Spring Boot JPA

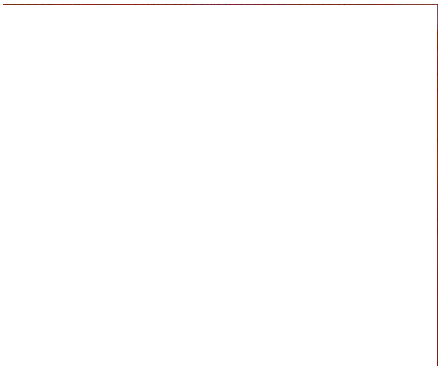
What is JPA?

Spring Boot JPA is a Java specification for managing **relational** data in Java applications. It allows us to access and persist data between Java object/ class and relational database. JPA follows **Object-Relation Mapping** (ORM). It is a set of interfaces. It also provides a runtime **EntityManager** API for processing queries and transactions on the objects against the database. It uses a platform-independent object-oriented query language JPQL (Java Persistent Query Language).

In the context of persistence, it covers three areas:

- The Java Persistence API
- **Object-Relational** metadata
- The API itself, defined in the **persistence** package

JPA is not a framework. It defines a concept that can be implemented by any framework.



Why should we use JPA?

JPA is simpler, cleaner, and less labor-intensive than JDBC, SQL, and hand-written mapping. JPA is suitable for non-performance oriented complex applications. The main advantage of JPA over JDBC is that, in JPA, data is represented by objects and classes while in JDBC data is represented by tables and records. It uses POJO to represent persistent data that simplifies database programming. There are some other advantages of JPA:

- JPA avoids writing DDL in a database-specific dialect of SQL. Instead of this, it allows mapping in XML or using Java annotations.
- JPA allows us to avoid writing DML in the database-specific dialect of SQL.
- JPA allows us to save and load Java objects and graphs without any DML language at all.
- When we need to perform queries JPQL, it allows us to express the queries in terms of Java entities rather than the (native) SQL table and columns.

JPA Features

There are following features of JPA:

- It is a powerful repository and custom **object-mapping abstraction.**

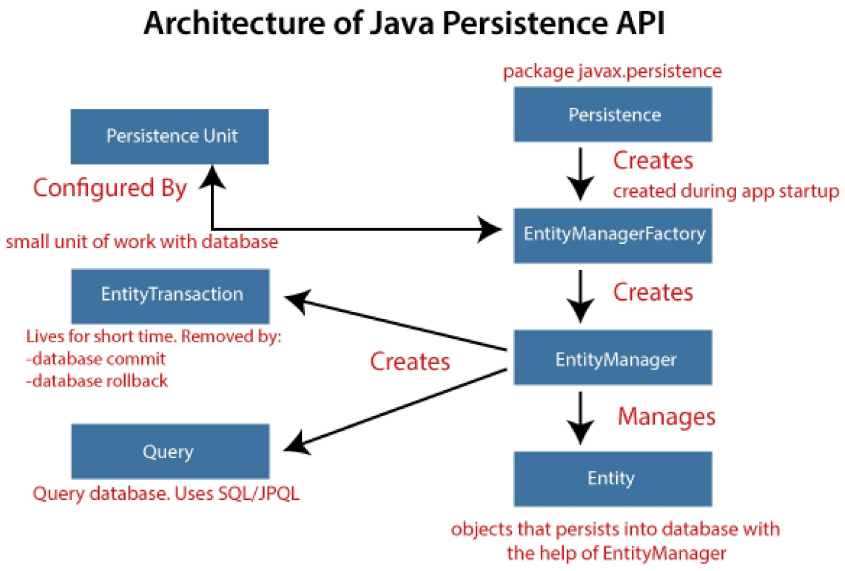
- It supports for **cross-store persistence**. It means an entity can be partially stored in MySQL and Neo4j (Graph Database Management System).
- It dynamically generates queries from queries methods name.
- The domain base classes provide basic properties.
- It supports transparent auditing.
- Possibility to integrate custom repository code.
- It is easy to integrate with Spring Framework with the custom namespace.

JPA Architecture

JPA is a source to store business entities as relational entities. It shows how to define a POJO as an entity and how to manage entities with relation.

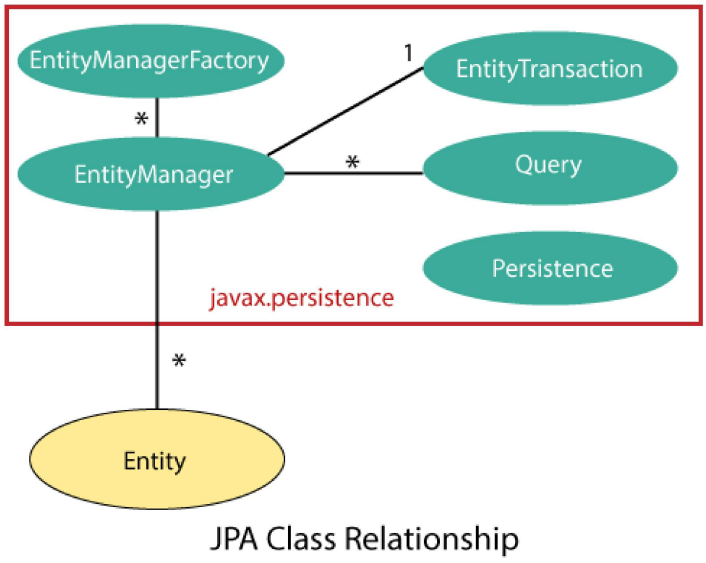
The following figure describes the class-level architecture of JPA that describes the core classes and interfaces of JPA that is defined in the **javax persistence** package. The JPA architecture contains the following units:

- **Persistence:** It is a class that contains static methods to obtain an EntityManagerFactory instance.
- **EntityManagerFactory:** It is a factory class of EntityManager. It creates and manages multiple instances of EntityManager.
- **EntityManager:** It is an interface. It controls the persistence operations on objects. It works for the Query instance.
- **Entity:** The entities are the persistence objects stores as a record in the database.
- **Persistence Unit:** It defines a set of all entity classes. In an application, EntityManager instances manage it. The set of entity classes represents the data contained within a single data store.
- **EntityTransaction:** It has a **one-to-one** relationship with the EntityManager class. For each EntityManager, operations are maintained by EntityTransaction class.
- **Query:** It is an interface that is implemented by each JPA vendor to obtain relation objects that meet the criteria.



JPA Class Relationships

The classes and interfaces that we have discussed above maintain a relationship. The following figure shows the relationship between classes and interfaces.



JPA Class Relationship

- The relationship between EntityManager and EntiyTransaction is **one-to-one**. There is an EntityTransaction instance for each EntityManager operation.
- The relationship between EntityManageFactory and EntiyManager is **one-to-many**. It is a factory class to EntityManager instance.
- The relationship between EntityManager and Query is **one-to-many**. We can execute any number of queries by using an instance of EntityManager class.
- The relationship between EntityManager and Entity is **one-to-many**. An EntityManager instance can manage multiple Entities.

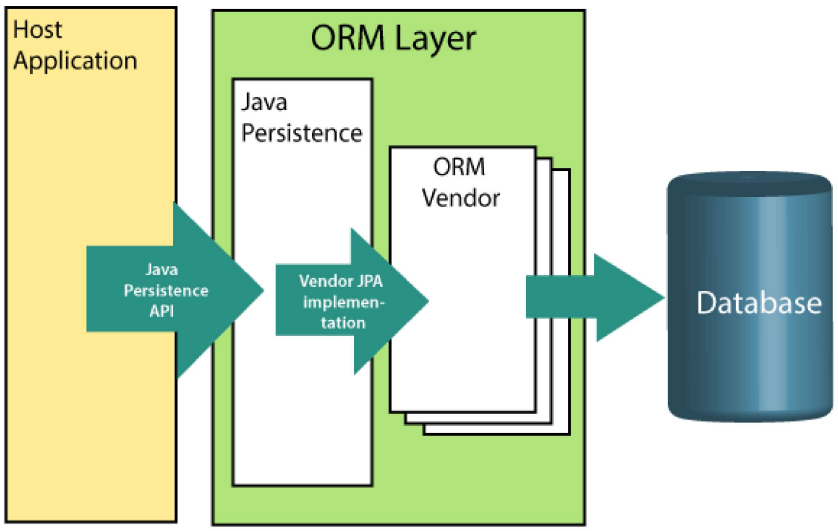
JPA Implementations

JPA is an open-source API. There is various enterprises vendor such as Eclipse, RedHat, Oracle, etc. that provides new products by adding the JPA in them. There are some popular JPA implementations frameworks such as **Hibernate, EclipseLink, DataNucleus,** etc. It is also known as **Object-Relation Mapping** (ORM) tool.

Object-Relation Mapping (ORM)

In ORM, the mapping of Java objects to database tables, and vice-versa is called **Object-Relational Mapping.** The ORM mapping works as a bridge between a **relational database** (tables and records) and **Java application** (classes and objects).

In the following figure, the ORM layer is an adapter layer. It adapts the language of object graphs to the language of SQL and relation tables.



The ORM layer exists between the application and the database. It converts the Java classes and objects so that they can be stored and managed in a relational database. By default, the name that persists become the name of the table, and fields become columns. Once an application sets-up, each table row corresponds to an object.

JPA Versions

Earlier versions of EJB defines the persistence layer combined with the business logic layer using **javax.ejb.EntityBean** Interface. EJB specification includes the definition of JPA.

While introducing EJB 3.0, the persistence layer was separated and specified as JPA 1.0 (Java Persistence API). The specifications of this API were released along with the specifications of JAVA EE5 on May 11, 2006, using JSR 220.

In 2019, JPA renamed to **Jakarta Persistence**. The latest version of JPA is **2.2**. It supports the following features:

- Java 8, data and time API
- CDI Injection in AttributeConvertes
- It makes annotations @Repeatable

Difference between JPA and Hibernate

JPA: JPA is a Java specification that is used to access, manage, and persist data between Java object and relational database. It is a standard approach for ORM.

Hibernate: It is a lightweight, open-source ORM tool that is used to store Java objects in the relational database system. It is a provider of JPA. It follows a common approach provided by JPA.

The following table describes the differences between JPA and Hibernate.

JPA	Hibernate
JPA is a **Java specification** for mapping relation data in Java application.	Hibernate is an **ORM framework** that deals with data persistence.
JPA does not provide any implementation classes.	It provides implementation classes.
It uses platform-independent query language called **JPQL** (Java Persistence Query Language).	It uses its own query language called **HQL** (Hibernate Query Language).
It is defined in **javax.persistence** package.	It is defined in **org.hibernate** package.
It is implemented in various ORM tools like **Hibernate, EclipseLink,** etc.	Hibernate is the **provider** of JPA.
JPA uses **EntityManager** for handling the persistence of data.	In Hibernate uses **Session** for handling the persistence of data.

Spring Boot Starter Data JPA

Spring Boot provides starter dependency **spring-boot-starter-data-jpa** to connect Spring Boot application with relational database efficiently. The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
<version>2.2.2.RELEASE</version>
</dependency>
```

Spring Boot JPA Example

Let's create a Spring Boot application that uses JPA to connect to the database. In the following example, we have used in-memory database **Apache Derby.**

Apache Derby: It is an **open-source, embedded** relational database implemented entirely in Java. It is available under the Apache License 2.0. There are following advantages of Apache Derby:

- It is easy to install, deploy, and use.
- It is based on Java, JDBC, and SQL standards.
- It provides an embedded JDBC driver that allows us to embed Derby in any Java-based solution.
- It also supports client/server mode with the Derby Network Client JDBC driver, and Derby Network Server.

Spring Boot can auto-configure an embedded database such as **H2, HSQL,** and **Derbydatabases**. We do not need to provide any connection URLs. We need only include a build dependency on the embedded database that we want to use.

In Spring Boot, we can easily integrate Apache Derby database just by adding **Derby** dependency in pom.xml file.

```
<dependency>
<groupId>org.apache.derby</groupId>
<artifactId>derby</artifactId>
<scope>runtime</scope>
</dependency>
```

Step 1: Open Spring Initializr https://start.spring.io/.

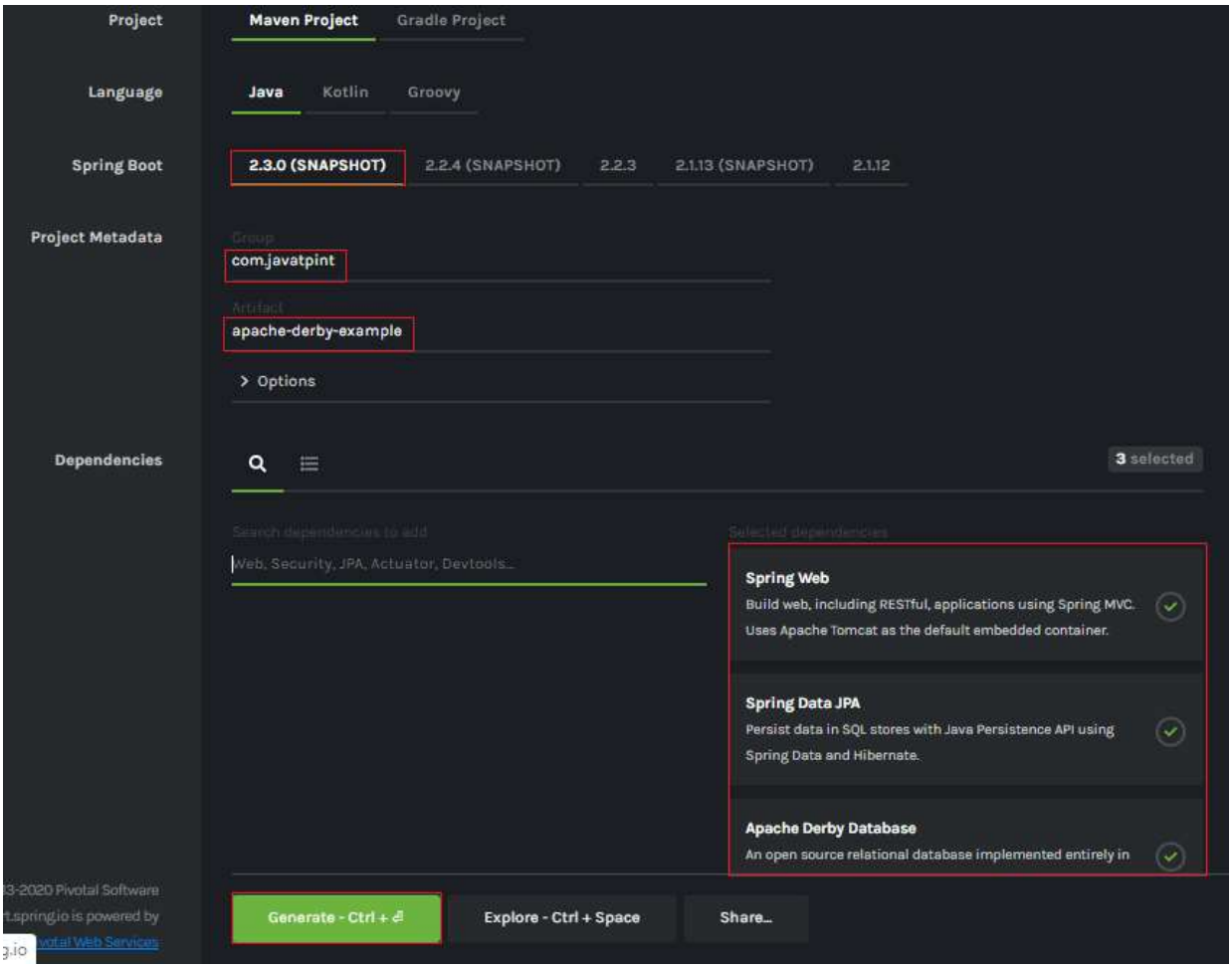
Step 2: Select the latest version of Spring Boot **2.3.0(SNAPSHOT)**

Step 3: Provide the **Group** name. We have provided **com.javatpoint.**

Step 4: Provide the **Artifact** Id. We have provided **apache-derby-example**.

Step 5: Add the dependencies: **Spring Web, Spring Data JPA,** and **Apache Derby Database**.

Step 6: Click on the **Generate** button. When we click on the Generate button, it wraps the project in a Jar file and downloads it to the local system.



Step 7: Extract the Jar file and paste it into the STS workspace.

Step 8: Import the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder apache-derby-example -> Finish

It takes some time to import.

Step 9: Create a package with the name **com.javatpoint.model** in the folder **src/main/java.**

Step 10: Create a class with the name **UserRecord** in the package **com.javatpoint.model** and do the following:

- Define three variables **id, name,** and **email**.
- Generate Getters and Setter.
  Right-click on the file -> Source -> Generate Getters and Setters
- Define a default constructor.
- Mark the class as an **Entity** by using the annotation **@Entity.**
- Mark **Id** as the primary key by using the annotation **@Id.**

UserRecord.java

```
package com.javatpoint.model;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class UserRecord
{
@Id
private int id;
private String name;
private String email;
//default conatructor
public UserRecord()
{
}
public int getId()
{
return id;
}
public void setId(int id)
{
this.id = id;
}
public String getName()
{
return name;
}
public void setName(String name)
{
this.name = name;
}
public String getEmail()
{
return email;
```

```
}
public void setEmail(String email)
{
this.email = email;
}
}
```

Step 11: Create a package with the name **com.javatpoint.controller** in the folder **src/main/java.**

Step 12: Create a Controller class with the name **UserController** in the package **com.javatpoint.controller** and do the following:

- Mark the class as a controller by using the annotation **@RestController.**
- Autowired the class **UserService** by using the annotation **@Autowired**.
- We have defined two mappings, one for **getting all users** and the other for **add-user.**

UserController.java

```
package com.javatpoint.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import com.javatpoint.model.UserRecord;
import com.javatpoint.service.UserService;
import java.util.List;
@RestController
public class UserController
{
@Autowired
private UserService userService;
@RequestMapping("/")
public List<UserRecord> getAllUser()
{
return userService.getAllUsers();
}
@RequestMapping(value="/add-user", method=RequestMethod.POST)
public void addUser(@RequestBody UserRecord userRecord)
{
userService.addUser(userRecord);
}
}
```

Step 13: Create a package with the name **com.javatpoint.service** in the folder **src/main/java.**

Step 14: Create a Service class with the name **UserController** in the package **com.javatpoint.service** and do the following:

- Mark the class as service by using the annotation **@Service.**
- Autowired the **UserRepository**
- Define a method **getAllUsers()** that returns a List of
- Define another method name **addUser()** that saves the user record.

UserService.java

```
package com.javatpoint.service;
import java.util.List;
import java.util.ArrayList;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.javatpoint.model.UserRecord;
import com.javatpoint.repository.UserRepository;
@Service
public class UserService
{
@Autowired
private UserRepository userRepository;
public List<UserRecord> getAllUsers()
{
List<UserRecord>userRecords = new ArrayList<>();
userRepository.findAll().forEach(userRecords::add);
return userRecords;
}
public void addUser(UserRecord userRecord)
{
userRepository.save(userRecord);
}
}
```

Step 15: Create a package with the name **com.javatpoint.repository** in the folder **src/main/java.**

Step 16: Create a repository interface with the name **UserRepository** in the package **com.javatpoint.repository** and extends **CrudRepository**.

UserRepository.java

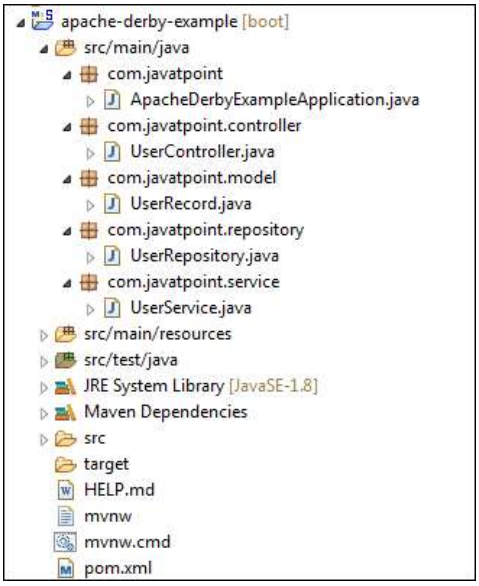
```
package com.javatpoint.repository;
import org.springframework.data.repository.CrudRepository;
import com.javatpoint.model.UserRecord;
public interface UserRepository extends CrudRepository<UserRecord, String>
{
}
```

Step 17: Now, open the **ApacheDerbyExampleApplication.java** file. It created by default when we set-up an application.

ApacheDerbyExampleApplication.java

```
package com.javatpoint;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class ApacheDerbyExampleApplication
{
public static void main(String[] args)
{
SpringApplication.run(ApacheDerbyExampleApplication.class, args);
}
}
```

Now, we have set-up all the necessary classes and packages according to the requirements. Notice that we have not provided any **connection URL** for the database. After completing all the above steps, the project directory looks like the following:



Let's run the application.

Step 18: Open the **ApacheDerbyExampleApplication.java** file and run it as Java Application.

Step 19: Open the browser and invoke the URL http://localhost:8080/. It returns an empty list because we have not added any user in the List.

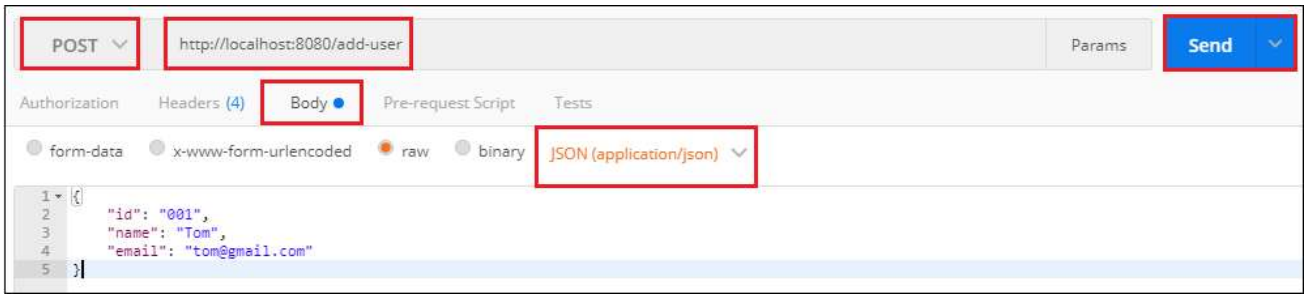
To add a user to the database, we will send a **POST** request by using the **Postman**.

Step 20: Open the **Postman** and do the following:

- Select the **POST**
- Invoke the URL http://localhost:8080/add-user.
- Click on the **Body**
- Select Content-Type as **JSON**(application/json).
- Insert the data which want to insert in the database. We have inserted the following data:

```
{
"id": "001",
"name": "Tom",
"email": "tom@gmail.com"
}
```

- Click on the **Send** button.



When we click on the Send button, it shows **Status:200 OK**. It means the request has been successfully executed.

Step 21: Open the browser and invoke the URL http://localhost:8080. It returns the user that we have inserted in the database.

Download Apache derby Example Project

← Prev

Next →

Artificial Intelligence Tutorial

AI

AWS Tutorial

AWS

Selenium tutorial

Selenium

Cloud tutorial

Cloud

Hadoop tutorial

Hadoop

ReactJS Tutorial

ReactJS

Data Science Tutorial

D. Science

Angular 7 Tutorial

Angular 7

Blockchain Tutorial

Blockchain

Git Tutorial

Git

Machine Learning Tutorial

ML

DevOps Tutorial

DevOps

B.Tech / MCA

DBMS tutorial

DBMS

Data Structures tutorial

DS

DAA tutorial

DAA

Operating System tutorial

OS

Computer Network tutorial

C. Network

Compiler Design tutorial

Compiler D.

Computer Organization and Architecture

COA

Discrete Mathematics Tutorial

D. Math.

Ethical Hacking Tutorial

E. Hacking

Computer Graphics Tutorial

C. Graphics

Software Engineering Tutorial

Software E.

html tutorial

Web Tech.

Cyber Security tutorial

Cyber Sec.

Automata Tutorial

Automata

C Language tutorial

C

C++ tutorial

C++

Java tutorial

Java

.Net Framework tutorial

.Net

Python tutorial

Python

List of Programs

Programs

Control Systems tutorial

Control S.

Data Mining Tutorial

Data Mining