|  |  |
| --- | --- |
| **SPRING CORE - CONCEPTS – THEORY - PPT** | |
|  |  |
| * Spring boot uses spring behind the scenes, has embedded server – tomcat. * Can be run standalone using – jar file * Deployed using war file – bunch of jar files | Inversion of control –   * Giving the control of creating and managing object. * injecting object dependencies to spring. |
| Maven –   * Based on dependencies – maven will download the jar files. (spring, hibernate, apache, JSON). * Build management and dependencies * Jar files - Local repo, remote repo. * POM.xml – * Config file of pjt   + GAV- Project name, version etc   + Dependencies - List of projects depending on – spring, hibernate…etc – GAV of other projects.   + Custom tasks to run - Junit reports. | Dependency Injection –   * Constructor – required dependencies - @Autowired on constructor * Setter – optional dependencies - @Autowired on setter * Field – not recommended – makes code harder for unit testing – why ?,how?.   AutoWiring –   * Spring will look for a class that matches by type – class or interface and it is injected automatically. |
| File structure –   * Src/main/java – java source code * Src/main/resources – properties / config files * Src/main/webapp – JSP files, web.xml, xml files, web config files * Src/test – unit testing code * Target – compiled code - .class files * Src/main/resources/static – html , css , js , images ,……etc * Src/main/resources/static/templates – * Management.endpoints.web.exposure.include = health , info | @Qualifier –   * In controller class, in constructor argument - @Qualifier(“name of the class , bean which needs to be picked up”). * Mark it as @Auto wired * Call it in one of the controller method * Higher priority , more specific than @primary * Along with dependency injection – class name – first character lowercase   @Primary –   * In the @component class * Just call the method in the controller class * The primary class object is called for it’s behaviour |
| Application Properties –   * Variable values   Dependencies   * Spring-boot-starter-web * A collection of maven dependencies grouped together * Dev tools – auto re-start the application * Actuator – application health, metrics – by exposing end points - /health, /info ,/beams , /mappings , ….etc in app.properties file. * Starter-security – app.properties – spring.security.user.name=scott * Spring.security.user.password=tiger * To secure endpoints * Exclude end points | Lazy Initialization – on the class @Lazy   * By default when the application starts all beans are initialized – all objects are created of @Component class * Here a bean will be initialized only when required for dependency injection, explicitly requested. * Is added on a class * Injecting means –the object is injected to controller and then the objects are called for their state / behaviour. * In app.properties – spring.main.lazy-initialization=true – lazy initialize all beans. * When end point is accessed – spring will determine the dependencies – initialize the dependencies and then creates instance of rest controller and in jects the bean. |
| Annotations -  @Value (${coach.name})  @Component – marks the class as spring bean (java object managed by spring)   * Makes bean available for dependency injection * Scan java classes, automatically register the beans in spring container.   @Autowired – in a controller   * On a constructor * Give the interface / class reference as arg * Assign it to a local variable * Now since interface is implemented by a class marked as @Component – available for dependency injection – it is automatically injected by the local variable   @RestController –   * Using local variable call the method – here automatically the object picked up is only one – marked as @Component.   @SpringBootApplication –   * Auto configuration, component scanning * @EnableAutoConfiguration – auto – configuration support * @ComponentScan – component scanning of current, sub packages, * @Configuration – register extra beans with @Bean * Creates application context, registers all beans, starts tomcat. | Bean scopes –   * Singleton – only one instance / object created. * This one bean will be injected everywhere – all dependencies will access this bean. * In the @component class we can mention @Scope(ConfigurableBeanFactory.SCOPE\_SINGLETON) * PROTOTYPE – new bean for each container request * Request – new for new http request * Session – new for new session * Application – Web app servlet context   Bean LifeCycle –   * Custom code during bean initialization * Custom code during bean destruction * Inside @Component class * Annotate for a method - @PostConstruct * This executes before a bean is initialized * @PreDestroy – executed before object is de-referenced   Configuration –   * @Configuration – on a class * @Bean on a method – returns the object of the class * Method name = bean ID * @Qualifier – give the method name, do not give the @configuration class name * To make third party class available for spring framework * Use third party class as bean * Inject this bean into controller and use the behaviours of the bean. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**HIBERNATE - JPA**

|  |  |
| --- | --- |
| Hibernate -   * Framework to persist / save java objects into database * ORM - Mapping between class and table * To minimize jdbc code * Write methods – instead of queries * Implements JPA | JPA – Jakarta persistence API   * API for ORM * Defines interfaces – * You need to give implementation * entityManager |
| * Uses jdbc for all database communications * @GeneratedValue(strategy=GenerationType.IDENTITY) | Entity Manager methods – to create queries   * Persist(entity object) – to save * Find (entity class, id(pk)) – to find * Merge(entity object) – to update * Remove(entity object) – to delete * createQuery(“query”,student.class) * TypedQuery |
| Entity class –   * Java class @Entity which is mapped to a database table – fields to columns * A public no-arg constr is a must | ID Generation Strategies   * AUTO * IDENTITY * SEQUENCE * TABLE * UUID |
| DAO –   * Needs JPA entity manager for saving / retrieving entities * Jpa entity manager needs a data source * Spring creates – Jpa entity manager, data source using app.properties * Inject EM into DAO. | * Entity manager object is created * On that call persist(object) * In save impl – paste the above line. * DAO-save-EM-persist * DAO is calling EM object * @Transactional |
|  | |
| STEPS –   * Create entity * Define DAO interface * Define DAO implementation – inject EM * Update main app | * @Repository – for registering DAO implementations, |
| JPQL – in queries give capitals for entity   * Uses entity name, entity fields * TypedQuery<Student> theQuery = theEntityManager.createQuery("FROM Student" , Student.class); * Return theQuery.getResultList();   Write queries for –   * Delete an entity * Delete based on condition * Delete all | Look at update one more time   * Only argument – accepts student * int rows = theEntityManager.createQuery("delete from Student where id=30").executeUpdate(); * return rows; |
|  |  |
|  |  |

|  |  |
| --- | --- |
| REST API’S - REPRESENTATIONAL STATE TRANSFER | |
| JSON – java script object notation |  |
| HTTP METHODS –   * POST, GET, PUT, DELETE   HTTP REQUEST MESSAGE –   * Request line – the HTTP command * Header variables – request metadata * Message body   HTTP RESPONSE MESSAGE –   * Response line – server protocol, status code * Header variables – response metadata * Message body | HTTP Response – status codes   * 100-199 – informational * 200-299 – successful * 300-399 – Redirection * 400-499 – client error * 500-599 – server error   Path Variables –   * /api/students/{studentId} * @PathVariable – inside the method argument * The case in the url request is the same as path variable argument id |
| DATA BINDING –   * JSON (calls getters) – JACKSON – (calls setters) - Java POJO * JSON data being passed to REST controller is converted to POJO. * Java object being returned from REST controller is converted to JSON. | * Response Entity is a wrapper for the HTTP response object   + http code   + http body   + http headers |
| Exception Handling –   * Custom error response class – status , message , timestamp – getters , setters , constrs. * Custom exception class – StudentNotFoundException extends RuntimeException – constr – message * In controller check and throw the exception – still it is not handled. * Add exception handler method in rest controller itself * Exception handler method - @ExceptionHandler * This method will return response Entity<ERRORRESPONSECLASS> * In the argument get the StudentNotFound object * You’ll set message from the StudentNotFoundException class * Return the error object, * HttpStatus.NOT\_FOUND | Global Exception Handling –   * We need this for all controllers * @ControllerAdvice on a class * Pre-process requests to controllers * Post-process responses to handle exceptions * Create new StudentRestExceptionHandler class * Add the @ExceptionHandler method here |
|  |  |

|  |  |
| --- | --- |
| REST API DESIGN – 1HR | |
| REST API – 1 hr   * Identify the main resource / entity – noun – employee – give empty constructor for dependency injection – it is mandatory * Provide toString | Requirements   * Get a list of employees * Get a single employee by Id * Add a new employee * Update an employee * Delete an employee |
| Architecture – REST CONTROLLER – SERVICE – DAO   * Service layer – for custom business logic * Integrate data from multiple sources - DAO’S / repositories | @Entity  @Table(name="employee")  @Id  @GeneratedValue(strategy=GenerationType.IDENTITY)  @Column(name="id") |
| Add –   * @Repository on class * @Autowired on constructor | * Use http methods to assign action on resource * Use plural form – employees * Do not use verbs in url pattern * Just use – “employees” – use http methods to assign actions * url pattern remains the same but the methods will be different. |
| * Component-scanning will register the service implementaton | Methods which contains arguments –   * findById – int id – find(entity.class, id) * update – entity – merge(entity object) * deletebyid – int id – find , remove(entity) |
| * @PathVariable – findById , Delete * @postMapping – responseEntity - @RequestBody of entity – in the argument * Return – re.status(httpstatus.created).body(entitytype object) | Return – entity – findById , Update - merge  Void – delete-remove, persist ,  List - findAll |
| ResponseEntity | Headers |
|  |  |
|  |  |

|  |  |
| --- | --- |
| SPRING DATA JPA – 1 hr   * All methods – pagination, sorting | |
| * Give entity type, primary key – get crud methods * Create DAO – give entity type, primary key – get crud methods * Spring data jpa provides - JpaRepository interface, exposes methods | STEPS –   * PUBLIC INTERFACE EmployeeRepository extends JpaRepository<Employee, Integer> * Use employeeRepository to inject as dependency in the place of dao in service implementation layer * In findByiD method – make it reference to localvariable and convert it to entity class and return the entity object |
|  | Take a look at put and post mapping   * Path variable, requestbody |
| Spring data REST   * Using existing jpa repository – get all the rest api crud features for free   STEPS –   * Spring data rest will scan the project for jpaRepository * Expose rest api’s for each entity type for jpa repository * End points will be created based on entity type * First character of entity type will be lower and in a plural form – add “s” at the end | Development process –   * Add spring data rest dependency * That’s all   Only 3 things required –   * Entity * Jpa repository * Spring data rest dependency   In the files –   * Entity * EmployeeRepository |
| Customize the url  Spring.data.rest.base-path=/api |  |
| Configuration | @RepositoryRestResource(path=”members”)  Localhost:8080//members   * Spring.data.rest.base-path * Spring.data.rest.default-page-size * Spring.data.rest.max-page-size |
| pagination | Default – 20 pages  Navigate to different pages using –  Localhost:8080/employees?page=0  Localhost:8080/employees?page=1 |
| Sorting | Localhost:8080/employees?sort=lastName  Localhost:8080/employees?sort=firstName,desc  Localhost:8080/employees?sort=lastNmae,firstName,asc |
|  |  |
|  |  |

|  |  |
| --- | --- |
| **MAPPINGS** | |
| * Primary key , foreign key (column in one table which refers to primary key in another table) * cascade – apply the same operation to related entities | ONE TO ONE   * create instructor detail table * create instructor table – FK – ins\_detail * create instructor detail class * create instructor class |
| Fetch types – Eager loading , Lazy loading   * Eager – retrieve everything * Lazy – retrieve on request   Mention at @OneToMany or similar kind of relationships not in join column…etc  Session scope | @OneToOne  @JoinColumn(name="instructor\_detail\_id")  private InstructorDetail instructorDetail;  Inside Instructor class which has foreign key to primary key of instructor detail |
| Entity Life cycle   * Detach * Merge * Persist * Remove * Refresh | Cascade types – if applied to entity, it will be applied to related entity also   * Persist * Remove * Refresh * Detach * Merge * all |
| ONE TO MANY | @OneToOne(cascade=CascadeType.ALL) |
| Instructor – courses  Courses table – contains FK to instructor\_id  The many table contains foreign key to the one table – to prevent redundancy in the one table | * Give methods in dao interface – give implementation – give command line runner / controller – save, findById – of the Instructor |
| Course class  @ManyToOne  @JoinColumn(name="instructor\_id")  private Instructor instructor;  inside the course class since it contains FK to One | Bi-directional   * Inside Instructor\_detail class * add new field to reference Instructor * getter/setter methods for instructor * @OneToOne(mappedBy="instructorDetail") * private Instructor instructor; |
| Instructor class   * Add private List<course> courses – as many courses are there for a instructor * @OneToMany(mappedBy="instructor")   + private List<Course> courses; | * MappedBy – tells Hibernate – look at the instructorDetail property in the Instructor class * Use info from @JoinColumn from Instructor class |
| * Look at the instructor property in the Course class * Use information from the Course class @JoinColumn * To help find associated courses for instructor | * Give the findInstructorDetailById – in this get the instructor – by calling getter of the instructor field created |
| Give add() method inside instructor class – which creates arrayList and adds courses inside it and assigns the course to this instructor | Cascading , mappedby(primaryKey of the present class) , fetch types – define them on relationships - @OneToOne…etc  @JoinColumn – ( foreign key ) |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| API DESIGN BASED ON DOCUMENTATION – 1hr | |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| ARCHITECTURES, CONTAINERS | |
| SERVLETS |  |
| SPRING BOOT |  |
| MVC |  |
| REST |  |
| DATA – JPA FLOW |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| **CODING TASKS** | |
| Actuator | management.endpoints.web.exposure.include=health,info  management.info.env.enabled=true |
| Spring security | spring.security.user.name=test  spring.security.user.password=bro |
|  |  |
|  |  |
|  |  |