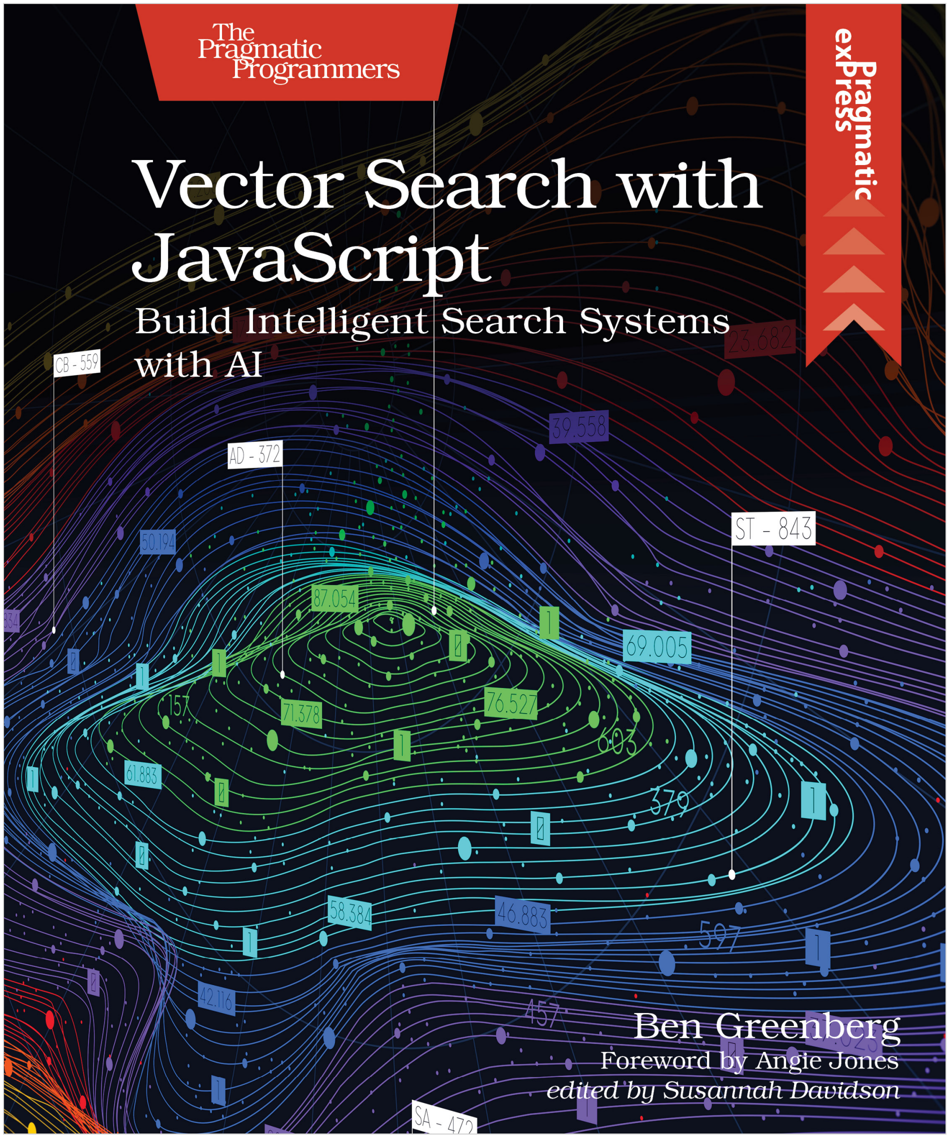

The Pragmatic Programmers
Pragmatic exPress
Vector Search with JavaScript
Build Intelligent Search Systems with AI
Ben Greenberg
Foreword by Angie Jones
edited by Susannah Davidson

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Generating Embeddings for User Queries

In the search service layer, the user query journey begins when the user enters a search term. Once the query is received, the service converts it into a vector representation using an embedding model. This transformation is crucial because it allows the system to compare the meaning behind the query with stored data. The query vector is then compared against the embeddings of articles in the database, using the dot product as the similarity metric.

The system then ranks the results based on the similarity scores generated by the comparison. Higher similarity scores indicate closer matches between the user's search term and the stored data. The system can return the top results to the user depending on how we set up the ranking algorithm. The system repeats this process whenever the user enters a new search query, ensuring the search results remain relevant and up-to-date.

The first step in the search process is to convert the user query into a vector embedding. The good news is that you have already become well-versed in this process! We have also set up the necessary skeleton for the search service in the services/ folder by scaffolding the file structure and defining the core functions.

We will use the functionality in embeddings/createEmbedding.js to generate the vector representation of the user query. search/embedQuery.js will call this function and handle the flow to the next step in the search process. We can reuse the existing embedding generation logic because we built createEmbedding to be versatile and flexible, as the process of generating an embedding is essentially the same regardless of the input source. The primary difference is what the final output of the createEmbedding function is, and we can see that in the following code snippet from that function:

implementing_vector_generation_service/createEmbedding.js

```
if (type === 'article') {
  await saveEmbedding(identifier, embedding, type);
  console.log(
    `Successfully saved embedding for article ID: ${identifier}`
  );
} else if (type === 'query') {
  console.log(
    `Generated embedding for query with ID: ${identifier}`
  );
  return embedding;
}
```

This snippet shows that the function can differentiate between an article and a query based on the type parameter. For an article, the system saves the embedding to the database using saveEmbedding, while it returns the embedding to the caller for a query. This flexibility allows us to seamlessly integrate the embedding generation process into the search service without duplicating code or creating unnecessary complexity.

If you're eager to test how query embeddings are generated before wiring them into the full search flow, you can run a quick CLI utility we've prepared for that purpose. Before running it, make sure you've set your OpenAI API key in a .env file at the root of the project, or in your shell environment using the variable OPENAI_API_KEY.

creating_vector_search_service/testEmbeddingCLI.js

```
import 'dotenv/config';
import readline from 'readline';
import { createEmbedding } from '../embeddings/createEmbedding.js';
import { ValidationError } from '../utils/errorHandlers.js';

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.question('Enter a search query: ', async (query) => {
  try {
    if (!query) {
      throw new ValidationError('Query content is missing', 'query');
    }

    const embedding = await createEmbedding('cli-test-query', query, 'query');
    console.log('\nFirst 5 dimensions of the embedding:\n');
    console.log(embedding.slice(0, 5).map(n => n.toFixed(4)).join(', ') + '...');
  } catch (error) {
    console.error('Error generating embedding:', error.message);
  } finally {
    rl.close();
  }
});
```

To run the CLI utility, execute the following command in your terminal:

```
bash node testEmbeddingCLI.js
```

Enter a search query when prompted. The script generates an embedding using the OpenAI API and prints the first few dimensions of the resulting vector. This gives you a quick, firsthand look at what embedding a query looks like before you integrate it into the broader search system.

After generating a query embedding from the command line, you can now integrate that functionality into the service layer. Build out the code in embedQuery.js to move one step closer to a fully functional vector search service.

creating_vector_search_service/embedQuery.js

```
import { createEmbedding } from '../embeddings/createEmbedding';
import { performSearch } from './performSearch';
import { ValidationError } from '../utils/errorHandlers';

/**
 * Embeds a user query and performs a search.
 *
 * @param {string} queryId - The unique ID for the query.
 * @param {string} query - The search query entered by the user.
 * @returns {Promise<object>} - The search results.
 */
const embedAndSearch = async (queryId, query) => {
  try {
    if (!query) {
      throw new ValidationError('Query content is missing', 'query');
    }

    const queryEmbedding = await createEmbedding(
      queryId, query, 'query'
    );
    const searchResults = await performSearch(queryEmbedding);

    return searchResults;
  } catch (error) {
    if (error instanceof ValidationError) {
      console.error(
        `Validation Error: ${error.message}, Field: ${error.field}`
      );
    } else {
      console.error(
        `Error: ${error.message}`
      );
    }
    throw error;
  }
};

export { embedAndSearch };
```

The embedAndSearch function is the entry point for embedding a user query and performing the vector search. First, it validates the query content to ensure it is not empty. If it passes validation, it calls createEmbedding to generate the query embedding and then passes the new embedding to performSearch to find the most relevant articles. If an error occurs during the process, the system catches and handles it appropriately to remain robust and user-friendly.

Speaking about validation, you may have noticed that we included a ValidationError class in the error handling, but we have not yet written the code for it. This will be a new custom error class similar to the others we have defined, like EmbeddingServiceError. Let's implement that now by re-opening the errorHandlers.js file in utils/ and adding the following code after the existing error classes:

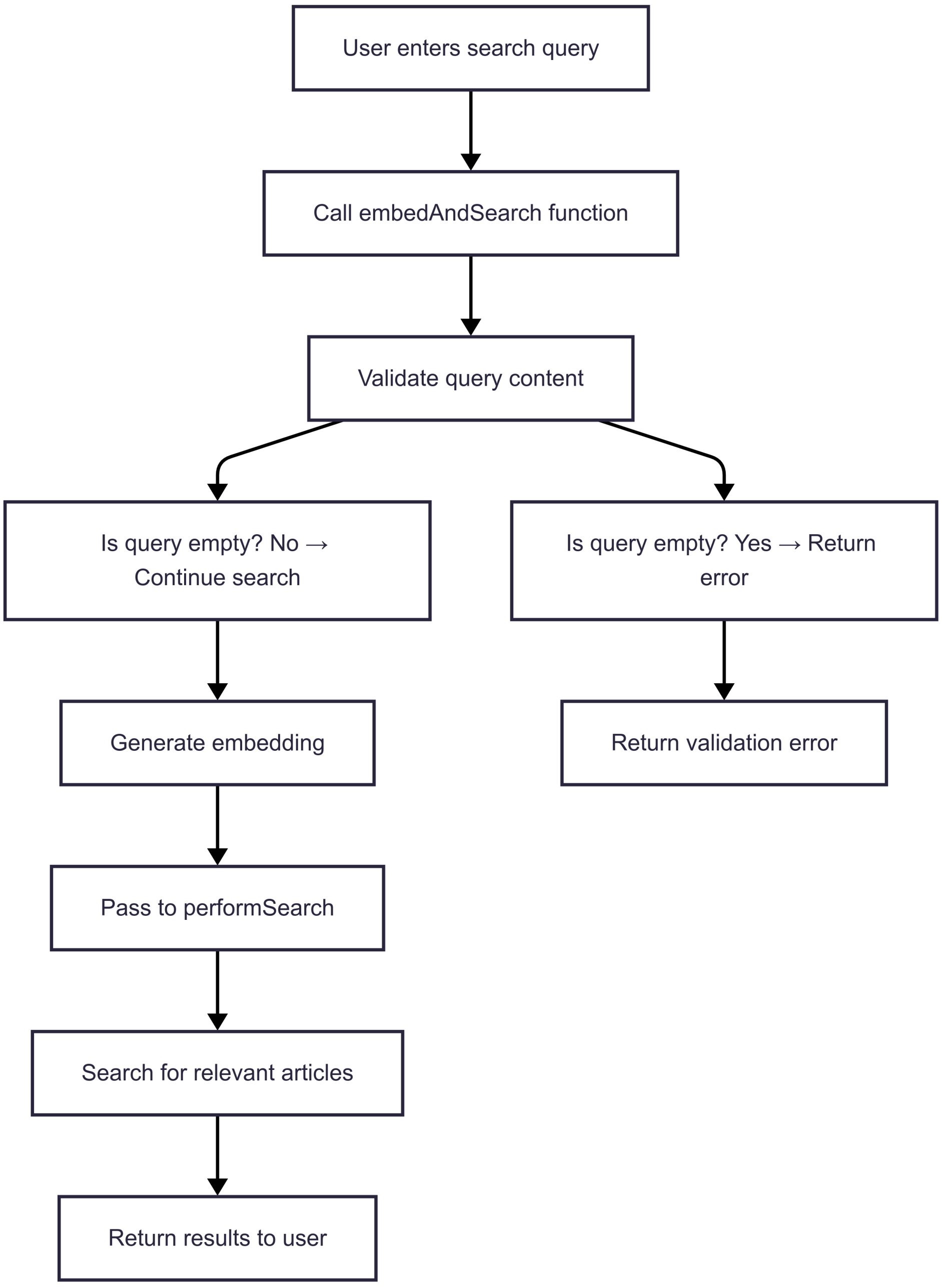
creating_vector_search_service/errorHandlers.js

```
class ValidationError extends Error {
    constructor(message) {
        super(message);
        this.name = 'ValidationError';
        Error.captureStackTrace(this, this.constructor);
    }
}
```

Make sure to add the new ValidationError to the export statement at the end of the file to make it accessible to other application parts like embedQuery.js. With this new error class in place, we can now handle validation errors in a more structured and consistent manner, improving the overall reliability of our system.

The code for embedQuery is lightweight because it leverages the composability of the existing services and functions we have built and will continue to build out. While vector search is a complex topic, implementing it in our codebases doesn't have to be equally complex.

Before we build out the logic for performing similarity searches using the dot product, let's quickly recap the flow of the user query embedding and search process from the perspective of the invoked functions.



- The user enters a search query, and the system calls the `embedAndSearch` function.
- The query content is validated to ensure it is not empty.

- The query is converted into a vector embedding using the `createEmbedding` function.
- The query embedding is passed to the `performSearch` function to find the most relevant articles.
- The search results are returned to the user, completing the search process.

As you can see, we are close to having a fully functional vector search service. The next step is to implement the similarity search logic at the heart of a vector search service.