REST API DEVELOPMENT USING JAVA FOR ROBOTICS APPLICATIONS

CONTENTS:

- Problem Statement
- Methodology
- Block Diagram
- Software Tools Used
- API Design & Endpoints

- Algorithm
- JWT Mechanism
- Database,Configurations,Dependencies, Deployment
- Working
- Future Enhancements
- Github Link

PROBLEM STATEMENT

REST API Development using Java for Robotics Applications

Building a scalable and efficient REST API using Java for multiple applications. The API

will handle database access, algorithmic computations (such as pathfinding), AI and image

processing. The system will be designed with modular endpoints, allowing seamless integration

with various applications. It will support real-time processing, authentication, and optimized

performance using Java-based frameworks like Spring Boot.

METHODOLOGY

1.Set up the Spring Boot project in intellije ide,necessary dependencies,java21

2.Design and implement REST API and endpoints which takes request and responses in json.

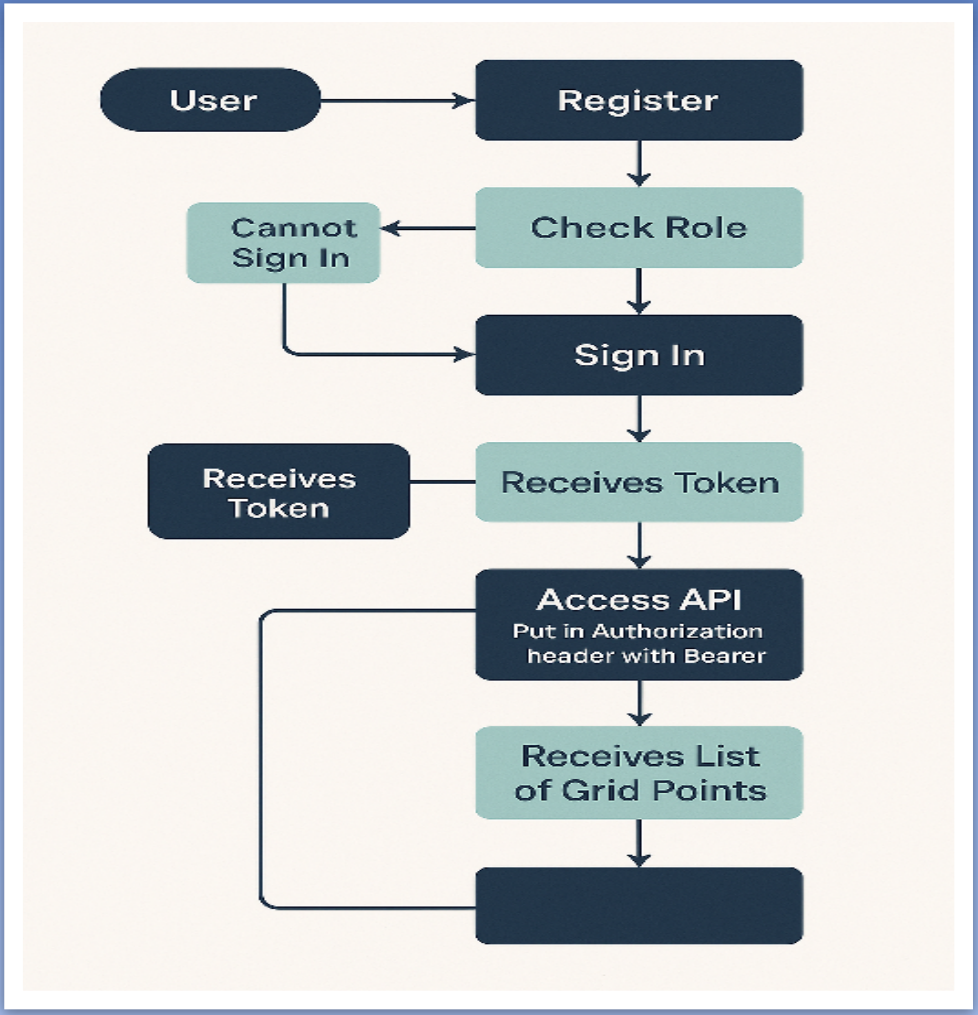
3.Develop a grid-based shortest path algorithm.

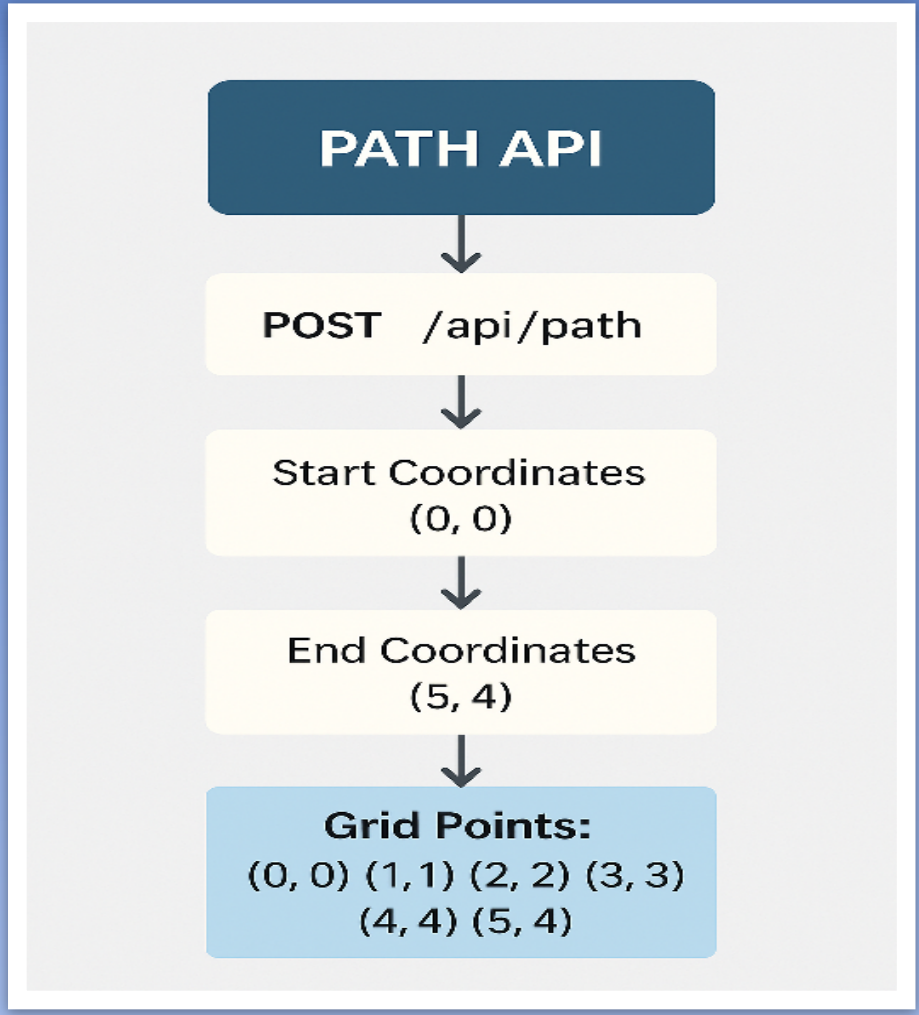
4.Use H2 in-memory database and other confiqurations.

5.Implement JWT authentication for authorizated and authenticated API access.

6.Test the API and security functionality using Postman.

BLOCK DIAGRAM

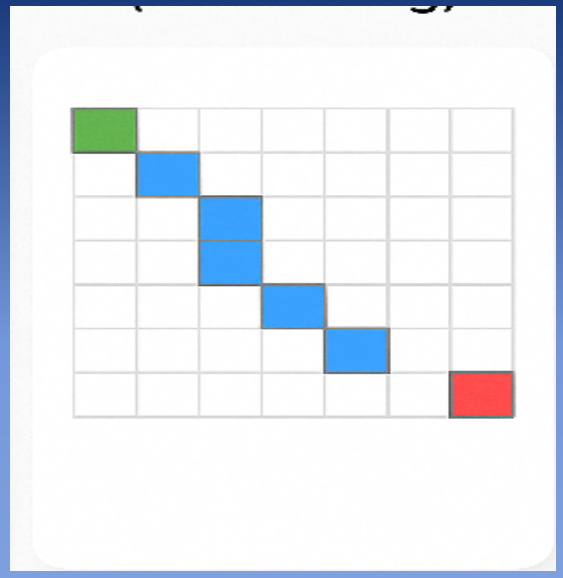




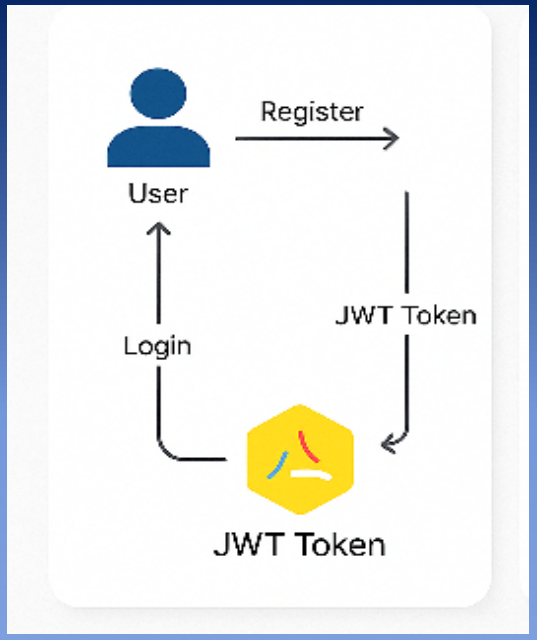
SOFTWARE TOOLS USED:

- Intellij IDEA
- Postman
- SpringBoot
- Spring Security,Spring JPA
- H2 Datbase,Lombok
- Maven
- Java 24,Github

API DESIGN AND ENDPOINTS



SHORTEST PATH ALGORITHM



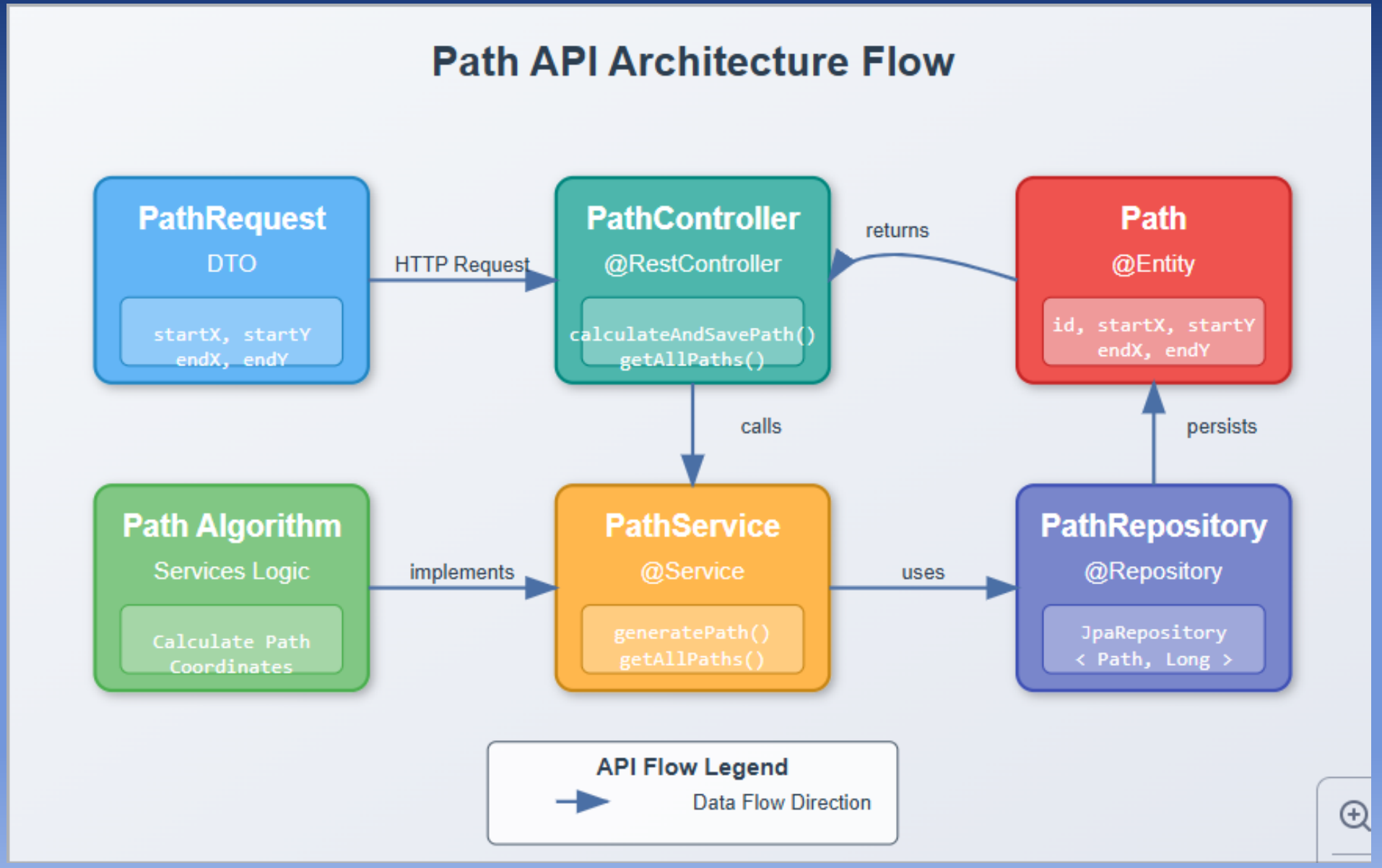
JWT MECHANISM

PROJECT BREAKDOWN

API DESIGN AND ENDPOINTS

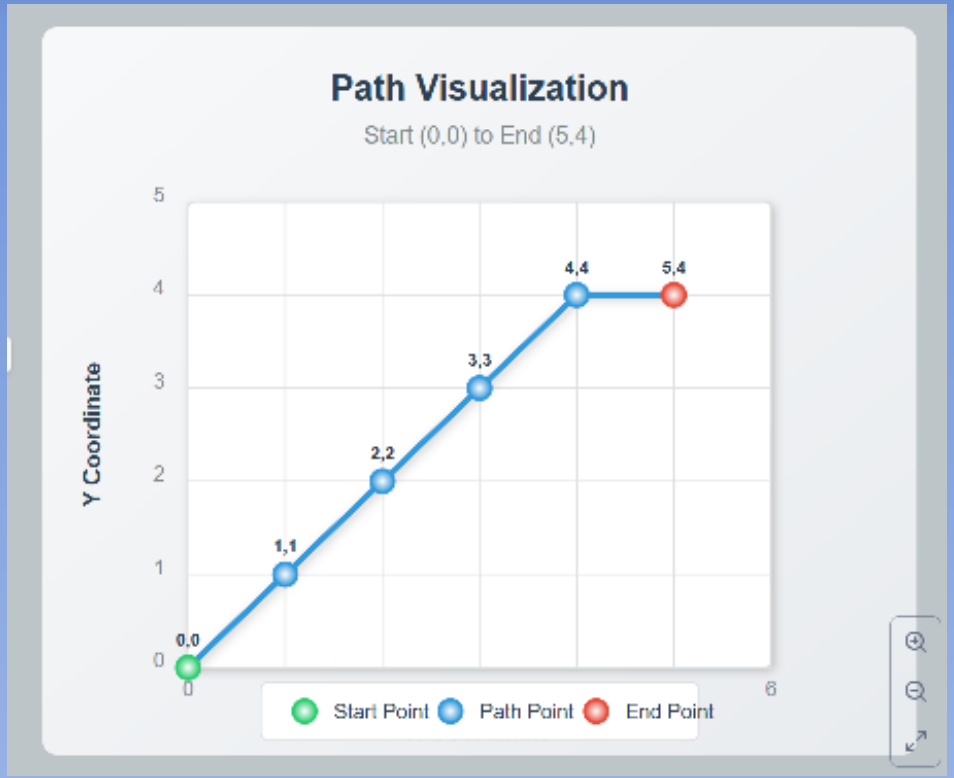
PATH MODEL	PATH DTO	PATH SERVICE	PATH CONTROLLER	PATH REPOSITORY

@Id
@GeneratedValue
private Long id;
 	
private int startX;
private int startY;
private int endX;
private int endY;
 	
private String pathPoints;
	@Data
	public class PathRequest {
	private int startX;
	private int startY;
	private int endX;
	private int endY;
	}
		1.Algorithm
		2. @Autowired private PathRepository pathRepository;
		3.Methods to generate path,get all path,calculate and save all path using the path repository object.
			@RestController
			@RequestMapping("/api/path")
			public class PathController {
			
			@Autowired
			private PathService pathService;
			@PostMapping
			public Path calculateAndSavePath(@RequestBody PathRequest request)
			@GetMapping
			public List<Path> getAllPaths() {
			return pathService.getAllPaths();
			}
				public interface PathRepository extends JpaRepository<Path, Long> {
				}


Path API Architecture Flow
PathRequest
DTO
startX, startY
endX, endY
HTTP Request
PathController
@RestController
calculateAndSavePath()
getAllPaths()
returns
Path
@Entity
id, startX, startY
endX, endY
calls
persists
Path Algorithm
Services Logic
Calculate Path
Coordinates
implements
PathService
@Service
generatePath()
getAllPaths()
uses
PathRepository
@Repository
JpaRepository
< Path, Long >
API Flow Legend
Data Flow Direction

SHORTEST PATH ALGORITHM

1. **Determine Movement Direction**
   The algorithm starts by figuring out how to move from the start to the end point using
   int dx = Integer.compare(x, i); and int dy = Integer.compare(y, j);
   which decide if movement along each axis should be +1, -1, or 0.
2. **Initialize the Path**
   A list is created to hold the path using List<Point> path = new LinkedList<>();, and the
3. starting point is added with path.add(new Point(p, q));.
3. **Traverse Step-by-Step**
   A while loop while (p != x || q != y) runs until the destination is reached. Inside, it moves diagonally
   when both p != x and q != y by
   incrementing p += dx and q += dy, or straight when only one direction remains.
   After every move, path.add(new Point(p, q)); records the step.
4. **Complete Path Generation**
   The loop continues until the current point matches the target. The final path list contains the shortest
   path made of Point objects from start to end, allowing smooth and efficient traversal.



SHORTEST PATH ALGORITHM

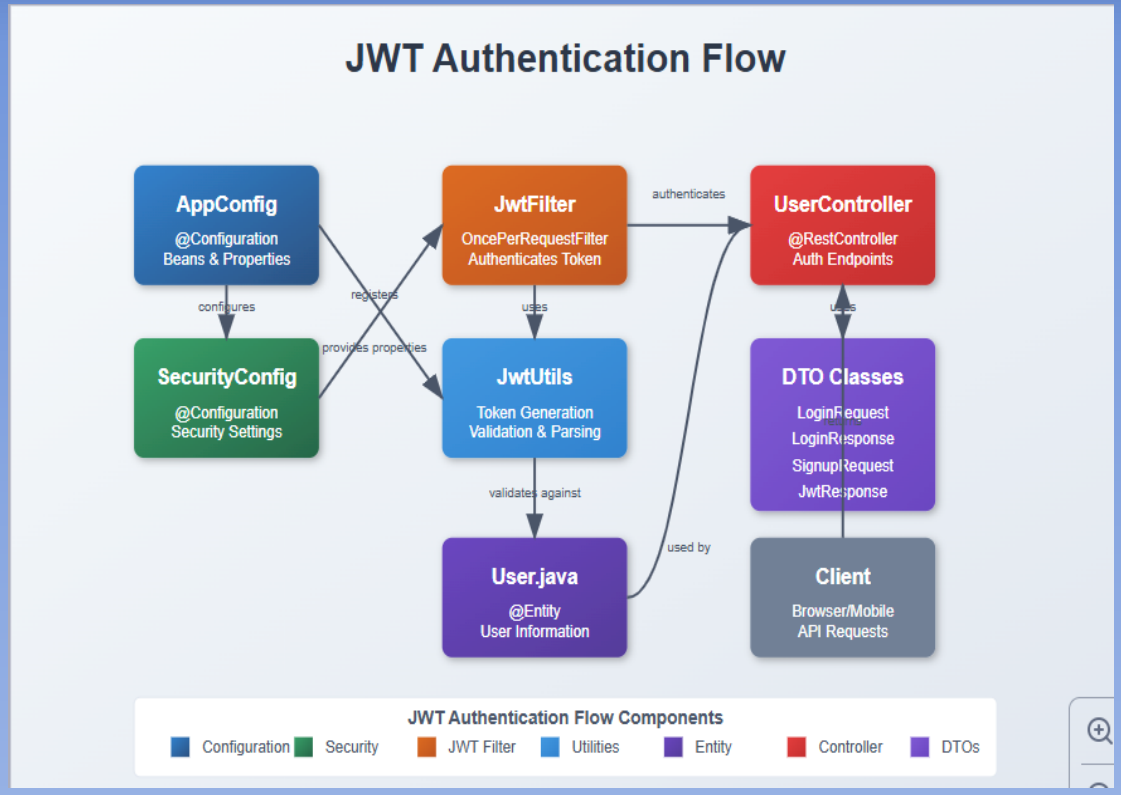
```
import java.awt.Point;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class PathGenerator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int startX = scanner.nextInt(), startY = scanner.nextInt(),
endX = scanner.nextInt(), endY = scanner.nextInt();
        for (Point point : generatePath(startX, startY, endX, endY)) {
            System.out.print("(" + point.x + "," + point.y + ") ");
        }
    }
}
```

```
    public static List<Point> generatePath(int i, int j, int x, int y)
{
        List<Point> path = new ArrayList<>();
        int p = i, q = j, dx = Integer.compare(x, i), dy = Integer.
compare(y, j);
        path.add(new Point(p, q));
        while (p != x || q != y) {
            if (p != x) p += dx;
            if (q != y) q += dy;
            path.add(new Point(p, q));
        }
        return path;
```

JWT MECHANISM

Header	Payload	Signature

{
"alg": "HS256",
"typ": "JWT"
}
	{
	"sub": "john.doe",
	"name": "John Doe", "role": "admin)
	"iat": 1614560983,
	"exp": 1614564583
	}
		Signature = HMACSHA256(
		
		base64UrlEncode(header) + "." + base64UrlEncode(payload),
		secret)



JWT MECHANISM

APP CONFIG	JWTUtil	Security Configurations	User	User Controller	JWTFilter	DTO	User repository
Defines how Spring Security loads user details (username, password, role) from the database during authentication. 4o	generating, extracting, and validating JWT tokens using a secret key and claims.	Spring Security filter chain, disables CSRF, sets up authorization rules, and adds the JWT filter.	Entity class representing a user with fields like username, password, and role, stored in the database.	Exposes endpoints for user registration and login; returns JWT on successful login.	Intercepts requests, extracts JWT from headers, validates it, and sets the authenticated user in SecurityContext.	Java objects for transferring data between client and server during registration and login.	UserRepository interface handles database operations for the User entity and throws an exception if a user is not found by

DATABASE AND CONFIGURATIONS,

1.**DATABASE:**
H2 in-memory used to store data like users, tokens, paths etc.
2.**JPA:**Manage and access relational data
3.**Hibernate**:A popular JPA implementation that translates Java code (entities) into SQL and interacts with the database.
jpa cant happen without hibernate jpa is a interface and hibernate implements it
5. **JDBC Driver**:va Database Connectivity driver that enables Java apps to connect and communicate with databases (like the H2 driver here).

```
spring.application.name=JWT
#spring.datasource.url=jdbc:h2:mem:testdb
#spring.datasource.driverClassName=org.h2.Driver
#spring.datasource.username=sa
#spring.datasource.password=
#spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
#spring.h2.console.enabled=true
#spring.jpa.hibernate.ddl-auto=update
#logging.level.org.hibernate.SQL=debug
```

DEPENDENCIES AND DEPLOYMENT

DEPENDENCIES
Spring Boot Starter Web
Spring Boot Starter Security
Spring Boot Starter Data JPA
h2
lombok
spring-boot-devtools
spring-boot-starter-test

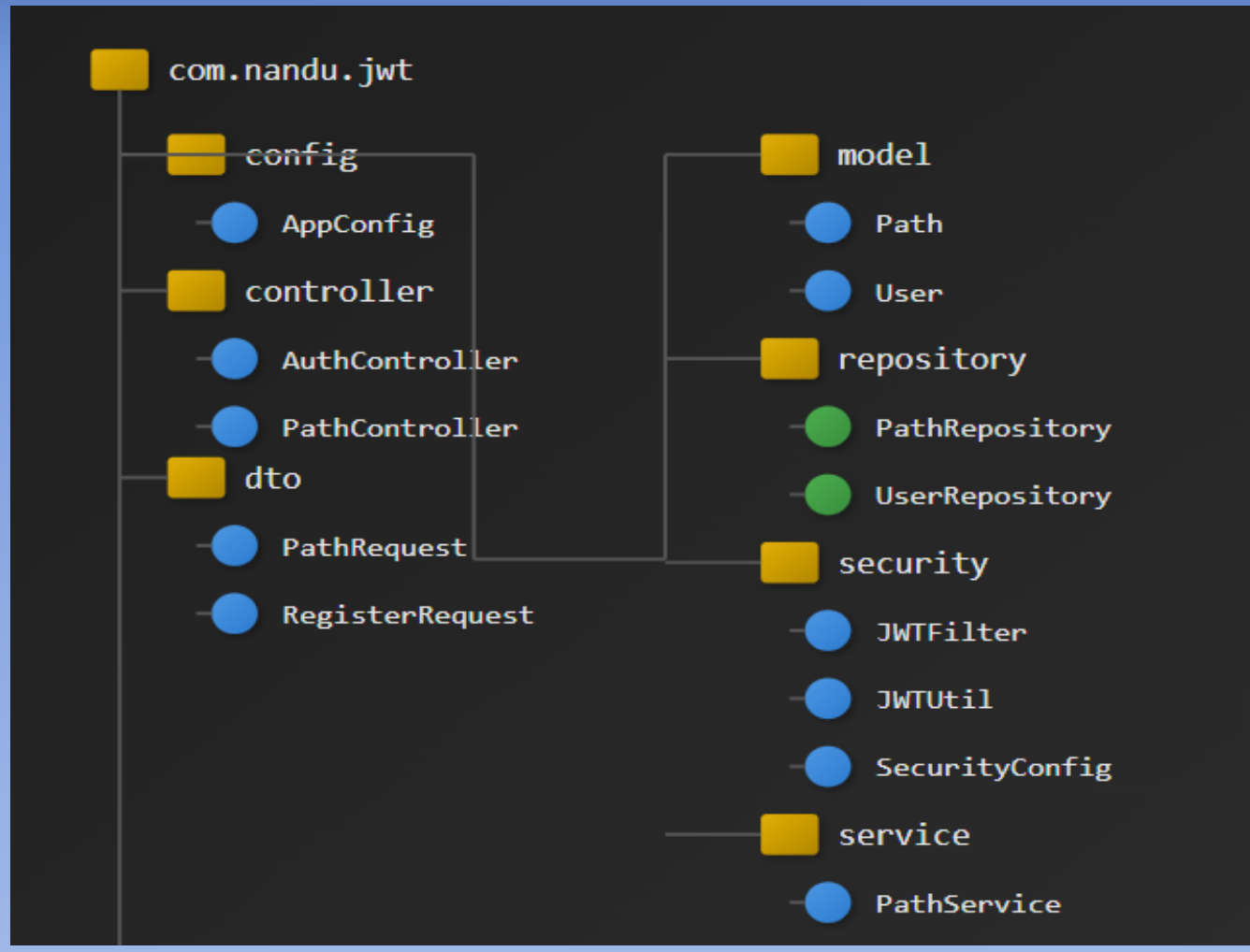
spring-security-test
jjwt-api
jjwt-impl
jjwt-jackson

APPLICATION FILE:
@SpringBootApplication
SpringApplication.run(classname);

The application is deployed locally and runs on embedded tomcat server at http://localhost:8080

WORKING(POSTMAN)

- **Register User**
  POST /auth/register
  json
  { "username": "john", "password": "1234", "role": "ROLE_USER" }
- **Login for Token**
  POST /auth/login
  json
  { "username": "john", "password": "1234" }
  ➜ Response: { "token": "..." }
- **Access Path API**
  POST /api/path
  Header: Authorization: Bearer <token>
  json
  { "robotId": "R1", "startX": 0, "startY": 0, "endX": 5, "endY": 54}
  ➜ Response: Computed shortest path

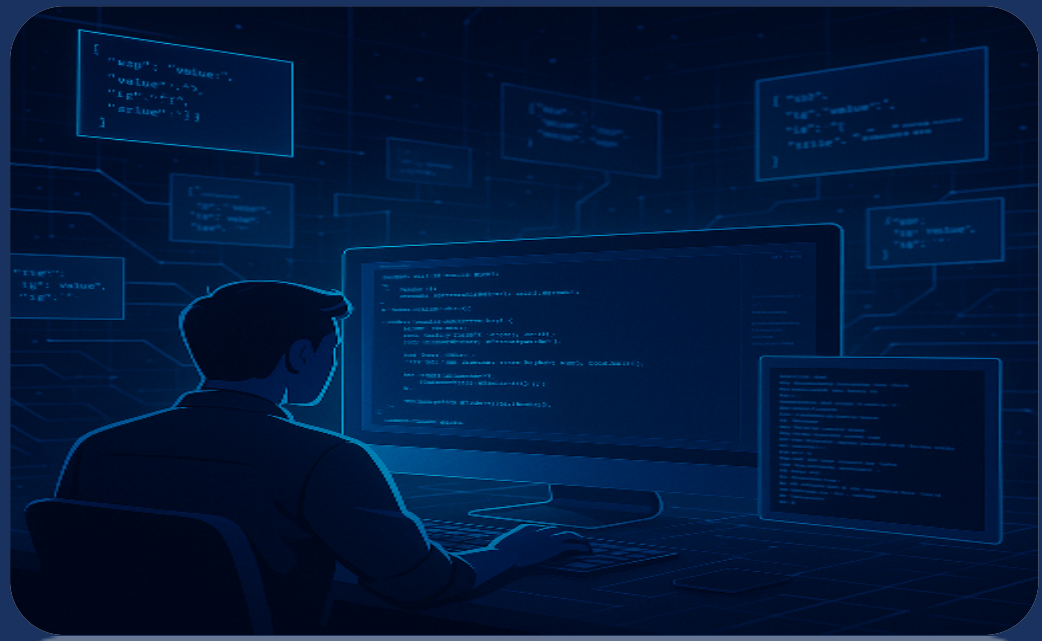


FUTURE ENHANCEMENTS:

- Real Database like SQL
- Advance algorithms for pathfinding like Dijkstra, AI and Image Processing
- Real Time Processing by Web Socket
- Cloud Deployment
- Frontend Integration to interact with the api

GITHUB LINK:

HTTPS://GITHUB.COM/NANDINI-3006/RNXG/TREE/MASTER



THANK YOU

BY

NANDINI