Development of Radar Coverage Analysis Microservice

Anand Prakash
adprakash2006@gmail.com

June 8, 2025

1 Overview of the Radar Coverage Analysis Microservice

The **Radar Coverage Analysis microservice** calculates radar coverage based on input parameters (e.g., transmit power, frequency) provided by an **Angular UI** and standard parameters (e.g., refractivity gradient) stored in a **PostgreSQL database**. The microservice exposes **REST APIs** to compute received power vs. distance and 3D coverage coordinates, which the Angular UI visualizes as 2D (power vs. distance) and 3D (coverage volume) plots. The system follows a **three-tier architecture**:

- **Client Tier**: Angular UI for user input and visualization.
- **Application Tier**: **Spring Boot** microservice for calculations and API endpoints.
- **Data Tier**: PostgreSQL database for storing standard radar parameters.

This explanation covers the development process, including architecture, implementation, and data flow, with a focus on clarity and technical detail.

2 System Architecture

The system is structured as follows:

1. **Angular UI (Client Tier)**:
   - Allows users to input radar parameters (e.g., transmit power, frequency, antenna gains).
   - Sends HTTP requests to the Spring Boot microservice.
   - Visualizes results using **Chart.js** for 2D plots and **Three.js** for 3D coverage.
   - Communicates with the backend via REST APIs.
2. **Spring Boot Microservice (Application Tier)**:
   - Exposes REST endpoints (e.g., `/api/radar/power`, `/api/radar/coverage`).
   - Retrieves standard parameters from the database.
   - Performs radar coverage calculations using the radar equation and atmospheric model.

Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

- Returns JSON responses with calculated data.

3. **PostgreSQL Database (Data Tier)**:
   - Stores standard radar parameters (e.g., refractivity gradient, minimum detectable signal).
   - Accessed by the microservice via **Spring Data JPA**.

3 Development Steps

3.1 Database Setup (PostgreSQL)

The database stores standard radar parameters to ensure consistency across calculations. A single table, `radar_parameters`, holds these values.

Schema:

```
CREATE TABLE radar_parameters (
    id SERIAL PRIMARY KEY,
    refractivity_gradient DOUBLE PRECISION DEFAULT -39e-6,
    min_detectable_signal DOUBLE PRECISION DEFAULT 1e-13,
    speed_of_light DOUBLE PRECISION DEFAULT 3e8
);
```

Initial Data:

```
INSERT INTO radar_parameters (refractivity_gradient,
   min_detectable_signal, speed_of_light)
VALUES (-39e-6, 1e-13, 3e8);
```

Purpose:

- `refractivity_gradient`: Models atmospheric effects on signal propagation.
- `min_detectable_signal`: Threshold for target detection.
- `speed_of_light`: Used to calculate wavelength.

Development Tasks:

- Install PostgreSQL and create the database.
- Use a tool like pgAdmin or DBeaver to execute the schema and insert initial data.
- Configure the Spring Boot application to connect to PostgreSQL.

3.2 Spring Boot Microservice Development

The microservice is built using **Spring Boot**, exposing REST APIs to handle radar calculations. It retrieves standard parameters from the database and combines them with user inputs from the Angular UI.

Project Structure:

Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
RadarMicroservice
 pom.xml
 src/main/java/com/xai/radar
     RadarApplication.java
     model
        RadarSystem.java
        AtmosphericModel.java
        RadarCoverageCalculator.java
        RadarParameters.java
     repository
        RadarParametersRepository.java
     service
        RadarService.java
     controller
        RadarController.java
     dto
         RadarInput.java
         PowerResponse.java
         CoverageResponse.java
 src/main/resources
     application.properties
```

pom.xml (Key Dependencies):

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.xai</groupId>
    <artifactId>radar-microservice</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.2</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.7.4</version>
        </dependency>
        <dependency>
            <groupId>gov.nist.math</groupId>
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
            <artifactId>jama</artifactId>
            <version>1.0.3</version>
        </dependency>
    </dependencies>
</project>
```

application.properties:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/radar_db
spring.datasource.username=admin
spring.datasource.password=secret
spring.jpa.hibernate.ddl-auto=update
server.port=8080
```

RadarParameters.java:

```
package com.xai.radar.model;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class RadarParameters {
    @Id
    private Long id;
    private double refractivityGradient;
    private double minDetectableSignal;
    private double speedOfLight;
    // Getters and setters
}
```

RadarSystem.java:

```
package com.xai.radar.model;

public class RadarSystem {
    private final double transmitPower; // watts
    private final double transmitGain;  // linear scale
    private final double receiveGain;   // linear scale
    private final double frequency;     // Hz
    private final double rcs;           // m^2
    private final double wavelength;

    public RadarSystem(double transmitPower, double transmitGain,
       double receiveGain,
                       double frequency, double rcs, double
                          speedOfLight) {
        this.transmitPower = transmitPower;
        this.transmitGain = transmitGain;
        this.receiveGain = receiveGain;
        this.frequency = frequency;
        this.rcs = rcs;
        this.wavelength = speedOfLight / frequency;
    }
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
    // Getters
}
```

AtmosphericModel.java:

```
package com.xai.radar.model;

public class AtmosphericModel {
    private final double refractivityGradient;

    public AtmosphericModel(double refractivityGradient) {
        this.refractivityGradient = refractivityGradient;
    }

    public double[] calculatePathLossExponent(double[] distances) {
        double[] pathLoss = new double[distances.length];
        for (int i = 0; i < distances.length; i++) {
            pathLoss[i] = 4 + refractivityGradient * distances[i];
            pathLoss[i] = Math.max(2, Math.min(6, pathLoss[i]));
        }
        return pathLoss;
    }
}
```

RadarCoverageCalculator.java:

```
package com.xai.radar.model;

import Jama.Matrix;

public class RadarCoverageCalculator {
    private final RadarSystem radar;
    private final AtmosphericModel atmosphere;
    private final double[] distances;
    private final double pi = Math.PI;
    private final double minDetectableSignal;

    public RadarCoverageCalculator(RadarSystem radar,
       AtmosphericModel atmosphere,
                                   double[] distances, double
                                      minDetectableSignal) {
        this.radar = radar;
        this.atmosphere = atmosphere;
        this.distances = distances;
        this.minDetectableSignal = minDetectableSignal;
    }

    public double[] calculateReceivedPower() {
        double[] pathLossExponent = atmosphere.
           calculatePathLossExponent(distances);
        double[] power = new double[distances.length];
        for (int i = 0; i < distances.length; i++) {
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
            power[i] = (radar.getTransmitPower() * radar.
               getTransmitGain() * radar.getReceiveGain() *
                    Math.pow(radar.getWavelength(), 2) * radar.
                       getRcs()) /
                    (Math.pow(4 * pi, 3) * Math.pow(distances[i],
                       pathLossExponent[i]));
        }
        return power;
    }

    public double getMaxRange(double[] receivedPower) {
        for (int i = receivedPower.length - 1; i >= 0; i--) {
            if (receivedPower[i] >= minDetectableSignal) {
                return distances[i];
            }
        }
        return 0;
    }

    public Matrix[] generateCoverageCoordinates(double maxRange) {
        int azimuthSteps = 360;
        int elevationSteps = 90;
        Matrix x = new Matrix(elevationSteps, azimuthSteps);
        Matrix y = new Matrix(elevationSteps, azimuthSteps);
        Matrix z = new Matrix(elevationSteps, azimuthSteps);

        double[] azimuth = linspace(0, 2 * pi, azimuthSteps);
        double[] elevation = linspace(0, pi / 2, elevationSteps);

        for (int i = 0; i < elevationSteps; i++) {
            for (int j = 0; j < azimuthSteps; j++) {
                x.set(i, j, maxRange * Math.sin(elevation[i]) * Math
                   .cos(azimuth[j]));
                y.set(i, j, maxRange * Math.sin(elevation[i]) * Math
                   .sin(azimuth[j]));
                z.set(i, j, maxRange * Math.cos(elevation[i]));
            }
        }
        return new Matrix[]{x, y, z};
    }

    private double[] linspace(double start, double end, int num) {
        double[] result = new double[num];
        double step = (end - start) / (num - 1);
        for (int i = 0; i < num; i++) {
            result[i] = start + i * step;
        }
        return result;
    }
}
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

RadarInput.java:

```
package com.xai.radar.dto;

public class RadarInput {
    private double transmitPower;
    private double transmitGain;
    private double receiveGain;
    private double frequency;
    private double rcs;
    // Getters and setters
}
```

PowerResponse.java:

```
package com.xai.radar.dto;

public class PowerResponse {
    private double[] distances;
    private double[] receivedPower;
    private double[] receivedPowerDb;
    private double minDetectableSignalDb;

    public PowerResponse(double[] distances, double[] receivedPower,
                         double[] receivedPowerDb, double
                            minDetectableSignalDb) {
        this.distances = distances;
        this.receivedPower = receivedPower;
        this.receivedPowerDb = receivedPowerDb;
        this.minDetectableSignalDb = minDetectableSignalDb;
    }
    // Getters and setters
}
```

CoverageResponse.java:

```
package com.xai.radar.dto;

public class CoverageResponse {
    private double[][] x;
    private double[][] y;
    private double[][] z;
    private double maxRange;

    public CoverageResponse(double[][] x, double[][] y, double[][] z
       , double maxRange) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.maxRange = maxRange;
    }
    // Getters and setters
}
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

RadarParametersRepository.java:

```
package com.xai.radar.repository;

import com.xai.radar.model.RadarParameters;
import org.springframework.data.jpa.repository.JpaRepository;

public interface RadarParametersRepository extends JpaRepository<
   RadarParameters, Long> {
}
```

RadarService.java:

```
package com.xai.radar.service;

import com.xai.radar.dto.CoverageResponse;
import com.xai.radar.dto.PowerResponse;
import com.xai.radar.dto.RadarInput;
import com.xai.radar.model.*;
import com.xai.radar.repository.RadarParametersRepository;
import org.springframework.stereotype.Service;
import Jama.Matrix;

@Service
public class RadarService {
    private final RadarParametersRepository repository;

    public RadarService(RadarParametersRepository repository) {
        this.repository = repository;
    }

    public PowerResponse calculatePower(RadarInput input) {
        RadarParameters params = repository.findById(1L)
                .orElseThrow(() -> new RuntimeException("Default
                   parameters not found"));
        RadarSystem radar = new RadarSystem(
                input.getTransmitPower(), input.getTransmitGain(),
                   input.getReceiveGain(),
                input.getFrequency(), input.getRcs(), params.
                   getSpeedOfLight());
        AtmosphericModel atmosphere = new AtmosphericModel(params.
           getRefractivityGradient());
        double[] distances = linspace(1e3, 3e6, 1000);
        RadarCoverageCalculator calculator = new
           RadarCoverageCalculator(
                radar, atmosphere, distances, params.
                   getMinDetectableSignal());
        double[] receivedPower = calculator.calculateReceivedPower()
           ;
        double[] receivedPowerDb = new double[receivedPower.length];
        for (int i = 0; i < receivedPower.length; i++) {
            receivedPowerDb[i] = 10 * Math.log10(receivedPower[i]);
        }
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
        double minDetectableSignalDb = 10 * Math.log10(params.
            getMinDetectableSignal());
        return new PowerResponse(distances, receivedPower,
            receivedPowerDb, minDetectableSignalDb);
    }

    public CoverageResponse calculateCoverage(RadarInput input) {
        RadarParameters params = repository.findById(1L)
                .orElseThrow(() -> new RuntimeException("Default
                    parameters not found"));
        RadarSystem radar = new RadarSystem(
                input.getTransmitPower(), input.getTransmitGain(),
                    input.getReceiveGain(),
                input.getFrequency(), input.getRcs(), params.
                    getSpeedOfLight());
        AtmosphericModel atmosphere = new AtmosphericModel(params.
            getRefractivityGradient());
        double[] distances = linspace(1e3, 3e6, 1000);
        RadarCoverageCalculator calculator = new
            RadarCoverageCalculator(
                radar, atmosphere, distances, params.
                    getMinDetectableSignal());
        double[] receivedPower = calculator.calculateReceivedPower()
            ;
        double maxRange = calculator.getMaxRange(receivedPower);
        Matrix[] coordinates = calculator.
            generateCoverageCoordinates(maxRange);
        return new CoverageResponse(
                coordinates[0].getArray(), coordinates[1].getArray()
                    , coordinates[2].getArray(), maxRange);
    }

    private double[] linspace(double start, double end, int num) {
        double[] result = new double[num];
        double step = (end - start) / (num - 1);
        for (int i = 0; i < num; i++) {
            result[i] = start + i * step;
        }
        return result;
    }
}
```

RadarController.java:

```
package com.xai.radar.controller;

import com.xai.radar.dto.CoverageResponse;
import com.xai.radar.dto.PowerResponse;
import com.xai.radar.dto.RadarInput;
import com.xai.radar.service.RadarService;
import org.springframework.web.bind.annotation.*;
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
@RestController
@RequestMapping("/api/radar")
public class RadarController {
    private final RadarService service;

    public RadarController(RadarService service) {
        this.service = service;
    }

    @PostMapping("/power")
    public PowerResponse getPowerVsDistance(@RequestBody RadarInput
       input) {
        return service.calculatePower(input);
    }

    @PostMapping("/coverage")
    public CoverageResponse getCoverageArea(@RequestBody RadarInput
       input) {
        return service.calculateCoverage(input);
    }
}
```

Development Tasks:

- Create the Spring Boot project using Spring Initializr with dependencies (Web, JPA, PostgreSQL, JAMA).
- Implement the model, repository, service, and controller classes as shown.
- Configure the database connection in `application.properties`.
- Build and run the application with `mvn spring-boot:run`.

3.3 Angular UI Development

The **Angular UI** allows users to input radar parameters, send them to the microservice, and visualize the results in 2D and 3D.

Project Structure:

```
RadarUI
 src
    app
       app.component.html
       app.component.ts
       app.module.ts
       radar.service.ts
       power-response.model.ts
       coverage-response.model.ts
 package.json
 angular.json
```

package.json (Key Dependencies):

Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
{
  "dependencies": {
    "@angular/core": "^18.0.0",
    "@angular/forms": "^18.0.0",
    "@angular/common": "^18.0.0",
    "@angular/http": "^8.0.0-beta.10",
    "chart.js": "^4.4.3",
    "three": "^0.167.0",
    "rxjs": "^7.8.0"
  }
}
```

power-response.model.ts:

```
export interface PowerResponse {
  distances: number[];
  receivedPower: number[];
  receivedPowerDb: number[];
  minDetectableSignalDb: number;
}
```

coverage-response.model.ts:

```
export interface CoverageResponse {
  x: number[][];
  y: number[][];
  z: number[][];
  maxRange: number;
}
```

radar.service.ts:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { PowerResponse } from './power-response.model';
import { CoverageResponse } from './coverage-response.model';

@Injectable({
  providedIn: 'root'
})
export class RadarService {
  private apiUrl = 'http://localhost:8080/api/radar';

  constructor(private http: HttpClient) {}

  calculatePower(input: any): Observable<PowerResponse> {
    return this.http.post<PowerResponse>(`${this.apiUrl}/power`,
       input);
  }

  calculateCoverage(input: any): Observable<CoverageResponse> {
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
    return this.http.post<CoverageResponse>(`${this.apiUrl}/coverage
       `, input);
  }
}
```

app.module.ts:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { ChartModule } from 'angular-highcharts';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, HttpClientModule, FormsModule,
     ChartModule],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

app.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { RadarService } from './radar.service';
import { PowerResponse } from './power-response.model';
import { CoverageResponse } from './coverage-response.model';
import * as Chart from 'chart.js';
import * as THREE from 'three';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit {
  radarInput = {
    transmitPower: 1e6,
    transmitGain: 30,
    receiveGain: 30,
    frequency: 10e6,
    rcs: 1
  };
  powerChart: Chart | undefined;
  scene: THREE.Scene | undefined;
  camera: THREE.PerspectiveCamera | undefined;
  renderer: THREE.WebGLRenderer | undefined;

  constructor(private radarService: RadarService) {}

  ngOnInit() {
    this.init3DScene();
  }
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
calculate() {
  this.radarService.calculatePower(this.radarInput).subscribe(
     power => {
    this.renderPowerChart(power);
  });
  this.radarService.calculateCoverage(this.radarInput).subscribe(
     coverage => {
    this.renderCoverage(coverage);
  });
}

private renderPowerChart(data: PowerResponse) {
  const ctx = document.getElementById('powerChart') as
     HTMLCanvasElement;
  this.powerChart = new Chart(ctx, {
    type: 'line',
    data: {
      labels: data.distances.map(d => (d / 1000).toString()),
      datasets: [
        {
          label: 'Received Power (dB)',
          data: data.receivedPowerDb,
          borderColor: 'blue',
          fill: false
        },
        {
          label: 'Min Detectable Signal (dB)',
          data: data.distances.map(() => data.
             minDetectableSignalDb),
          borderColor: 'red',
          borderDash: [5, 5],
          fill: false
        }
      ]
    },
    options: {
      responsive: true,
      scales: {
        x: { title: { display: true, text: 'Distance (km)' } },
        y: { title: { display: true, text: 'Power (dB)' } }
      }
    }
  });
}

private init3DScene() {
  this.scene = new THREE.Scene();
  this.camera = new THREE.PerspectiveCamera(75, window.innerWidth
     / window.innerHeight, 0.1, 10000);
  this.camera.position.z = 2000;
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
    this.renderer = new THREE.WebGLRenderer({ canvas: document.
       getElementById('coverageCanvas') as HTMLCanvasElement });
    this.renderer.setSize(window.innerWidth, window.innerHeight);
  }

  private renderCoverage(data: CoverageResponse) {
    if (!this.scene || !this.renderer || !this.camera) return;

    this.scene.clear();
    const geometry = new THREE.BufferGeometry();
    const positions: number[] = [];
    const indices: number[] = [];

    const x = data.x;
    const y = data.y;
    const z = data.z;

    for (let i = 0; i < x.length - 1; i++) {
      for (let j = 0; j < x[0].length - 1; j++) {
        const idx = i * x[0].length + j;
        positions.push(x[i][j] / 1000, y[i][j] / 1000, z[i][j] /
           1000);
        positions.push(x[i][j + 1] / 1000, y[i][j + 1] / 1000, z[i][
           j + 1] / 1000);
        positions.push(x[i + 1][j] / 1000, y[i + 1][j] / 1000, z[i +
            1][j] / 1000);
        positions.push(x[i + 1][j + 1] / 1000, y[i + 1][j + 1] /
           1000, z[i + 1][j + 1] / 1000);

        indices.push(idx * 4, idx * 4 + 1, idx * 4 + 2);
        indices.push(idx * 4 + 1, idx * 4 + 3, idx * 4 + 2);
      }
    }

    geometry.setAttribute('position', new THREE.
       Float32BufferAttribute(positions, 3));
    geometry.setIndex(indices);
    const material = new THREE.MeshBasicMaterial({ color: 0x00ffff,
       wireframe: true });
    const mesh = new THREE.Mesh(geometry, material);
    this.scene.add(mesh);

    this.renderer.render(this.scene, this.camera);
  }
}
```

app.component.html:

```
<div style="padding: 20px;">
  <h1>Radar Coverage Analysis</h1>
  <form>
    <div>
```


Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

```
      <label>Transmit Power (W):</label>
      <input type="number" [(ngModel)]="radarInput.transmitPower"
         name="transmitPower" required>
    </div>
    <div>
      <label>Transmit Gain:</label>
      <input type="number" [(ngModel)]="radarInput.transmitGain"
         name="transmitGain" required>
    </div>
    <div>
      <label>Receive Gain:</label>
      <input type="number" [(ngModel)]="radarInput.receiveGain" name
         ="receiveGain" required>
    </div>
    <div>
      <label>Frequency (Hz):</label>
      <input type="number" [(ngModel)]="radarInput.frequency" name="
         frequency" required>
    </div>
    <div>
      <label>RCS (m²):</label>
      <input type="number" [(ngModel)]="radarInput.rcs" name="rcs"
         required>
    </div>
    <button type="button" (click)="calculate()">Calculate</button>
  </form>

  <h2>Power vs. Distance</h2>
  <canvas id="powerChart"></canvas>

  <h2>3D Coverage Area</h2>
  <canvas id="coverageCanvas"></canvas>
</div>
```

Development Tasks:

- Create an Angular project with `ng new RadarUI`.
- Install dependencies: `npm install chart.js three`.
- Implement the service, models, and component as shown.
- Run the application with `ng serve`.

4 Complete Flow

1. **Client (Angular UI)**:
   - User inputs radar parameters (e.g., transmit power, frequency) via the form.
   - Clicking "Calculate" triggers HTTP POST requests to `/api/radar/power` and `/api/radar/coverage` with the input data.

Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

2. **API Gateway (Optional)**:
    - Routes requests to the Spring Boot microservice (not implemented here but can be added with **Spring Cloud Gateway**).
3. **Web Server (Spring Boot)**:
    - **Controller**: `RadarController` receives the POST request with `RadarInput`.
    - **Service**: `RadarService` retrieves standard parameters from the database, initializes `RadarSystem` and `AtmosphericModel`, and uses `RadarCoverageCalculator` to compute results.
    - **Repository**: `RadarParametersRepository` fetches standard parameters (e.g., refractivity gradient) from **PostgreSQL**.
    - **Response**: Returns JSON with power data or 3D coordinates.
4. **Database (PostgreSQL)**:
    - Stores standard parameters in the `radar_parameters` table.
    - Queried via Spring Data JPA to provide defaults like `min_detectable_signal`.
5. **Client (Angular UI)**:
    - Receives JSON responses.
    - Uses **Chart.js** to plot received power vs. distance (2D line chart).
    - Uses **Three.js** to render a 3D wireframe mesh of the coverage volume.

Example Flow:

- User enters: `transmitPower=1e6`, `transmitGain=30`, `receiveGain=30`, `frequency=10e6`, `rcs=1`.
- Angular sends POST to `/api/radar/power` with JSON:

```
{
  "transmitPower": 1000000,
  "transmitGain": 30,
  "receiveGain": 30,
  "frequency": 10000000,
  "rcs": 1
}
```

- Spring Boot retrieves `refractivityGradient`, `minDetectableSignal`, `speedOfLight` from the database.
- Calculates received power and returns:

```
{
  "distances": [1000, 3000, ..., 3000000],
  "receivedPower": [1.23e-10, 4.56e-11, ...],
  "receivedPowerDb": [-100.0, -103.4, ...],
  "minDetectableSignalDb": -130.0
}
```

- Angular renders a 2D chart with Chart.js and a 3D mesh with Three.js.

Anand Prakash, 08 Jun2025, adprakash2006@gmail.com

5 Additional Considerations

- **Security**: Add **Spring Security** with JWT authentication to protect endpoints. The Angular UI can include a login form to obtain tokens.
- **Error Handling**: Implement `@ExceptionHandler` in Spring Boot for cases like missing database records or invalid inputs. Display errors in the Angular UI.
- **Scalability**: Deploy the microservice in **Docker** containers and use **Kubernetes** for orchestration. Angular can be hosted on a static server like **Nginx**.
- **Validation**: Use `@Valid` in Spring Boot and Angular form validation to ensure input parameters are within realistic ranges (e.g., positive frequency).
- **Performance**: Cache database queries with **Spring Cache** for standard parameters. Optimize **Three.js** rendering by reducing mesh complexity.

6 Summary

- **Backend**: **Spring Boot** microservice with REST APIs (`/api/radar/power`, `/api/radar/covera` calculates radar coverage using user inputs and database parameters.
- **Database**: **PostgreSQL** stores standard parameters, accessed via **Spring Data JPA**.
- **Frontend**: **Angular UI** collects inputs, sends requests, and visualizes results with **Chart.js** (2D) and **Three.js** (3D).
- **Flow**: User inputs → Angular POSTs to Spring Boot → Service queries database and calculates → JSON response → Angular renders plots.

Anand Prakash, 08 Jun2025, adprakash2006@gmail.com