Semantic Search with Spring Boot & Redis

Raphael De Lio
10 min read
·
Apr 29, 2025

TL;DR:
*You're building a **semantic search app** using **Spring Boot** and **Redis**.*

*Instead of matching exact words, semantic search finds **meaning** using **Vector Similarity Search (VSS)**.*

*It works by turning movie synopses into **vectors** with **embedding models**, storing them in **Redis** (as a vector database), and finding the closest matches to user queries.*

Video: What is semantic search?

A traditional searching system works by matching the words a user types with the words stored in a database or document collection. It usually looks for exact or partial matches without understanding the meaning behind the words.

Semantic searching, on the other hand, shines because instead of just matching words, it tries to understand the meaning behind what the user is asking. It focuses on the concepts, not just the keywords, making it much easier for users to find what they really want.

In a movie streaming service, for example, if a movie's synopsis is stored in a database as **"A cowboy doll feels threatened when a**

new space toy becomes his owner's favorite," but the user searches for **"jealous toy struggles with new rival,"** a traditional search system might not find the movie because the exact words don't line up.

But a semantic a semantic search system can still connect the two ideas and bring up the right movie. It understands the *meaning* behind your query — not just the exact words.

Behind the scenes, this works thanks to **vector similarity search**. It turns text (or images, or audio) into vectors — lists of numbers —store them in a vector database and then finds the ones closest to your query.

Today, **we're gonna build a vector similarity search app that lets users find movies based on the *meaning* of their synopsis — not just exact keyword matches**. So that even if they don't know the title, they can still get the right movie based on a generic description of the synopsis.

To do that, we'll build a Spring Boot app from scratch and plug in **Redis OM Spring**. It'll handle turning our data into vectors, storing them in Redis, and running fast vector searches when users send a query.

Redis as a Vector Database

Video: [What is a vector database?]()

In the last 15 years, Redis became the foundational infrastructure for realtime applications. Today, with Redis 8, it's commited to becoming the foundational infrastructure for AI applications as well.

Redis 8 not only turns the community version of Redis into a Vector Database, but also makes it the fastest and most scalable database in the market today. **Redis 8 allows you to scale to one billion vectors without penalizing latency.**

Learn more: https://redis.io/blog/searching-1-billion-vectors-with-redis-8/

Redis OM Spring

To allow our users and customers to take full advantage of everything Redis can do — with the speed Redis is known for — we decided to implement **Redis OM Spring**, a library built on top of Spring Data Redis.

Redis OM Spring allows our users to easily communicate with Redis, model their entities as **JSONs** or Hashes, efficiently query them by levaraging the **Redis Query Engine** and even take advantage of probabilistic data structures such as **Count-min Sketch, Bloom Filters, Cuckoo Filters**, and more.

Redis OM Spring on GitHub: https://github.com/redis/redis-om-spring

Dataset

The dataset we'll be looking is a catalog of thousands of movies. Each of these movies has metadata such as its **title**, **cast**, **genre**, **year**, and **synopsis**. The JSON file representing this dataset can be found in the repository that accompanies this article.

Sample:

```json
{
  "title": "Toy Story",
  "year": 1995,
  "cast": [
   "Tim Allen",
   "Tom Hanks",
   "Don Rickles"
  ],
  "genres": [
   "Animated",
   "Comedy"
  ],
  "href": "Toy_Story",
  "extract": "Toy Story is a 1995 American computer-animated comedy film
directed by John Lasseter, produced by Pixar Animation Studios and released
by Walt Disney Pictures. The first installment in the  Toy Story franchise,
it was the first entirely computer-animated feature film, as well as the
first feature film from Pixar. It was written by Joss Whedon, Andrew Stanton,
Joel Cohen, and Alec Sokolow from a story by Lasseter, Stanton, Pete Docter,
and Joe Ranft. The film features music by Randy Newman, was produced by
Bonnie Arnold and Ralph Guggenheim, and was executive-produced by Steve Jobs
and Edwin Catmull. The film features the voices of Tom Hanks, Tim Allen, Don
Rickles, Jim Varney, Wallace Shawn, John Ratzenberger, Annie Potts, R. Lee
Ermey, John Morris, Laurie Metcalf, and Erik von Detten.",
  "thumbnail":
```

```
"https://upload.wikimedia.org/wikipedia/en/1/13/Toy_Story.jpg",
  "thumbnail_width": 250,
  "thumbnail_height": 373
}
```

Building the Application

Our application will be built using Spring Boot with Redis OM Spring. **It will allow movies to be searched by their synopsis based on semantic search rather than keyword matching.** Besides that, our application will also allow its users to perform **hybrid search**, **a technique that combines vector similarity with traditional filtering and sorting.**

0. GitHub Repository

The full application can be found on GitHub: https://github.com/redis/redis-om-spring/tree/main/demos/roms-vss-movies/

1. Add the required dependencies

From a Spring Boot application, add the following dependencies to your Maven or Gradle file:

```
<!-- Redis OM Spring for Redis object mapping and vector search -->
<dependency>
    <groupId>com.redis.om.spring</groupId>
    <artifactId>redis-om-spring</artifactId>
    <version>0.9.11</version>
</dependency>

<!-- Redis OM Spring uses Spring AI for creating embeddings (vectors) -->
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-openai</artifactId>
    <version>1.0.0-M6</version>
</dependency>
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-transformers</artifactId>
    <version>1.0.0-M6</version>
</dependency>
```

2. Define the `Movie` entity

Redis OM Spring provides two annotations that makes it easy to vectorize data and perform vector similarity search from within Spring Boot.

- `@Vectorize`: Automatically generates vector embeddings from the text field
- `@Indexed`: Enables vector indexing on a field for efficient search

The core of the implementation is the `Movie` class with Redis vector indexing annotations:

```
@RedisHash // This annotation is used by Redis OM Spring to store the entity
as a hash in Redis
public class Movie {

    @Id // IDs are automatically generated by Redis OM Spring as ULID
    private String title;

    @Indexed(sortable = true) // This annotation enables indexing on the
field for filtering and sorting
    private int year;

    @Indexed
    private List<String> cast;
```

```java
    @Indexed
    private List<String> genres;

    private String href;

    // This annotation automatically generates vector embeddings from the
text
    @Vectorize(
            destination = "embeddedExtract", // The field where the embedding
will be stored
            embeddingType = EmbeddingType.SENTENCE, // Type of embedding to
generate (Sentence, Image, face, or word)
            provider = EmbeddingProvider.OPENAI, // The provider for
generating embeddings (OpenAI, Transformers, VertexAI, etc.)
            openAiEmbeddingModel =
OpenAiApi.EmbeddingModel.TEXT_EMBEDDING_3_LARGE // The specific OpenAI model
to use for embeddings
    )
    private String extract;

    // This defines the vector field that will store the embeddings
    // The indexed annotation enables vector search on this field
    @Indexed(
            schemaFieldType = SchemaFieldType.VECTOR, // Defines the field
type as a vector
            algorithm = VectorField.VectorAlgorithm.FLAT, // The algorithm
used for vector search (FLAT or HNSW)
            type = VectorType.FLOAT32,
            dimension = 3072, // The dimension of the vector (must match the
embedding model)
            distanceMetric = DistanceMetric.COSINE, // The distance metric
used for similarity search (Cosine or Euclidean)
            initialCapacity = 10
    )
    private byte[] embeddedExtract;

    private String thumbnail;
    private int thumbnailWidth;
    private int thumbnailHeight;

    // Getters and setters...
}
```

In this example we're using OpenAI's embedding model that requires an **OpenAI API Key** to be set in the `application.properties` file of your application:

```
redis.om.spring.ai.open-ai.api-key=${OPEN_AI_KEY}
```

If an embedding model is not specified, Redis OM Spring will use a Hugging Face's Transformers model all-MiniLM-L6-v2 by default. In this case, make sure you match the number of dimensions in the indexed annotation to 384 which is the number of dimensions created by the default embedding model.

3. Repository Interface

A simple repository interface that extends `RedisEnhancedRepository`. This will be used to load the data into Redis using the `saveAll()` method:

```
public interface MovieRepository extends RedisEnhancedRepository<Movie,
String> {}
```

This provides basic CRUD operations for `Movie` entities, with the first generic parameter being the entity type and the second being the ID type.

4. Search Service

The search service uses two beans provided by Redis OM Spring:

- `EntityStream`: For creating a stream of entities to perform searches. **The Entity Stream must not be confused with the Java Streams API.** The Entity Stream will generate a Redis Command that will be sent to Redis so that Redis can perform the searching, filtering and sorting efficiently on its side.
- `Embedder`: Used for generating the embedding for the query sent by the user. It will be generated following the configuration of the `@Vectorize` annotation defined in the `Movie` class.

The search functionality is implemented in the `SearchService`:

```java
@Service
public class SearchService {

    private static final Logger logger =
LoggerFactory.getLogger(SearchService.class);
    private final EntityStream entityStream;
    private final Embedder embedder;

    public SearchService(EntityStream entityStream, Embedder embedder) {
        this.entityStream = entityStream;
        this.embedder = embedder;
    }

    public List<Pair<Movie, Double>> search(
            String query,
            Integer yearMin,
            Integer yearMax,
            List<String> cast,
            List<String> genres,
            Integer numberOfNearestNeighbors) {
        logger.info("Received text: {}", query);
        logger.info("Received yearMin: {} yearMax: {}", yearMin, yearMax);
        logger.info("Received cast: {}", cast);
        logger.info("Received genres: {}", genres);

        if (numberOfNearestNeighbors == null) numberOfNearestNeighbors = 3;
        if (yearMin == null) yearMin = 1900;
        if (yearMax == null) yearMax = 2100;

        // Convert query text to vector embedding
        byte[] embeddedQuery =
embedder.getTextEmbeddingsAsBytes(List.of(query), Movie$.EXTRACT).getFirst();

        // Perform vector search with additional filters
        SearchStream<Movie> stream = entityStream.of(Movie.class);
        return stream
                // KNN search for nearest vectors
                .filter(Movie$.EMBEDDED_EXTRACT.knn(numberOfNearestNeighbors,
embeddedQuery))
                // Additional metadata filters (hybrid search)
                .filter(Movie$.YEAR.between(yearMin, yearMax))
                .filter(Movie$.CAST.eq(cast))
                .filter(Movie$.GENRES.eq(genres))
                // Sort by similarity score
                .sorted(Movie$._EMBEDDED_EXTRACT_SCORE)
                // Return both the movie and its similarity score
                .map(Fields.of(Movie$._THIS, Movie$._EMBEDDED_EXTRACT_SCORE))
                .collect(Collectors.toList());
    }
}
```

Key features of the search service:

- Uses `EntityStream` to create a search stream for `Movie` entities
- Converts the text query into a vector embedding
- Uses K-nearest neighbors (KNN) search to find similar vectors
- Applies additional filters for hybrid search (combining vector and traditional search)
- Returns pairs of movies and their similarity scores

5. Movie Service for Data Loading

The `MovieService` handles loading movie data into Redis. It reads a JSON file containing movie date and save the movies into Redis.

It may take one or two minutes to load the data for the thousands of movies in the file because the embedding generation is done in the background. The `@Vectorize` annotation will generate the embeddings for the `extract` field before the movie is saved into Redis.

```
@Service
public class MovieService {

    private static final Logger log =
LoggerFactory.getLogger(MovieService.class);
    private final ObjectMapper objectMapper;
    private final ResourceLoader resourceLoader;
    private final MovieRepository movieRepository;

    public MovieService(ObjectMapper objectMapper, ResourceLoader
resourceLoader, MovieRepository movieRepository) {
        this.objectMapper = objectMapper;
        this.resourceLoader = resourceLoader;
        this.movieRepository = movieRepository;
    }

    public void loadAndSaveMovies(String filePath) throws Exception {
        Resource resource = resourceLoader.getResource("classpath:" +
filePath);
        try (InputStream is = resource.getInputStream()) {
            List<Movie> movies = objectMapper.readValue(is, new
TypeReference<>() {});
            List<Movie> unprocessedMovies = movies.stream()
                    .filter(movie ->
!movieRepository.existsById(movie.getTitle()) &&
                            movie.getYear() > 1980
                    ).toList();
            long systemMillis = System.currentTimeMillis();
            movieRepository.saveAll(unprocessedMovies);
            long elapsedMillis = System.currentTimeMillis() - systemMillis;
            log.info("Saved " + movies.size() + " movies in " + elapsedMillis
+ " ms");
        }
    }
    public boolean isDataLoaded() {
        return movieRepository.count() > 0;
    }
}
```

5. Search Controller

The REST controller exposes the search endpoint:

```
@RestController
public class SearchController {

    private final SearchService searchService;

    public SearchController(SearchService searchService) {
        this.searchService = searchService;
    }

    @GetMapping("/search")
    public Map<String, Object> search(
            @RequestParam(required = false) String text,
            @RequestParam(required = false) Integer yearMin,
            @RequestParam(required = false) Integer yearMax,
            @RequestParam(required = false) List<String> cast,
            @RequestParam(required = false) List<String> genres,
            @RequestParam(required = false) Integer numberOfNearestNeighbors
    ) {
        List<Pair<Movie, Double>> matchedMovies = searchService.search(
                text,
                yearMin,
                yearMax,
                cast,
                genres,
                numberOfNearestNeighbors
        );
        return Map.of(
                "matchedMovies", matchedMovies,
                "count", matchedMovies.size()
        );
    }
}
```

6. Application Bootstrap

The main application class initializes Redis OM Spring and loads data. The `@EnableRedisEnhancedRepositories` annotation activates Redis OM Spring's repository support:

```
@SpringBootApplication
@EnableRedisEnhancedRepositories(basePackages = {"com.redis.om.vssmovies*"})
public class Redis8DemoVectorSimilaritySearchApplication {

    public static void main(String[] args) {

SpringApplication.run(Redis8DemoVectorSimilaritySearchApplication.class,
args);
    }

    @Bean
    CommandLineRunner loadData(MovieService movieService) {
        return args -> {
            if (movieService.isDataLoaded()) {
                System.out.println("Data already loaded. Skipping data
load.");
                return;
            }
            movieService.loadAndSaveMovies("movies.json");
        };
    }
}
```

7. Sample Requests

You can make requests to the search endpoint:

```
GET http://localhost:8082/search?text=A movie about a young boy who goes to a
wizardry school

GET
http://localhost:8082/search?numberOfNearestNeighbors=1&yearMin=1970&yearMax=
1990&text=A movie about a kid and a scientist who go back in time

GET http://localhost:8082/search?cast=Dee Wallace,Henry Thomas&text=A boy who
becomes friend with an alien
```

Sample request:

```
GET
http://localhost:8082/search?numberOfNearestNeighbors=1&yearMin=1970&yearMax=
1990&text=A movie about a kid and a scientist who go back in time
```

Sample response:

```
{
  "count": 1,
  "matchedMovies": [
    {
      "first": { // matched movie
        "title": "Back to the Future",
        "year": 1985,
        "cast": [
          "Michael J. Fox",
          "Christopher Lloyd"
        ],
        "genres": [
          "Science Fiction"
        ],
        "extract": "Back to the Future is a 1985 American science fiction
film directed by Robert Zemeckis and written by Zemeckis, and Bob Gale. It
stars Michael J. Fox, Christopher Lloyd, Lea Thompson, Crispin Glover, and
Thomas F. Wilson. Set in 1985, it follows Marty McFly (Fox), a teenager
accidentally sent back to 1955 in a time-traveling DeLorean automobile built
by his eccentric scientist friend Emmett \"Doc\" Brown (Lloyd), where he
inadvertently prevents his future parents from falling in love – threatening
his own existence – and is forced to reconcile them and somehow get back to
the future.",
        "thumbnail":
"https://upload.wikimedia.org/wikipedia/en/d/d2/Back_to_the_Future.jpg"
      },
      "second": 0.463297247887 // similarity score (the lowest the closest)
    }
  ]
}
```

Wrapping up

And that’s it — you now have a working semantic search app using Spring Boot and Redis.

Instead of relying on exact keyword matches, your app understands the meaning behind the query. Redis handles the heavy part: embedding storage, similarity search, and even traditional filters — all at lightning speed.

With Redis OM Spring, you get an easy way to integrate this into your Java apps. You only need two annotations: `@Vectorize` and `@Indexed` and two Beans: `EntityStream` and `Embedder`.

Whether you’re building search, recommendations, or AI-powered assistants, this setup gives you a solid and scalable foundation.

Try it out, tweak the filters, explore other models, and see how far you can go!

More AI Resources

The best way to stay on the path of learning AI is by following the recipes available on the Redis AI Resources GitHub repository. There you can find dozens of recipes that will get you to start building AI apps, fast!