Build a REST Web Service Using Spring Boot

This Lab provides a step-by-step guide for building and configuring a REST web service in Java using the Spring Boot

Prerequisites:

- Spring Tool Suite 3/4
- Java 1.8
- MySQL 5.5
- MySQL Workbench 5.2
- Postman

The Spring Tool Suite is an open-source, Eclipse-based IDE distribution that provides a superset of the Java EE distribution of Eclipse. It includes features that making working with Spring applications even easier.

Business Scenario:

An application has to be developed for management of interns hired by Yash technologies. The application will provide following services:

1. Based on technical evaluation, intern will be hired and details will be entered in system. Interns technical level will be decided based on college semester marks. Interns will have to go through training based on levels and clear certifications.
2. Intern's details can be retrieved by HR to assign projects to intern.
3. Intern's personal details can be updated.
4. Intern's level can be updated based on semester scores.
5. Intern's details will be removed from system if intern has completed internship.

There will be two applications to suffice need of above system.

InternsUIApp will be developed using Angular/ReactJS and InternsBusinessApp will be business application.

To create a prototype of an InternsBusinessApp, We will use Spring Boot REST

Spring Boot - Introduction

Spring Boot is an open source Java-based framework used to create a micro Service. It is developed by Pivotal Team and is used to build stand-alone and production ready spring applications. This chapter will give you an introduction to Spring Boot and familiarizes you with its basic concepts.

What is Spring Boot?

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can **just run**. You can get started with minimum configurations without the need for an entire Spring configuration setup.

Advantages

Spring Boot offers the following advantages to its developers −

- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

Goals

Spring Boot is designed with the following goals −

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way
- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application

Why Spring Boot?

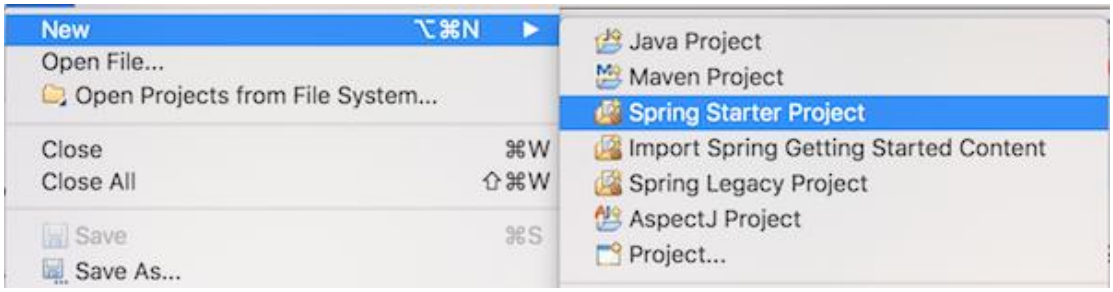
You can choose Spring Boot because of the features and benefits it offers as given here −

- It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
- It provides a powerful batch processing and manages REST endpoints.
- In Spring Boot, everything is auto configured; no manual configurations are needed.
- It offers annotation-based spring application
- Eases dependency management
- It includes Embedded Servlet Container

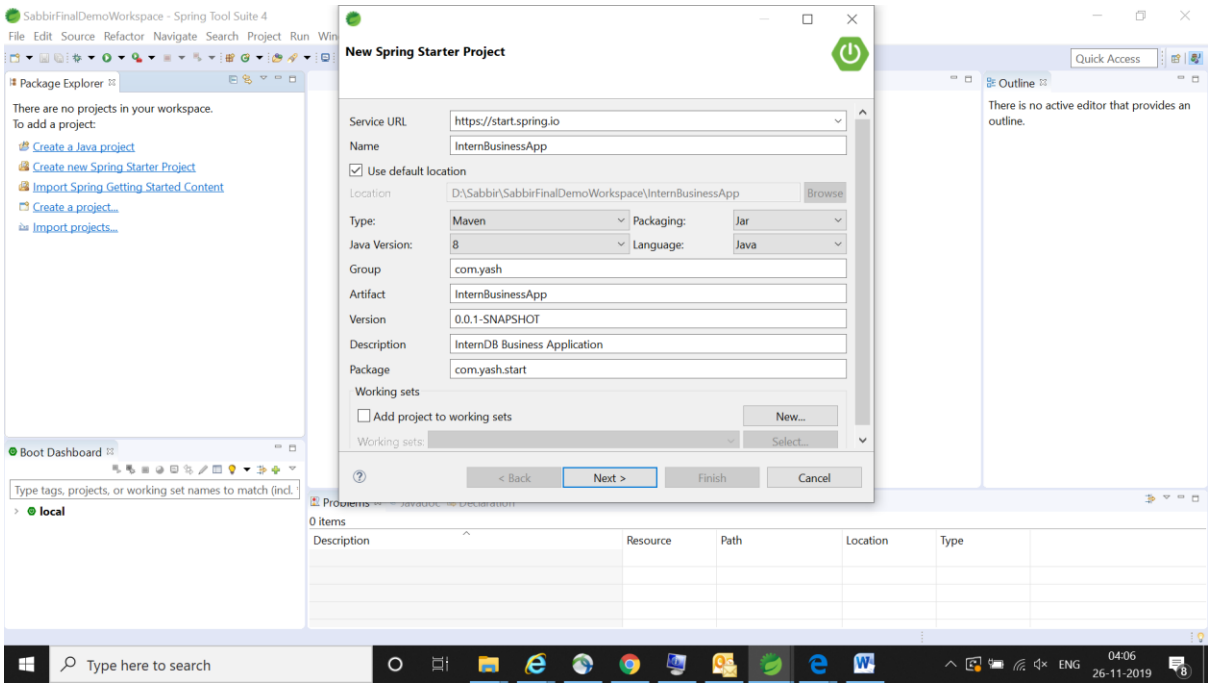
Spring Boot REST Project

Create new Spring Starter Project as shown in below image. Eclipse should have Spring support for this, all the latest Eclipse release has built-in

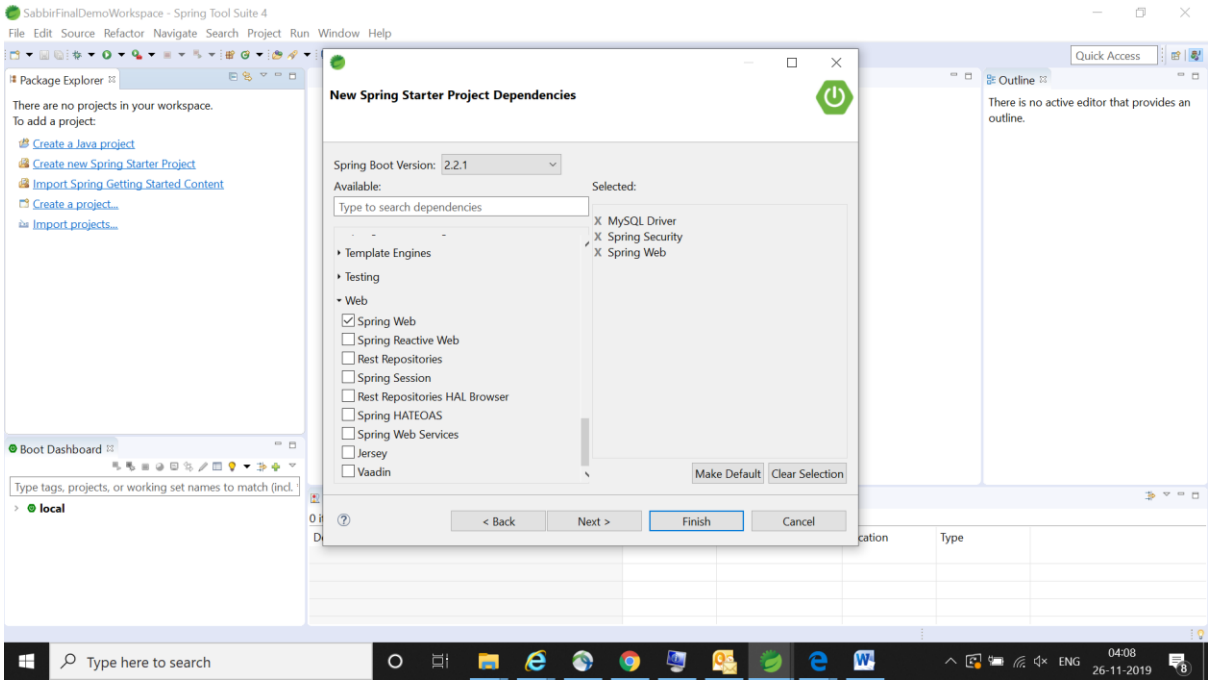
Spring support.



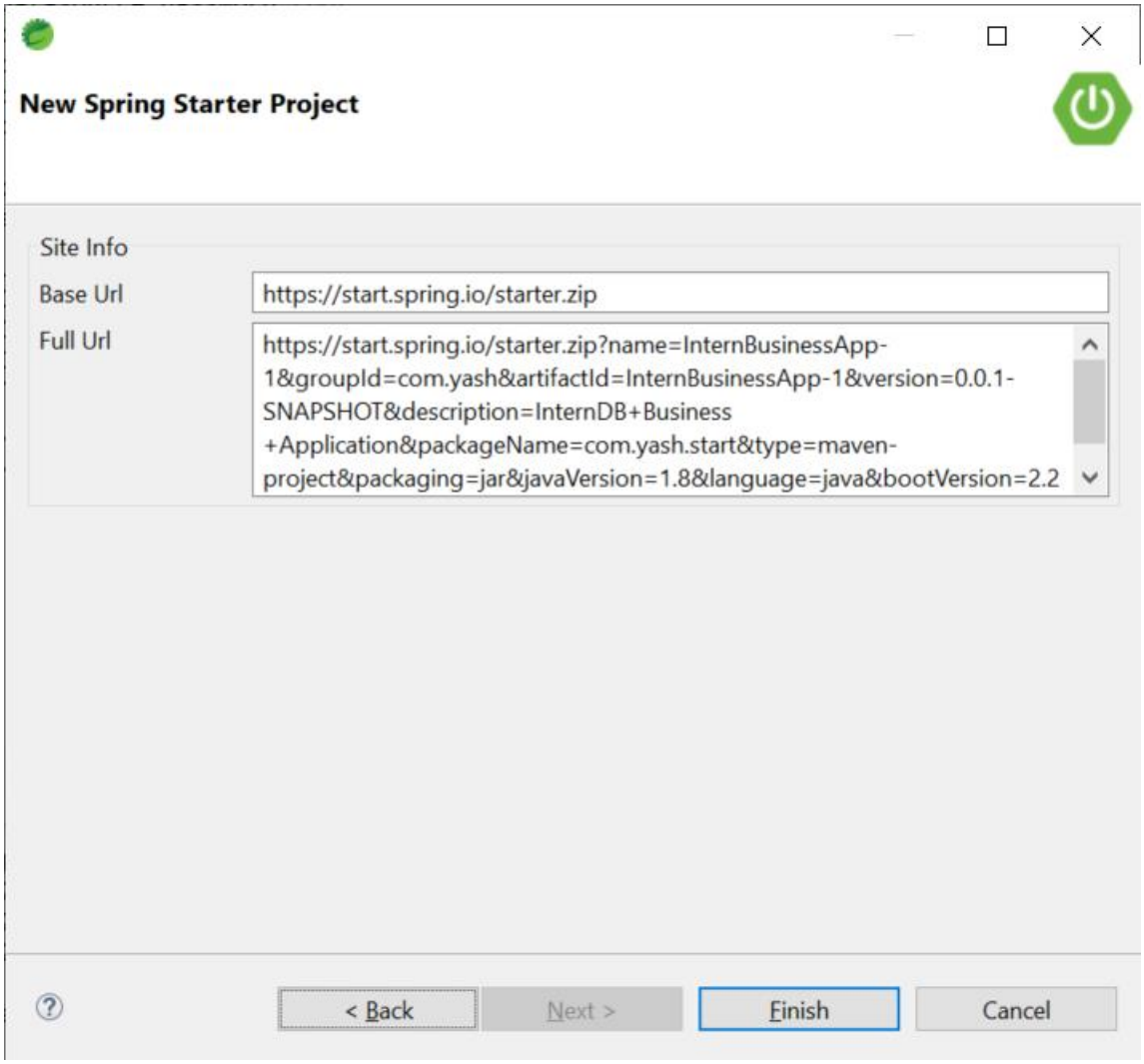
In the next popup screen, provide project basic details. We will use Maven.



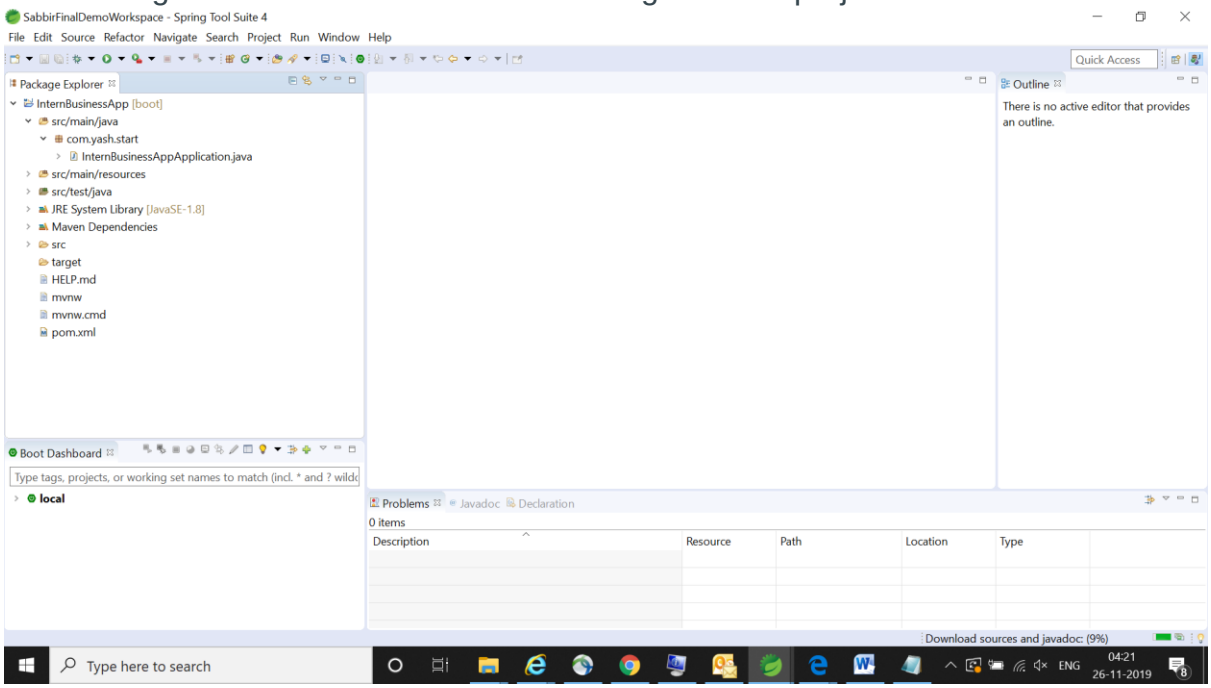
Next screen asks to add dependencies apart from Spring Boot. Add “MySQL Driver”, “Spring Web” dependencies and click next.



We could have click on Finish in the earlier screen, but clicking on next provides us the Spring Initializer URL that we can use to create this project easily through the command line.

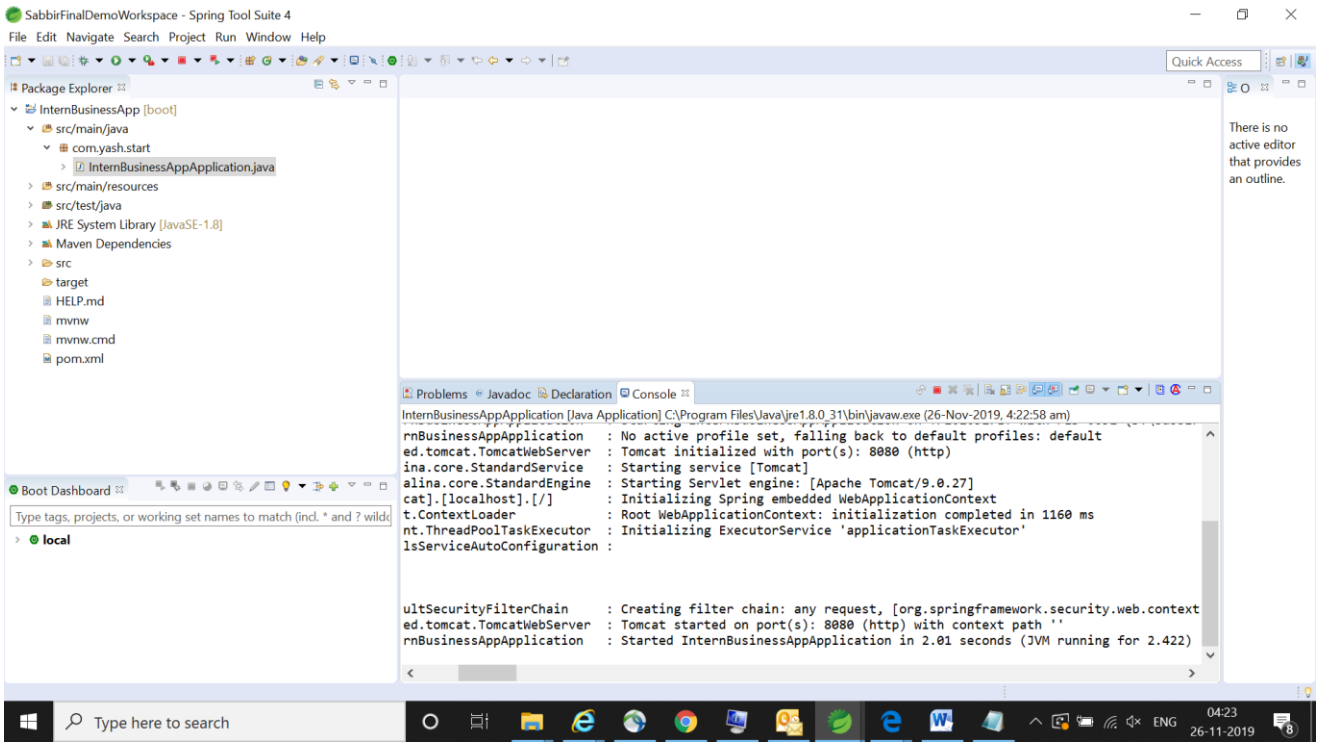


Below image shows the contents of the auto-generated project.



Most important of these is InternBusinessAppApplication class that is configured as Spring Boot application and contains java main method. When we run this class, it automatically runs our project as Spring Boot Web project.
The entry point of the spring boot application is the class contains **@SpringBootApplication** annotation and the main method. We have applied @SpringBootApplication on InternBusinessAppApplication
Spring Boot automatically scans all the components included in the project by using **@ComponentScan** annotation. @EntityScan configures the base packages used by auto-configuration when scanning for entity classes.

```
package com.yash.start;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@EnableAutoConfiguration
@ComponentScan(basePackages="com.yash.*")
@EntityScan(basePackages="com.yash.entity")
public class InternBusinessAppApplication {
	public static void main(String[] args) {
		SpringApplication.run(InternBusinessAppApplication.class, args);
	}
}
```



If you will check the console logs, it should print `Tomcat started on port(s): 8080 (http) with context path ''`.
Maven is a build automation tool used primarily for Java projects.

An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins.

POM is an acronym for Project Object Model. The pom. xml file contains information of project and configuration information for the maven to build the project such as dependencies, build directory, source directory, test source directory, plugin, goals etc.

Spring Boot Starters

Handling dependency management is a difficult task for big projects. Spring Boot resolves this problem by providing a set of dependencies for developers convenience.

For example, if you want to use Spring and JPA for database access, it is sufficient if you include **spring-boot-starter-data-jpa** dependency in your project.

Note that all Spring Boot starters follow the same naming pattern **spring-boot-starter-** *, where * indicates that it is a type of the application.

Mention below dependencies in pom.xml

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

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

         <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-activemq</artifactId>
         </dependency>
      <dependency>
          <groupId>org.apache.activemq</groupId>
          <artifactId>activemq-broker</artifactId>
      </dependency>
       <dependency>
          <groupId>com.fasterxml.jackson.core</groupId>
          <artifactId>jackson-databind</artifactId>
      </dependency>
```

Spring Boot Starter Actuator dependency is used to monitor and manage your application.

Spring Boot Starter web dependency is used to write a Rest Endpoints.
Spring Boot Starter Test dependency is used for writing Test cases
Spring-boot-starter-activemq is starter for JMS messaging using Apache ActiveMQ
activemq-broker is the ActiveMQ Message Broker implementation
jackson-databind is the general data-binding functionality for Jackson
Jackson JSON Parser API provides easy way to convert JSON to POJO Object and supports easy conversion to Map from JSON data. Jackson supports generics too and directly converts them from JSON to object.

Properties file

Properties files are used to keep 'N' number of properties in a single file to run the application in a different environment. In Spring Boot, properties are kept in the application.properties file under the classpath.

The application.properties file is located in the src/main/resources directory.
Put below entries in application.properties file.

```
spring.datasource.url=jdbc:mysql://localhost:3306/InternDB
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.tomcat.max-wait=20000
spring.datasource.tomcat.max-active=50
spring.datasource.tomcat.max-idle=20
spring.datasource.tomcat.min-idle=15

server.port=8089
spring.application.name=InternBusinessService


spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.properties.hibernate.id.new_generator_mappings = false
spring.jpa.properties.hibernate.format_sql = true
spring.jpa.hibernate.ddl-auto= update

logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Note that in the code shown above the Spring Boot application `InternBusinessService` starts on the port 8089.

A **Java Enum** is a special **Java** type used to define collections of constants. More precisely, a **Java enum** type is a special kind of **Java** class. An **enum** can contain constants, methods etc. **Java enums** were added in **Java** 5.

Interns will be assigned levels based on semester marks obtained by Intern.

Create an enum Levels in package com.yash.helper

```
package com.yash.helper;

public enum Levels {
	
	BEGINNER(0),INTERMEDIATE(1),ADVANCED(2);
	
	private int level;
	private Levels(int level){
		this.level=level;
	}
```

```
        public int getLevel(){
                return level;
        }

}
```

An **entity** is a lightweight persistence domain object. Typically an **entity** represents a table in a relational database, and each **entity** instance corresponds to a row in that table.
The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

Requirements for Entity Classes

An entity class must follow these requirements:
The class must be annotated with the javax.persistence.Entity annotation.
The class must have a public or protected, no-argument constructor. The class may have other constructors.
The class must not be declared final. No methods or persistent instance variables must be declared final.
The entity class must implement the Serializable interface.
Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
Persistent instance variables must be declared private, protected, or package-private, and can only be accessed directly by the entity class's methods. Clients must access the entity's state through accessor or business methods.

Persistent Fields and Properties in Entity Classes

The persistent state of an entity can be accessed either through the entity's instance variables or through JavaBeans-style properties. The fields or properties must be of the following Java language types:

- Java primitive types
- java.lang.String
- Other serializable types including:
  - Wrappers of Java primitive types
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - User-defined serializable types
  - byte[]
  - Byte[]
  - char[]

    - Character[]
- Enumerated types
- Other entities and/or collections of entities
- Embeddable classes

Entities may either use persistent fields or persistent properties. If the mapping annotations are applied to the entity's instance variables, the entity uses persistent fields. If the mapping annotations are applied to the entity's getter methods for JavaBeans-style properties, the entity uses persistent properties. You cannot apply mapping annotations to both fields and properties in a single entity.

Persistent Fields

If the entity class uses persistent fields, the Persistence runtime accesses entity class instance variables directly. All fields not annotated javax.persistence.Transient or not marked as Java transient will be persisted to the data store. The object/relational mapping annotations must be applied to the instance variables.

Persistent Properties

If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components. JavaBeans-style properties use getter and setter methods that are typically named after the entity class's instance variable names. For every persistent property property of type Type of the entity, there is a getter method getProperty and setter method setProperty. If the property is a boolean, you may use isProperty instead of getProperty. For example, if a Interns entity uses persistent properties, and has a private instance variable called internFirstName, the class defines a getInternFirstName and setInternFirstName method for retrieving and setting the state of the internFirstName instance variable.

The method signature for single-valued persistent properties are as follows:

```
Type getProperty()
void setProperty(Type type)
```

Collection-valued persistent fields and properties must use the supported Java collection interfaces regardless of whether the entity uses persistent fields or properties. The following collection interfaces may be used:

- java.util.Collection
- java.util.Set
- java.util.List
- java.util.Map

If the entity class uses persistent fields, the type in the above method signatures must be one of these collection types. Generic variants of these collection types may also be used. For example, if the Interns entity has a persistent property that contains a set of phone numbers, it would have the following methods:

```
Set<PhoneNumber> getPhoneNumbers() {}
void setPhoneNumbers(Set<PhoneNumber>) {}
```

The object/relational mapping annotations for must be applied to the getter methods. Mapping annotations cannot be applied to fields or properties annotated @Transient or marked transient.

Primary Keys in Entities

Each entity has a unique object identifier. A Interns entity, for example, might be identified by a internId. The unique identifier, or primary key, enables clients to locate a particular entity instance. Every entity must have a primary key. An entity may have either a simple or a composite primary key.

Simple primary keys use the javax.persistence.Id annotation to denote the primary key property or field.

Composite primary keys must correspond to either a single persistent property or field, or to a set of single persistent properties or fields. Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the javax.persistence.EmbeddedId and javax.persistence.IdClass annotations.

The primary key, or the property or field of a composite primary key, must be one of the following Java language types:

- Java primitive types
- Java primitive wrapper types
- java.lang.String
- java.util.Date (the temporal type should be DATE)
- java.sql.Date

Floating point types should never be used in primary keys. If you use a generated primary key, only integral types will be portable.

Primary Key Classes

A primary key class must meet these requirements:

- The access control modifier of the class must be public.
- The properties of the primary key class must be public or protected if property-based access is used.
- The class must have a public default constructor.
- The class must implement the hashCode() and equals(Object other) methods.
- The class must be serializable.
- A composite primary key must be represented and mapped to multiple fields or properties of the entity class, or must be represented and mapped as an embeddable class.
- If the class is mapped to multiple fields or properties of the entity class, the names and types of the primary key fields or properties in the primary key class must match those of the entity class.

In our application. We will create Interns entity class in package com.yash.entity. To specify level of intern we will utilize enum Levels we created in previous step. We will not persist semester marks in database table, hence field semester1Marks,semester2Marks and semester3Marks are marked as transient.

@DynamicUpdate is a class-level annotation that can be applied to a JPA entity. It ensures that Hibernate uses only the modified columns in the SQL statement that it generates for the update of an entity.
We will discuss @NamedQueries annotation while discussing DAO Layer.

Full code of Interns.java is as below.

```
package com.yash.entity;
import java.io.Serializable
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.persistence.Transient;
import com.yash.helper.Levels;
import org.hibernate.annotations.DynamicUpdate;
@Entity
@DynamicUpdate
@Table(name="Interns")
/*
 * Business Entity represents database table Interns
 */
@NamedQueries(
		{
		@NamedQuery(name="findAllInterns",query="from Interns I"),
		@NamedQuery(name="findInternById",query="from Interns I where
I.internId=:Id")
		}
		)
public class Interns implements Serializable{
	@Id
	@Column(name="intern_id")
	private int internId;
	@Column(name="intern_first_name")
	private String internFirstName;
	@Column(name="intern_last_name")
	private String internLastName;
	@Column(name="intern_age")
	private int internAge;
	@Column(name="intern_level")
	private Levels internLevel;
	@Transient
	private int semester1Marks;
	@Transient
	private int semester2Marks;
	@Transient
	private int semester3Marks;

	public Levels getInternLevel() {
		return internLevel;
	}

	public void setInternLevel(Levels internLevel) {
		this.internLevel = internLevel;
```

```
	}

	public int getSemester1Marks() {
		return semester1Marks;
	}

	public void setSemester1Marks(int semester1Marks) {
		this.semester1Marks = semester1Marks;
	}

	public int getSemester2Marks() {
		return semester2Marks;
	}

	public void setSemester2Marks(int semester2Marks) {
		this.semester2Marks = semester2Marks;
	}

	public int getSemester3Marks() {
		return semester3Marks;
	}

	public void setSemester3Marks(int semester3Marks) {
		this.semester3Marks = semester3Marks;
	}

	public int getInternAge() {
		return internAge;
	}

	public void setInternAge(int internAge) {
		this.internAge = internAge;
	}

	public int getInternId() {
		return internId;
	}

	public void setInternId(int internId) {
		this.internId = internId;
	}

	public String getInternFirstName() {
		return internFirstName;
	}

	public void setInternFirstName(String internFirstName) {
		this.internFirstName = internFirstName;
	}

	public String getInternLastName() {
		return internLastName;
	}

	public void setInternLastName(String internLastName) {
		this.internLastName = internLastName;
	}

	@Override
```

```
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + internAge;
		result = prime * result
				+ ((internFirstName == null) ? 0 :
internFirstName.hashCode());
		result = prime * result + internId;
		result = prime * result
				+ ((internLastName == null) ? 0 :
internLastName.hashCode());
		result = prime * result
				+ ((internLevel == null) ? 0 : internLevel.hashCode());
		result = prime * result + semester1Marks;
		result = prime * result + semester2Marks;
		result = prime * result + semester3Marks;
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Interns other = (Interns) obj;
		if (internAge != other.internAge)
			return false;
		if (internFirstName == null) {
			if (other.internFirstName != null)
				return false;
		} else if (!internFirstName.equals(other.internFirstName))
			return false;
		if (internId != other.internId)
			return false;
		if (internLastName == null) {
			if (other.internLastName != null)
				return false;
		} else if (!internLastName.equals(other.internLastName))
			return false;
		if (internLevel != other.internLevel)
			return false;
		if (semester1Marks != other.semester1Marks)
			return false;
		if (semester2Marks != other.semester2Marks)
			return false;
		if (semester3Marks != other.semester3Marks)
			return false;
		return true;
	}
	@Override
	public String toString() {
		return "Interns [internId=" + internId + ", internFirstName="
				+ internFirstName + ", internLastName=" +
internLastName
				+ ", internAge=" + internAge + ", internLevel=" +
internLevel
```

```
                        + ", semester1Marks=" + semester1Marks + ",
semester2Marks="
                        + semester2Marks + ", semester3Marks=" + semester3Marks
+ "]";
        }
}
```

Before we start developing data layer, We need to understand difference between REST and CRUD(create, retrieve, update, delete)
There are six guiding constraints of REST. These are:

Client-Server Mandate

This mandate underscores the fact that REST is a distributed approach via the nature of separation between client and server. Each service has multiple capabilities and listens for requests. Requests are made by a consumer and accepted or rejected by the server.

Statelessness

Due to the nature of statelessness, it is a guiding principle of RESTful architecture. It mandates what kind of commands can be offered between client and server. Implementing stateless requests means the communication between consumer and service is initiated by the request, and the request contains all the information necessary for the server to respond.

Cache

Cache mandates that server responses be labelled as either cacheable or not. Caching helps to mitigate some of the constraints of statelessness. For example, a request that is cached by the consumer in an attempt to avoid re-submitting the same request twice.

Interface / Uniform Contract

RESTful architecture follows the principles that define a Uniform Contract. This prohibits the use of multiple, self-contained interfaces within an API. Instead, one interface is distributed by hypermedia connections.

Layered System

This principle is the one that makes RESTful architecture so scalable. In a Layered System, multiple layers are used to grow and expand the interface. None of the layers can see into the other.
This allows for new commands and middleware to be added without impacting the original commands and functioning between client and server.

Optional: Code-On-Demand

RESTful applications don't have to include Code-On-Demand, but they must have Client-Server, Statelessness, Caching, Uniform Contract, and Layered Systems. Code-on-Demand allows logic within clients to be separate from that within servers. This allows them to be updated independently of server logic.
REST refers to a set of defining principles for developing API. It uses HTTP protocols like GET, PUT, POST to link resources to actions within a client-server relationship. In addition to the client-server mandate, it has several other defining constraints. The principles of RESTful architecture serve to create a stable and reliable application that offers simplicity and end-user satisfaction.

CRUD: Foundation and Principles

CRUD is an acronym for CREATE, READ, UPDATE, DELETE. These form the standard database commands that are the foundation of CRUD.
CRUD's origins are in database records.
By definition, CRUD is more of a cycle than an architectural system. On any dynamic website, there are likely multiple CRUD cycles that exist.
For instance, a buyer on an eCommerce site can CREATE an account, UPDATE account information, and DELETE things from a shopping cart.
A Warehouse Operations Manager using the same site can CREATE shipping records, RETRIEVE them as needed, and UPDATE supply lists. Retrieve is sometimes substituted for READ in the CRUD cycle.

Database Origins

The CRUD cycle is designed as a method of functions for enhancing persistent storage—with a database of records, for instance. As the name suggests, persistent storage outlives the processes that created it. These functions embody all the hallmarks of a relational database application.
In modern software development, CRUD has transcended its origins as foundational functions of a database and now maps itself to design principles for dynamic applications like HTTP protocol, DDS, and SQL.

Principles of CRUD

As mentioned above, the principles of the CRUD cycle are defined as CREATE, READ/RETRIEVE, UPDATE, and DELETE.
Create
CREATE procedures generate new records via INSERT statements.
Read/Retrieve
READ procedures reads the data based on input parameters. Similarly, RETRIEVE procedures grab records based on input parameters.
Update
UPDATE procedures modify records without overwriting them.
Delete
DELETE procedures delete where specified.

REST and CRUD Similarities

If you look at the two as we have described above, it may be difficult to understand why they are often treated in the same way. REST is a robust API architecture and CRUD is a cycle for keeping records current and permanent. The lack of clarity between the two is lost for many when they fail to determine when CRUD ends and REST begins. We mentioned above that CRUD can be mapped to DDS, SQL, and HTTP protocols. And that HTTP protocols are the link between resources in RESTful architecture, a core piece of REST's foundation.

Mapping CRUD principles to REST means understanding that GET, PUT, POST and CREATE, READ, UPDATE, DELETE have striking similarities because the former grouping

applies the principles of the latter. However, it is also important to note that a RESTful piece of software architecture means more than mapping GET, PUT, POST commands.

REST and CRUD: What's the Difference?

CRUD is a cycle that can be mapped to REST, by design. Permanence, as defined in the context of CRUD, is a smart way for applications to mitigate operational commands between clients and services. But REST governs much more than permanence within its principles of architecture.

Here are some of the ways that REST is not only different than CRUD but also offers much more:

- REST is an architectural system centered around resources and hypermedia, via HTTP protocols
- CRUD is a cycle meant for maintaining permanent records in a database setting
- CRUD principles are mapped to REST commands to comply with the goals of RESTful architecture

In REST:

- Representations must be uniform with regard to resources
- Hypermedia represents relationships between resources
- Only one entry into an API to create one self-contained interface, then hyperlink to create relationships

Developers select REST for a number of reasons:

1. Performance
2. Scalability
3. Simplicity

4. Modifiability
5. Visibility
6. Portability
7. Reliability

DAO Design pattern

We will implement DAO Design pattern. DAO stands for Data Access Object.
DAO Design Pattern is used to separate the data persistence logic in a separate layer. This way, the service remains completely in dark about how the low-level operations to access the database is done. This is known as the principle of **Separation of Logic**.
With DAO design pattern, we have following components on which our design depends:

- The model which is transferred from one layer to the other. In our application logical data model is **Interns** entity class.
- The interfaces which provides a flexible design. In our application we will create interface **InternsDAO** which will specify data operations for entity Interns.
- The interface implementation which is a concrete implementation of the persistence logic. We will implement data operations for entity interns in **InternsDAOImpl** using persistence provider as hibernate.

Create new package com.yash.dao. Create an interface InternsDAO.

```
package com.yash.dao;
import java.util.List;
import com.yash.entity.Interns;
public interface InternsDAO {
	List<Interns> getAllInterns();
	Interns getInternById(int internId);
	boolean storeInternData(Interns intern);
	boolean updateIntern(Interns intern);
	boolean updateInternLevel(Interns intern);
	boolean removeIntern(int internId);
}
```

We will provide implementation of InternsDAO interface in InternsDAOImpl implementation class.
We will use Spring JPA and Hibernate as persistence provider.
To enable JPA in a Spring Boot application, we have already mentioned spring-boot-starter and spring-boot-starter-data-jpa dependencies in pom.xml.
The *spring-boot-starter* contains the necessary auto-configuration for Spring JPA. Also, the *spring*-boot-starter-jpa project references all the necessary dependencies such as hibernate-entitymanager.
Spring Boot configures *Hibernate* **as the default JPA provider**, so it's no longer necessary to define the *entityManagerFactory* bean unless we want to customize it.
Spring Boot can also auto-configure the *dataSource* **bean, depending on the database we're using.**

Since we are using MySQL database we have mentioned *mysql-connector-java* dependency in pom.xml.
We have already mentioned below properties for Spring JPA Hibernate configuration in application.properties file.

```
spring.datasource.url=jdbc:mysql://localhost:3306/InternsDB
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.tomcat.max-wait=20000
spring.datasource.tomcat.max-active=50
spring.datasource.tomcat.max-idle=20
spring.datasource.tomcat.min-idle=15
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.properties.hibernate.id.new_generator_mappings = false
spring.jpa.properties.hibernate.format_sql = true
spring.jpa.hibernate.ddl-auto= update

logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Spring @Repository Annotation

We have applied @Repository annotation on InternsDAOImpl.

```
@Repository
public class InternsDAOImpl implements InternsDAO {
}
```

Spring Repository annotation is a specialization of @Component annotation, so Spring Repository classes are auto detected by spring framework through classpath scanning.
Spring Repository is very close to DAO pattern where DAO classes are responsible for providing CRUD operations on database tables
The EntityManager itself is created by the Spring Boot using the information in the application.properties. So to use it at runtime, we simply need to request it be injected InternsDAOImpl. We do this via @PersistenceContext

```
@PersistenceContext
private EntityManager manager;
```

We will now discuss implementation of method getAlInterns() declared in InternsDAO interface.
In business entity class Interns we have applied annotation @NamedQueries

```
@NamedQueries(
		{
		@NamedQuery(name="findAllInterns",query="from Interns I"),
		@NamedQuery(name="findInternById",query="from Interns I where
I.internId=:Id")
		}
		)
public class Interns implements Serializable {}
```

JPA Named Queries

A named query is a statically defined query with a predefined unchangeable query string. Using named queries instead of dynamic queries may improve code organization by

separating the JPQL query strings from the Java code. It also enforces the use of query parameters rather than embedding literals dynamically into the query string and results in more efficient queries.

The following @NamedQuery annotation defines a query whose name is " findAllInterns" that retrieves all the Interns objects in the database and “findInternById” that retrieves Interns object based on intern id in the database

```
@NamedQueries(
		{
		@NamedQuery(name="findAllInterns",query="from Interns I"),
		@NamedQuery(name="findInternById",query="from Interns I where
I.internId=:Id")
		})
```

We have used this named queries in following methods

```
	@Override
	public List<Interns> getAllInterns() {
		// TODO Auto-generated method stub
		Query query=manager.createNamedQuery("findAllInterns");
		List<Interns> interns=query.getResultList();
		return interns;}

	@Override
	public Interns getInternById(int internId) {
		// TODO Auto-generated method stub
		Query query=manager.createNamedQuery("findInternById");
		query.setParameter("Id", internId);
		return (Interns)query.getSingleResult();
	}
```

In EntityManager interface method createNamedQuery () creates an instance of Query for executing a Java Persistence named query language statement. Query interface is used to control query execution.

getResultList() returns List of Interns entity objects

getSingleResult() returns a single Intern entity object

Remaining methods storeInternData() for persisting Intern object, updateIntern() updates Intern details, updateInternLevel() updates intern level and removeIntern() removes intern details based on intern id.

Only all these methods @Transactional is applied.

By using @Transactional, many important aspects such as transaction propagation are handled automatically by Spring. In this case if another transactional method is called by businessLogic(), that method will have the option of joining the ongoing transaction.

```
	@Transactional
	public boolean storeInternData(Interns intern) {
		// TODO Auto-generated method stub
		manager.persist(intern);
		Interns intern_found=getInternById(intern.getInternId());
		if(intern_found!=null){
		return true;}
		else{
			return false;
		}

	}


	@Override
	 @Transactional
	public boolean updateIntern(Interns intern) {
		// TODO Auto-generated method stub
		 manager.merge(intern);
		Interns intern_db=getInternById(intern.getInternId());
		if(!intern.equals(intern_db))
			return false;
		return true;
}
	@Override
	 @Transactional
	public boolean updateInternLevel(Interns intern) {
		// TODO Auto-generated method stub
		Interns intern_db=getInternById(intern.getInternId());
		Levels intern_Level_updated_before = intern_db.getInternLevel();
		Levels intern_Level_updated = intern.getInternLevel();
	  Interns interns= manager.find(Interns.class, intern.getInternId());
		interns.setInternLevel(intern_Level_updated);
		if(intern_Level_updated_before==intern_Level_updated)
			return false;
		return true;
	}

	@Override
	@Transactional
	public boolean removeIntern(int internId) {
		// TODO Auto-generated method stub
		Interns intern=getInternById(internId);
		manager.remove(intern);
		return true;

	}
```

1. persist() of EntityManager makes an instance of Interns managed and persistent.

2. merge() of EntityManager is used to merge the changes made to a detached object into the persistence context. merge does not directly update the object into the database, it merges the changes into the persistence context (transaction).

3. remove() of EntityManager is used to remove entity Interns based on intern id

Full implementation class is as below

```
package com.yash.dao;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import com.yash.entity.Interns;
import com.yash.helper.Levels;
@Repository
public class InternsDAOImpl implements InternsDAO {
	@PersistenceContext
	private EntityManager manager;
	public List<Interns> getAllInterns() {
		// TODO Auto-generated method stub
		Query query=manager.createNamedQuery("findAllInterns");
		List<Interns> interns=query.getResultList();
		return interns;
	}
	public Interns getInternById(int internId) {
		// TODO Auto-generated method stub
		Query query=manager.createNamedQuery("findInternById");
		query.setParameter("Id", internId);
		return (Interns)query.getSingleResult();
	}
	@Override
	@Transactional
	public boolean storeInternData(Interns intern) {
		// TODO Auto-generated method stub
		manager.persist(intern);
		Interns intern_found=getInternById(intern.getInternId());
		if(intern_found!=null){
		return true;}
		else{
			return false;
		}
	}

	@Override
	@Transactional
	public boolean updateIntern(Interns intern) {
		// TODO Auto-generated method stub
	manager.merge(intern);
		Interns intern_db=getInternById(intern.getInternId());
		if(intern.equals(intern_db))
			return false;
		return true;

	@Override
	@Transactional
```

```
	public boolean updateInternLevel(Interns intern) {
		// TODO Auto-generated method stub
		Interns intern_db=getInternById(intern.getInternId());
		Levels intern_Level_updated_before = intern_db.getInternLevel();
		Levels intern_Level_updated = intern.getInternLevel();
		Interns interns= manager.find(Interns.class, intern.getInternId());
		interns.setInternLevel(intern_Level_updated);
		if(intern_Level_updated_before==intern_Level_updated)
			return false;
		return true
	}
	@Override
	@Transactional
	public boolean removeIntern(int internId) {
		// TODO Auto-generated method stub
		Interns intern=getInternById(internId);
		manager.remove(intern);
		return true;
	}

}
```

A business exception is thrown when a business rule within our application is violated. We will create custom exception handling class InternsException in com.yash.exception package. We will utilise this class in the service layer.

```
package com.yash.exception;
public class InternsException extends RuntimeException{
	private String message;
	public InternsException(String message){
		this.message=message;
	}
	public String getMessage() {
		return message;
	}
}
```

Service Layer

"Service layer" is an architectural term. It refers to a portion of the system that sits somewhere in the middle of a multi-tier architecture, below the user interaction layer but above the data access layer. Business logic can be *implemented* in the service layer, thereby *enforcing* business rules.

The idea behind such a layer is to have an architecture which can support multiple presentation layers such as web, mobile, etc. Mostly it has a separate physical tier of its own to cleanly segregate it with any presentation layer. This provides easier management, better abstraction and scalability supporting large number of simultaneous clients.

A business component consists of a public interface describing the contract the component is offering and a hidden implementation. Technically the public part is a

collection of interfaces, DTO(Data transfer Object) classes and exceptions, while the hidden part includes the implementation of the interfaces.

We will create business interface InternsService to expose business services for an business entity class Interns.

Business interface will have declaration of following business operations:

1. Retrieval of all interns service
2. Retrieval of intern based on intern id
3. Registration of intern service
4. Updating intern service
5. Deregistration of intern service

Create package com.yash.service and create business interface InternsService

```
package com.yash.service;
import java.util.List;
import com.yash.entity.Interns;
public interface InternsService {
	List<Interns> retrieveInternsService();
	Interns retrieveInternsByIdService(int internId);
	boolean registerInternService(Interns interns);
	boolean updateInternService(Interns interns);
	boolean updateInternLevelService(Interns interns);
	boolean removeInternService(int internId);


}
```

We will provide implementation of above business interface in InternsServiceImpl class
Note that we will mark this class as business service component implementation using @Service.

```
@Service
public class InternsServiceImpl implements InternsService {
}
```

Spring @Service annotation is used with classes that provide some business functionalities. Spring context will autodetect these classes when annotation-based configuration and class path scanning is used.
Spring @Autowired annotation is used for automatic dependency injection.

```
@Autowired
	private InternsDAO internsDAO;
```

In above code we will inject InternsDAOImpl on field internsDAO using @Autowired. Since we have only one implementation of InternsDAO, @Qualifier is not required.

Below are implementation of business methods in InternsServiceImpl

1. retrieveInternsService() provides service of retrieving intern details
2. retrieveInternsByIdService(internId) provides service of fetching intern details based on intern id
3. determineLevelBySemesterMarks() provides the service to determine level of intern based on semester scores.
4. registerInternService() registers intern details based on eligibility criteria
5. updateInternService() updates intern details
6. updateInternLevelService() updates interns level based on eligibility criteria

```
package com.yash.service;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.yash.dao.InternsDAO;
import com.yash.entity.Interns;
import com.yash.exception.InternsException;
import com.yash.helper.Levels;
@Service
public class InternsServiceImpl implements InternsService {
	@Autowired
	private InternsDAO internsDAO;
	public List<Interns> retrieveInternsService() {
		// TODO Auto-generated method stub
		List<Interns> interns= internsDAO.getAllInterns();
		if(interns.isEmpty()){
			throw new InternsException("No interns records found");
		}else{
			return interns;
		}
	}
	public Interns retrieveInternsByIdService(int internId) {
		// TODO Auto-generated method stub
		Interns intern=internsDAO.getInternById(internId);
		if(intern.getInternId()==0){
			throw new InternsException("Intern not found");
		}
		return intern;
	}
	public Levels determineLevelBySemesterMarks(Interns interns){
		int sem1Marks=interns.getSemester1Marks();
		int sem2Marks=interns.getSemester2Marks();
		int sem3Marks=interns.getSemester3Marks();
		int semAverage=(sem1Marks+sem2Marks+sem3Marks)/3;
		if(semAverage<=50){
			throw new InternsException("Intern did not match
eligibility");
		}
		if(semAverage>50 && semAverage<=60){
			return Levels.BEGINNER;
		}else if(semAverage>60 && semAverage<70){
			return Levels.INTERMEDIATE;
		}else{
			return Levels.ADVANCED;
		}
	}
	public boolean registerInternService(Interns interns) {
```

```
		// TODO Auto-generated method stub
		Levels level=determineLevelBySemesterMarks(interns);
		interns.setInternLevel(level);
		return internsDAO.storeInternData(interns);
	}
	@Override
	public boolean updateInternService(Interns interns) {
		// TODO Auto-generated method stub
		Levels level=determineLevelBySemesterMarks(interns);
		interns.setInternLevel(level);
		boolean checkUpdation=internsDAO.updateIntern(interns);
		if(checkUpdation==false)
			throw new InternsException("Updation failed.....");

		return checkUpdation;
	}

    @Override
    public boolean updateInternLevelService(Interns interns) {
		// TODO Auto-generated method stub
		Levels level=determineLevelBySemesterMarks(interns);
		interns.setInternLevel(level);
		boolean checkUpdation=internsDAO.updateInternLevel(interns);
		if(checkUpdation==false)
			throw new InternsException("Updation failed.....");
		return checkUpdation;
	}
}

@Override
	public boolean removeInternService(int internId) {
		// TODO Auto-generated method stub
		boolean checkRemoval=internsDAO.removeIntern(internId);
		if(checkRemoval==false)
			throw new InternsException("Deletion failed....");
		return checkRemoval;
	}
```

Web Layer

The web layer is the uppermost layer of a web application. It is responsible of processing user's input and returning the correct response back to the user. The web layer must also handle the exceptions thrown by the other layers. Because the web layer is the entry point of our application, it must take care of authentication and act as a first line of defense against unauthorized users.

Spring MVC

Spring MVC is the primary web framework built on the Servlet API. It is build on the popular MVC design pattern. *MVC (Model-View-Controller)* is a software architecture pattern, which separates application into three areas: model, view, and controller. The model represents a Java object carrying data. The view represents the visualization of the data that the model contains. The controller controls the data flow into model object and updates the view when the data changes. It separates the view and model.
We will utilise Spring REST module of Spring framework for creating restful web services.
We will create a controller class InternsController to handle incoming request from UI application in package com.yash.controller

```
@RestController
@RequestMapping("/internsApp")
public class InternsController {
}
```

Spring RestController annotation is a convenience annotation that is itself annotated with @Controller and @ResponseBody. This annotation is applied to a class to mark it as a request handler.
Spring RestController annotation is used to create RESTful web services using Spring MVC. Spring RestController takes care of mapping request data to the defined request handler method. Once response body is generated from the handler method, it converts it to JSON or XML response.
The `@RequestMapping` annotation provides "routing" information. It tells Spring that any HTTP request with the `/internsApp` path should be mapped to the `InternsController` methods.

Spring Boot provides integration with three JSON mapping libraries:

- Gson
- Jackson
- JSON-B

Jackson is the preferred and default library. Auto-configuration for Jackson is provided and Jackson is part of `spring-boot-starter-json`. When Jackson is on the classpath an `ObjectMapper` bean is automatically configured. So in our RestController response body generated from the handler method wil be implicitly converted to JSON response.

RestController will invoke business service based on URI mapping. We have injected business service interface using @Autowired. Since as of now there is only one implementation of InternsService interface, @Qualifier is not required.

```
@Autowired
	private InternsService internService;
```

HTTP has defined few sets of methods which indicate the type of action to be performed on the resources. The URL is a sentence, where resources are nouns and HTTP methods are verbs.

HTTP GET

Use GET requests **to retrieve resource representation/information only** – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be **safe methods**. Additionally, GET APIs should be **idempotent**, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The paths should contain the plural form of resources and the HTTP method should define the kind of action to be performed on the resource.

For any given HTTP GET API, if the resource is found on the server then it must return HTTP response code 200 (OK) – along with response body which is usually either XML or JSON content (due to their platform independent nature).

In case resource is NOT found on server then it must return HTTP response code 404 (NOT FOUND). Similarly, if it is determined that GET request itself is not correctly formed then server will return HTTP response code 400 (BAD REQUEST).

The resource should always be **plural** in the API endpoint and if we want to access one instance of the resource, we can always pass the id in the URL.

In REST, primary data representation is called **Resource**. Having a strong and consistent REST resource naming strategy – will definitely prove your one of the best design decisions in long term.

The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a Intern), and so on. In other words, any concept that might be the target of an author's

hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

-Roy Fielding's dissertation

A **resource can be a singleton or a collection**. For example, "`interns`" is a collection resource and "`intern`" is a singleton resource. We can identify "`interns`" collection resource using the URI "`/yash-interns`". We can identify a single "`intern`" resource using the URI "`/yash-interns/{internId}`".

A **resource may contain sub-collection resources** also. For example, sub-collection resource "`projects`" of a particular "`intern`" can be identified using the URN "`/yash-interns/{internId}/projects`". Similarly, a singleton resource "`project`" inside the sub-collection resource "`projects`" can be identified as follows: "`/yash-interns/{internId}/projects/{projectId}`".

REST APIs use Uniform Resource Identifiers (URIs) to address resources. REST API designers should create URIs that convey a REST API's resource model to its potential client developers. When resources are named well, an API is intuitive and easy to use. If done poorly, that same API can feel difficult to use and understand

You can follow any casing convention, but make sure it is consistent across the application. If the request body or response type is JSON then please follow camelCase to maintain the consistency.

REST Resource Naming Best Practices

Use nouns to represent resources

RESTful URI should refer to a resource that is a thing (noun) instead of referring to an action (verb) because nouns have properties which verbs do not have – similar to resources have attributes. Some examples of a resource are:

- Users of the system
- User Accounts
- Network Devices etc.

and their resource URIs can be designed as below:

```
http://api.example.com/device-management/managed-devices
http://api.example.com/device-management/managed-devices/{device-id}
http://api.example.com/user-management/users/
http://api.example.com/user-management/users/{id}
```

For more clarity, let's divide the **resource archetypes** into four categories (document, collection, store and controller) and then **you should always target to put a resource into one archetype and then use it's naming convention consistently**. *For uniformity's sake, resist the temptation to design resources that are hybrids of more than one archetype.*

1. document

A document resource is a singular concept that is akin to an object instance or database record. In REST, you can view it as a single resource inside resource collection. A document's state representation typically includes both fields with values and links to other related resources.

Use "singular" name to denote document resource archetype.

```
http://api.example.com/device-management/managed-devices/{device-id}
http://api.example.com/user-management/users/{id}
http://api.example.com/user-management/users/admin
```

2. collection

A collection resource is a server-managed directory of resources. Clients may propose new resources to be added to a collection. However, it is up to the collection to choose to create a new resource, or not. A collection resource chooses what it wants to contain and also decides the URIs of each contained resource.

Use "plural" name to denote collection resource archetype.

```
http://api.example.com/device-management/managed-devices
http://api.example.com/user-management/users
http://api.example.com/user-management/users/{id}/accounts
```

3. store

A store is a client-managed resource repository. A store resource lets an API client put resources in, get them back out, and decide when to delete them. A store never generates new URIs. Instead, each stored resource has a URI that was chosen by a client when it was initially put into the store.

Use “plural” name to denote store resource archetype.

```
http://api.example.com/cart-management/users/{id}/carts
http://api.example.com/song-management/users/{id}/playlists
```

4. *controller*

A controller resource models a procedural concept. Controller resources are like executable functions, with parameters and return values; inputs and outputs.

Use “verb” to denote controller archetype.

```
http://api.example.com/cart-management/users/{id}/cart/checkout
http://api.example.com/song-management/users/{id}/playlist/play
```

Consistency is the key

Use consistent resource naming conventions and URI formatting for minimum ambiguity and maximum readability and maintainability. You may implement below design hints to achieve consistency:

1. *Use forward slash (/) to indicate a hierarchical relationships*

The forward slash (/) character is used in the path portion of the URI to indicate a hierarchical relationship between resources. e.g.

```
http://api.example.com/device-management
http://api.example.com/device-management/managed-devices
http://api.example.com/device-management/managed-devices/{id}
http://api.example.com/device-management/managed-devices/{id}/scripts
http://api.example.com/device-management/managed-devices/{id}/scripts/{id}
```

2. *Do not use trailing forward slash (/) in URIs*

As the last character within a URI’s path, a forward slash (/) adds no semantic value and may cause confusion. It’s better to drop them completely.

```
http://api.example.com/device-management/managed-devices/
http://api.example.com/device-management/managed-devices          /*This is much better version*/
```

3. *Use hyphens (-) to improve the readability of URIs*

To make your URIs easy for people to scan and interpret, use the hyphen (-) character to improve the readability of names in long path segments.

```
http://api.example.com/inventory-management/managed-entities/{id}/install-script-location  //More readable
http://api.example.com/inventory-management/managedEntities/{id}/installScriptLocation  //Less readable
```

4. Do not use underscores (_)

It's possible to use an underscore in place of a hyphen to be used as separator – But depending on the application's font, it's possible that the underscore (_) character can either get partially obscured or completely hidden in some browsers or screens.

To avoid this confusion, use hyphens (-) instead of underscores (_).

```
http://api.example.com/inventory-management/managed-entities/{id}/install-script-location  //More readable
http://api.example.com/inventory_management/managed_entities/{id}/install_script_location  //More error prone
```

5. Use lowercase letters in URIs

When convenient, lowercase letters should be consistently preferred in URI paths.

RFC 3986 defines URIs as case-sensitive except for the scheme and host components. e.g.

```
http://api.example.org/my-folder/my-doc  //1
HTTP://API.EXAMPLE.ORG/my-folder/my-doc  //2
http://api.example.org/My-Folder/my-doc  //3
```

In above examples, 1 and 2 are same but 3 is not as it uses **My-Folder** in capital letters.

6. Do not use file extensions

File extensions look bad and do not add any advantage. Removing them decrease the length of URIs as well. No reason to keep them.

Apart from above reason, if you want to highlight the media type of API using file extension then you should rely on the media type, as communicated through the `Content-Type` header, to determine how to process the body's content.

```
http://api.example.com/device-management/managed-devices.xml  /*Do not use it*/
http://api.example.com/device-management/managed-devices  /*This is correct URI*/
```

Never use CRUD function names in URIs

URIs should not be used to indicate that a CRUD function is performed. URIs should be used to uniquely identify resources and not any action upon them. HTTP request methods should be used to indicate which CRUD function is performed.

```
HTTP GET http://api.example.com/device-management/managed-devices  //Get all devices
HTTP POST http://api.example.com/device-management/managed-devices  //Create new Device

HTTP GET http://api.example.com/device-management/managed-devices/{id}  //Get device for given Id
HTTP PUT http://api.example.com/device-management/managed-devices/{id}  //Update device for given Id
HTTP DELETE http://api.example.com/device-management/managed-devices/{id}  //Delete device for given Id
```

Use query component to filter URI collection

Many times, you will come across requirements where you will need a collection of resources sorted, filtered or limited based on some certain resource attribute. For this, do not create new APIs – rather enable sorting, filtering and pagination capabilities in resource collection API and pass the input parameters as query parameters. e.g.

```
http://api.example.com/device-management/managed-devices
http://api.example.com/device-management/managed-devices?region=USA
http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ
http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ&sort=installation-date
```

Create package com.yash.controller and create a class InternsController
Complete listing of code of InternsController is as below:

```
package com.yash.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import com.yash.entity.Interns;
import com.yash.service.InternsService;

@RestController
@RequestMapping("/interns-management")
public class InternsController {
	@Autowired
	private InternsService internService;
	@GetMapping("/yash-interns")
	public ResponseEntity<List<Interns>> retrieveAllInterns(){
		List<Interns> internsList=internService.retrieveInternsService();
		ResponseEntity<List<Interns>> response=new
ResponseEntity<List<Interns>>(internsList,HttpStatus.OK);
		return response;
	}
	@GetMapping("/yash-interns/{internId}")
	public ResponseEntity<Interns> retrieveInternById(@PathVariable("internId")
int internId){
		Interns interns=internService.retrieveInternsByIdService(internId);
		ResponseEntity<Interns> response=null;
		if(interns.getInternId()!=0){
		response=new ResponseEntity<Interns>(interns,HttpStatus.FOUND);
		}else{
		response=new ResponseEntity<Interns>(interns,HttpStatus.NOT_FOUND);
		}
		return response;
	}

	@PostMapping("yash-interns")
	public ResponseEntity<Void> registerIntern(@RequestBody Interns interns){
		boolean
registrationResult=internService.registerInternService(interns);
		ResponseEntity<Void> response=null;
		if(registrationResult){
			response=new ResponseEntity<Void>(HttpStatus.CREATED);
		}else{
			response=new ResponseEntity<Void>(HttpStatus.CONFLICT);
		}
		return response;
	}
	@PutMapping("yash-interns-manage")
	public ResponseEntity<Void> updateLevel(@RequestBody Interns interns)
	{
		boolean updateIntern=internService.updateInternService(interns);
		ResponseEntity<Void> response=null;
		if(updateIntern==true)
			response=new ResponseEntity<Void>(HttpStatus.ACCEPTED);
		else
			response=new ResponseEntity<Void>(HttpStatus.NOT_MODIFIED);
		return response;
	}
	@PatchMapping("yash-interns-level")
	public ResponseEntity<Void> updateInternLevel(@RequestBody Interns interns)
	{
		boolean updateLevel=internService.updateInternLevelService(interns);
		ResponseEntity<Void> response=null;
		if(updateLevel==true)
			response=new ResponseEntity<Void>(HttpStatus.ACCEPTED);
		else
```

```
                    response=new ResponseEntity<Void>(HttpStatus.NOT_MODIFIED);
              return response;
        }
}
@DeleteMapping("yash-interns-manage/{internId}")
      public ResponseEntity<Void> deleteStudent(@PathVariable("internId")int
internId){
      ResponseEntity<Void> response=null;
    boolean internRemoved=internService.removeInternService(internId);
    if(internRemoved){
                    response=new ResponseEntity<Void>(HttpStatus.OK);
              }else{
                    response=new ResponseEntity<Void>(HttpStatus.NOT_FOUND);

              }
              return response;
}
```

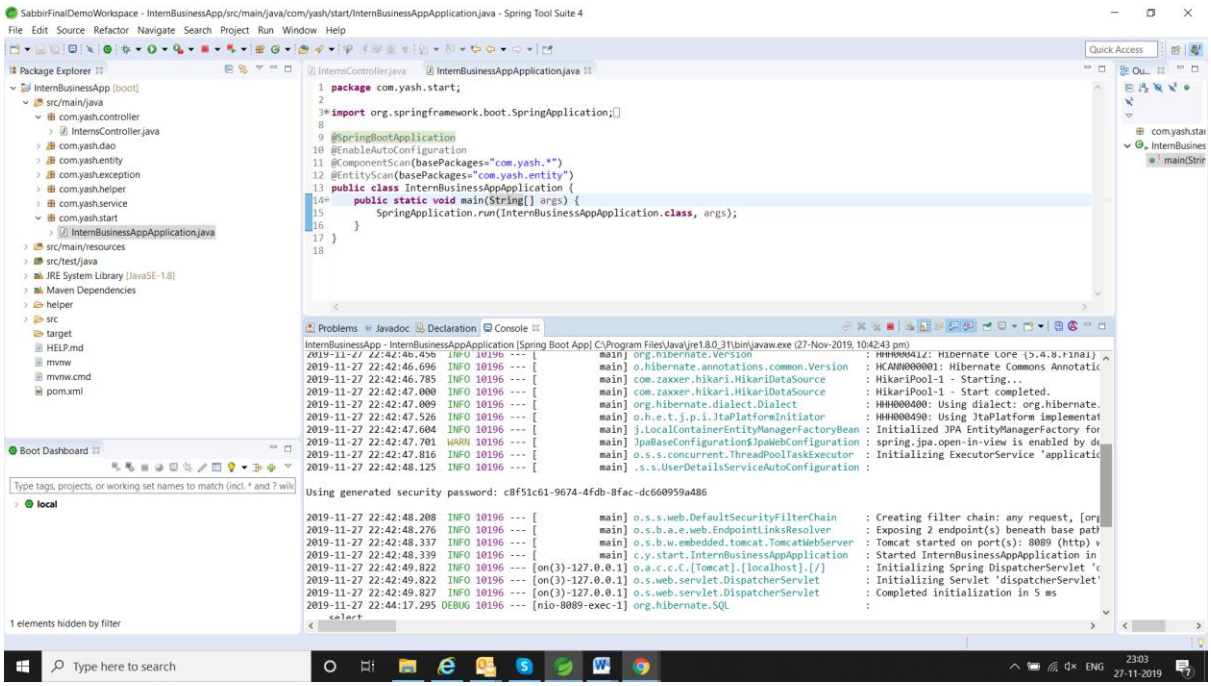
Let's discuss each method in InternsController.

For retrieval of all interns details we have retrieveAllInterns(). We will disucss pagination in next section of lab.

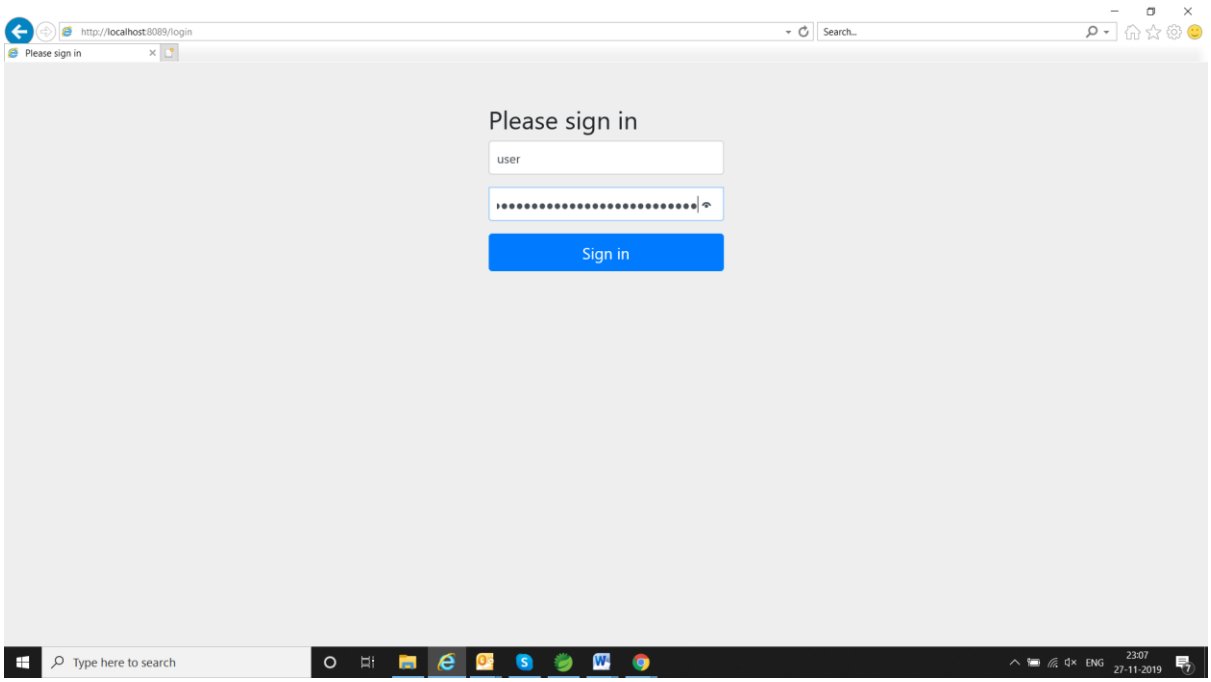
```
@GetMapping("/yash-interns")
      public ResponseEntity<List<Interns>> retrieveAllInterns(){
              List<Interns> internsList=internService.retrieveInternsService();
              ResponseEntity<List<Interns>> response=new
ResponseEntity<List<Interns>>(internsList,HttpStatus.OK);
              return response;
      }
```

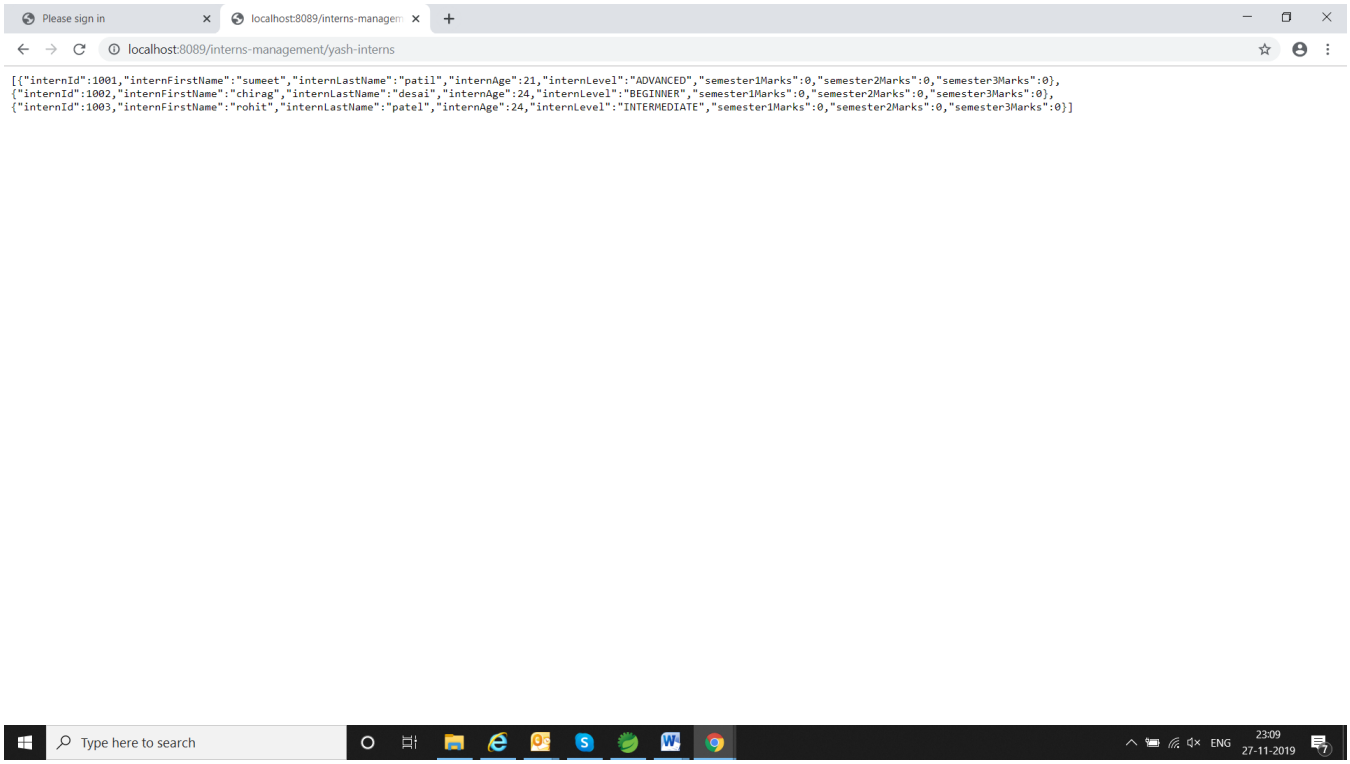
@GetMapping annotation maps HTTP GET requests onto specific handler methods. It is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).

Return type of method is ResponseEntity<List<Interns>>
`ResponseEntity` represents an HTTP response, including headers, body, and status. While `@ResponseBody` puts the return value into the body of the response, `ResponseEntity` also allows us to add headers and status code.
To test method in browser, run `InternBusinessAppApplication`
Spring boot application starts with embedded tomcat on port 8089



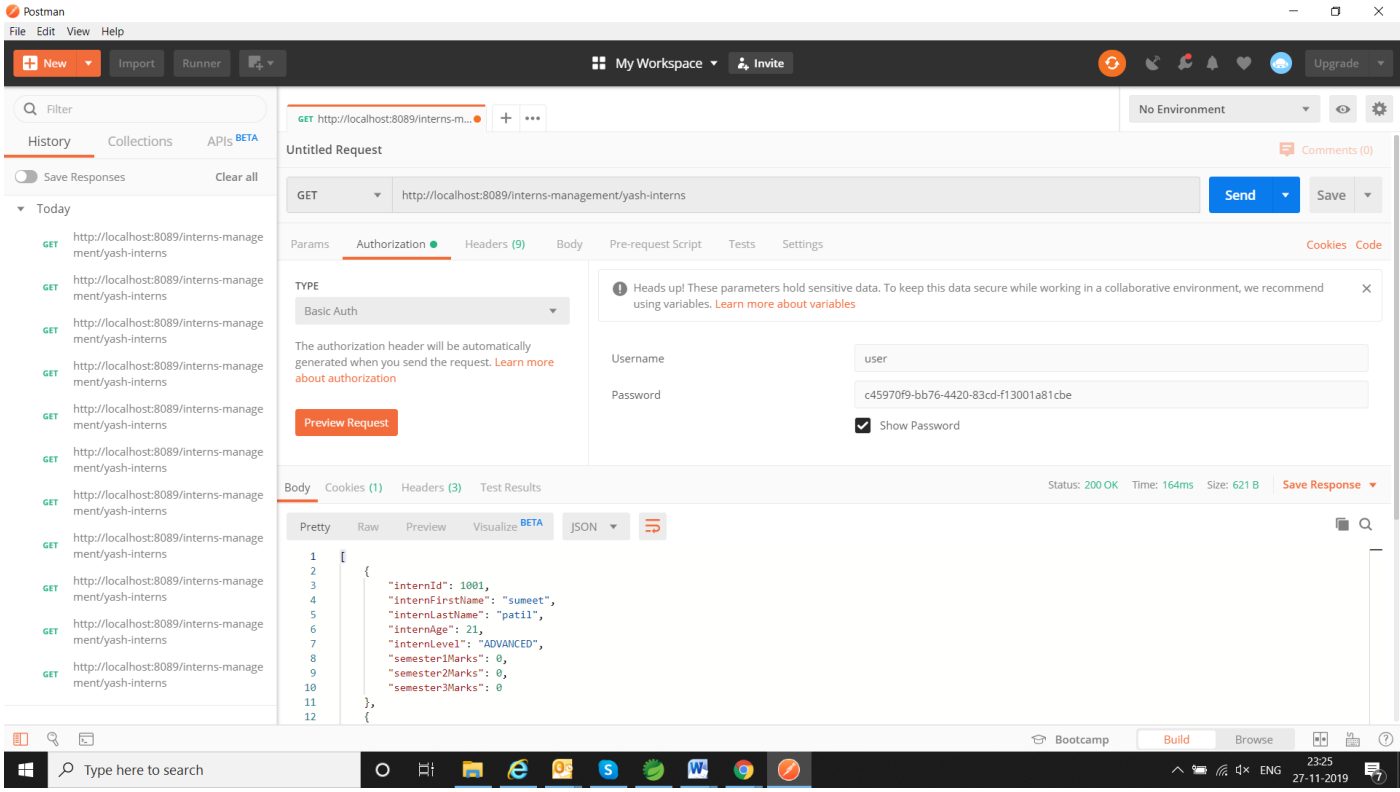
In the browser type url: http://localhost:8089/interns-management/yash-interns
Since we have mentioned spring dependency for security a login page is displayed. Enter username as “user” and generated security password mentioned on console of our application.



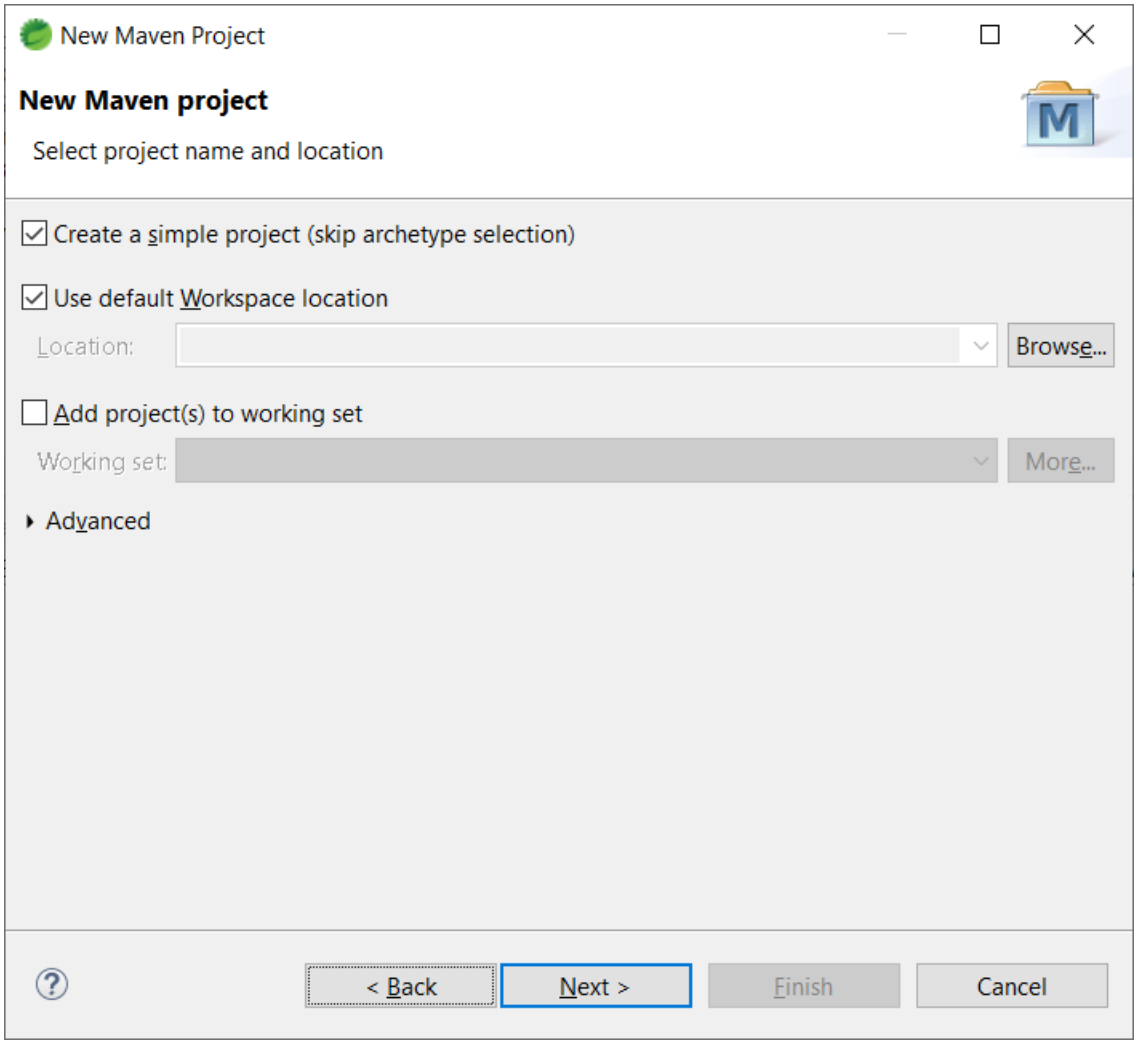


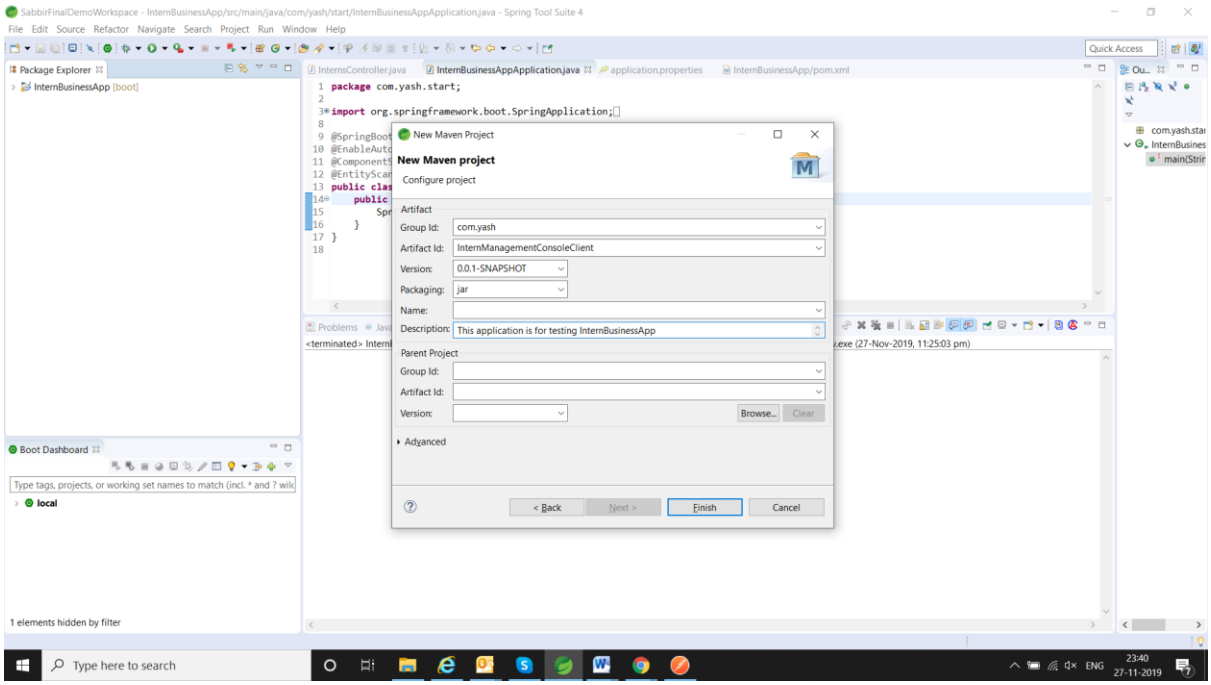
You can see JSON array with records of Interns. Since marks are trainsient and not persisted in database have default value 0.

Testing in Postman



Now we will create console client application for testing purpose. In next section of lab we will discuss how to make Restful call from InternsUIApp through angular/react.





In client application as well we will use Spring Boot.
Mention below dependencies in pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>2.2.1.RELEASE</version>
            <relativePath/> <!-- lookup parent from repository -->
      </parent>
  <groupId>com.yash</groupId>
  <artifactId>InternManagementConsoleClient</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <description>This application is for testing InternBusinessApp</description>
  <properties>
            <java.version>1.8</java.version>
      </properties>
      <dependencies>
         <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
          </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-activemq</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.activemq</groupId>
            <artifactId>activemq-broker</artifactId>
        </dependency>

         <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </dependency>
      </dependencies>
      <build>
            <plugins>
                  <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                  </plugin>
            </plugins>
      </build>
</project>
```

In application.properties,

```
server.port=8081
spring.application.name = InternManagementConsoleClient
```

Important Note: Business entity will have business validation and database validation rules. Ideally We should never expose business entity to client application. In next section we will create model wrapper class for our business entity Interns and return model wrapper to client.

For testing purpose,

copy com.yash.entity.Interns and com.yash.helper package in InternManagementConsoleClient application. Refactor rename com.yash.entity to com.yash.model.
Since we are trying to convert JSON returned by our web service to a java object using jackson mapper, we have to create the model class for this. Note that this model class will be very similar to the entity class used in the web service, except that here we don't need JPA annotations.

Spring RestTemplate

Spring RestTemplate provides a convenient way to test RESTful web services.

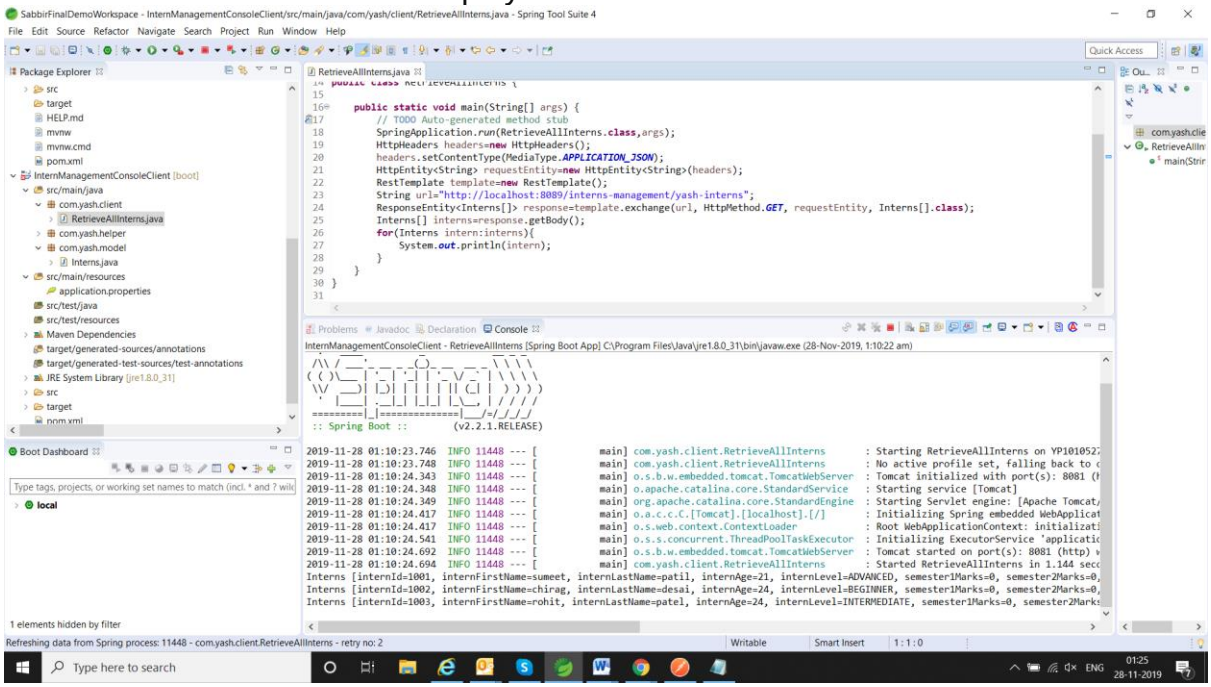
- Spring RestTemplate class is part of spring-web, introduced in Spring 3.
- We can use RestTemplate to test HTTP based restful web services, it doesn't support HTTPS protocol.
- RestTemplate class provides overloaded methods for different HTTP methods, such as GET, POST, PUT, DELETE etc.

Create a package com.yash.client and a class named RetrieveAllInterns to test retrieval of all interns details.
HttpHeaders class is a data structure representing HTTP request or response headers, mapping String header names to a list of String values, and also offers accessors for common application-level data types.

```
package com.yash.client;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
import com.yash.model.Interns;
@SpringBootApplication
@EnableAutoConfiguration
public class RetrieveAllInterns {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SpringApplication.run(RetrieveAllInterns.class,args);
		HttpHeaders headers=new HttpHeaders();
		headers.setContentType(MediaType.APPLICATION_JSON);
		HttpEntity<String> requestEntity=new HttpEntity<String>(headers);
		RestTemplate template=new RestTemplate();
		String url="http://localhost:8089/interns-management/yash-interns";
		ResponseEntity<Interns[]> response=template.exchange(url,
HttpMethod.GET, requestEntity, Interns[].class);
	  Interns[] interns=response.getBody();
	  for(Interns intern:interns){
		System.out.println(intern);
	  }
	}
```

```
}
```

Please ensure `InternBusinessAppApplication` in InternBusinessApp is running
Run RetrieveAllInterns and it displays all interns details on another console.

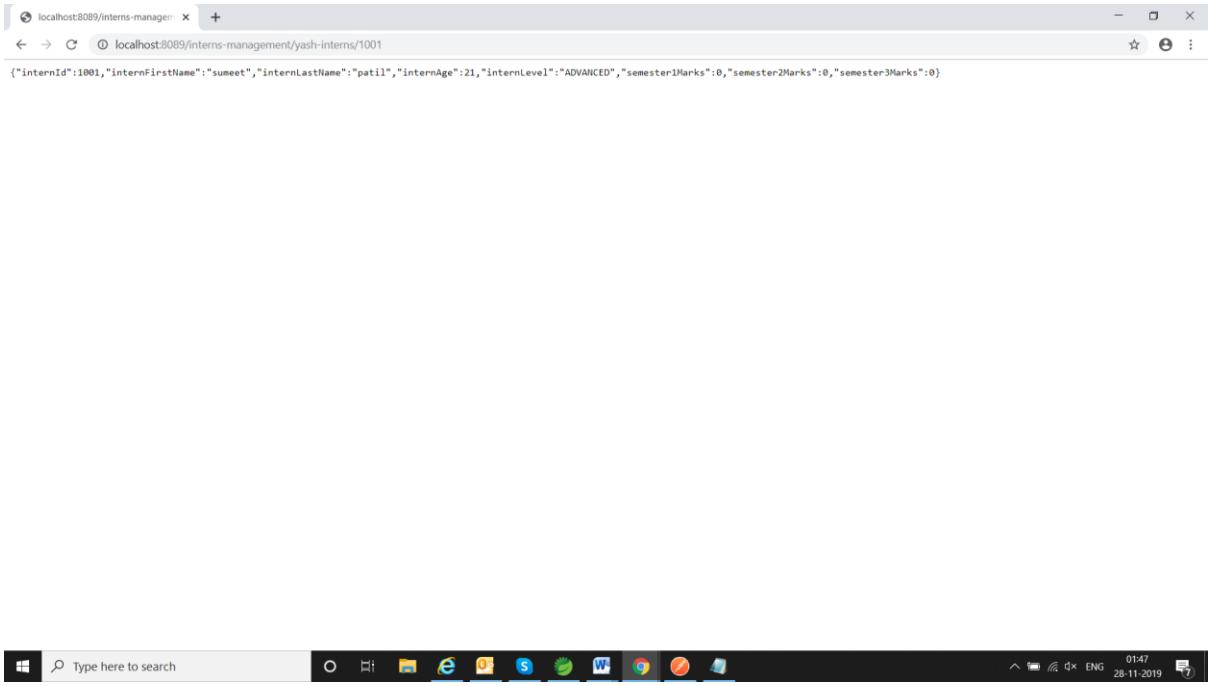


We will now complete service for retrieval of intern based on internId.
In InternBusinessApp, InternsController
As discussed previously HTTP URI for fetching interns details based on intern id will be
/yash-interns/{internId}
@PathVariable is used to extract values from the HTTP request.
There is a difference between @RequestParam and @PathVariable
As the name suggests, `@RequestParam` is used to get the request parameters from URL, also known as query parameters, while `@PathVariable` extracts values from URI.

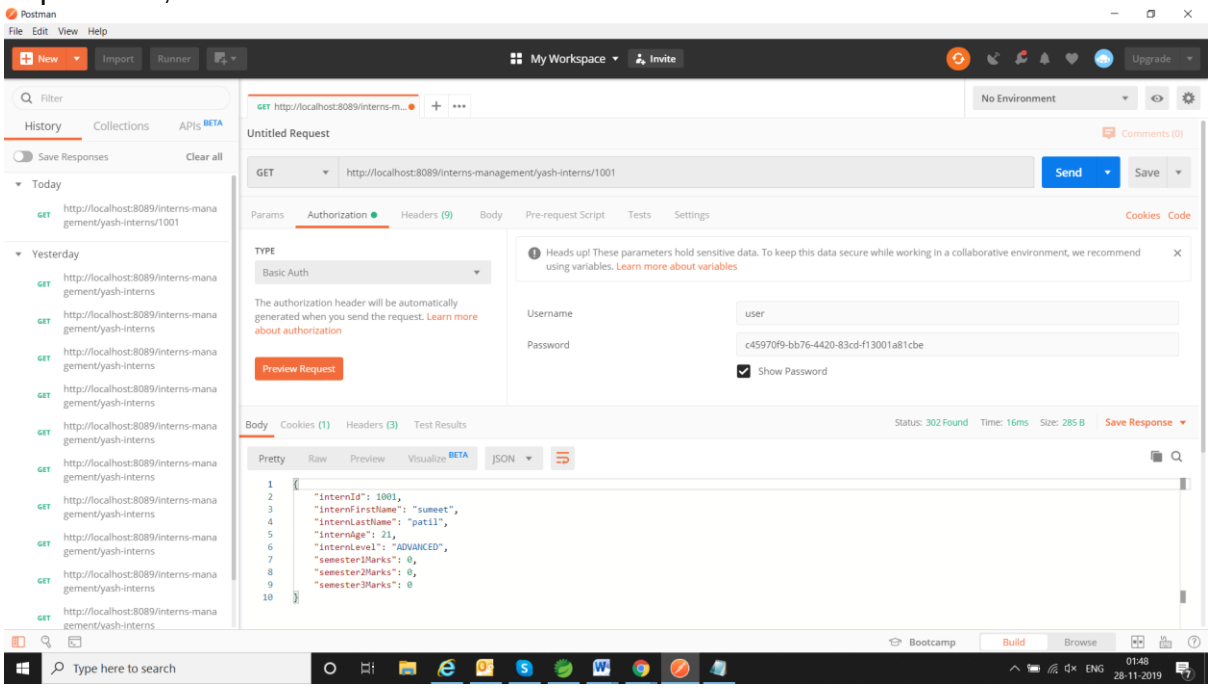
HttpStatus is enumeration of HTTP status codes.

```
      @GetMapping("/yash-interns/{internId}")
      public ResponseEntity<Interns> retrieveInternById(@PathVariable("internId")
int internId){
            Interns interns=internService.retrieveInternsByIdService(internId);
            ResponseEntity<Interns> response=null;
            if(interns.getInternId()!=0){
            response=new ResponseEntity<Interns>(interns,HttpStatus.FOUND);
            }else{
            response=new ResponseEntity<Interns>(interns,HttpStatus.NOT_FOUND);
            }
            return response;
        }
```

To test service in browser,



In postman,



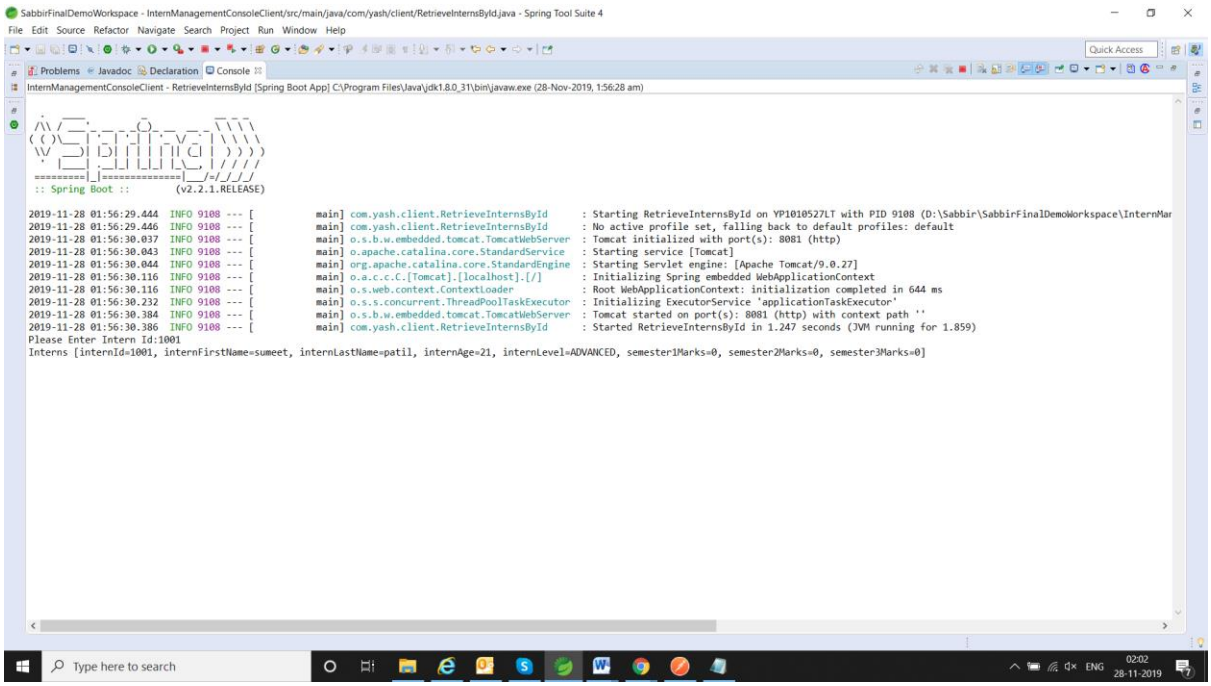
Testing using RestTemplate,
Create a new RetrieveInternsById class in com.yash.client package,
In RestTemplate class, exchange() does below

Execute the HTTP method to the given URI template, writing the given HttpEntity to the request, and returns the response as ResponseEntity.

```
package com.yash.client;
import java.util.Scanner;
```

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
import com.yash.model.Interns;

@SpringBootApplication
@EnableAutoConfiguration
public class RetrieveInternsById {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SpringApplication.run(RetrieveInternsById.class, args);
		RestTemplate template=new RestTemplate();
		HttpHeaders headers=new HttpHeaders();
		headers.setContentType(MediaType.APPLICATION_JSON);

		HttpEntity<String> requestEntity=new HttpEntity<String>(headers);
		String url="http://localhost:8089/interns-management/yash-
interns/{internId}";
		try(Scanner scanner=new Scanner(System.in);){

			System.out.print("Please Enter Intern Id:");
			int internId=scanner.nextInt();

			ResponseEntity<Interns> response=

	template.exchange(url,HttpMethod.GET,requestEntity,Interns.class,internId);
			Interns interns=response.getBody();
		  System.out.println(interns);
		}catch(Exception e) {
			e.printStackTrace();
		}



	}

}
```

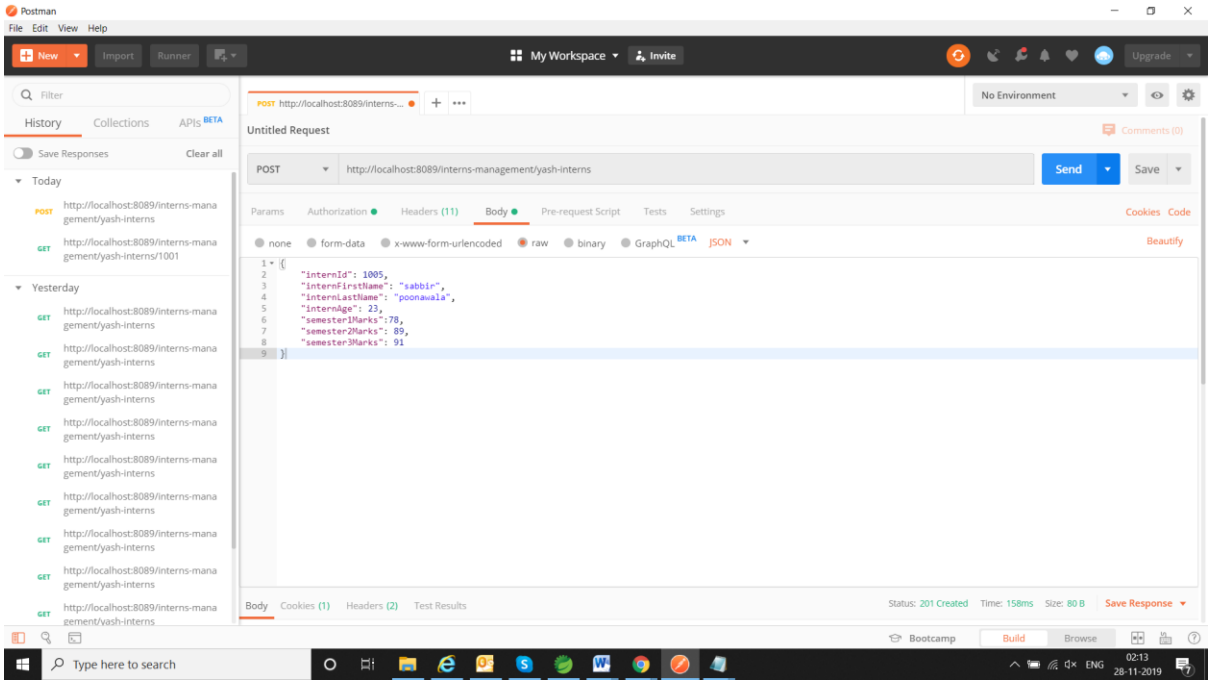
Execute this class, it prompts for Intern id and gives below output,



For Intern registration service,
HttpStatus.CREATED is Http Response status code 201
HttpStatus.CONFLICT is HTTP Response status code 409
@RequestBody annotation indicates a method parameter should be bound to the body of the web request.

```
@PostMapping("yash-interns")
	public ResponseEntity<Void> registerIntern(@RequestBody Interns interns){
		boolean registrationResult=internService.registerInternService(interns);
		ResponseEntity<Void> response=null;
		if(registrationResult){
			response=new ResponseEntity<Void>(HttpStatus.CREATED);
		}else{
			response=new ResponseEntity<Void>(HttpStatus.CONFLICT);
		}
		return response;
	}
```

To Test this method in Postman,

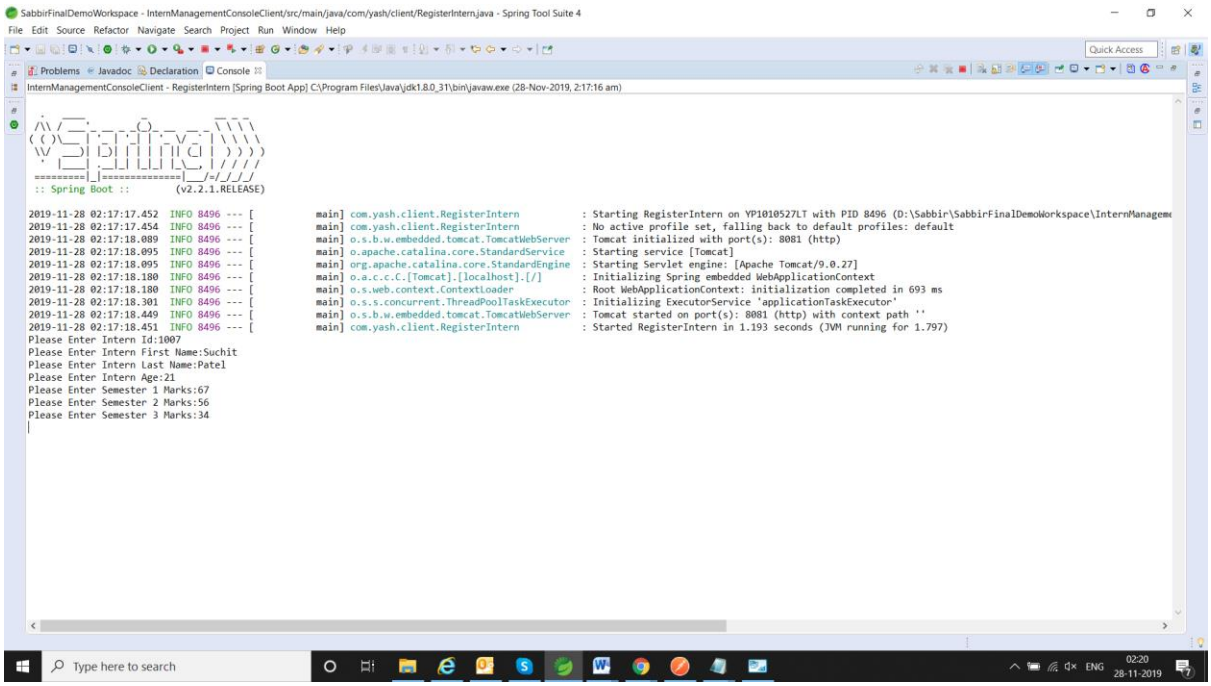


To Test using RestTemplate,
In InternManagementConsoleClient create a class RegisterIntern,

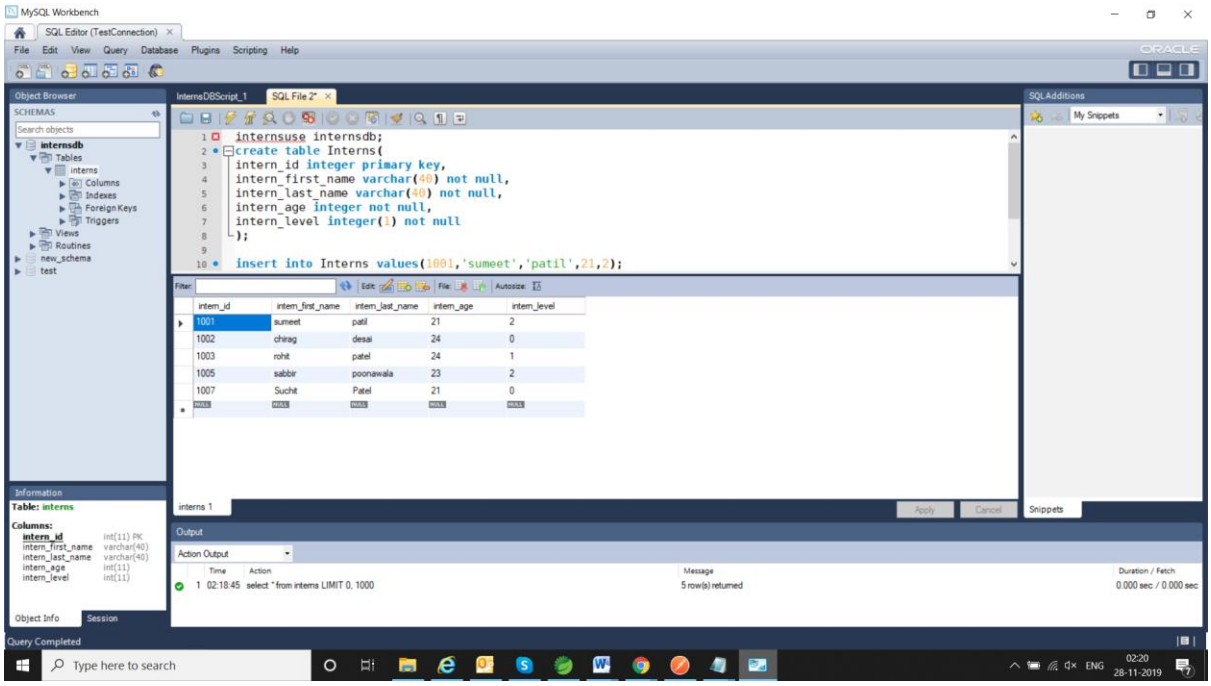
```
package com.yash.client;

import java.util.Scanner;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.web.client.RestTemplate;

import com.yash.model.Interns;

@SpringBootApplication
@EnableAutoConfiguration
public class RegisterIntern {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SpringApplication.run(RegisterIntern.class, args);
		Interns intern=new Interns();
		try(
		Scanner scanner=new Scanner(System.in);
		){
			System.out.print("Please Enter Intern Id:");
			int internId=scanner.nextInt();
			System.out.print("Please Enter Intern First Name:");
			String internFirstName=scanner.next();
```

```
                System.out.print("Please Enter Intern Last Name:");
                String internLastName=scanner.next();
                System.out.print("Please Enter Intern Age:");
                int internAge=scanner.nextInt();
                System.out.print("Please Enter Semester 1 Marks:");
                int semesterMarks1=scanner.nextInt();

                System.out.print("Please Enter Semester 2 Marks:");
                int semesterMarks2=scanner.nextInt();

                System.out.print("Please Enter Semester 3 Marks:");
                int semesterMarks3=scanner.nextInt();

                intern.setInternId(internId);
                intern.setInternFirstName(internFirstName);
                intern.setInternLastName(internLastName);
                intern.setInternAge(internAge);
                intern.setSemester1Marks(semesterMarks1);
                intern.setSemester2Marks(semesterMarks2);
                intern.setSemester3Marks(semesterMarks3);
            }
            catch(Exception e){
                e.printStackTrace();
            }

            RestTemplate template=new RestTemplate();
            HttpHeaders headers=new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);
            String url="http://localhost:8089/interns-management/yash-interns";
            template.postForObject(url, intern, Interns.class);
        }

}
```

Run this class and enter inputs on console



Verify if record is successfully inserted in database



For Update of level of Intern, We have method updateInternLevel() in InternsController

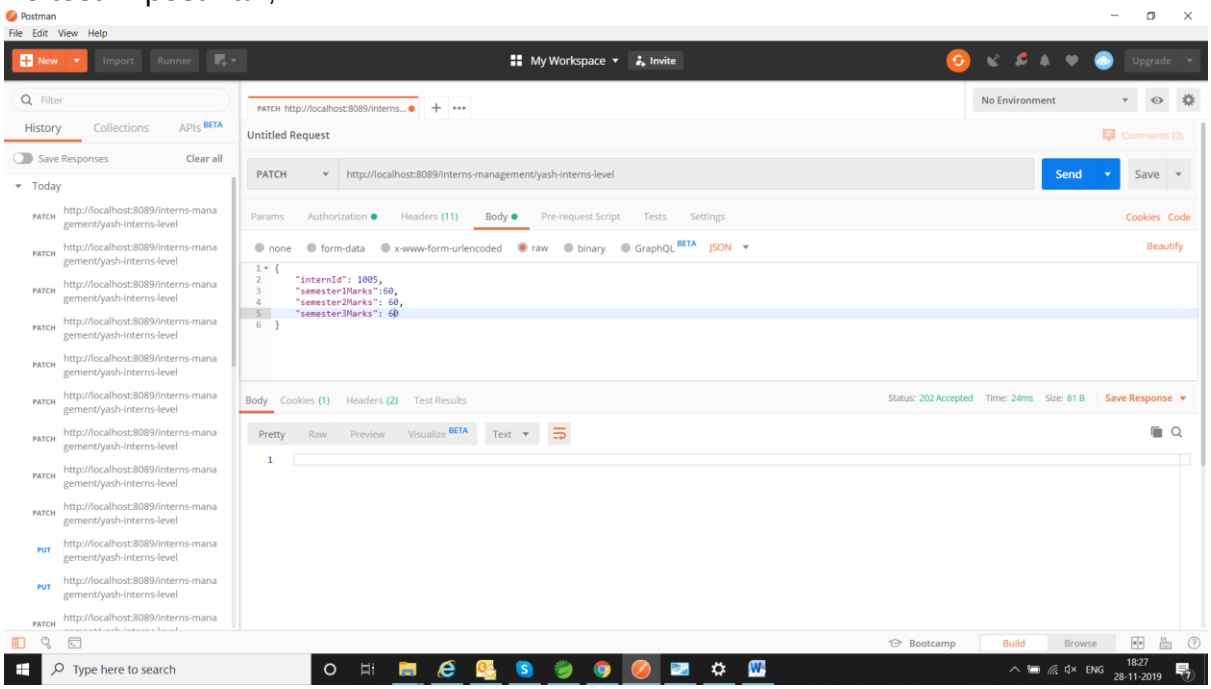
```
@PatchMapping("yash-interns-level")
	public ResponseEntity<Void> updateInternLevel(@RequestBody Interns interns)
	{
		boolean updateLevel=internService.updateInternLevelService(interns);
		ResponseEntity<Void> response=null;
		if(updateLevel==true)
			response=new ResponseEntity<Void>(HttpStatus.ACCEPTED);
		else
			response=new ResponseEntity<Void>(HttpStatus.NOT_MODIFIED);
		return response;
	}
```

Difference between PUT and PATCH Requests

PUT and PATCH are HTTP verbs and they both relate to updating a resource.

In a PUT request, the enclosed entity is considered to be a modified version of the resource stored on the origin server, and the client is requesting that the stored version be replaced.
With PATCH, however, the enclosed entity contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version.
Also, another difference is that when you want to update a resource with PUT request, you have to send the full payload as the request whereas with PATCH, you only send the parameters which you want to update.
When we send a PATCH request, however, we only send the data which we want to update. In other words, we only send the level to update, no need to send the first name or last name.
For this reason, PATCH request requires less bandwidth.

To test in postman,



To test using RestTemplate,

In InternsManagementConsoleClient application, create UpdateInternsLevel in com.yash.client package.

```
package com.yash.client;

import java.util.Scanner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.web.client.RestTemplate;
```

```
import com.yash.model.Interns;

@SpringBootApplication
@EnableAutoConfiguration
public class UpdateInternsLevel {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SpringApplication.run(RegisterIntern.class, args);
		Interns intern=new Interns();
		try(
		Scanner scanner=new Scanner(System.in);
		){
			System.out.print("Please Enter Intern Id:");
			int internId=scanner.nextInt();

			System.out.print("Please Enter Semester 1 Marks:");
			int semesterMarks1=scanner.nextInt();

			System.out.print("Please Enter Semester 2 Marks:");
			int semesterMarks2=scanner.nextInt();

			System.out.print("Please Enter Semester 3 Marks:");
			int semesterMarks3=scanner.nextInt();

			intern.setInternId(internId);
			intern.setSemester1Marks(semesterMarks1);
			intern.setSemester2Marks(semesterMarks2);
			intern.setSemester3Marks(semesterMarks3);
		}
		catch(Exception e){
			e.printStackTrace();
		}
		RestTemplate template=new RestTemplate();
		HttpHeaders headers=new HttpHeaders();
		headers.setContentType(MediaType.APPLICATION_JSON);
		String url="http://localhost:8089/interns-management/yash-interns-
level";
		template.patchForObject(url,intern, Interns.class);

	}

}
```

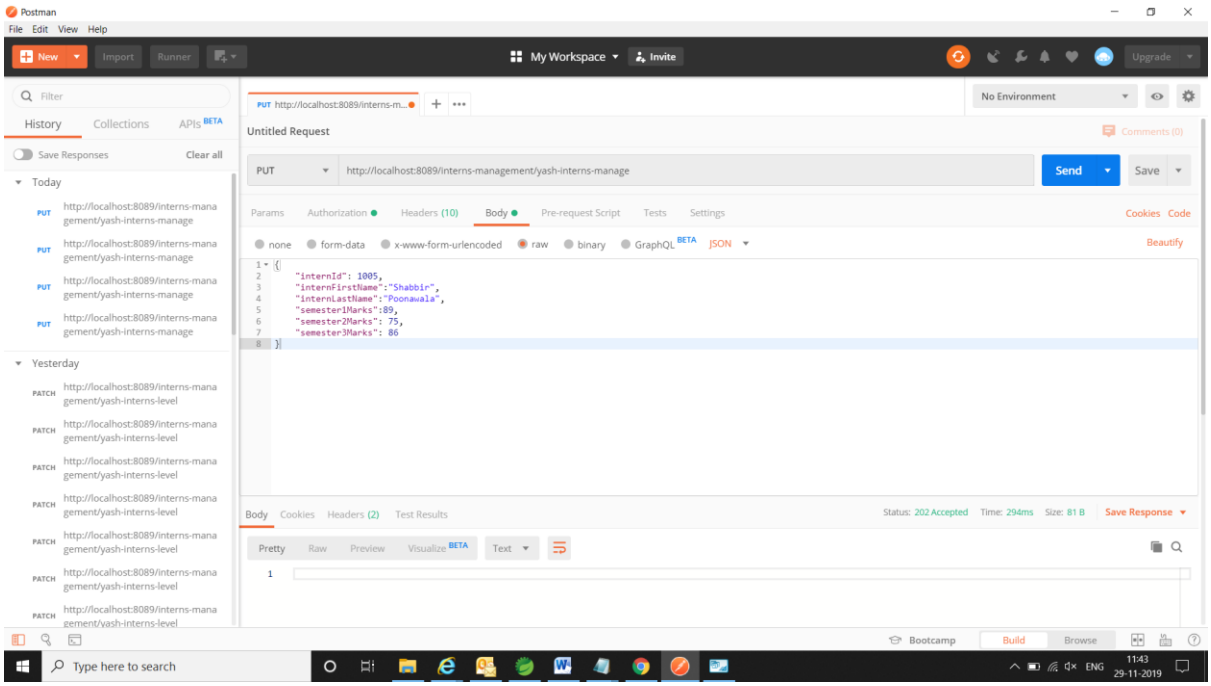
To update Intern details

In InternsController handler method is,

```
	@PutMapping("yash-interns-manage")
	public ResponseEntity<Void> updateIntern(@RequestBody Interns interns)
	{
		boolean updateIntern=internService.updateInternService(interns);
		ResponseEntity<Void> response=null;
```

```
            if(updateIntern==true)
                  response=new ResponseEntity<Void>(HttpStatus.ACCEPTED);
            else
                  response=new ResponseEntity<Void>(HttpStatus.NOT_MODIFIED);
            return response;
      }
```

To test above method in Postman,



As discussed earlier, in case of put, payload must include all parameters i.e intern id,firstname,lastname,semester1marks,semester2marks,semester3marks.

To update only one of field of Intern, You should use patch.

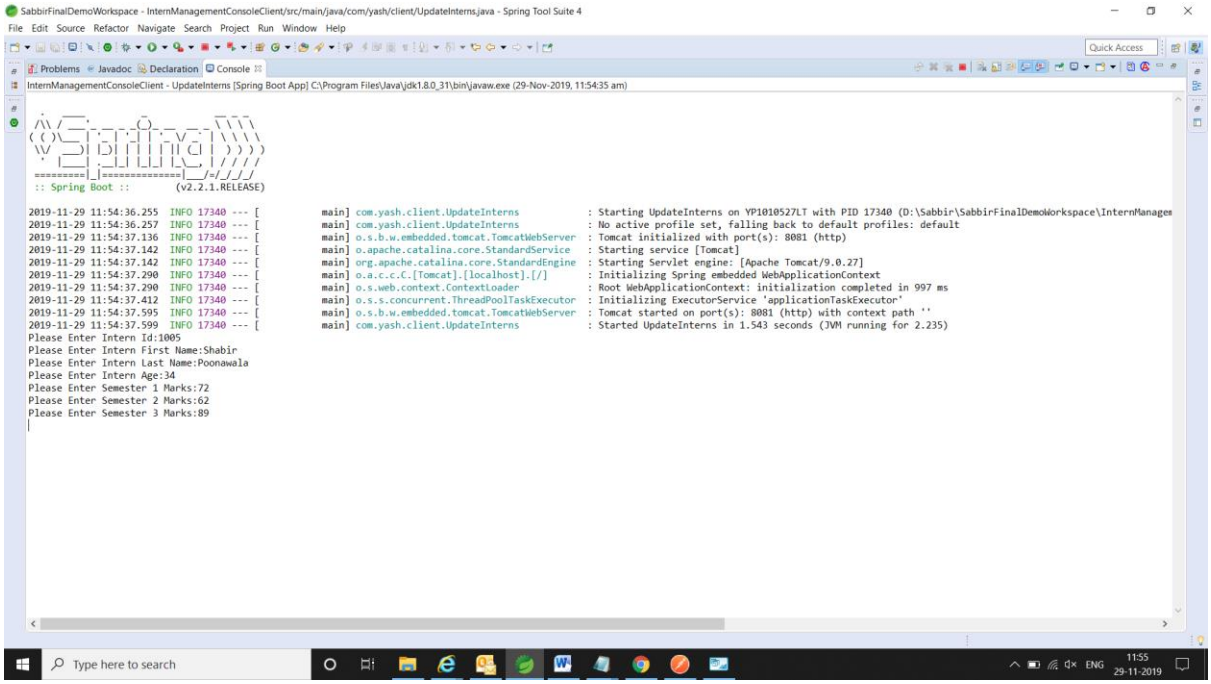
To test above method using RestTemplate,

Create a class UpdateInterns

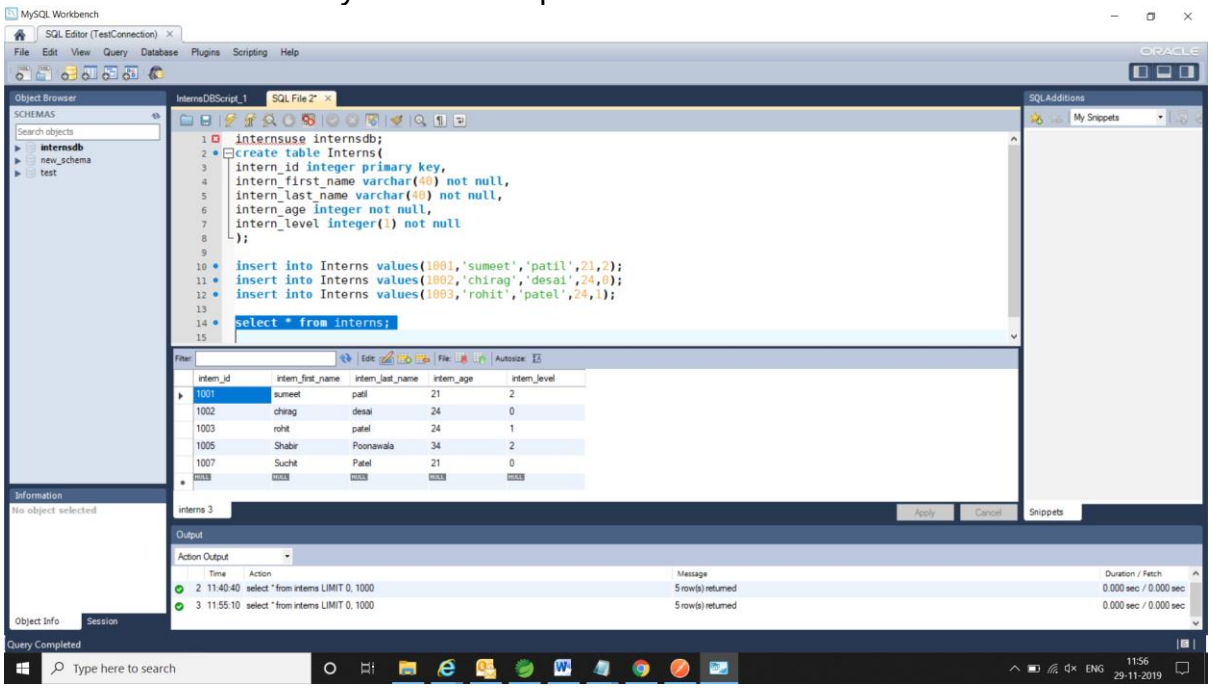
```
package com.yash.client;

import java.util.Scanner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.web.client.RestTemplate;
import com.yash.model.Interns;

@SpringBootApplication
@EnableAutoConfiguration
```

```
public class UpdateInterns {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SpringApplication.run(RegisterIntern.class, args);
		Interns intern=new Interns();
		try(
		Scanner scanner=new Scanner(System.in);
		){
			System.out.print("Please Enter Intern Id:");
			int internId=scanner.nextInt();
			System.out.print("Please Enter Intern First Name:");
			String internFirstName=scanner.next();
			System.out.print("Please Enter Intern Last Name:");
			String internLastName=scanner.next();
			System.out.print("Please Enter Intern Age:");
			int internAge=scanner.nextInt();
			System.out.print("Please Enter Semester 1 Marks:");
			int semesterMarks1=scanner.nextInt();

			System.out.print("Please Enter Semester 2 Marks:");
			int semesterMarks2=scanner.nextInt();

			System.out.print("Please Enter Semester 3 Marks:");
			int semesterMarks3=scanner.nextInt();

			intern.setInternId(internId);
			intern.setInternFirstName(internFirstName);
			intern.setInternLastName(internLastName);
			intern.setInternAge(internAge);
			intern.setSemester1Marks(semesterMarks1);
			intern.setSemester2Marks(semesterMarks2);
			intern.setSemester3Marks(semesterMarks3);
		}
		catch(Exception e){
			e.printStackTrace();
		}
		RestTemplate template=new RestTemplate();
		HttpHeaders headers=new HttpHeaders();
		headers.setContentType(MediaType.APPLICATION_JSON);
		String url="http://localhost:8089/interns/updateLevel";
		template.put(url, intern);

	}

}
```

Run above class and give inputs on console,



Check database to verify if record is updated.

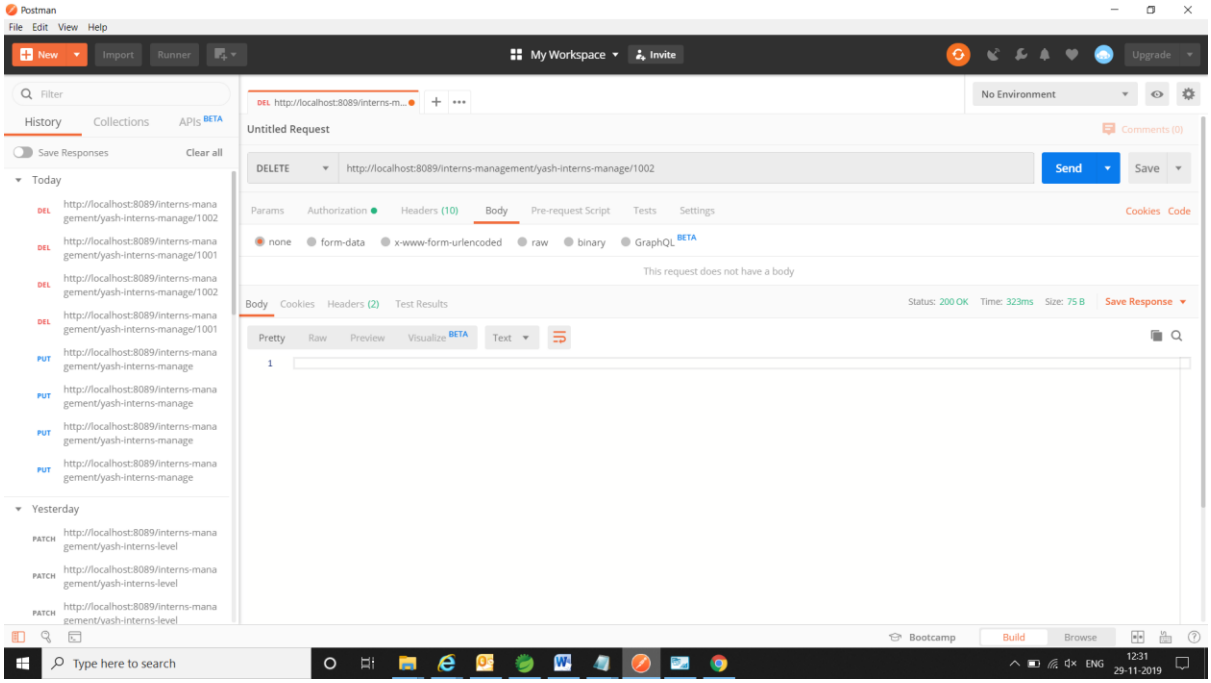


For removing interns service,
In InternsController we have below method,
@DeleteMapping annotation is for mapping HTTP DELETE requests onto specific handler methods. Intern details will be removed based on internId specified in URI.

```
@DeleteMapping("yash-interns-manage/{internId}")
```

```
	public ResponseEntity<Void> deleteStudent(@PathVariable("internId")int
internId){
		ResponseEntity<Void> response=null;
	    boolean internRemoved=internService.removeInternService(internId);
	    if(internRemoved){
				response=new ResponseEntity<Void>(HttpStatus.OK);
			}else{
				response=new ResponseEntity<Void>(HttpStatus.NOT_FOUND);

			}
			return response;
		}
```

To test this method in PostMan,

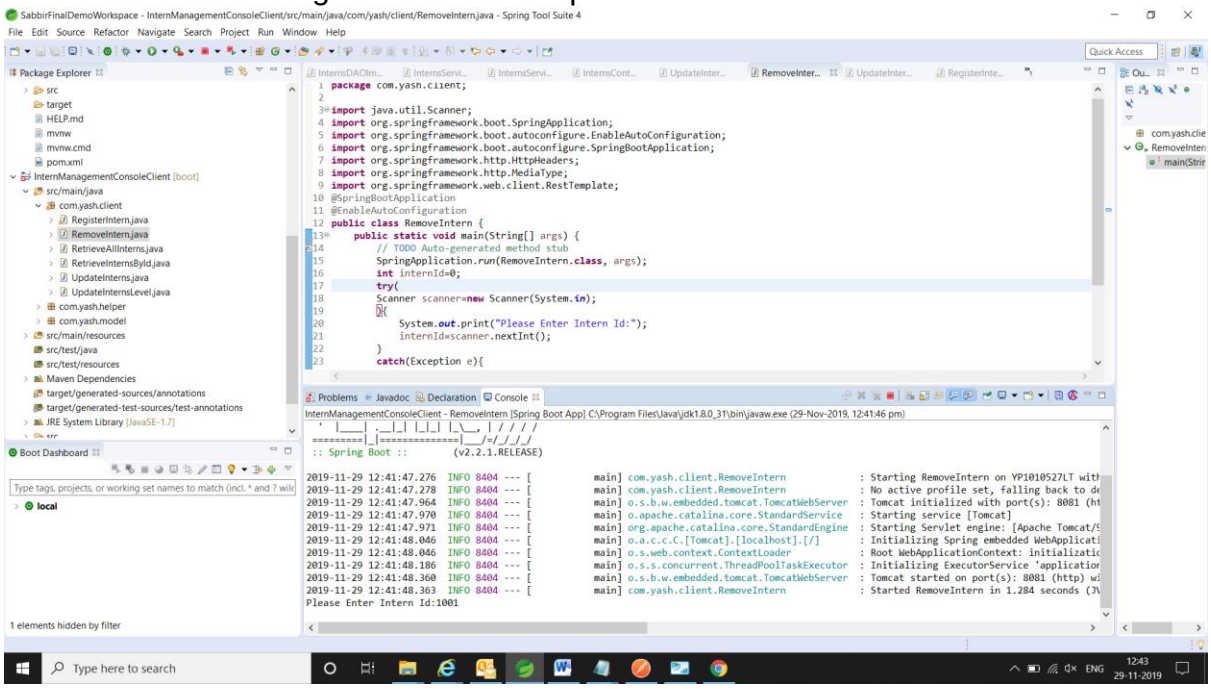


To test above method using RestTemplate,
Create a class RemoveIntern

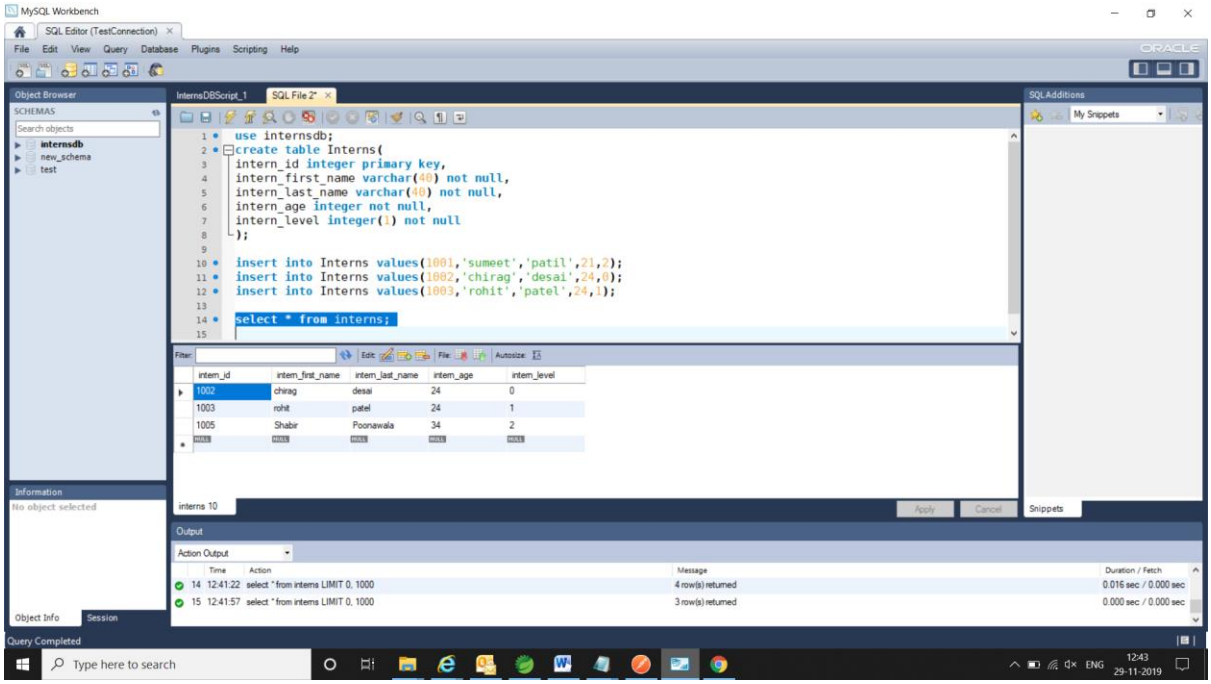
```
package com.yash.client;

import java.util.Scanner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.web.client.RestTemplate;
@SpringBootApplication
@EnableAutoConfiguration
public class RemoveIntern {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SpringApplication.run(RemoveIntern.class, args);
		int internId=0;
		try(
		Scanner scanner=new Scanner(System.in);
```

```
            ){
                  System.out.print("Please Enter Intern Id:");
                  internId=scanner.nextInt();
            }
            catch(Exception e){
                  e.printStackTrace();
            }
            RestTemplate template=new RestTemplate();
            HttpHeaders headers=new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);
            String url="http://localhost:8089/interns-management/yash-interns-
manage/{internId}";
            template.delete(url, internId);
        }
    }
```

Run above class and give intern id as input.



Verify in database,



Spring boot pagination and sorting

Pagination in web applications is a mechanism to separate a big result set into smaller chunks.

Paging and sorting is mostly required when we are displaying domain data in tabular format in UI. Assuming that many records are there in interns table, we would like to display only 10 records at a time in one page.

Page

The findAll(Pageable pageable) method by default returns a Page object.
A Page object provides lots of extra useful information other than just list of employees in current page.

E.g. A Page object has the number of total pages, number of the current page and well as whether current page is first page or last page.

Finding total pages invokes an additional count() query causing an extra overhead cost. Be sure when you are using it.

Slice

Slice is very much similar to Page, except it does not provide the number of total pages in database. It helps in improving performance when we do not need to display total number pages in UI.

In this example we will utilise Spring Data.

Create a new business entity called Intern in com.yash.entity package. We could have utilised Interns we created in previous example. But Spring data requires a field “id” in entity class. Since we had specified internId in Interns We will have to refactor at multiple locations in code. To avoid that we will create new business entity Intern.

```
package com.yash.entity;
import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Transient;

import org.hibernate.annotations.DynamicUpdate;

import com.yash.helper.Levels;

@Entity
@DynamicUpdate
@Table(name="Interns")
/*
 * Business Entity represents database table Interns
 */
public class Intern implements Serializable {
	@Id
	@Column(name="intern_id")
	private int id;
	@Column(name="intern_first_name")
	private String internFirstName;
	@Column(name="intern_last_name")
	private String internLastName;
	@Column(name="intern_age")
	private int internAge;
	@Column(name="intern_level")
	private Levels internLevel;
	@Transient
	private int semester1Marks;
	@Transient
	private int semester2Marks;
	@Transient
	private int semester3Marks;

	public Levels getInternLevel() {
```

```
            return internLevel;
      }

      public void setInternLevel(Levels internLevel) {
            this.internLevel = internLevel;
      }

      public int getSemester1Marks() {
            return semester1Marks;
      }

      public void setSemester1Marks(int semester1Marks) {
            this.semester1Marks = semester1Marks;
      }

      public int getSemester2Marks() {
            return semester2Marks;
      }

      public void setSemester2Marks(int semester2Marks) {
            this.semester2Marks = semester2Marks;
      }

      public int getSemester3Marks() {
            return semester3Marks;
      }

      public void setSemester3Marks(int semester3Marks) {
            this.semester3Marks = semester3Marks;
      }

      public int getInternAge() {
            return internAge;
      }

      public void setInternAge(int internAge) {
            this.internAge = internAge;
      }

      public int getInternId() {
            return id;
      }

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

      public String getInternFirstName() {
            return internFirstName;
      }

      public void setInternFirstName(String internFirstName) {
            this.internFirstName = internFirstName;
      }

      public String getInternLastName() {
            return internLastName;
      }
```

```
	public void setInternLastName(String internLastName) {
		this.internLastName = internLastName;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + id;
		result = prime * result + internAge;
		result = prime * result + ((internFirstName == null) ? 0 :
internFirstName.hashCode());
		result = prime * result + ((internLastName == null) ? 0 :
internLastName.hashCode());
		result = prime * result + ((internLevel == null) ? 0 :
internLevel.hashCode());
		result = prime * result + semester1Marks;
		result = prime * result + semester2Marks;
		result = prime * result + semester3Marks;
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Intern other = (Intern) obj;
		if (id != other.id)
			return false;
		if (internAge != other.internAge)
			return false;
		if (internFirstName == null) {
			if (other.internFirstName != null)
				return false;
		} else if (!internFirstName.equals(other.internFirstName))
			return false;
		if (internLastName == null) {
			if (other.internLastName != null)
				return false;
		} else if (!internLastName.equals(other.internLastName))
			return false;
		if (internLevel != other.internLevel)
			return false;
		if (semester1Marks != other.semester1Marks)
			return false;
		if (semester2Marks != other.semester2Marks)
			return false;
		if (semester3Marks != other.semester3Marks)
			return false;
		return true;
	}

	@Override
	public String toString() {
```

```
            return "Intern [id=" + id + ", internFirstName=" + internFirstName +
", internLastName=" + internLastName
                        + ", internAge=" + internAge + ", internLevel=" +
internLevel + ", semester1Marks=" + semester1Marks
                        + ", semester2Marks=" + semester2Marks + ",
semester3Marks=" + semester3Marks + "]";
      }

}
```

Create an interface InternsRepository in com.yash.dao package and annotate interface as Repository.

CrudRepository is an interface and extends Spring data Repository interface. CrudRepository provides generic CRUD operation on a repository for a specific type. It has generic methods for CRUD operation.
By having it extend *PagingAndSortingRepository*, we get *findAll(Pageable pageable)* and *findAll(Sort sort)* methods for paging and sorting.

```
package com.yash.dao;

import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.stereotype.Repository;
import com.yash.entity.Intern;
@Repository
public interface InternsRepository extends PagingAndSortingRepository<Intern,
Long> {

}
```

Declare new method getAllIntern() in InternService interface,

```
package com.yash.service;
import java.util.List;

import com.yash.entity.Intern;
import com.yash.entity.Interns;
public interface InternsService {
      List<Interns> retrieveInternsService();
      Interns retrieveInternsByIdService(int internId);
      boolean registerInternService(Interns interns);
      boolean updateInternService(Interns interns);
      boolean updateInternLevelService(Interns interns);
      boolean removeInternService(int internId);
      List<Intern> getAllIntern(Integer pageNo, Integer pageSize, String sortBy);

}
```

Inject InternsRepository in InternsServiceImpl

```
@Autowired
      private InternsRepository internsRepository;
```

and provide implementation of getAllIntern() as below,

PageRequest implements interface Pageable which contains pagination information.If the client only wants to display a “slice” of a list of items, it needs to provide some input parameters that describe this slice. In Spring Data, these parameters are bundled within the Pageable interface. It provides the following methods,

int getPageNumber(); // number of the current page

int getPageSize(); // size of the pages

Sort getSort(); // sorting parameters

Whenever we want to load only a slice of a full list of items, we can use a Pageable instance as an input parameter, as it provides the number of the page to load as well as the size of the pages. Through the Sort class, it also allows to define fields to sort by and the direction in which they should be sorted (ascending or descending).
The most common way to create a Pageable instance is to use the PageRequest implementation

```
Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by(sortBy));
```

```
@Override
      public List<Intern> getAllIntern(Integer pageNo, Integer pageSize, String
sortBy) {
            // TODO Auto-generated method stub
            Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by(sortBy));

        Page<Intern> pagedResult = internsRepository.findAll(paging);

        if(pagedResult.hasContent()) {
            return pagedResult.getContent();
        } else {
            return new ArrayList<Intern>();
        }
      }
```

Pagination and sorting techniques

Paging WITHOUT sorting

To apply only pagination in result set, we shall create Pageable object without any Sort information.

Pageable paging = PageRequest.of(pageNo, pageSize);

Page<Intern> pagedResult = repository.findAll(paging);

Paging WITH sorting

To apply only pagination in result set, we shall create `Pageable` object with desired `Sort` column name.

```
Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by("internFirstName"));

Page<Intern> pagedResult = repository.findAll(paging);
```

By default, records are ordered in **DESCENDING** order. To choose **ASCENDING** order, use `.ascending()` method.

```
Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by("internFirstName ").ascending());

Page<Intern> pagedResult = repository.findAll(paging);
```

Sorting only

If there is no need to page, and only sorting is required, we can create Sort object for that.

```
Sort sortOrder = Sort.by("internFirstName");

List<Intern> list = repository.findAll(sortOrder);
```

If we wish to apply sorting on multiple columns or group by sort, then that is also possible by creating Sort using simple builder pattern steps.

```
Sort firstNameSort = Sort.by("internFirstName");
Sort lastNameSort = Sort.by("internLastName");

Sort groupBySort = firstNameSort.and(lastNameSort);

List<Intern> list = repository.findAll(groupBySort);
```

Difference between Page and Slice

Page

The findAll(Pageable pageable) method by default returns a Page object.
A Page object provides lots of extra useful information other than just list of employees in current page.

E.g. A Page object has the number of total pages, number of the current page and well as whether current page is first page or last page.

Finding total pages invokes an additional count() query causing an extra overhead cost. Be sure when you are using it.

Slice

Slice is very much similar to Page, except it does not provide the number of total pages in database. It helps in improving performance when we do not need to display total number pages in UI.

Generally, Slice is used in case navigation is consist of Next page and Previous page links.

To use Slice, we have to implement our own custom methods.

```
package com.yash.dao;

import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Slice;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import com.yash.entity.Intern;
@Repository
public interface InternsRepositorySlice extends CrudRepository<Intern, Long> {
	public Slice<Intern> findByInternFirstName(String internFirstName, Pageable
pageable);

}
```

Remember that is we use PagingAndSortingRepository, default return type is Page.

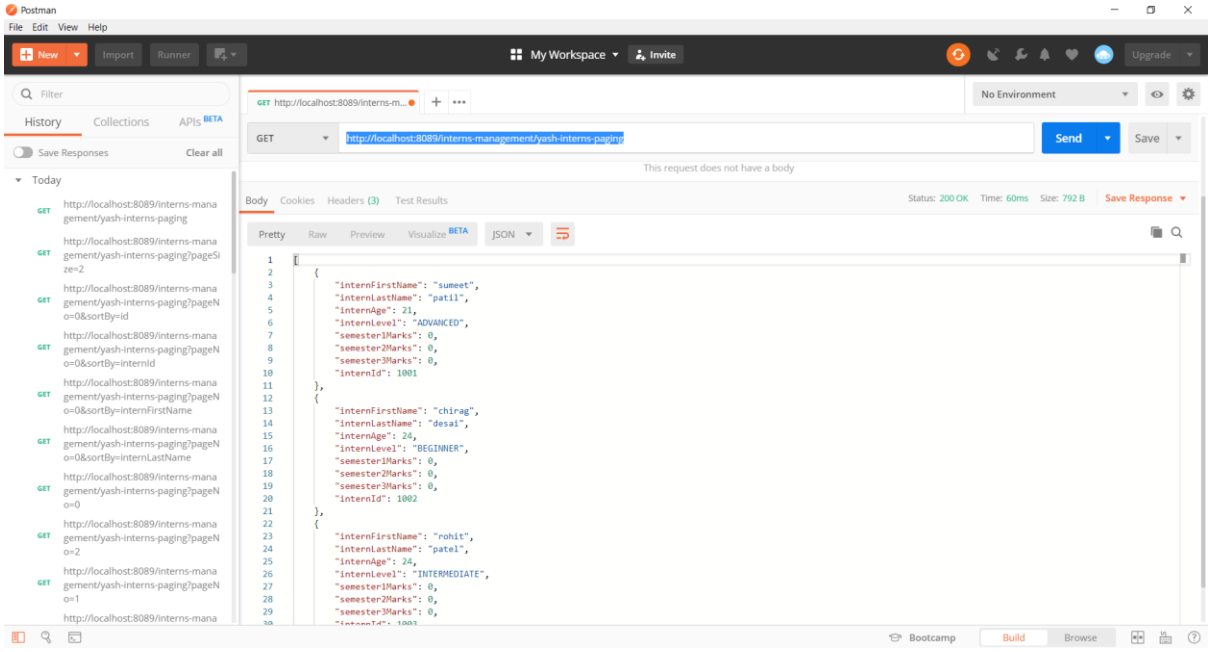
```
@Override
	public List<Intern> getAllInternSlice(Integer pageNo, Integer pageSize,
String sortBy) {
		// TODO Auto-generated method stub
		Pageable paging = PageRequest.of(pageNo, pageSize,
Sort.by("internFirstName").descending());
		Slice<Intern> slicedResult =
internsRepositorySlice.findByInternFirstName("sabbir", paging);
		List<Intern> internList = slicedResult.getContent();
		return internList;

	}
```

In InternsController We will add new method with pagination implementation

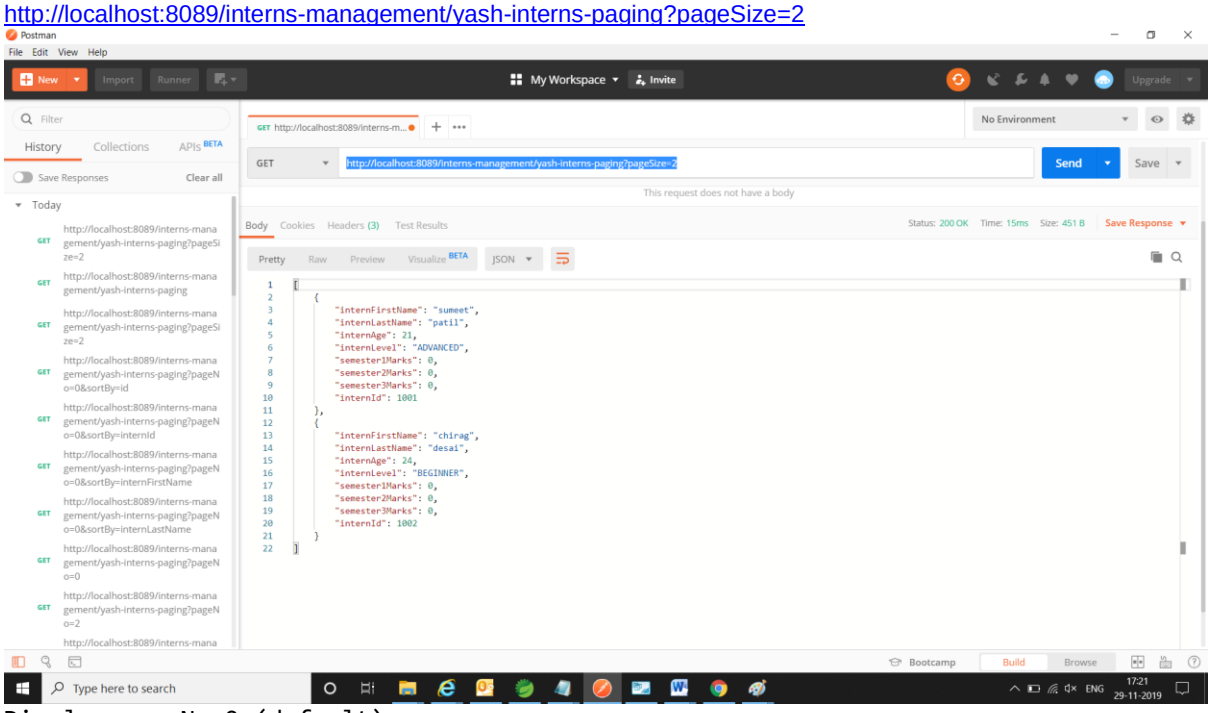
```
@GetMapping("yash-interns-paging")
    public ResponseEntity<List<Intern>> getAllInterns(
                        @RequestParam(defaultValue = "0") Integer pageNo,
                        @RequestParam(defaultValue = "10") Integer pageSize,
                        @RequestParam(defaultValue = "id") String sortBy)
    {
        List<Intern> list = internService.getAllIntern(pageNo, pageSize, sortBy);
        return new ResponseEntity<List<Intern>>(list, new HttpHeaders(),
HttpStatus.OK);
    }
```

Test service in postman,
Default page number is 0, page size is 10 and default sort column is ‘id’

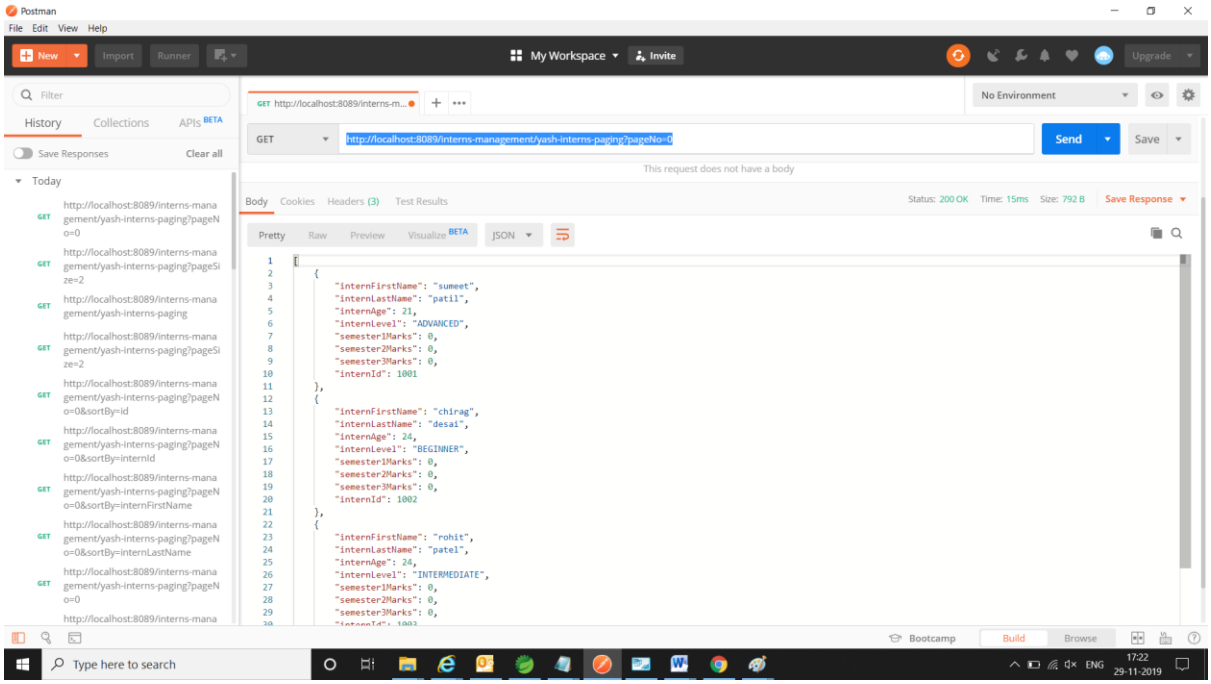
http://localhost:8089/interns-management/yash-interns-paging



Display only two records.
http://localhost:8089/interns-management/yash-interns-paging?pageSize=2

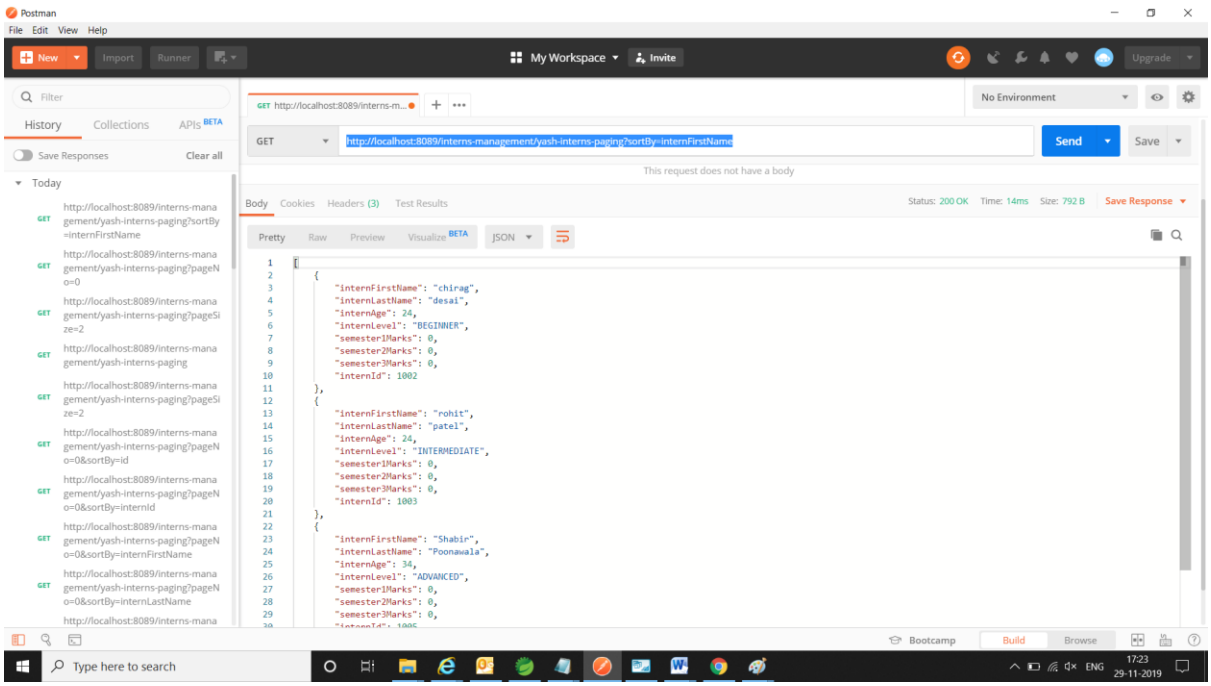


`Display pageNo=0 (default)`
http://localhost:8089/interns-management/yash-interns-paging?pageNo=0

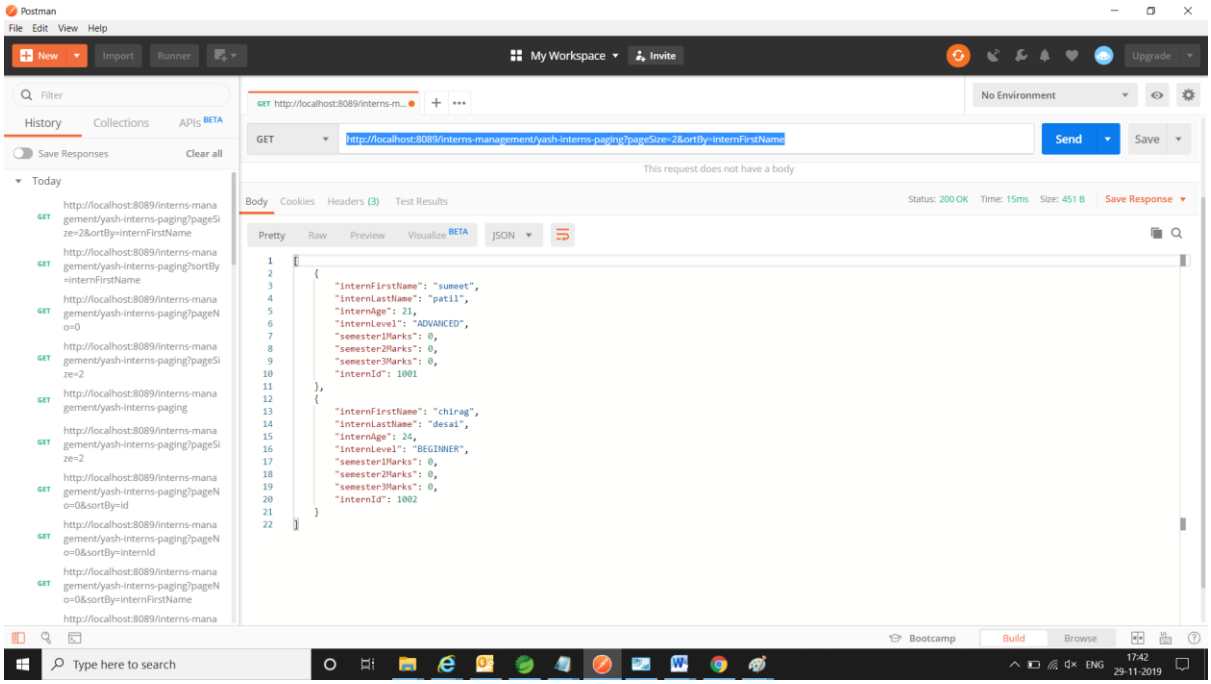


Since we have fewer records in table at moment we will not be able to see pageNo=1,2 etc

http://localhost:8089/interns-management/yash-interns-paging?sortBy=internFirstName



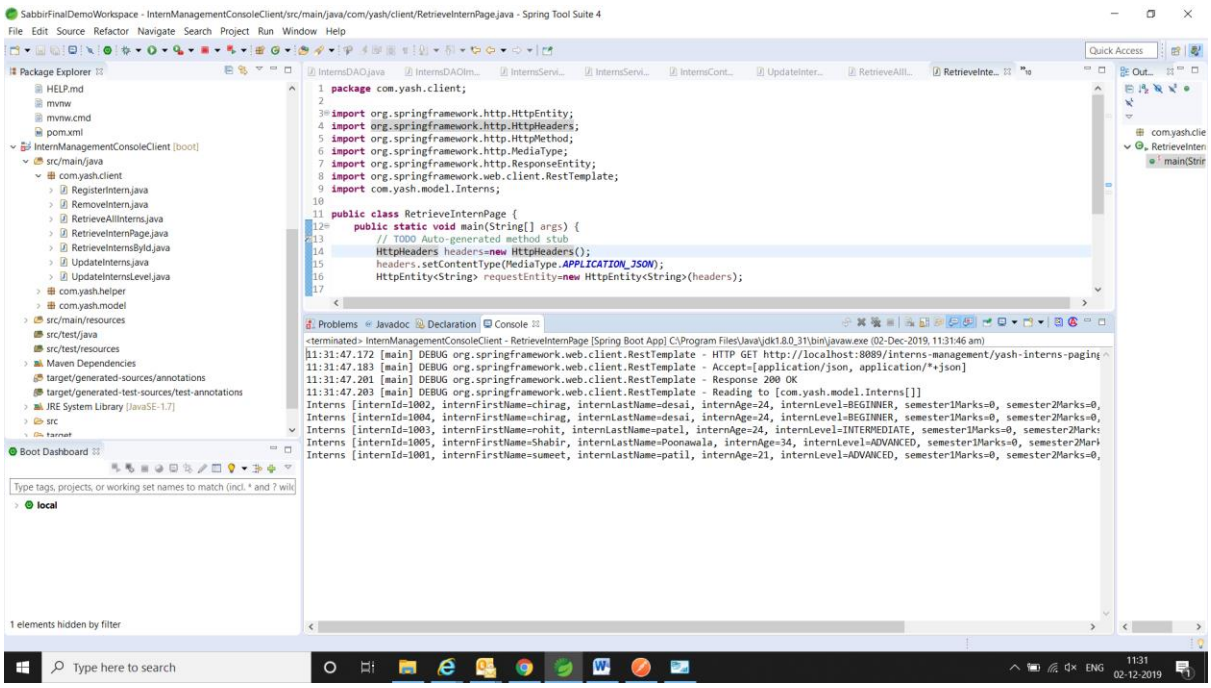
http://localhost:8089/interns-management/yash-interns-paging?pageSize=2&ortBy=internFirstName



To test pagination using RestTemplate, Create a class RetrieveInternPage in InternManagementConsoleClient in com.yash.client package.

```
package com.yash.client;

import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
import com.yash.model.Interns;

public class RetrieveInternPage {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		HttpHeaders headers=new HttpHeaders();
		headers.setContentType(MediaType.APPLICATION_JSON);
		HttpEntity<String> requestEntity=new HttpEntity<String>(headers);
		
		String url="http://localhost:8089/interns-management/yash-interns-
paging?pageNo={pageNo}&pageSize={pageSize}&sortBy={sortBy}";
		RestTemplate template=new RestTemplate();
		ResponseEntity<Interns[]> response=template.exchange(url,
HttpMethod.GET, requestEntity, Interns[].class,0,5,"internFirstName");
	    Interns[] interns=response.getBody();
	    for(Interns intern:interns){
		System.out.println(intern);
	    }
	}
}
```

Observe the output on console,



Slice is a sized chunk of data with an indication of whether there is more data available.

Spring supports Slice as a return type of the query method. Pageable parameter must be specified by the same query method.

Slice avoids triggering a count query to calculate the overall number of pages as that might be expensive. A Slice only knows about whether a next or previous Slice is available, which might be sufficient when walking through a larger result set.

To see implementation of Slice, we will add new method in InternsService and implement method in InternsServiceImpl.

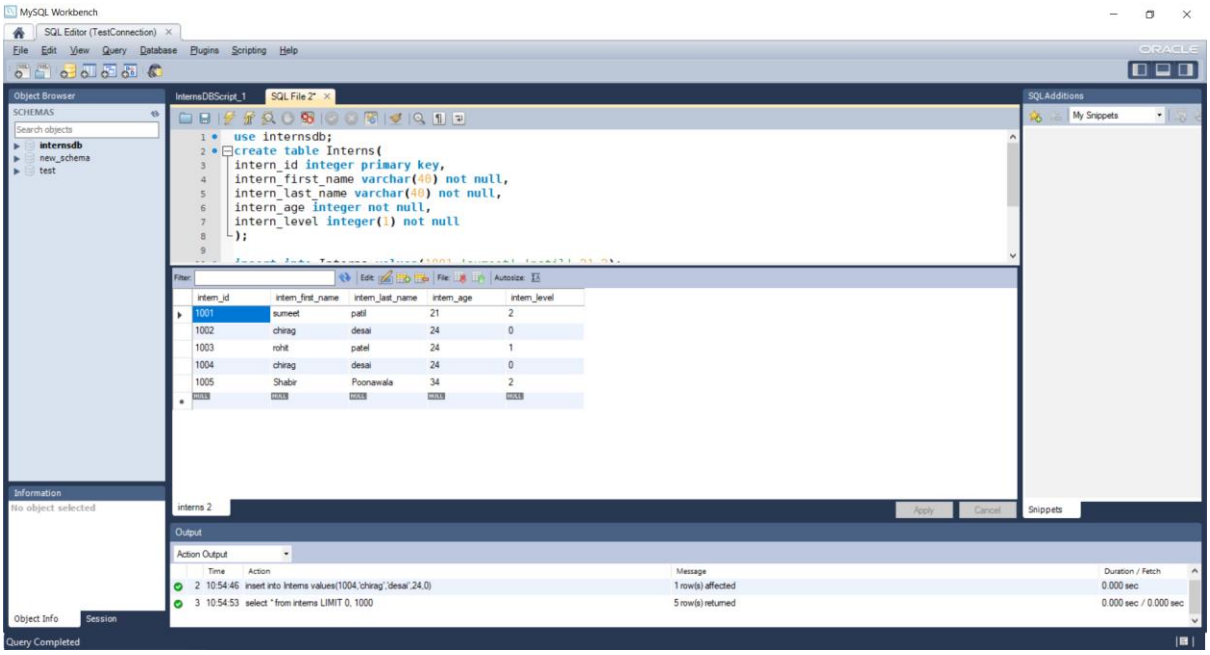
```
List<Intern> getAllInternSlice(Integer pageNo, Integer pageSize, String
sortBy,String internFirstName);
```

```
@Override
	public List<Intern> getAllInternSlice(Integer pageNo, Integer pageSize,
String sortBy,String internFirstName) {
		// TODO Auto-generated method stub
		Pageable paging = PageRequest.of(pageNo, pageSize,
Sort.by(sortBy).descending());
		Slice<Intern> slicedResult =
internsRepositorySlice.findByInternFirstName(internFirstName, paging);
		System.out.println("Slice number:"+slicedResult.getNumber());
	    int numberOfElements = slicedResult.getNumberOfElements();
	    System.out.println("Number of elements:"+numberOfElements);

		List<Intern> internList = slicedResult.getContent();
		return internList;

	}
```

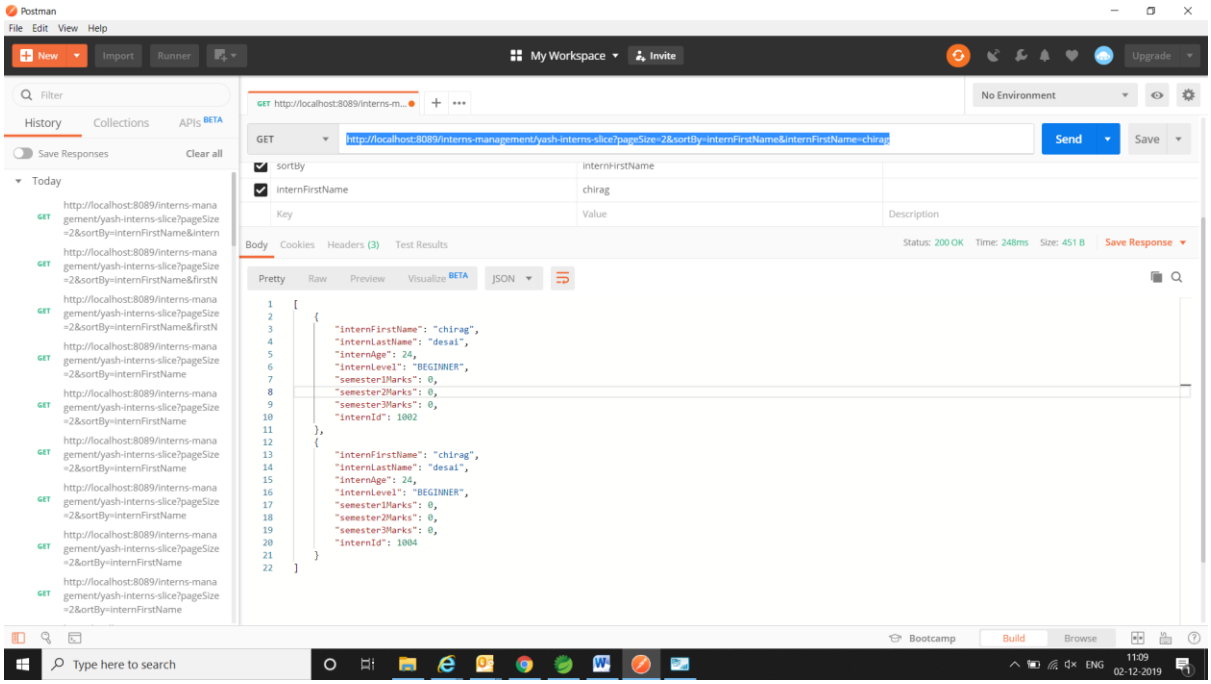
We have put System.out.println() to understand how slice works.
In InternsController,

```
@GetMapping("yash-interns-slice")
    public ResponseEntity<List<Intern>> getAllInternsSlice(
                        @RequestParam(defaultValue = "0") Integer pageNo,
                        @RequestParam(defaultValue = "10") Integer pageSize,
                        @RequestParam(defaultValue = "id") String sortBy,
                        @RequestParam String internFirstName)
    {
        List<Intern> list = internService.getAllInternSlice(pageNo, pageSize,
sortBy,internFirstName);
        return new ResponseEntity<List<Intern>>(list, new HttpHeaders(),
HttpStatus.OK);
    }
```

We have put one more record with same internFirstName to see implementation of Slice.



To test in postman, specify following URI,
http://localhost:8089/interns-management/yash-interns-slice?pageSize=2&sortBy=internFirstName&internFirstName=chirag

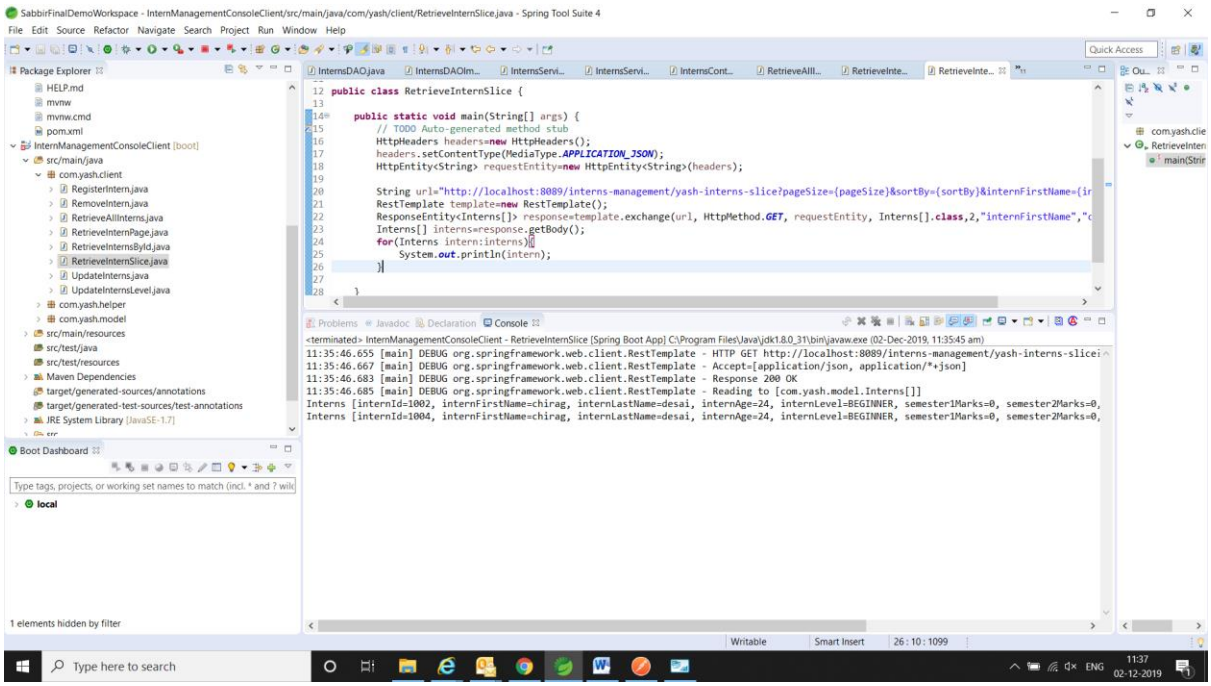


Observe on console of business application println() statements,

Slice number: 0
Number of elements: 2
To test Slice implementation using RestTemplate, create a class RetrieveInternSlice in InternManagementConsoleClient with below code,

```
package com.yash.client;

import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

import com.yash.model.Interns;

public class RetrieveInternSlice {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		HttpHeaders headers=new HttpHeaders();
		headers.setContentType(MediaType.APPLICATION_JSON);
		HttpEntity<String> requestEntity=new HttpEntity<String>(headers);
		
		String url="http://localhost:8089/interns-management/yash-interns-
slice?pageSize={pageSize}&sortBy={sortBy}&internFirstName={internFirstName}";
		RestTemplate template=new RestTemplate();
		ResponseEntity<Interns[]> response=template.exchange(url,
HttpMethod.GET, requestEntity, Interns[].class,2,"internFirstName","chirag");
	  Interns[] interns=response.getBody();
	  for(Interns intern:interns){
		System.out.println(intern);
	  }

	}

}
```

Run this class and observe output.



We need to apply security on above application.

Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.
Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements

We will apply basic authentication.
In InternsBusinessApp, pom.xml mention below dependency,

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

We will apply annotation based security. Create a package com.yash.security
Create a configuration class for security,

```
package com.yash.security;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.config.annotation.authentication.builders.Authenticat
ionManagerBuilder;
```

```
import
org.springframework.security.config.annotation.method.configuration.EnableGlobalMe
thodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfig
urerAdapter;
import org.springframework.security.web.AuthenticationEntryPoint;

@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled=true)
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {
	@Autowired
	private AuthenticationEntryPoint authEntryPoint;
	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http.csrf().disable().authorizeRequests()
				.anyRequest().authenticated()
				.and().httpBasic()
				.authenticationEntryPoint(authEntryPoint);
	}
	@Autowired
	public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
		auth.inMemoryAuthentication()
	  .withUser("recruiter")
		.password("{noop}yash123")
		.roles("JuniorRecruiter")
    .and()
		.withUser("adminrecruiter")
		.password("{noop}yash1234")
		.credentialsExpired(false)
		.accountExpired(false)
		.accountLocked(false)
		.authorities("WRITE_PRIVILEGES", "READ_PRIVILEGES")
		.roles("SeniorRecruiter");
	}
}
```

The @EnableWebSecurity is a marker annotation. It allows Spring to find (it's a @Configuration and, therefore, @Component) and automatically apply the class to the global WebSecurity.

EnableGlobalMethodSecurity provides AOP security on methods, some of annotation it will enable are PreAuthorize PostAuthorize also it has support for JSR-250.

By extending WebSecurityConfigurerAdapter and only a few lines of code we are able to do the following:

- Require the user to be authenticated prior to accessing any URL within our application

- Create a user with the username “user”, password “password”, and role of “ROLE_USER”
- Enables HTTP Basic and Form based authentication
- Spring Security will automatically render a login page and logout success page for you

AuthenticationManagerBuilder builds **AuthenticationManager** using which in-memory, JDBC and LDAP authentication is performed. To perform in-memory authentication **AuthenticationManagerBuilder** provides inMemoryAuthentication() method which returns **InMemoryUserDetailsManagerConfigurer** using which we can add user with the method withUser. This method returns **UserDetailsBuilder** using which we assign password by the method password(). It again returns **UserDetailsBuilder** and add it now role with its method roles. In java configuration we need to extend WebSecurityConfigurerAdapter class and override a method of this class configureGlobal().

A custom **AuthenticationEntryPoint** can be used to set necessary response headers, content-type, and so on before sending the response back to the client.The **org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint** class is a built-in **AuthenticationEntryPoint** implementation, which will get invoked for basic authentication to commence. A custom entry point can be created by implementing the **org.springframework.security.web.AuthenticationEntryPoint** interface.

Create a custom AuthenticationEntryPoint in com.yash.security package,

```
package com.yash.security;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.security.core.AuthenticationException;
import
org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint;
import org.springframework.stereotype.Component;
@Component
public class AuthenticationEntryPoint extends BasicAuthenticationEntryPoint {
      @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
             AuthenticationException authEx)
     throws IOException {
        response.addHeader("WWW-Authenticate", "Basic realm=" +getRealmName());
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        PrintWriter writer = response.getWriter();
        writer.println("HTTP Status 401 - " + authEx.getMessage());
    }

      @Override
    public void afterPropertiesSet() {
        setRealmName("developer");
        super.afterPropertiesSet();
    }
}
```

We will apply security on URI in InternsController,

```
package com.yash.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.yash.entity.Intern;
import com.yash.entity.Interns;
import com.yash.service.InternsService;
@RestController
@RequestMapping("/interns-management")
@EnableAspectJAutoProxy
public class InternsController {
	public InternsController() {
	}
	@Autowired
	private InternsService internService;
	@GetMapping("/yash-interns")
	@PreAuthorize (value="hasRole('JuniorRecruiter') or
hasRole('SeniorRecruiter')")
	public ResponseEntity<List<Interns>> retrieveAllInterns(){
		List<Interns> internsList=internService.retrieveInternsService();
		ResponseEntity<List<Interns>> response=new
ResponseEntity<List<Interns>>(internsList,HttpStatus.OK);
		return response;
	}
	@GetMapping("/yash-interns/{internId}")
	@PreAuthorize (value="hasRole('JuniorRecruiter') or
hasRole('SeniorRecruiter')")
	public ResponseEntity<Interns> retrieveInternById(@PathVariable("internId")
int internId){
		System.out.println("--retrieve intern by id--");
		Interns interns=internService.retrieveInternsByIdService(internId);
		ResponseEntity<Interns> response=null;
		if(interns.getInternId()!=0){
		response=new ResponseEntity<Interns>(interns,HttpStatus.FOUND);
		}else{
		response=new ResponseEntity<Interns>(interns,HttpStatus.NOT_FOUND);
		}
		return response;
	}
	@PostMapping("yash-interns")
	@PreAuthorize ("hasRole('SeniorRecruiter')")
```

```
	public ResponseEntity<Void> registerIntern(@RequestBody Interns interns){
		boolean
registrationResult=internService.registerInternService(interns);
		ResponseEntity<Void> response=null;
		if(registrationResult){
			response=new ResponseEntity<Void>(HttpStatus.CREATED);
		}else{
			response=new ResponseEntity<Void>(HttpStatus.CONFLICT);
		}
		return response;
	}
	@PutMapping("yash-interns-manage")
	@PreAuthorize ("hasRole('SeniorRecruiter')")
	public ResponseEntity<Void> updateIntern(@RequestBody Interns interns)
	{
		boolean updateIntern=internService.updateInternService(interns);
		ResponseEntity<Void> response=null;
		if(updateIntern==true)
			response=new ResponseEntity<Void>(HttpStatus.ACCEPTED);
		else
			response=new ResponseEntity<Void>(HttpStatus.NOT_MODIFIED);
		return response;
	}
	@PatchMapping("yash-interns-level")
	@PreAuthorize ("hasRole('SeniorRecruiter')")
	public ResponseEntity<Void> updateInternLevel(@RequestBody Interns interns)
	{
		boolean updateLevel=internService.updateInternLevelService(interns);
		ResponseEntity<Void> response=null;
		if(updateLevel==true)
			response=new ResponseEntity<Void>(HttpStatus.ACCEPTED);
		else
			response=new ResponseEntity<Void>(HttpStatus.NOT_MODIFIED);
		return response;
	}
	@DeleteMapping("yash-interns-manage/{internId}")
	@PreAuthorize ("hasRole('SeniorRecruiter')")
	public ResponseEntity<Void> deleteStudent(@PathVariable("internId")int
internId){
	ResponseEntity<Void> response=null;
    boolean internRemoved=internService.removeInternService(internId);
    if(internRemoved){
			response=new ResponseEntity<Void>(HttpStatus.OK);
		}else{
			response=new ResponseEntity<Void>(HttpStatus.NOT_FOUND);
		}
		return response;
	}
    @GetMapping("yash-interns-paging")
    @PreAuthorize (value="hasRole('JuniorRecruiter') or
hasRole('SeniorRecruiter')")
    public ResponseEntity<List<Intern>> getAllInternsPage(
                        @RequestParam(defaultValue = "0") Integer pageNo,
                        @RequestParam(defaultValue = "10") Integer pageSize,
                        @RequestParam(defaultValue = "id") String sortBy)
    {
        List<Intern> list = internService.getAllInternPage(pageNo, pageSize,
sortBy);
```

```
        return new ResponseEntity<List<Intern>>(list, new HttpHeaders(),
HttpStatus.OK);
    }
    @GetMapping("yash-interns-slice")
    @PreAuthorize (value="hasRole('JuniorRecruiter') or
hasRole('SeniorRecruiter')")
    public ResponseEntity<List<Intern>> getAllInternsSlice(
                        @RequestParam(defaultValue = "0") Integer pageNo,
                        @RequestParam(defaultValue = "10") Integer pageSize,
                        @RequestParam(defaultValue = "id") String sortBy,
                        @RequestParam String internFirstName)
    {
        List<Intern> list = internService.getAllInternSlice(pageNo, pageSize,
sortBy,internFirstName);
        return new ResponseEntity<List<Intern>>(list, new HttpHeaders(),
HttpStatus.OK);
    }
}
```

@EnableAspectJAutoProxy enables support for handling components marked with AspectJ's @Aspect annotation, similar to functionality found in Spring's <aop:aspectj-autoproxy> XML element.

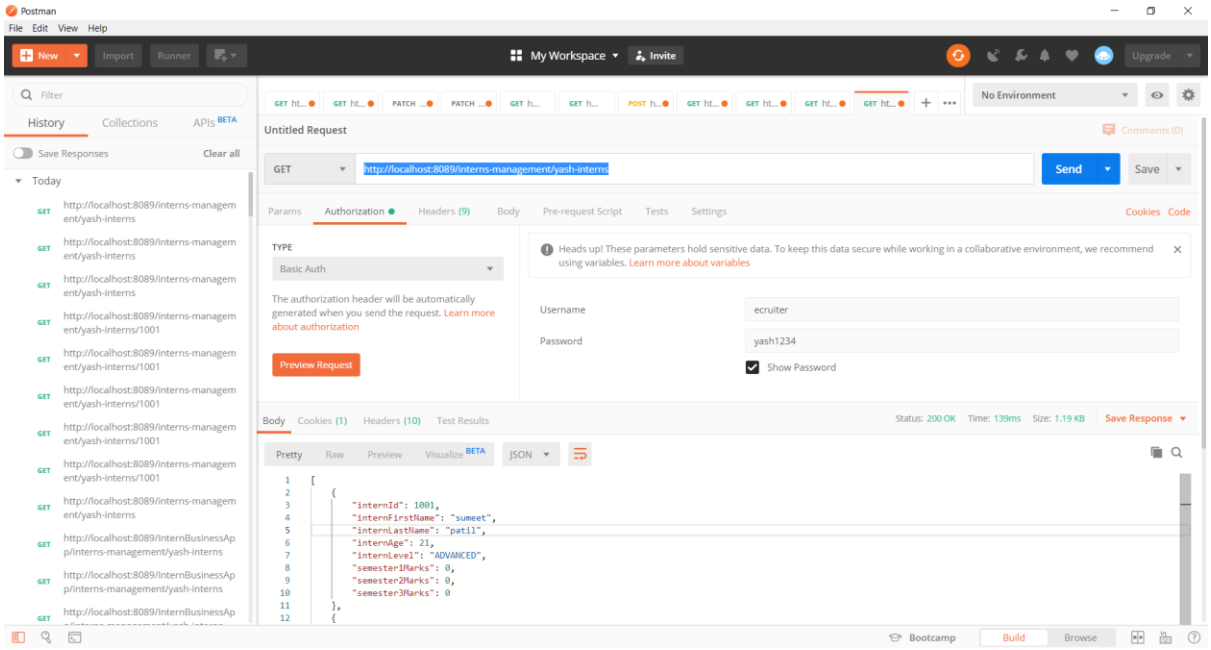
Method-level security is implemented by placing the @PreAuthorize and @PostAuthorize annotation on controller methods. This annotation contains a Spring Expression Language (SpEL) snippet that is assessed to determine if the request should be authenticated. If access is not granted, the method is not executed and an HTTP Unauthorized is returned.

Test using postman.

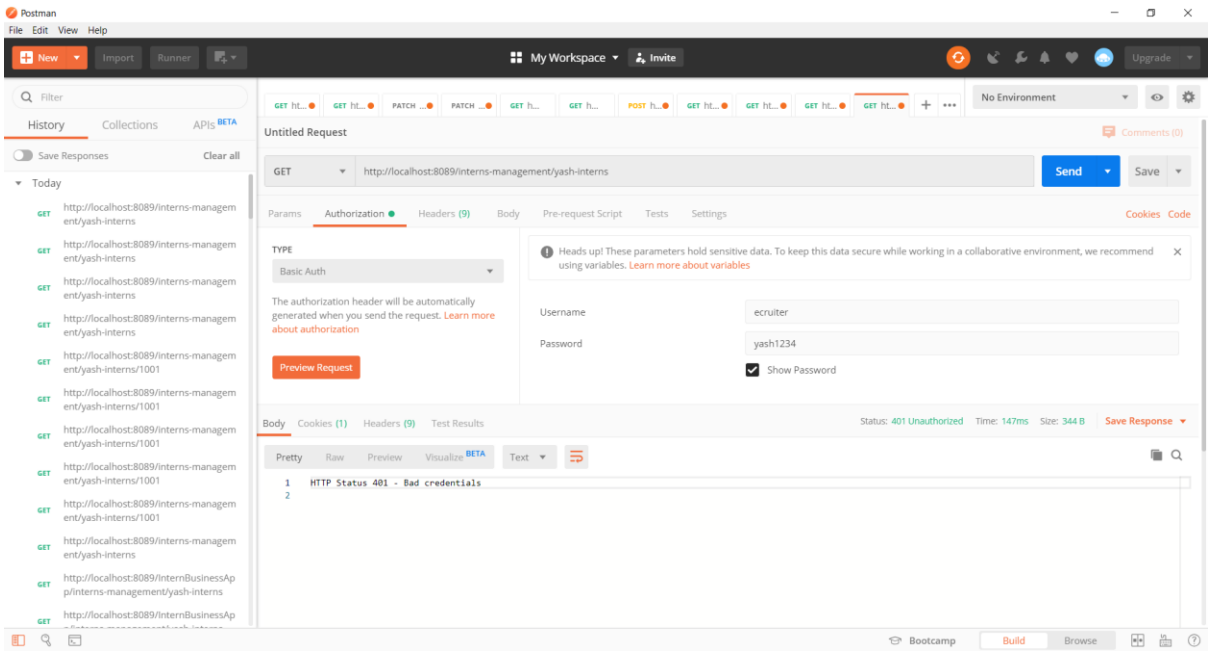
GET: http://localhost:8089/interns-management/yash-interns

Authorization: Basic AUTH

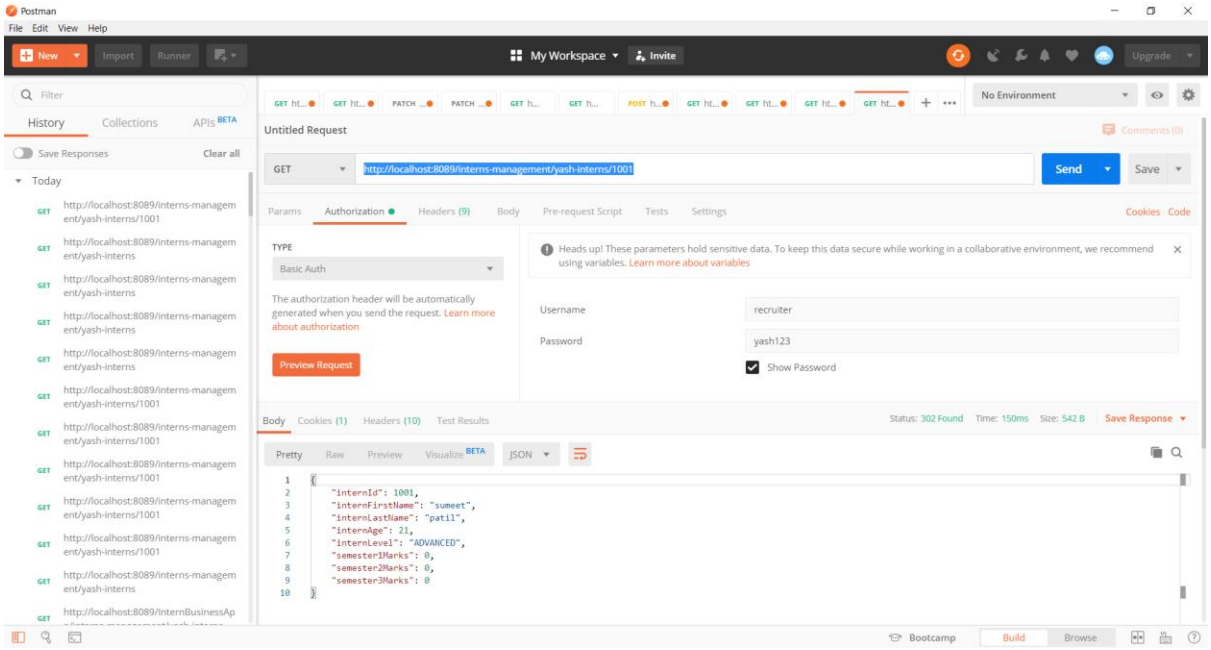
Username: recruiter
Password: yash123



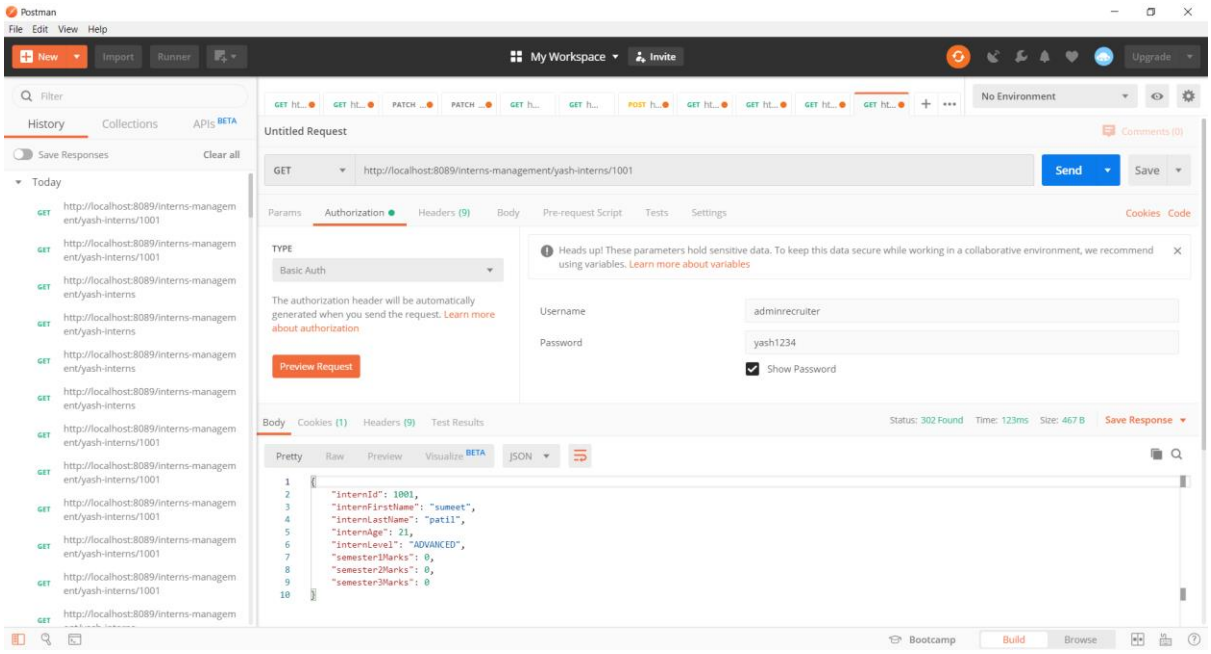
Give incorrect username or password,



GET: http://localhost:8089/interns-management/yash-interns/1001



Even user with SeniorRecruiter role can view data,



POST: http://localhost:8089/interns-management/yash-interns

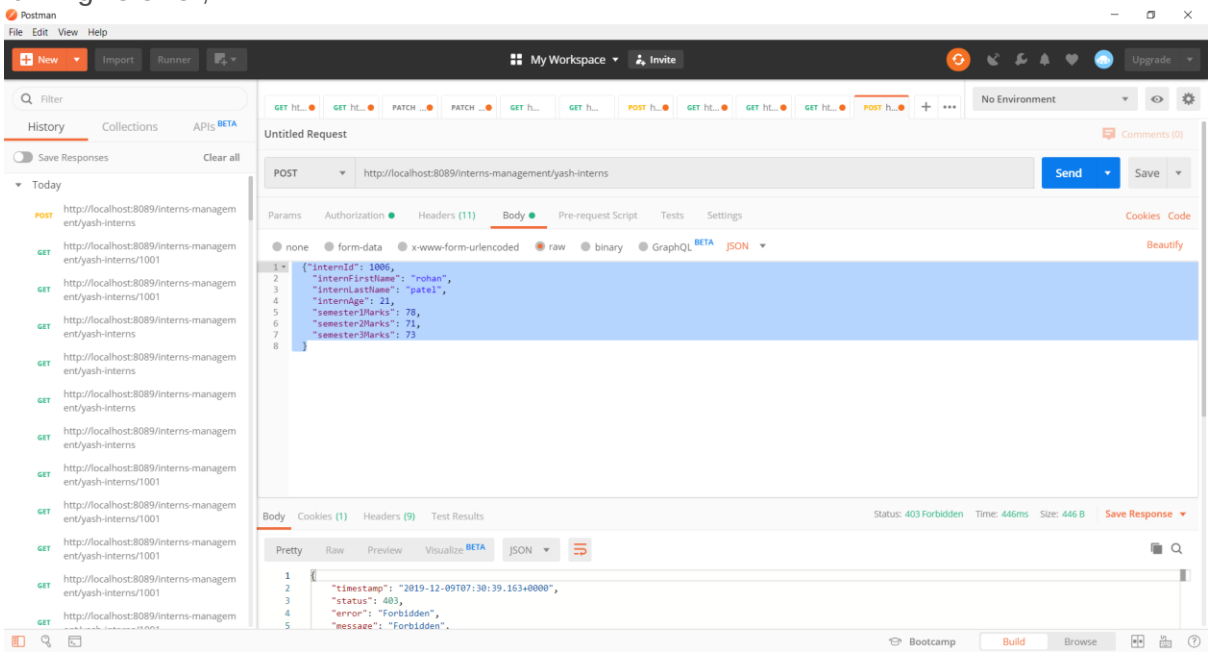
JSON data,

```
{
  "internId": 1006,
  "internFirstName": "rohan",
  "internLastName": "patel",
  "internAge": 21,
  "semester1Marks": 78,
  "semester2Marks": 71,
  "semester3Marks": 73
}
```

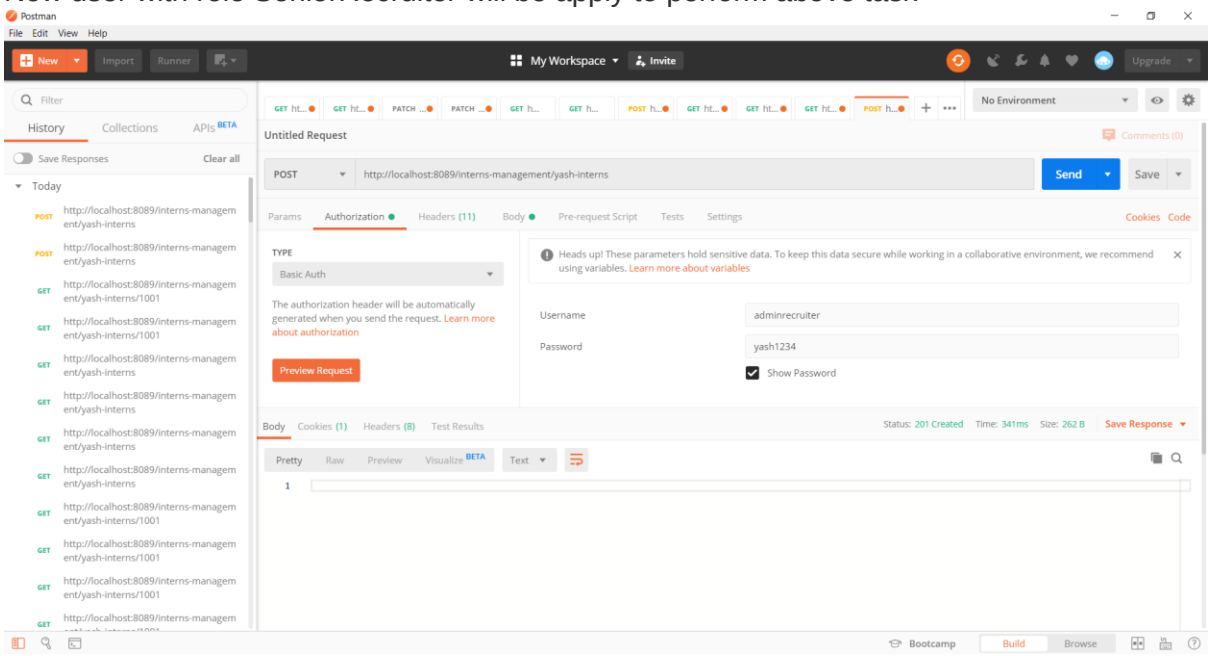
Authorization
Username: recruiter
Password: yash123

Since we have granted access for post,put and delete only to user with role SeniorRecruiter,

It will give error,



Now user with role SeniorRecruiter will be apply to perform above task



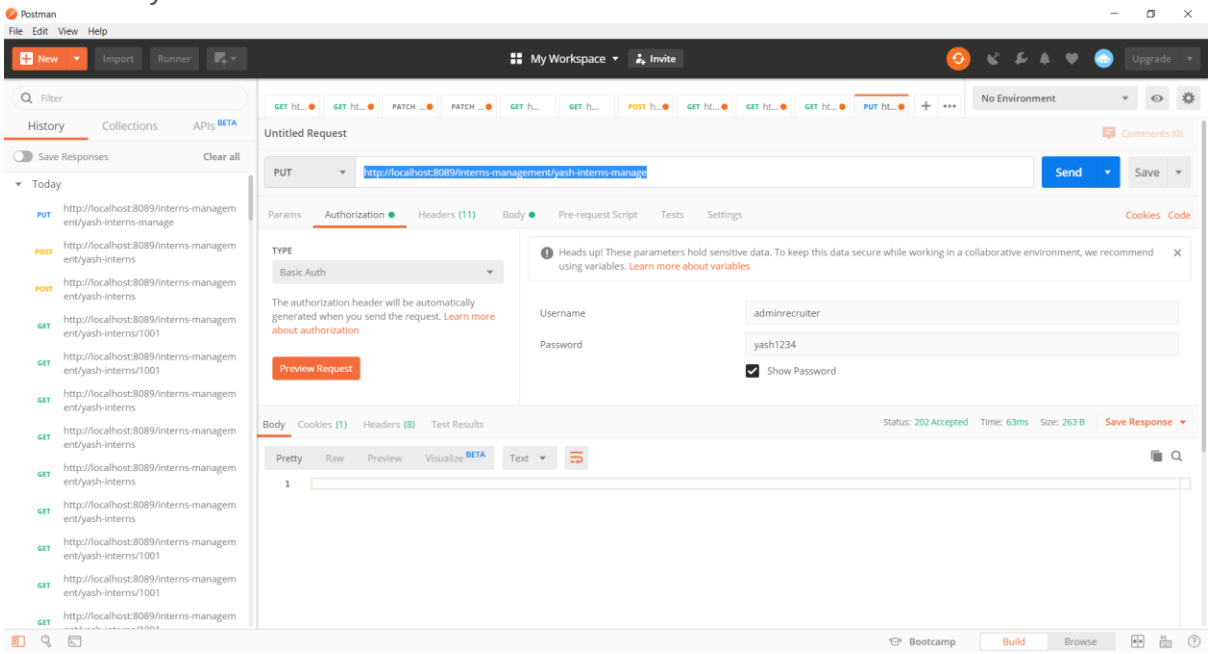
Similarly for PUT and DELETE,

PUT: http://localhost:8089/interns-management/yash-interns-manage

Authorization

Username: adminrecruiter

Password: yash1234



DELETE: http://localhost:8089/interns-management/yash-interns-manage/1006

Username: adminrecruiter
Password: yash1234