SpringBoot

💡 Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

💡 Spring Boot **helps developers create applications that just run**. Specifically, it lets you create standalone applications that run on their own, without relying on an external web server

💡 It has Several Modules IOC, AOP, DAO, Context, ORM, WEB MVC etc…

- Spring doesn't support JSP, we can add support by adding dependency,

```
<dependency>
   <groupId>org.apache.tomcat</groupId>
   <artifactId>tomcat-jasper</artifactId>
   <version>10.1.4</version>
</dependency>
```

```
server.port = 9090
spring.jpa.hibernate.ddl-auto = update
spring.datasource.url = jdbc:mysql://localhost:3306/demo
spring.datasource.username = root
spring.datasource.password = db006
spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver
```

▼ Intro

Spring Features:

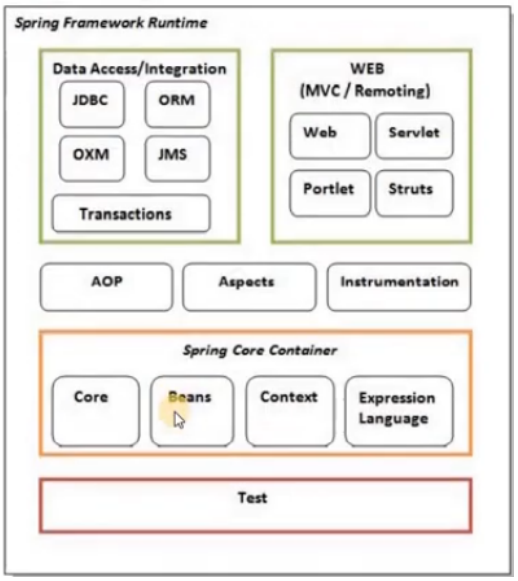
1. POJO
2. Dependency Injection
3. MVC
4. REST
5. Security
6. Batch
7. Data
8. AOP etc. . .

Advantages:

- Predefined Templates - Templates for JDBC, Hibernate, JPA etc . . .
- Loose Coupling - Achieved by Dependency Injection (DI)
- Easy to Test - DI makes it easier to test.
- LightWeight - Achieved by POJO Implementation
- Fast Development - Easy development of JavaEE application

- Powerful Abstraction - JMS, JDBC, JPA & JTA
- Declarative Support - Provides support for Caching, Validation, Transactions & Formatting
- Reduces Boiler Plate Code

Spring Architecture / Spring Modules:



▼ Features

- 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

▼ REST (REST = REpresentational State Transfer)

- Data Received from the server depends on url
- URL represents a particular page
- Data formats,
  - JSON
  - XML
  - XML is bulky, hence JSON is popular
- Spring boot dev tools, helps us to save code changes without stopping the process (***it's a dependency***)
- Client can see data by,
  - URL
  - By HTTP method types using Postman, Swagger etc . . .
    - Request Method Types,

Method	*Path*	*Description*
GET	/topics	Gets all topics
GET	/topics/id	Gets the topic
POST	/topics	Create new topic
PUT	/topics/id	Updates the topic

Method	Path	Description
DELETE	/topics/id	Deletes the topic

- *@RestController* handles REST Operations
- *@RequestMapping* maps requests to handler classes or handler methods
- *ResponseEntity<?>* carries data from server to client
  - server - program
  - client - postman/swagger/browser
- *@EnableSwagger2* enables us to *test REST app* using any browser.
  - add this annotation to main class.

▼ SpringBoot JPA MVC H2 Example

▼ Save

> 💡 *libraries required :* ***web, mysql connector, JPA***

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/demo
spring.datasource.username=root
spring.datasource.password=db006
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

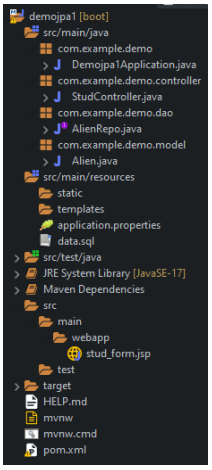
```
package com.example.demo.dao;

public interface AlienRepo extends CrudRepository<Alien, Integer>{

}
```

```
package com.example.demo.controller

@Controller
public class StudController {

  @Autowired
  AlienRepo repo;

  @RequestMapping("/")
  public String stud() {
    return "stud_form.jsp";
  }

  @RequestMapping("sub_stud")
  public String addStud(Alien alien) {
    repo.save(alien);
    return "stud_form.jsp";
  }
}
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
  <form action="sub_stud">
    <input type="text" name="name"  placeholder="name"/><br>
    <input type="text"  name="id"  placeholder="id"/> <br>
     <input type="text" name="marks"  placeholder="marks"/><br>
     <input type="submit" name="submit" />
  </form>
```

```
package com.example.demo;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class Student {
  @Id
  private int id;
  private String name;
  private int marks;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getMarks() {
    return marks;
  }

  public void setMarks(int marks) {
    this.marks = marks;
  }
}
```

```
</body>
</html>
```



- ***Controllers*** should be saved in ***controllers folder***.
- ***Model/Entities*** should be saved in ***model folder***.
- ***EntityRepos/DAO*** (***for CRUD operations***) should be saved in ***dao folder***.
- ***.jsp*** files should be saved in ***src/main/webapp folder***.

▼ *Fetch*

```
@Controller
public class StudController {

  @Autowired
  AlienRepo repo;

  @RequestMapping("get_alien")
  public ModelAndView showStud(@RequestParam int id) {
    ModelAndView mv = new ModelAndView("show_stud.jsp");
    Alien alien = repo.findById(id).orElse(new Alien());
    mv.addObject(alien);
    return mv;
  }
}
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>${alien}
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
  <form action="get_alien">
    <input type="text" name="id" placeholder="enter id"/>
    <input type="submit">
  </form>
</body>
</html>
```

▼ *Custom Queries*

▼ Single Class

```
public interface AlienRepo extends CrudRepository<Alien, Integer> {
  List<Alien> findByMarks(int marks);

  List<Alien> findByIdGreaterThan(int id);

  @Query(value = "from Alien where name like '%h%' ", nativeQuery = true)
  List<Alien> findByNameHas();
}
```

- queries should be in ***JPQL (similar to HQL)***
- The methods ***findMyMarks()***, ***findByIdGreaterThan()*** not needed definition, since it is managed by spring framework and ***marks*** & ***id*** are the ***properties of entity*** class.

```
@Controller
public class StudController {

  @Autowired
  AlienRepo repo;

  @RequestMapping("/")
  public String stud() {
    return "stud_form.jsp";
  }

  @RequestMapping("show_studs")
  public ModelAndView showStud(@RequestPar
    ModelAndView mv = new ModelAndView("sh
    Alien alien = repo.findById(id).orElse

    System.out.println(repo.findByMarks(10
    System.out.println(repo.findByIdGreate
    System.out.println(repo.findByNameHas(

    mv.addObject(alien);
    return mv;
```

```
  }
}
```

▼ Two Classes

```
@Entity
class Topic{

  @Id
  int id;
  String name;

  //getters

  //setters
}
```

```
@Entity
class Course{
  @Id
  int id;
  String name;
  Topic topic;

  //getters

  //setters
}
```

```
interface CourseRepo extends JpaRepository<Course, Integer>{
  public List<Course> findByTopicId(String topicId);
}
```

▼ *REST Example*

💡 Produce data from server called produces & if server is accepting data then it's called consumes.

▼ *Fetch Data as JSON*

💡 ***jackson-core*** library is ***responsible*** for converting ***java object*** into ***json***

```
@Controller
public class StudController {

  @Autowired
  AlienRepo repo;

  @RequestMapping("/")
  public String stud() {
    return "stud_form.jsp";
  }

  @RequestMapping("/alien")
  @ResponseBody
  public List<Alien> getAliens() {
    return repo.findAll();
  }

  @RequestMapping("/alien/{id}")
  @ResponseBody
  public Optional<Alien> getAlien(@PathVariable("id") int id) {
    return repo.findById(id);
  }
}
```

```
public interface AlienRepo extends JpaRepository<Alien, Int
  List<Alien> findByMarks(int marks);

  List<Alien> findByIdGreaterThan(int id);

  @Query("from Alien where name like '%h%' ")
  List<Alien> findByNameHas();
}
```

▼ *Postman*

💡 Used to send API requests, fetch data and lot more. (Available for windows as app.)

▼ *Content Negotiation*

💡 Providing data to client in different formats (***i.e; JSON, XML etc . . .***) is called Content Negotiation

```
<dependency>
   <groupId>com.fasterxml.jackson.dataformat</groupId>
   <artifactId>jackson-dataformat-xml</artifactId>
   <version>2.14.1</version>
</dependency>
```

```
@RequestMapping(path = ”/form-name”, produces = {”applica
```

▼ *Send Data To Server*

```
@RestController
public class StudController {

  @Autowired
  AlienRepo repo;

  @RequestMapping("/")
  public String stud() {
    return "stud_form.jsp";
  }

  @PostMapping(path="sub_stud" consumes={application/xml})
  public Alien addStud(@RequestBody Alien alien) {
    repo.save(alien);
    return alien;
  }

  @GetMapping("/alien")
  public List<Alien> getAliens() {
    return repo.findAll();
  }

  @RequestMapping("/alien/{id}")
  @ResponseBody
  public Optional<Alien> getAlien(@PathVariable("id") int id) {
    return repo.findById(id);
  }
}
```

▼ *Update Data in Server (real time)*

```
  @PutMapping(path="/aliens/{id}")
public Department update(@RequestBody Department department
                        ,@PathVariable Long id) {
    Department dept = repo.findById(id).get();
    if (!Objects.nonNull(dept)) {
      dept.setName(department.getName());
      dept.setAddress(department.getAddress());
      dept.setCode(department.getCode());
    }
    return repo.save(dept);
  }
```

```
@RestController
public class StudController {

  @Autowired
  AlienRepo repo;

  @PutMapping(path="/alien" consumes={application/xml})
  public Alien saveOrUpdateAlien(@RequestBody Alien alien)
    repo.save(alien);
    return alien;
  }
}
```

▼ *Delete Data in Server (real time)*

```
@RestController
public class StudController {

  @Autowired
  AlienRepo repo;

  @DeleteMapping(path="/alien/{id}")
  public String deleteAlien(int id) {
    Alien alien = repo.getOne(id);
    repo.delete(alien);
    return "deleted";
  }
}
```

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.14.1</version>
</dependency>
```

▼ Annotations

Annotation	*Use*
@Controller	It's used to mark a class as a web request handler.
@RequestMapping("/form-name*")*	This is used to map to the Spring MVC controller method.
@RequestMapping(path = "/form-name*", produces = {"*application/xml*"})*	server produces data only in xml format.
@RequestMapping(path = "/form-name*", conusmes = {"*application/json*"})*	server accepts data only in json format.
@PostMapping("/add_alien*") @PostMapping(path="*/add_alien*" consumes={"*applciation/xml*"})*	To specify method supports POST. (sends data)
@DeleteMapping("/get_alien/{id}*")*	To specify method supports Delete (deletes data)
@PutMapping(path="/update_alien*" consumes={"*applciation/xml*"})*	To specify method supports Update (updates data)
@GetMapping("/get_alien/{id}*")*	To specify method supports GET & it is default. (fetches data)
@RestController	It is ths combination of the *@controller* and *@ResponseBody* annotation.
@RequestBody	A request body is *data sent by the client* **to your API**
*@RequestParam("*attr_name*") @RequestParam*	Used to bind a web request parameter to a method parameter.
@Query("query in JPQL" *)*	Allows us to write custom queries.
*@PathVariable("*property_name*")*	Represents different kinds of parameters in the incoming request
*@RepositoryRestResource(collectionResourceRel = "*aliens*", path = "*aliens*")*	It is optional and is used to customize the REST endpoint.
@Bean	It is applied on a method to specify that it returns a bean to be managed by Spring context.
@EnableSwagger2	Enables us to *test REST app* using any browser by adding dependencies.
*@ExceptionHandler(*AlienNotFoundException.class*)*	Used to handle the specific exceptions and sending the custom responses to the client
@ControllerAdvice	Allows to handle exceptions across the whole application in one global handling component.
@Repository	Used to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects.
fun(*@ModelAttribute Employee employee*)	bind the form details object to Employee object
@Service	Used to indicate that they're holding the business logic.
@EnableWebSecurity	The `@EnableWebSecurity` is a marker annotation. It allows Spring to find (it's a `@Configuration` and, therefore, `@component`) and automatically aplly to the global `WebSecurity`.

▼ Lombok

💡 ***Lombok plugin should be added in pom.xml*** ***(*** Removes Bioler plate code ***)***

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
```

```
  <optional>true</optional>
</dependency>
```

Annotation	Use
@Data	Getters + Setters + ToString + RequiredArgsConstructor
@Setter	Generate Setters
@Getter	Generate Getters
@NoArgsContructor	Generate no args construtor
@AllArgsContructor	Generate all args construtor
@Builder	Applies Builder Design pattern to our class

▼ application.Properties to application.yaml

```
spring:
    datasource:
        url: jdbc:mysql://localhost:3306/demo
        driver-class-name: com.mysql.cj.jdbc.Driver
        password: db006
        username: root
    jpa:
        hibernate:
            ddl-auto: update
```

- It is widely used.
- Human readable.
- online tools are available to convert ***application.properties*** to ***application.yml***

▼ Adding Validation

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- check data sent by client
- client sends data in JSON format
- JSON is translated into java object in controller
- while translating we need to check data

Steps:

1. add validation dependency
2. use validation annotation in entity class
3. in controller use ***@Valid*** along with ***@RequestBody***
4. ***@Valid*** check the validation errors.
5. If validation fails, method arguement not possible exception is thrown.
6. Controller advice should handle those exception.

Annotation	Use
*@NotBlank (***message =" "** *)*	Field shouldn't be blank.
*@Length (*max=5, min=1*)*	Checks Length
*@Size (*max=5, min=1*)*	Checks Size
@Email	Checks Email Structure

Annotation	Use
@Positive	Checks positive or not.
@Negitive	Checks negitive or not.
@PositiveOrZero	Checks positive or zero.
@NegitiveOrZero	Checks negitive or not .

Annotation	Use
@Future	Checks date future or not.
@FutureOrPresent	Checks date is future or present.
@Past	Checks date past or not.
@PastOrPresent	Checks date is past or present.

Implementation :

```
@Entity
public class Department {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long Id;

  @Email
  private String name;

  private String address;
  private String code;
}
```

```
@PostMapping("/departments")
  public Department save(@Valid @RequestBody Department depart
    return service.save(department);
  }
```

▼ Exception Handling

- ***@ControllerAdvice*** is the class that handles all exceptions in application.
- In Controller Advice Class, we need to write ***@ExceptionHandler(AlienNotFoundException.class)*** annotation in each method.

```
class DepartmentServiceImpl{
@Override
  public Department fetchOne(Long id) throws DepartmentNotFoundException {
    Optional<Department> dept = repo.findById(id);
    if(dept.isPresent())
      throw new DepartmentNotFoundException("department ledhu ra ayya");
    return repo.findById(id).get();
  }
}
```

```
public class DepartmentNotFoundException extends Exception {
  public DepartmentNotFoundException(String message) {
    super(message);
  }
}
```

```
@ControllerAdvice
@ResponseStatus
public class RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler {
  @ExceptionHandler(DepartmentNotFoundException.class)
  public ResponseEntity<ErrorMessage> departmentNotFoundException(DepartmentNotFoundException exception,
      WebRequest request) {
    ErrorMessage message = new ErrorMessage(HttpStatus.NOT_FOUND, exception.getMessage());
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(message);
  }
}
```

```
package com.example.demo.entity;

import org.springframework.http.HttpStatus;

public class ErrorMessage {
  public HttpStatus getStatus() {
    return status;
  }
  public void setStatus(HttpStatus status) {
    this.status = status;
  }

  public String getMessage() {
    return message;
  }

  public void setMessage(String message) {
    this.message = message;
  }

  private HttpStatus status;
  private String message;

  public ErrorMessage(HttpStatus status, String message) {
    super();
    this.status = status;
    this.message = message;
  }

}
```

▼ Unit Testing (2:05)

💡 Junit5 & Mockito dependencies are needed.

🧪 Junit

▼ Service Layer Implementation

```
class DepartmentServiceTest {

 @Autowired
```

```
@Repository
public interface DepartmentRepository extends JpaRepository<
```

```
  private DepartmentService service;

  @MockBean
  private DepartmentRepository repo;

  @BeforeEach
  void setUp() {
    Department department = new Department(1L, "IT", "madanapalle","IT-03");
    Mockito.when(repo.findByName("IT")).thenReturn(department);
  }

  @Test
  public void whenValidDepartmentName_thenDepartmentShouldFound() {
    String deptName = "IT";
    Department dept = service.fetchByName(deptName);

    // here we dont want to hit database, so we have to mock the data.
    Assertions.assertEquals(deptName, dept.getName());
  }
}
```

```
  public Department findByName(String name);
}
```

```
public interface DepartmentService {
  public Department fetchByName(String name);
}
```

```
@Service
public class DepartmentServiceImpl implements DepartmentServ
  @Autowired
  private DepartmentRepository repo;

  @Override
  public Department fetchByName(String name) {
    return repo.findByName(name);
  }
}
```

▼ Repository Layer Implementation

```
@DataJpaTest
class DepartmentRepositoryTest {

  @Autowired
  private DepartmentRepository repo;

  @Autowired
  private TestEntityManager manager;   // (pre-defined)

  @BeforeEach
  void setUp() throws Exception {
    Department dept = new Department(1L, "IT", "tpt", "It-07");
    manager.persist(dept);
  }

  @Test
  public void whenFindById_thenReturnDepartment() {
    Department dept = repo.findById(1L).get();
    Assertions.assertEquals(dept.getName(), "IT");
  }
}
```

▼ Controller Layer Implementation

```
@WebMvcTest(DepartmentController.class)
class DepartmentControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private DepartmentService service;

  private Department dept;

  @BeforeEach
  void setUp() throws Exception {
    dept = new Department(1L, "IT", "mpl", "IT-10");
  }

  @Test
  void saveDept() {
    Department inputDept = new Department(1L, "IT", "mpl", "IT-10");
    Mockito.when(service.save(inputDept)).thenReturn(dept);
    mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate:"/departments")
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\r\n"
        + "  \"name\": \"IT\",\r\n"
        + "  \"address\": \"madanapalle\"\r\n}"))
      .andExpect(MockMvcResultMatchers.status().isOk());
  }
```

```
  @Test
  void fetchDept() {
    Mockito.when(service.fetchById(1L)).thenReturn(dept);
    mockMvc.perform(get(urlTemplate:"/departments/1")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath(expression:"$.name").value(dept.getName()));
  }

}
```

▼ Adding Loggers

```
@RestController
public class DepartmentController {
  @Autowired
  private DepartmentService service;

  private final Logger LOGGER = (Logger) LoggerFactory
                               .getLogger(DepartmentController.class);

  @GetMapping("/departments")
  public List<Department> fetch() {
    LOGGER.info("INSIDE ALL FETCH");
    return service.fetch();
  }
}
```

▼ Profiles

- The configuration is different for dev,test and prod environments.
  - To solve this problem, we use profiles.

```
spring:                                                          //active environment
    profiles:
    active: qa

management:
  endpoints:
    web:
      exposure:
        include: "*"
---                                                              //dev environment
spring:
    profiles: dev
    datasource:
        url: jdbc:mysql://localhost:3306/demo-dev
        driver-class-name: com.mysql.cj.jdbc.Driver
        password: db006
        username: root
    jpa:
        hibernate:
            ddl-auto: update
---                                                              //qa environment
spring:
    profiles: qa
    datasource:
        url: jdbc:mysql://localhost:3306/demo-qa
        driver-class-name: com.mysql.cj.jdbc.Driver
        password: db006
        username: root
    jpa:
        hibernate:
            ddl-auto: update
---                                                              //prod environment
spring:
    profiles: prod
    datasource:
        url: jdbc:mysql://localhost:3306/demo-prod
        driver-class-name: com.mysql.cj.jdbc.Driver
        password: db006
        username: root
    jpa:
        hibernate:
            ddl-auto: update
```

▼ Deploy app into Production

- We have to create our .jar files

```
mvn clean install
java -jar filename --spring.profiles.active=prod
```

▼ Actuator

> 💡 Actuator dependency needed

- used to monitor application after deployement (i.e; performance, health, memory, beans created . . .)

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

Custom Actuator Endpoint :

- @ReadOperation maps to Http.Get
- @WriteOperation maps to Http.Post
- @DeleteOperation maps to Http.Delete

```
@Component
@Endpoint(id = "features")
public class FeatureEndPoint{
  private final Map<String, Feature> featureMap = new ConcurrentHashMap<>();

  public FeatureEndPoint(){
    featureMap.put("Department",    new Feature(isEnabled: true));
    featureMap.put("Authorization", new Feature(isEnabled: false));
    featureMap.put("Authenticate",  new Feature(isEnabled: false));
  }

  @ReadOperation
  public Map<String, Feature> features(){
    return featureMap;
  }

  @ReadOperation
  public Feature feature(@Selector String featureName){
    return featureMap.get(featureName);
  }

  @Data
  @NoArgsConstructor
  @AllArgsConstructor
  private static class Feature{
    private boolean isEnabled;
  }

}
```

Exclude Actuator Endpoint

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
        exclude: "env,beans"
```

▼ Spring Data Rest

💡 We dont need any controller to do CRUD Operations

💡 Rest Repositories, web services dependency required (***available in spring starter project itself***)

```
package com.example.demo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

@Entity
public class Alien {
  @Id
  @GeneratedValue
  private int id;
  private String name;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

```
package com.example.demo.model;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.Repositor

@RepositoryRestResource(collectionResourceRel = "aliens", path
public interface AlienRepo extends JpaRepository<Alien, Intege

}
```

```
spring.jpa.hibernate.ddl-auto = update
spring.datasource.url = jdbc:mysql://localhost:3306/demo
spring.datasource.username = root
spring.datasource.password = db006
spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver
```

▼ Spring Security

💡 Provide default user-name & pswd. ***Username :*** user, ***pswd :*** generates new pswd each time app runs, we can set our own pswd & user.

```
//application.properties file
spring.security.user.name = user_name
spring.security.user.password = pswd
```

▼ Intro

💡 Provides application level security

- It is a Application Security Framework & provides,
  - ***Login & Logout functionality.***
  - ***Allow/Block access to URLs to logged in users.***
  - ***Allow/Block access to URLs to logged in users AND with certain roles.***
- Spring Security can do,
  - ***Username/password authentication.***
  - ***SSO/Okta/LDAP.***
  - ***App level authorization.***

    - ***Intra App authorization like OAuth.***
    - ***Method level security.***
    - ***Microservice security (using tokens, JWT)***
- Default Behaviour,
    - ***Adds madnatory authentication for URLs.***
    - ***Adds Login form***
    - ***Handles login error***
    -

Five Spring Security Concepts

Authentication

> 💡 Verifying the identity of a user or process

- Types,
    - ***Knowledge based Authentication.***
        - ex : username, password
    - ***Possession based Authentication.***
        - ex : phone/text messages, key cards, badges, access token device
    - ***Multi factor Authentication***
        - combination of knowledge & possession
        - ex : two step verification (i.e; enter both password & otp)

Authorization

> 💡 Granting someone the ability to access a resource (***determines their access rights***)

- Are they allowed to do this?

Principal

> 💡 Principal is the person, who is identified by process of authentication. (***i.e; currently logged in user***)

Authority

> 💡 Authority is simply nothing but ***permission.***

- Authorities are fine-grained permissions.

Role

> 💡 Group of authorities that are assigned together. (***i.e; group of authorities assigned to a person***)

- Admin can access everthing.
- User has limited access.
- Roles are coarse-grained permissions. (i.e; varies to each & everyone)

Configure Spring Security Authentication

- Steps,
  - Get hold of AuthenticationManagerBuilder.
  - Set the configuration on it.

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().withUser("gangadhar")
    .password("hello").roles("ADMIN")
    .and().withUser("ganesh").password("world").roles("USER");
  }

  @Bean
  public PasswordEncoder getEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

```
@RestController
public class SimpleController {
  @RequestMapping("/")
  public String show() {
    return "<h1>hello world</h1>";
  }
}
```

▼ Configure Spring Security Authorization

💡 To check whether the user has the authority to perform request.

💡 HttpSecurity object enables us to configure the authorization. (***i.e; we can configure paths, access restrictions***)

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("gangadhar").password("hello").roles("ADMIN")
        .and()
        .withUser("ganesh").password("world").roles("USER");
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
          .antMatchers("/admin").hasRole("ADMIN")
          .antMatchers("/user").hasAnyRole("USER","ADMIN")
          .antMatchers("/").permitAll().and().formLogin();
  }

  @Bean
  public PasswordEncoder getEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

```
@RestController
public class SimpleController {
  @RequestMapping("/")
  public String everyone() {
    return ("<h1>welcome</h1>");
  }

  @RequestMapping("/user")
  public String user() {
    return ("<h1>Welcome user/admin</h1>");
  }

  @RequestMapping("/admin")
  public String admin() {
    return ("<h1>welcome admin</h1>");
  }
}
```

▼ JWT

💡 JWT come into picture only after user is authenticated & is used for further interaction with website (***i.e; server remembers user using this JWT***)

▼ Intro

💡 Used for further interactions with the website after authenticated. (***server remembers the user using this token***)

> 💡 web, security, jjwt dependencies are needed.

> 💡 It is a Authorization strategy (***Session Token, JSON web token are some strategies***)

- Provides us a way to authorize API requests, using token based authentication.
- **Session Token** - Reference Tokens (***it has the address of the user object***)
- **JSON Web Token** - Value Tokens (***token itself is the object***)

▼ JWT Structure

```
fjkyhauiew7ASFDASfdh8q34kahf93ASDf98y3f.askdjfh98w3yoiasASDFAfAEIF8UHIAU3897YA.AKJASHDFasdfashdLFHAF272HKAU_HSD87QAY3

# format

Header.payload.signature
```

- JWT has 3 parts,
  - ***Header*** - ***To specify algorithm used to encode & decode the JWT token.*** ***(*** algo & token type ***)***
  - ***Payload*** - ***contains the values that are used to pass the information. (do not store values in payload) (*** data ***)***
  - ***Signature*** - ***Created based on the server authentication. verifies the token (*** signature ***)***

▼ How it Works

- JWT Working Steps,
  - ***Authenticate using userId/Password.***
  - ***Authentication Complete.***
  - ***Creates JWT for future authorization purpose.***
  - ***Server sends JWT to Client.***
  - ***Saved on client side.***
    - ***in Local storage.***
    - ***as a cookie.***
  - ***JWT is passed in HTTP Header for every request***
    - ***HTTP Header is a (key,value) pair***
    - ***Authorization : Bearer + JWT***
  - ***Server examines request & verifies the signature***
    - ***base64 algo for header & payload***

▼ Implementation

```
<dependency>
      <groupId>io.jsonwebtoken</groupId>
      <artifactId>jjwt</artifactId>
      <version>0.9.1</version>
</dependency>
```

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
</dependency>
```

▼ JDBC Authentication

▼ *With Default Schemas*

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  @Autowired
  DataSource dataSource;

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource);
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin").hasRole("ADMIN")
        .antMatchers("/user").hasAnyRole("ADMIN", "USER")
        .antMatchers("/").permitAll().and().formLogin();
  }

  @Bean
  public PasswordEncoder getPasswordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

```
@RestController
public class SimpleController {
  @RequestMapping("/")
  public String all() {
    return "<h1>Welcome !<h1>";
  }
  @RequestMapping("/admin")
  public String admin() {
    return "<h1>Welcome admin!<h1>";
  }

  @RequestMapping("/user")
  public String user() {
    return "<h1>Welcome user!<h1>";
  }
}
```

```
insert into users(username, password, enabled)
          values('admin', 'pass', true);
insert into users(username, password, enabled)
          values('user', 'pass', true);

insert into authorities(username, authority)
          values('admin','ROLE_ADMIN');
insert into authorities(username, authority)
          values('user','ROLE_USER');
```

```
create table users(
      username varchar(50) not null primary key,
      password varchar(50) not null,
      enabled boolean not null);

  create table authorities (
      username varchar(50) not null,
      authority varchar(50) not null,
      constraint fk_authorities_users foreign key(username)
      create unique index ix_auth_username on authorities (
```

▼ *With Custom Schemas*

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  @Autowired
  DataSource dataSource;

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource)
        .usersByUsernameQuery("select username, password, enabled from users where username = ?")
        .authoritiesByUsernameQuery("select username, authority from authorities where username = ?");
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/admin").hasRole("ADMIN").antMatchers("/user").hasAnyRole("ADMIN", "USER")
        .antMatchers("/").permitAll().and().formLogin();
  }

  @Bean
  public PasswordEncoder getPasswordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

```
@RestController
public class SimpleController {
  @RequestMapping("/")
  public String all() {
    return "<h1>Welcome !<h1>";
  }
  @RequestMapping("/admin")
  public String admin() {
    return "<h1>Welcome admin!<h1>";
  }

  @RequestMapping("/user")
  public String user() {
    return "<h1>Welcome user!<h1>";
  }
}
```

▼ JPA Authentication

```
  @Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int id;
  private String userName;
  private String password;
  private boolean active;
  private String roles;
  public int getId() {
    return id;
  }
```

```
@EnableWebSecurity
public class JpaConfig extends WebSecurityConfigurerAdapter
  @Autowired
  UserDetailsService userDetailsService;

  @Override
  protected void configure(AuthenticationManagerBuilder auth
    auth.userDetailsService(userDetailsService);
  }

  @Override
  protected void configure(HttpSecurity http) throws Excepti
```

```
  public void setId(int id) {
    this.id = id;
  }
  public String getUserName() {
    return userName;
  }
  public void setUserName(String userName) {
    this.userName = userName;
  }
  public String getPassword() {
    return password;
  }
  public void setPassword(String password) {
    this.password = password;
  }
  public boolean isActive() {
    return active;
  }
  public void setActive(boolean active) {
    this.active = active;
  }
  public String getRoles() {
    return roles;
  }
  public void setRoles(String roles) {
    this.roles = roles;
  }
}
```

```
@Repository
public interface UserRepo extends JpaRepository<User, Integer> {
  Optional<User> findByUserName(String userName);
}
```

```
public class MyUserDetails implements UserDetails {

  private String userName;
  private String password;
  private boolean active;
  private List<GrantedAuthority> authorities;

  public MyUserDetails(User user) {
    this.userName = user.getUserName();
    this.password = user.getPassword();
    this.active = user.isActive();
    this.authorities = Arrays.stream(user.getRoles().split(",")).map(SimpleGrantedAuthority::new)
        .collect(Collectors.toList());
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {
    // TODO Auto-generated method stub
    return authorities;
  }

  @Override
  public String getPassword() {
    return password;
  }

  @Override
  public String getUsername() {
    return userName;
  }

  @Override
  public boolean isAccountNonExpired() {
    return true;
  }

  @Override
  public boolean isAccountNonLocked() {
    return true;
  }

  @Override
  public boolean isCredentialsNonExpired() {
    return true;
  }
```

```
    http.authorizeRequests().antMatchers("/admin").hasRole("
        .antMatchers("/").permitAll().and().formLogin();
  }

  @Bean
  public PasswordEncoder getPasswordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

```
@RestController
public class SimpleController {
  @GetMapping("/")
  public String all() {
    return "<h1>Welcome !<h1>";
  }

  @GetMapping("/admin")
  public String admin() {
    return "<h1>Welcome admin!<h1>";
  }

  @GetMapping("/user")
  public String user() {
    return "<h1>Welcome user!<h1>";
  }
}
```

```
@Service
public class MyUserDetailsService implements UserDetailsServ
  @Autowired
  UserRepo userRepo;

  @Override
  public UserDetails loadUserByUsername(String username) thr
    Optional<User> user = userRepo.findByUserName(username);
    user.orElseThrow(() -> new UsernameNotFoundException("us
    return user.map(MyUserDetails::new).get();
  }
}
```

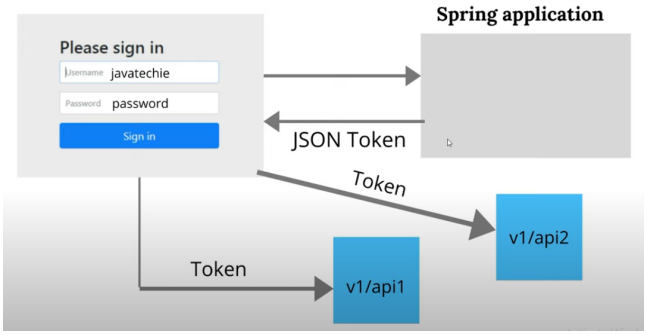
```
@SpringBootApplication
@EnableJpaRepositories(basePackageClasses = UserRepo.class)
public class JpaSecurityApplication {

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

```
  @Override
  public boolean isEnabled() {
    return active;
  }
}
```

▼ Spring Security (3hrs project notes)

> 💡 It follows completely stateless authentication mechanism. (***i.e; user input/user state are never stored in cookies/server memory***)

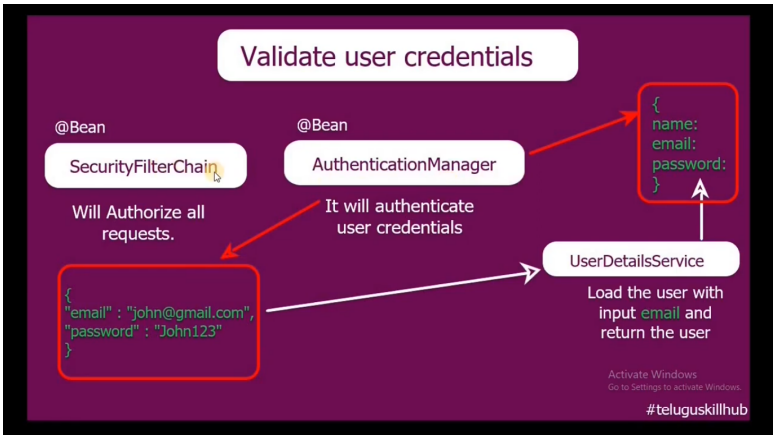
- Authentication ways,
  - Form based Authentication —> Activates instantly when we add security dependency.
  - Basic Authentication
  - JWT Authentication



  - OAuth Authentication

▼ JWT Authentication

- Validate User Credentials
  - ***SecurityFilterChain*** —> will authorize all requests.
  - ***AuthenticationManager*** —> will authenticate user credentials
  - ***UserDetailsService*** —> is an interface, which loads user with input email & return user



- Generate JWT for authenticated user
- Handle API's using valid token

▼ Jwt Implementation

```
package com.example.taskmanagement.util;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

@Service
public class JwtUtil {

  // secret key to encode token
  private String secret = "gangadhar";

  //extracts username from the token using claims
  public String extractUsername(String token) {
    return extractClaim(token, Claims::getSubject);
  }
  //extracts expiration date from the token using claims
  public Date extractExpiration(String token) {
    return extractClaim(token, Claims::getExpiration);
  }

  // extract claim using token
  public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
    final Claims claims = extractAllClaims(token);
    return claimsResolver.apply(claims);
  }
  // extract all claim using token
  private Claims extractAllClaims(String token) {
    return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();
  }

  //check whether token is expired or not
  private Boolean isTokenExpired(String token) {
    return extractExpiration(token).before(new Date());
  }


  //generate token
  public String generateToken(String username) {
    Map<String, Object> claims = new HashMap<>();
    return createToken(claims, username);
  }

  //create token using username(subject), claims
  private String createToken(Map<String, Object> claims, String subject) {

    return Jwts.builder()
        .setClaims(claims)
        .setSubject(subject)
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
        .signWith(SignatureAlgorithm.HS256, secret)
        .compact();
  }

  //validates token using token and user details(name)
  public Boolean validateToken(String token, UserDetails userDetails) {
    final String username = extractUsername(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
  }
}
```

```
@Component
public class JwtFilter extends OncePerRequestFilter {

  @Autowired
  private JwtUtil util;

  @Autowired
  private CustomUserDetailsService service;

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor

@Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
  private String email;
  private String password;
}
```

```
import org.springframework.security.core.userdetails.User;
@Service
public class CustomUserDetailsService implements UserDetails

  @Autowired
  private UserRepository repo;

  @Override
  public UserDetails loadUserByUsername(String username) thr
    User user = repo.findByName(username);
    return new User(user.getName(), user.getPassword(),
        new ArrayList<>());
  }
}
```

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAda

  @Autowired
  private CustomUserDetailsService service;

  @Autowired
  private JwtFilter filter;

  @Override
  protected void configure(AuthenticationManagerBuilder auth
    auth.userDetailsService(service);
  }

  @Override
  protected void configure(HttpSecurity http) throws Excepti
    http.csrf().disable()
      .authorizeRequests()
      .antMatchers("/authenticate")
      .permitAll()
      .anyRequest()
      .authenticated()
      .and()
      .exceptionHandling()
      .and()
      .sessionManagement()
      .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.addFilterBefore(filter,UsernamePasswordAuthentication
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }

  @Bean(name = BeanIds.AUTHENTICATION_MANAGER)
  public AuthenticationManager manager() throws Exception {
    return super.authenticationManagerBean();
  }
}
```

```
@RestController
public class Controller {

  @Autowired
```

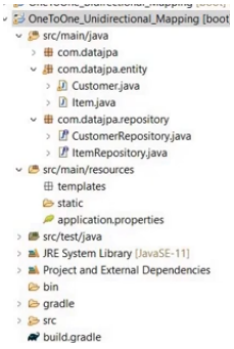
```
      throws ServletException, IOException {

    // extract authorization header from servlet request.
    String authorizationHeader = request.getHeader("Authorization");

    String token = null;
    String username = null;

    // Bearer kajshgd23jhk2j3d@#.aisdi2hj3DASDo2h38QDJ#2.k2j3d@#aisdi2hj3DASD
    // from above text we only want the token but not the text bearer
    // so we can exclude that as follows and extracts username for validation
    if (authorizationHeader != null && authorizationHeader.startsWith("Bearer")) {
      token = authorizationHeader.substring(7);
      username = util.extractUsername(token);
    }

    // validating the user by token
    if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
      UserDetails details = service.loadUserByUsername(username);
      if (util.validateToken(token, details)) {
        UsernamePasswordAuthenticationToken userToken = new UsernamePasswordAuthenticationToken(details, null,
            details.getAuthorities());
        userToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(userToken);
      }
    }

    filterChain.doFilter(request, response);
  }
}
```

```
  private JwtUtil util;

  @Autowired
  private AuthenticationManager manager;

  @PostMapping("/authenticate")
  public String generateToken(@RequestBody AuthRequest reque
    try {
      manager.authenticate(new UsernamePasswordAuthenticatio
    } catch (Exception e) {
      throw new Exception("invalid user info");
    }
    return util.generateToken(request.getName());
  }
}
```

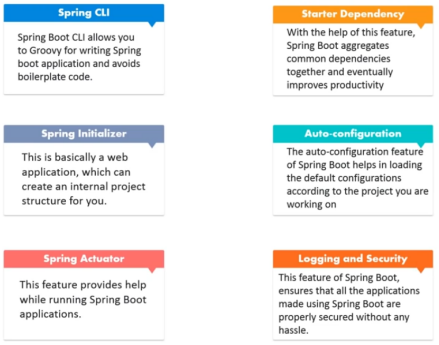
```
// method in SecurityConfig class

@Override
protected void configure(HttpSecurity http) throws Excepti
    http.csrf().disable()
    .authorizeRequests()
    .antMatchers("/authenticate")
    .permitAll()
    .anyRequest()
    .authenticated();
}
```

- Next add ***{Authorization}*** as key & ***{Bearer token}*** as value.
- Next authenticate the user using the JWT token using filter.

▼ Class Pics

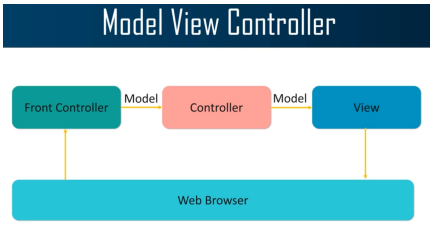


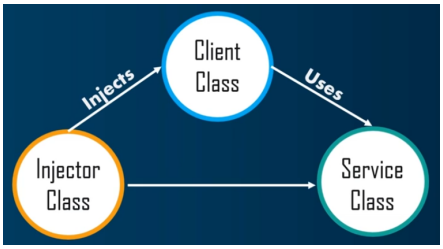
// project structure



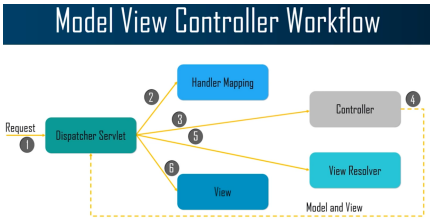
// Spring features

// why Spring?





// Class types in Dependency Injection



▼ Real Time

▼ *Package standards in Project*

💡 Used to achieve easy code management.

Package Name	***Use***
.entity	for entities
.controller	for controllers
.repository	interface for repositories
.service	interface for service (business logic)
.serviceImpl	for service implementation
.payload	for Dto classes
.exception	for exceptions

▼ *DTO (Data Transfer Object)*

💡 Use to transfer the data between client and server.

💡 Dto classes should be created in .payload package.

💡 It can be acheived by using model mapper dependency.

```
@Setter
@Getter
public class UserDto{
  private Long id;
  private String name;
}
```

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.1.1</version>
</dependency>
```

```
@Data
@Entity
public class User{
  @Id
  private Long id;
  private String name;

  @OneToMany
  private Tasks tasks;
}
```

▼ Points To Remember

- The ***@RequestParam*** is used to extract query parameters while ***@PathVariable*** is used to extract data right from the URI.
- ***@Bean*** is just for the metadata definition to create the bean(equivalent to tag) ***@Autowired*** is to inject the dependancy into a bean(equivalent to ref XML tag/attribute).
- ***@ControllerAdvice*** in the context of exception handling is just another way of doing exception handling at a global level using ***@Exceptionhandler*** annotation.