SpringBoot Project Setup: -

<http://techbus.safaribooksonline.com/video/programming/9781491959749>

<https://www.dineshonjava.com/spring-boot-interview-questions-and-answers/>

<http://www.softwaretestinghelp.com/restful-web-services-interview-question/>

* Create an spring boot application by using initializer site <http://start.spring.io/> [Proffered way]
* Or we can use STS and select spring startup project. Internally STS also use this spring boot starter site only to create application that’s why I personally don’t prefer it.
* Or create a maven/gradle project by any your favorite site and add spring boot specific jars that's it.

compile('org.springframework.boot:spring-boot-starter-web')

compile('org.springframework.boot:spring-boot-starter-data-jpa')

testCompile('org.springframework.boot:spring-boot-starter-test')

runtime('org.springframework.boot:spring-boot-devtools')

* Use devtools runtime dependency to automatically dependency for hot deployment

Logging:-

* Spring boot use slf4j API for logging.
* Default logging label is info, we can change it as below

logging.label.root=debug

logging.label.org.springframework.web=warn # change framework label logging as warn

logging.file=bizops.log

looging.path=logs

* Advance logging
* Create logback-spring.xml file and create profiles for logging.
* Log will be printed as per the profile

Actuator:-

compile('org.springframework.boot:spring-boot-starter-actuator')

* Spring boot provides some readymade services (end point urls) to monitor and manage your application.
* By default these services runs on the same host and port where our application run.
* Default actuator: mapping, health, logfile, autoconfig, bean, info, env, error and many more
* We can change default configure of actuators also like

endpoint.autoconfig.path=/ac

endpoint.autoconfig.enabled=false # default is true

endpoint.context-path:=/actuator # to manage actuator path

http://localhost:8080/actuator/health # earlier one was http://localhost:8080/health

endpoint.health.sensitive=true // in this case it returns only {status:up}. by default is false and returns many more info

* we can customize existing actuator
* Example: create a class by which implements actuator interface (HealthIndicator) and overwride it's method (health)
* We can write our own custom actuator by implementing EndPoint interface and overwires it's method

Gradle:- gradlew clean build bootRun and many more

* *Install builsShip plugin in eclipse for gradle support (build.gradle).*
* *gradle.org.*
* *gradle syntax are similar to groovy syntax and it becomes easy to use.*
* *It has diff tasks and tasks have defined in different plugin*
* *Gradle* internally *use maven pom and maven repository in order to pull plugins and dependency jars*
* *Gradle creates an execution jar file by jar task, similarly it has war task to generate war file, in case of war exclude default tomcat server*
* *Why use gradle not maven, gradle project itself contains gradlew file which allow us to run different gradle command and hence no need to install gradle separately and it solves version problem. It's managed by gradle wrapper task defined in build.gradle file*

*Spring boot application deeper look:-*

* Add one main (Application.java) class and controller class

@SpringBootApplication

public class Application {

public static void main(final String[] args) throws InterruptedException {

SpringApplication.run(Application.class, args)) {

} }

* *And rest controller*

@RestController

@RequestMapping("/1.0/tasks/")

public class TasksResourceController {

}

* Here we have used @SpringBootApplication annotation which includes @ComponentScan, @Autoconfiguration, @Configuration
* @ComponentScan("packagename") # default all the packages/subpackages are scanned where Application.java resides
* @Autoconfiguration : provides some initial configuration for spring boot application like by default spring boot use tomcat container and is it's 80 port etc, later on we can overwrite it. To provide default configuration springboot app includes a spring-boot-autoconfiguration.jar
* @Configuration is used to configure custom beans, all these beans are created and registered in DI container

@Configuration

ApplicationConfiguration{

private String dbDriver;

@Bean(name="message") #by default it's name would be function name

public Strig message(){

return "Hello SpringBoot"

}

@Bean

public void setDBConfiguration(@Value("$(dbDriver)") String dbDriver){

this.dbDriver = dbDriver

}

}

@Autowired

String message;

* application.property (applicaiton.yml) file is used to overrides it's default configuration , we can pass application arguments to overwrite default configuration

applicaiton.properties #also used for adding some new configuration, and for profiling

server.address=amplify

server.port=8080

server.context-path=/bizops

version=1.4

* There are three ways to pass the values to properties.
* By passing program argument(highest priority)
* By configuration file (applicaiton.properties) or application.yml (used after program argument)
* providing default value in annotation like @Value("${name:defaultName}") (least priority)

@Value("${name:defaultName}")

String name;

Initial configuration:-

* *Implement CommandLineRunner/* *ApplcationRunner interface and overwrite it's public void run(String ... args) method*

Profile:- --spring.profiles.active=test

* SprigBoot Profile: enable us to run our app with multiple configuration like testing, development
* Create application.<profile>.properties file with respect to each profile. ex. applicaiton.test.properties
* If we do not pass the program argument for profile then default applicaiton.priorities file is used
* select any one of them either by defining spring.profiles.active=dev in pplicaiton.properties
* we can define all profiles in a single application.properties file by creating section

SpringBoot with database:-

* *Add schema.sql in resource folder. This file holds all DDL queries and executed just after the application starts up.*
* *Data.sql is used for DML.*

*spring.datasource.schema=classpath:myschema.sql # overwride default schema.sql*

* *h2 is in memory db and can be used in production*
* *Or use external db say oracle 12c.*

*compile('com.oracle.jdbc:ojdbc7:12.1.0.2',)*

*compile('org.springframework.boot:spring-boot-starter-data-jpa')*

*spring.datasource.url=jdbc:oracle:thin:@localhost:1521/orclPDB*

*spring.datasource.username=business*

*spring.datasource.password=amplify*

*spring.jsp.database.dialect=org.hibernate.dialect.OracleDialect/MySQLDialect*

*spring.jpa.hibernate.ddl-auto=none or create, create-drop,update, validate, default is create: create if does not exist*

*Create an entity class by using @Entity anotation*

*@Entity*

*class Employee{*

*@Id*

*int id;*

*}*

* *Acquire an object of EntityManager by using @Autowired annotation. later on it can be used to create queries or criteria*

*@Autowired*

*EntityManager em;*

*#em.getCriteriaBuilder();*

*#em.createQuery()*

* *But it's not a good approach to use entity manager, there is much better approach to use Spring provided repositories interfaces*

*public interface EmplyeeRepository extends JpaRepository<Employee, Integer>*

Cache:- compile(org.springframework.boot:spring-boot-cache)

* Add @CacheResult to any function. The function is called only one time with respect to unique parameter and the result is cached

@CacheResult(cacheName ="price")

public int getPrice(String itemId){....}

* Next time when the function is called with same parameter fun's body is not executed and the result is serverd from cache.
* Above is default way provided by spring boot for cache. If we want more powerful cache we can switch to some other cache provided like EH cache (Populer)

compile(net.sf.ehcache.ehcache)

spring.cache.ehcache.config=classpath:ehcache.xml

* We can get cache from CacheManger also

@Autowired

CacheManager cacheManager;

Cache priceCache = cacheManager.get("price"); # here price is CacheName

priceCache.get("funcParam") # here funcParam is the function Parameter under which data is cached

Security:-

* I have used Identity Cloud Service (IDCS) has some feature, like isAccountExpired, isAccountLocked, isAccountActive etc, and if it's active then return a client id token, and next url will come with token to authorize the resources

SpringBoot Test:-

* The @SpringBootTest annotation tells Spring Boot to go and look for a main configuration class (one with @SpringBootApplication for instance), and use that to start a Spring application context

@SpringBootTest

@RunWith(SpringRunner.class)

# use TestRestTemplate to make rest call

@Autowired

private TestRestTemplate restTemplate;

* We do use cucumber to UI automation and backend integration testing

Protocol:-

* GET: get some get from server, does not have body, not secure, req param is appended in url itself
* POST: Used to create or update the recored/date, it has body, # Genereally create a data/record
* PUT: Used to create and udate data/record, # geneally used ot update existing data/record
* 1XX :information # request has received, it's being processes, processes etc. exam if we are uploading a medeia (vedio/audo) in multipart
* 2XX :success
* 3XX: redirect
* 4xx: client side error
* 5XX: server side error.

Error Handling and validation:-

* I believe for error handling we should create our own approach rather than using the framework and any provided.
* Use our own approach and get more hold on message and code etc

class enum ErrorType{

GENERIC\_ERROR("0001", "The system is unable to complete the task. Contact your supervisor."),

}

UserNotFoundException extends RunTimeException {

private final String description;

private final String errMsgKey;

private final String errorCode;

UserNotFoundExceptionType(Error error){

super(error.getErrorMessage());

this.errMsgKey = code.getErrMsgKey();

this.errorCode = code.getErrCode();

this.description = "some des";

}

* Implementation of validation######
* Create a validator calss and also define a validate function with respect to request body/param,
* Inside the validator function validates the request and if it has issue then throw the error from there only.

API Versioning:-

* add the version in url like http://application/v1/ but is not a good approach to change the urls for each version
* We can read the version from property type or from the build script (build.gradle or pom.mvn)

like http://application/"${version}"/

add the @RequestMapping("/1.0/") at the top of controller class

* By adding header: Accept: application/api-version2+json

Restful: - swagger tool to generate Rest EndPoint from .yaml or .json files

* Representational State Transfer
* Rest is an Architectural Style, not any standard
* Rest assumes that everything is a resource that is identified by unique URL
* Each resource transfers its representational state from server to client
* Like when we hit any URL and say it returns response in some representational state
* This state depends on mime type header, could be plain text, image, json, xml html, pdf, .doc, etc
* It's uses only HTTP protocol
* It's works on stateless mechanism