Project 10 Coding guideline

Contents

[Technologies 2](#_Toc13229272)

[Application Architecture 2](#_Toc13229273)

[Spring Boot Coding guidelines 3](#_Toc13229274)

[1] Create DTO 3](#_Toc13229275)

[2] Create Form Bean 4](#_Toc13229276)

[3] Create DAO 4](#_Toc13229277)

[4] Create Service 5](#_Toc13229278)

[5] Create Controllers 6](#_Toc13229279)

[Create Singe Entry Form Page 7](#_Toc13229280)

[Create component controller 7](#_Toc13229281)

[Create component html view 8](#_Toc13229282)

[Register component and define routes 8](#_Toc13229283)

[Create List Page with pagination and search 9](#_Toc13229284)

[Create component controller 9](#_Toc13229285)

[Create component list html view 9](#_Toc13229286)

[Register component and define routes 11](#_Toc13229287)

[Common Components 11](#_Toc13229288)

[HttpService 11](#_Toc13229289)

[ServiceLocatorService 11](#_Toc13229290)

[EndpointService 11](#_Toc13229291)

[DataValidator 11](#_Toc13229292)

[BaseCtl 11](#_Toc13229293)

[BaseListCtl 11](#_Toc13229294)

[In build Angular components 12](#_Toc13229295)

[ActivatedRoute 12](#_Toc13229296)

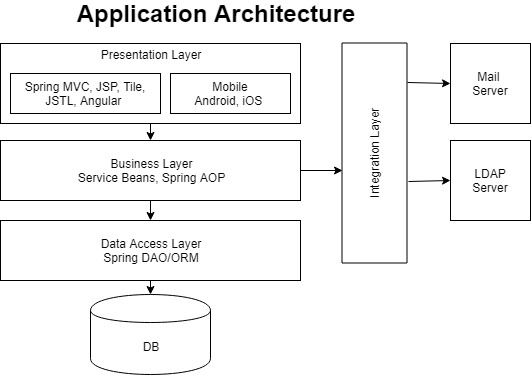
[Router 12](#_Toc13229297)

[HttpClient 12](#_Toc13229298)

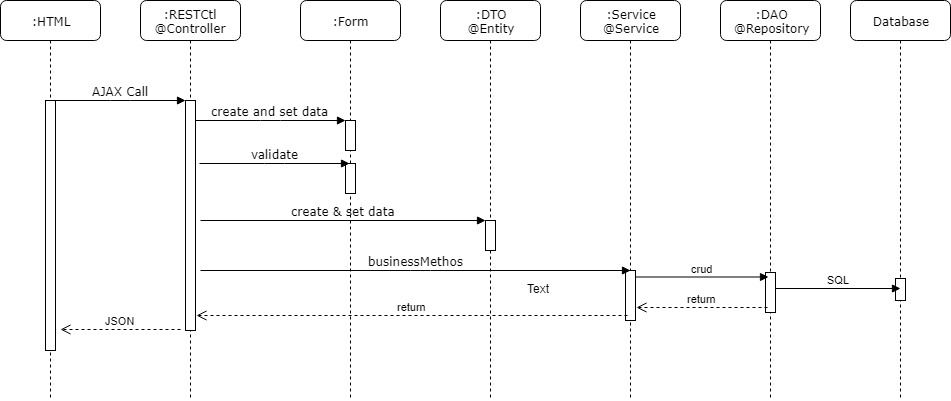
# Technologies

1. Spring Boot
2. JPA
3. Angular 1
4. Angular 6
5. Bootstrap

# Application Architecture



Layered Architecture



Component Sequence Diagram

# Spring Boot Coding guidelines

Follow following code guideline to develop a user Develop am

## 1] Create DTO

DTO stand for Data Transfer Object. It is a POJO class, contains attributes and its setter and getter methods. DTOs are created as per database model.

You can create a DTO using following steps:

1. Extend BaseDTO
2. Map DTO with @Entity annotation.
3. Map attributes with correct columns and data types
4. Override following methods
   1. getValue(): returns label of dropdown list
   2. orderBY(): attributes that will short the list
   3. uniqueKeys(): unique attributes of dto
5. DTOs are created as per database model

**Example code**

**@Entity**

**@Table(name = "NCS\_ORG")**

public class OrgDTO extends **BaseDTO** {

**@Column(name = "NAME", length = 50)**

private String name = null;

**@Column(name = "EMAIL", length = 100)**

private String email = null;

//Getter and setter methods

@Override

public String **getValue**() {

return name;

}

@Override

public LinkedHashMap<String, String> **orderBY**() {

LinkedHashMap<String, String> map

= new LinkedHashMap<String, String>();

map.put("name", "asc");

return map;

}

@Override

public LinkedHashMap<String, Object> **uniqueKeys**() {

LinkedHashMap<String, Object> map

= new LinkedHashMap<String, Object>();

map.put("name", name);

map.put("email", email);

return map;

}

}

Method uniqueKey() contains unique keys attributes of DTO. Record will not be inserted if these keys are duplicate.

Method orderBY() contains attributes by which DTO list will be ordered.

## 2] Create Form Bean

Form Bean is a POJO class. It binds bean attributes with HTML for input elements. It validates user input data and coverts valid data into DTO. Form beans are created by seeing UI screens.

You can create a Form bean using following steps:

1. Extend BaseForm
2. Define attribute as per your HTML form
3. Apply input validation using annotations
4. Override getDTO() method that will create respective bean and populate data into bean from Form attributes.

**Example Code**

public class OrgForm extends BaseForm {

**@NotEmpty**

private String name = null;

**@NotEmpty**

private String email = null;

@Override

public BaseDTO **getDto**() {

OrgDTO dto = new OrgDTO();

dto.setName(name);

dto.setEmail(email);

return dto;

}

}

## 3] Create DAO

DAO contains data access logics. DAO follows bridge design pattern that allows changes in implementation without impacting business layer of application. DAOs are created as per database model. DAO is defined by @Repository annotation.

You can create a DAO using following steps:

1. DAO is implemented using interface and implementation class.
2. Create DAO interface by extending BaseDAOInt<DTO>. Pass respective DTO class as its generic type.
3. Create DAO implementation class by implementing above interface and extending BaseDAOImpl<DTO>. Pass respective DTO class as its generic type.
4. Override getDTOClass() that will return class object of respective DTO.
5. Override getWhereClause() method to build search criteria. It is used by search() method to search records.
6. Introduce more methods in DAO for your custom queries.

**Example Code**

public interface OrgDAOInt extends BaseDAOInt<OrgDTO> {}

**@Repository**

public class OrgDAOImpl extends BaseDAOImpl<OrgDTO> implements OrgDAOInt {

/\*\*

\* Where clause for search

\*/

protected List<Predicate> **getWhereClause**( OrgDTO dto,

CriteriaBuilder builder, Root<OrgDTO> qRoot) {

// Create where conditions

List<Predicate> andPredicates

= new ArrayList<Predicate>();

if (!isEmptyString(dto.getName())) {

andPredicates.add( builder.like(

qRoot.get("name"), dto.getName() + "%"));

}

if (!isEmptyString(dto.getEmail())) {

andPredicates.add( builder.equal(

qRoot.get("email"), dto.getEmail()));

}

return andPredicates;

}

@Override

public Class<OrgDTO> **getDTOClass**() {

return OrgDTO.class;

}

}

## 4] Create Service

Service contains business logics. Service follows bridge design pattern that allows changes in implementation without impacting presentation layer of application. Service does transaction handling. Service is defined by @Service annotation.

You can create a service using following steps:

1. Service is implemented using interface and implementation class.
2. Create Service interface by extending BaseServiceInt<DTO>. Pass respective DTO class as its generic type.
3. Create Service implementation class by implementing above interface and extending BaseServiceImpl<DTO, OrgDAOInt>. Pass respective DTO class and DAO interface as its generic types.
4. Add more business methods in Service for your custom operations.

**Example Code**

public interface OrgServiceInt extends **BaseServiceInt**<OrgDTO> {

public OrgDTO findByName(String name, UserContext ctx);

}

**@Service**

@Transactional

public class OrgServiceImpl extends **BaseServiceImpl**<OrgDTO, OrgDAOInt> implements OrgServiceInt {

@Transactional(readOnly = true)

public OrgDTO findByName(String name,UserContext ctx){

return baseDao.findByUniqueKey("name", name, ctx);

}

}

## 5] Create Controllers

Controllers are REST controller in this application. Controller contains navigation logics and Control the application navigation flow. Rest API are exposed using Rest controllers. Controller communicats to service classes to perform business operations.

You can REST controller using following steps:

1. Define controller class using @RestController
2. Map controller with URL using @RequestMapping
3. Extend controller class by BaseCtl class.
4. Service is implemented using interface and implementation class.
5. Pass respective Form Bean, DTO and Service interface of controller to BaseCtl<Form, DTO, ServiceInt> generics.
6. Define preload method if you required preload data on the page.
7. BaseCtl provides endpoints for:
   1. /get
   2. /get/id
   3. /save
   4. /delete
   5. /search
   6. add more end points if required

**Example Code**

@RestController

**@RequestMapping(value = "Role")**

public class RoleCtl extends **BaseCtl<RoleForm, RoleDTO, RoleServiceInt>** {

**@GetMapping("/preload")**

**public ORSResponse preload() {**

ORSResponse res = new ORSResponse(true);

return res;

}

**@GetMapping("name/{name}")**

**public ORSResponse get(@PathVariable String name) {**

ORSResponse res = new ORSResponse(true);

RoleDTO dto = baseService.findByName(name, userContext);

System.out.println("Role " + dto);

if (dto != null) {

res.addData(dto);

} else {

res.setSuccess(false);

res.addMessage("Record not found");

}

return res;

}

}

# Angular JS 1.x

T

Angular JS 1.x

# 

# Angular 6 Coding guidelines

## Create Singe Entry Form Page

Single entry form add or edit a single record. For example, add/edit College, Student or Marksheet

## Create component controller

1. student.component.ts file
2. Define StudentComponent class that inherits BaseCtl class in this file. Make sure selector and templateUrl properties have correct entry.

@Component({

selector: 'app-student',

templateUrl: './student.component.html',

styleUrls: ['./student.component.css']

})

export class StudentComponent extends BaseCtl { .. }

1. Define constructor in this class and inject ServiceLocatorService and ActivatedRoute object.

constructor(public locator: ServiceLocatorService,

public route: ActivatedRoute) {

super(locator.endpoints.STUDENT, locator, route);

}

1. Make sure pass respective server endpoint to super constructor. Here end point is locator.endpoints.STUDENT.
2. Override validateForm(form) method to validate input html form elements. Add properties which will be validated.

validateForm(form) {

let flag = true;

let validator = this.serviceLocator.dataValidator;

flag = flag && validator.isNotNullObject(form.firstName);

flag = flag && validator.isNotNullObject(form.lastName);

flag = flag && validator.isNotNullObject(form.mobileNo);

return flag;

}

1. Override populateForm(form, data) method to populate data, fetched from into input form. Make entries of properties received form server.

populateForm(form, data) {

form.id = data.id;

form.collegeId = data.collegeId;

form.email = data.email;

form.firstName = data.firstName;

form.lastName = data.lastName;

form.mobileNo = data.mobileNo;

}

## Create component html view

1. student.component.html file
2. Define placeholder to display error and success message of input form:

<div \*ngIf="form.error" style="color:red"> {{form.message}}</div>

<div \*ngIf="!form.error" style="color:green">{{form.message}}</div>

1. Define html form to enter user data and create placeholders to display input error messages. Form data are bound with form.data object and input error messages are bund with form.inouterror object.

<form>

FirstName:<input name="fName" [(ngModel)]="form.data.firstName">

<span style="color: red">{{form.inputerror.firstName}}</span>

Last Name:<input name="lName" [(ngModel)]="form.data.lastName">

<span style="color: red"> {{form.inputerror.lastName}} </span>

Email: <input name="email" [(ngModel)]="form.data.email">

<span style="color: red"> {{form.inputerror.email}} </span>

<button (click)="submit()" [disabled]="!validate()">Save</button>

<button (click)="forward('/studentlist')" >List</button>

</form>

1. Save button will call “submit()” method to submit data to the server.

<button (click)="submit()" [disabled]="!validate()">Save</button>

1. If you want to navigate to some other page on button click then call forward(/url) method. Here we are forwarding request to student list page forward('/studentlist').

## Register component and define routes

1. Register component in app.module.js file

@NgModule({

declarations: [

StudentComponent,

StudentListComponent,

], .. })

1. Define routes in app-routing.module.ts file

{path: 'student', component: StudentComponent },

{path: 'student/:id', component: StudentComponent},

{path: 'studentlist', component: StudentListComponent }

1. Define link to navigate on these pages using

<a [routerLink]="['/student']">Student</a> |

<a [routerLink]="['/studentlist']">Student</a>

1. Forward to these components from any other component using forward() method:

<button (click)="forward('/studentlist')" >List</button>

# Create List Page with pagination and search

List page displayed list of records and provides pagination and searching capability.

## Create component controller

1. student-list.component.ts file
2. Define StudentListComponent class which inherits BaseListCtl class in this file

@Component({

selector: 'app-student-list',

templateUrl: './student-list.component.html',

})

export class StudentListComponent extends BaseListCtl {..}

1. Define constructor in this class and inject ServiceLocatorService and ActivatedRoute object.

constructor(public locator: ServiceLocatorService,

public route: ActivatedRoute) {

super(locator.endpoints.STUDENT, locator, route);

}

}

1. Make sure pass respective server endpoint to super constructor. Here end point is locator.endpoints.STUDENT.

## Create component list html view

List page displays list with pagination and search elements:

1. Create student-list.component.html file
2. Define placeholder to display error and success message of input form:

<div \*ngIf="form.error" style="color:red"> {{form.message}}</div>

<div \*ngIf="!form.error" style="color:green">{{form.message}}</div>

1. Define html search form to enter search data. Form data are bound with form.searchParams object.

<form>

FirstName:

<input name="fName" [(ngModel)]="form.searchParams.firstName">

Last Name:

<input name="lName" [(ngModel)]="form.searchParams.lastName">

Email:<input name="email" [(ngModel)]="form.searchParams.email">

<button (click)="submit()"> Search</button>

</form>

1. Search button will call “submit()” method to submit search data to the server.

<button (click)="submit()"> Search</button>

1. Create a table and display list elements from form.list object using \*ngFor directive.

<table>

<tr>

<th>ID</th>

<th>First Name</th>

<th>Last Name</th>

<th>Mobile</th>

<th>Email</th>

</tr>

<tr \*ngFor="let m of form.list ">

<td> {{ m.id }} </td>

<td> {{ m.firstName }} </td>

<td> {{ m.lastName }} </td>

<td> {{ m.mobileNo }} </td>

<td> {{ m.email }} </td>

<td> <button (click)="forward( '/student/' + m.id)">Edit</button>

</td>

<td> <button (click)="delete(m.id)">Delete</button> </td>

</tr>

</table>

1. Add edit and delete button and call forward('/student/' + m.id)and delete(m.id) method to edit and delete list record.
2. Add two pagination buttons Next and Previous which will call next() and previous() methods respectively.

<button (click)="previous()" [disabled]="form.pageNo == 0" > Previous </button>

<button (click)="next()"> Next </button>

## Register component and define routes

1. Register component in app.module.js file

@NgModule({

declarations: [

StudentComponent,

StudentListComponent,

], .. })

1. Define routes in app-routing.module.ts file

{path: 'student', component: StudentComponent },

{path: 'student/:id', component: StudentComponent},

{path: 'studentlist', component: StudentListComponent }

1. Define link to navigate on these pages using

<a [routerLink]="['/student']">Student</a> |

<a [routerLink]="['/studentlist']">Student</a>

1. Forward to these components from any other component using forward() method:

<button (click)="forward('/studentlist')" >List</button>

# Common Components

Single entry form add or edit a single record. For example, add/edit College, Student or Marksheet

## HttpService

TODO

## ServiceLocatorService

TODO

## EndpointService

TODO

## DataValidator

TODO

## BaseCtl

TODO

## BaseListCtl

TODO

# In build Angular components

## ActivatedRoute

TODO

## Router

TODO

## HttpClient

TODO