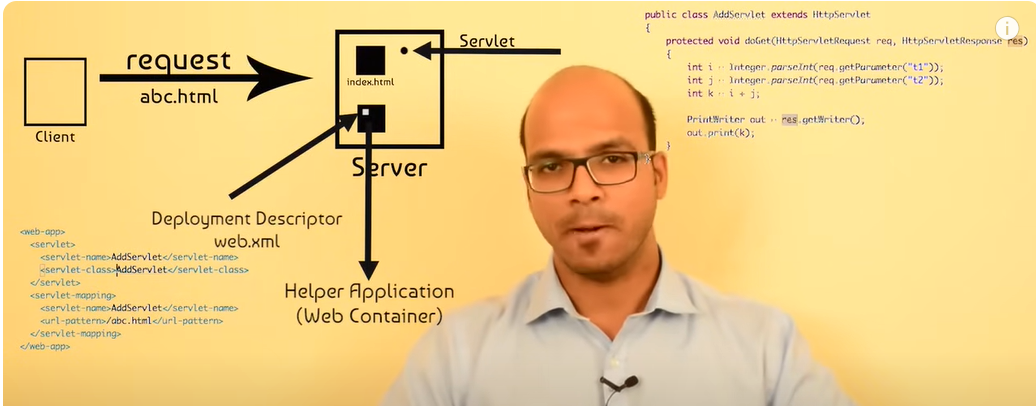
Video Reference: <https://www.youtube.com/watch?v=Jl9OKQ92SJU&list=PLsyeobzWxl7q6oUFts2erdot6jxF_lisP&index=9>

# Servlet

Reference Video: https://www.youtube.com/watch?v=7TOmdDJc14s



A client sends a request so server. The server responds with a static or dynamic page. There is an index.html which sends to the client and the loop closes. When there is a request for a new html, let’s say abc.html the response has to be built. So the server goes to a helper application called **Web Container.** In the Web Container there would be Servlets. Example for Web Container would be Tomcat, Jboss, Glassfish, WebSphere. etc. These are all app servers, but they have features of web servers.

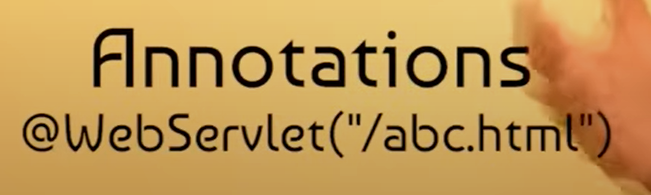
Servlet is basically a java file which can take the request from the client and process the request and provide response in the form of html page.

Deployment descriptor mentions which servlet should be called for which request and mapping can be done using xml files or annotations.

Inside the container there is a file called as Deployment Descriptor which mentions for which request which Servlet should be called. This file is called **web.xml**. As we can see in the above screen shot, the Web.XML has 2 tags. Name tag will have the Servlet class name. URL pattern will have the URL for that Servlet.

The Servlet is a normal class which extends HttpServlet. HttpServlet will provide all the features to take in the request, process information and send a response back. That response will be in HTML format and goes from the server to client.

Web.XML has a replacement called as Annotations.



# JSP (Java Server Pages)

Basically, clients understand HTML, and for a Servlet to output HTML we need to write all the HTML tags ourselves which is not feasible. Therefore JSP comes into picture where we write *java code inside HTML.*

# Maven

This is a build tool to help download dependencies, compile code and build packages.

# Spring Core

By default a DemoSpringApplication.java file is created where we can start the code. In our example we create a new class called Alien. In that @Component is used to annotate that an object of this class will have to be created by spring itself, and not by the coder manually. That’s the advantage of spring. In core java we need to create and close a java instance on our own.

**package** com.mandu.first;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** Alien {

**public** **void** code() {

System.***out***.println("I am coding");

System.***out***.println("My name is Mandanna");

}

}

How to call this Alien class from main? Check below:

SpringApplication.Run is a Static helper that can be used to run a [SpringApplication](eclipse-javadoc:%E2%98%82=demo-spring/C:%5C/Users%5C/prith%5C/.m2%5C/repository%5C/org%5C/springframework%5C/boot%5C/spring-boot%5C/3.4.1%5C/spring-boot-3.4.1.jar=/maven.pomderived=/true=/=/maven.groupId=/org.springframework.boot=/=/maven.artifactId=/spring-boot=/=/maven.version=/3.4.1=/=/maven.scope=/compile=/=/maven.pomderived=/true=/%3Corg.springframework.boot(SpringApplication.class%E2%98%83SpringApplication~run~Ljava.lang.Class%5C%3C*%3E;~%5C%E2%98%83Ljava.lang.String;%E2%98%82SpringApplication) from the specified source using default settings. It **Returns** the running [ApplicationContext](eclipse-javadoc:%E2%98%82=demo-spring/C:%5C/Users%5C/prith%5C/.m2%5C/repository%5C/org%5C/springframework%5C/boot%5C/spring-boot%5C/3.4.1%5C/spring-boot-3.4.1.jar=/maven.pomderived=/true=/=/maven.groupId=/org.springframework.boot=/=/maven.artifactId=/spring-boot=/=/maven.version=/3.4.1=/=/maven.scope=/compile=/=/maven.pomderived=/true=/%3Corg.springframework.boot(SpringApplication.class%E2%98%83SpringApplication~run~Ljava.lang.Class%5C%3C*%3E;~%5C%E2%98%83Ljava.lang.String;%E2%98%82ApplicationContext). getBean() method creates an instance and runs an instance of the class.

@SpringBootApplication

public class DemoSpringApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(DemoSpringApplication.class, args);

Alien obj = context.getBean(Alien.class);

obj.code();

}}

# Spring Boot Autowire

In the below code, we are not creating an object of Laptop. The @Autowired annotation suggests to spring framework that it has to create the object and the below code runs smoothly.

@Component

**public** **class** Alien {

@Autowired

Laptop lap;

**public** **void** code() {

lap.compile();

}

}

## Read about BeanFactory, Singleton V/S Prototype

By default Spring Container will give 1 object. So even if we instantiate 2 objects, variables set in the first object will reflect. When we want multiple objects we have something called as **Scope.** This is defined in the Spring.XML. By default the scope is Singleton. If we need multiple objects we need to define the scope as *Prototype.* So when the container asks for bean it will return as many instances of the object as the number of times we instantiate the object.

Also in Prototype if we do not instantiate the object is does not get created at all. In case of Singleton, based on Spring.xml it creates a default instance.

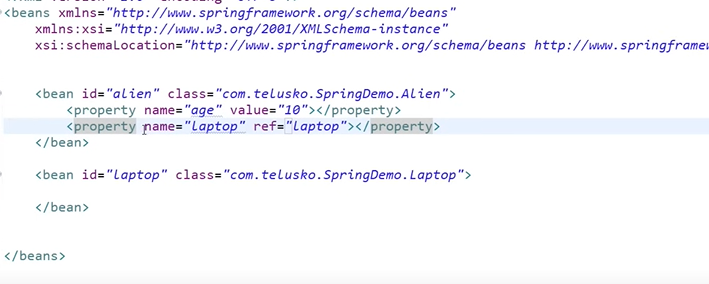
## Setter injection

We can create getter and setter methods for a class, but if we want to set a default value to any variable we can also set it as a property in Spring.XML like below. This is because as a singleton object when spring creates an instance, it can set the variable at that time itself. It will call the setter method for that property and set the value.

<bean id="alien" class="com.telusko.SpringDemo.Alien" autowire="byType">

<property name="age" value="10"></property>

If we have to reference a class, in the below example we have defined Laptop as a separate class, and it is called from the Alien class. So we add another property called “Laptop” which will reference to the separate bean created for Laptop class. And this is referred from the Alien bean by adding a new property. Now, since this is a reference variable and not a type variable, we need to replace “value” with “ref” and mention the class name. In that case the container will create an instance of Laptop at the time of instantiating Alien as well.



# Controller, Service and Repository Layers

In Spring Boot, the Controller, Service, and Repository are key components that form the foundation of a typical web application architecture. They help in achieving separation of concerns and organizing your code in a maintainable way.

**Controller:**

* **Purpose:** Handles incoming HTTP requests, processes them, and returns responses.
* **Responsibility:**
  + Receives requests from clients (e.g., browsers).
  + Invokes the appropriate service methods to perform business logic.
  + Prepares data for the view (if applicable).
  + Returns the appropriate response to the client (e.g., JSON, HTML).
* **Annotation:** @Controller or @RestController (for RESTful APIs)

**Service:**

* **Purpose:** Contains the core business logic of the application.
* **Responsibility:**
  + Implements the business rules and operations.
  + Interacts with repositories to access and manipulate data.
  + Performs any necessary calculations or transformations.
  + Handles transactions (if applicable).
* **Annotation:** @Service

**Repository:**

1. **Purpose:** Provides an abstraction layer for data access.
2. **Responsibility:**
   * Interacts with the database or any other data source.
   * Performs CRUD (Create, Read, Update, Delete) operations on data.
   * Translates database-specific exceptions into more meaningful exceptions.
3. **Annotation:** @Repository

**How they work together:**

* A client sends an HTTP request to the application.
* The appropriate controller receives the request.
* The controller invokes the corresponding service method.
* The service method performs the required business logic, interacting with repositories if necessary.
* The service method returns the result to the controller.
* The controller prepares the response and sends it back to the client.

**Benefits of this pattern:**

* **Separation of concerns:** Each component has a clear responsibility, making the code easier to understand, test, and maintain.
* **Testability:** It's easy to write unit tests for each component in isolation.
* **Scalability:** You can easily add new features and modify existing ones without impacting other parts of the application.
* **Maintainability:** The code is well-organized and follows a clear structure.

# Spring boot JDBC

I had created a spring boot JDBC project using one of the websites and then imported that project to IntelliJ. Below is how the initial .java file looks like:

@SpringBootApplication  
public class SpringJdbcDemoApplication {  
  
 public static void main(String[] args) {  
 //we created an object of ApplicationContext and used it to get the bean of Alien class  
 ApplicationContext context = SpringApplication.*run*(SpringJdbcDemoApplication.class, args);  
 Alien alien1 = context.getBean(Alien.class);  
   
 //We are setting the values of the object alien1  
 alien1.setId(1);  
 alien1.setName("Mandanna");  
 alien1.setTech("Java");  
   
 //We are getting the bean of AlienRepo class and saving the object alien1  
 AlienRepo repo = context.getBean(AlienRepo.class);  
 repo.save(alien1);  
 //We are printing the list of all the objects in the table  
 System.*out*.println(repo.findAll());  
 }  
}

Below is how the AlienRepo class looks like. This is where all the action is.

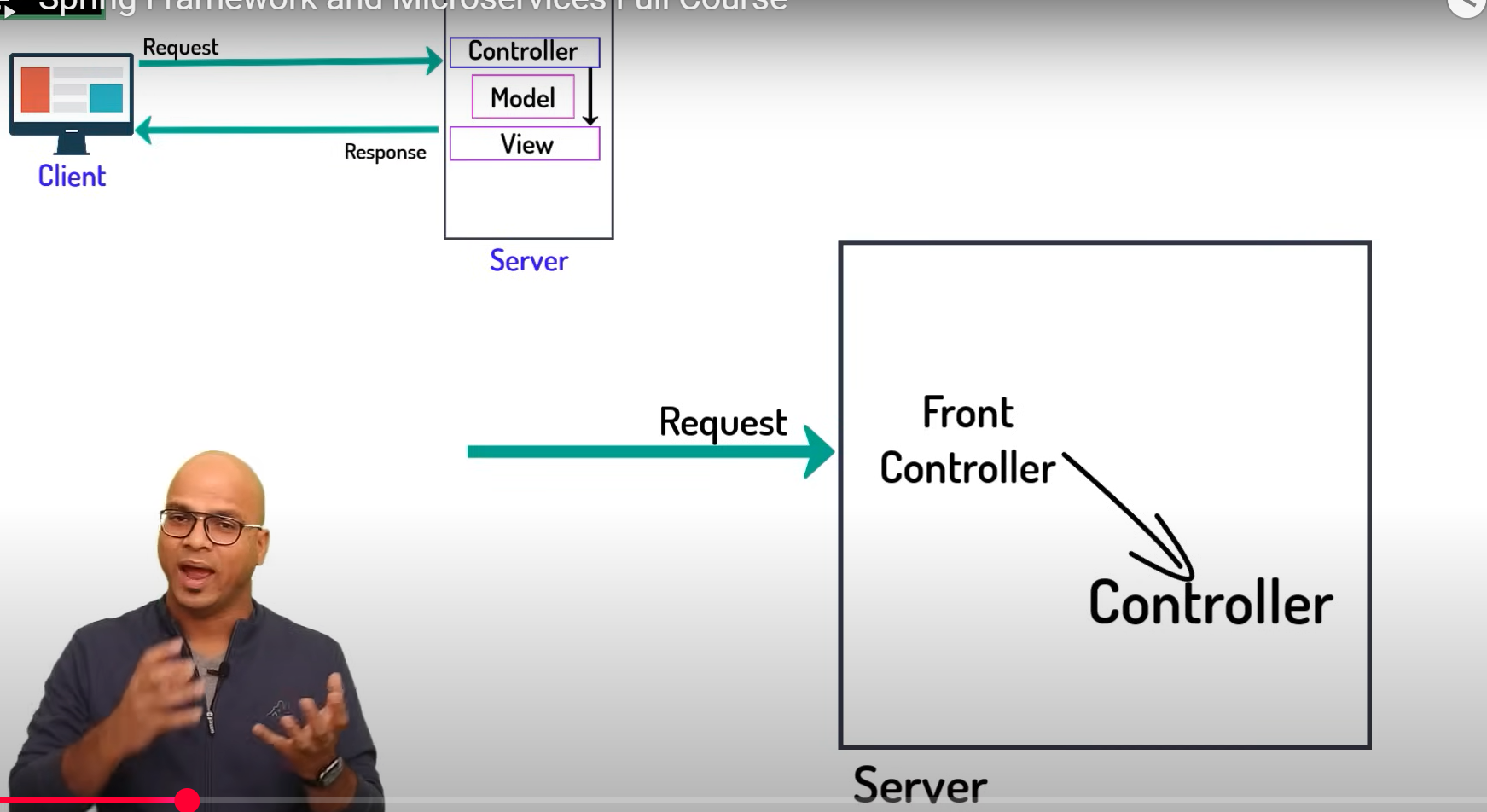
//This is a repository class. It is used to interact with the database  
//We are using the annotation @Repository to tell SpringBoot that this is a repository class  
  
@Repository  
public class AlienRepo {  
  
// JdbcTemplate handles the repetitive tasks of JDBC programming, such as managing database connections, executing SQL queries,  
// and handling result sets. This allows you to focus on writing the actual SQL logic without worrying  
// about the underlying plumbing.  
 @Autowired  
 private JdbcTemplate template;  
  
  
 //Instead of creating an object of JdbcTemplate we use the annotation Autowired so that framework does it  
 @Autowired  
 public JdbcTemplate getTemplate() {  
 return template;  
 }  
  
 public void setTemplate(JdbcTemplate template) {  
 this.template = template;  
 }  
  
 //We need to create the Alien table in the database before running this code  
 //That is done in a .sql file in the resources folder with a create table command  
  
 //here this is a prepare statement. The ?s would be replaced by actual values later  
 String sql = "insert into alien (id, name, tech) values (?,?,?)";  
 public void save(Alien alien){  
 //We are using the update method of JdbcTemplate to insert the values into the table  
 template.update(sql, alien.getId(), alien.getName(), alien.getTech());  
  
 }

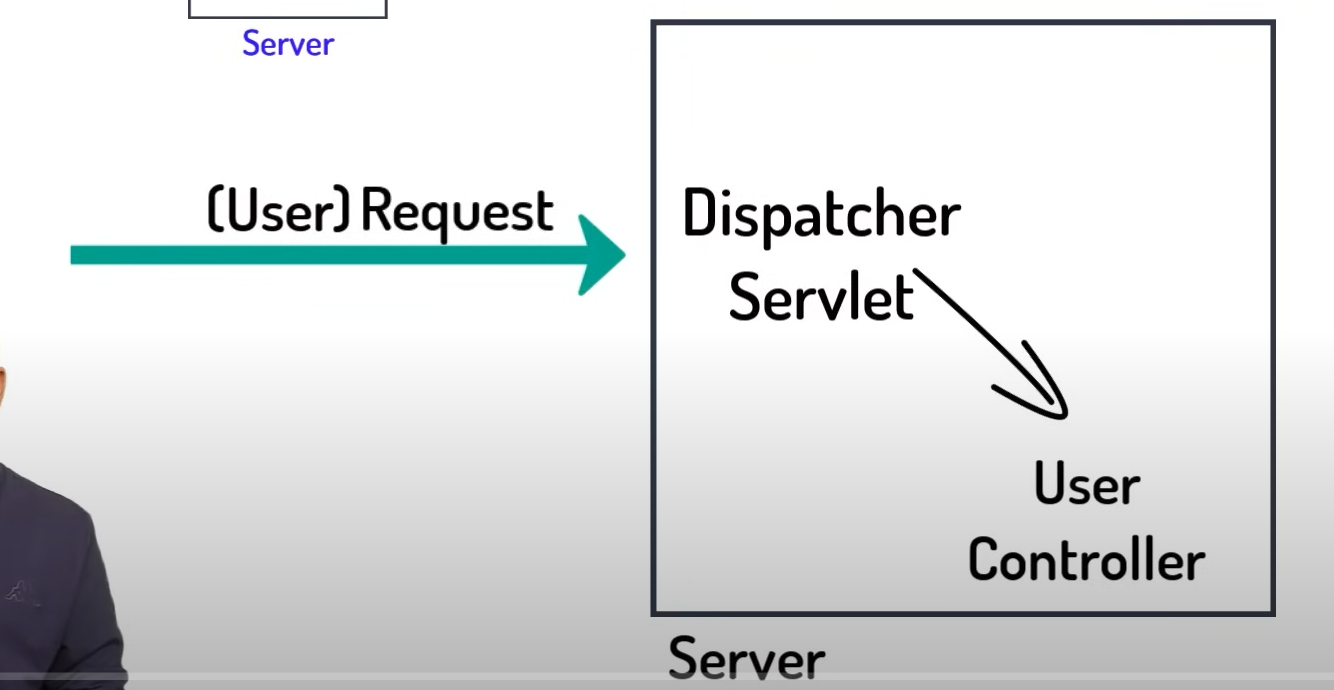
//This method is used to get all the objects in the table  
 public List<Alien> findAll(){  
  
 String str = "select \* from alien";  
 //We are using the query method of JdbcTemplate to get all the objects in the table  
 //We are using the RowMapper interface to map the rows of the table to the object Alien  
   
 RowMapper<Alien> rowMapper = new RowMapper<Alien>() {  
 @Override  
 //This is the mapRow method of the RowMapper interface  
 //For each row in the table, a new object of Alien is created and the values are set  
 //finally the object is returned  
 public Alien mapRow(java.sql.ResultSet rs, int rowNum) throws java.sql.SQLException {  
 Alien alien = new Alien();  
 alien.setId(rs.getInt(1));  
 alien.setName(rs.getString(2));  
 alien.setTech(rs.getString(3));  
 return alien;  
 }  
 };  
   
 //We are using the query method of JdbcTemplate to get a list of all the objects in the table  
 List<Alien> alienList = template.query(str, rowMapper);  
 return alienList;  
  
 }  
  
  
  
}

# Spring MVC

When there is a request from any clients, Controller accepts the request, view gives back the response, and Model basically holds the data. This is to structure the code in an organized way. This will work as a normal Servlet, but there is a separation of concerns.

In Servlets, one servlet could take only 1 request, but Controllers can take multiple requests. There is another layer called Front Controller which will first accept the request and then route to the right Controller. The Front Controller would be directly built by Spring Framework. E.g. If there is a User Controller, there is something called as a Dispatch Servlet which would route the request to a User Controller.





## Creating a Spring Boot project in IntelliJ community version

Go to <https://start.spring.io/>

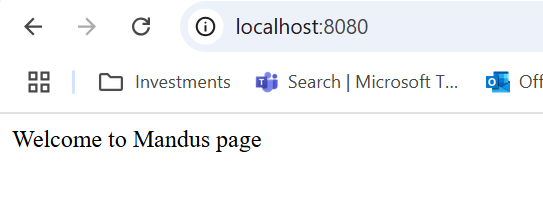
Select the Web dependency and that should reflect in pom.xml as the below:

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

We need to create a Controller which will render a view in the form of JSP. So we created a folder under main called “webapp”.

package com.mandu.demoMvcBoot;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
//The @Controller annotation is used to define a controller and to mark it as a Spring MVC Controller.  
//The @Controller annotation indicates that a particular class serves the role of a controller.  
@Controller  
public class HomePageController {  
 //The RequestMapping annotaton is used to map web requests onto specific handler classes and/or handler methods.  
 //We need to specify the path, and in the below example, we have specified the path as “/”, which is the home page.  
 @RequestMapping("/")  
 public void home() {  
  
 //return "index.jsp";  
 System.*out*.println("This is home page");  
 }  
  
}

Output of this on browser will be as below:



If we want let’s say add 2 numbers and print the results on a new page the index.jsp page would look like the below:

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

pageEncoding="ISO-8859-1"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Insert title here</title>

</head>

<body>

Welcome to Mandu's page

<form action="add">

Enter 1st Number: <input type="text" name="num1">

Enter 2nd Number: <input type="text" name="num2">

<input type="submit">

</body>

</html>

HomePageController.java would look like the below:

We need to create a method to handle the “add” action as below. Here it is accepting num1 and num2 and returning the result in another jsp called return jsp.

@RequestMapping("/add")

**public** String add(@RequestParam("num1") **int** i, @RequestParam("num2") **int** j, HttpSession session) {

**int** sum = i + j;

session.setAttribute("sum", sum);

**return** "result";

//return "result.jsp";

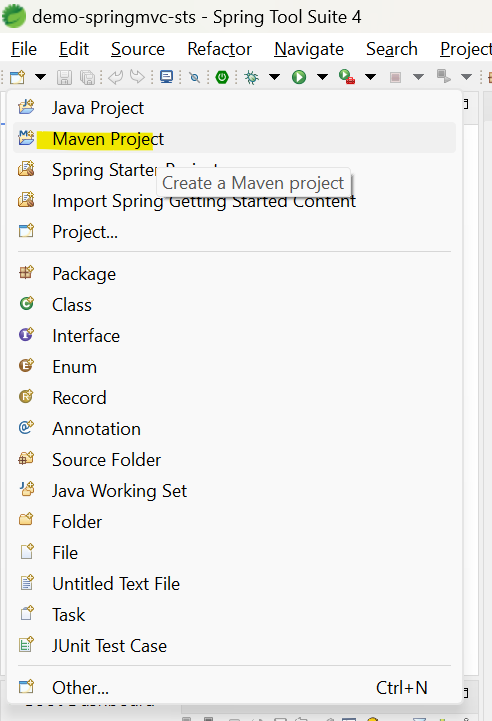
}

**There are further topics on how to use ModelView and other objects to make the code much simpler. Refer to the project “demo-springmvc-sts” in “D:\Programming\Spring” path to understand. There are a lot of comments in the code that will help understand the concepts clearly.**

# Creating a project using MVC (not sprint boot)

Here we will have to do some configurations ourselves. We will do the same project as above but without Spring Boot.

We need to create a new Maven project (not a spring project)



There are a lot of configurations that need to be done on web.xml and serlet.xml for this to work. Can watch videos on how to do that.

There are also Get and Post Request Methods that need to be specified when we create a method. Get method is used to get data from the server. Post is used to send data from a client to server. Refer to methods

@RequestMapping(value = "/addAlien", method = RequestMethod.*POST*)  
public String addAlien (@ModelAttribute Alien a, Model m) {

AND

//@GetMapping("/getAlien") is a shortcut for @RequestMapping(value = "/getAlien", method = RequestMethod.GET)  
//GetMapping is used to map the request to the /getAlien path that we created in the index.jsp file.  
@GetMapping("/getAlien")  
public String getAliens(Model m) {

These are present inside HomePageController.java file inside demo-mvc-sts project.

# Spring ORM

Spring ORMs are used to map a class to a table. Generally each class is associated with a table and we can map every column in a database to a value in the class and work like that.

Hybernate connects with database. There is application (Spring). In between we will have Spring ORM which will connect these 2. Benefits would be that Hybernate instance will be created by Spring ORM. Configuration will be done by Spring. Spring transactions will be handled by Spring Transactions.

When we want to work with databases we can use normal JDBC or we an use Hybernate. For big applications we can go with Hybernate as it handles transactions in a good way.

3.21

2:54 – tomcat server

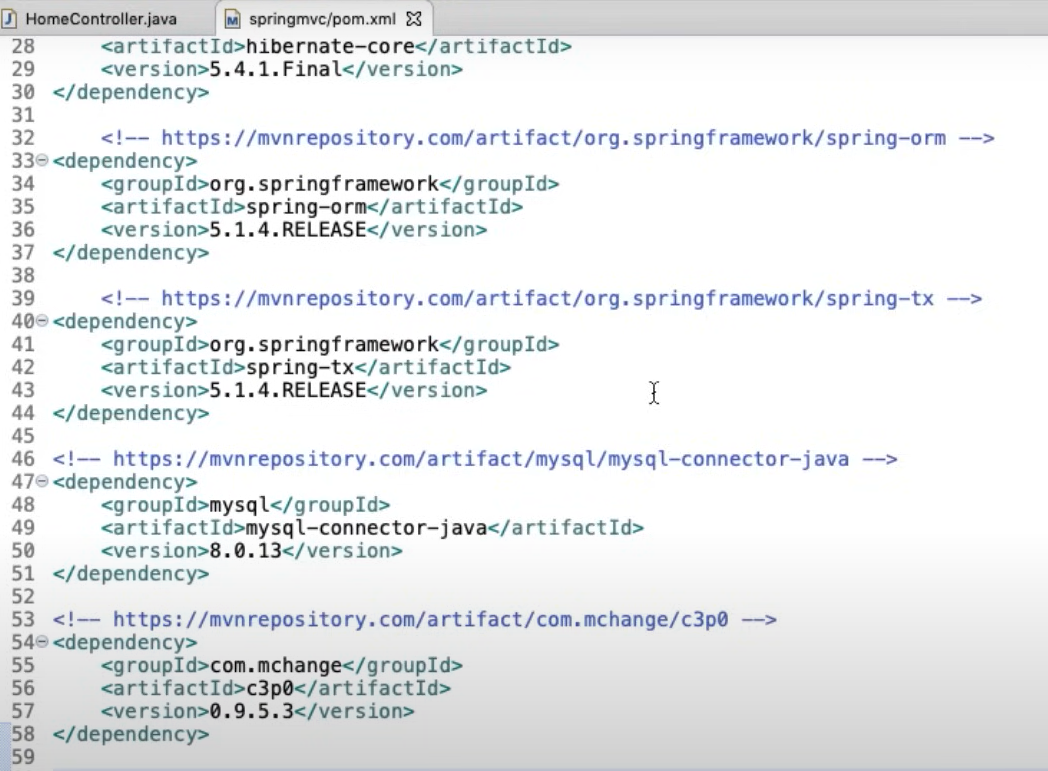
**Dependancies:**

For working on Hibernate (which is an ORM framework), we need to add related dependencies. Go to the website,

* 1. Maven repositories, and add **Hibernate core** and copy it to POM.XML.
  2. Once that is done, we will also need **Spring ORM,** which would be the link between Hibernate and the application.
  3. When we work on Hibernate we need to work with Transactions (begin transaction, commit transaction. Etc). So we add **Spring Transaction** also to POM.XML.
  4. We will also need a database. We need to install My SQL and configure. But we will need a MYSQL connector. Depending on the DBMS (Postgress..etc) we ned
  5. We will also have to POOL connections and we need to use some third party JARS for that. We could install C3P0.

Also in the servlet.xml we need to configure database properties.. etc.

DAO (Data Access Object) – For each data entity, we create some classes called DAO objects. These are basically classes that are used for all CRUD operations on each data entity, which could mostly be the data tables. These are commonly created under a separate package.



Also, we would have to configure all these dependancies in Servlet.XML. Those can be searched and copy pasted.

We also need to install MY SQL. Install the community version. We need MySQL server and MySQL Workbench. Workbench acts as a GUI to the server.

When we create the class for the data entity, we need to use the annotation @Entity so that compiler knows how to interact with this class with Hibernate. Also you may notice that @Id is specified when we define the primary key for that data object.

@Entity  
public class Alien {  
 @Id  
 private int aid;  
 private String aname;

Inside the AlienDao.class, the methods would look like this. Some key things to remember:

SessionFactory is an important object for creating and managing sessions. getCurrentSession uses the session that is already present, rather than creating a new session for executing the transaction.

createQuery basiscally does a select \* from the specified class. We do not have to write the query. This is used in the getAliens() method below to get a list of all the rows in the table.

session.save(a); is used in addAlien basically inserts a new row into the table.

To get a particular item we use session.get(Alien.class, aid); where aid is the primary key, and it returns a particular row from the database.

@Transactional will create, commit a transaction.. etc as required by Hibernate

@Component  
public class AlienDao {  
 @Autowired  
 private SessionFactory sessionFactory;  
  
 public AlienDao() {  
 }  
  
 @Transactional  
 public List<Alien> getAliens() {  
 Session session = this.sessionFactory.getCurrentSession();  
 List<Alien> aliens = session.createQuery("from Alien", Alien.class).list();  
 return aliens;  
 }  
  
 @Transactional  
 public void addAlien(Alien a) {  
 Session session = this.sessionFactory.getCurrentSession();  
 session.save(a);  
 }  
  
 @Transactional  
 public Alien getAlien(int aid) {  
 Session session = this.sessionFactory.getCurrentSession();  
 Alien a = (Alien)session.get(Alien.class, aid);  
 return a;  
 }  
}

# Spring Data JPA

JPA is part of spring boot framework.

**package** com.telusko.**springmvcboot**;

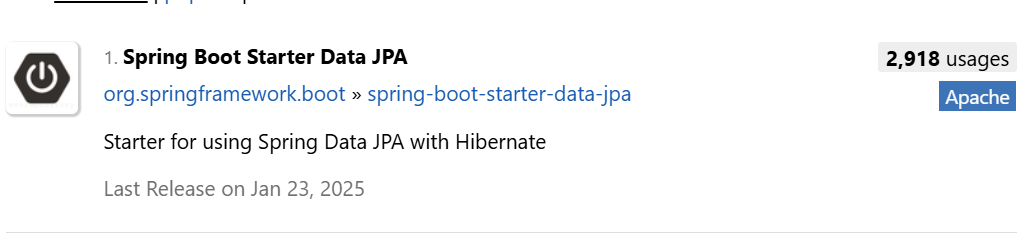
**import** org.springframework.data.jpa.repository.JpaRepository;

**Refer to project “D:\Programming\Spring\CopiedFromNavin\29 demo”**

We need to add the following dependencies into POM.XML, in addition to the standard spring MVC project with hard coded data values:

My SQL connector

Spring Boot Data – Used to connect to database using spring applications



In summary, this works as below. Like in Spring MVC, the Entity class is defined and it is typically mapped to a table. The entity class has to be annotated as @Entity and the primary key needs to be annotated as @Id.

Once this is done, we need to create an interface with naming convention repo<EntityName>. All the CRUD operations for the entity like getList, getSingleRecord, AddEntity, are coded as methods in this repo. This is also put under a different package under src/main/java.

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.telusko.springmvcboot.model.Alien;

//**JpaRepository<Alien,Integer>** is the interface that provides all the CRUD operations

//Since we are importing this, it will provide all the CRUD operations for the Alien class

//The magic is that we do not have to implement these methods as they have already been defined in the interface. We just have to extend this interface

**public** **interface** AlienRepo **extends** JpaRepository<Alien,Integer> {

}

But the limitation of this is that we can operate only with the primary key. If we want to operate with other search params, we need to use **Query DSL.**

# Query DSL

**public** **interface** AlienRepo **extends** JpaRepository<Alien,Integer> {

//findByAname is a query DSL method that is provided by Spring Data JPA

//Query DSL is a domain specific language that is used to query the database. It automatically generates the query for us

//This is specifically created by us to form queries based on the variables we want.

//In findByName we can also do Order by Ascending or Descending by adding the keyword Asc or Desc

//The method name should be in the format findBy<VariableName>

List<Alien> findByAname(String aname);

}

This is now invoked from the controller as normal.

@GetMapping("getAlienByName")

**public** String getAliensByName(@RequestParam String aname, Model m)

{

List<Alien> aliens= repo.findByAname(aname);

m.addAttribute("result",aliens);

**return** "showAliens";

}

# Query Annotation

JPA provides create / read / delete functions based on primary key. If we want to query by another key we can write our own methods. Sometimes we will want to use our own query.

List<Alien> findByAnameOrderByAidDesc(String aname);

//If we need to use complex queries, we can use the @Query annotation like below.

//:name here is the placeholder for the variable that we will pass in the function. In the function, which is QueryByName

//in this case, we have to annotate the same placeholder with @Param as shown below

//Now, this method is available to be called from the repo.

@Query("from Alien where aname= :name")

List<Alien> QueryByAname(@Param("name") String aname);

}

# REST (Representational State)

REST = Server sends out the current state of the data object to client.

Following are the differences between how we have received data from server in case of Spring MVC using JSP requests to a Restful service

1. We are requesting data with nouns and not URLs..(There is no getAlienByName?aname=Lehan). The noun would be something like /Alien
2. Stateless – Server would not remember an earlier session from the client. That means if we want to maintain a login session OR some data we need to inform the server by sending a token. First request, server gives a token and that token will have to be sent back to the server when there are further requests.
3. REST uses inbuilt methods of the HTTP protocol like GET / POST (create) / PUT (update) / DELETE

//@Controller

//We are using @Controller annotation to tell Spring Boot that this class is a controller.

//However, since this is a REST API, we will be using @RestController annotation instead of @Controller.

//By doing this we don't have to specify @ResponseBody for each method.

@RestController

**public** **class** AlienController {

@Autowired

AlienRepo repo;

//@GetMapping is to annotate the method as a request handler. This will be part of the URL request

//ResponseBody is used to return the data in the form of JSON. Otherwise based on our config in application.properties, it will return the data in the form of HTML

@GetMapping("aliens")

**public** List<Alien> getAliens()

{

List<Alien> aliens= repo.findAll();

**return** aliens;

//If we notice, we are returning an object of Alien. This object will be converted to JSON and sent as a response.

//This is automatically performed by Spring Boot with the help of Jackson library. We can find the Jackson library in the Maven dependencies.

//Jackson can also be used to convert JSON to Java object.

}

@GetMapping("alien/{aid}")

**public** Alien getAlien(@PathVariable("aid") **int** aid) {

Alien alien = repo.getOne(aid);

**return** alien;

}

//If we notice for the same /alien URL, we have two methods with different HTTP methods.

//This is possible because we have different HTTP methods for each method.

//Spring Boot will automatically call the method based on the HTTP method.

@PostMapping("alien")

**public** Alien addAlien(Alien alien) {

repo.save(alien);

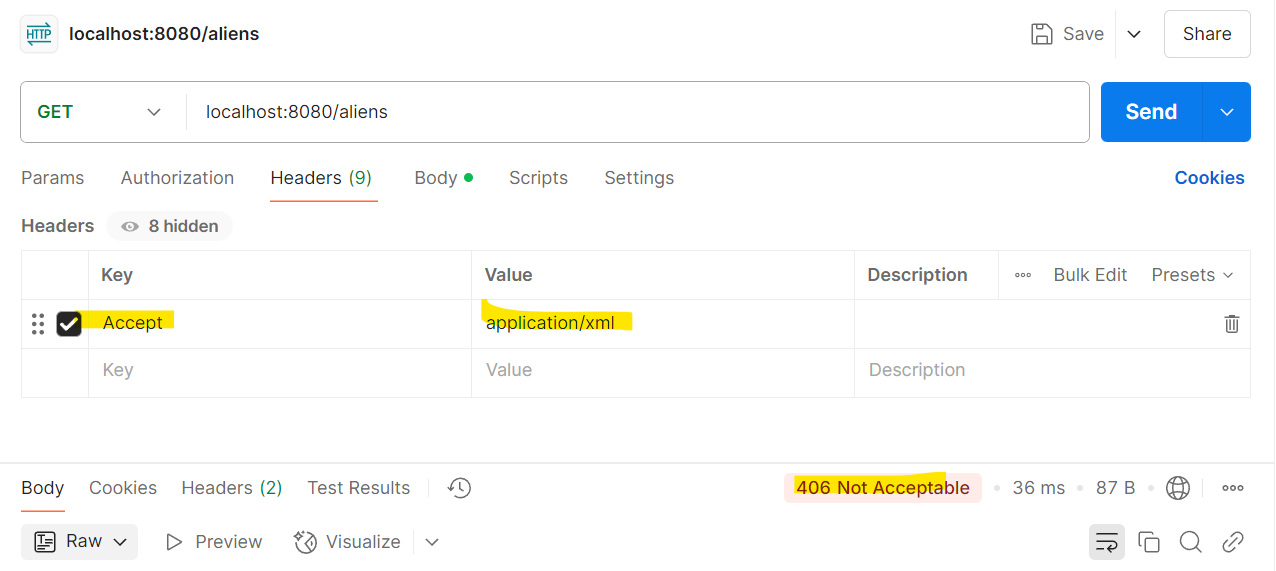
**return** alien;

}

}

## Content Negotiation

The output of above code will always be JSON. What if we want to support XML? Can simulate this in Postman as below. But we get a 406 error which says not acceptable.



To support converting of JSON to XML based on request, we need to import the below JAR file. Make sure the version of Jackson is the same as what is existing:

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-xml -->

<dependency>

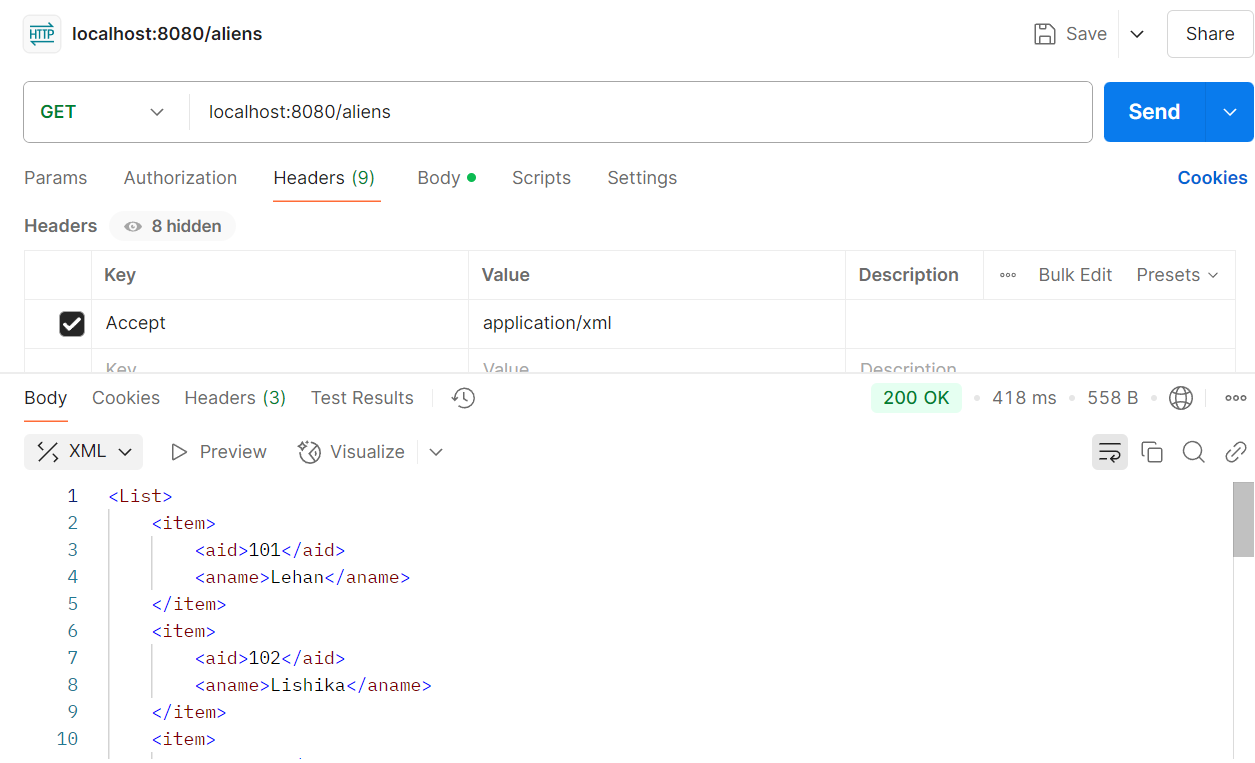
<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

<version>2.9.9</version>

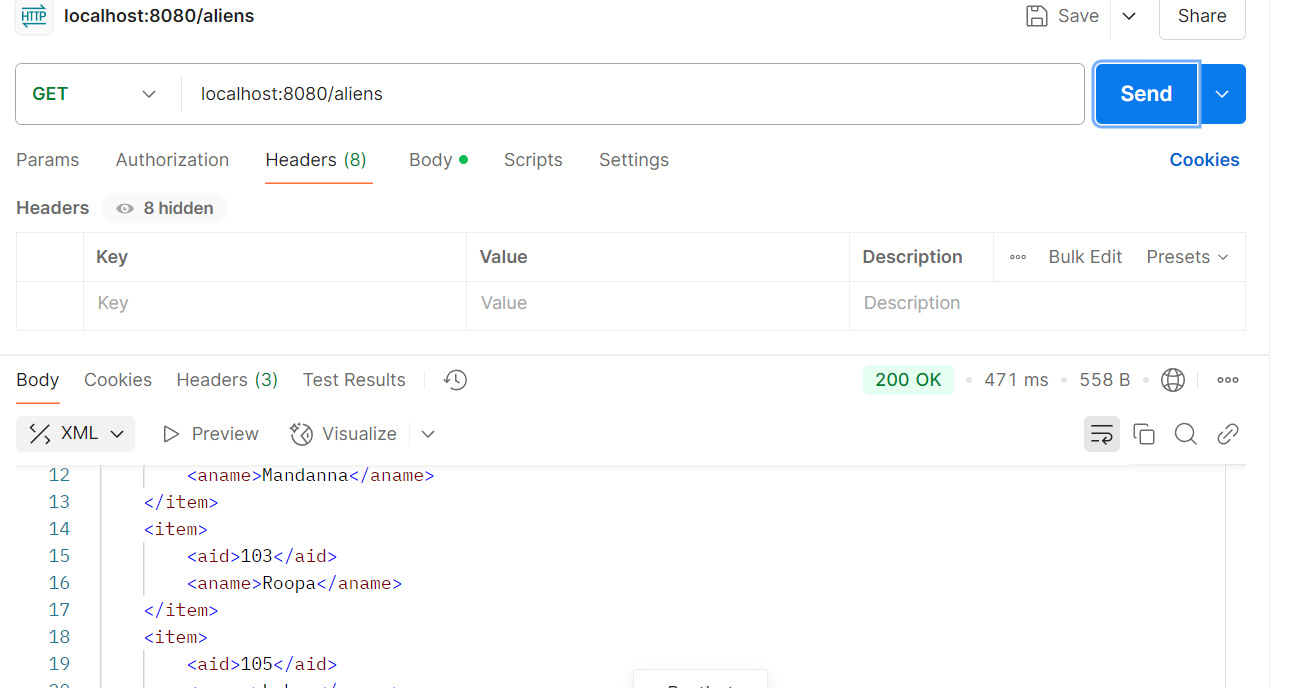
</dependency>

Once this is imported we do not get that error and the response is received in XML format.



## Produces Attribute

If I want to restrict from the server side saying I only want to return XML, we can do that.



//@GetMapping is to annotate the method as a request handler. This will be part of the URL request

//ResponseBody is used to return the data in the form of JSON. Otherwise based on our config in application.properties, it will return the data in the form of HTML

//produces is used to specify the type of data that the method will return. In this case, it will return XML

//If JSON is requested, then the method will not be called. This is because we have specified produces as XML.

@GetMapping(path="aliens",produces= {"application/xml"})

**public** List<Alien> getAliens()

{

List<Alien> aliens= repo.findAll();

**return** aliens;

}

## Request Mapping and Consumes Attribute

What if the data coming from client is in JSON / XML formats? The server side needs to be able to accept the format.

@PostMapping(path="alien",consumes = "application/json")

//consumes is the reverse of produces and is used to specify the type of data that the method will accept.

//We could restrict the server to accept application/xml OR application/json

**public** Alien addAlien(@RequestBody Alien alien) {

//We are using @RequestBody annotation to tell Spring Boot that the data is coming in the form of JSON.

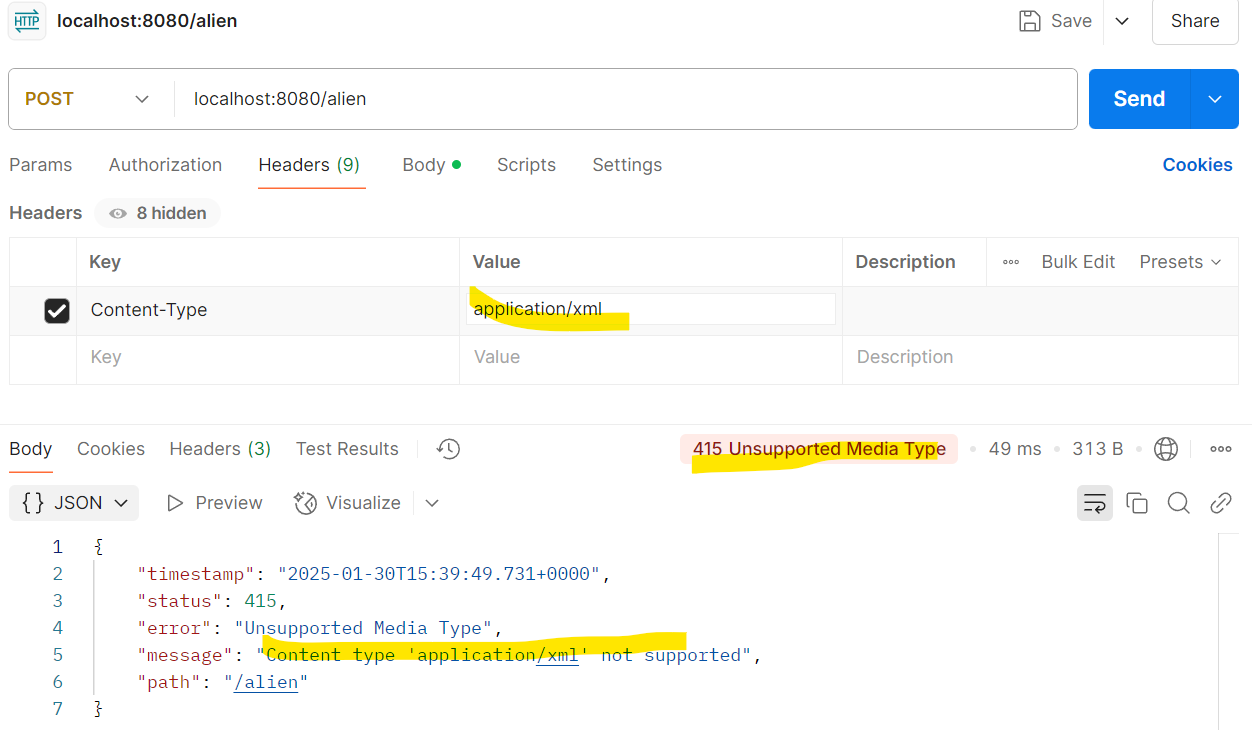
//This is the reverse of @ResponseBody where we are returning the data in the form of JSON.

repo.save(alien);

**return** alien;

}

If we now send an XML request there would be an error as below as we are restricting that the method consumes only JSON.



# Logging

Logging methods are called **Cross Cutting Concerns** as they are valid concerns and are common to entire business logic, and can be called from the entire code.

## AOP (Aspect Oriented Programming)

<https://docs.spring.io/spring-framework/reference/core/aop.html>

* Aspect: A modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented by using regular classes (the [schema-based approach](https://docs.spring.io/spring-framework/reference/core/aop/schema.html)) or regular classes annotated with the @Aspect annotation (the [@AspectJ style](https://docs.spring.io/spring-framework/reference/core/aop/ataspectj.html)).
* Join point: A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.
* Advice: Action taken by an aspect at a particular join point. Different types of advice include "around", "before", and "after" advice. (Advice types are discussed later.) Many AOP frameworks, including Spring, model an advice as an interceptor and maintain a chain of interceptors around the join point.
* Weaving: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Spring AOP includes the following types of advice:

* Before advice: Advice that runs before a join point but that does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
* After returning advice: Advice to be run after a join point completes normally (for example, if a method returns without throwing an exception).
* After throwing advice: Advice to be run if a method exits by throwing an exception.
* After (finally) advice: Advice to be run regardless of the means by which a join point exits (normal or exceptional return).
* Around advice: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

//@Aspect is used to annotate that this class is an Aspect  
//An aspect is a class that implements cross-cutting concerns. These are the aspects of the application that affect multiple classes.  
//@Component is used to indicate that this class is a Spring component  
@Aspect  
@Component  
public class LoggingAspect {  
  
 //@Before is used to indicate that this method should be executed before the getAllProducts method is called  
 //Syntax for the @Before annotation is @Before("execution(public \* <package>.<class>.<method>())")  
 @Before("execution(public \* com.mandu.productManagement.service.ProductService.getAllProducts())")  
 public void log(){  
 System.*out*.println("method called getAllProducts");  
 }  
}

The beauty of this is that the actual method doesn’t even know that its being logged, and the code there is very neat.

public List<Product> getAllProducts() {  
 return db.findAll();  
}

## Logger

Instead of doing a sys out like in above section LOGGER is generally used.

@Aspect  
@Component  
public class LoggingAspect {  
  
 //In the below line, we are creating a logger object with the name "LoggingAspect"  
 //The LoggerFactory.getLogger method is used to get a logger object  
 //Syntax for the LoggerFactory.getLogger method is LoggerFactory.getLogger(<class name>.class)  
 public static final Logger *LOGGER* = LoggerFactory.*getLogger*(LoggingAspect.class);  
 //@Before is used to indicate that this method should be executed before the getAllProducts method is called  
 //Syntax for the @Before annotation is @Before("execution(public \* <package>.<class>.<method>())")  
 @Before("execution(public \* com.mandu.productManagement.service.ProductService.getAllProducts())")  
 public void log(){  
 //In the below line, we are logging a message using the logger object  
 *LOGGER*.info("method called getAllProducts");  
 }  
}

In application.properties file we can set some properties like the below.

logging.level.root = info //This states that the INFO level logs alone will be shown  
logging.file = app.log // this states that logs are written to a separate file called app.log

## How to use Finally

Used to log after the execution of a method. Exactly same as @Before, just that we need to have a different annotation @After

@After("execution(public \* com.mandu.productManagement.service.ProductService.getAllProducts())")  
public void logAfter(){  
 //In the below line, we are logging a message using the logger object  
 *LOGGER*.info("method getAllProducts executed");  
}

@After by default means After(finally). This Advice runs regardless of the means by which a join point exits (normal or exceptional return). If we want the advise to be printed only if the method is successfully executed we need to use **@AfterReturning**

//@AfterThrow is used to indicate that this method should be executed after an exception is thrown in the getAllProducts method  
 @AfterThrowing("execution(public \* com.mandu.productManagement.service.ProductService.getAllProducts())")  
 public void logIssue(){  
 //In the below line, we are logging a message using the logger object  
 *LOGGER*.info("Issue in getAllProducts method");  
 }  
}

# Spring Security

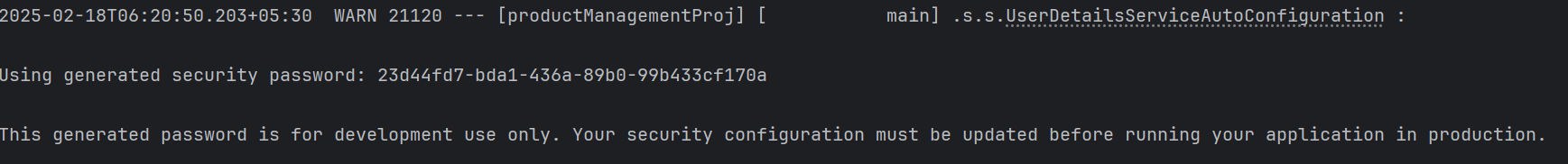
This is to secure the application. Just by adding the below maven dependency, the framework generates a login page (we did not have to create a login.jsp page) and generates a password when the application comes up.

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-security -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>

I had created a home.jsp under webapp folder and when I try to access that it automatically creates a login page as below:

A screenshot of a computer

AI-generated content may be incorrect.

The password is generated when the application is coming up, and can be seen in the console. Default user name is “user”.   


## Authentication using database

Refer to project “D:\Programming\ProductManagementDemo\DemoJavaProjProductMgmnt”

A summary of the code is as below and we can follow the comments to understand the code:

//This is the entry point for spring security. This class is responsible for authenticating the user  
//@EnableWebSecurity annotation is used to enable web security in the application  
  
@Configuration  
@EnableWebSecurity  
public class AppSecurityConfig {  
 @Autowired  
 private UserDetailsService userDetailsService;  
 //authenticationProvider() method is used to authenticate the user. It uses the userDetailsService to get the user details and NoOpPasswordEncoder to encode the password  
 //we have created a MyUserDetailsService class that implements UserDetailsService interface. This class is used to get the user details from the database  
 //We have also created a MyUserDetailsService class that implements UserDetails interface. This class is used to get the user details from the database  
 //Within MyUserDetailsService, we have created a method loadUserByUsername() that takes the username as an argument and returns the user details  
 //We have also created a UserPrincipal class that implements UserDetails interface. This class is used to get the user details from the database  
 //Within UserPrincipal, we have created a method getAuthorities() that returns the authorities of the user  
 //Username and passwords entered by the user is compared with the database values using the authenticationProvider() method  
 //authenticationProvider() method is used to authenticate the user. It uses the userDetailsService to get the user details and NoOpPasswordEncoder to encode the password  
 //We have also created a UserDetailsService class that implements UserDetailsService interface. This class is used to get the user details from the database  
  
 @Bean  
 public AuthenticationProvider authenticationProvider() {  
 DaoAuthenticationProvider provider = new DaoAuthenticationProvider();  
 provider.setUserDetailsService(userDetailsService);  
 provider.setPasswordEncoder(NoOpPasswordEncoder.*getInstance*());  
 return provider;  
 }

## Bcrypt

Bcrypt is a password hashing function designed to be computationally intensive. It's commonly used for securely storing passwords in databases. There are many websites that can generate Bcrypt hashed passwords. No of cycles mean the number of iterations. More cycles the better.

In code, we just need to change the password encoder to BCryptPasswordEncoder. The libraries are already present in spring.

@Bean  
public AuthenticationProvider authenticationProvider() {  
 DaoAuthenticationProvider provider = new DaoAuthenticationProvider();  
 provider.setUserDetailsService(userDetailsService);  
 //provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance());  
  
//here we have used BCryptPasswordEncoder to encode the password.  
 provider.setPasswordEncoder(new BCryptPasswordEncoder());  
 return provider;  
}

## SecurityFilterChain

//This method is used to configure the security filter chain. It is used to configure the security filter chain that is responsible for authenticating the user  
 //http.authorizeHttpRequests is used to authorize the requests. It is used to authorize the requests based on the URL patterns.  
 //In this case, we have authorized all the requests using anyRequest().authenticated()  
 //http.formLogin is used to configure the form login. It is used to configure the form login page, login processing URL, default success URL, and failure URL  
 //http.httpBasic is used to configure the HTTP basic authentication. It is used to configure the HTTP basic authentication  
 //all API calls are also authenticated using httpBasic(Customizer.withDefaults())  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{  
 http  
 .csrf(AbstractHttpConfigurer::disable)  
  
 .authorizeHttpRequests(request -> request.anyRequest().authenticated())  
 .formLogin(Customizer.*withDefaults*())  
 .httpBasic(Customizer.*withDefaults*());  
  
 return http.build();  
 }

# Microservices

Drawbacks of monolith applications:

1. Team Dependencies – The entire team is dependant on each other if there are multiple modules for a release. Every team / module needs to be ready for release.
2. Scalability – E.g. if Amazon has a sale, we would have to scale the entire application if Amazon was monolith. In case of microservices we can scale whichever module needs scaling.
3. Technology – We need to stick to one technology.
4. Availability - Even if 1 module crashes, the entire application crashes.

Microservices would have to be self-contained. Then we can scale them the way we want (horizontally or vertically).

What are the challenges with Microservices:

1. Configuring how the microservices communicate needs to be configured. Although one-time this is complex.
2. We need to design the microservices properly. If not designed well, this will get worse than monolith.
3. Security – Service specific access needs to be enabled

# Cloud Ready V/S Cloud Native

If we have to lift and shift an on premise application to cloud, we need to make some changes to config, property files. Etc so that it can communicate effectively with the cloud services.

Cloud Native – Applications built for cloud and can be easily deployed on cloud. There are certain rules that developers need to follow to ensure the application is cloud native.

There are a set of rules / standards called **12-factor app**  as below:

1. Codebase – Once codebase tracked in revision control, many deploys. We should use something like a GitHub / GitLab etc to have 1 codebase. There could be multiple environments. **1 codebase for 1 application.** Not 1 codebase for multiple applications.
2. Dependencies – Explicitly declare and isolate dependencies. Earlier days we used to have a library folder where all the dependencies were copied and used. Now, when we need to deploy we will have to share all the dependencies, which is not a great idea. There could be version mismatch issues. Etc. So we need to have a central file like a manifest file OR a maven file where all the dependencies are listed. These dependencies are kept separate and not mixed with the codebase. When we need to share the application with someone else, we just need to share the maven file and they can download the dependencies with the exact versions stated in maven file.
3. Config – Store configuration in the environment. No hard coded stuff. We need store the configuration in external files.
4. Backing services – Treat backing services as attached resources. Have a lose coupling between the application and backing services. E.g. for backing services could be a a MySQL database, messaging queues.. etc. We need to be able to easily switch let’s say the database from MySQL to Postgress with some config and minimal code changes.
5. Build, Release and Run – Strictly separate build and run stages. We need to create a package and then move the package to an environment where this needs to be run. Each release there should be a different version. If something goes wrong, we should be able to go to the previous version.
6. Processes – Execute the app as one or more stateless .. In the earlier days we used to have stateful services. All services we would authenticate once and then the service would know the client when there is a request next time. The problem with this is that we would be limiting the resources. If something goes wrong with the service, the client would say I was here last time and now my data is lost.! So we need to make the processes stateless and not store any data. If there is any data the data is coming from attached backing service. The advantage would be that any time we could remove the process. Even scaling would be easy as the request would hit a different process next time.
7. Port Binding – Export services via port binding. Every service would be having a different port number. Because we do not know which server they would be hosted in. Only thing we know is that they offer different services.
8. Concurrency – Scale out via the process model. Instead of going for vertical scaling go for horizontal scaling. Instead of using one instance we should be able to create multiple instances.
9. Disposability – Maximise robustness with fast startup and graceful shutdown. Basically, every time we open something that needs to be closed. The moment we want to dispose a service it should be easy without losing data. Any connections need to be closed properly by storing data in database.
10. Dev / Prod parity – Keep development, staging and production as similar as possible. There is always some gap between development and staging environments for example. If we are building in a different environment and deploying in a different environment there will be some config issues. Dev and Ops team need to come together to develop and push changes as soon as possible to server so that the gaps are less. We can create images like Docker. CICD is an example.
11. Logs – Treat logs as event streams. Create a log file with a proper logging service. Every process needs to generate a log. Once that is done we can aggregate at one place.
12. Admin process – Run admin/management tasks as one-off processes. There should always be an admin services to manage things from outside.

# Building blocks of a microservices architecture

* API Gateway: There could be more than 1 instances of a service. When a client is requesting, API gateways route the request to a particular service
* Service Registry – There would be multiple services. Each service need not remember the IPs and locations of other services. This is where they all need to be registered at Service Registry.
* Load Balancer – Depending on load, the load balancer creates more or reduces the number of instances of a service.

In terms of dependencies, we will need 2 more dependencies as below:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
</dependency>

## Feign

This is also called naming server. This provides a declarative way of describing a service, where we do not have to mention the IP address and Port of the microservice we are calling.

## Eureka Server

Since Microservices call each other, they need to be registered on a Eureka Server. **This is a Service Discovery Server.** Once they are registered, the microservices call the Eureka server through their Eureka Clients (dependency in the above POM.xml) to get details of the microservice registered on Eureka Server.

A diagram of service

AI-generated content may be incorrect.

This needs to be created as a separate web application with below dependencies.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
</dependency>

Web – because this is a web application

Eureka Server because this is the server where all the services are going to be registered and discovered.

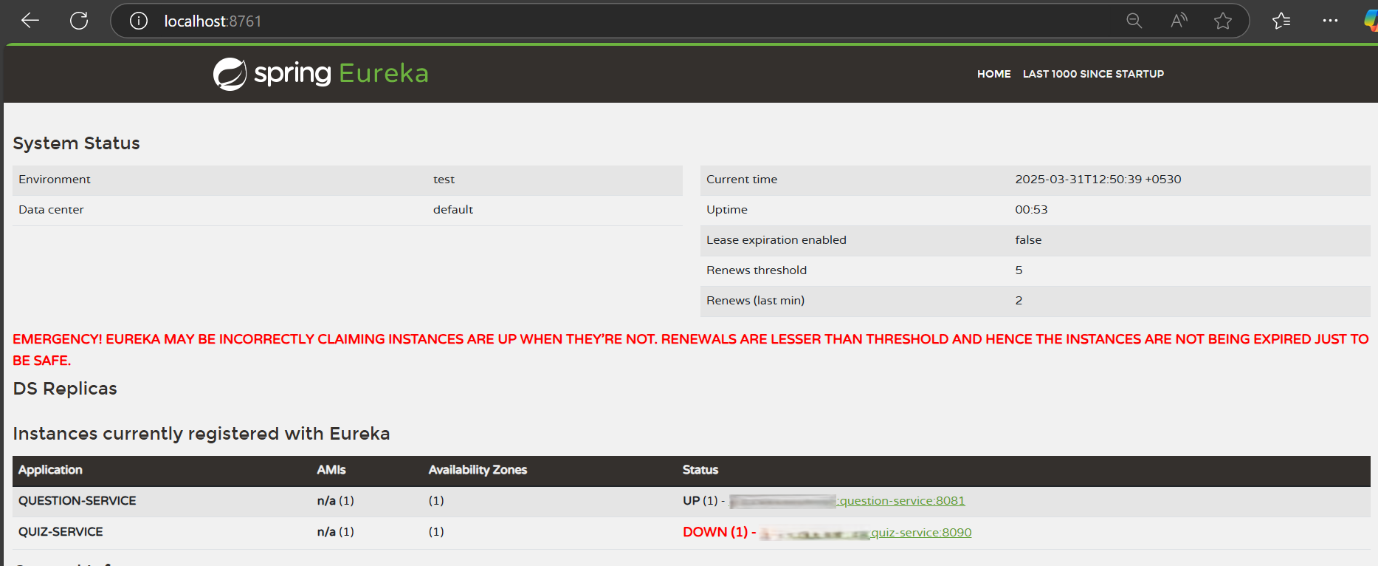
On the Service Registry main, we need to use the annotation @ EnableEurekaServer

@SpringBootApplication  
@EnableEurekaServer  
  
public class ServiceRegistryApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ServiceRegistryApplication.class, args);  
 }  
  
}

Some further configurations are required. Below configurations are done inside application.properties

spring.application.name=service-registry  
# We need to give the Eureka Server a proper name. Like there are application names for  
# all the services  
server.port = 8761  
#this is the default port that Eureka Server runs on  
eureka.instance.hostname=localhost  
# This is the hostname of the Eureka Server. We can also use the IP address of the server  
eureka.client.fetch-registry=false  
# This is the default value. It tells the Eureka Server to fetch the registry from other Eureka Servers  
eureka.client.register-with-eureka=false  
# This is the default value. It tells the Eureka Server to register itself with other Eureka Servers

Once this is done and we run the Service Registry application, it comes up on port 8761 (default which can be changed). Post that, we bring up any applications on the same server they automatically register on service-registry. On a browser, we can access the service-registry application as below and it shows all the registered applications.



## Changes on microservice

We need to create a new interface that can talk to another service that is registered in the Eureka Server. This is enabled by import org.springframework.cloud.**openfeign.FeignClient;**

@FeignClient("QUESTION-SERVICE")  
public interface QuizInterface {  
 //generate quiz  
 @GetMapping("question/generate")  
 public ResponseEntity<List<Integer>> getQuestionsForQuiz(@RequestParam String categoryName,  
 @RequestParam Integer noOfQuestions);  
  
 //get questions based on question ids  
  
 @PostMapping("question/getQuestions")  
 public ResponseEntity<List<QuestionWrapper>> getQuestionsForQuestionIds(@RequestBody List<Integer> questionIds);

In the above code snippet of a sample interface, @FeignClient("question-service") is used to specify the name of the service that we want to communicate with. Inside the interface, we just define the methods that would be called on the service. The @GetMapping path should reflect the exact path defied on the service.

With @FeignClient load balancing is done automatically, and requests are directed to question-service in this example as per load considerations.

Once the interface is setup, that can be instantiated from Service layer and can be used normally like how we use a DAO layer. Refer to below code snipped as an example:

@Autowired  
QuizInterface quizInterface;  
  
  
public ResponseEntity<Quiz> createQuiz(String category, int numQ, String title) {  
  
 List<Integer> questionList = quizInterface.getQuestionsForQuiz(category, numQ).getBody();

## API Gateway

Now, in any scenario with Microservices, we would have multiple services running on multiple servers and ports and that’s confusing. As an end-user they would see all these services together as one application.

The other challenge is with authentication. Once authenticated with one service, we cannot have the users authenticate with another service.

So API Gateways will have to be created as a separate application. We need the below 2 dependencies to create a new API Gateway:

A screenshot of a black and white screen

AI-generated content may be incorrect.

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-gateway</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>

Gateway is for API Gateway, and Eureka Client is because we need to register to Eureka Server.

**For the services to be working, we will need the API GATEWAY as well as SERVICE REGISTRY to be up.**