- To add **dev tools,** add the following dependency in *pom.xml* file.

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

- *IntelliJ idea* specific settings:
  - **Setting >> Build, Execution, Deployment >> Compiler**
    - Select the option "**Build project automatically**"
    - So that the build will be automatically generated if u do any changes to the files (in *running* or *debugging* state).
  - **Advanced Setting >>**
    - Select the option "**Allow auto-make to start even if developed applicatoin is currently running**"

Auditing

- Auditing in Spring Boot allows you to automatically populate certain fields, such as creation and modification timestamps, as well as the user who created or modified the entity.
- Create Auditable base Entity using the following annotations:
  - **@EntityListeners(*AuditingEntityListener.class*)**
    -
```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class PostEntity {
```
  - Now you can use the annotations **@CreatedAt**, **@LastModifiedAt**, **@CreatedBy**, **@LastModifiedBy** and create the fields inside the ***Entity***.
    -
```
@CreatedDate
private LocalDateTime createdDate;

@LastModifiedDate
private LocalDateTime updatedDate;

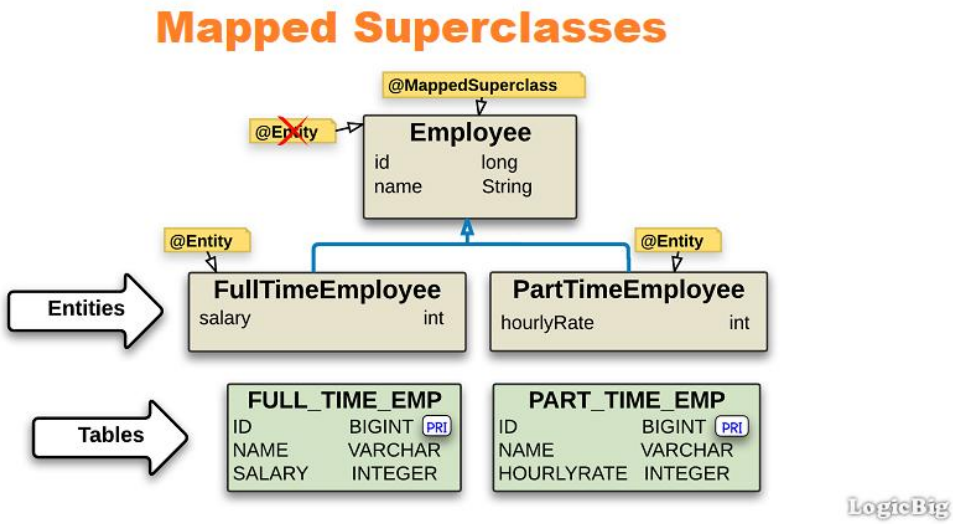
@CreatedBy
private String createdBy;

@LastModifiedBy
private String updatedBy;
```
      - These are writtn inside entity; not DTO
  - Now, to make all these things work, you need to add **@EnableJpaAuditing** in any of the config file (file having **@Configuration** annotation)
    -
```
@Configuration  no usages  Alok Ra
@EnableJpaAuditing
public class AppConfig {
```
  - Now it'll work properly when someone use **post**, **put** request.
    -

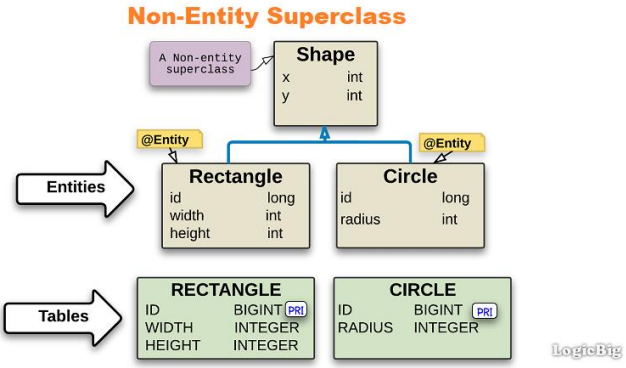
id	description	title	created_date	updated_date	created_by	updated_by
2	post 2 description	post 2	5-12-22 20:21:41.085533	25-12-22 20:21:41.085533	[NULL]	[NULL]

    - Now if someone do some changes, it'll be visible in the Database;
- Instead of writing **@EntityListener** in any particular entity, its better to create one class and write this annotation there. Whichever *Entity* wants to be audited, they can directly inherit that class.

- There is a annotation **@MappedSuperClass,**
  - lets say you want some common fields to be there in some Entities.
  - So, instead of writing those in every Entity, its better to write in a parent class and extend that class from the actual Entity class.
  - But, here the fields that are defined in the parent class will not be included in the table by default.
  - If you write **@MappedSuperClass** in the parent class, only then those fields will be part of the Entity table.



  -
    - **@MappedSuperClass** is writtn in *@Employee*, so the entities (FullTimeEmployee, PartTimeEmployee) are having the *id, name* fields in their tables.



  -
    - **@MappedSuperClass** is not written for the parent class *Shape*, so the Entities (Rectangle, Circle) are not having *x, y* fields in their tables.
- But till now, you'll only get **@CreatedAt** and **@LastModifiedAt** , but the fields for **@CreatedBy**, **@LastModifiedBy** will be **null** only, as it doesn't know who made the changes.

- Its part of spring security to fetch the current user and add that user in the database for createdBy or updatedBy fields.
- For now, we can use some dummy name for this.

➢ You need to write a class that implement the interface **AuditorAware**

```
public class AuditorAwareImpl implements AuditorAware<String> {
    @Override  no usages
    public Optional<String> getCurrentAuditor() {
        // get security context
        // get authentication
        // get principle
        // get username
        return Optional.of( value: "Alok Ranjan Joshi");
    }
}
```

- For now I am returning "Alok Ranjan Joshi" as the modifier/creator name.

➢ After this, you need to update the annotation **@EnableJpaAuditing** of the config class.

```
@Configuration  no usages   Alok Ranjan Joshi *
@EnableJpaAuditing(auditorAwareRef = "getAuditorAware")
public class AppConfig {

    @Bean  no usages  new *
    AuditorAware<String> getAuditorAware() {
        return new AuditorAwareImpl();
    }
```

- **auditorAwareRef** takes the bean name.
- As the method name is *getAuditorAware*, by default method name becomes the bean name if you are not specifically mentioning a different name for the bean.

➢ F

➢ F

- So, All the steps are:
  - Create Auditable base Entity using the following Annotations
    - **@EntityListeners(AuditingEntityListener.class)**.
    - You can create the fields using **@CreatedBy**, **@LastModifiedBy**, **@CreatedAt**, **@LastModifiedAt**.
  - If you are creating a super class for *auditing*, then use **@MappedSuperClass** on that parent class, otherwise the entities that inherit that parent class won't contain the fields defined in that parent class.
  - Now write **@EnableJpaAuditing** in any config class (annotated with *@Configuration*).
  - Create an class implementing the interface **AuditorAware** interface.
  - Then edit the **config** file's annotation **@EntityListeners** and pass the *bean name* (by default the method name) for the **auditorAwareRef**.
  -

- Optimized steps:
  - Enable auditing (MANDATORY)
    -
```
@EnableJpaAuditing
@SpringBootApplication
public class Application {
}
```
    - Better to write in the main application file (@SpringBootApplication)
  - Create a base audit class (BEST PRACTICE)
    -
```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
@Getter
@Setter
public abstract class Auditable {

    @CreatedDate
    @Column(updatable = false)
    private Instant createdAt;

    @LastModifiedDate
    private Instant updatedAt;

    @CreatedBy
    @Column(updatable = false)
    private Long createdBy;

    @LastModifiedBy
    private Long updatedBy;
}
```
    - **@MappedSuperClass** is necessary; without that the fields will not be available in the ***entities*** *that inherits this* ***Auditable*** *class.*
    - **AuditingEntityListener** class contains all the methods that is being triggered to update the entity before **create/update.**

```
@PrePersist
public void touchForCreate(Object target) {
    Assert.notNull(target, message: "Entity must not be null");
    if (this.handler != null) {
        AuditingHandler object = (AuditingHandler) this.handler.getObject();
        if (object != null) {
            object.markCreated(target);
        }
    }

}
```

- It is triggered before **create** event.

```
@PreUpdate
public void touchForUpdate(Object target) {
    Assert.notNull(target, message: "Entity must not be null");
    if (this.handler != null) {
        AuditingHandler object = (AuditingHandler) this.handler.getObject()
        if (object != null) {
            object.markModified(target);
        }
    }

}
```

- It is triggered before **update** event.

- You can write your own methods using the annotations **@PrePersist, @PreUpdate, @PreRemove** that will be triggered before the respective operations.

- Handling **@CreatedBy** and **@LastModifiedBy**

```
@Component
public class AuditorAwareImpl implements AuditorAware<Long> {

    @Override
    public Optional<Long> getCurrentAuditor() {
        // Example with Spring Security
        return Optional.of(
            SecurityContextHolder.getContext()
                .getAuthentication()
                .getPrincipal()
                .getUserId()
        );
    }
}
```

- You need to create the bean of this class. Either use **@Component** or **@Configuration**, **@Bean** .
- If you are having more than one classes that implements **AuditorAware**, and you are creating multiple beans, then you need to pass **auditorAwareRef** inside **@EnableJpaAuditing**.

```
@Configuration  no usages  Alok Ranjan Joshi *
@EnableJpaAuditing(auditorAwareRef = "getAuditorAware")
public class AppConfig {

    @Bean  no usages  new *
    AuditorAware<String> getAuditorAware() {
        return new AuditorAwareImpl();
    }
```

  - Like here, the bean name will be same as the method name i.e. **getAuditorAware** so I am passing the bean name in the **auditorAwareRef**.
  - But in case of single bean, no need of passing this.

- Extend the base class in entities

```
@Entity
public class Project extends Auditable {
```

➢ **Overall Annotations** ----------

- @EnableJpaAuditing(AuditingEntityListener.class)
- @MappedSuperClass
- @CreatedAt, @LastModifiedAt, @CreatedBy, @LastModifiedBy
- @PrePersist, @PreUpdate, @PreRemove
- **AuditorAware** class has to be implemented for @CreatedBy, @LastModifiedBy
- ----------

➢ **To keep track of every version like different versions for every change and all will be containing who changed and what was changed and all, you can use Hibernate Envers .**

➢

RestClient

- RestClient is a synchronous HTTP client that offers a modern fluent API. It offers an abstraction over HTTP libraries that allows for convenient conversion from a Java object to an HTTP request, and the creation of objects from an HTTP response.
- Generally in case of *microservice* architecture, one service needs to call another service.
  - Don't think @RestController and RestClient are same.
  - RestController create API; RestClient consumes those APIs.
- Lets say you have 2 services in your microservice architecture; and one service needs to call another service (API call) then RestClient is used;
  - **Order Service** ——**calls**——► **User Service** (for example)
- Steps:
  - First you'll configure the RestClient giving some base url. For example http://localohost:8080
  - After that you can call any type of method i.e. get, post, put, patch, delete with request body, headers etc etc just like normal API call (just like API call is made using *axios* in react)
  - Either you can create the object of type RestClient using **RestClient.builder()…..build()** method or create a *bean* of this and inject that everywhere.

```
@Configuration
public class RestClientConfig {

    @Bean
    RestClient paymentRestClient() {
        return RestClient.builder()
                .baseUrl("http://payment-service:8080")
                .build();
    }
}
```

  -
-

- Steps
  - First create one **@Bean** to create bean of ***RestClient***.

```
@Configuration  no usages
public class RestClientConfig {

    @Value("${employeeService.base.url}")  1 usage
    private String BASE_URL;

    @Bean  no usages
    @Qualifier("employee-rest-client")
    public RestClient getEmployeeServiceBaseClient() {
        return RestClient.builder()
                .baseUrl(BASE_URL)
                .defaultHeader(CONTENT_TYPE, APPLICATION_JSON_VALUE)
                .build();
    }
}
```

    - Here better to use **@Qualifier** because one RestClient will handle one base-url.
    - If you want to create multiple RestClient (because in microservices, so many base urls will be there), so you need to create multiple beans of RestClients.
    - Here, **CONTENT_TYPE** and **APPLICATION_JSON_VALUE** are nothing but the constant static fields just to set the header values:
      - `public static final String CONTENT_TYPE = "Content-Type";`
      - `public static final String APPLICATION_JSON_VALUE = "application/json";`
  - After this, you need to write **services**; but better to create a directory called **client** and write the services there; so that ***services*** of the current application and ***clients*** will be different. But you need to use **@Service** annotation here.
- **ParameterizedTypeReference** is an ***abstract*** class so you need to write
  - **new ParameterizedTypeReference<**any type**>()** {} (curly braces is important)
  - By default **generic types** are vanished during runtime in Java.
  - So, if you pass **List<User>**, then it passes **List.class** and it doesn't know about the User.
  - So to preserve that, we need to pass **new ParameterizedTypeReference<List<User>>() {}**

- So, it captures the type using reflection and stores as metadata.

➢ After that, you can use that rest client object (bean in springboot) to call the apis.

```
@Override  1 usage
public ApiResponse<List<EmployeeDTO>> getAllEmployees() {
    try {

        return restClient.get() RequestHeadersUriSpec<capture of ?>
                .uri( uri: "employees") capture of ?
                .retrieve() ResponseSpec
                .body(new ParameterizedTypeReference<ApiResponse<List<EmployeeDTO>>>() {
                });

    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

➢ If you want to pass some path params:

```
@Override  1 usage
public ApiResponse<EmployeeDTO> getEmployeeById(Long id) {
    try {

        return restClient.get() RequestHeadersUriSpec<capture of ?>
                .uri( uri: "employees/{iddd}", id) capture of ?
                .retrieve() ResponseSpec
                .body(new ParameterizedTypeReference<ApiResponse<EmployeeDTO>>() {
                });


    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

- Here you don't need to write the same name inside the uri string.
- To explain that, I have written **iddd** instead of **id**.
- If multiple path variables are there then:
  - ```
    .uri( uri: "employees/{iddd}/{another}",  ...uriVariables: id, 656)
    ```
  - The order of the values matters;
    ```

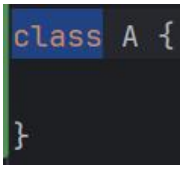
- To handle the exceptions that were got in the api request, use **onStatus** or **exchange**

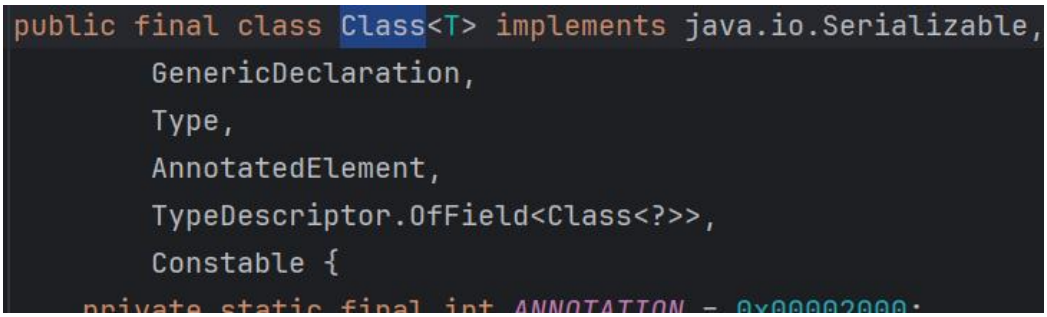
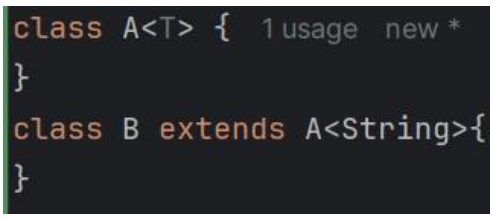
```
ApiResponse<EmployeeDTO> employeeDTOApiResponse = restClient.post() RequestBodyUriSpec
        .uri( uri: "employees") RequestBodySpec
        .body(employeeDTO)
        .retrieve() ResponseSpec
        .onStatus(HttpStatusCode::is4xxClientError, ( HttpRequest req,  ClientHttpResponse res) -> {
            System.out.println(res.getBody().readAllBytes());
            try {
                throw new ResourceNotFoundException("could not create the employee.");
            } catch (ResourceNotFoundException e) {
                throw new RuntimeException(e);
            }
        })
        .body(new ParameterizedTypeReference<ApiResponse<EmployeeDTO>>() {
        });
```

Logging

- F
- F
- F
- F
- F
- F
- F
- F

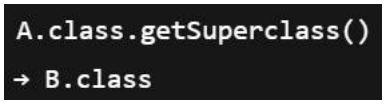
How the Java code is run?

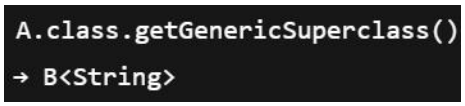
- Lets say you have created one class **A** (the file name is **A.java** of course)
- So, when you compile it, one file **A.class** will be generated. Its just a file that is being stored in the disc.
- After the compilation done, you run that **A.class** file using **java A**
  - Now, 2 things will be generated; one object of type **Class** and one **metaspace**
  - One variable will be created which will be of type **Class** ; its nothing but **A.class**
  - And one metaspace, which will contain all the metadata about the class A like *variables, methods, etc etc.*
  - That **A.class** is the reference of that **metaspace**.
  - In code, we generally perform ***reflection*** using **A.class** so it feels like **A.class** contains everything; but it is just a reference to that **metaspace**.
- So, there will be nothing like **class A** during the run time; JVM creates **A.class** which is of type **Class** and it keeps the reference of **metaspace** which contains all the metadata about the *class A*.
- **NOTE**: **class** keyword is used to create one class; **Class** is a actual class.
  - 
```
class A {

}
```
class (lowercase **c**) is used to define one class
  - 
```
public final class Class<T> implements java.io.Serializable,
        GenericDeclaration,
        Type,
        AnnotatedElement,
        TypeDescriptor.OfField<Class<?>>,
        Constable {
    private static final int ANNOTATION = 0x00002000;
```
    - It is an actual class which name is **Class**.
    - This is why it is being said "**Class** is itself a **class,** and every **class** in Java is an object of type **Class**"
- In case of generic type, it stores the generic information in **metaspace**.
  - 
```
class A<T> {  1 usage  new *
}
class B extends A<String>{
}
```

- You need to pass raw type of type just like the above, I have passed **String** while inheriting **A** from **B**, otherwise how can it know which type is being passed.
- Because, after the compilation, the **generics of objects are gone**; the only way to preserve this is **inheriting the generic class passing the raw** (in my case *String*) **type**.
- **ParameterizedTypeReference** (abstract class) works in this way; we create an object of a anonymous subclass which inherit ParameterizedTypeReference to keep the generics preserved.
  - It provides some useful methods by default; so instead of creating our own class to preserve the generics, we use this abstract class.
-
```
System.out.println(B.class);
System.out.println(B.class.getSuperclass());
System.out.println(B.class.getGenericSuperclass());
```
-
```
class com.alok.projects.prod_ready_features.B
class com.alok.projects.prod_ready_features.A
com.alok.projects.prod_ready_features.A<java.lang.String>
```
  - output
- **getGenericSuperClass()** gives you the super class with generic type; it is of type **ParameterizedType**
- **getSuperClass()** gives you the super class only without generics; it is of type **Class**.
-
```
Type  (interface)
 ├── Class
 ├── ParameterizedType   ← YOU ARE HERE
 ├── TypeVariable
 ├── WildcardType
 └── GenericArrayType
```
  - Both **Class**, and **ParameterizedType** implements **Type** (which is an interface).
  - So, if you think like the below:
    -
```
Type c = B.class.getSuperclass();
ParameterizedType ptype = (ParameterizedType) c;
```
  - It'll not work; **why?** Read the next topic about upcasting & downcasting.
-
```
System.out.println(B.class.getSuperclass() instanceof Class); // true
System.out.println(B.class.getGenericSuperclass() instanceof ParameterizedType); // true
```
  - Both will be **true**.

- To see the **metaspace** of **B**, execute the command "**javap -v B**"
  - 

  - You'll find something like this in the output.
- 

  (here reversed; A <-> B)
- 

  (here reversed; A <-> B)

Downcasting during runtime

- Lets say one interface is there named as **X** and 2 classes **A** and **B** are implementing that interface.

```
interface X {  2 usages  2 implementations  new *
    void m1();  no usages  2 implementations  new *
}
class A implements X{  no usages  new *
    public void m1() {  no usages  new *
        System.out.println("m1 method in A");
    }
    public void m2() {  no usages  new *
        System.out.println("m2 method in A");
    }
}
class B implements X{  no usages  new *
    public void m1() {  no usages  new *
        System.out.println("m1 method in B");
    }
    public void m3() {  no usages  new *
        System.out.println("m3 method in B");
    }
}
```

  - **A** and **B** are having one extra functions each **m2** and **m3** respectively.
- Now its obvious, if we create a variable of type **X** and store the object of type **A** then we'll not be able to call the method **m2**.

```
X ob = new A();
ob.m2();
```

Cannot resolve method 'm2' in 'X'

Cast qualifier to 'com.alok.project

- So there are 2 things, **variable type** and **object type**.
  - Lets say one **interface** or **class** is there (in our example **X**)
  - So, the child class may be containing **same** or **more** number of methods.
  - So, when we store an **object of type child** in a **variable of type parent**, then the variable can ignore the methods that is not in the parent class/interface.

```
// variable X
// object A
X obX = new A();
```

- But vice versa is not true; if we have a **object of type parent** and **variable of type child**, and if we try to store that object to this variable it'll not be possible;
  - Because, that object (of type parent) might be containing less number of fields and methods than the child.
  - We know, to convert **parent type to child type**, we have to do downcast; but only if the **object type is same as the child**.

➢ Consider the following examples to get a clear picture:

-
```
class X {  1 usage  2 inheritors  new *
    void m1() {  no usages  2 overrides  new *
        System.out.println("m1 in X");
    }
}
class A extends X{  no usages  new *
    public void m1() {  no usages  new *
        System.out.println("m1 method in A");
    }
    public void m2() {  no usages  new *
        System.out.println("m2 method in A");
    }
}
```

-
```
class X {  1 usage  2
}
class A extends X{
}
```

- This is my classes (parent is **X**, child is **A**)

-
```
X obX = new A();
```

  - It is completely fine; implicitly **upcasting** is happening here.
  - Object type: **A**, variable type: **X**
  - A contains all the fields and methods that is present in X.
  - So, we can access all the things from **obX**; so there is no error.

-
```
X obX = new A();
A obA = (A) obX;
```

  - It is also same; as we know **downcasting** has to be mentioned explicitely.
  - Object type: **A**, variable type: **A**

- So, **obA** can access all the variables of **A**; so no error.

```
X obX = new X();
A obA = (A) obX;
```

- It'll give **run-time** error; not compile-time.
- Object type: **X**, Variable type: **A**
- **A** might be containing more number of **methods/fields** than **X**.
- So, it'll give run-time error.

- So, there might be a thinking:
  - Lets say **A** and **B** are inheriting **X**.
  - We'll create one **object** of type **A**, and store that in a **variable** of type **X** (upcasting).
  - Then we'll downcast that to **B**, so now we can convert the **A** type to **B** type; it is not possible;

```
X obX = new A();
B obB = (B) obX;
```
XXXX

  - Because, even if you change the **variable type**, but the actual **object** is same only;
- **So, upcasting and downcasting is only possible when the object contains all the fields and methods that is being present in the type of which variable is being created.**
- F
- F
- F
- F
- F
- F
- F
- F