return Long.*parseLong*(m.group(1));} @Override public void update(Person p) {  
 restTemplate.put(personUrl, p, p.getId()); } @Override public void delete(Long id) {  
restTemplate.delete(personUrl, id); } }

**Controller class**

@RequestMapping(value = "/cars", method = RequestMethod.GET) public String getAll(Model model) {model.addAttribute("cars", carService.getAll());return "carList";}@GetMapping(value = "/addCar") public String addCar(@ModelAttribute("car") Car car) {return "addCar";} @PostMapping(value = "/addCar")public String add(@Valid Car car, BindingResult bindingResult, RedirectAttributes redirectAttributes) { if (bindingResult.hasErrors()) {return "addCar";} else {

carService.add(car); return "redirect:/cars";}}@GetMapping(value = "/cars/{id}")public String get(@PathVariable Long id, Model model) { model.addAttribute("car", carService.get(id).get());

return "carDetail";}@PostMapping(value = "/cars/{id}")public String update(@Valid Car car, BindingResult bindingResult, RedirectAttributes redirectAttributes) {if (bindingResult.hasErrors()) {

// car.id already set by binding return "carDetail";} else {carService.update(car);return "redirect:/cars";}} @PostMapping(value = "/cars/delete") public String delete(Long carId) {

carService.delete(carId); return "redirect:/cars";}

**For Server Side: Service class**

@Service@Transactional **public class** PersonServiceImpl **implements** PersonService {  
@Autowired **private** PersonRepository **personRepository**; @Override **public** Person get(Long id) {  
 **return personRepository**.getOne(id);} @Override **public** List<Person> getAll() { **return personRepository**.findAll();}@Override **public** Long add(Person p) {**personRepository**.save(p);  
**return** p.getId(); }@Override **public void** update(Person p) { **personRepository**.save(p); }@Override  
**public void** delete(Long id) {**personRepository**.deleteById(id); }}

**Controller class** RestController **public class** PersonController {@Autowired **private** PersonService **personService**; @GetMapping(value = **"/person/"**) **public** List<Person> getAll() { **return personService**.getAll(); } @GetMapping(value = **"/person/{id}"**) **public** Person get(@PathVariable **long** id) { **return personService**.get(id); } @PostMapping(value = **"/person/"**) **public** Long save(@RequestBody Person person) { **return personService**.add(person);} @PostMapping(value = **"/person/redirect/"**) **public** RedirectView post(@RequestBody Person person) { **long** id = **personService**.add(person); **return new** RedirectView(**"/person/"** + id);}@PutMapping(value = **"/person/{id}"**) **public void** put(@PathVariable **long** id, @RequestBody Person person) { **if** (id != person.getId()) { **throw new** IllegalArgumentException(); } **personService**.update(person);} @DeleteMapping(**"/person/{id}"**) **public void** delete(@PathVariable **long** id) { **personService**.delete(id); }}

**Spring Form:form:**

<form:form modelAttribute="user" action="../cars/${car.id}> <form:input path="password"/> <**form:select path="authorities" itemLabel="name" itemValue="id" items="${**authorities**}"**/>

<**c:forEach var="car" items="${**cars**}"**> <**tr**> <**td**>**${**car.make**}**</**td**> <**td**>**${**car.model**}**</**td**>  
<**td**>**${**car.year**}**</**td**> <**td**>**${**car.color**}**</**td**> <**td**><**a href="cars/${**car.id**}"**>edit</**a**></**td**> </**tr**>  
</**c:forEach**>

<a href="<c:url value="/test.html" var="testvar" />">TEST</a>

<c:out value="${testvar}"/>

**ErrorMessage.Properties**: Size.state = State must have two characters. Size.firstName = Size of the {0} must be between {2} and {1}. NotEmpty= {0} field must have a value. password.Size = {0} must have a length between {2} and {1}. firstName = First Name lastName = Last Name nickname = Nick Name birthdate = Birth Date

**AOP:** @Aspect @Component public class AopDemo{

@Around(**"execution(\* edu.mum.cs544.bank.service.\*.\*(..))"**) **public** Object time (ProceedingJoinPoint call) **throws** Throwable {StopWatch sw = **new** StopWatch(); sw.start(call.getSignature().getName()); Object retVal = call.proceed(); sw.stop();  
 **long** totaltime=sw.getLastTaskTimeMillis(); System.***out***.println(**"Time to execute "**+call.getSignature().getName()+**" = "**+totaltime+**" ms"**);**return** retVal;}

@After("execution(\* edu.mum.cs544.EmailSender.sendEmail(..)) && args(email, message)")

public void log(JoinPoint joinpoint, String email, String message) { System.out.println(new Date() + " method= "+joinpoint.getSignature().getName() + " email address= "+ email + " message= " + message); IEmailSender emailSender = (IEmailSender) joinpoint.getTarget(); System.out.println("outgoing mail server = "+emailSender.getOutgoingMailServer());} }

**To run AOP:** @Configuration @ComponentScan("edu.mum.cs544.bank") @EnableAspectJAutoProxy

public class Config { }

**Spring Boot run**: @SpringBootApplication public class SpringbootwebjspdemoApplication {

public static void main(String[] args) {SpringApplication.run(SpringbootwebjspdemoApplication.class, args); }}