eazy bytes

Key points of JDBC

Intro to JDBC

- ✓JDBC or Java Database Connectivity is a specification from Core Java that provides a standard abstraction for java apps to communicate with various databases.
- ✓JDBC API along with the database driver is capable of accessing databases
- ✓JDBC is a base framework or standard for frameworks like Hibernate, Spring Data JPA, MyBatis etc.

Steps in JDBC to access DB

We need to follow the below steps to access DB using JDBC,

1) Load Driver Class
2) Obtain a DB connection
3) Obtain a statement using connection object
4) Execute the query
5) Process the result set
6) Close the connection

Problem with JDBC

- ✓Developers are forced to follow all the steps mentioned to perform any kind of operation with DB which results in lot of duplicate code at many places.
- ✓Developers needs to handle the checked exceptions that will throw from the API.
- ✓JDBC is database dependent

INTRO TO JDBC & PROBLEMS WITH IT

eazy bytes

```
public Optional<Contact> findById(int id) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");
        connection = DriverManager.getConnection("url","username","pwd");
        statement = connection.prepareStatement(
                "select id, name, message from contact");
        statement.setInt(1, id);
        resultSet = statement.executeQuery();
        Contact contact = null;
        if(resultSet.next()) {
            contact = new Contact(
                    resultSet.getInt("id"),
                    resultSet.getString("name"),
                    resultSet.getString("message"));
        }
        return Optional.of(contact);
    } catch (SQLException e) {
        // ??? What should be done here ???
    } finally {
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {}
        }
        if (statement != null) {
            try {
                statement.close();
            } catch (SQLException e) {}
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {}
        }
    }
    return null;
}
```

Sample program using JDBC to fetch a record from the database table. You can see there is lot of boilerplate code present

JDBC Program.txt

INTRO TO SPRING JDBC

eazy bytes

- *Spring JDBC simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving application code to provide SQL and extract results. It does this magic by providing JDBC templates which developers can use inside their applications.*
- *Below is the maven dependency that we need to add to any Spring/SpringBoot projects in order to use Spring JDBC provided templates,*

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

Spring provides many templates for JDBC related activities. Among them, the famous ones are JdbcTemplate , NamedParameterJdbcTemplate.

JdbcTemplate is the classic and most popular Spring JDBC approach. This provides "lowest-level" approach and all others templates uses JdbcTemplate under the covers.

NamedParameterJdbcTemplate wraps a JdbcTemplate to provide named parameters instead of the traditional JDBC ? placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.

Spring JDBC - who does what?

Action	Spring JDBC	Developer
Define connection parameters		✕
Open the connection	✕	
Specify the SQL statement		✕
Declare parameters and provide parameter values		✕
Prepare and run the statement	✕	
Set up the loop to iterate through the results (if any).	✕	
Do the work for each iteration		✕
Process any exception	✕	
Handle transactions	✕	
Close the connection, the statement, and the resultset	✕	

USING JdbcTemplate

eazy bytes

- *JdbcTemplate is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors, such as forgetting to close the connection. It performs the basic tasks of the core JDBC workflow (such as statement creation and execution), leaving application code to provide SQL and extract results.*

- *You can use JdbcTemplate within a DAO implementation through direct instantiation with a DataSource reference, or you can configure it in a Spring IoC container and give it to DAOs as a bean reference.*

- *We need to follow the below steps in order to configure JdbcTemplate inside a Spring Web application (With out Spring Boot)*

Step 1 : First we need to create a Data Source Bean inside Web application with the DB credentials like mentioned below.

```
@Bean
public DataSource myDataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/eazyschool");
    dataSource.setUsername("user");
    dataSource.setPassword("password");
    return dataSource;
}
```

USING JdbcTemplate

eazy bytes

Step 2 : Inside any Repository/DAO classes where we want to execute queries, we need to create a bean/object of JdbcTemplate by injecting data source bean

```
@Repository
public class PersonDAOImpl implements PersonDAO {
    JdbcTemplate jdbcTemplate;

    @Autowired
    public PersonDAOImpl(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }

}
```

- *Instances of the JdbcTemplate class are thread-safe, once configured. This is important because if needed we can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs (or repositories).*

USING JdbcTemplate

eazy bytes

Sample usage of JdbcTemplate for SELECT queries,

```
// The following query gets the number of rows in a table
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from person", Integer.class);
```

```
// The following query uses a bind variable
int countOfPersonsNamedJoe = this.jdbcTemplate.queryForObject(
        "select count(*) from person where first_name = ?", Integer.class, "Joe");
```

```
// The following query looks for a string column based on a condition:
String lastName = this.jdbcTemplate.queryForObject("select last_name from person where id = ?",
        String.class, 1212L);
```

USING JdbcTemplate

eazy bytes

Sample usage of JdbcTemplate for Updating (INSERT, UPDATE, and DELETE)

```
// The following example inserts a new entry
this.jdbcTemplate.update("insert into person (first_name, last_name) values (?, ?)",
        "Jon", "Doe");
```

```
// The following example updates an existing entry
this.jdbcTemplate.update("update person set last_name = ? where id = ?",
        "Maria", 5276L);
```

```
// The following example deletes an entry
this.jdbcTemplate.update("delete from person where id = ?", 5276L);
```

USING JdbcTemplate

eazy bytes

Other JdbcTemplate Operations

```
// You can use the execute(..) method to run any arbitrary SQL. Consequently, the method is often
// used for DDL statements.
this.jdbcTemplate.execute("create table person (id integer, name varchar(100))");
```

```
// The following example invokes a stored procedure
this.jdbcTemplate.update("call SUPPORT.REFRESH_PERSON_SUMMARY(?)",5276L);
```

USING ROWMAPPER

eazy bytes

- *RowMapper interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.*
- *RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.*
- *It defines only one method mapRow that accepts ResultSet instance and int as parameters. Below is the sample usage of it,*

```
private final RowMapper<Person> personRowMapper = (resultSet, rowNum) -> {
    Person person = new Person();
    person.setFirstName(resultSet.getString("first_name"));
    person.setLastName(resultSet.getString("last_name"));
    return person;
};
```

```
public List<Person> findAllPersons() {
    return this.jdbcTemplate.query("select first_name, last_name from person",personRowMapper);
}
```

QUICK TIP

eazy bytes

Do you know, if the column names in a table and field names inside a POJO/Bean are matching, then we can use BeanPropertyRowMapper which is provided by Spring framework.

Spring BeanPropertyRowMapper, class saves you a lot of time since we don't have to define the mappings like we do inside a RowMapper implementation.

```
public List<Holiday> findAllHolidays() {
    String sql = "SELECT * FROM HOLIDAYS";
    var rowMapper = BeanPropertyRowMapper.newInstance(Holiday.class);
    return jdbcTemplate.query(sql, rowMapper);
}
```

USING NamedParameterJdbcTemplate

eazy bytes

- *The NamedParameterJdbcTemplate class adds support for programming JDBC statements by using named parameters, as opposed to programming JDBC statements using only classic placeholder ('?') arguments. The NamedParameterJdbcTemplate class wraps a JdbcTemplate and delegates to the wrapped JdbcTemplate to do much of its work.*

- *The following example shows how to use NamedParameterJdbcTemplate*

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfPersonsByFirstName(String firstName) {
    String sql = "select count(*) from Person where first_name = :first_name";
    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name", firstName);
    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

QUICK TIP

eazy bytes

Do you know, with Spring Boot working with JdbcTemplate is very easy. Spring Boot auto configures DataSource, JdbcTemplate and NamedParameterJdbcTemplate classes based on the DB connection details mentioned in the property file and you can @Autowire them directly into your own repository classes, as shown in the following example.

You can customize some properties of the template by using the `spring.jdbc.template.*` properties, like mentioned below,

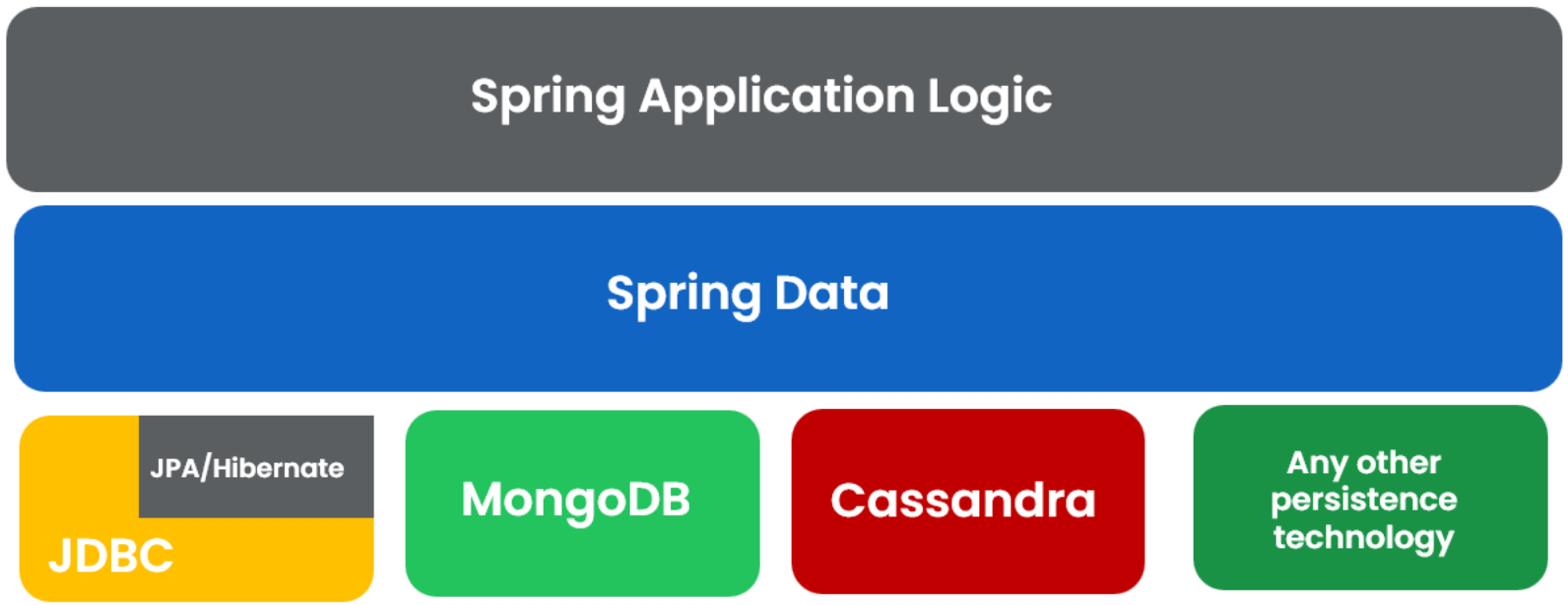
`spring.jdbc.template.max-rows=500`

```java
@Repository
public class HolidaysRepository {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public HolidaysRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
```

INTRO TO SPRING DATA

eazy bytes

Spring Data is a Spring ecosystem project that simplifies the persistence layer's development by providing implementations according to the persistence technology we use. This way, we only need to write a few lines of code to define the repositories of our Spring app.



Spring Data is a high-level layer that simplifies the persistence implementation by unifying the various technologies under the same abstractions.

INTRO TO SPRING DATA

eazy bytes

- *Whichever persistence technology your app uses, Spring Data provides a common set of interfaces (contracts) you extend to define the app's persistence capabilities.*
- *The central interface in the Spring Data repository abstraction is Repository*

Repository is the most abstract contract. If you extend this contract, your app recognizes the interface you write as a particular Spring Data repository. Still, you won't inherit any predefined operations (such as adding a new record, retrieving all the records, or getting a record by its primary key). The Repository interface doesn't declare any method (it is a marker interface).

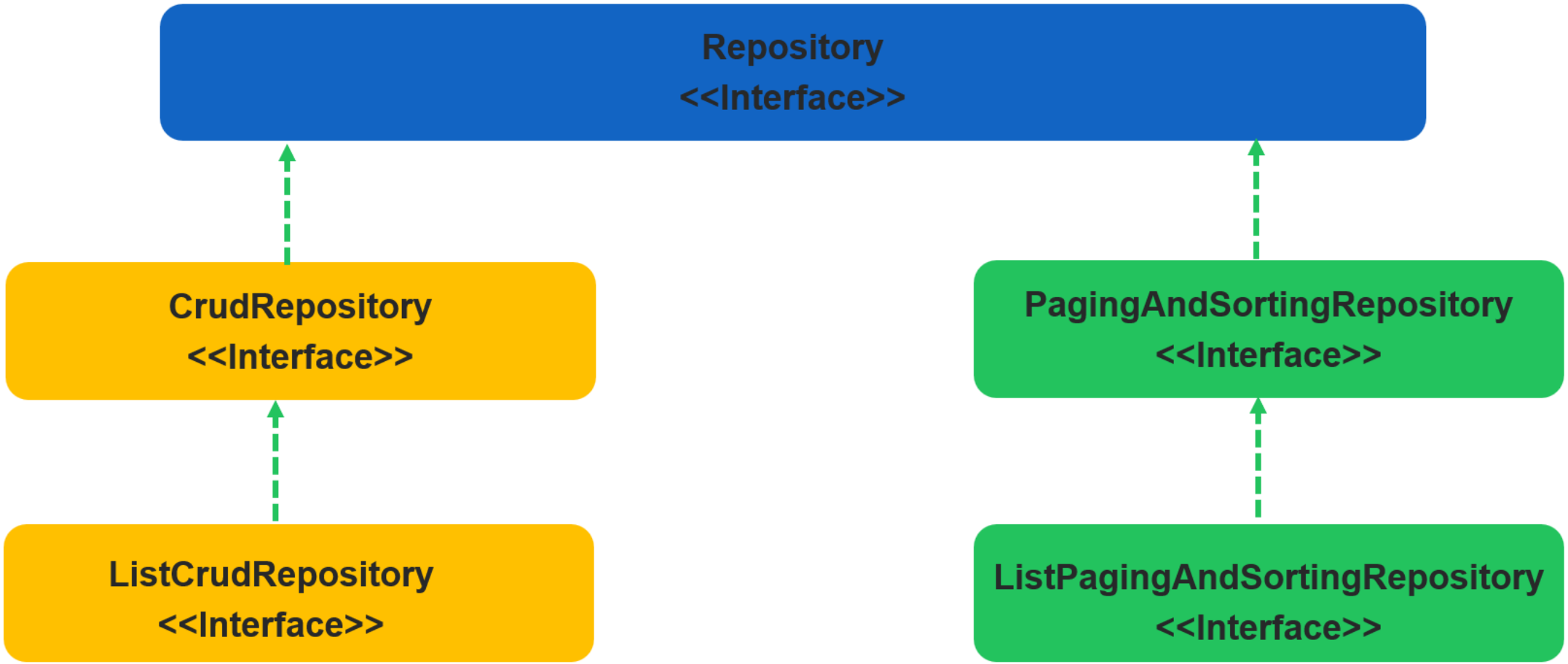
CrudRepository is the simplest Spring Data contract that also provides some persistence capabilities. If you extend this contract to define your app's persistence capabilities, you get the simplest operations for creating, retrieving, updating, and deleting records. **ListCrudRepository** is an extension to CrudRepository returning List instead of Iterable where ever applicable.

PagingAndSortingRepository provide methods to retrieve entities using the pagination and sorting abstraction. **ListPagingAndSortingRepository** an extension to PagingAndSortingRepository returning List instead of Iterable where ever applicable.

INTRO TO SPRING DATA

eazy bytes

To implement your app's repositories using Spring Data, you extend specific interfaces. The main interfaces that represent Spring Data contracts are Repository, CrudRepository, ListCrudRepository, PagingAndSortingRepository and ListPagingAndSortingRepository. You extend one of these contracts to implement your app's persistence capabilities.



QUICK TIP

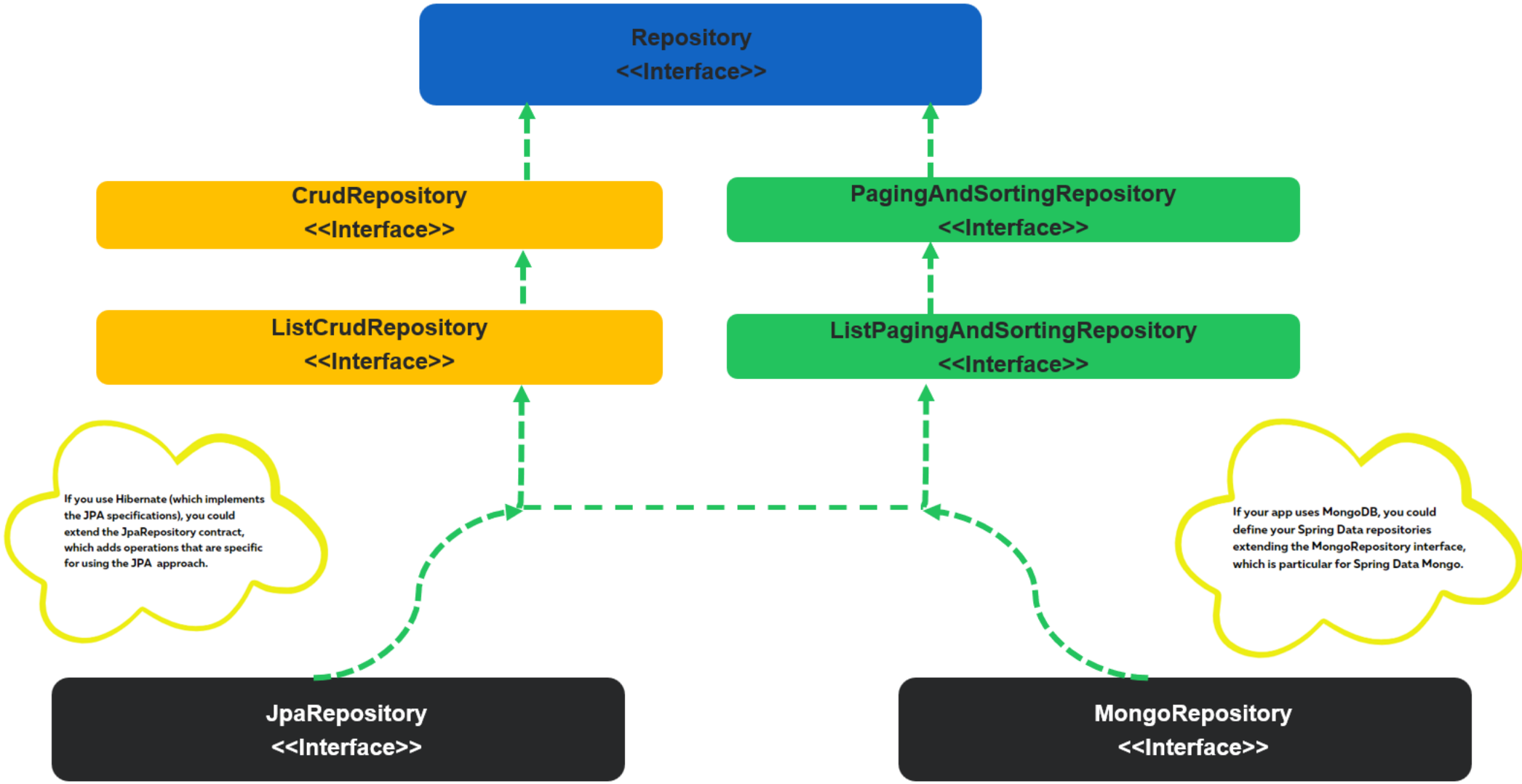
eazy bytes

Do you know,

- ✓ **We should not confuse between @Repository annotation and Spring Data Repository interface.**
- ✓ **Spring Data provides multiple interfaces that extend one another by following the principle called interface segregation. This helps apps to extend what they want instead of always following fat implementation.**
- ✓ **Some Spring Data modules might provide specific contracts to the technology they represent. For example, using Spring Data JPA, you also can extend the JpaRepository interface directly and similarly using Spring Data Mongo module to your app provides a particular contract named MongoRepository**

INTRO TO SPRING DATA

eazy bytes



INTRO TO SPRING DATA JPA

eazy bytes

- *Spring Data JPA is available to Spring Boot applications with the JPA starter. This starter dependency not only brings in Spring Data JPA, but also transitively includes Hibernate as the JPA implementation.*
- *Below is the maven dependency that we need to add to any SpringBoot projects in order to use Spring Data JPA,*

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

JPA is just a specification that defines an object-relational mapping (ORM) standard for storing, accessing, and managing Java objects in a relational database. Hibernate is the most popular and widely used implementation of JPA specifications. By default, Spring Data JPA uses Hibernate as a JPA provider.

- *We need to follow the below steps in order to query a DB using Spring Data JPA inside a Spring Boot applications,*

Step 1 : We need to indicate a java POJO class as an entity class by using annotations like @Entity, @Table, @Column

```
@Entity
@Table(name="contact_msg")
public class Contact extends BaseEntity{

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO,generator="native")
    @GenericGenerator(name = "native",strategy = "native")
    @Column(name = "contact_id")
    private int contactId;
```

INTRO TO SPRING DATA JPA

eazy bytes

Step 2 : We need to create interfaces for a given table entity by extending framework provided Repository interfaces. This helps us to run the basic CRUD operations on the table w/o writing method implementations.

```
@Repository
public interface ContactRepository extends CrudRepository<Contact, Integer> {


}
```

Step 3 : Enable JPA functionality and scanning by using the annotations @EnableJpaRepositories and @EntityScan

```
@SpringBootApplication
@EnableJpaRepositories("com.eazybytes.eazyschool.repository")
@EntityScan("com.eazybytes.eazyschool.model")
public class EazyschoolApplication {
```

INTRO TO SPRING DATA JPA

eazy bytes

Step 4 : We can inject repository beans into any controller/service classes and execute the required DB operations.

```java
@Service
public class ContactService {

    @Autowired
    private ContactRepository contactRepository;

    public boolean saveMessageDetails(Contact contact){
        boolean isSaved = false;
        Contact savedContact = contactRepository.save(contact);
        if(null != savedContact && savedContact.getContactId()>0) {
            isSaved = true;
        }
        return isSaved;
    }
```

Derived Query Methods in Spring Data JPA

eazy bytes

- *With Spring Data JPA, we can use the method names to derive a query and fetch the results w/o writing code manually like with traditional JDBC*
- *As a developer we Just need to define the query methods in a repository interface that extends one of the Spring Data's repositories such as CrudRepository. Spring Data JPA will create queries and implementation at runtime automatically by parsing these method names.*
- *Below are few examples,*

```
// find persons by last name
List<Person> findByLastName(String lastName);
```

```
// find person by email
Person findByEmail(String email);
```

```
// find person by email and last name
Person findByEmailAndLastname(String email, String lastname);
```

QUICK TIP

eazy bytes

Do you know a derived query method name has two main components separated by the first By keyword.

1. The introducer clause like find, read, query, count, or get which tells Spring Data JPA what you want to do with the method. This clause can contain further expressions, such as Distinct to set a distinct flag on the query to be created.

2. The criteria clause that starts after the first By keyword. The first By acts as a delimiter to indicate the start of the actual query criteria. The criteria clause is where you define conditions on entity properties and concatenate them with And and Or keywords.

Using readBy, getBy, and queryBy in place of findBy will behave the same. For example, readByEmail(String email) is same as findByEmail(String email).

Derived Query Methods in Spring Data JPA

eazy bytes

Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
Distinct	findDistinctByLastnameAndFirstname	select distinct … where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname,findByFirstnameIs,findBy FirstnameEquals	… where x.firstname = ?1
Between	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1

Derived Query Methods in Spring Data JPA

eazy bytes

Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
IsNull, Null	findByAge(Is)Null	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1

Derived Query Methods in Spring Data JPA

eazy bytes

Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstname) = UPPER(?1)

Spring Data JPA is a powerful tool that provides an extra layer of abstraction on top of an existing JPA providers like Hibernate. The derived query feature is one of the most loved features of Spring Data JPA.

Auditing Support By Spring Data JPA

eazy bytes

- *Spring Data provides sophisticated support to transparently keep track of who created or changed an entity and when the change happened. To benefit from that functionality, you have to equip your entity classes with auditing metadata that can be defined either using annotations or by implementing an interface.*
- *Additionally, auditing has to be enabled either through Annotation configuration or XML configuration to register the required infrastructure components.*
- *Below are the steps that needs to be followed,*

Step 1 : We need to use the annotations to indicate the audit related columns inside DB tables. Spring Data JPA ships with an entity listener that can be used to trigger the capturing of auditing information. We must register the AuditingEntityListener to be used for all the required entities.

```java
@Data
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public class BaseEntity {

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdAt;
    @CreatedBy
    @Column(updatable = false)
    private String createdBy;
    @LastModifiedDate
    @Column(insertable = false)
    private LocalDateTime updatedAt;
    @LastModifiedBy
    @Column(insertable = false)
    private String updatedBy;
}
```

@CreatedDate, @CreatedBy, @LastModifiedDate, @LastModifiedBy are the key annotations that support JPA auditing

Auditing Support By Spring Data JPA

eazy bytes

Step 2 : Date related info will be fetched from the server by JPA but for CreatedBy & UpdatedBy we need to let JPA know how to fetch that info by implementing AuditorAware interface like shown below,

```
@Component("auditAwareImpl")
public class AuditAwareImpl implements AuditorAware<String> {

    @Override
    public Optional<String> getCurrentAuditor() {
            return Optional.ofNullable(SecurityContextHolder.getContext()
                    .getAuthentication().getName());
    }
}
```

Step 3 : Enable JPA auditing by annotating a configuration class with the @EnableJpaAuditing annotation.

```
@SpringBootApplication
@EnableJpaRepositories("com.eazybytes.eazyschool.repository")
@EntityScan("com.eazybytes.eazyschool.model")
@EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")
public class EazyschoolApplication {
```

QUICK TIP

eazy bytes

Do you know we can print the queries that are being formed and executed by Spring Data JPA by enabling the below properties,

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format_sql=true

- ✓ **show-sql property will print the query on the console/logs where as format_sql property will print the queries in a readable friendly style.**
- ✓ **But please make sure to leverage them in non-prod environments only as they impact the performance of the web application.**

Spring MVC Custom Validations

eazy bytes

We have seen before using Bean validations like Max, Min, Size etc. we can do validations on the input received. Now let's try to define custom validations that are specific to our business requirements. For the same we need to follow the below steps,

Step 1 : Suppose if we have a requirement to not allow some weak passwords inside our user registration form, we first need to create a custom annotation like below. Here we need to provide the class name where the actual validation logic present.

```
@Documented
@Constraint(validatedBy = PasswordStrengthValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface PasswordValidator {
    String message() default "Please choose a strong password";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Spring MVC Custom Validations

eazy bytes

Step 2 : We need to create a class that implements ConstraintValidator interface and overriding the isValid() method like shown below,

```
public class PasswordStrengthValidator implements
        ConstraintValidator<PasswordValidator, String> {

    List<String> weakPasswords;

    @Override
    public void initialize(PasswordValidator passwordValidator) {
        weakPasswords = Arrays.asList("12345", "password", "qwerty");
    }

    @Override
    public boolean isValid(String passwordField,
                           ConstraintValidatorContext cxt) {
        return passwordField != null && (!weakPasswords.contains(passwordField));
    }
}
```

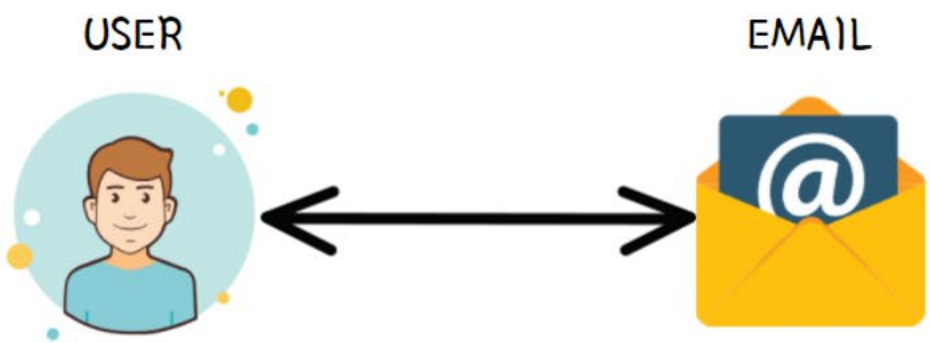
Step 3 : Finally we can mention the annotation that we created on top of the field inside a POJO class,

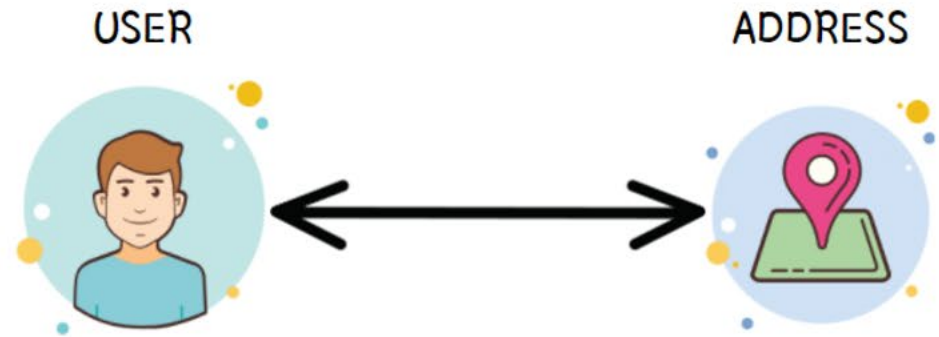
```
@NotBlank(message="Password must not be blank")
@Size(min=5, message="Password must be at least 5 characters long")
@PasswordValidator
private String pwd;
```

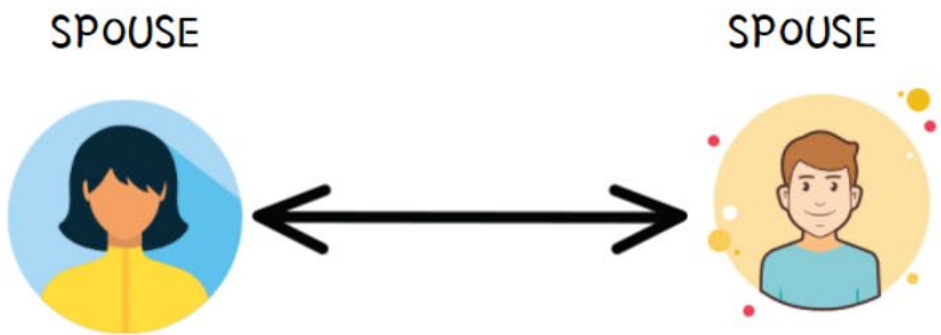
One to One Relationship inside JPA

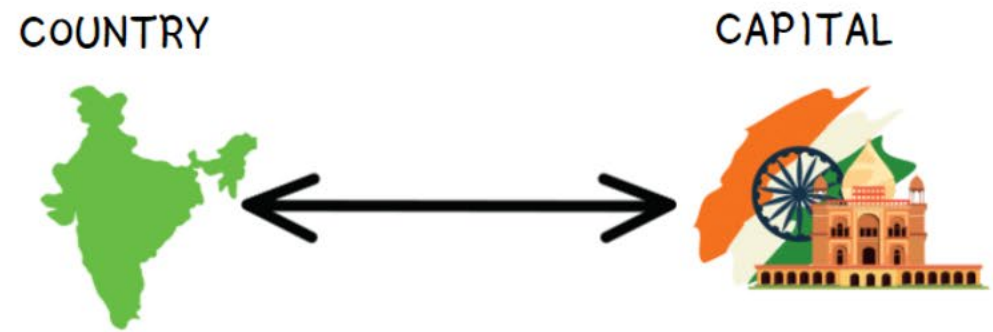
eazy bytes

- *First, what is a one-to-one relationship? It's a relationship where a record in one entity (table) is associated with exactly one record in another entity (table).*
- *Below are few real-life examples of one-to-one relationships,*









One to One Relationship inside JPA

eazy bytes

- *Spring Data JPA allow developers to build one-to-one relationship between the entities with simple configurations. For example if we want to build a one-to-one between Person and Address entities, then we can configure like mentioned below.*

```
@Data
@Entity
public class Person {

    @OneToOne(fetch = FetchType.EAGER,cascade = CascadeType.ALL, targetEntity = Address.class)
    @JoinColumn(name = "address_id", referencedColumnName = "addressId",nullable = true)
    private Address address;
}
```

- *In Spring Data JPA, a one-to-one relationship between two entities is declared by using the @OneToOne annotation. Using it we can configure FetchType, cascade effects, targetEntity.*
- *The @JoinColumn annotation is used to specify the foreign key column relationship details between the 2 entities. "name" defines the name of the foreign key column, referencedColumnName indicates the field name inside the target entity class, nullable defines whether foreign key column can be nullable or not.*

JPA FETCH

eazy bytes

- *Based on the fetch configurations that developer has done, JPA allows entities to load the objects with which they have a relationship.*

We can declare the fetch value in the @OneToOne, @OneToMany, @ManyToOne and @ManyToMany annotations. These annotations have an attribute called fetch that serves to indicate the type of fetch we want to perform. It have two valid values: FetchType.EAGER and FetchType.LAZY

With LAZY configuration, we are telling JPA that we want to lazily load the relation entities, so when retrieving an entity, its relations will not be loaded until unless we try refer the related entity using getter method. On the contrary, with EAGER it will load it's relation entities as well.

By default all ToMany relationships are LAZY, while ToOne relationships are EAGER.

eazy bytes

Key points of Cascade Types

Intro to Cascade

✓JPA allows us to propagate entity state changes from Parents to Child entities. This concept is calling Cascading in JPA.

✓The cascade configuration option accepts an array of CascadeTypes.

Cascade Types

The cascade types supported by JPA are as below:

1) CascadeType.PERSIST
2) CascadeType.MERGE
3) CascadeType.REFRESH
4) CascadeType.REMOVE
5) CascadeType.DETACH
6) CascadeType.ALL

Best Practices

✓ Cascading makes sense only for Parent – Child associations (where the Parent entity state transition being cascaded to its Child entities). Cascading from Child to Parent is not very useful and not recommended.

✓ There is no default cascade type in JPA. By default, no operation is cascaded.

CASCADE TYPES

eazy bytes

CascadeType.PERSIST : means that save() or persist() operations cascade to related entities.

CascadeType.MERGE : means that related entities are merged when the owning entity is merged.

CascadeType.REFRESH : means the child entity also gets reloaded from the database whenever the parent entity is refreshed.

CascadeType.REMOVE : means propagates the remove operation from parent to child entity.

CascadeType.DETACH : means detach all child entities if a “manual detach” occurs for parent.

CascadeType.ALL : is shorthand for all of the above cascade operations.

Spring Security AuthenticationProvider

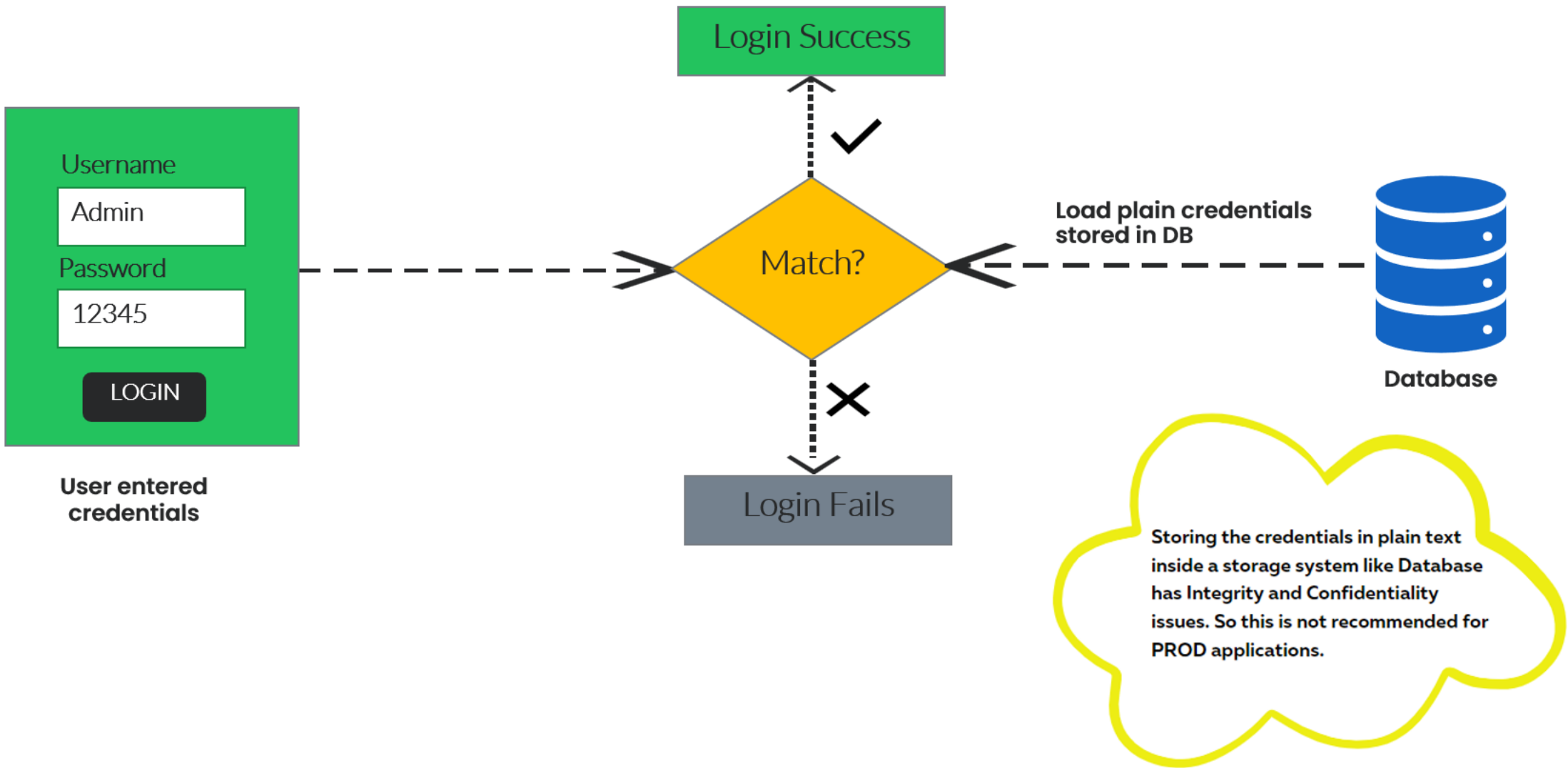
eazy bytes

- *As of now we are performing the login operation using the inMemoryAuthentication. But the idle way is to perform login check against a DB table or any other storage system which is more secure.*
- *For the same, Spring Security allow us to write our own custom logic to authenticate a user based on our requirements by implementing AuthenticationProvider interface. Below is the sample implementation of the same,*

```
@Component
public class EazySchoolUsernamePwdAuthenticationProvider implements AuthenticationProvider
{
    @Override
    public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
        String name = authentication.getName();
        String password = authentication.getCredentials().toString();
        if (authenticateUserBasedonYourRequirement()) {
            return new UsernamePasswordAuthenticationToken(
                    name, password, new ArrayList<>());
        } else {
            return null;
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

How Passwords Validated W/O PasswordEncoder

eazy bytes



eazy bytes

Different ways of Pwd management

Encoding

- ✓ Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography.
- ✓ It involves no secret and completely reversible.
- ✓ Encoding can't be used for securing data. Below are the various publicly available algorithms used for encoding.

Ex: ASCII, BASE64, UNICODE

Encryption

- ✓ Encryption is defined as the process of transforming data in such a way that guarantees confidentiality.
- ✓ To achieve confidentiality,encryption requires the use of a secret which, in cryptographic terms, we call a "key".
- ✓ Encryption can be reversible by using decryption with the help of the "key". As long as the "key" is confidential, encryption can be considered as secured.

Hashing

- ✓ In hashing, data is converted to the hash value using some hashing function.
- ✓ Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.
- ✓ Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data

QUICK TIP

eazy bytes

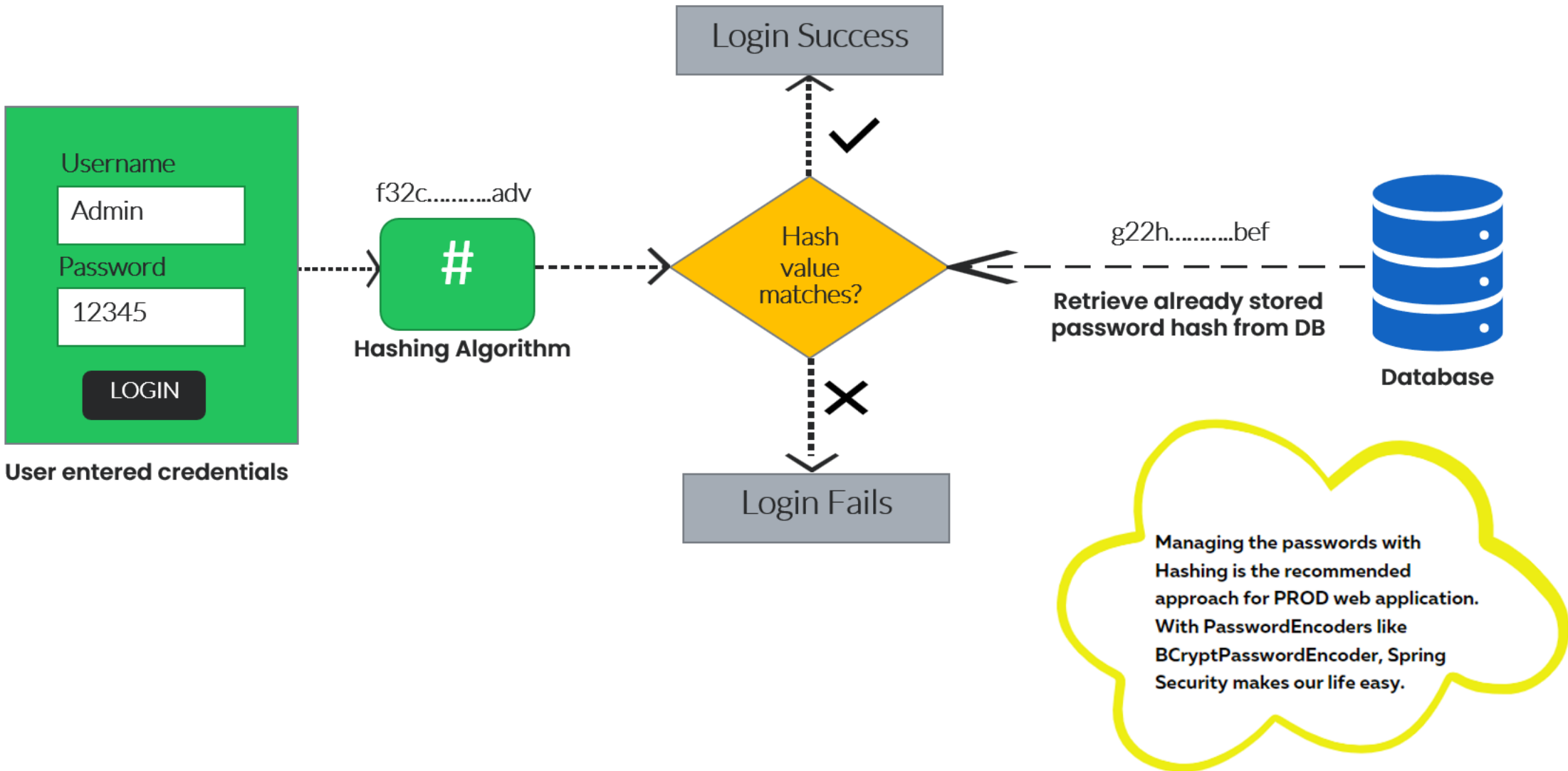
Do you know Spring Security provides various PasswordEncoders to help developers with hashing of the secured data like password.

Different Implementations of PasswordEncoders provided by Spring Security

- ✓ **NoOpPasswordEncoder (No hashing stores in plain text)**
- ✓ **StandardPasswordEncoder**
- ✓ **Pbkdf2PasswordEncoder**
- ✓ **BCryptPasswordEncoder (Most Commonly used)**
- ✓ **SCryptPasswordEncoder**

How Passwords Validated With Hashing &PasswordEncoder

eazy bytes



QUICK TIP

eazy bytes

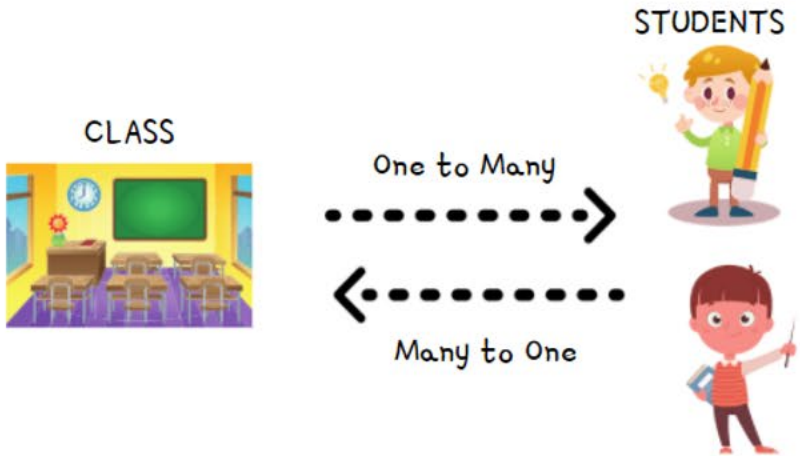
Do you know we can disable the javax bean validations by the Spring Data JPA using the below property,

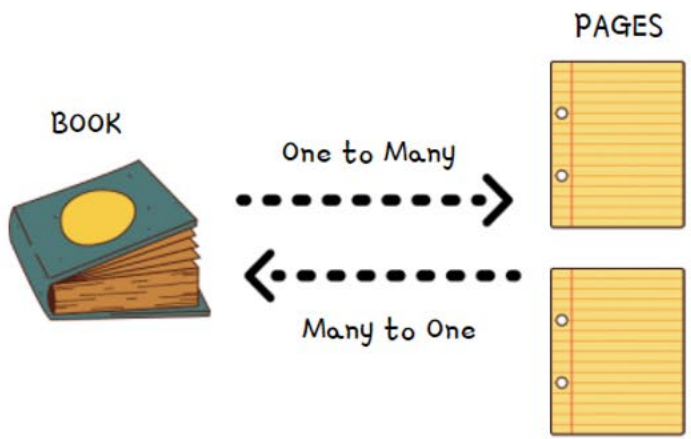
spring.jpa.properties.javax.persistence.validation.mode=none

One to Many & Many to One Relationship inside JPA

eazy bytes

- *A one-to-many relationship refers to the relationship between two entities/tables A and B in which one element/row of A may only be linked to many elements/rows of B, but a member of B is linked to only one element/row of A.*
- *The opposite of one-to-many is many-to-one relationship.*
- *Below are few real-life examples of one-to-many and many-to-one relationships,*





One to Many & Many to One Relationship inside JPA

eazy bytes

- *Spring Data JPA allow developers to build one-to-many & many-to-one relationships between the entities with simple configurations. Below are the sample configurations between Class and Persons,*

```java
@Entity
public class Person extends BaseEntity{

    @ManyToOne(fetch = FetchType.LAZY, optional = true)
    @JoinColumn(name = "class_id", referencedColumnName = "classId", nullable = true)
    private EazyClass eazyClass;
}
```

- *The @ManyToOne annotation is used to define a many-to-one relationship between two entities. The child entity, that has the join column, is called the owner of the relationship.*
- *The @JoinColumn annotation is used to specify the foreign key column details.*

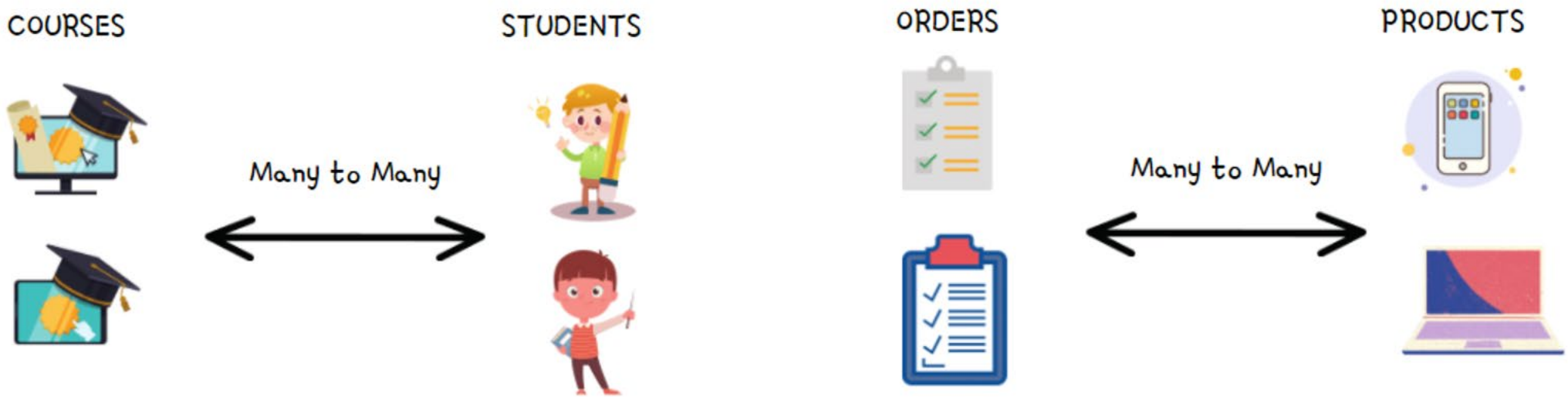
- *A one-to-many relationship between two entities is defined by using the @OneToMany annotation. It also declares the mappedBy element to indicate the entity that owns the bidirectional relationship. Usually, the child entity is one that owns the relationship and the parent entity contains the @OneToMany annotation.*

```java
@Entity
@Table(name = "class")
public class EazyClass extends BaseEntity{

    @OneToMany(mappedBy = "eazyClass", fetch = FetchType.LAZY,
            cascade = CascadeType.PERSIST,targetEntity = Person.class)
    private Set<Person> persons;
}
```

Many to Many inside JPA

eazy bytes

- *A many-to-many relationship refers to the relationship between two entities/tables A and B in which one element/row of A are associated with many elements/rows of B and vice versa.*
- *Below are few real-life examples of many-to-many relationship,*



Many to Many inside JPA

eazy bytes

- *Spring Data JPA allow developers to build many-to-many relationship between the entities with simple configurations. Below are the sample configurations between Courses and Persons,*

```
@Entity
public class Person extends BaseEntity{

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.PERSIST)
    @JoinTable(name = "person_courses",
            joinColumns = {
                    @JoinColumn(name = "person_id", referencedColumnName = "personId")},
            inverseJoinColumns = {
                    @JoinColumn(name = "course_id", referencedColumnName = "courseId")})
    private Set<Courses> courses = new HashSet<>();
}
```

- *A many-to-many relationship between two entities is defined by using the @ManyToMany annotation.*
- *The @JoinTable annotation defines the join table between two entities*

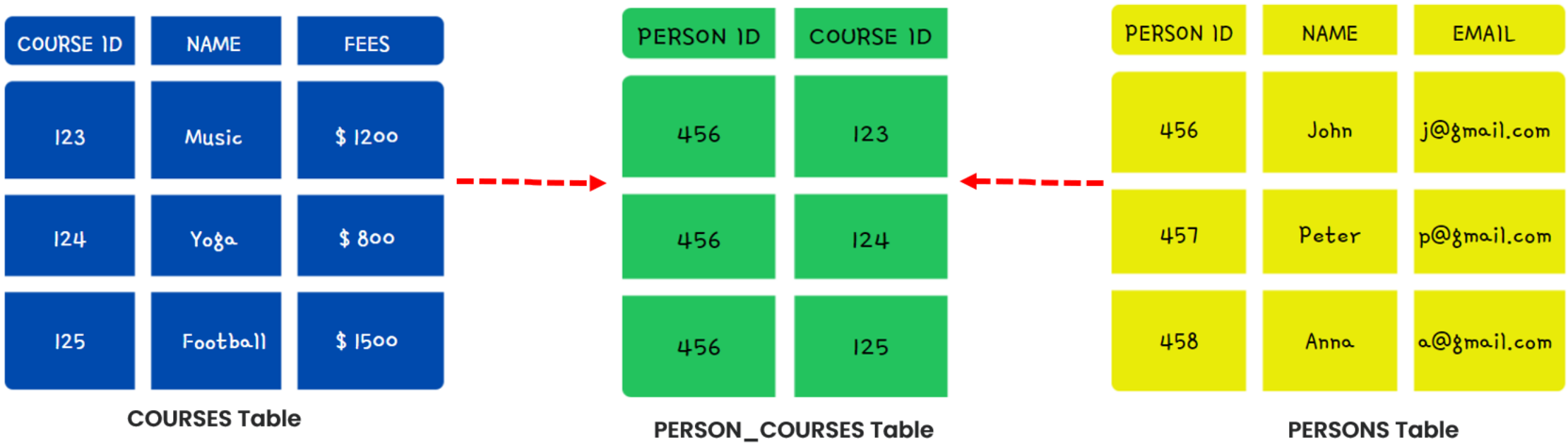
- *In a bidirectional relationship of @ManyToMany, only one entity can own the relationship. Here we choose Courses as the owning entity. We usually mention mappedBy parameter on the owning entity.*

```
@Entity
public class Courses extends BaseEntity{

    @ManyToMany(mappedBy = "courses", fetch = FetchType.EAGER
            ,cascade = CascadeType.PERSIST)
    private Set<Person> persons = new HashSet<>();
}
```

Many to Many inside JPA

eazy bytes

- *Why do we really need a third table in Many to Many relationship? Lets assume we have below tables and records which has many-many relationship between them.*
- *If we need to represent multiple courses that a same student enrolled, the best way is to maintain a middle third table like shown below. Otherwise we need to maintain lot of duplicate rows inside Person table.*



eazy bytes

Sorting

Introduction

✓Spring Data JPA supports Sorting to Query results with easier configurations.

✓Spring Data JPA provides default implementations of Sorting with the help of PagingAndSortingRepository interface

✓There are two ways to achieve Sorting in Spring Data JPA:

1) Static Sorting
2) Dynamic Sorting

Static Sorting

✓ Static sorting refers to the mechanism where the retrieved data is always sorted by specified columns and directions. The columns and sort directions are defined at the development time and cannot be changed at runtime.

✓ Below is an examples of Static Sorting,

```
List<Person> findByOrderByNameDesc()
```

Dynamic Sorting

✓ By using dynamic sorting, you can choose the sorting column and direction at runtime to sort the query results.

✓ Dynamic sorting provides more flexibility in choosing sort columns and directions with the help of Sort parameter to your query method. The Sort class is just a specification that provides sorting options for database queries. Below is an example,

```
Sort sort=Sort.by("name").descending()
.and(Sort.by("age"));
```

Pagination

Introduction

- ✓ Spring Data JPA supports Pagination which helps easy to manage & display large amount of data in various pages inside web applications
- ✓ Just like the special Sort parameter, we have for the dynamic sorting, Spring Data JPA supports another special parameter called Pageable for paginating the query results.
- ✓ We can combine both pagination and dynamic sorting with the help of Pageable

Dynamic Sorting

- ✓ Whenever we want to apply pagination to query results, all we need to do is just add Pageable parameter to the query method definition and set the return by Page<T> like below,

```
Pageable pageable = PageRequest.of(0, 5, Sort.by("name").
descending());
```

```
Page<Person> findByName(String name, Pageable pageable);
```

eazy bytes

Custom Queries with JPA

Introduction

✓ Derived queries are good as long as they are not complex. As the number of query parameters goes beyond 3 to 4, you need a more flexible strategy.

✓ For such complicated or custom scenarios, Spring Data JPA allow developers to write their own queries with the help of below annotations,

`@Query`
`@NamedQuery`
`@NamedNativeQuery`

@Query Annotation

✓ The @Query annotation defines queries directly on repository methods. This gives you full flexibility to run any query without following the method naming conventions.

✓ With the help of @Query annotation we can write queries in the form of JPQL or Native SQL query.

✓ When ever we are writing a native SQL query then we need to mention `nativeQuery = true` inside @Query annotation

Named Queries

✓ For bigger applications where they may have 1000s of queries scatted across the application, it would make sense for them to maintain all these queries in a single place logically by using annotations, properties and XML files.

✓ We can create named queries easily with the below annotations on top of an entity class,

`@NamedQuery` - Used to define a JPQL named query.

`@NamedNativeQuery` - Used to define a native SQL named query.

JPQL inside Spring Data JPA

eazy bytes

JPQL

Introduction

- ✓ The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification.
- ✓ JPQL is used to make queries against entities stored in a relational database. It is heavily inspired by SQL, and its queries resemble SQL queries in syntax, but operate against JPA entity objects rather than directly with database tables.
- ✓ The only drawback of using JPQL is that it supports a subset of the SQL standard. So, it may not be a great choice for complex queries.

JPQL Example

- ✓ Below is an example of JPQL. You can observe we are using the entity names and fields present inside it instead of using table and column names,

```
@Query("SELECT c FROM Contact c WHERE c.contactId = ?1
ORDER BY c.createdAt DESC")
List<Contact> findByIdOrderByCreatedDesc(long id);
```

Sorting Examples

eazy bytes

Static Sorting using derived query from method name

```
List<Person> findByNameOrderByAgeDesc(String name);
```

Static Sorting using @Query annotation with JPQL

```
@Query("SELECT p FROM person p WHERE p.age > ?1 ORDER BY p.name DESC")
List<Person> findByAgeGreaterThanJPQL(int age);
```

Static Sorting using @Query annotation with native query

```
@Query(value = "SELECT * FROM person p WHERE p.name = :givenName ORDER BY p.age ASC",
        nativeQuery = true)
List<Person> findByGivenNameNativeSQL(@Param("givenName") String givenName);
```

Sorting Examples

eazy bytes

Static Sorting using NamedQuery & NamedNativeQuery annotations

```
@NamedQuery(name = "Person.findByAgeGreaterThanNamedJPQL",
        query = "SELECT p FROM Person p WHERE p.age > :age ORDER BY p.name ASC")
@NamedNativeQuery(name = "Person.findAllNamedNativeSQL",
        query = "SELECT * FROM person p ORDER BY p.age DESC")
@Entity
public class Person extends BaseEntity{
```

Dynamic Sorting using Sort parameter. The Sort parameter can be passed with @Query, @NamedQuery annotations. Sort fields can be add dynamically as well based on the request params from the UI/API.

```
Sort sort = Sort.by("name").descending().and(Sort.by("age"));
List<Person> persons = personRepository.findByName("John", sort);
```

Spring Data JPA does not support dynamic sorting for native SQL queries, as it would require the updating of the actual SQL query defined, which cannot be done correctly by the Spring Data JPA.

Pagination Examples

eazy bytes

Below is an example where we are telling to JPA to fetch the first page by considering the total page size as 5

```
Pageable pageable = PageRequest.of(0, 5);
Page<Contact> msgPage = contactRepository.findByStatus("Open",pageable);
```

Below is an example where we are applying both pagination & sorting dynamically based on the input received.

```
public Page<Contact> findMsgsWithOpenStatus(int pageNum,String sortField,
                                            String sortDir){
    int pageSize = 5;
    Pageable pageable = PageRequest.of(pageNum - 1, pageSize,
            sortDir.equals("asc") ? Sort.by(sortField).ascending()
                    : Sort.by(sortField).descending());
    Page<Contact> msgPage = contactRepository.findByStatus("Open",pageable);
    return msgPage;
}
```

Dynamic sorting is not supported by named queries. So while using Pagination along with Named Queries, make sure that Pageable instance does not contain a Sort object.

@Query Examples

eazy bytes

@Query example with positional parameters

```
// Positional parameters
@Query("SELECT c FROM Contact c WHERE c.status = ?1 AND c.name = ?2 ORDER BY c.createdAt DESC")
List<Contact> findByGivenQueryOrderByCreatedDesc(String status, String name);
```

@Query example with named parameters

```
// named parameters
@Query("SELECT c FROM Contact c WHERE c.status = :status AND c.name = :name ORDER BY c.createdAt DESC")
List<Contact> findByGivenQueryOrderByCreatedDesc(@Param("status") String status,
                                                 @Param("name") String name);
```

@Query Examples

eazy bytes

Using @Query, we can also run UPDATE, DELETE, INSERT as well

```
@Transactional
@Modifying
@Query("UPDATE Contact c SET c.status = ?1 WHERE c.contactId = ?2")
int updateStatusById(String status, int id);
```

Whenever we are using queries that change the state of the database, they should be treated differently. We need to explicitly tell Spring Data JPA that our custom query changes the data by annotating the repository method with an additional @Modifying annotation. It will then execute the custom query as an update operation.

On whichever method/class we declare @Transactional the boundary of transaction starts and boundary ends when method execution completes. Lets say you are updating entity1 and entity2. Now while saving entity2 an exception occur, then as enitiy1 comes in same transaction so entity1 will be rollback with entity2.Any exception will result in rollback of all JPA transactions with DB.

@NamedQuery & @ NamedNativeQuery Examples

eazy bytes

@NamedQuery example declared on top of the entity class

```
@Entity
@NamedQuery(name="Contact.findOpenMsgs",query = "SELECT c FROM Contact c WHERE c.status = :status")
public class Contact extends BaseEntity{
```

@NamedNativeQuery example declared on top of the entity class

```
@Entity
@NamedNativeQuery(name = "Contact.findOpenMsgsNative",
        query = "SELECT * FROM contact_msg c WHERE c.status = :status",resultClass = Contact.class)
public class Contact extends BaseEntity{
```

For @NamedQuery as long as the method name inside Repository class matches with the name of the query we should be good. Where as for @NamedNativeQuery apart from query name & method name match, we should also mention @Query(nativeQuery = true) on top of the Repository method.

QUICK TIP

eazy bytes

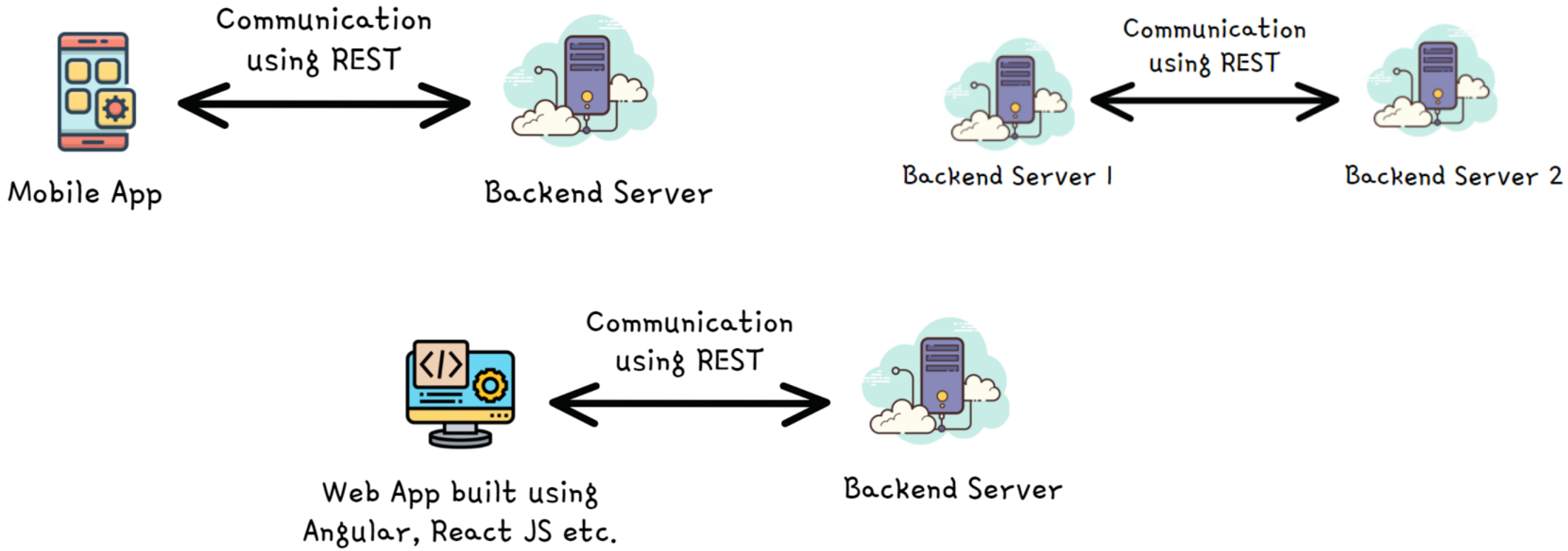
Do you know we can create multiple named queries and named native queries using the annotations @NamedQueries and @NamedNativeQueries. Below is the syntax of the same,

```
@NamedQueries({
        @NamedQuery(name = "one", query = "?"),
        @NamedQuery(name = "two", query = "?")
})

@NamedNativeQueries({
        @NamedNativeQuery(name = "one", query = "?“, resultClass = ?),
        @NamedNativeQuery(name = "two", query = "?“, , resultClass = ?)
})
```

Implementing REST Services

eazy bytes

- *REST (Representational state transfer) services are one of the most often encountered ways to implement communication between two web apps. REST offers access to functionality the server exposes through endpoints a client can call.*
- *Below are the different use cases where REST services are being used most frequently these days,*



Implementing REST Services

eazy bytes

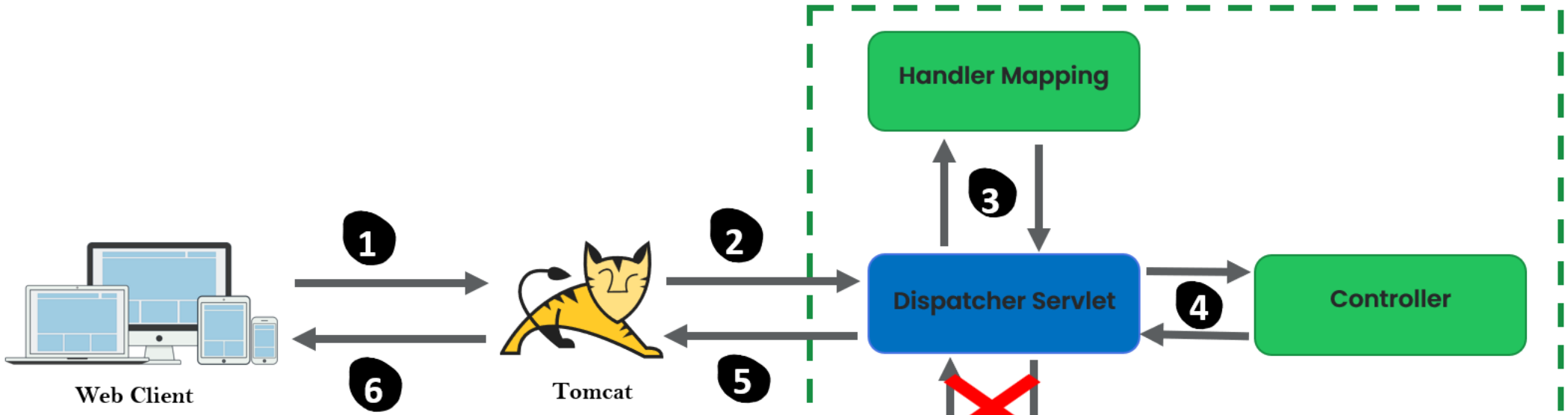
Below is the sample code implementing Rest Service using Spring MVC style but only with the addition of @ResponseBody annotation

```java
@Controller
public class ContactRestController {

    @Autowired
    ContactRepository contactRepository;

    @GetMapping("/getMessagesByStatus")
    @ResponseBody
    public List<Contact> getMessagesByStatus(@RequestParam(name = "status") String status) {
        return contactRepository.findByStatus(status);
    }

}
```

The @ResponseBody annotation tells the dispatcher servlet that the controller's action will not return a view name but the data sent directly in the HTTP response.

SPRING MVC ARCHITECTURE WITH REST SERVICE

eazy bytes



Spring MVC Web Application with REST Service

When implementing REST endpoints, the Spring MVC flow changes. The app no longer needs a view resolver because the client needs the data returned by the controller's action directly. Once the controller's action completes, the dispatcher servlet returns the HTTP response without rendering any view.

Implementing REST Services

eazy bytes

Instead of mentioning @ResponseBody on each method inside a Controller class which is a duplicate code, Spring offers the @RestController annotation, a combination of @Controller and @ResponseBody.

```java
@RestController
public class ContactRestController {

    @Autowired
    ContactRepository contactRepository;

    @GetMapping("/getMessagesByStatus")
    public List<Contact> getMessagesByStatus(@RequestParam(name = "status") String status) {
        return contactRepository.findByStatus(status);
    }

}
```

@RestController = @Controller + @ResponseBody

Different Annotation & Classes in REST

eazy bytes

@RestController – can be used to put on top of a call. This will save developers from mentioning @ResponseBody on each methods

@ResponseBody – can be used on top of a method to build a Rest API when we are using @Controller on top of a Java class

ResponseEntity<T> - Allow developers to send response body, status, and headers on the HTTP response.

@RestControllerAdvice – is used to mark the class as a REST controller advice. Along with @ExceptionHandler, this can be used to handle exceptions globally inside app.

RequestEntity<T> - Allow developers to receive the request body, header in a HTTP request.

@RequestHeader & @RequestBody – is used to receive the request body and header individually.

CROSS-ORIGIN RESOURCE SHARING (CORS)

eazy bytes

CORS is a protocol that enables scripts running on a browser client to interact with resources from a different origin. For example, if a UI app wishes to make an API call running on a different domain, it would be blocked from doing so by default due to CORS. So CORS is not a security issue/attack but the default protection provided by browsers to stop sharing the data/communication between different origins.

"other origins" means the URL being accessed differs from the location that the JavaScript is running from, by having:

- a different scheme (HTTP or HTTPS)
- a different domain
- a different port

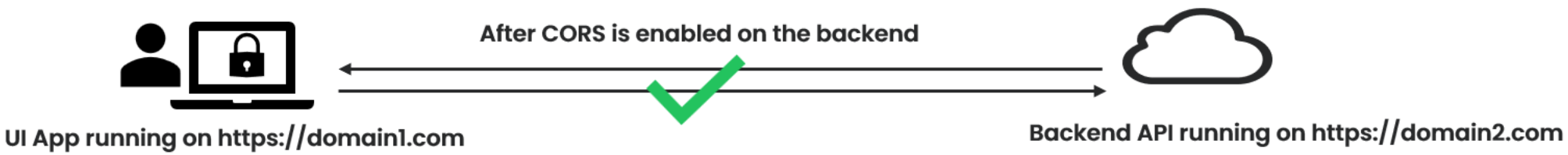
CROSS-ORIGIN RESOURCE SHARING (CORS)

eazy bytes

If we have a valid scenario, where a Web APP UI deployed on a server is trying to communicate with a REST service deployed on another server, then these kind of communications we can allow with the help of @CrossOrigin annotation. @CrossOrigin allows clients from any domain to consume the API.

@CrossOrigin annotation can be mentioned on top of a class or method like mentioned below,

```
@CrossOrigin(origins = "http://localhost:8080") // Will allow on specified domain

@CrossOrigin(origins = "*") // Will allow any domain
```



QUICK TIP

eazy bytes

Do you know Jackson library provide annotations to control the way we send data in REST API JSON response. Below are the few annotations,

@JsonProperty – This annotation can be mentioned on top of any field of a POJO class. While sending the JSON response, instead of sending the field name Jackson will send the property name that we mentioned.

```
@JsonProperty("person_name")
private String name;
```

@JsonIgnore – This annotation will make sure to not send the information present inside the given field. This is useful to filter the data which is sensitive or unnecessary in the response.

```
@JsonIgnore
private LocalDateTime createdAt;
```

We can also use @JsonIgnoreProperties(value = { "createdAt" }) on top of a POJO class if we want to mention multiple fields.

Consuming REST Services

eazy bytes

- *Apart from building the Rest service, often we may need to consume the Rest Services exposed by other third party vendors. So knowing how to consume Rest services is equally important.*
- *Below are the most commonly used approaches that are provided by Spring framework,*

OpenFeign - A tool offered by the Spring Cloud project. Using this very similar to like how build Repositories with Spring Data JPA. In similar way we just need to write interfaces but not implementation code.

RestTemplate—A well-known tool developers have been using since Spring 3 to call REST endpoints. RestTemplate is often used today in Spring apps. But this is deprecated in the favor of WebClient.

WebClient— Created as part of the Spring Web Reactive module, and will be replacing the classic RestTemplate. This is introduced to support all modes of invocation like Sync and Async (non-blocking)

Consuming REST Services using OpenFeign

eazy bytes

In order to consume the REST services using OpenFeign, we need to follow the below steps.

Step 1 : After adding all the required dependencies inside the pom.xml, we need to create a proxy interface with all the details around API that we are going to consume. Inside the interface we need to create method name matching the details of the destination API method we are going to consume,

```
@FeignClient(name = "contact", url = "http://localhost:8080/api/contact",
    configuration = ProjectConfiguration.class)
public interface ContactProxy {

    @RequestMapping(method = RequestMethod.GET, value = "/getMessagesByStatus")
    @Headers(value = "Content-Type: application/json")
    public List<Contact> getMessagesByStatus(@RequestParam("status") String status);

}
```

Consuming REST Services using OpenFeign

eazy bytes

Step 2 : If we need to send the authentication details, then create a bean with the required details,

```
@Bean
public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
    return new BasicAuthRequestInterceptor("admin@eazyschool.com", "admin");
}
```

Step 3 : Finally we are good to use the proxy object to make a Rest call like shown below,

```
@Autowired
ContactProxy contactProxy;

@GetMapping("/getMessages")
public List<Contact> getMessages(@RequestParam("status") String status) {
    return contactProxy.getMessagesByStatus(status);
}
```

Consuming REST Services using RestTemplate

eazy bytes

In order to consume the REST services using RestTemplate, we need to follow the below steps.

Step 1 : After adding all the required dependencies inside the pom.xml, we need to create a RestTemplate bean along with the authentication details if any like below,

```
@Bean
public RestTemplate restTemplate() {
    RestTemplateBuilder restTemplateBuilder =
            new RestTemplateBuilder();
    return restTemplateBuilder.basicAuthentication
            ("admin@eazyschool.com", "admin").build();
}
```

Consuming REST Services using RestTemplate

eazy bytes

Step 2 : Using RestTemplate methods like exchange(), we can consume a Rest Service like mentioned below,

```
@PostMapping("/saveMsg")
public ResponseEntity<Response> saveMsg(@RequestBody Contact contact){
    String uri = "http://localhost:8080/api/contact/saveMsg";
    HttpHeaders headers = new HttpHeaders();
    headers.add("invocationFrom","RestTemplate");
    HttpEntity<Contact> httpEntity = new HttpEntity<>(contact, headers);
    ResponseEntity<Response> responseEntity = restTemplate.exchange(uri, HttpMethod.POST,
                    httpEntity,Response.class);
    return responseEntity;
}
```

Consuming REST Services using WebClient

eazy bytes

In order to consume the REST services using WebClient, we need to follow the below steps.

Step 1 : After adding all the required dependencies inside the pom.xml, we need to create a WebClient bean along with the authentication details if any like below,

```
@Bean
public WebClient webClient() {
    return WebClient.builder()
            .filter(ExchangeFilterFunctions.
                    basicAuthentication("admin@eazyschool.com", "admin"))
            .build();
}
```

Consuming REST Services using WebClient

eazy bytes

Step 2 : Using WebClient methods like post(), we can consume a Rest Service like mentioned below,

```
@Autowired
WebClient webClient;

@PostMapping("/saveMessage")
public Mono<Response> saveMessage(@RequestBody Contact contact){
    String uri = "http://localhost:8080/api/contact/saveMsg";
    return webClient.post().uri(uri)
            .header("invocationFrom","WebClient")
            .body(Mono.just(contact),Contact.class)
            .retrieve()
            .bodyToMono(Response.class);
}
```

Spring Data REST

eazy bytes

- *Apart from doing a magic of automatically creating repository implementations based on interfaces you define in your code, Spring Data has another feature that can help you define REST APIs for repositories created by Spring Data.*
- *For the same, we have Spring Data REST which is another member of the Spring Data family.*
- *To start using Spring Data REST in our project, we just need to add the following dependency to our pom.xml,*

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

We can check all the REST APIs exposed by Spring Data Rest by opening the URL http://localhost:8080/profile in the browser.

We just need to add the Spring Data Rest starter project and believe it or not, that's the only change is expected from Developer. Post that the application gets auto-configuration that enables automatic creation of a REST API for any repositories that were created by Spring Data

The REST APIs will be created for any kind of implementations like Spring Data JPA, Spring Data Mongo etc. The REST endpoints that Spring Data REST generates are at least as good as (and possibly even better than) the ones Developers creates ☺.

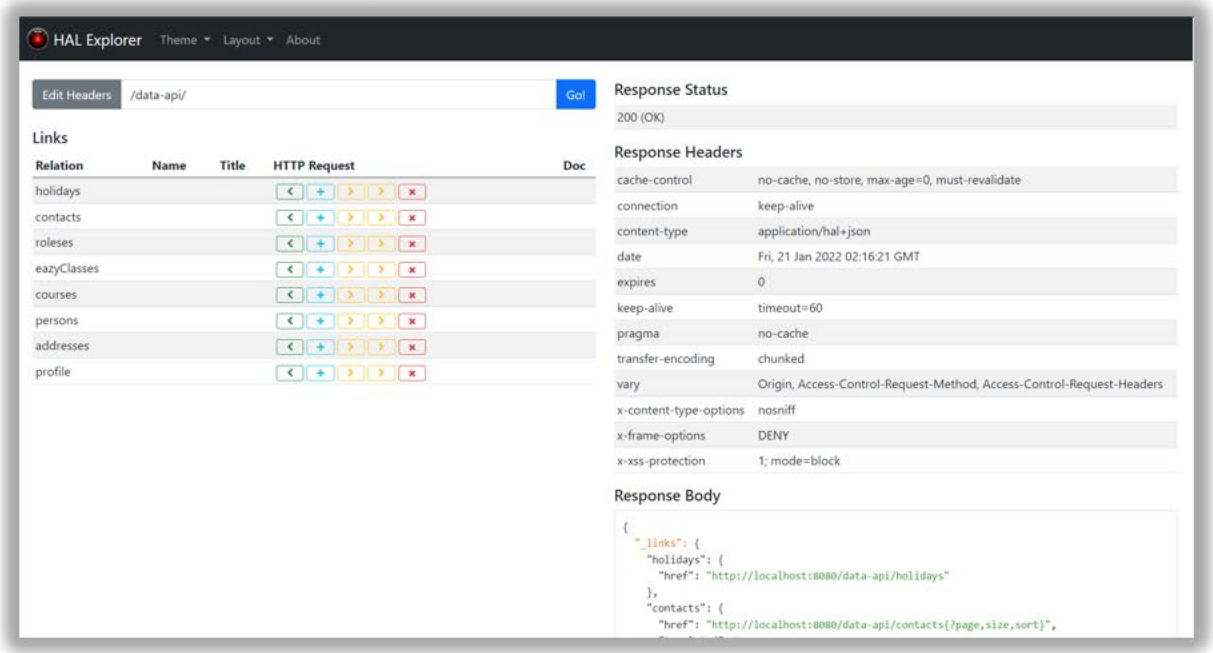
By adding the HAL Explorer along with Spring Data REST, we can look all the APIs exposed by Spring Data REST by opening the http://localhost:8080/ URL in the browser assuming that your App started at 8080 port itself.

HAL Explorer

eazy bytes

- *HAL (Hypertext Application Language) is a simple format that gives a consistent and easy way to hyperlink between resources in your API.*
- *Adopting HAL will make your API explorable, and its documentation easily discoverable from within the API itself. In short, it will make your API easier to work with and therefore more attractive to client developers.*
- *To start using HAL explorer along with Spring Data REST in our project, we just need to add the following dependency to our pom.xml,*

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-rest-hal-explorer</artifactId>
</dependency>
```



We can check the HAL explorer by opening the URL http://localhost:8080/ in the browser. The UI is an Angular based web application that lets you easily explore HAL and HAL-FORMS based HTTP responses.

QUICK TIP

eazy bytes

Do you know we can change the default path exposed by Spring Data REST using the below configurations. This will change the default path as per your configurations.

```
spring.data.rest.basePath=/data-api
```

Using @RepositoryRestResource annotation, we can also control the way the paths of the repositories are being exposed.

```
@RepositoryRestResource(path = "courses")
```

For some reason, if you don't want to expose a Repository, then we can do the below configuration on top of the Repository class,

```
@RepositoryRestResource(exported = false)
```

eazy bytes

Logging

Introduction

✓By default we don't have to worry about logging if we are using Spring Boot. Most of the configurations and logging done by the Spring Boot itself.

✓Usually we have the following types of logging,

- FATAL (Logback doesn't have)
- ERROR
- WARN
- INFO
- DEBUG
- TRACE

Logging Types

✓ In Java we have many logging frameworks like Java Util logging, Log4J2, SLF4J, Logback. By default, if you use the "Starters", Logback is used for logging.

✓ Appropriate Logback routing is also included in Spring Boot to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.

✓ By default, ERROR-level, WARN-level, and INFO-level messages are logged. But we can change them based on our requirements and environments.

Logging inside SpringBoot

eazy bytes

- We can enable debug logging or trace logging by mentioning the below properties inside application.properties file,

application.properties

```
1  debug=true
2  trace=true
```

Alternatively we can provide the flags while starting the application like mentioned below,

$ java -jar myapp.jar --debug

- The way logging works is if we enable trace then all above severities like Debug, Info, Warn & Error are also printed. Suppose if we enable error logging only, then all lower severities like Warn, Info, Debug, Trace will not be logged.

- If needed we can control the logging at the package level by mentioning properties like below inside application.properties,

application.properties

```
1  logging.level.root=INFO
2  logging.level.com.eazybytes.eazyschool.aspects = ERROR
```

Logging inside SpringBoot

eazy bytes

- *It is often useful to be able to group related loggers together so that they can all be configured at the same time. For example, I may have an requirement to change the logging levels for all my project related packages very frequently. To help with this, Spring Boot allows you to define logging groups. Below is the sample configuration,*

application.properties

```
# Intialize log group eazyschool_error
logging.group.eazyschool_error=com.eazybytes.eazyschool.aspects, com.eazybytes.eazyschool.controller
# Set log level to log group eazyschool_error
logging.level.eazyschool_error=ERROR
```

- *If your terminal supports ANSI, color output is used to aid readability of your logs. You can set the below property to enable the same.*

application.properties

```
spring.output.ansi.enabled=ALWAYS
```

Logging inside SpringBoot

eazy bytes

- *Below is the default logging format in Spring Boot. Below is a sample logger message and format details of it,*

2025-01-21 08:02:46.035 INFO 15084 --- [restartedMain] c.e.eazyschool.EazyschoolApplication : Started EazyschoolApplication in 5.189 seconds (JVM running for 2123.116)

- ✓ Date and Time: Millisecond precision and easily sortable.
- ✓ Log Level: ERROR, WARN, INFO, DEBUG, or TRACE.
- ✓ Process ID.
- ✓ A --- separator to distinguish the start of actual log messages.
- ✓ Thread name: Enclosed in square brackets (may be truncated for console output).
- ✓ Logger name: This is usually the source class name (often abbreviated).
  The log message.

- *By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, we can create a file with the name logback.xml inside class path and define all our logging requirements in it.*

QUICK TIP

eazy bytes

Do you know Lombok has various annotations to help developers with logging based on the logging framework being used,

@Slf4j will generate the below code and we can use the log variable directly,

```
@Slf4j
 public class LogExample {

 }
```

will generate:

```
 public class LogExample {
    private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);
 }
```

Similarly using @CommonsLog, @Log4j2 will generate log variable from the respective library class.

eazy bytes

Configurations

Introduction

✓Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources, include Java properties files, YAML files, environment variables, and command-line arguments.

✓By default, Spring Boot look for the configurations or properties inside application.properties/yaml present in the classpath location. But we can have other property files as well and make SpringBoot to read from them.

Config/Properties Preferences

✓ Spring Boot uses a very particular order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones):

- Properties present inside files like application.properties
- OS Environmental variables
- Java System properties (System.getProperties())
- JNDI attributes from java:comp/env.
- ServletContext init parameters.
- ServletConfig init parameters.
- Command line arguments.

Reading properties with @Value

eazy bytes

- *We can read the properties/configurations, defined inside a properties file with the help of @Value annotation like shown below,*

application.properties

```
1  eazyschool.pageSize=10
2  eazyschool.contact.successMsg=Your message is submitted successfully.
```

@Value annotation leverages SpEL expressions to read the configurations present inside the properties file

```
@Controller
public class DashboardController {

    @Value("${eazyschool.pageSize}")
    private int defaultPageSize;

    @Value("${eazyschool.contact.successMsg}")
    private String message;
```

Reading properties using Environment

eazy bytes

- *Along with @Value, we can read the properties/configurations loaded with the help of Environment bean as well which is created by Spring framework. Apart from user defined properties, using Environment we can read any environment specific properties as well.*

application.properties

```
eazyschool.pageSize=10
eazyschool.contact.successMsg=Your message is submitted successfully.
```

```
@Controller
public class DashboardController {

    @Autowired
    private Environment environment;

    private void logProperties() {
        log.info(environment.getProperty("eazyschool.pageSize"));
        log.info(environment.getProperty("eazyschool.contact.successMsg"));
        log.info(environment.getProperty("JAVA_HOME"));
    }
```

Reading properties with @ConfigurationProperties

eazy bytes

- *SpringBoot allow to load all the properties which are logically together into a java bean. For the same we can use @ConfigurationProperties annotation on top of a java bean by providing the prefix value. We need to make sure to use the names inside bean and properties file.*
- *Please follow below steps to read the properties using @ConfigurationProperties ,*

  - *Step 1 : We need to maintain the properties like below which have same prefix like 'eazyschool'*

application.properties

```
eazyschool.pageSize=10
eazyschool.contact.pageSize=5
eazyschool.contact.successMsg=Your message is submitted successfully.
eazyschool.branches[0]=NewYork
eazyschool.branches[1]=Delhi
eazyschool.branches[2]=Paris
eazyschool.branches[3]=Singapore
```

Reading properties with @ConfigurationProperties

eazy bytes

- *Step 2 : Create a bean like below with all the required details,*

```
@Component("eazySchoolProps")
@Data
@PropertySource("classpath:some.properties")
@ConfigurationProperties(prefix = "eazyschool")
@Validated
public class EazySchoolProps {

    @Min(value=5, message="must be between 5 and 25")
    @Max(value=25, message="must be between 5 and 25")
    private int pageSize;
    private Map<String, String> contact;
    private List<String> branches;
}
```

@PropertySource – can be used to mention the property file name if we are using something other than application.properties

@ConfigurationProperties – can be used to mention the prefix value that needs to be considered while loading the properties into a given bean

@Validated – can be used if we want to perform validations on the properties based on the validation mentioned on the field

Reading properties with @ConfigurationProperties

eazy bytes

- *Step 3 : Finally we can inject the bean which we created in the previous step and start reading the properties from it using java style like shown below,*

```
@Service
public class ContactService {

    @Autowired
    EazySchoolProps eazySchoolProps;

    public Page<Contact> findMsgsWithOpenStatus(int pageNum,String sortField, String sortDir){
        int pageSize = eazySchoolProps.getPageSize();
        if(null!=eazySchoolProps.getContact() && null!=eazySchoolProps.getContact().get("pageSize")){
            pageSize = Integer.parseInt(eazySchoolProps.getContact().get("pageSize").trim());
        }
```

Reading properties with @PropertySource

eazy bytes

- *Sometimes we maintain properties inside files which has name is not equal to application.properties. In those scenarios, if we try to use the @Value, it will not work.*
- *First we need to communicate to SpringBoot about the property files with the below steps,*

We need to create a class with @Configuration and @PropertySource annotation like mentioned below. Here we need to mention the property file name. Post these changes, we can refer the properties present inside the config.properties using @Value annotation or Environment bean. ignoreResourceNotFound = true will not throw an exception in case the file is missing.

```
@Configuration
@PropertySource(value = "classpath:config.properties", ignoreResourceNotFound = true)
public class AppConfig {

}
```

We can configure multiple property files as well using @PropertySources annotation like mentioned here,

```
@Configuration
@PropertySources({
        @PropertySource(value = "classpath:config.properties",ignoreResourceNotFound = true),
        @PropertySource("classpath:server.properties")
})
public class AppConfig {

}
```

eazy bytes

Profiles

Introduction

✓Spring provides a great tool for grouping configuration properties into so-called profiles(dev, uat, prod) allowing us to activate a bunch of configurations based on the active profile.

✓Profiles are perfect for setting up our application for different environments, but they're also being used in another use cases like Bean creation based on a profile etc.

✓So basically a profile can influence the application properties loaded and beans which are loaded into the Spring context.

Configuring Profiles

✓ The default profile is always active. Spring Boot loads all properties in application.properties into the default profile.

✓ We can create another profiles by creating property files like below,

application_prod.properties ---------> for prod profile

application_uat.properties ---------> for uat profile

✓ We can activate a specific profile using spring.profiles.active property like below,

```
spring.profiles.active=prod
```

QUICK TIP

eazy bytes

Do you know there are many ways to activate a profile. Below are the most commonly used.

- ✓ By mentioning `spring.profiles.active=prod` inside the properties files.
- ✓ Using environment variables like below,

```
export SPRING_PROFILES_ACTIVE=prod
java -jar myApp-0.0.1-SNAPSHOT.jar
```

- ✓ Using Java System property,

```
java "-Dspring-boot.run.profiles=prod" -jar myApp-0.0.1-SNAPSHOT.jar
mvn spring-boot:run "-Dspring-boot.run.profiles=prod"
```

- ✓ Activating a profile programmatically invoking the method `setAdditionalProfiles("prod")` inside SpringApplication class.
- ✓ Using @ActiveProfiles while doing testing,

```
@SpringBootTest
@ActiveProfiles({"uat"})
```

Conditional Bean creation using Profiles

eazy bytes

- *With the help of Profiles we can create the Bean conditionally. Below is an example where we can create different beans based on the active profile,*

```
@Component
@Profile("!prod")
public class EazySchoolNonProdUsernamePwdAuthenticationProvider
        implements AuthenticationProvider
{
```

```
@Component
@Profile("prod")
public class EazySchoolUsernamePwdAuthenticationProvider
        implements AuthenticationProvider
{
```

Spring Boot Actuator

eazy bytes

- *Actuator offers production-ready features such as monitoring and metrics to Spring Boot applications. Actuator's features are provided by way of several endpoints, which are made available over HTTP.*
- *To enable Actuator, we can mention the below dependency inside pom.xml,*

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

Once the Actuator dependency added to the project, we can navigate to http://localhost:8080/actuator to check the list of APIs exposed by it.

In a machine, an actuator is a component that's responsible for controlling and moving a mechanism. In a Spring Boot application, the Spring Boot Actuator plays that same role, enabling us to see inside of a running application and, to some degree, control how the application behaves.

Using endpoints exposed by Actuator, we can know the following type of info,

- ✓ Health
- ✓ Configuration properties
- ✓ Logging levels
- ✓ Memory consumption
- ✓ Beans information
- ✓ Metrics etc.

By default, Actuator doesn't expose many of the endpoints since they have sensitive information. We can expose them using the below property,

```
management.endpoints.web.exposure.include=*
```

Spring Boot Actuator

eazy bytes

API paths provided by Actuator

HTTP method	Path	Description
GET	/auditevents	Produces a report of any audit events that have been fired.
GET	/conditions	Produces a report of autoconfiguration conditions that either passed or failed, leading to the beans created in the application context.
GET	/configprops	Describes all configuration properties along with the current values.
GET	/beans	Describes all the beans in the Spring application context.
GET, POST, DELETE	/env	Produces a report of all property sources and their properties available to the Spring application.
GET	/env/{toMatch}	Describes the value of a single environment property.
GET	/heapdump	Downloads the heap dump.
GET	/health	Returns the aggregate health of the application and (possibly) the health of external dependent applications.
GET	/httptrace	Produces a trace of the most recent 100 requests.
GET	/info	Returns any developer-defined information about the application.
GET	/loggers	Produces a list of packages in the application along with their configured and effective logging levels.

Spring Boot Actuator

eazy bytes

API paths provided by Actuator

HTTP method	Path	Description
GET, POST	/loggers/{name}	Returns the configured and effective logging level of a given logger. The effective logging level can be set with a POST request.
GET	/mappings	Produces a report of all HTTP mappings and their corresponding handler methods.
GET	/scheduledtasks	Lists all scheduled tasks.
GET	/threaddump	Returns a report of all application threads.
GET	/metrics	Returns a list of all metrics categories.
GET	/metrics/{name}	Returns a multidimensional set of values for a given metric.

We can use Spring Boot Admin which is an administrative frontend web application that makes Actuator endpoints more consumable & readable by humans.

eazy bytes

Deploying Spring Boot Applications

Spring Boot's flexible packaging options provide a great deal of choice when it comes to deploying your application.

You can deploy Spring Boot applications to a variety of cloud platforms, to virtual/real machines

Since **AWS** is a famous cloud provider used by majority of the Organizations, let's explore on how to deploy a SpringBoot application using AWS

The SpringBoot Apps can be deployed to

- ✓ Public Clouds like AWS, Azure, GCP
- ✓ Kubernetes
- ✓ Heroku
- ✓ OpenShift etc.

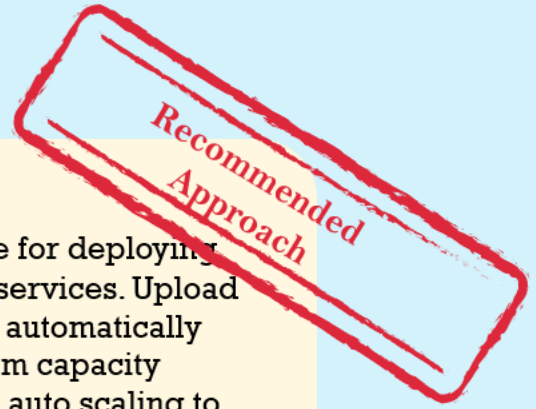
The common approaches to deploy a Spring Boot inside AWS is using either **EC2** or **Elastic Beanstalk**

eazy bytes

AWS EC2 vs AWS Elastic Beanstalk

Amazon Elastic Compute Cloud (EC2) is a virtual cloud infrastructure service offered by AWS that provides users on-demand computing resources through which users can create powerful servers in the cloud. Additionally, EC2 enables users to get a virtual machine up and running in just a few clicks.

With EC2, we can deploy our Spring Boot application into AWS Cloud. But it is a traditional approach and has many drawbacks. When used EC2, the Developer or Organization, **need to take care of installing required software, libraries like Java, Tomcat and handle Auto scaling, load balancing etc. manually.**



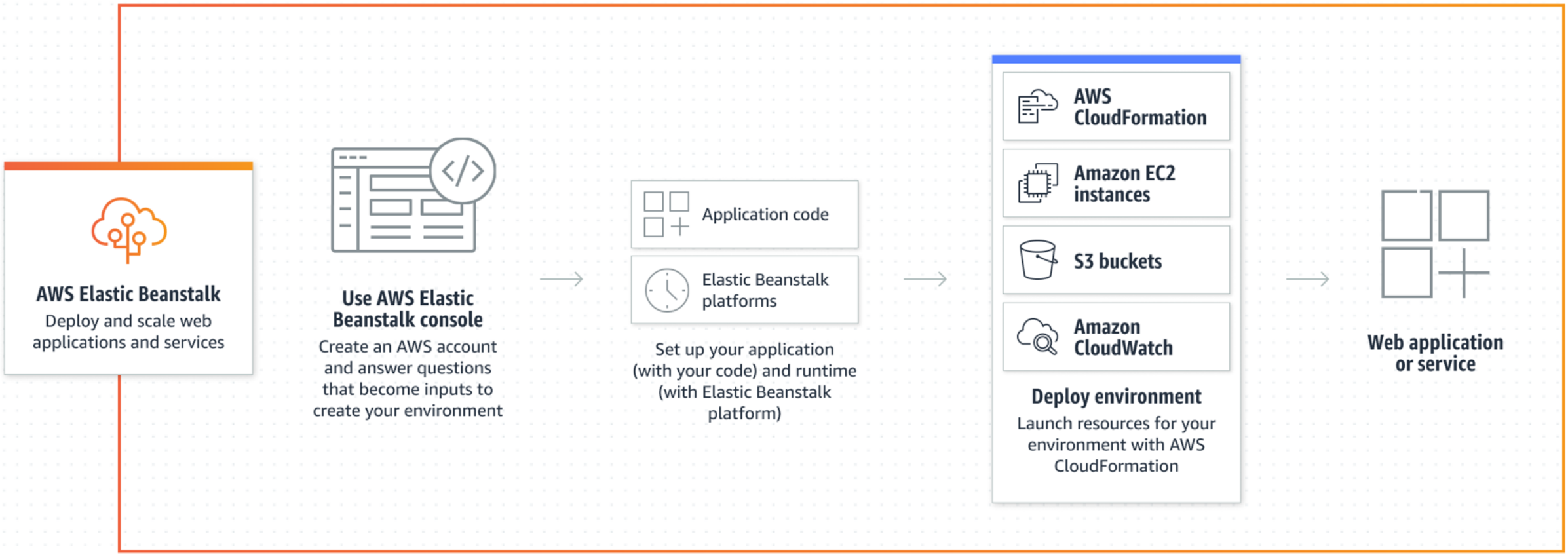
Elastic Beanstalk is a service for deploying and scaling web applications and services. Upload your code and Elastic Beanstalk automatically handles the deployment—from capacity provisioning, load balancing, and auto scaling to application health monitoring.

With Beanstalk, Developers can **focus on writing code instead of provisioning and managing infrastructure**.

Elastic Beanstalk supports applications developed in **Go, Java, .NET, Node.js, PHP, Python, and Ruby**. When you deploy your application, Elastic Beanstalk builds the selected supported platform version and provisions one or more AWS resources, such as Amazon EC2 instances, to run your application.

AWS Elastic Beanstalk - How it works

eazy bytes



SOURCE: https://aws.amazon.com/elasticbeanstalk/

THANK YOU

See you next time!

eazy bytes