# Implementing Caching with Redis in a Spring Boot Application

## Project Overview

This project demonstrates the implementation of server-side caching using Redis in a Spring Boot application. The application fetches weather data and caches it to improve response times and reduce backend load. Redis serves as the caching layer, with caching set up to store weather information with a Time-To-Live (TTL) setting.

## Project Goals

1. Implement Server-Side Caching: Use Redis as a cache layer for weather data.  
2. Reduce Latency and Improve Performance: Cache responses to reduce repeated data fetching from primary sources.  
3. Enable Cache Invalidation: Apply TTL to cached data for automatic removal and avoid stale data.

## Technologies Used

- Spring Boot: For backend service and REST API development.  
- Redis: As the caching solution for improved data retrieval.  
- Java: Programming language.

## Theory and Key Concepts

### 1. Types of Caching

- Client-Side Cache: Stored on the client’s browser to reduce server requests.  
- Server-Side Cache: Stored on a centralized server; Redis is an example of this type.

### 2. Cache Invalidation Techniques

- Time-Based Invalidation (TTL): Automatically removes cache after a specified time.  
- LRU and LFU: Techniques that remove the least used cache items when storage is limited.

## Step-by-Step Implementation

1. Setup Spring Boot Project: Initialize a Spring Boot project with `spring-boot-starter-data-redis` dependency for Redis support.  
2. Configure Redis: Setup `RedisConfig` class to define the cache manager with TTL for cached entries.

@Bean  
public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {  
 RedisCacheConfiguration cacheConfig = RedisCacheConfiguration.defaultCacheConfig()  
 .entryTtl(Duration.ofMinutes(5)); // Set TTL to 5 minutes  
  
 return RedisCacheManager.builder(redisConnectionFactory)  
 .cacheDefaults(cacheConfig)  
 .build();  
}

3. Create the Weather API Endpoint: Develop a REST controller in `WeatherController` that provides weather data.  
4. Implement Caching in the Service Layer: Use `@Cacheable` in `WeatherService` to cache responses based on the `city` parameter.

@Cacheable(value = "weatherCache", key = "#city", unless = "#result == null")  
public String getWeather(String city) {  
 // Simulate delay and fetch weather data  
 simulateSlowService();  
 return "Weather in " + city + ": Sunny";  
}

5. Run and Test the Application: Start the application, access the weather endpoint, and check Redis cache with the Redis CLI commands below.

## Redis CLI Output

1. Starting Redis Server and Connecting:  
```bash  
root@Adira:~# sudo service redis-server start  
root@Adira:~# redis-cli  
```  
2. Checking Redis Cache Entries:  
- Pinging Redis to verify connection:  
```bash  
127.0.0.1:6379> ping  
PONG  
```  
- Observing cached entries after querying different cities:  
```bash  
127.0.0.1:6379> keys \*  
1) "weatherCache::NewYork"  
```  
- Multiple entries after additional requests:  
```bash  
127.0.0.1:6379> keys \*  
1) "weatherCache::NewYork"  
2) "weatherCache::India"  
```  
- Cache expiration in effect:  
```bash  
127.0.0.1:6379> keys \*  
(empty array)  
```

## Conclusion

This project demonstrates the benefits of caching for performance improvement and how Redis can be effectively integrated as a caching layer in Spring Boot. The application reduces latency by avoiding repeated data fetching, making the backend more efficient.