MongoDB Press

<packt>

Building Modern Data Applications with MongoDB and .NET

Design smarter, scalable data systems with search and vector workflows using C# and .NET

Luce Carter

Building Modern Data Applications with MongoDB and .NET

Design smarter, scalable data systems with search and vector workflows using C# and .NET

Luce Carter

‹packt›

Building Modern Data Applications with MongoDB and .NET

Portfolio Director: Sunith Shetty
Relationship Lead: Sathya Mohan
Project Manager: Aniket Shetty and Sathya Mohan
Content Engineer: Siddhant Jain
Technical Editor: Shweta Amale and Aysha Nadeem
Copy Editor: Siddhant Jain
Indexer: Pratik Shirodkar
Proofreader: Siddhant Jain
Production Designer: Shantanu Zagade and Vijay Kamble
Growth Lead: Ankur Mulasi

First published: August 2026
Production reference: 1250826

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80778-613-7
www.packtpub.com

To my parents, who have always supported and believed in me,
even when they don't fully understand what I do or write about!

– Luce Carter

Contributors

About the author

Luce Carter is a Senior Developer Advocate at MongoDB, based in Manchester, UK. She is a big fan of .NET, always being open about how discovering C# led to her first real friends in her teens, which saved her from poor mental health. She loves learning and teaching what she learns. She is a big believer in meeting learners where they are with a learning style that suits them. She has previously published books on MongoDB and .NET, as well as taught workshops, created YouTube videos, livestreamed on Twitch, and written tutorials. Away from the screen, she can be found walking, travelling, playing squash, or occasionally playing video or board games. She is also in the 10-year club as a Microsoft Most Valuable Professional (MVP), as well as a Contentful Developer Hero.

As well as thanking my parents, I want to extend my gratitude to many others who made this possible. To my chosen family, both human and four-legged, thank you for the support, cuddles and purrs, unwavering support and pride in what I do, and for thinking writing a book is kind of cool! To Northern Rebound, thanks for being such a welcoming and wonderful community, even for someone with questionable squash skills but plenty of enthusiasm! To my godson Archie, you aren't born when I write this, but you will be soon, so I hope you know Auntie Luce was thinking of you and excited to meet you!

About the reviewers

Apoorva Joshi is currently a Staff AI Developer Advocate at MongoDB. She has a diverse engineering background with a bachelor's in electrical engineering, a master's in computer engineering, and several years of experience as a data scientist, applying AI to problems in the cybersecurity space. She now uses that applied AI expertise to help data science and engineering teams at large enterprises and startups build production-grade AI applications with MongoDB and Voyage AI.

Erik Hatcher, a Staff Developer Advocate for MongoDB Search, develops code and content for search-based applications. He co-authored *Lucene in Action*, the definitive guide to using the Lucene library. Lucene powers the core of Atlas Search and many other search engines that serve the majority of the world's applications these days. As a Lucene and open source enthusiast, Erik serves as a member of the Apache Software Foundation and a committer on the Lucene project. Prior to MongoDB, Erik co-founded Lucidworks, where he assisted in the development of many production search systems and helped build out an enterprise search platform.

Parker Faucher is a self-taught software engineer with over seven years of experience in technical education. He has authored more than 100 educational videos for MongoDB, establishing himself as a knowledgeable resource in database technologies. Currently, Parker focuses on AI and search technologies, exploring innovative solutions in these rapidly evolving fields. When not advancing his technical expertise, Parker enjoys spending quality time with his family and maintains an avid interest in collecting comic books.

Rishit Bhatia is a Senior Product Manager at MongoDB, where he leads developer experience for various language ecosystems. This includes, but is not limited to, the official MongoDB .NET/C# driver and the MongoDB Entity Framework Core provider. With a graduate degree from Carnegie Mellon University and a background in software engineering, he brings a hands-on technical perspective to product work. He is passionate about making developer-facing products intuitive, well-documented, and genuinely delightful to use.

Stanimira Vlaeva is a Staff Developer Advocate at MongoDB and a Google Developer Expert for Google Cloud. She is passionate about explaining complex technical topics in an understandable way, live-coding, and helping developers build with confidence.

Table of Contents

Preface

Modern applications are expected to do more than store and retrieve data. They need to answer richer questions, return relevant results quickly, adapt to changing user intent, and increasingly support AI-powered experiences that feel natural to use.

That shift has changed what we build as developers. It is no longer enough to think in terms of CRUD alone. We also need to think about data shaping, ranking, semantic meaning, and how all of that fits into a real application architecture.

This book is designed to help you build exactly that kind of application with MongoDB and .NET.

Together, we will build and evolve a running Blazor Server application called Leafy Library. We will start with the core building blocks of intelligent data processing using aggregation pipelines, then progressively add full-text search, autocomplete, facets, vector search, and production-minded best practices. Along the way, we will focus on practical implementation details with the MongoDB C# Driver, and keep each concept grounded in working code that you can apply to your own projects.

My goal with this book is simple: to share practical patterns that help you move from *it works* to *it works well in production*, while keeping the learning process approachable and enjoyable.

Who this book is for

This book is for intermediate .NET and C# developers who already understand the basics of working with MongoDB, including the document model and CRUD operations, and now want to build more intelligent, search-driven, and AI-aware applications. If you are building modern application features such as analytics dashboards, relevance-based search, semantic retrieval, or recommendation-based experiences, this book is for you.

What this book covers

Chapter 1, *Transform Your Data with the Aggregation Framework*, introduces aggregation fundamentals and explains how pipeline stages filter, reshape, and compute data directly in MongoDB.

Chapter 2, *Add Aggregations to the Application*, brings aggregation into Leafy Library with practical service-layer implementations and UI integration.

Chapter 3, *Best Practices for Aggregation Pipelines*, focuses on pipeline design choices that improve performance, maintainability, and clarity as your workloads grow.

Chapter 4, *Search Your Data with MongoDB Search*, introduces MongoDB Search concepts, index mappings, analyzers, and the mechanics behind relevance.

Chapter 5, *Add MongoDB Search to the Application*, integrates keyword search, autocomplete, and facets into Leafy Library using the MongoDB C# driver.

Chapter 6, *Best Practices for MongoDB Search*, explores how to tune indexes and queries for better relevance and performance in production scenarios.

Chapter 7, *Search Your Data Semantically with MongoDB Vector Search*, introduces semantic search, embeddings, and vector indexes, including modern embedding workflows.

Chapter 8, *Add Semantic Search to the Application*, applies vector search end to end in the application, from model updates to retrieval flow.

Chapter 9, *Best Practices for MongoDB Vector Search*, covers quality, latency, and cost trade-offs, including practical tuning strategies for real workloads.

Chapter 10, *Key Takeaways and Next Steps*, consolidates the core lessons from the book and provides practical guidance for continuing your implementation journey.

Chapter 11, *Appendix*, contains an in-depth explanation of the sample application, Leafy Library, including it's architecture and requirements for running it.

To get the most out of this book

To make your learning experience smoother, you should be comfortable with the following before you begin:

- C# and .NET application development fundamentals
- Basic MongoDB usage, including CRUD operations and common query patterns
- Running and debugging a local .NET application

For hands-on chapters, you will also need:

- A MongoDB cluster
- .NET SDK

Where relevant, each chapter includes technical requirements and setup notes before implementation begins.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://mdb.link/building-modern-data-applications-with-mongodb-codesnippets-repo`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing`. Check them out!

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book* section at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "While `$set` allows you to add or modify fields, `$unset` enables you to remove them."

A block of code is set as follows:

```
namespace Leafy_Library.Models;

public class LoanSummaryStats
{
    public int ActiveLoans { get; set; }
    public int Overdue { get; set; }
    public int DueSoon { get; set; }
    public int Returned { get; set; }
    public double? AvgReadingDays { get; set; }
}
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "On the **Indexes** tab for your collection, you will find a toggle to switch between standard indexes and search indexes."

> Warnings or important notes appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

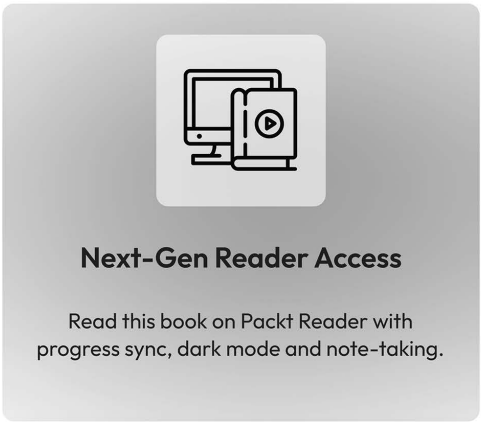
Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to Unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require one

Share your thoughts

Once you've read *Building Modern Data Applications with MongoDB and .NET*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1:

Advance Your Queries with Aggregations

In the first part of this book, you will develop a solid understanding of **MongoDB's aggregation framework** and its fundamental concepts. You will explore what the aggregation framework is, how it goes beyond simple queries and **create**, **read**, **update**, **and delete** (**CRUD**) operations, and how it enables powerful retrieval, computational, and transformational queries on your data. You will then apply these concepts to the sample library Blazor application, Leafy Library, to support meaningful features based on the user's loan history. Next, you will explore best practices for using aggregation pipelines and learn how to optimize not just the stages but also their order to build performant and powerful data-backed applications. By the end of this part of the book, you will have the knowledge needed to start querying and transforming your MongoDB data with confidence.

This part of the book includes the following chapters:

- *Chapter 1, Transform Your Data with the Aggregation Framework*
- *Chapter 2, Add Aggregations to the Application*
- *Chapter 3, Best Practices for Aggregation Pipelines*

1

Transform Your Data with the Aggregation Framework

Create, read, update, and delete (**CRUD**) are the most foundational operations that developers and database administrators expect from a database. These operations provide the ability to query information, update existing data, and remove data that is no longer required.

However, intelligent data applications often need to do more than simply retrieve or modify individual documents. In many cases, you might want to perform more advanced queries or compute values across multiple documents. For example, calculating averages, grouping data, or finding the highest or lowest values in the data.

Without native support for these operations, this kind of data processing would need to happen in application code after retrieving data from the database. This approach is inefficient because it pulls more data than necessary, increases latency, decreases performance, and places additional responsibility on the application, and the developers. Because these operations are both fundamental and common to working with data, it makes sense for a database such as MongoDB to provide them directly. This is where the aggregation framework comes in.

This chapter starts with aggregations because they are the foundation for the rest of the book. In later parts, you will build on these concepts when combining aggregation pipelines with full-text and semantic search to design modern data applications.

The following topics are covered in this chapter:

- Building aggregation pipelines for intelligent data processing
- Filtering and selecting data in aggregation pipelines
- Leveraging compute stages for analytics
- Using accumulator operators
- Enriching data with other specialized operations

Technical requirements

This chapter is primarily conceptual. The code snippets are included to illustrate key ideas in context, not as a required implementation exercise.

If you want to run the examples yourself, you will need the following:

- A MongoDB deployment (MongoDB Atlas or local) with the `books` collection loaded (the data is available in the `Data` folder of the sample repository; see *Chapter 11*, *Appendix* for more information).
- .NET 8 SDK or later.
- MongoDB C# Driver package (`MongoDB.Driver`).
- `mongosh` for running JSON-based aggregation examples.

The supporting code snippets for this chapter are available in the book's repository at `https://mdb.link/building-modern-data-applications-with-mongodb-codesnippets-repo`.

Building aggregation pipelines for intelligent data processing

In MongoDB, the aggregation framework is a very powerful feature. It was introduced in MongoDB 2.2 in 2012 and has continued to evolve significantly. You can think of the aggregation framework as an assembly line in a factory. The input to one section, or stage as it is called in the aggregation framework, is the output from the previous stage. Of course, if it is the first stage, then the input is the data stored in the database.

This design allows you to chain together multiple stages to carry out advanced queries and computations. These stages can transform data, filter it, group it, or compute new values before the results ever reach your application. In some cases, aggregation pipelines can also write results back to the database. In others, they simply reshape the data flowing through the pipeline, reducing the amount of processing required in your application code and allowing developers to focus on building features rather than handling data adjustments.

The great thing about MongoDB is that aggregation pipelines are defined using JSON syntax, just like the rest of the query language. This makes pipelines readable and expressive, without requiring knowledge of a separate query language.

Aggregation stages generally fall into one of the following categories:

- Filtering and selection stages
- Transformation stages and compute stages
- Enrichment stages

Over the course of this chapter, you will explore the different types of stages and how they affect the data as it flows through the pipeline. You will see how stages can filter, transform, group, and compute values, and how the output of one stage becomes the input to the next. Alongside the MongoDB syntax, you will also see equivalent examples written in C#.

Throughout this book, the C# examples primarily use the MongoDB .NET driver's strongly typed API and builder classes to define filters and pipeline stages. The builder classes are essentially helper methods that simplify building filters, aggregations, and managing indexes. This keeps queries readable and aligned with the domain model, making them easier to understand. However, for simpler queries in this chapter, lambda expressions are used where appropriate to improve clarity.

In *Chapter 2*, *Add Aggregations to the Application*, you will apply these concepts within the library application by building aggregation pipelines that demonstrate how to structure more intelligent data processing within a real-world context. In *Chapter 3*, *Best Practices for Aggregation Pipelines*, you will learn how to optimize aggregation pipelines to improve performance when working with larger datasets and more complex operations.

In *Part 2, Searching Your Data* and *Part 3, Semantically Searching Your Data*, you will learn how aggregations work alongside features such as full-text search and semantic search. While these capabilities can be used independently, combining them with aggregation pipelines allows you to create data-driven applications that go far beyond basic retrieval and updates.

Now that you understand what the aggregation framework is, it is time to explore the different categories of stages available and how these can be applied, starting with filtering and selecting data.

Filtering and selecting data in aggregation pipelines

As mentioned at the start of the chapter, CRUD is a foundational set of operations that should be available in any database. For this reason, the ability to filter and select data in your pipelines is very important. The aggregation framework provides dedicated stages that allow you to do exactly that within a pipeline.

In this section, you will examine what stages are available and how they can be used for filtering and shaping data, starting with the ability to filter data to retrieve only what you need from the database. In CRUD, this is of course *read*, but in the aggregation framework, this is known as `$match`.

> As you will see across the examples in this chapter, stages and operators always start with a $ sign.

$match

If you are familiar with SQL, or are used to the CRUD operations in MongoDB, `$match` is the equivalent of `SELECT … WHERE` in SQL, or `Find()` in MongoDB.

`$match` is used to filter documents in the pipeline. You can specify criteria that the documents must meet in order to be passed on to the next stage in the pipeline. You can match against one or more criteria, and the pipeline will execute against it. Any documents that don't meet these criteria will not be passed on to the next stage and will be removed from the pipeline.

Just like with the `Find()` query, `$match` uses standard MongoDB query syntax. This means you can filter your documents based on the value of fields, comparison operators, logical conditions, and more.

For example, imagine you have a collection of approximately 6,500 books in the library application, each with attributes such as title, author, year, and language. If, like me, you can only read English fluently, you can use `$match` to find all books in the library that are in English because there is a language field on the document. Only the matching book documents would then proceed through the rest of the pipeline.

```
{
  "$match": {
    "language": "en"
  }
}
```

With the MongoDB C# driver, you can call `Aggregate()` on your `IMongoCollection` instance that connects to the collection and performs these stages. It then has methods on it for the stages, including `Match()`.

```
List<Book> englishBooks = booksCollection
     .Aggregate()
    .Match(b => b.Language == "en").ToList();
```

This mirrors `Find()`, but as part of an aggregation pipeline.

If you want to find more recent books in English, you can carry out a slightly more complex `$match` against multiple fields, taking advantage of support for logical operators like greater than or equal to.

```
List<Book> englishBooks = booksCollection
     .Aggregate()
    .Match(b => b.Language == "en" && b.Year >= 2000).ToList();
```

At this point, only documents that match these criteria will continue in the pipeline.

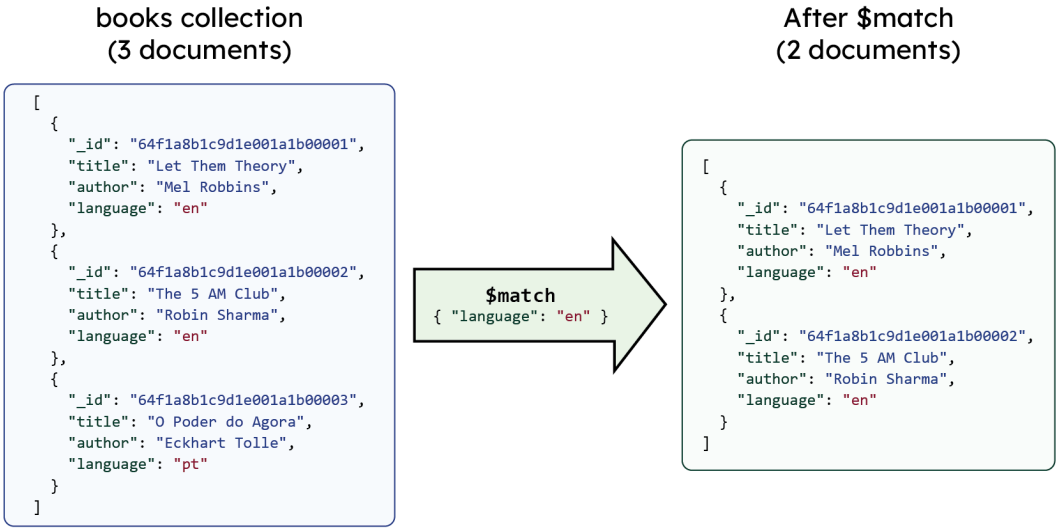


Figure 1.1 - $match stage in the aggregation framework

$project

After filtering documents with `$match`, the next most common requirement is selecting or reshaping the data that flows through the pipeline. This is the role of `$project`.

You can think of `$match` as determining which documents continue through the pipeline and `$project` as determining what each document looks like as it moves forward.

The `$project` stage controls which fields are included in the output and how those fields are shaped. This makes the `$project` stage very useful when you want to reduce the amount of data returned, shape the output for an API response, or even prepare documents for further transformation in later stages.

The following is what a `$project` stage looks like when passed as a JSON object to a tool such as `mongosh`, MongoDB's CLI tool. In this example, you specify that you want to exclude the unique `_id` field and only keep the title, author, year, and pages fields in the document going forward. This doesn't impact the documents stored in the database, it simply shapes the documents in the pipeline.

```
{
  "$project": {
    "_id": 0,
    "title": 1,
    "author": 1,
    "year": 1,
    "pages": 1
  }
}
```

In C#, projection is expressed naturally using the strongly typed API.

```
var projectedBooks = booksCollection
    .Aggregate()
    .Project(b => new
    {
        b.Title,
        b.Author,
        b.Year,
        b.Pages
    })
    .ToList();
```

Here, the documents flowing through the pipeline are reshaped to include only the specified properties.

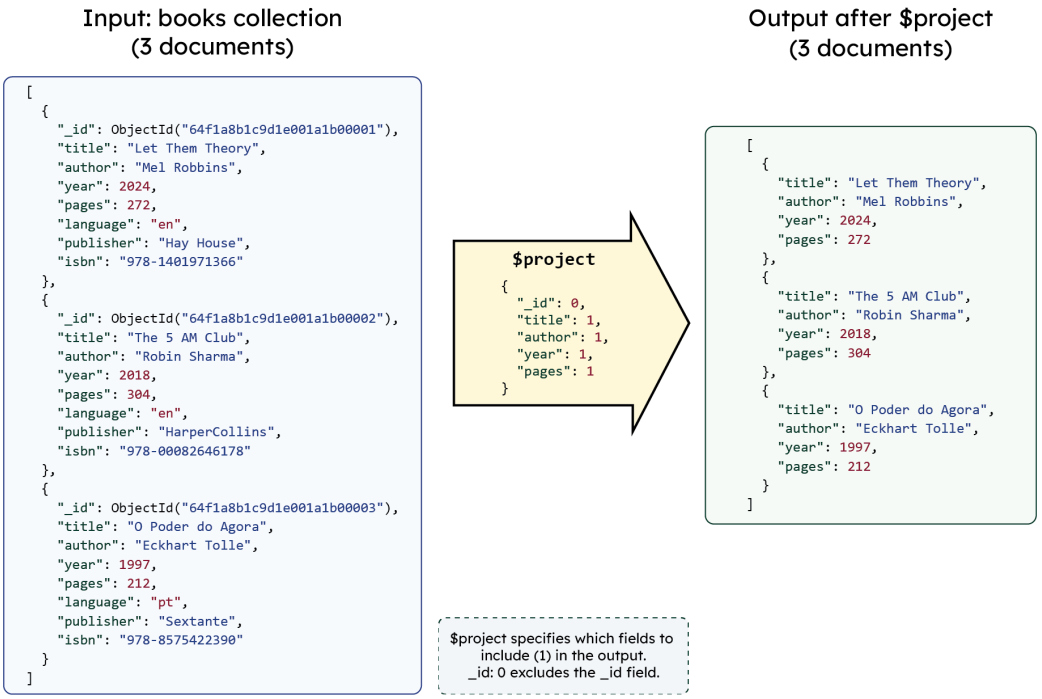


Figure 1.2 - $project in aggregation framework

$sort

After filtering and shaping your data, you may want to control the order in which documents are returned. This is the role of `$sort`. The `$sort` stage arranges documents in ascending or descending order based on one or more fields. Sorting is commonly used for:

- Ordering results alphabetically
- Preparing data for pagination
- Identifying the highest or lowest values

In the library, you might want to sort books by their release year, with the most recently released books appearing first.

When writing MongoDB syntax in JSON, `-1` is used to specify descending order and `1` is used for ascending order.

```
{
  "$sort": {
    "year":  -1
  }
}
```

In C#, the driver provides two helpful methods, `SortByDescending` or `SortBy` to make it more readable and easier to find the methods you might want.

```
var sortedBooks = booksCollection
    .Aggregate()
    .SortByDescending(b => b.Year)
    .ToList();
```

As mentioned earlier, you can sort based on one or more fields, so it's possible to sort not just by the year of release but also by the title in alphabetical order.

```
var sortedBooks = booksCollection
    .Aggregate()
    .SortByDescending(b => b.Year)
    .SortBy(b => b.Title)
    .ToList();
```

The documents in this new sorted order will then continue to the next stage of the pipeline.

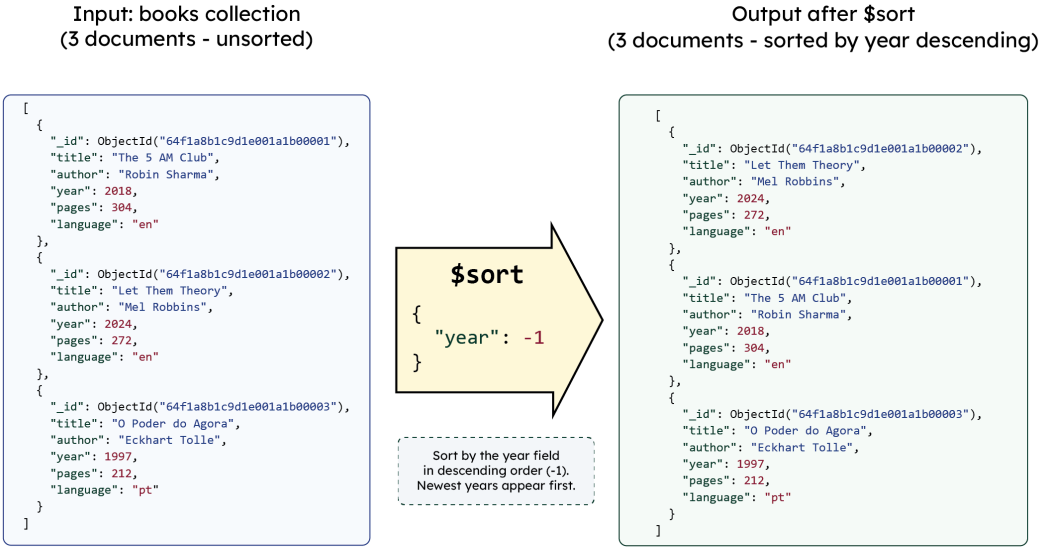


Figure 1.3 - $sort in aggregation framework

$skip

In many applications, especially those with user interfaces, you may not want to return every matching document at once. `$skip` allows you to skip a certain number of documents in the pipeline.

It is commonly used alongside `$limit` and `$sort` to achieve pagination.

```
{
  "$skip": 20
}
```

In C#, you can achieve the same result using the `Skip()` method:

```
var pagedBooks = booksCollection
    .Aggregate()
     .Skip(20)
    .ToList();
```

This skips the first 20 documents currently flowing through the pipeline.

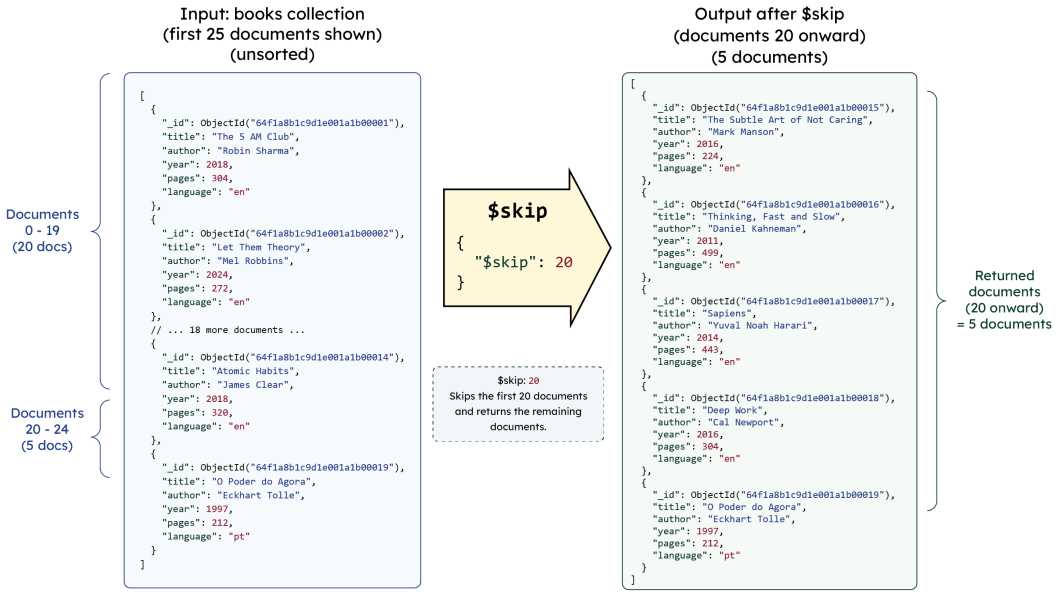


Figure 1.4 - $skip in aggregation framework

$limit and $sort

The `$limit` stage restricts the number of documents that are allowed to continue through the aggregation pipeline. It is commonly used to return only the top N results, implement pagination, or prevent unnecessarily large result sets from being returned, thereby improving both performance and efficiency.

The following example limits the pipeline output to the first 10 documents:

```
{
  "$limit": 10
}
```

As you can see, if you're using a tool such as `mongosh`, the JSON to apply a `$limit` stage is very simple. The same can be said for C#.

```
var limitedBooks = booksCollection
    .Aggregate()
    .Limit(10)
    .ToList();
```

This limits the results to 10 documents. You could combine this with `$sort` to find the top 10 books in the library with the most available copies.

```
var tenMostAvailableBooks = booksCollection
    .Aggregate()
    .SortByDescending(b => b.Available)
    .Limit(10)
    .ToList();
```

This means that only the top 10 most available books are passed to the next stage of the pipeline, or returned as the output if it is the last stage. You could then use other operators, which will be covered later in the chapter, to carry out analysis such as identifying which book is loaned out the most or the average number of times each book is loaned, to help adjust stock levels. Perhaps if there are too many copies of these books, they can be donated to a school or another organization.

In addition to filtering and selecting documents, aggregation pipelines can also transform documents as they pass through the pipeline. These transformations do not modify the stored data but instead reshape the documents used by later stages.

$set

So far, you have looked at filtering documents with `$match` and reshaping them with `$project`. Another common requirement when working with data is adding new fields or modifying existing values as documents flow through the pipeline. This is the role of `$set`.

The `$set` stage allows you to add new fields to each document or overwrite existing ones. It is important to note that this does not update the document stored in the collection. Instead, the transformation only applies to the documents as they move through the pipeline.

You may also see `$addFields` used in place of `$set`. In the aggregation framework, these two stages are interchangeable and perform the same function.

You might use `$set` to:

- Add computed fields
- Create flags based on conditions
- Prepare documents for grouping or further analysis
- Simplify logic that would otherwise reside in application code

For example, you may want to identify whether a book should be considered “modern” based on its publication year.

```
{
 "$set": {
      "isModern": { "$gte": ["$year", 2000] }
  }
}
```

In this case, a new Boolean field called `isModern` is added to each document in the pipeline. The value, `true` or `false`, is calculated based on the year field.

However, because C# is strongly typed, the result shape must still be compatible with the type being returned. If you introduce new fields that do not exist in your `Book` class, you typically project into a new type that includes those properties.

```
public class BookWithFlags
{
    public string Title { get; set; } = null!;
    public string Author { get; set; } = null!;
    public int Year { get; set; }
    public bool IsModern { get; set; }
}

var booksWithFlags = booksCollection
    .Aggregate()
    .Project(b => new BookWithFlags
    {
        Title = b.Title,
        Author = b.Author,
        Year = b.Year,
        IsModern = b.Year >= 2000
    })
    .ToList();
```

Here, the pipeline computes a new value and projects it into a new type that contains the additional property. The original `Book` documents stored in the database remain unchanged.

The C# driver also provides a `Set()` stage that directly represents `$set` in the aggregation framework.

```
var booksWithFlag = booksCollection
    .Aggregate()
    .Set(b => new
    {
        IsModern = b.Year >= 2000
    })
    .ToList();
```

In strongly typed pipelines, you may still need to project into a compatible result type if the output shape differs from the original `Book` model.

Both approaches represent the same underlying pipeline behavior. The choice depends on whether you are primarily reshaping results (`Project()`) or modeling the aggregation stage more explicitly (`Set()`).

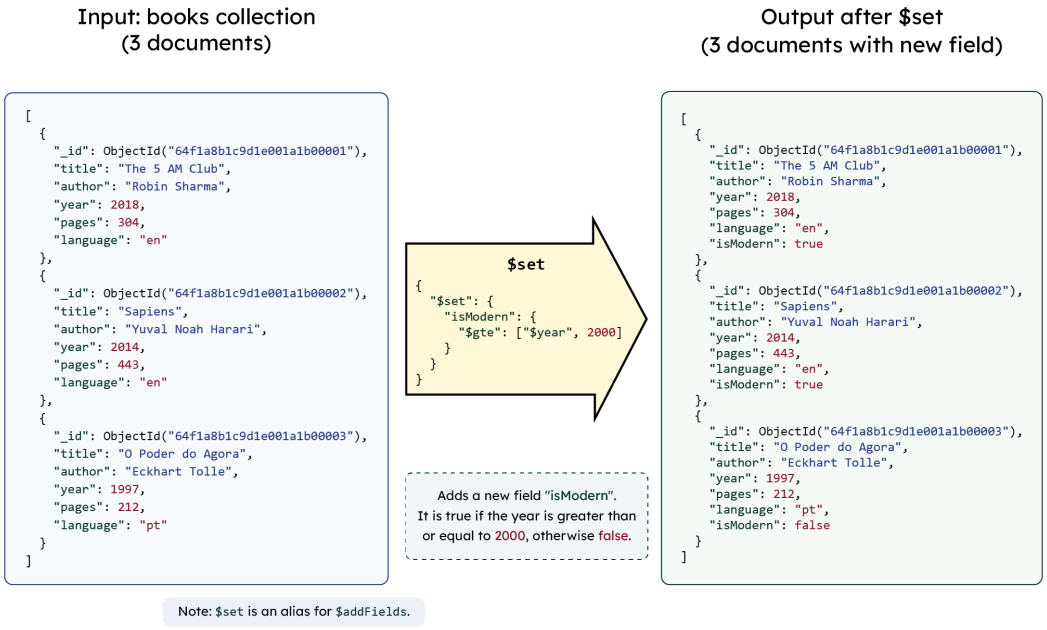


Figure 1.5 - $set in the aggregation framework

$unset

While `$set` allows you to add or modify fields, `$unset` enables you to remove them. The `$unset` stage removes one or more fields from documents flowing through the pipeline. Like `$set`, this affects only the pipeline output and does not change the documents in the collection.

You might use `$unset` to:

- Exclude sensitive fields, such as addresses or phone numbers
- Remove internal metadata
- Simplify documents before returning them

Achieving `$unset` is straightforward. If you are using JSON in languages like JavaScript or tools like `mongosh` that accept JSON, you simply call `$unset` and pass it the fields to be removed from the documents in the pipeline.

```
{
  "$unset": { "plot": 0}
}
```

In strongly typed C#, removing fields is typically expressed through projection by selecting only the fields you want to retain.

```
var cleanedBooks = booksCollection
    .Aggregate()
    .Project(b => new
    {
        b.Id,
        b.Title,
        b.Author,
        b.Year
    })
    .ToList();
```

Up to this point, the stages you have seen primarily control which documents move through the pipeline and what they look like. The next category of stages allows you to perform calculations across groups of documents.

Leveraging compute stages for analytics

Filtering and selecting data is often the first step. In many real-world applications, you eventually need to compute values across sets of documents. For example, you might find the average rating of books per author, count how many books exist per language, or identify the oldest and newest books in a particular category.

In the aggregation framework, these kinds of operations are handled through stages that group data together and apply calculations across those groups. The most important stage for this is `$group`.

> Operators such as `$avg`, `$min`, and `$max` are not stages themselves. They are accumulator operators, and are most commonly used inside stages such as `$group`.

$count

Sometimes, you don't need the actual documents themselves but instead the total number of documents in the database or the number of documents that match specific criteria. This is where `$count` comes in. It returns a single document containing the total number of documents currently in the pipeline.

```
{
  "$count": "totalBooks"
}
```

The use of `totalBooks` here is just a variable name. You can name it whatever you want. However, when the document is returned, it will contain a field matching that name with the count value.

```
var countResult = booksCollection
    .Aggregate()
    .Match(b => b.Language == "en")
    .Count()
    .FirstOrDefault();

long totalBooks = countResult?.Count ?? 0;
```

Here, `$count` is being used after filtering with `$match` to return the number of books in English.

$group

The $group stage allows you to group documents together based on one or more fields and then compute values for each group. Conceptually, you can think of it as first deciding what defines a group, such as an author, and then calculating one or more values for each group, such as a count, average, minimum, or maximum.

A common example in a library system is grouping books by author and counting how many books each author has in the catalog.

```
{
  "$group": {
    "_id": "$author",
    "bookCount": { "$sum": 1 }
  }
}
```

Here, _id represents the grouping key. Each distinct author becomes a group, and bookCount increments by 1 per document.

For simple grouping and calculations, using the strongly typed API is very readable.

```
public class BooksPerAuthor
{
    public string Author { get; set; } = null!;
    public int BookCount { get; set; }
}

var results = booksCollection
    .Aggregate()
    .Group(
        b => b.Author,
        g => new BooksPerAuthor
        {
            Author = g.Key,
            BookCount = g.Count()
        })
    .ToList();
```

As you can see, this is very readable and shows how you can use `$group` with the C# driver to transform raw documents into meaningful summaries by aggregating data across values. It's your go-to tool for analytics-style queries such as counts.

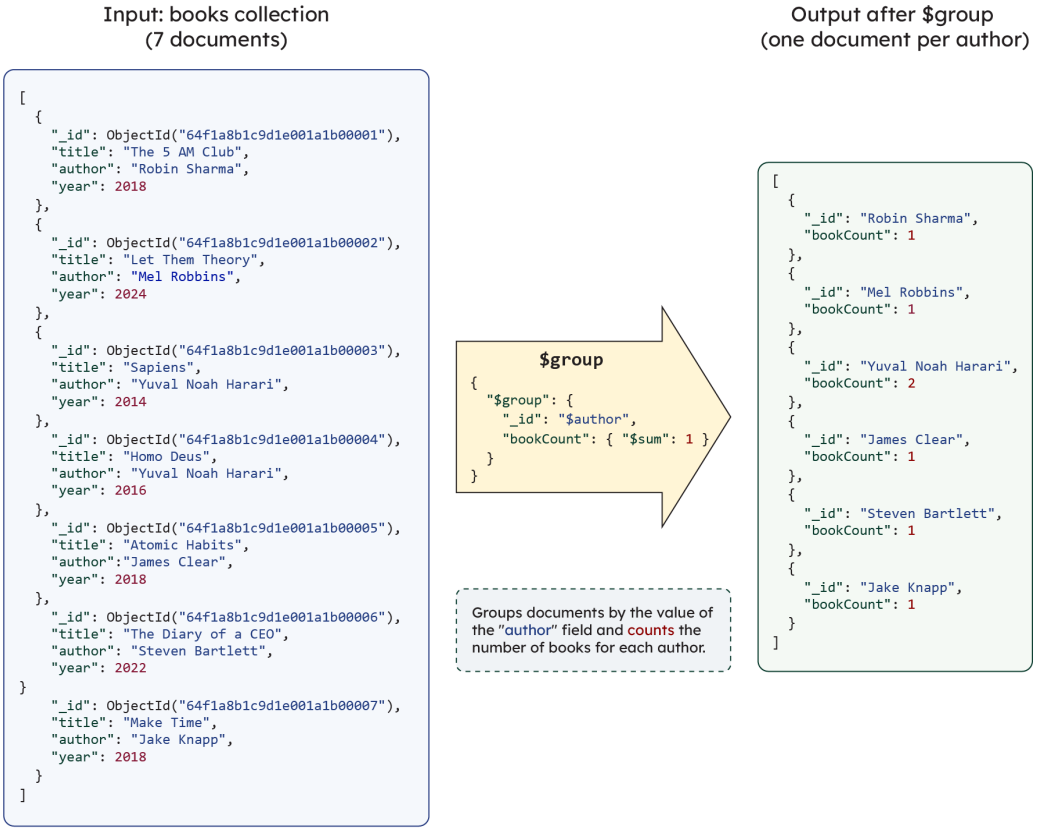


Figure 1.6 - $group in aggregation framework

Using accumulator operators

Once you understand how the `$group` stage works, the next step is understanding accumulator operators. These operators are used to define the computation that should be performed for each group.

Accumulator operators are most commonly used inside the `$group` stage. The stage defines how documents are grouped, and the accumulator operators define what should be calculated for each group.

Some of the most common accumulator operators are:

- `$sum`: Adds values together, or counts documents when used as `{ "$sum": 1 }`
- `$avg`: Calculates the average value
- `$min`: Returns the smallest value in the group
- `$max`: Returns the largest value in the group
- `$first`/`$last`: Returns the first or last value based on the current sort order
- `$push`: Collects values into an array (including duplicates)
- `$addToSet`: Collects unique values into an array (removes duplicates)

You will see these operators used throughout real-world pipelines because they cover most analytics requirements: counts, totals, averages, ranges, and also "show me the values in each group."

For example, if you want the average publication year per author, along with the earliest and latest year, you could use `$avg`, `$min`, and `$max`.

```
{
  "$group": {
    "_id": "$author",
    "avgYear": { "$avg": "$year" },
    "minYear": { "$min": "$year" },
    "maxYear": { "$max": "$year" }
  }
}
```

The following code example uses the `Group()` stage with standard C# aggregation methods and gives you a single result per author, with computed values based on all of that author's books.

```
public class AuthorYearStats
{
    public string Author { get; set; } = null!;
    public double AvgYear { get; set; }
    public int MinYear { get; set; }
    public int MaxYear { get; set; }
}

var results = booksCollection
    .Aggregate()
    .Group(
```

```
        b => b.Author,
        g => new AuthorYearStats
        {
            Author = g.Key,
            AvgYear = g.Average(x => x.Year),
            MinYear = g.Min(x => x.Year),
            MaxYear = g.Max(x => x.Year)
        })
    .ToList();
```

If `year` can be `null` in your data, you will typically filter those out first using `$match` (or handle nulls in your projection), otherwise, your averages and min/max calculations may not behave as expected.

```
{
    "$match": {
      "year": { "$ne": null }
    }
  },
  {
    "$group": {
      "_id": "$author",
      "avgYear": { "$avg": "$year" },
      "minYear": { "$min": "$year" },
      "maxYear": { "$max": "$year" }
    }
}
```

This can also be easily applied using the C# driver and the standard C# syntax of checking for null inside the `Match()` call.

```
var results = booksCollection
    .Aggregate()
    .Match(b => b.Year != null)
    .Group(
        b => b.Author,
        g => new AuthorYearStats
        {
```

```
            Author = g.Key,
            AvgYear = g.Average(x => x.Year),
            MinYear = g.Min(x => x.Year),
            MaxYear = g.Max(x => x.Year)
        })
    .ToList();
```

Another common pattern is using `$sum` for counts. When you use `{ "$sum": 1 }`, MongoDB increments by one for every document in the group.

```
{
  "$group": {
    "_id": "$language",
    "bookCount": { "$sum": 1 }
  }
}
```

As ever, the MongoDB C# driver provides a more expressive and type-safe way to achieve the same result.

```
public class CountByLanguage
{
    public string Language { get; set; } = null!;
    public int BookCount { get; set; }
}

var results = booksCollection
    .Aggregate()
    .Group(
        b => b.Language,
        g => new CountByLanguage
        {
            Language = g.Key,
            BookCount = g.Count()
        })
    .ToList();
```

Conceptually, both approaches are identical; they group documents by a field and count how many documents fall into each group. The difference lies in how you express the operation, raw aggregation syntax versus a strongly-typed API.

Counting with `$sum:1` is one of the most common aggregation patterns and forms the foundation for many analytics queries. Thanks to the availability of `Count()` in the C# driver's strongly-typed API, the underlying behavior is the same but with a clear understanding of what is being carried out.

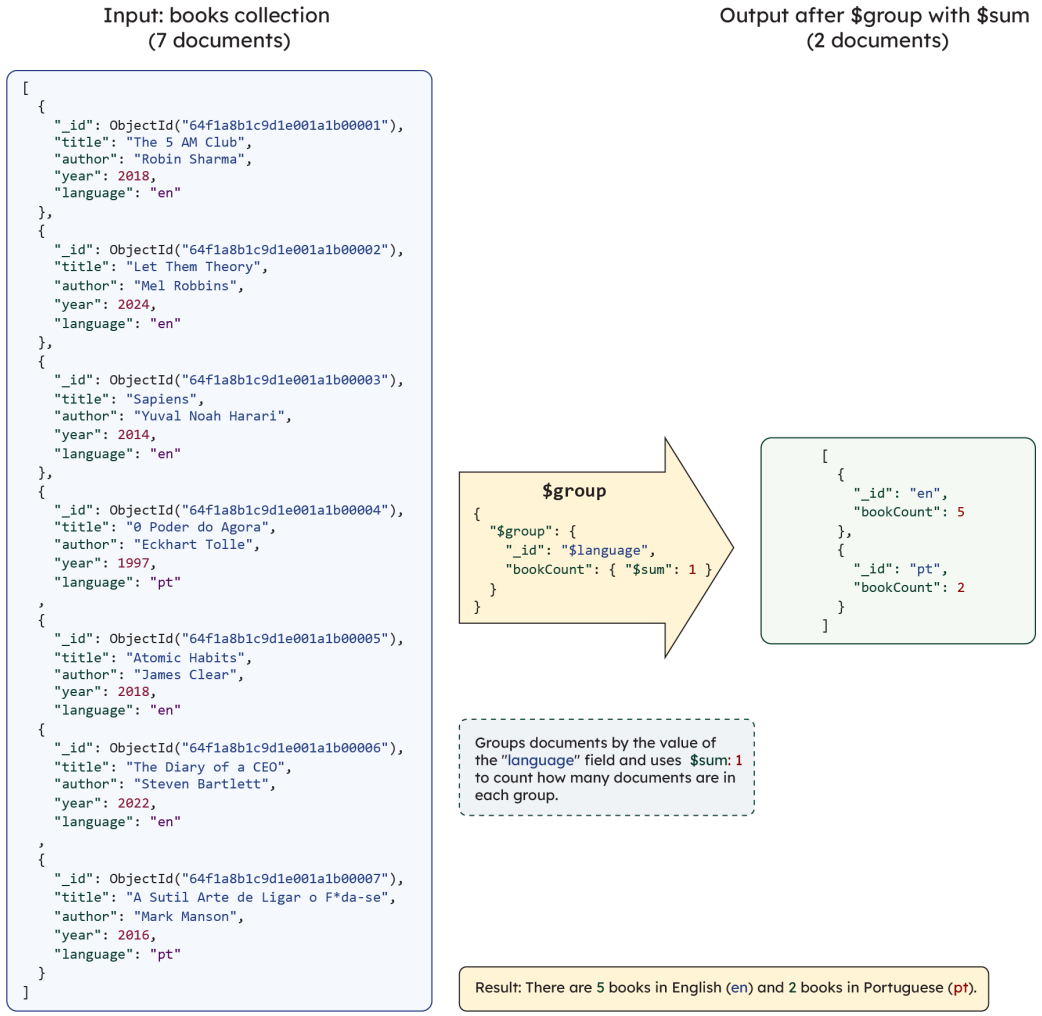


Figure 1.7 - $sum in aggregation framework

$push and $addToSet

Sometimes the goal isn't a single number. You might want to collect values from each group into an array. As discussed earlier, `$push` collects all values, including duplicates, and `$addToSet` collects unique values only.

For example, you might want to collect all genres seen for each author.

```
{
  "$group": {
    "_id": "$author",
    "genres": { "$addToSet": "$genres" }
  }
}
```

In practice, though, because genres is an array, you would typically `$unwind` it first, then group. This is because `$unwind` allows you to create documents for each value in the array, so you want to be able to access them individually.

```
[
  { "$unwind": "$genres" },
  {
    "$group": {
      "_id": "$author",
      "genres": { "$addToSet": "$genres" }
    }
  }
]
```

Use `$push` and `$addToSet` when you need to reshape grouped data into arrays rather than scalar values. In real-world scenarios, especially with nested arrays, this often goes hand in hand with `$unwind`, allowing you to aggregate individual elements easily before rebuilding them into meaningful groups of results.

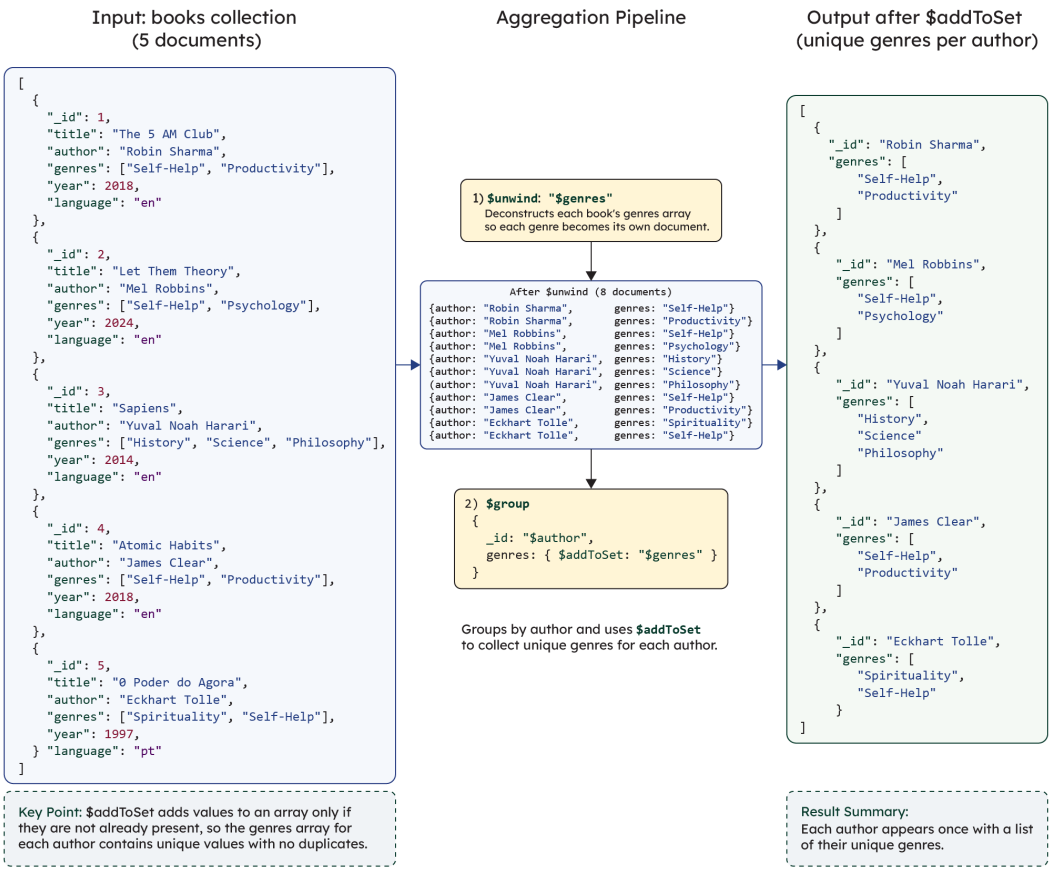


Figure 1.8 - $addToSet in aggregation framework

First and last with $first/$last

The `$first` and `$last` operators are useful when combined with sorting. For example, you can sort by year and then capture the earliest and latest book titles for each author.

Without an explicit sort, MongoDB processes documents in pipeline order, which may not match business meaning, such as earliest or latest by year. In practice, this means `$first` and `$last` should usually be read as first and last in the current sorted sequence, not first and last in absolute terms.

In some cases, you may want a quick way to group values and immediately sort them by frequency. MongoDB provides a dedicated stage for this called `$sortByCount`.

$sortByCount

Sometimes you don't need a full `$group` stage, you simply want to count how often a value appears and sort the results. This is where `$sortByCount` is useful. It groups documents by a specified expression and automatically returns the results in descending order by count, which is the only supported sort direction.

While this stage is more specialized than `$match` or `$group`, it is included here because it offers a concise pattern for a common analytics question: *Which values appear most often?*

For example, suppose the library has books in many different languages, and you want to find the most common language in the library.

```
{
  "$sortByCount": "$language"
}
```

In the C# driver, this stage is exposed using `SortByCount()`. The stage returns a result containing the grouped value and the number of documents for that value. You can then project this result into a type that is easier to work with in the rest of your application.

```
public class CountByValue
{
    public string Id { get; set; } = null!;
    public long Count { get; set; }
}

var results = booksCollection
    .Aggregate()
    .SortByCount(b => b.Language)
    .Project(r => new CountByValue
    {
        Id = r.Id,
        Count = r.Count
    })
  .ToList()
```

This returns documents with the two properties as seen in the `CountByValue` class: `Id` (which is the language) and `Count` (which is the number of documents with that value).

$setWindowFields

So far, the computations you have explored using `$group` have produced one output document per group. In many cases, that is exactly what you want, for example, a summary per author, per language, or per genre.

Sometimes, however, you want to calculate values across an ordered set of documents while still keeping the original documents in the output. This is the role of `$setWindowFields`.

Unlike `$group` which collapses documents into a single result per group, `$setWindowFields` keeps the original documents and adds computed values based on a defined `"window"` of neighboring documents.

A window is a set of documents around the current document that MongoDB uses to calculate a value. Instead of computing something across the whole collection, MongoDB calculates the result using only the documents inside that window.

This makes it useful for ranking, running totals, moving averages, and analytics-style queries.

For example, you might want to add a running count of books ordered by publication year.

```
{
  "$setWindowFields": {
    "sortBy": { "year": 1 },
    "output": {
      "runningCount": {
        "$count": {},
        "window": { "documents": ["unbounded", "current"] }
      }
    }
  }
}
```

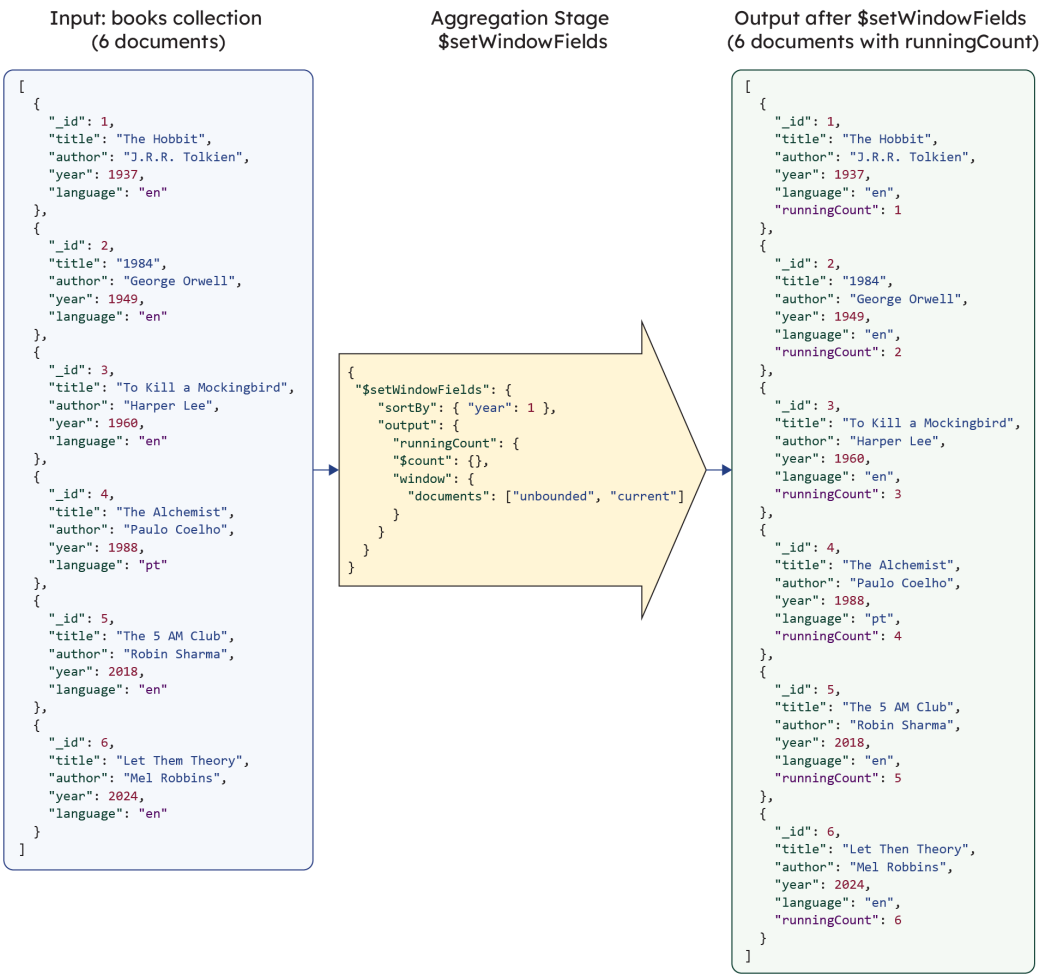


Figure 1.9 - $setWindowFields in aggregation framework

In the above JSON example, documents are sorted by year and a running count is calculated from the start of the dataset (`"unbounded"`) up to the current document, with the computed value added as a new field. The C# driver provides support for `$setWindowFields` but because window-based computations require sort definitions, window boundaries, and output configuration, the strongly typed API is naturally more verbose than earlier examples such as `$match` or `$group`.

For introductory examples, it is often clearer to express the stage using MongoDB's JSON syntax and then map the results into a strongly typed class.

```
public class BookWithRunningCount
{
    public string Title { get; set; } = null!;
    public int Year { get; set; }
    public long RunningCount { get; set; }
}

var setWindowFieldsStage = """
{
  "$setWindowFields": {
    "sortBy": { "year": 1 },
    "output": {
      "runningCount": {
        "$count": {},
        "window": { "documents": ["unbounded", "current"] }
      }
    }
  }
}
""";

var docs = booksCollection
    .Aggregate()
    .Match(b => b.Year != null)
    .AppendStage<BsonDocument>(setWindowFieldsStage)
    .ToList();

var results = docs.Select(doc => new BookWithRunningCount
{
    Title = doc["title"].AsString,
    Year = doc["year"].AsInt32,
    RunningCount = doc["runningCount"].ToInt64()
}).ToList();
```

Here, the aggregation stage runs entirely on the server, and the final mapping into `BookWithRunningCount` occurs in your application code. This keeps the example readable while still demonstrating the capabilities of the aggregation framework.

In many applications, `$group` is sufficient. If you only need one result per group, it is simple and more concise. `$setWindowFields` becomes useful when you:

- Need per-document analytics
- Want to preserve the original documents
- Require calculations based on document order

The important takeaway at this stage is not memorizing the driver syntax but understanding the concept. Window stages allow you to compute values across ordered documents without collapsing them into grouped summaries.

Enriching data with other specialized operations

Up to this point in the chapter, you have focused on three core capabilities of the aggregation framework:

- Filtering and selecting data using stages such as `$match`, `$project`, `$sort`, `$limit`, and `$skip`.
- Transforming documents with stages such as `$set`, and `$unset`.
- Computing values using `$group`, accumulator operators, `$sortByCount`, and `$setWindowFields`.

These stages allow you to filter, reshape, and analyze data within a single collection. In many applications, that is enough. However, real-world applications often require more than working with data in isolation. You may need to:

- Combine data from multiple collections
- Work with nested arrays
- Perform faceted searches
- Persist aggregation results
- Apply geospatial constraints

The aggregation framework supports all these scenarios through additional stages that enrich or extend the data flowing through the pipeline.

$lookup

One of the most powerful enrichment stages is `$lookup`. It allows you to join data from another collection into the current pipeline. While MongoDB is not a relational database and does not enforce relational constraints, `$lookup` provides a way to combine related data when needed.

In the library application example, each author document stores a list of book IDs for the books they have authored. If you want to retrieve authors along with the full book details for each referenced book, you can use `$lookup` to enrich the author documents during aggregation.

```
db.authors.aggregate([
{
  "$lookup": {
    "from": "books",
    "localField": "books",
    "foreignField": "_id",
    "as": "booksWritten"
  }
}
]);
```

This aggregation reads from the `books` collection, matches documents where `_id` matches the value in the books array, and adds the results as an array in the `booksWritten` array.

```
List<Author> lookedUpAuthors = authorsCollection
    .Aggregate()
    .Lookup<Author, Book, Author>(
        foreignCollection: booksCollection,
        localField: a => a.Books,
        foreignField: b => b.Id,
        @as: a => a.BooksWritten)
    .ToList();
```

This is expressed using the fluent API with the C# driver. The `Books` property contains the stored book identifiers, and `BooksWritten` is populated during the aggregation process with the matching `Book` documents.

`$lookup` allows you to enrich documents at query time without restructuring your schema purely for reading, and the original collections remain independent.

`$lookup` is a powerful stage, but it is not always required. In many MongoDB applications, related data is embedded directly within documents rather than joined at query time. In *Chapter 3*, you will see how data modeling decisions influence whether stages like `$lookup` are necessary and how they impact performance.

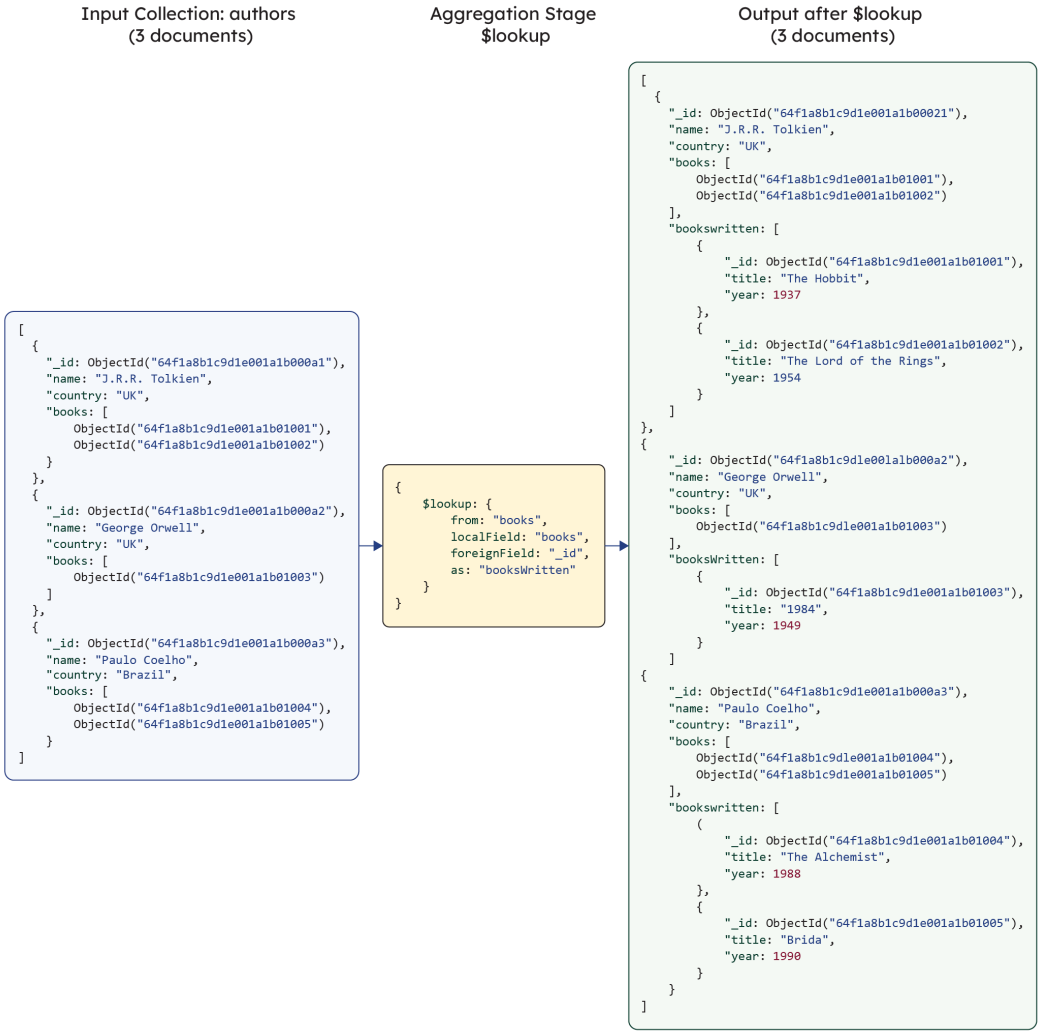


Figure 1.10 - $lookup in aggregation framework

$unwind

When working with arrays, you may need to treat each element as its own document within the pipeline. This is where `$unwind` comes in.

For example, if each book has multiple genres and you want to group books by genre, you first need to flatten the genres array.

```
{
  "$unwind": "$genres"
}
```

The `$unwind` stage can then be used in the aggregation pipeline as follows:"

```
var results = booksCollection
    .Aggregate()
    .Unwind(b => b.Genres)
    .ToList();
```

Each genre in the array becomes its own document in the pipeline. It is often used before `$group` when you want to compute statistics across array elements rather than entire documents.

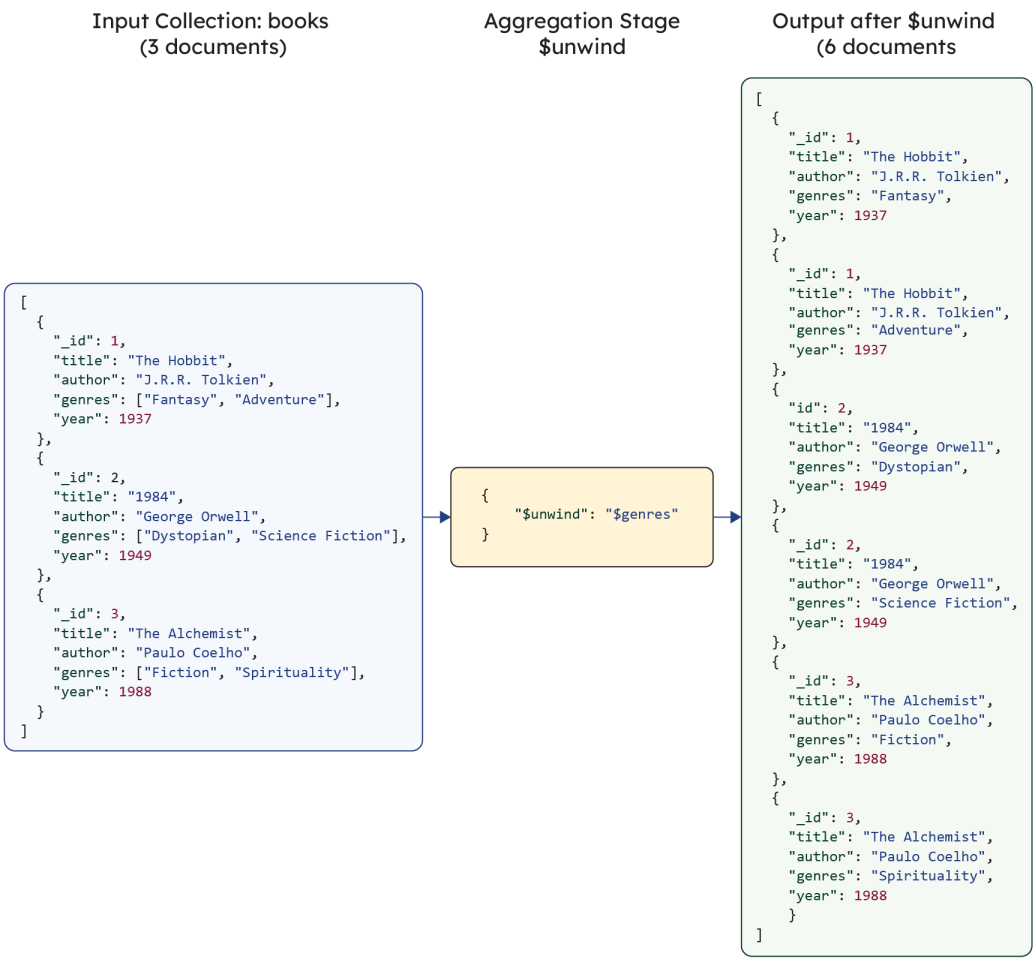


Figure 1.11 - $unwind in aggregation framework

$facet

Sometimes you need multiple aggregation results from the same filtered data. Running separate queries would require scanning the data multiple times. The `$facet` stage allows multiple sub-pipelines to run in parallel against the same input.

For example, you might want to break down books by language or count the number of books per publication year.

```
{
  "$facet": {
```

```
    "byLanguage": [
      { "$sortByCount": "$language" }
    ],
    "byYear": [
      { "$group": { "_id": "$year", "count": { "$sum": 1 } } }
    ]
  }
}
```

Each facet runs independently against the same input documents, and the results are returned together in a single response.

In C#, `$facet` can be expressed using the fluent API, but the structure becomes more involved and complex. For now, the key thing to understand is that `$facet` enables multiple aggregation views in one pipeline execution.

$out and $merge

Until now, the pipelines have returned results directly to the application. However, aggregation pipelines can also write results back to a collection. Two stages support this behavior: `$out` and `$merge`. While they appear similar, they serve slightly different purposes.

$out

The `$out` stage writes the results of the aggregation pipeline to a specific collection. If the collection already exists, it is replaced.

```
{
  "$out": "authorStatistics"
}
```

This creates (or replaces) a collection called `authorStatistics` containing the pipeline results. `$out` is useful when you want to create reporting collections, materialize precomputed views, or persist aggregation results as a new dataset. Because `$out` replaces the target collection, it is typically used when you want a full refresh of the computed data.

$merge

The `$merge` stage is more flexible. Instead of replacing the entire collection, `$merge` allows you to define how incoming documents should be handled.

```
{
  "$merge": {
    "into": "authorStatistics",
    "on": "_id",
    "whenMatched": "replace",
    "whenNotMatched": "insert"
  }
}
```

Here, documents are matched based on `_id`. If a match is found, the existing document is replaced. If no match is found, a new document is created. `$merge` allows you to update existing documents, insert new ones, control conflict behavior, and maintain incremental aggregation results.

Both stages move aggregation beyond query-time analysis and into data transformation workflows. They allow you to build pipelines that not only analyze data but also persist computed results for later use.

In C#, these stages can be expressed using `AppendStage()` with the appropriate JSON definition, as they are typically placed at the end of a pipeline.

```
var pipeline = booksCollection
    .Aggregate()
    .Group(b => b.Year, g => new
    {
        Year = g.Key,
        Count = g.Count()
    })
    .AppendStage<BsonDocument>("{ \"$out\": \"booksByYear\" }");
```

Because these stages write to collections rather than returning results, they are usually executed without calling `ToList()`, which is used to make the data iterable with methods such as `foreach`.

Summary

In this chapter, you were introduced to MongoDB's aggregation framework and explored how pipelines allow you to filter, transform, compute, and enrich data using a sequence of stages, much like an assembly line. Rather than pushing complex data processing into application code, pipelines let you filter, reshape, analyze, and enrich data directly within the database.

Next, you explored transformation stages such as `$set` and `$unset`, which allow you to add, modify, or remove fields as documents flow through the pipeline. These stages are especially useful when preparing data for later analysis or shaping results for later use in application code without changing the stored documents themselves.

From there, you moved on to compute stages, where the aggregation framework becomes especially powerful. You examined `$count` for returning totals, `$group` for producing summaries across sets of documents, and accumulator operators such as `$sum`, `$avg`, `$min`, `$max`, `$push`, and `$addToSet`. These operators allow you to calculate counts, averages, ranges, and collections of values for each group, making them essential for reporting and analytical workloads.

The chapter also introduced more specialized stages, including `$sortByCount`, which provides a concise way to group values and sort them by frequency, and `$setWindowFields`, which enables calculations across ordered documents while preserving the original results. This introduced an important distinction between grouped summaries and per-document analytics.

Finally, the chapter covered data enrichment and extended pipeline capabilities, including `$lookup` for joining related data from another collection, `$unwind` for flattening arrays, `$facet` for running multiple sub-pipelines in parallel, and `$merge` and `$out` for persisting aggregation results to a collection. These stages show that aggregation pipelines are not limited to simple filtering or summarization but can support much broader data processing workflows.

In the next chapter, you will build on what you have learned and actually apply some aggregations to the library application, giving you a taste of how aggregations are used and how stages work together to support real-world scenarios. In *Chapter 3*, you will also explore performance and best practices when working with aggregation pipelines to ensure you get the most out of them and avoid common mistakes developers make.

Additional resources

The following resources provide useful information for learning more about the MongoDB aggregation framework:

- MongoDB aggregation framework documentation: `https://mdb.link/building-modern-data-applications-with-mongodb-aggregation-docs`.
- MongoDB University – Fundamentals of Data Transformation course and skill badge: `https://mdb.link/building-modern-data-applications-with-mongodb-aggregation-course`.

2

Add Aggregations to the Application

In the previous chapter, you explored what the aggregation framework is in MongoDB and how you can use it for everything from filtering and shaping data to performing more advanced computations. Understanding the individual stages is useful, but you will truly appreciate the power of aggregation when you see it solving real problems in an actual application.

As discussed in the *Chapter 11*, *Appendix*, the application you will use throughout this book is Leafy Library, a library management application. Right now, Leafy Library has a catalog page for listing and reserving books, and a dashboard for librarians to manage reservations and loans. However, what it does not yet have is a dashboard for the readers themselves.

In this chapter, you will change that. You will build a user dashboard that presents useful information based on each reader's borrowing history. This dashboard will display current active loans, highlight the user's top-read genres, present their loan history by month, and include data on the average reading time per book, allowing users to gain a comprehensive understanding of their reading habits.

You will build this dashboard step by step, by implementing the aggregation stages discussed in the previous chapter. By the end of the chapter, you will have seen how individual stages can be combined into practical pipelines that support real application features.

The following topics are covered in this chapter:

- Building the first pipeline – Active Loans
- Building the second pipeline – Your Top Genres
- Building the third pipeline – Your Loan History

- Building the final pipeline – Loan Summary Cards
- Error handling and things to watch out for

Technical requirements

To follow along with this chapter, you will need the following prerequisites:

- .NET 10 SDK installed on your machine.
- An editor or IDE such as Visual Studio, VS Code, or Rider.
- The sample repository (`https://mdb.link/devrel.book.building-modern-data-applications-with-mongodb-repo`) forked on the start branch, or if you prefer, you can run this in GitHub Codespaces.
- A MongoDB connection string added to the application settings (Atlas, Community, and Enterprise Advanced are all supported; if using Atlas, an M0 cluster is sufficient).
- A 32-character minimum **JSON Web Token** (**JWT**) secret configured in the application settings.

The supporting code snippets for this chapter are available in the book's repository at `https://mdb.link/building-modern-data-applications-with-mongodb-codesnippets-repo`

> If you would like a complete overview of the Leafy Library application architecture and implementation, refer to *Chapter 11*, *Appendix*.

Building the first pipeline – Active Loans

To begin building your new user dashboard, start with the most important piece of information a user will need: the books they currently have on loan. From a user's perspective, this answers a simple but important question: *what books do I currently have on loan, and when are they due back?*

In the application, this information is stored in the `issueDetails` collection. Each document represents an interaction between a user and a book, such as borrowing or returning it.

Below is an example document from the `issueDetails` collection, showing the available information that you will use:

```
{
  "_id": "64d4c964f0d056ea6bf0f3d8B0451409434",
  "book": {
```

```
    "_id": "0451409434",
    "title": "Fantastic Voyage: Microcosm"
  },
  "borrowDate": {
    "$date": "2021-09-08T18:44:12.207Z"
  },
  "dueDate": {
    "$date": "2021-09-29T18:44:12.207Z"
  },
  "recordType": "borrowedBook",
  "returned": true,
  "returnedDate": {
    "$date": "2021-09-21T20:26:15.229Z"
  },
  "user": {
    "_id": "64d4c964f0d056ea6bf0f3d8",
    "name": "OldSchool Alligator"
  }
}
```

For this section, you are only interested in documents that represent books currently on loan and have not yet been returned. Each new feature added in this chapter will be implemented step by step, working from the backend to the frontend and making the required file and code changes until the feature can be run and seen in action.

Adding new LoanSummary model class

As part of the new dashboard, you will define a new model that takes advantage of the strongly typed nature of C# to power the dashboard's active loans section.

Start by creating a new class called `LoanSummary` inside the `Models` folder of the Leafy Library project. Then, paste the following code into the new file:

```
using MongoDB.Bson.Serialization.Attributes;

namespace Leafy_Library.Models;
```

```
[BsonIgnoreExtraElements]
public class LoanSummary
{
    [BsonElement("title")]
    public string Title { get; set; } = string.Empty;

    [BsonElement("borrowDate")]
    public DateTime? BorrowDate { get; set; }

    [BsonElement("dueDate")]
    public DateTime? DueDate { get; set; }

    [BsonElement("daysToDue")]
    public int DaysToDue { get; set; }

    [BsonElement("status")]
    public string Status { get; set; } = string.Empty;
}
```

This class contains properties relevant to a loan summary, including the due date and the number of days until the due date. You will also see attributes added from the MongoDB C# driver, which handles mapping it to documents.

The `[BsonIgnoreExtraElements]` attribute is added to the class so that the MongoDB C# driver ignores any fields in the retrieved documents that do not have corresponding properties in the class. This allows you to focus on using properties that matter to your application while excluding unnecessary fields without causing errors.

The `[BsonElement]` attributes help you handle case differences between the naming conventions in C# and how they are capitalized in documents in the database.

Updating the IssueDetailsService.cs with GetLoanSummariesAsync method

All the logic for the dashboard will reside in the `IssueDetailsService` file, which already exists in the project and is used for other pages. You will add new methods to it that can be called from the UI to display this new information.

For each method you add, you will develop it gradually, with explanations throughout to help you understand what is happening at each stage:

1. Start by opening the `IssueDetailsService.cs` file and add the following method outline to the code:

```
public async Task<List<LoanSummary>> GetLoanSummariesAsync(string
userId)
{

}
```

2. Next, within this method, you will define the filter using the `$match` stage to find all documents where the user is currently the logged-in user, the book is marked as borrowed, and it has not yet been returned.

```
 var matchFilter = UserMatch(userId)
            & Builders<IssueDetail>.Filter.Eq(x => x.RecordType,
IssueDetailType.BorrowedBook)
            & Builders<IssueDetail>.Filter.Eq(x => x.Returned,
false);
```

3. Now, you will add a stage that takes advantage of the `$addFields` stage, as discussed in *Chapter 1*, *Transform Your Data with the Aggregation Framework*, to calculate whether the book is overdue for return, due soon, or in good standing using a `$switch expression`. As discussed in *Chapter 1*, because C# is a strongly typed language, there is no neat way to do this natively with the driver, so instead, you will use `BsonDocument` to build up the stage. Below the `$match` stage, paste the following code to calculate the status of the book:

```
var statusStage = new BsonDocument("$addFields",
            new BsonDocument("status", new BsonDocument("$switch",
new BsonDocument
            {
                { "branches", new BsonArray
                    {
                        new BsonDocument { { "case", new
BsonDocument("$lt", new BsonArray { "$daysToDue", 0 }) }, { "then",
"Overdue" } },
                        new BsonDocument { { "case", new
BsonDocument("$lte", new BsonArray { "$daysToDue", DueSoonDays }) },
{ "then", "Due soon" } }
```

```
                    }
                },
                { "default", "OK" }
            })));
```

4. Finally, for the last stage definition, you will use the `$project` stage because you only need certain fields from the document:

```
var projectStage = new BsonDocument("$project", new BsonDocument
        {
            { "title", "$book.title" },
            { "borrowDate", 1 },
            { "dueDate", 1 },
            { "daysToDue", 1 },
            { "status", 1 }
        });
```

Before bringing the pipeline together, you also need a stage that calculates the number of days until each book is due. For this, you will use the `$dateDiff` stage, which is already defined at the top of the service so it can be reused in multiple dashboard aggregations.

This is what the `DaysToDueStage` looks like:

```
private static readonly BsonDocument DaysToDueStage =
new("$addFields",
        new BsonDocument("daysToDue", new BsonDocument("$dateDiff",
new BsonDocument
        {
            { "startDate", "$$NOW" },
            { "endDate", "$dueDate" },
            { "unit", "day" }
        })));
```

5. Now you can bring it all together by concluding the method with a return statement that performs the aggregation, including `Match()`, the new stages you defined in the previous steps, and sorting by the due date:

```
return await _issueDetails
            .Aggregate()
            .Match(matchFilter)
            .AppendStage<BsonDocument>(DaysToDueStage)
```

```
            .AppendStage<BsonDocument>(statusStage)
            .AppendStage<LoanSummary>(projectStage)
            .SortBy(x => x.DueDate)
            .ToListAsync();
```

In this section, you implemented the `GetLoanSummariesAsync` method in `IssueDetailsService.cs`, which uses a MongoDB aggregation pipeline to retrieve a summarized view of a user's currently borrowed books. The pipeline begins with a `$match` stage to filter documents belonging to the logged-in user that are marked as borrowed and not yet returned. It then appends a `$addFields` stage using the pre-defined `DaysToDueStage` to calculate the number of days remaining until each book is due, followed by a second `$addFields` stage using a `$switch` expression to assign a status of `Overdue`, `Due soon`, or `OK` to each loan. Finally, a `$project` stage shapes the output into a `LoanSummary` object containing only the fields needed for display, and the results are sorted in ascending order by due date.

This is all you need to do in this service class to get the information needed to provide a loan summary. Now, you can move on to updating the dashboard to show the new summary to the logged-in user.

Updating Dashboard.razor to display the user's loan summary

In this section, you will enhance the dashboard page in two stages. First, by adding a new code block inside the `Dashboard.razor` file that will call your new method in `IssueDetailsService.cs` and then the actual components on the page that will display this information. By the end of the section, you will have a widget on the Dashboard page that summarizes current loans and their status for the logged-in user.

Creating the code block

Before defining the code block at the bottom of the file, you need to add two pieces of code at the top. The first one checks that the user is logged in via the `[Authorize]` attribute and gives you access to user details. The second one injects the `IssueDetailsService.cs` class so you can call your new method.

1. Paste the following at the top of the `Dashboard.razor` file to add the new attribute and injected class:

```
@attribute [Authorize]
@inject IssueDetailService IssueDetailService
```

2. Then define the code block at the bottom of the file so you can run code from the Razor file to access values used in the markup:

```
@code {

}
```

3. Then, inside the code block, add the following variables so you can access the user's loans and their authentication state:

```
// This will contain the loan summary details of all loans for the
currently logged in user
private List<LoanSummary>? Loans;
// This gives us access to the AuthStateTask which allows us to
access details such as the user ID of the logged in user
[CascadingParameter]
private Task<AuthenticationState>? AuthStateTask { get; set; }
```

4. Next, you need to set up `OnInitializedAsync()` so the data will be populated when the page loads:

```
protected override async Task OnInitializedAsync()
    {

}
```

5. Inside the method, paste the following code that will call your new `GetLoanSummariesAsync` method if the user is logged in and populate the `loans` variable with their loan summaries data:

```
if (AuthStateTask is not null)
        {
            var authState = await AuthStateTask;
            var userId = authState.User.FindFirst("sub")?.Value;
            if (!string.IsNullOrEmpty(userId))
            {
                loans = await IssueDetailService.
GetLoanSummariesAsync(userId);
            }
        }
        loans ??= new();
```

Now that these variables are in place, your dashboard is ready to display the user's loan summary when the page loads. The next step is to render this data in the UI, which will allow users to easily view and interact with their loans.

Updating the actual UI code

To keep the UI nicely formatted, you will create a table that clearly displays the information retrieved from the loan summaries.

Paste the following code below the `title-wrapper` div inside the same `Dashboard.razor` file:

```
<div class="loans-section">
       <h2>Your loans right now</h2>

       @if (loans is null)
       {
           <p>Loading...</p>
       }
       else if (loans.Count == 0)
       {
           <p class="no-data">You have no active loans.</p>
       }
       else
       {
           <table class="loans-table">
               <thead>
                   <tr>
                       <th>Title</th>
                       <th>Borrowed date</th>
                       <th>Due date</th>
                       <th>Days until due</th>
                       <th>Status</th>
                   </tr>
               </thead>
               <tbody>
                   @foreach (var loan in loans)
                   {
                       <tr>
                           <td>@loan.Title</td>
```

```
                            <td>@loan.BorrowDate?.ToString("dd MMM
yyyy")</td>
                            <td>@loan.DueDate?.ToString("dd MMM yyyy")</
td>
                            <td>
                                @if (loan.DaysToDue < 0)
                                {
                                    <span>@Math.Abs(loan.DaysToDue) days
overdue</span>
                                }
                                else
                                {
                                    @loan.DaysToDue
                                }
                            </td>
                            <td>
                                <span class="status-badge status-@loan.
Status.ToLower().Replace(" ", "-")">@loan.Status</span>
                            </td>
                        </tr>
                    }
                </tbody>
            </table>
        }
    </div>
```

To make things easier for you and save time, the CSS classes are already defined in `Dashboard.razor.css` so it will already be nicely formatted with the classes assigned.

Testing out the loan summaries table

Now it's time to run the application and see the new dashboard in action! If you haven't done so already, make sure to check *Chapter 11*, *Appendix* for instructions on populating your database with the data in the `Data` folder if you are unsure how. This will ensure that you have books, authors, and other items available for loan.

If you load the application and log in, you can use any username you want. The authentication is set up to create a new user the first time you use a username. Just make sure you use the same username throughout the rest of the book. Then, click **Dashboard** in the navigation menu at the top. You should see your new loan summary information!

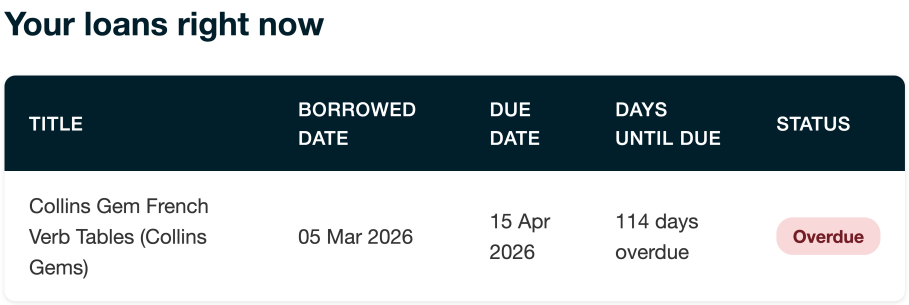


Figure 2.1 – Table displaying any current loans for the logged-in user, including the title, borrowed date, due date, and status

If you haven't used the application before, you may not see any information here. However, if you return to the home page and reserve a book as the first user of the application, or update your user information in the `users` collection of the database using a tool such as MongoDB Compass, you will see a **Librarian Dashboard** link in the navigation menu. From there, you can turn a reservation into a loan.

You can then return to the dashboard to see the loan stats, as shown in *Figure 2.1*. The borrowed date and due date have been changed manually for this example to show the **Overdue** status in action. However, by default, you should see a status of **OK** because it defaults to not being overdue.

Building the second pipeline – Your Top Genres

In this section, you will use the `$lookup` stage, along with a few other stages, to generate some interesting statistics about a user's most-read genres. The challenge here is that while books in the `books` collection contain a `genres` field, the `issueDetails` collection only stores a small subset of book information: the `_id` and `title`. This means the loan history on its own does not contain enough data to group by genre.

To solve this, `$lookup` will be used to enrich the loan history by pulling in corresponding book documents from the `books` collection. Once you have these documents, you can access their genres and use that information to build a picture of the user's reading habits.

Adding new GenreCount model class

Similar to the loan summary feature you just implemented, you need a new model class to represent a genre and the number of times it appears in the user's loan history.

To create this model, add a new class called `GenreCount` to the `Models` folder, then replace any existing boilerplate code with the following:

```
using MongoDB.Bson.Serialization.Attributes;

namespace Leafy_Library.Models;

[BsonIgnoreExtraElements]
public class GenreCount
{
    [BsonId]
    public string Genre { get; set; } = string.Empty;

    [BsonElement("count")]
    public int Count { get; set; }
}
```

Now that you have defined a model class to hold the information needed for this section, you can move on to using it in the `IssueDetailsService` class. This model will keep a running total of the genres appearing in the loan history.

Updating the IssueDetailsService.cs with GetTopGenresAsync method

In this section, you will add the new method inside `IssueDetailsService`, underneath the `GetLoanSummariesAsync` method added earlier, to use the MongoDB C# driver to populate the top genres information and return it as a list of `GenreCount` objects.

To get started, follow these steps:

1. Add the following method outline to the `IssueDetailsService` class:

```
public async Task<List<GenreCount>> GetTopGenresAsync(string userId)
{
}
```

2. Next, define the `groupStage` variable. This stage groups the results by genre and increments the count by `1` each time a genre appears, allowing you to determine how often each genre occurs across the user's loan history. Paste the following inside the method:

```
var groupStage = new BsonDocument("$group", new BsonDocument
        {
            { "_id", "$bookDetails.genres" },
            { "count", new BsonDocument("$sum", 1) }
        });
```

3. Now you can define the return statement that pulls this all together into an aggregation call. Paste the following code:

```
 return await _issueDetails
            .Aggregate()
            .Match(UserMatch(userId))
            .Lookup("books", "book._id", "_id", "bookDetails")
            .Unwind("bookDetails")
            .Unwind("bookDetails.genres")
            .AppendStage<GenreCount>(groupStage)
            .SortByDescending(x => x.Count)
            .Limit(5)
            .ToListAsync();
```

Several stages work together in this pipeline, showcasing the power of the aggregation framework. First, it filters the pipeline using the `$match` stage, so only documents that belong to the currently logged-in user proceed through the `$match` stage. Then it looks up the book details for each book in the `books` object from the `issueDetails` documents for that user, using the `$lookup` stage, and introduces them as an array field called `bookDetails`. This new array then needs to be unwound into individual book details using `$unwind`, and then the genres array inside these new individual book details also requires unwinding. Finally, the group stage is applied to each genre, and returns the five most-read genres using `$limit`.

Updating Dashboard.razor to show the new TopGenres widget

Let's begin by updating the code block and adding UI elements to the page that use the new `topGenres` variable, which will be available. You will start by adding the necessary variables and then initialize them using the `GetTopGenresAsync` method defined in the previous section.

1. Start by adding the following new variables below the existing `loans` variable in the code block:

```
private List<GenreCount>? topGenres;
private int maxGenreCount;
```

2. Inside the `if` block, add the following code below the line added earlier to initialize the `loans` variable and calculate `maxGenreCount`:

```
topGenres = await IssueDetailService.GetTopGenresAsync(userId);
maxGenreCount = topGenres.Count > 0 ? topGenres.Max(g => g.Count) :
0;
```

3. Paste the following UI code just below the div that was added for the loans section:

```
<div class="genres-section">
        <h2>Your top genres</h2>

        @if (topGenres is null)
        {
            <p>Loading...</p>
        }
        else if (topGenres.Count == 0)
        {
            <p class="no-data">No genre data yet.</p>
        }
        else
        {
            <div class="genre-bars">
                @foreach (var genre in topGenres)
                {
                    var pct = maxGenreCount > 0 ? (int)(100.0 *
```

```
genre.Count / maxGenreCount) : 0;
                    <div class="genre-row">
                        <span class="genre-label">@genre.Genre</
span>
                        <div class="genre-bar-track">
                            <div class="genre-bar-fill"
style="width: @(pct)%"></div>
                        </div>
                        <span class="genre-count">@genre.Count</
span>
                    </div>
                }
            </div>
        }
    </div>
```

This loops through the `topGenres` list retrieved from the service class and generates bars on the screen that show not only the top genres but also the count of how often each genre has been read. At this point, you have a Dashboard screen that displays loan summary information for the logged-in user and now a widget showing the most-read genres by the user.

Run the application again and revisit the dashboard to see it in action! Try reserving a few books. You can also mark them as returned and watch your top genres list grow!

Figure 2.2 - Stacked bar graph showing the user's top genres, with the genre names on the left and the number of loans for each genre on the right

Building the third pipeline – Your Loan History

Another useful piece of information to add to the dashboard is loan history by month, which shows how many loans were taken out each month. This helps users see how their borrowing activity changes over time. This is also a great way to see the `$group` and `$count` stages in action, as well as the `$datetostring` expression operator.

As with the previous widgets, you will introduce a new model class, update the `issueDetailsService` class with a new method that will utilize this model, and then update the dashboard page so the code calls the new method and the UI displays it.

Adding new MonthlyLoanCount model

Now that you can identify a user's most-read genres, you can add another stat to the dashboard: the number of loans made each month. To represent this data in the application, you need a new model that stores the month and the number of loans recorded during that month. This class will resemble the `GenreCount` class, but will track loans instead of genres.

In the `Models` folder, create a new class called `MonthlyLoanCount` and replace the boilerplate code with the following:

```
using MongoDB.Bson.Serialization.Attributes;

namespace Leafy_Library.Models;

[BsonIgnoreExtraElements]
public class MonthlyLoanCount
{
    [BsonId]
    public string Month { get; set; } = string.Empty;

    [BsonElement("count")]
    public int Count { get; set; }
}
```

As you can see, all you need for this model is the month, which acts as the `Id`, and the `count` that will hold the number of loans recorded during that month.

Updating the IssueDetailsService with the GetLoanHistoryByMonthAsync method

Now that you have the model class available, you can add a new `GetLoanHistoryByMonthAsync` method to the service class. This method will call the collection using the MongoDB C# driver and perform the aggregation steps necessary to gather and calculate the loan history information for each month.

Add the following method below the existing methods in the service class:

```
public async Task<List<MonthlyLoanCount>>
GetLoanHistoryByMonthAsync(string userId)
    {
        var groupStage = new BsonDocument("$group", new BsonDocument
        {
            { "_id", new BsonDocument("$dateToString", new BsonDocument
                {
                    { "format", "%Y-%m" },
                    { "date", "$borrowDate" }
                })
            },
            { "count", new BsonDocument("$sum", 1) }
        });

        return await _issueDetails
            .Aggregate()
            .Match(UserMatch(userId) & Builders<IssueDetail>.Filter.Ne(x
=> x.BorrowDate, null))
            .AppendStage<MonthlyLoanCount>(groupStage)
            .SortBy(x => x.Month)
            .ToListAsync();
    }
```

This uses a new expression operator, `$dateToString`. In this case, during the group stage, the `$borrowDate` field is converted into a string with the format `YYYY-MM`, such as `2026-01` (excluding any documents where the `BorrowDate` is null) and a new document is created for each month. For each instance of that month, it updates the count value by `1`.

The aggregation then filters documents in the `issueDetails` collection to those belonging to the currently logged-in user, adds the group stage, and then sorts by month in ascending order.

With this information available, you can now make use of it in the `Dashboard.razor` file.

Updating Dashboard.razor to show the monthly loan history chart

You will start by updating the code block so that the variables and data returned by the new method in `issueDetailsService` are available to the UI. Then, you will add the UI elements needed to display the monthly loan history.

To bring these pieces together on the dashboard, follow these steps:

1. At the top of the code block in `Dashboard.razor`, below the `topGenres` variable, add the following:

```
private List<MonthlyLoanCount>? monthlyHistory;
 private int maxMonthCount;
```

2. Inside the `if` block, add the following after the call to `GetTopGenresAsync`:

```
monthlyHistory = await IssueDetailService.
GetLoanHistoryByMonthAsync(userId);
 maxMonthCount = monthlyHistory.Count > 0 ? monthlyHistory.Max(m =>
m.Count) : 0;
```

3. Outside the `if` block, you need to add the following method to help format the date for display:

```
private static string FormatMonth(string yyyyMm)
    {
        return DateTime.TryParseExact(yyyyMm, "yyyy-MM", null,
System.Globalization.DateTimeStyles.None, out var date)
            ? date.ToString("MMM yyyy")
            : yyyyMm;
    }
```

4. Now, add the following UI elements outside the code block and below the 'Your top genres' div:

```
<div class="history-section">
        <h2>Loan history by month</h2>

        @if (monthlyHistory is null)
        {
            <p>Loading...</p>
        }
        else if (monthlyHistory.Count == 0)
        {
            <p class="no-data">No loan history yet.</p>
        }
        else
        {
            <div class="history-bars">
                @foreach (var month in monthlyHistory)
                {
                    var pct = maxMonthCount > 0 ? (int)(100.0 *
month.Count / maxMonthCount) : 0;
                    <div class="history-row">
                        <span class="history-label">@
FormatMonth(month.Month)</span>
                        <div class="history-bar-track">
                            <div class="history-bar-fill"
style="width: @(pct)%"></div>
                        </div>
                        <span class="history-count">@month.Count</
span>
                    </div>
                }
            </div>
        }
    </div>
```

That is all you need to do; now you can see it in action! In this section, you added a new variable to hold your loan histories by month information, initialized it by calling the method you created in the previous section, added a method to nicely format the date for display, and then added the new widget to your UI.

Run the application and return to the Dashboard page. If your dataset contains loan history, you should see the new monthly loan history chart below the genres chart, as shown in *Figure 2.3*.

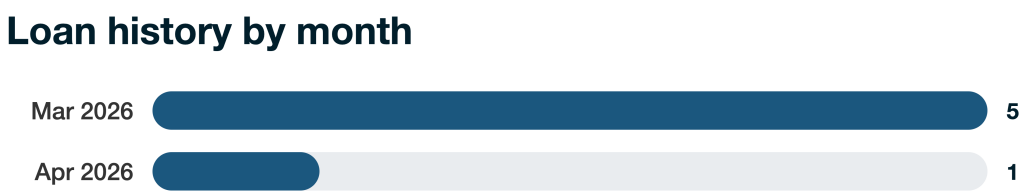


Figure 2.3 - Bar graph for logged-in users showing loan history by month with month and year on the left and count of loans in that month on the right

If you want, you can edit the documents inside the `issueDetails` collection for your user and change the dates in them to see how it impacts the results in the graph. You should see it map to the dates with the correct count of books loaned in that month.

Building the final pipeline – Loan Summary Cards

You now have some interesting stats available about a user's current and past loan history. However, it would be nice to have a summary at the top of the page that the user can look at at a glance without needing to scan the full dashboard. This summary will show the number of active loans, overdue books, books due soon, the number of books that have been returned, and most interesting of all, the average reading time per book.

For this, you are going to use the `$facet` stage. This will allow you to compute all five stats in a single query. Throughout this section, you will implement the steps needed to gather the required information and update the code for Leafy Library to display it.

Adding a new LoanSummaryStats model class

To represent these summary statistics in the application, you need another model class. This model will hold the values for each statistic that will eventually be displayed at the top of the dashboard.

In the `Models` folder, create a new class called `LoanSummaryStats.cs` and replace the existing code with the following:

```
namespace Leafy_Library.Models;

public class LoanSummaryStats
{
    public int ActiveLoans { get; set; }
    public int Overdue { get; set; }
    public int DueSoon { get; set; }
    public int Returned { get; set; }
    public double? AvgReadingDays { get; set; }
}
```

This holds the values for your stats, including using `double` to represent the average days to read a book, as you want to support decimal places.

Now that you have this model class available, you can move on to updating the service class to implement a method that uses this new model.

Updating the IssueDetailsService class with GetLoanSummaryStatsAsync

As mentioned earlier, this method will use the `$facet` stage to calculate several loan statistics in a single aggregation, but you will also need to define a stage as a variable to calculate the number of days until each loan is due

To implement the method, follow these steps:

1. Inside `IssueDetailsService.cs`, add the new method as an empty method. You will add the aggregation logic in the subsequent steps:

```
 public async Task<LoanSummaryStats> GetLoanSummaryStatsAsync(string
userId)
{

}
```

2. Now let's build the `$facet` stage definition. Paste the following code at the top of the method:

```
var facetStage = new BsonDocument("$facet", new BsonDocument
        {
            { "activeLoans", new BsonArray
                {
                    new BsonDocument("$match", new
BsonDocument("returned", false)),
                    new BsonDocument("$count", "count")
                }
            },
            { "overdue", new BsonArray
                {
                    new BsonDocument("$match", new BsonDocument
                    {
                        { "returned", false },
                        { "daysToDue", new BsonDocument("$lt", 0) }
                    }),
                    new BsonDocument("$count", "count")
                }
            },
            { "dueSoon", new BsonArray
                {
                    new BsonDocument("$match", new BsonDocument
                    {
                        { "returned", false },
                        { "daysToDue", new BsonDocument { { "$gte",
0 }, { "$lte", DueSoonDays } } }
                    }),
                    new BsonDocument("$count", "count")
                }
            },
            { "returned", new BsonArray
                {
                    new BsonDocument("$match", new
BsonDocument("returned", true)),
                    new BsonDocument("$count", "count")
```

```
                }
            },
            { "avgReadingTime", new BsonArray
                {
                    new BsonDocument("$match", new
BsonDocument("returned", true)),
                    new BsonDocument("$project", new
BsonDocument("readingDays",
                        new BsonDocument("$dateDiff", new
BsonDocument
                        {
                            { "startDate", "$borrowDate" },
                            { "endDate", "$returnedDate" },
                            { "unit", "day" }
                        }))),
                    new BsonDocument("$group", new BsonDocument
                    {
                        { "_id", BsonNull.Value },
                        { "avgDays", new BsonDocument("$avg",
"$readingDays") }
                    })
                }
            }
        });
```

This is a great example of how multiple stages can be combined to produce something much more powerful. With `$facet`, each piece of analysis you want to perform runs as its own mini-pipeline, all within a single aggregation. Each entry inside the `$facet` stage represents a different calculation:

- `activeLoans` filters documents where `returned` is `false` and then uses `$count` to return the total number of active loans.
- `overdue` builds on this by filtering documents where `returned` is `false` and `daysToDue` is less than `0`, meaning the book is overdue, and again counts the results.
- `dueSoon` looks for books that are not yet returned and have a `daysToDue` value between `0` and `DueSoonDays`. In this case, `DueSoonDays` is set to `3` in the service, so this captures books due today or within the next few days.
- `returned` simply filters for returned books and counts them.

The final facet, `avgReadingTime`, is slightly more involved as it performs a calculation rather than just a count.

It starts by matching only returned books, since you need both a borrow and return date. It then uses `$project` to create a new field called `readingDays`, which calculates the difference between `borrowDate` and `returnedDate` using `$dateDiff`.

Once that value has been calculated for each document, the pipeline uses `$group` with the `$avg` accumulator to calculate the average reading time across all returned books.

3. Next, you need to call `Aggregate` on the `booksCollection` to feed in this stage and get back the results:

```
var result = await _issueDetails
            .Aggregate()
            .Match(UserMatch(userId))
            .AppendStage<BsonDocument>(DaysToDueStage)
            .AppendStage<BsonDocument>(facetStage)
            .FirstOrDefaultAsync();
```

This appends the existing `DaysToDueStage` at the top of the class as well as your new `facetStage`.

4. Finally, you want to process the results, and if there are any, extract the count. Add the following below the previous code:

```
if (result is null)
  return new LoanSummaryStats();

        static int ExtractCount(BsonDocument doc, string field)
        {
            var arr = doc[field].AsBsonArray;
            return arr.Count > 0 ? arr[0].AsBsonDocument["count"].
AsInt32 : 0;
        }

        return new LoanSummaryStats
        {
            ActiveLoans = ExtractCount(result, "activeLoans"),
```

```
            Overdue = ExtractCount(result, "overdue"),
            DueSoon = ExtractCount(result, "dueSoon"),
            Returned = ExtractCount(result, "returned"),
            AvgReadingDays = result["avgReadingTime"].AsBsonArray is
{ Count: > 0 } arr
                ? arr[0].AsBsonDocument["avgDays"].AsNullableDouble
                : null
        };
```

Now you can move on to adding these new cards to your dashboard!

Updating Dashboard.razor to show loan summary cards and due-soon alerts

This will sound familiar now, but you will take the same approach of updating the code block to call the new method, add any new variables you need, and move on to updating the UI with the new cards.

To make these changes, follow these steps:

1. Inside the code block underneath the existing variables, paste the following:

```
private LoanSummaryStats summaryStats = new();
private int dueSoonCount;
```

2. Inside the existing `if` block, add the following code to initialize the values:

```
summaryStats = await IssueDetailService.
GetLoanSummaryStatsAsync(userId);
dueSoonCount = summaryStats.DueSoon;
```

3. Now it is time to add the cards to the UI. These cards should appear above the other stats components, so add the following above the **Your loans history** section:

```
<div class="summary-cards">
        <div class="summary-card">
            <span class="summary-value">@summaryStats.ActiveLoans</
span>
            <span class="summary-label">Active loans</span>
        </div>
        <div class="summary-card summary-overdue">
            <span class="summary-value">@summaryStats.Overdue</span>
```

```
            <span class="summary-label">Overdue</span>
        </div>
        <div class="summary-card summary-due-soon">
            <span class="summary-value">@summaryStats.DueSoon</span>
            <span class="summary-label">Due soon</span>
        </div>
        <div class="summary-card summary-returned">
            <span class="summary-value">@summaryStats.Returned</
span>
            <span class="summary-label">Returned</span>
        </div>
        <div class="summary-card summary-avg">
            <span class="summary-value">@(summaryStats.
AvgReadingDays.HasValue ? $"{summaryStats.AvgReadingDays.Value:F1}"
: "—")</span>
            <span class="summary-label">Avg. days to read</span>
        </div>
    </div>
```

Apart from the average reading time card, which does some formatting, these cards mostly display values obtained from the `summaryStats` variable.

Now that you have access to the loan stats, including those due soon, you can add an additional card at the very top of the page, above the new summary cards, to highlight how many books are due soon.

4. Add the following above the **summary cards** div:

```
@if (dueSoonCount > 0)
    {
        <div class="due-soon-banner">
            <span class="due-soon-icon"> &#9888; </span>
            You have <strong>@dueSoonCount</strong> book@
(dueSoonCount == 1 ? "" : "s") due soon!
        </div>
    }
```

And with that, you have a finished dashboard! You have added new variables to hold the loan summary stats using the new model class you defined and a count for books due soon. You added two new widgets to your UI, and it's now set up to display the information for books due soon and summary stats.

Run the application for the final time in this chapter, and you should see your new cards on the screen with interesting information.

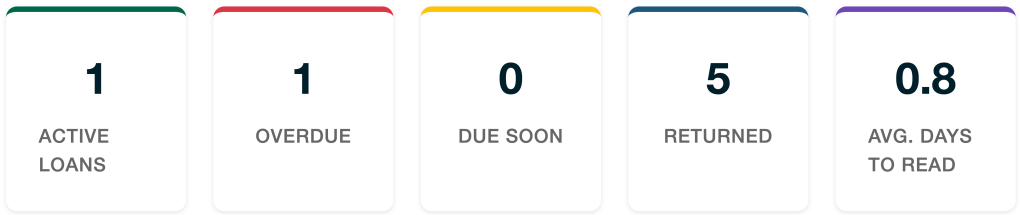


Figure 2.4 - Colored cards for loan summary stats showing active loans, overdue, due soon, returned, and average days to read with counts per card

If you don't have any books due soon but would like to see the new banner in action, you can always update the `dueDate` field in the `issueDetails` collection for a book currently on loan, then refresh the page.

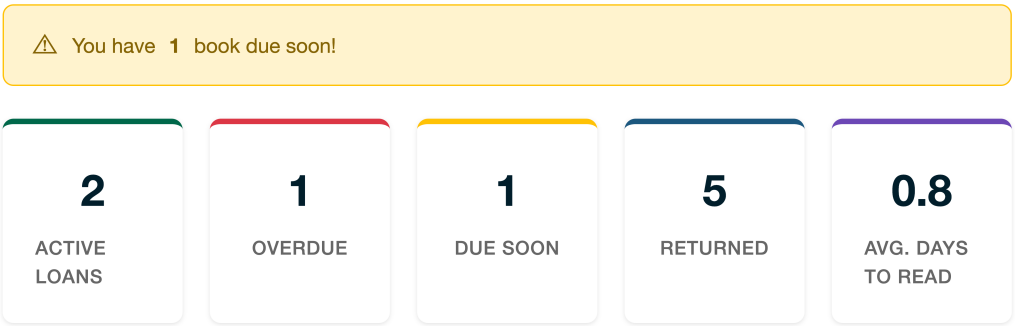


Figure 2.5 - Loan summary statistics with a banner above indicating that one loan is due soon

The `avgReadingTime` facet uses `$dateDiff` with `"unit", "day"` which returns whole days. If your returned books were borrowed and returned on the same day, a common scenario when testing the application, the difference is `0` days for all of them. You can fix this by editing the documents in the database so the return date is later than the borrowed date. Of course, this wouldn't happen in a real-world scenario!

Error handling and things to watch out for

The aggregation pipelines you built in this chapter work well under normal conditions, but there are a few MongoDB and driver-specific behaviors worth being aware of before you deploy something like this to production. In this section, you will look at a few potential pitfalls that can trip you up when working with these pipelines in a real application.

BsonSerializationException and field mismatches

Each of your pipelines projects into a specific model class: `LoanSummary`, `GenreCount`, `MonthlyLoanCount`, and `LoanSummaryStats`. If the shape of the documents coming out of the pipeline does not match the model, the MongoDB C# driver will throw a `BsonSerializationException`.

This most commonly happens when the aggregation returns a field with an unexpected type. For example, if `daysToDue` comes back as a double instead of an int due to how a stage computes it. The `[BsonIgnoreExtraElements]` attribute you added to your models helps with one side of this. It instructs the driver to silently ignore any fields in the document that do not have a corresponding property on the class. Without it, extra fields in the result would also cause a serialization error.

What it does not protect you from is a type mismatch on fields that are mapped. If you encounter serialization errors, the fastest way to diagnose them is to temporarily swap the typed `AppendStage<YourModel>` for `AppendStage<BsonDocument>` and inspect the raw output to see what the pipeline is actually returning.

Null fields in aggregation

You might have actually encountered this issue during development if you hadn't addressed it in the code as discussed below. Some books in the dataset might have a null `borrowDate`. When `$dateToString` is applied to a null value, MongoDB returns null for the `_id` of that group, which then deserializes to an empty string.

This issue was avoided by adding a `Ne` filter to exclude null `borrowDate` documents before the group stage, which is a good general pattern to keep in mind. Anytime you use expression operators such as `$dateToString`, or `$dateDiff` on a `date` field, it is worth considering what happens if that field is absent or null in some documents. MongoDB will not throw an error; it will just silently produce null or unexpected values that can be tricky to track down in the UI.

$facet and the 100 MB memory limit

The `$facet` stage you use for the summary cards runs multiple sub-pipelines in parallel, which is powerful, but it comes with a constraint worth knowing about. By default, MongoDB limits each pipeline stage to 100 MB of RAM. For a library application with a modest number of users, this is unlikely to be a concern, but if you run this against a collection with millions of documents, the `$facet` stage could hit that limit.

If you encounter this, MongoDB provides an `allowDiskUse` option on the aggregation that allows the pipeline to spill to disk when it exceeds the memory limit. In the C# driver, this can be enabled via the `AggregateOptions`:

```
var options = new AggregateOptions { AllowDiskUse = true };
_issueDetails.Aggregate(options)...
```

Timeout and connection errors

You may have already seen a `MongoConnectionException` if you ran the application before the connection string was correctly configured. By default, the driver spends 30 seconds trying to find a suitable server before giving up with a timeout error. In a production application, you would typically configure a shorter `ServerSelectionTimeout` to fail faster and make the problem more apparent to the user, rather than having requests hang for 30 seconds.

These pipelines all run as single aggregation calls, which means if the connection to MongoDB is lost mid-request, the entire call fails rather than partially completing. For read operations like these, this is acceptable as you simply get no data back. It is worth keeping in mind for the write operations elsewhere in the application, where partial failures require more careful handling.

Summary

What a fun chapter! In the first hands-on chapter of this book, you have implemented several aggregation pipelines using the C# driver to add features to the user dashboard:

- **Active loans**: Used `$match`, `$addFields`, and `$project` to retrieve the user's currently active loans.
- **Your top genres**: Introduced `$group`, `$lookup`, and `$unwind` to identify the genres that appear most often in the user's loan history.
- **Your loan history**: Introduced `$dateToString` to group and display loan activity by month.
- **Loan summary cards**: Used `$facet`, `$count`, `$dateDiff`, and `$avg` to calculate several loan statistics in a single aggregation.

You now have a dashboard in the library system that can provide useful stats for the logged-in user. Along the way, you have seen how to take advantage of many great aggregation stages, so you could even add your own widgets if you wanted to!

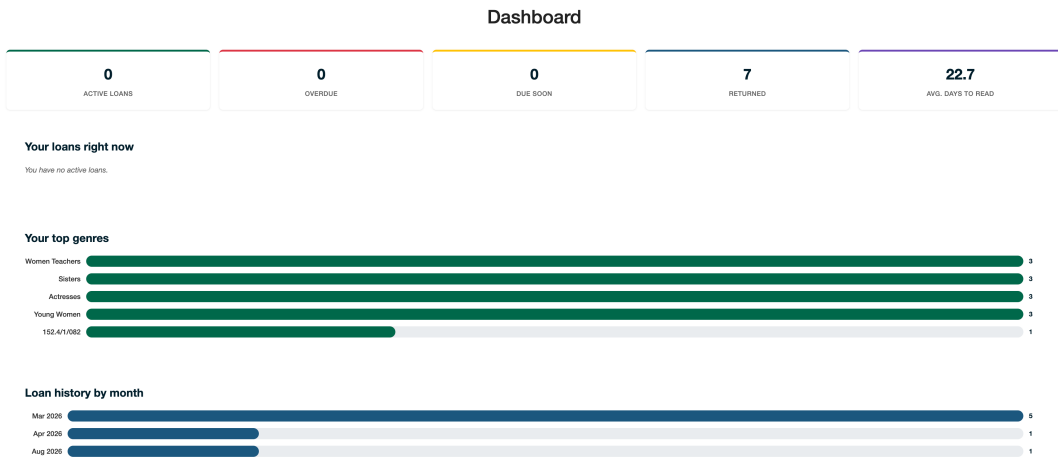


Figure 2.6 - View of the complete dashboard with all the widgets added

In this chapter, you focused on building useful features with aggregation pipelines. In the next chapter, we will look at how to make these pipelines more robust and efficient by reviewing aggregation best practices.

3

Best Practices for Aggregation Pipelines

In the previous chapter, we focused on building aggregation pipelines to power real features in the Leafy Library application. You saw how different stages can be combined to produce meaningful insights for users. In this chapter, you will take a step back and look at how to make those pipelines more efficient, scalable, and easier to maintain.

Aggregation is powerful, but performance is not automatic. A well-designed pipeline can be extremely fast, while a poorly designed one can become expensive as your dataset grows. The difference usually comes down to how data flows through the pipeline, how much work each stage performs, and how well your indexes support those operations.

The following topics are covered in this chapter:

- Designing a performant aggregation pipeline
- Data modeling considerations for aggregation pipelines
- Applying the ESR guideline to optimize indexes
- Optimizing indexes across the application
- Aggregation anti-patterns to avoid
- Keeping pipelines maintainable in .NET

> If you would like a complete overview of the Leafy Library application architecture and implementation, refer to *Chapter 11*, *Appendix*.

Designing a performant aggregation pipeline

When thinking about aggregation performance, the most important idea is simple, *the less data your pipeline has, the faster it will be*. Every document that flows through your pipeline adds cost. Every stage that transforms that data adds more cost. So, your goal is not to minimize the number of stages, but to minimize unnecessary work.

In this section, you will see how considering what data is included at each stage of the pipeline can make a difference in how the pipeline performs. By the end, you will understand not only what can impact performance but also how you can mitigate these issues with correct stage choices and by considering anti-patterns.

Understanding how data flows through a pipeline

MongoDB processes aggregation pipelines in a streaming fashion where possible, much like an assembly line. Each stage receives documents from the previous stage, applies its transformations, and passes the results on.

However, not all stages behave this way. Some stages are blocking, meaning they must see all incoming documents before producing any output. Examples include:

- `$sort`
- `$group`
- `$facet`
- `$bucket`

This difference matters because blocking stages are often where performance issues appear. They may hold large amounts of data in memory, delay downstream stages, or spill over to disk usage when memory limits are exceeded.

In contrast, streaming stages, such as `$match`, `$project`, and `$addFields`, can pass documents through immediately, making them much cheaper in most cases.

Consider what this looks like in a real application. If you have a `$group` stage that aggregates across thousands of loan records, MongoDB must hold all those records in memory, complete the grouping, and only then pass results to the next stage. Any downstream stages, such as a `$sort` or `$project`, are completely idle during this time. The user is waiting, and the pipeline is not making progress they can see.

This becomes more significant when blocking stages appear early in a pipeline, or when they operate on unreduced data. A `$group` that runs before a `$match` may process far more documents than necessary. A `$sort` on 50,000 documents takes considerably more memory and time than a `$sort` on 500. These delays compound: each blocking stage creates a pause that everything downstream must wait out.

MongoDB enforces a default memory limit of 100 MB per stage. If a blocking stage exceeds this, the operation may fail unless `AllowDiskUse` is enabled. Even with disk use allowed, performance degrades significantly as data spills to disk. In a web application, this might manifest as slow page loads, API timeouts, or queries that work fine in development but fail under production load.

MongoDB processes documents in batches, not one at a time. Each stage processes a batch and forwards it. This makes pipelines efficient, but it also means:

- Expensive stages impact entire batches.
- Large batches consume more memory per stage, increasing the risk of hitting MongoDB's 100 MB per-stage limit.
- Each stage receives only the documents passed by the previous stage, so the size of each batch directly determines the cost of every stage that follows.

When a stage significantly reduces the number of documents it passes on, every downstream stage automatically processes a smaller batch. A stage that cuts 10,000 documents down to 500 means every stage that follows is operating on one twentieth of the data it would otherwise handle.

To illustrate, consider two versions of a pipeline in Leafy Library that calculates overdue loans per member. In the first version, a `$group` stage runs across all 10,000 loan records before any filtering is applied. In the second, a `$match` stage initially filters the records to the 500 currently active ones, followed by a `$group` operation. The `$group` stage in the second pipeline holds less data in memory, completes faster, and passes a smaller batch to every subsequent stage. Since all following stages inherit this smaller batch size, the efficiency gains carry through to the end of the pipeline.

How MongoDB optimizes your pipeline

Before execution, the MongoDB query optimizer (built into all drivers and ODMs) analyzes your pipeline and may automatically:

- **Reorder stages where doing so does not change the result**. For example, a `$match` that filters on original document fields may be moved earlier in the pipeline. However, a `$match` that references a field added by `$addFields` cannot be moved earlier, as the field does not exist yet at that point.

- **Merge compatible stages to reduce the total number of operations**. Multiple consecutive `$match` stages can be combined into a single stage using `$and`. Adjacent `$sort` stages are consolidated into one, as only the final sort order matters. Multiple `$limit` stages are reduced to the smallest value.
- **Push $match and $sort stages to the front of the pipeline**, where possible, allowing indexes to satisfy them. Instead of scanning every document in the collection and then filtering, MongoDB reads only the relevant documents directly from the index.
- **Avoid loading fields from disk that are not used by any stage in the pipeline**. If a `$project` later discards certain fields, MongoDB may not read those fields from storage at all, reducing the amount of data processed from the start.

This means the pipeline MongoDB executes is not always exactly the one you wrote. For example, multiple `$match` stages may be combined, or early filters might be pushed into the query engine and executed using an index. This is why pipeline structure still matters. The clearer your intent, the easier it is for the optimizer to perform effectively.

Understanding how data moves through a pipeline, and where it can get stuck, gives you a foundation for making better design decisions. In the next section, you will look at the most effective way to put this knowledge into practice by reducing the amount of data in your pipeline as early as possible.

Reduce the amount of data early

This is the single most impactful optimization you can make. Filtering early using the `$match` stage reduces the number of documents read from disk and passed between stages, while also reducing memory usage in later stages.

In Leafy Library, your active loans pipeline begins with filtering by the current user, followed by filtering by document type, and then by returned status.

By the time you reach `$addFields` and `$project`, the dataset is already very small. A good rule to follow is to place `$match` as early as possible, ensure it can use an index, and avoid carrying unnecessary documents forward through the pipeline.

For a `$match` stage to take advantage of an index, the fields being filtered must have an index defined in your collection, and the stage must appear before any `$project`, `$group`, or `$addFields` stages that rename or transform those fields. Once a field has been reshaped, MongoDB can no longer trace it back to the underlying index. A common mistake is filtering on a computed or aliased

field introduced earlier in the pipeline, which forces a full collection scan regardless of existing indexes. Similarly, using certain operators such as `$where` or applying a `$match` after a `$lookup` on the joined data will also bypass index usage. Keeping your early `$match` conditions simple, with equality checks and range queries on indexed, unmodified fields, will give MongoDB the best opportunity to resolve the filter efficiently before any documents enter the rest of the pipeline.

Be careful with $sort and $limit

Sorting can become expensive quickly if it is not supported by an index. If you sort a large dataset without an index, MongoDB must load documents into memory and perform the sort manually.

If your dataset is large, it might exceed the 100 MB memory limit for a stage. When a `$sort` stage is immediately followed by a `$limit` stage, MongoDB can internally optimize these into a top-N sort. Instead of sorting the entire working set and discarding most of the results, MongoDB maintains a fixed-size buffer of only N documents throughout the `$sort` operation. It is worth noting that this optimization is automatically applied by the query planner, so you don't need to do anything special to enable it beyond keeping the two stages adjacent. If another stage, such as `$project` or `$addFields` is inserted between them, the query planner may not be able to combine them, and the optimization will be lost. Whenever possible, keep `$sort` and `$limit` together, and place them as early in the pipeline logic as possible, so the reduced document set is carried forward instead of the full working set.

Designing efficient $group stages

`$group` is one of the most powerful stages, but it is also one of the most memory-intensive. The cost depends on the number of unique group keys (the cardinality), the size of each grouped result, and the accumulator used.

Accumulators are operators used within a `$group` stage to compute a single value from multiple documents that share the same group key. As MongoDB processes documents through the group, each accumulator maintains an internal state that is updated for every document it encounters. The nature of that state determines how memory-intensive an accumulator is.

Some accumulators are efficient because their internal state remains small and fixed, regardless of how many documents are in the group. `$sum` and `$avg` only need to track a running total and count, while `$min` and `$max` only need to hold a single value at any given time, so memory usage remains constant throughout.

Other accumulators are more expensive because their state grows with each document processed. `$push` appends every matching value into an array, and `$addToSet` does the same while also de-duplicating entries. In both cases, the accumulator must hold all of those values in memory for the duration of the group operation. In a large dataset, this can become significant, particularly if the arrays being built are unbounded. If you're using `$push` or `$addToSet` across large datasets, you should consider whether you actually need every value, or whether a downstream `$slice` or a more targeted `$match` earlier in the pipeline could reduce the volume of data being accumulated.

In Leafy Library, grouping by genre is efficient because the number of genres is relatively small and only the count for each genre is stored. However, pushing entire documents into arrays would result in significantly higher memory usage.

Cardinality and why it matters

One of the biggest factors in `$group` performance is cardinality, which means the number of unique values being grouped. For example, grouping by genre is low cardinality (a small number of genres). In contrast, grouping by user ID is potentially high cardinality, and grouping by timestamp is extremely high cardinality.

In Leafy Library, grouping by genre works well because the number of possible values is small. If we instead grouped by something like borrowed timestamp down to the second, we would end up with almost as many groups as documents, which is far more expensive.

Avoid unnecessary $unwind and $group

A common anti-pattern observed when reviewing database usage with developers is using `$unwind` on an array, processing the documents, and then using `$group` to put them back into the original shape.

If your goal is to transform any array data within a document, it's often better to use `$map`, `$filter`, or `$reduce`. In our case, `$unwind` is necessary because the data is being grouped across documents, not just within them.

Use $lookup efficiently

`$lookup` allows you to join collections, but it should be used carefully. In Leafy Library, we use `.Lookup("books", "book._id", "_id", "bookDetails")`. This is efficient because it uses equality matching, the `_id` is indexed, and the join is selective. The key considerations to remember are to always index the foreign field, keep joined documents small, and use `$project` inside the pipeline if necessary.

If you immediately `$unwind` the result of a `$lookup`, MongoDB can optimize this internally, so there's no performance penalty for doing this. However, not all `$lookup` stages behave the same way. In our example, a simple equality match on `_id` is used, which is ideal. However, `$lookup` can become expensive when:

- The join field is not indexed
- The lookup returns large documents
- The lookup runs for every document in a large pipeline

There are also two main patterns:

- Simple lookup (`localField`/`foreignField`), which is efficient and index-friendly.
- Pipeline lookup, which is more flexible but potentially more expensive.

In most application scenarios, including Leafy Library, the simple lookup form is both sufficient and more efficient.

$facet and memory considerations

`$facet` is powerful because it allows multiple pipelines to run in parallel. However, it is a blocking stage. Each sub-pipeline runs on the same dataset and is subject to memory limits. As mentioned earlier, MongoDB limits each stage to 100 MB of memory by default. If that limit is exceeded, data might spill to disk if allowed, causing performance to degrade. In the C# driver, you can control this using `var options = new AggregateOptions { AllowDiskUse = true }`.

For Leafy Library, this is unlikely to be an issue, but it is important to bear in mind for larger datasets.

Validate assumptions with explain plans

One of the easiest mistakes to make when working with aggregations is assuming you know what is slow. Sometimes the obvious suspect really is the problem, but sometimes it is not. This is why it's worth validating assumptions using explain plans. An explain plan is a JSON response that shows the considerations the query engine took when processing and enacting the query. You do this by using `.explain()` at the end of a query request. If a pipeline becomes slow or starts to feel heavier than expected, the right response is not to guess but to inspect what MongoDB is actually doing.

This helps answer questions such as:

- *Is the initial filter using an index?* Look for `IXSCAN` versus `COLLSCAN` in the explain plan. A collection scan can indicate that an appropriate index is missing or not being used.
- *Is the sort supported efficiently?* Check whether the sort can use an index or requires additional processing.
- *How many documents reach the $group or $lookup stage?* A large number of documents reaching these stages can increase the amount of work the pipeline needs to perform.
- *Is the pipeline doing more work than expected?* Reviewing how documents move through the stages can help identify unnecessary processing.
- *How many documents are examined versus returned?* If MongoDB examines thousands of documents but returns only a few, the query may not be selective enough.
- *What is the sort behavior?* If a sort cannot use an index, MongoDB may need to perform a blocking sort, which can increase the cost of the operation.

The broad best practice is simple: *optimize based on evidence, not instinct*.

Explain plans are one of the most practical tools you have for improving performance. They allow you to move from guessing to understanding.

Data modeling considerations for aggregation pipelines

Aggregation performance is not just about pipelines, it is also about how your data is structured. Aggregation performance starts with document design. If your documents match your query patterns, your pipelines become simpler. In Leafy Library, `issueDetails` embed basic book information, which avoids the need for `$lookup` for common queries. This is intentional.

Embedding versus referencing in Leafy Library

Embedding versus referencing is a common discussion point when modeling data for any application. Embedding involves using nested objects inside a document to embed the information. It is the main implementation of the MongoDB mantra: "*Data that is accessed together should be stored together*."

Referencing is when you have documents in a separate collection that are referenced in another collection, often using `_id`. This is where `$lookup` is often most used, but because this is expensive as discussed earlier, it should be used sparingly.

In Leafy Library, we take a hybrid approach. We embed the book title and user info but reference the full book details when used to query genres. This allows the use of simple queries as much as possible while still enabling more complex queries.

If a pipeline feels overly complex, ask yourself or your team, "*Could this be simplified with a different document structure?*" or "*Are we repeatedly joining data that could be embedded?*" Aggregations should enhance your model, not compensate for it.

Applying the ESR guideline to optimize indexes

The **Equality**, **Sort**, **Range** (**ESR**) guideline helps you structure pipelines effectively and provides guidance for index optimization.

Equality means that fields used in exact matches come first. You likely saw this in the Leafy Library, where `UserMatch(userId)` was called to remove any documents from the pipeline that didn't match the current user. In fact, `UserMatch` is a helper method created specifically to allow the `$match` stage to be reused across multiple pipelines.

Sort means that the field used for sorting comes next in the pipeline. In Leafy Library, this was used a few times, such as with `.SortBy(i => i.DueDate)`.

Range should come last and applies to fields used in range conditions, for example, `daysToDue < 0`.

ESR is a guideline, however, not a rule. The most important principle is to design indexes and pipelines based on how your queries actually behave.

Why the ESR guideline is effective

The ESR guideline works because of how MongoDB uses indexes internally. An index is ordered, which means MongoDB can efficiently:

- Match exact values at the beginning of the index
- Maintain order for sorting
- Scan ranges at the end

If the order is incorrect, MongoDB may not be able to use the index efficiently, even if all the fields are present. For example, placing a range field before a sort field may prevent the index from supporting the sort properly. This is why the order matters just as much as the fields themselves.

Applying the ESR guideline to an aggregation pipeline

Now, you will see how this applies to Leafy Library. Over the course of this section, you will explore some of these patterns, see how they follow the ESR guideline, and learn which indexes can be applied to make the most of the guideline.

Active loans

This example uses both equality and sort. For equality, you have `user._id`, `recordType`, and `returned` matches. For sort, the results are sorted by `dueDate`. A suitable index for this query would be:

```
{ "user._id": 1, "recordType": 1, "returned": 1, "dueDate": 1 }
```

This index covers the fields used for the equality and sort parts of the ESR guideline.

User loan history

User loan history uses fewer fields in its query patterns, specifically `user._id` for equality and `borrowDate` for sorting.

A suitable index for this would be:

```
{ "user._id": 1, "borrowDate": -1 }
```

Because you want to sort by the most recently borrowed books, the `borrowDate` value is set to `-1` which means descending. It's not uncommon to have indexes on just one or two fields. In fact, every MongoDB collection has a default index on the `_id` field.

Book loan history

This is very similar to user loan history but applies to the `books` collection instead. It uses `book._id` for equality and `borrowDate` again for sorting. The index therefore looks very similar to the one used for user loan history:

```
{ "book._id": 1, "borrowDate": -1 }
```

This index allows MongoDB to efficiently find the records for a specific book and return them in order by borrowing date.

Admin borrowed books

This is another case where only two fields are considered: `recordType` for equality and `borrowDate` for sorting.

The recommended index for this will look quite familiar by now:

```
{ "recordType": 1, "borrowDate": -1 }
```

So far, we have only looked at the indexes that support queries in the user dashboard. Next, you will look at indexes that can support queries and operations across the wider application.

Optimizing indexes across the application

Some indexes can support queries and operations used throughout the application rather than on a specific page. For example, an index might support queries that fetch a list of books wherever that list is needed, as opposed to just from the Dashboard page. In this section, you will look at these application-wide indexes, starting with a special kind of index: **time-to-live (TTL)**.

```
{ "expirationDate": 1 }
```

When a user reserves a book, they have a limited amount of time to convert that reservation into a loan with the librarian. In `ReservationService.cs`, there's a `ReservationDurationHours` variable set to `12`. This is added as the field `expirationDate` in the `IssueDetail` model. This means you could use that field to create a TTL index, so that after 12 hours (or a different time if you update the value), the reservation is reversed, and the document is deleted from the collection after this time has passed.

Using TTL indexes for automatic expiration

It's worth taking a moment to discuss TTL indexes, especially in the context of Leafy Library. The TTL index on `expirationDate` is a good example of letting MongoDB handle operational housekeeping for you. If reservations should expire automatically after a certain point, a TTL index allows MongoDB to remove those documents without the application having to run its own cleanup job. While this isn't directly about aggregation performance, it is still a very useful indexing pattern for the wider application.

Another good index would be for the authors collection:

```
{ "sanitizedName": 1 }
```

Looking up the author and querying their sanitized name is also common, so it's a good idea to have an index on this field.

Finally, in the reviews collection, it's common to query by `bookId` for the review and the timestamp so the most recent reviews are surfaced first. A good index here would cover both:

```
{ "bookId": 1, "timestamp": -1 }
```

The following table provides a good summary and explanation for all the possible indexes that could be applied to the application.

Collection	Index	Why it helps
issueDetails	{ "user._id": 1, "recordType": 1, "returned": 1, "dueDate": 1 }	Supports active loans queries and sorting by due date
issueDetails	{ "user._id": 1, "borrowDate": -1 }	Supports user borrowing history sorted by most recent
issueDetails	{ "book._id": 1, "borrowDate": -1 }	Supports book-level history lookups
issueDetails	{ "recordType": 1, "borrowDate": -1 }	Supports admin borrow list with pagination
issueDetails	{ "expirationDate": 1 } with TTL	Expires reservations automatically
authors	{ "sanitizedName": 1 } (unique)	Supports fast author lookup by normalized name
reviews	{ "bookId": 1, "timestamp": -1 }	Supports book reviews ordered by most recent

Table 3.1 – Indexes present in the Leafy Library application

This table gives a clear idea of what indexes would be beneficial to have in the Leafy Library sample application, as well as what queries they support in the application. It's important to understand not just what to do, but why.

Applying indexes in Leafy Library

At the moment, these indexes are missing from the dataset. This is because when you download the sample repository, you need to provide your own MongoDB cluster and connection string, which will not include the data. Since the data is not yet imported, you can't create the indexes. As discussed in *Chapter 11*, *Appendix*, the data is available as JSON files in the `Data` folder that you can import using a tool such as `mongorestore` or MongoDB Compass.

However, the indexes themselves haven't been applied because the data is missing. Once you have this data imported and available as collections in your database, a good practice would be to apply the indexes from *Table 3.1* to the relevant collections. You probably won't see a noticeable difference in performance due to the small size of the database in MongoDB terms. Still, it's a good practice to think about why these indexes are useful and apply that thinking to real-world applications.

Once you have the data imported (so your library has books) and have applied the suggested indexes above, you will have a more performant library.

Avoid over-indexing

Indexes are valuable, but they are not free. Every additional index:

- Takes disk space
- Consumes memory
- Adds work to inserts and updates

The goal is not to index every field that appears in a pipeline. Instead, index fields that support common and important access patterns.

That is why usage patterns matter more than theoretical completeness. An index should support real queries that the application actually runs. Just like with data modeling, implement features because they suit how the application will use them, not just because you think you should.

Aggregation anti-patterns to avoid

Even well-intentioned aggregation pipelines can become inefficient as the amount of data they process grows. In this section, you will look at some common anti-patterns and how to avoid them.

1. **Applying $match too late**: This allows unnecessary data to flow through the pipeline. Always filter early.
2. **Joining on non-indexed fields**: This causes full scans in the foreign collection. Always index the `join` field.
3. **Flattening and rebuilding data**: This adds performance cost. Use array expressions where possible, such as `$map` and `$reduce`.
4. **Sorting large datasets without indexes**: This leads to memory-heavy operations. Align indexes with the sort patterns.
5. **Overusing $facet**: This can multiply costs if you are running multiple pipelines on large datasets. Ensure input data is already reduced.
6. **Grouping on high-cardinality groups**: This increases memory usage dramatically. Reconsider the grouping key or reduce the dataset earlier.

Keeping these anti-patterns in mind will help you identify potential performance issues before they become a problem. As you build aggregation pipelines, consider how much data each stage processes and whether filtering, indexing, or restructuring the pipeline could make it more efficient.

Keeping pipelines maintainable in .NET

Performance is important, but maintainability matters too. A pipeline that is difficult to read or modify can become problematic even if it performs well.

In C#, this often comes down to how the pipeline is written. A long chain of inline `BsonDocument` stages can work perfectly well, but it can also become hard to reason about and comprehend very quickly.

Some simple practices can be very helpful:

- **Reuse stage definitions when it makes sense**: In *Chapter 2*, *Add Aggregations to the Application*, the reusable `DaysToDue` stage was a good example. It centralized shared pipeline logic and reduced duplication between the active loans and summary card queries.
- **Name intermediate results clearly**: Models such as `LoanSummary`, `GenreCount`, and `MonthlyLoanCount` make the intent of the pipeline much clearer than returning anonymous or loosely shaped data everywhere.

- **Use strong typing where it improves clarity**: The C# driver's strongly typed API is very helpful for filters, projections, and sorts. For more advanced stages, raw `BsonDocument` stages are sometimes the most practical approach, and that's fine. The goal is not to avoid BSON completely but to keep the pipeline readable.

Aim for a good balance by using strongly typed filters and sorts where possible, BSON stages where the driver API would be awkward or unclear, and clear model classes for the output.

Each service method used in *Chapter 2* had a single purpose:

- Get active loans
- Get top genres
- Get monthly history
- Get summary card stats

This is a good pattern. Pipelines are easier to understand when the method surrounding them has one clear responsibility.

Summary

In this chapter, we explored how to design aggregation pipelines that are efficient, scalable, and maintainable. You learned the importance of reducing data as early in the pipeline as possible, understanding the difference between streaming and blocking stages, and using operators such as `$group`, `$lookup`, and `$facet` deliberately rather than by default. Alongside the pipeline itself, good data modeling plays an equally important role, as does applying the ESR guideline when creating indexes to ensure your queries resolve efficiently. Finally, validating performance using `explain()` and staying aware of memory limits and large dataset behavior will help you catch problems before they reach production. By combining thoughtful pipeline design, good data modeling, and effective indexing, you can build aggregation pipelines that scale with your application.

In *Part 2*, *Searching Your Data*, we'll take a look at MongoDB Search for carrying out full-text search, including what it is, how to add it to Leafy Library, and the best practices for making the most of it.

Additional resources

The following resources provide additional information to help you learn more about aggregation pipelines and get the most out of them:

- *MongoDB Aggregation Operations*: `https://mdb.link/building-modern-data-applications-with-mongodb-aggregation-docs`.
- *Analyze Query Performance*: `https://mdb.link/building-modern-data-applications-with-mongodb-query-analyzer-docs`.
- *MongoDB Aggregation Framework Masterclass: The Ultimate Guide to Data Pipelines* [YouTube]: `https://mdb.link/building-modern-data-applications-with-mongodb-agg-pipelines-yt-series`.
- *High Performance with MongoDB* [Book]: `https://mdb.link/building-modern-data-applications-with-mongodb-high-perf-book`.

Fundamentals of Data Transformation

Learn how to build aggregation pipelines to process, transform, and analyze data efficiently in MongoDB.

`https://mdb.link/badge-aggregation-press-donet`

Part 2:

Searching Your Data

In the second part of this book, you will learn how to search your data using lexical or full-text search with MongoDB Search. You will start by understanding what MongoDB Search is and why it matters for modern data applications, including how to create search indexes for the fields users are most likely to query. You will then put these concepts into practice by connecting the search bar in Leafy Library to MongoDB Search. In the final chapter of this part, you will explore best practices for getting the most out of MongoDB Search, including optimizing search indexes, working with synonyms, and analyzing search query performance. By the end of this part, you will have the conceptual and practical foundation needed to implement effective lexical search in your applications.

This part of the book includes the following chapters:

- *Chapter 4, Search Your Data with MongoDB Search*
- *Chapter 5, Add MongoDB Search to the Application*
- *Chapter 6, Best Practices for MongoDB Search*

4

Search Your Data with MongoDB Search

Search is a fundamental feature in modern applications. Whether you're browsing an e-commerce catalog, looking for a restaurant, or searching through documentation, the ability to quickly find relevant information is something users now expect by default.

Think about the last application you used that didn't have a search feature. It's rare, and when it happens, it's noticeable. As a developer, you need to not only store and retrieve data efficiently but also make that data discoverable.

MongoDB provides a powerful document database and a flexible data model. Combined with the official MongoDB C# driver, you can build robust applications that handle complex data structures with ease.

But storing data is only part of the story. To truly unlock the value of that data, you need ways to search it quickly, accurately, and in a way that feels natural to users. This is where MongoDB Search comes in.

MongoDB Search extends the core database with rich, full-text search capabilities that are deeply integrated into the platform. Instead of relying on external systems, you can build powerful search experiences directly within your MongoDB deployment. By the end of this chapter, you'll have a solid understanding of how MongoDB Search fits into your application architecture and how to start configuring it for real-world scenarios.

The following topics are covered in this chapter:

- What is MongoDB Search?
- Creating and configuring search indexes
- Managing search indexes
- Working with search analyzers

Technical requirements

This chapter is informational, so you do not need a local application, a running deployment, or any other setup to follow the discussion.

> If you would like a complete overview of the Leafy Library application architecture and implementation, refer to *Chapter 11*, *Appendix*.

What is MongoDB Search?

MongoDB Search is a full-text search capability built directly into MongoDB. It enables flexible, powerful, and performant search experiences directly within your database without needing to move data into a separate search engine.

Traditionally, implementing search in an application often meant introducing additional infrastructure, such as Elasticsearch or Apache Solr, alongside the primary database. This introduces challenges around data synchronization, adds operational overhead, and makes the overall architecture more complex. It also requires maintaining these additional tools and combining the results of queries from multiple systems.

MongoDB Search simplifies this by embedding search capabilities directly into the database layer. This means your data stays in one place as a single source of truth, your queries can combine search and data retrieval seamlessly, and overall architectural complexity is reduced.

MongoDB Search is powered by Apache Lucene, a well-known, highly performant open-source search library. MongoDB abstracts away the complexity of working with Lucene directly, giving you a developer-friendly interface through query patterns that are already familiar to you.

As a C# developer, you can easily integrate MongoDB Search with the MongoDB C# driver via aggregation pipelines, allowing you to define search stages alongside other operations.

Key features of MongoDB Search

MongoDB Search enables you to build flexible and intuitive search experiences by allowing you to:

- Perform full-text search across one or more fields, with ranking based on relevance.
- Handle fuzzy matching, useful for typos and small spelling mistakes.
- Use autocomplete to speed up the search experience as users type.
- Search across nested documents and arrays.
- Combine search queries with filters, sorting, and projections.
- Support multiple languages out of the box.

These features make MongoDB Search suitable for a wide range of use cases, including:

- Product search in an e-commerce application.
- Content discovery in blogs or other forms of knowledge bases.
- Filtering and searching listings, such as books, properties, restaurants, or events.

As you progress through this chapter, you'll see how these capabilities are configured and how they translate into practical developer workflows.

How MongoDB Search works

It's important to understand how MongoDB Search actually works. Each node in your cluster runs a separate background process called `mongot`, a Lucene-based engine responsible for building search indexes and executing search queries. You never interact with `mongot` directly; it operates transparently behind the scenes. However, knowing it exists helps explain something important. Search indexes have their own build lifecycle, their own definition format, and their own management API. To ground this in something concrete, let's return to Leafy Library.

Imagine a user types `adventure` into the library's search box. With a `$regex` query, MongoDB would scan for documents where a field contains the literal string `adventure`, returning matches in no particular order and with no indication of which result is most relevant. If the user misses the capitalization, adds an `s`, or accidentally types `adventure`, the relevant results start disappearing. MongoDB Search addresses these limitations by returning results ranked by relevance, tolerating variations in spelling, and searching across the `title`, `description`, and `genre` fields simultaneously, all within a single query.

Now that you understand what MongoDB Search is and why it belongs in your architecture, let's look at what it can actually do.

Creating and configuring search indexes

Before any `$search` query can work, a search index must exist on the collection. Unlike standard collection indexes, search indexes are not created automatically. They must be configured explicitly and separately from any other indexes on your collection.

Where to create a search index

MongoDB offers three ways to create a search index. MongoDB Compass is the point-and-click option, `mongosh` is the shell-based option, and the MongoDB C# driver is the application-code option. The best choice depends on whether you are exploring interactively, scripting repeatable setup, or wiring index creation into your application.

MongoDB Compass

The most accessible starting point for index management is MongoDB Compass, the free **graphical user interface (GUI)** tool. Search indexes can be created from the **Indexes** tab of a collection using **Create Search Index**, where the JSON mapping is defined. Compass also displays the status of an index while it is being built.

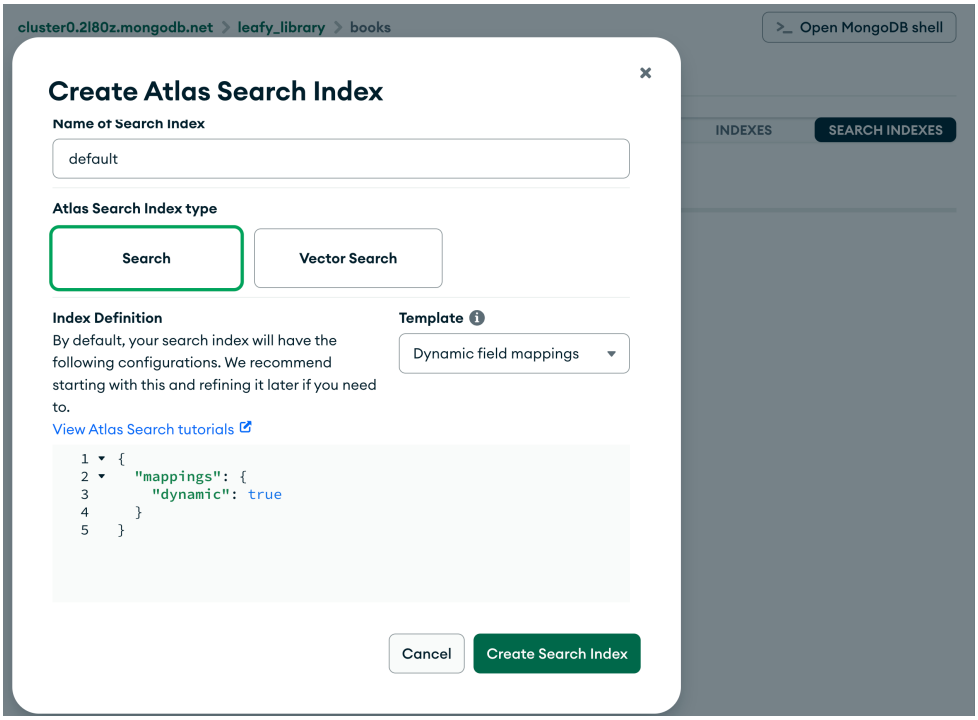


Figure 4.1 – Search index definition in MongoDB Compass

mongosh

If you prefer working in the shell/CLI, especially for scripting the setup, `createSearchIndex()` lets you create an index directly from the command line:

```
db.books.createSearchIndex("default",

  { mappings: { dynamic: true }}

)
```

This provides a convenient option when working directly from the command line or automating the setup through scripts.

MongoDB C# driver

If you want to create the index from application code, the C# driver exposes the `SearchIndexes` property on `IMongoCollection<T>`. This is the option to use when the index definition should be part of your deployment or seed logic.

Choosing a mapping strategy

Once you know how you will create the index, the next decision is how MongoDB should map the fields in your documents. Your two options are dynamic and static mapping.

Dynamic mapping

With dynamic mapping (“`dynamic": true`) you do not have to list every field in the collection. MongoDB Search discovers the supported field types in each document and indexes them for you, which is useful when your schema is still evolving or you want to prototype quickly.

In the JSON example below, notice that the mapping is minimal. The dynamic setting is set to true, and there are no field objects because MongoDB Search is deciding what to index automatically.

```
{
  "mappings": {
    "dynamic": true
  }
}
```

The C# example shows the same idea in driver code. The `BsonDocument` contains only the `mappings` element with dynamic set to true, so the code mirrors the JSON example rather than defining fields one by one.

```
var indexDefinition = new BsonDocument
{
    { "mappings", new BsonDocument { { "dynamic", true } } }
};
var model = new CreateSearchIndexModel("default", indexDefinition);
await booksCollection.SearchIndexes.CreateOneAsync(model);
```

Together, the two examples show the same pattern in two forms. The JSON definition is minimal, and the C# version uses the same minimal structure in code.

Dynamic mapping is excellent for getting started quickly. If you want to prototype a search feature or explore what MongoDB Search can do without investing time in schema planning, this is the right choice.

The trade-off is index size. A dynamic index includes every field in every document, including fields you'll never search against. For wide documents or large collections, this results in a larger index with higher memory usage. For production workloads, static mapping is usually a better fit.

typeSets

Dynamic mapping with `dynamic: true` indexes all supported field types automatically, but sometimes you want more control over which field types get indexed without having to list every field explicitly. That's where `typeSets` come in.

A `typeSet` is a named configuration that tells MongoDB Search which BSON field types to index automatically and how to map them. Instead of setting `dynamic` to `true`, you reference a `typeSet` by name, and only the field types you've defined in that `typeSet` will be indexed dynamically.

For example, in the Leafy Library, you might want to dynamically index string fields such as `title`, `description`, and `genre` without also indexing every `boolean` or `date` field in the document. A `typeSet` lets you do exactly that.

Here's an index definition using a `typeSet` to index only `string` and `number` fields:

```
{
  "mappings": {
    "dynamic": {
      "typeSet": "myTypeSet"
    }
  },
  "typeSets": [
    {
      "name": "myTypeSet",
      "types": [
        { "type": "string" },
        { "type": "number" }
      ]
    }
  ]
}
```

In C#, the `typeSet` array sits alongside `mappings` in the top-level definition, and `dynamic` becomes a `BsonDocument` referencing the `typeSet` by name rather than a simple `boolean`:

```
var indexDefinition = new BsonDocument
{
    { "mappings", new BsonDocument
        {
            { "dynamic", new BsonDocument("typeSet", "myTypeSet") }
        }
    },
    { "typeSets", new BsonArray
        {
            new BsonDocument
            {
                { "name", "myTypeSet" },
                { "types", new BsonArray
                    {
                        new BsonDocument("type", "string"),
                        new BsonDocument("type", "number")
                    }
```

```
                }
            }
        }
    }
};

var model = new CreateSearchIndexModel("default", indexDefinition);
await booksCollection.SearchIndexes.CreateOneAsync(model);
```

A few things worth noting:

- `typeSet` is optional; you only need it when you want to customize dynamic indexing behavior.
- Each `typeSet` must have a `name` and a `types` array, and each BSON type can only appear once within a `typeSet`.
- Not all BSON types are supported. `Document`, `embeddedDocuments`, `vector`, and `deprecated` field types cannot be configured this way.

A `typeSet` provides a middle ground between the convenience of dynamic mapping and the precision of static mapping. It offers the benefit of automatic indexing while limiting it to only the field types your search experience needs.

Static mapping

With static mapping (“`dynamic": false`), you explicitly define which fields to index. The index only includes what you tell it to include.

Here’s an example definition that indexes only the `title` and `description` fields in a `books` collection:

```
{
  "mappings": {
    "dynamic": false,
    "fields": {
      "title": { "type": "string" },
      "description": { "type": "string" }
    }
  }
}
```

As you can see, setting `dynamic` to `false` requires you to provide a `fields` object where you explicitly define the name of the fields and their data types. The result is a smaller, more focused index that covers only the fields your queries need. The trade-off is that any field not listed in the definition is invisible to `$search`, and will not be matched, regardless of what the query asks for. Hence, static mapping requires upfront planning to determine which fields your search experience needs to support.

Static mapping is also required for certain features. For example, the autocomplete feature, which is explored in more detail in the following section, requires its fields to be indexed with the `autocomplete` type, which can be configured only in a static definition or with `typeSet`. Although mapping fields dynamically to autocomplete is not recommended, dynamic autocomplete can apply `tokenization` settings to many string fields that are not user-facing search inputs, which increases index size and can introduce noisy suggestions from low-value fields.

> `SearchIndexes` on `IMongoCollection<T>` is a dedicated API for search index management, entirely separate from indexes used for standard MongoDB indexes. We'll explore it in more detail in *Chapter 5*, *Add MongoDB Search to the Application*.

Although both fields defined in the search index above are strings, static mapping in MongoDB Search actually supports many data types. The following table summarizes the supported data types and their corresponding MongoDB Search field types:

BSON type	**MongoDB Search field type**	**Notes/operators**
Boolean	`boolean`	Supports `equals` and `in`
Date	`date`	Supports `equals`, `facet`, `in`, `near`, `range`
Date	`date`	Used for `facet`
Double	`number`	Supports `equals`, `in`, `near`, `range`
Double	`number`	Used for `facet`
32-bit Integer	`number`	Supports `equals`, `in`, `facet`, `near`, `range`
32-bit Integer	`number`	Used for `facet`

BSON type	MongoDB Search field type	Notes/operators
64-bit Integer	`number`	Supports `equals`, `facet`, `in`, `near`, `range`
64-bit Integer	`number`	Used for `facet`
Object	`document`	Used for fields that are sub-documents
Object	`embeddedDocument`	Used for arrays of objects
ObjectId	`objectId`	Supports equality, in, range
String	`string`	Supports `moreLikeThis`, `phrase`, `queryString`, `regex`, `span`, `text`, `wildcard`
String	`token`	Used for `facet`
String	`autocomplete`	Used for autocomplete
String	`token`	Supports `equals`, `facet`, `in`, `range`
UUID / BSON Binary Subtype 4	`uuid`	Supports `equals` and `in`
Vector	`vector`	Used for `vectorSearch` to use lexical pre-filters
Array	`boolean`, `date`, `number`, `objectId`, `string`, `token`	Array field supports the operators for the contained element type

Table 4.1 – Most common field types in static mapping for MongoDB Search and what you can do with them

> Null values are automatically indexed; there is no separate field type for null.

The key takeaway is that static mapping gives you precise control over which fields can participate in a search. This makes it a better fit when you need a predictable index size, explicit field behavior, or features such as autocomplete. With that foundation in place, let's look at autocomplete in more detail.

Configuring autocomplete

Back in the static mapping section, we mentioned that autocomplete requires specific index configurations before it can be used. Now that you've covered static mapping and `typeSets`, you can look at what those configurations look like in practice.

To enable `autocomplete` on a field, you index it with `type: "autocomplete"` rather than `type: "string"`. The key additional option is `tokenization`, which controls how MongoDB Search breaks field values into partial tokens for matching as the user types.

There are three autocomplete strategies available:

- `edgeGram` (default) builds tokens from the left edge of each word. For example, typing `ca` matches **`cat`**, **`car`**, and **`can`**. This is the natural choice for most search-as-you-type experiences.
- `rightEdgeGram` builds tokens from the right end. This is useful for suffix matching, such as matching `ing` in `build`**`ing`** or `pdf` in a file name.
- `nGram` uses a sliding character window across the whole string. This means it matches anywhere in a word, like `ong` in `M`**`ong`**`oDB`, but it produces significantly larger indexes.

Below is a static index definition enabling autocomplete on the `title` field in Leafy Library:

```
{
  "mappings": {
    "dynamic": false,
    "fields": {
      "title": {
        "type": "autocomplete",
        "tokenization": "edgeGram",
        "minGrams": 2,
        "maxGrams": 15
      }
    }
  }
}
```

Notice that the field type changes from string to autocomplete, and the `tokenization` options control how partial matches are built.

In C#, you define it on the fields' value in the `mappings` object:

```
var indexDefinition = new BsonDocument
{
    { "mappings", new BsonDocument
        {
            { "dynamic", false },
            { "fields", new BsonDocument
                {
                    { "title", new BsonDocument
                        {
                            { "type", "autocomplete" },
                            { "tokenization", "edgeGram" },
                            { "minGrams", 2 },
                            { "maxGrams", 15 }
                        }
                    }
                }
            }
        }
    }
};

var model = new CreateSearchIndexModel("default", indexDefinition);
await booksCollection.SearchIndexes.CreateOneAsync(model);
```

This C# example uses the same shape again. The `title` field uses autocomplete, and the `BsonDocument` mirrors the JSON settings for `tokenization` and gram size.

Combining autocomplete with full text search

Sometimes, you need the same field to support both full-text search and autocomplete, so users can get suggestions as they type and still run a full search against the same data. You can achieve this by indexing a field under both types simultaneously, using an array of type definitions:

```
{
  "mappings": {
    "dynamic": false,
    "fields": {
      "title": [
```

```
        {
          "type": "string",
          "analyzer": "lucene.english"
        },
        {
          "type": "autocomplete",
          "tokenization": "edgeGram",
          "minGrams": 2,
          "maxGrams": 15
        }
      ]
    }
  }
}
```

In the above example, you can see how the `title` field is defined as an array of field types instead of a single object. This is what allows MongoDB Search to index the same data in two different ways.

The equivalent index definition can also be created using the C# driver:

```
var indexDefinition = new BsonDocument
{
    { "mappings", new BsonDocument
        {
            { "dynamic", false },
            { "fields", new BsonDocument
                {
                    { "title", new BsonArray
                        {
                            new BsonDocument
                            {
                                { "type", "string" },
                                { "analyzer", "lucene.english" }
                            },
                            new BsonDocument
                            {
                                { "type", "autocomplete" },
                                { "tokenization", "edgeGram" },
                                { "minGrams", 2 },
```

```
                                { "maxGrams", 15 }
                            }
                        }
                    }
                }
            }
        }
    }
};

var model = new CreateSearchIndexModel("default", indexDefinition);
await booksCollection.SearchIndexes.CreateOneAsync(model);
```

This gives the `title` field two representations in the index, one optimized for relevance-ranked full-text queries and one optimized for partial-match suggestions.

You can also enable autocomplete dynamically across all string fields using a `typeSet`. This is usually not the first choice because it can create autocomplete definitions for every matching string field and make the index larger than you need. It is most useful in prototypes or tightly controlled schemas where broad autocomplete coverage is intentional.

Autocomplete is not included in `dynamic: true` by default, so a `typeSet` is the only way to get dynamic autocomplete behavior:

```
{
  "mappings": {
    "dynamic": {
      "typeSet": "autocompleteSet"
    }
  },
  "typeSets": [
    {
      "name": "autocompleteSet",
      "types": [
        {
          "type": "autocomplete",
          "tokenization": "edgeGram",
          "minGrams": 2,
          "maxGrams": 15
```

```
        }
      ]
    }
  ]
}
```

In C#, this is defined inside the `typeSets` object:

```
var indexDefinition = new BsonDocument
{
    { "mappings", new BsonDocument
        {
            { "dynamic", new BsonDocument("typeSet", "autocompleteSet") }
        }
    },
    { "typeSets", new BsonArray
        {
            new BsonDocument
            {
                { "name", "autocompleteSet" },
                { "types", new BsonArray
                    {
                        new BsonDocument
                        {
                            { "type", "autocomplete" },
                            { "tokenization", "edgeGram" },
                            { "minGrams", 2 },
                            { "maxGrams", 15 }
                        }
                    }
                }
            }
        }
    }
};

var model = new CreateSearchIndexModel("default", indexDefinition);
await booksCollection.SearchIndexes.CreateOneAsync(model);
```

> Autocomplete indexes are larger than standard string indexes because of the additional tokens generated during tokenization. For large collections, consider a dedicated autocomplete index rather than combining it with your main search index.

The practical takeaway is that autocomplete is powerful, but it is worth scoping carefully. Use static `mappings` when you know which fields need it, and reserve dynamic autocomplete for cases where broader coverage is genuinely the right trade-off.

Understanding index state

After you create a search index, it does not immediately become available to start handling queries. The index enters a building state while MongoDB Search processes the collection and constructs the index. During this phase, `$search` queries against that collection return no results, as the index is not yet ready. In fact, if you query against an index name that doesn't exist, you won't get an error either. It is good to check both the state and the name if no results are being returned.

The index must reach the ready state before it can handle queries. This is one of the most common sources of confusion for developers new to MongoDB Search. It is easy to create an index, immediately run a search query, and assume something is wrong when no results are returned. To avoid this confusion, always verify that the index has finished building and reached the ready state before using it to serve search queries.

There is also a failed state, which indicates that the index build did not complete and needs investigation either by checking for issues in the index definition or the state of the cluster.

Managing search indexes

Once you have a search index in place, your work doesn't stop there. As your application evolves, you'll need to ensure your indexes are healthy, update their definitions as your search requirements change, and occasionally clean up indexes that are no longer needed. MongoDB provides tools for all of this.

Viewing index status

After creating a search index, the first thing to check is its status. MongoDB Search indexes move through a defined set of states:

- **Building**: The index is in the process of being created. Search queries return no results during this phase.
- **Ready**: The index is built and ready to handle queries.
- **Failed**: The build could not complete, so you should inspect the definition or the cluster state.

The simplest place to check status is in MongoDB Compass. On the **Indexes** tab for your collection, you will find a toggle to switch between standard indexes and search indexes. If you toggle the **search indexes** view, you will see the name, type, and status of all your search indexes. You can even expand a search index name to view the fields it's applied to.

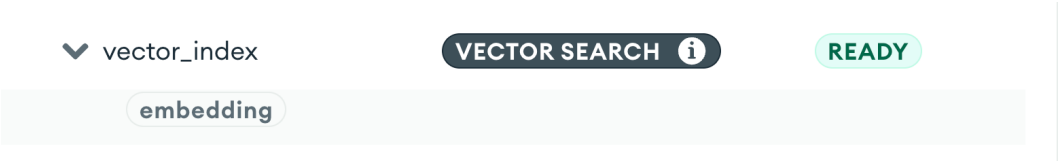


Figure 4.2 – Text search index status with indexed fields showing

To view indexes programmatically, you can list your indexes and inspect the result:

```
db.books.getSearchIndexes();
```

With the C# driver, the equivalent uses `ListAsync()` on the `SearchIndexes` property:

```
var cursor = await booksCollection.SearchIndexes.ListAsync();
var indexes = await cursor.ToListAsync();
foreach (var index in indexes)
{
    Console.WriteLine(index);
}
```

Each result includes the index name, type, and its current status. In a production application, you might use this check during startup to verify your search index is ready before enabling search features that depend on it.

Updating an index definition

As your search requirements evolve, such as when you add a new searchable field, enable autocomplete, or change how a field is mapped, you'll need to update your index definition.

When you update a search index, MongoDB rebuilds it in the background while the existing index continues to serve queries. There is no downtime or gap in search availability; the updated index seamlessly takes over once it reaches the ready state. This behavior is different from dropping and recreating a standard MongoDB index, where the old index disappears the moment you drop it.

In `mongosh`, you can update the index definition like this. Notice that the example changes the fields object, which is the part of the definition MongoDB Search will rebuild:

```
db.books.updateSearchIndex(
  "default",
  {
    mappings: {
      dynamic: false,
      fields: {
        title: { type: "string" },
        description: { type: "string" },
        genre: { type: "string" }
      }
    }
  }
);
```

In C#, you define an updated definition and then pass it to `UpdateAsync()`:

```
var updatedDefinition = new BsonDocument
{
    { "mappings", new BsonDocument
        {
            { "dynamic", false },
            { "fields", new BsonDocument
                {
                    { "title", new BsonDocument("type", "string") },
                    { "description", new BsonDocument("type", "string") },
                    { "genre", new BsonDocument("type", "string") }
                }
```

```
            }
        }
    }
};

await booksCollection.SearchIndexes.UpdateAsync("default",
updatedDefinition);
```

The index will re-enter the building state while the update is applied. Queries continue running against the previous index version until the new one is ready.

Pausing and resuming

A search index can be paused when it is temporarily not needed. Although this cannot be done specifically for an index, if a cluster deployment is paused, the index will also be paused. A paused index stops updating as documents in the collection change but retains its definition, so you don't need to recreate it from scratch when you resume.

This is useful for cost management when a collection is dormant or when you're working in a development environment and don't need search to be active.

> Pausing and resuming is available through the Atlas UI and the Atlas CLI for Atlas-deployed clusters but not currently through `mongosh` or the C# driver directly.

Deleting an index

Deleting a search index is permanent; the definition is gone and must be recreated from scratch if you need it again.

In `mongosh`, you can call `dropSearchIndex` and pass it the name of the index:

```
db.books.dropSearchIndex("default");
```

In C#, you can call `DropOneAsync` with the index name:

```
await booksCollection.SearchIndexes.DropOneAsync("default");
```

Notice that the example only needs the index name because deleting the index removes the whole definition.

The C# driver follows the same minimal pattern, with `DropOneAsync()` requiring only the index name because the entire index definition is removed:

```
await booksCollection.SearchIndexes.DropOneAsync("default");
```

Now that you can create and manage search indexes, it is worth exploring another factor that shapes how these indexes work: how MongoDB Search processes and interprets the text within these indexes. This is where analyzers come in.

Working with search analyzers

How does MongoDB Search determine that `running` and `runs` should match the same query? Or that common words, such as `the` and aren't worth indexing at all? The answer is analyzers.

An analyzer is a text processing pipeline that runs on content both when a document is indexed and when certain search operators are used. Raw text goes in, and a set of tokens comes out. MongoDB Search indexes and matches these tokens at query time.

Understanding the analyzer pipeline

Every analyzer consists of three components, each handling a different part of the process.

Character filters are an optional first step, and most analyzers don't use them, but they do exist. They pre-process the raw string before it reaches the tokenizer, stripping HTML tags, normalizing special characters, or applying other text transformations. If your data contains HTML markup, a character filter can clean it out before anything else happens.

The tokenizer splits the processed string into individual tokens. A standard tokenizer splits on whitespace and punctuation. A whitespace tokenizer splits only on spaces. The choice of tokenizer determines the basic shape of your tokens.

Token filters transform the tokens after splitting. Common examples include:

- **Lowercase**: Converts all tokens to lowercase, so `Library` and `library` match the same query.
- **Stop words**: Removes common words, such as `the`, `a`, and `is`, that add noise but no real search value.
- **Stemming**: Reduces words to their base form so that `libraries` and `library` both become the same stem and match the same search term.

To see this in practice with Leafy Library, imagine feeding a book's title and description through this pipeline at index time. When a user later types `Cats` into the search box, their query goes through the same pipeline. Both `Cats` from the query and `cat` from the document emerge as the same stem, and the match is made. Without an analyzer, those two strings would be treated as entirely different tokens.

Built-in analyzers

MongoDB Search comes with a set of built-in analyzers that cover the most common cases. You don't need to build your own unless you have a specific requirement they can't meet.

The most useful ones to know are:

- `lucene.standard`: This is the default analyzer. It splits on whitespace and punctuation, lowercases all tokens, and removes common stop words. It's a solid starting point for most text fields.
- `lucene.english`: Extends `lucene.standard` with English-language stemming. Libraries, library, and librarian all resolve to the same base form and match related queries. It's the best choice for English language.
- `lucene.whitespace`: Splits on whitespace only, with no other processing. It's useful when punctuation is meaningful and should be kept as part of the token.
- `lucene.keyword`: Is able to perform regex/wildcard queries on the full string.
- `keyword`: Use this for exact-match fields, such as tags, IDs, or statuses, where you want the full value treated as one.

To apply a built-in analyzer to a field, add it to the field definition in your index mapping. The JSON example below shows `lucene.english` applied to both the `title` and `description` fields:

```
{
  "mappings": {
    "dynamic": false,
    "fields": {
      "title": {
        "type": "string",
        "analyzer": "lucene.english"
      },
      "description": {
        "type": "string",
        "analyzer": "lucene.english"
```

```
      }
    }
  }
}
```

In C#, the analyzer is included as a field in the `BsonDocument` definition for each mapped field:

```
var indexDefinition = new BsonDocument
{
    { "mappings", new BsonDocument
        {
            { "dynamic", false },
            { "fields", new BsonDocument
                {
                    { "title", new BsonDocument
                        {
                            { "type", "string" },
                            { "analyzer", "lucene.english" }
                        }
                    },
                    { "description", new BsonDocument
                        {
                            { "type", "string" },
                            { "analyzer", "lucene.english" }
                        }
                    }
                }
            }
        }
    }
};
var model = new CreateSearchIndexModel("default", indexDefinition);
await booksCollection.SearchIndexes.CreateOneAsync(model);
```

In C#, notice the same analyzer on both fields again. The `BsonDocument` mirrors the JSON example by assigning `lucene.english` to each mapped field. You will see this in action in *Chapter 5*.

Custom analyzers

Built-in analyzers may not cover every situation. If you need to strip HTML from content fields before indexing, apply custom stop words, or combine components in a way the built-ins don't support, you can define your own.

A custom analyzer sits at the top level of the index definition alongside `mappings`, inside an analyzers array. Each entry has a name and explicitly defines its three components: these are character filters, the tokenizer, and token filters.

```
{
  "analyzers": [
    {
      "name": "htmlContentAnalyzer",
      "charFilters": [
        { "type": "htmlStrip" }
      ],
      "tokenizer": {
        "type": "standard"
      },
      "tokenFilters": [
        { "type": "lowercase" },
        { "type": "stopword", "tokens": ["the", "a", "an", "and"] }
      ]
    }
  ],
  "mappings": {
    "dynamic": false,
    "fields": {
      "description": {
        "type": "string",
        "analyzer": "htmlContentAnalyzer"
      }
    }
  }
}
```

In C#, the analyzers array sits alongside `mappings` in the top-level `BsonDocument` passed to `SearchIndexes.CreateOneAsync()`:

```
var indexDefinition = new BsonDocument
{
    { "analyzers", new BsonArray
        {
            new BsonDocument
            {
                { "name", "htmlContentAnalyzer" },
                { "charFilters", new BsonArray
                    {
                        new BsonDocument("type", "htmlStrip")
                    }
                },
                { "tokenizer", new BsonDocument("type", "standard") },
                { "tokenFilters", new BsonArray
                    {
                        new BsonDocument("type", "lowercase"),
                        new BsonDocument
                        {
                            { "type", "stopword" },
                            { "tokens", new BsonArray { "the", "a", "an",
"and" } }
                        }
                    }
                }
            }
        }
    },
    { "mappings", new BsonDocument
        {
            { "dynamic", false },
            { "fields", new BsonDocument
                {
                    { "description", new BsonDocument
                        {
                            { "type", "string" },
```

```
                            { "analyzer", "htmlContentAnalyzer" }
                        }
                    }
                }
            }
        }
    }
};

var model = new CreateSearchIndexModel("default", indexDefinition);
await booksCollection.SearchIndexes.CreateOneAsync(model);
```

Custom analyzers are worth the extra work only when the built-in analyzers cannot express the text-processing rules you need. If a built-in analyzer already provides the behavior you want, use it because it is simpler to maintain and easier to understand.

The consistency rule

The analyzer you use at index time must match the analyzer used at query time. If they differ, search results will be unpredictable because terms that should match won't. This is because the tokens in the index were produced by a different pipeline than the tokens from the query.

MongoDB Search handles this automatically. By default, it applies the same analyzer at query time as the one specified in the index definition. You only need to consider this explicitly when you override the query-time analyzer, which is an advanced use case covered in *Chapter 6*, *Best Practices for MongoDB Search*.

A note on resource usage

Analyzers are applied to every document as it is indexed. Custom analyzers with multiple filters add processing overhead, and applying analyzers broadly across many fields, especially in a dynamic index, can increase RAM usage as the index grows in complexity. Keep analyzers scoped to the fields that actually need them, and choose static `mappings` when you know which fields will be searched.

With analyzers understood, you now have a complete picture of how MongoDB Search works under the hood, from the separate index lifecycle to the text processing pipeline that shapes what gets stored, matched, and returned. Let's bring it all together.

Summary

In this chapter, you explored what MongoDB Search is, why it exists, and how to start using it effectively.

You learned that MongoDB Search brings full-text search capabilities directly into the database through a separate process, `mongot`, which is built on Apache Lucene. Instead of introducing an external search engine and the operational complexity that comes with it, you can provide relevance-ranked, user-facing search within the same platform your application already uses.

The chapter introduced the core capabilities of MongoDB Search at a conceptual level before focusing on index design decisions, including when to use dynamic mapping, when to use static mapping, and how those choices affect control and maintainability.

You also learned about practical index operations across Compass, `mongosh`, and the MongoDB C# driver, including how to check index states and update, pause, resume, and delete search indexes through the SearchIndexes API.

Finally, you explored analyzers as the tokenization layer behind search behavior, compared built-in and custom analyzer options, and learned about the consistency rule that keeps index-time and query-time analysis aligned.

At this point, you should have a solid foundation in MongoDB Search concepts, index management, and analyzer strategy before moving into hands-on application implementation.

In the next chapter, you will put these concepts into practice by building MongoDB Search into Leafy Library so users can finally use the search box.

Additional resources

The following resources provide additional information to help you learn more about MongoDB Search and access the related skill badge:

- *MongoDB Search Overview*: `https://mdb.link/building-modern-data-applications-with-mongodb-search-docs`.
- *MongoDB Text Search: B-Tree vs Inverted Index, Use Search Instead (Part 1)*, by *Erik Hatcher*: `https://mdb.link/building-modern-data-applications-with-mongodb-search-series`.

Search Fundamentals

Learn the fundamentals of MongoDB Search, including indexing, lexical search, and query optimization. Build efficient full-text search experiences in MongoDB applications.

https://mdb.link/badge-search-press-donet

5

Add MongoDB Search to the Application

In the previous chapter, you learned what MongoDB Search is and how it works under the hood. You explored search indexes, analyzers, and features such as fuzzy matching, autocomplete, and facets. This chapter builds on that foundation by integrating MongoDB Search into Leafy Library.

This chapter will begin by defining what you are building and where it fits in the application. Then, you will create the search index and implement keyword search and autocomplete using the MongoDB C# driver. Next, you will add facets to filter them by genre. The chapter also covers error handling, as search index creation has some quirks that are easy to overlook if you are not expecting them. Finally, you will learn how to test the whole implementation end-to-end. By the end of this chapter, you will have a robust search experience in Leafy Library: a search bar that returns relevant, ranked results as you type, suggestions that appear before you even finish typing, and facets that help narrow down what you are looking for.

The following topics are covered in the chapter:

- Scoping a search feature for the application
- Creating a search index and adding search to the application
- Adding autocomplete
- Adding facets for further refinement
- Error handling during index creation
- Testing the new search feature

Technical requirements

To follow along with this chapter, you will need the following prerequisites:

- .NET 10 SDK installed on your machine.
- An editor or IDE such as Visual Studio, Visual Studio Code, or Rider.
- The sample repository (`https://mdb.link/devrel.book.building-modern-data-applications-with-mongodb-repo`) forked on the start branch, or if you prefer, you can run this in GitHub Codespaces.
- A MongoDB connection string added to app settings.
- MongoDB Atlas, Community Edition, or Enterprise Advanced versions will all work. If you are using Atlas, an M0 tier cluster is enough.
- A 32-character minimum JWT secret token added to app settings.
- The sample data loaded from the `Data` folder, using tools such as MongoDB Compass or the `mongorestore` command from the CLI.

The supporting code snippets for this chapter are available in the book's repository at `https://mdb.link/building-modern-data-applications-with-mongodb-codesnippets-repo`.

> If you would like a complete overview of the Leafy Library application architecture and implementation, refer to *Chapter 11*, *Appendix*.

Scoping a search feature for the application

Before you write a single line of code, let's take a look at what you'll be building. Leafy Library already lets users browse books and authors, but there is no way to search for a particular book or author. For instance, if the user wants to read a book by Agatha Christie but cannot remember the title, they have to scroll through the entire catalog. Fortunately, you can provide a better search experience.

Here are the features you're going to add in this chapter:

- **Keyword search:** Enable users to type a word or phrase and receive a ranked list of matching books by searching across title, author name, and genre.
- **Autocomplete**: Users will be able to type into the search box, and suggestions will appear, helping them find what they want without typing the full title.
- **Genre facets:** Alongside the search results, users will be able to display a breakdown of how many results fall into each genre, allowing them to filter results by selecting a genre.

Thus, you will achieve three related features, all from the same search index and search box!

Application areas you will update

Leafy Library follows a layered architecture: a frontend, and a service layer that interacts with MongoDB. Adding search means you will be working in both.

At the service layer, you will add code to the following methods in `BookService`:

- `SearchAsync (string query, int page, int pageSize)`: Runs a `$search` query across title, author name, and genre, and returns a ranked, paginated list of matching Book documents.
- `SearchCountAsync (string query)`: Returns the total number of matching results for use in pagination.
- `GetAutocompleteSuggestionsAsync (string query)`: Runs an autocomplete query and returns a short list of title suggestions.
- `GetGenreFacetsAsync (string query)`: Returns a count of matching books per genre for the current search term.
- `SearchInGenreAsync (string query, string genre, int page, int pageSize)`: Narrows results to a specific genre using a compound query.

The `BookService` class encapsulates book-related data access logic, making it the appropriate place to add search query methods.

Search index management is handled separately in `DatabaseService`. You will add an `EnsureSearchIndexAsync` method to it to create an index on startup if it does not already exist and wait for the index to become ready before proceeding.

At the Blazor layer, the `BooksCatalogue.razor` component already contains a search input. You will connect it to call `BookService` as you type and display results, suggestions, and facets.

Deciding which fields to index

Before creating the search index, you need to determine which fields should be indexed and how they will be used. In the application, users will search across the title, author name, and genre. The field will also support autocompletion on `titles` and `genre` facets for filtering. *Table 5.1* summarizes the indexing strategy for each field:

Field	Mapping type	Purpose
`title`	`string` (`lucene.english`)	Full-text keyword search with stemming
`title`	`autocomplete` (`edgeGram`)	As-you-type suggestions
`authors.name`	`string`	Search by author name
`genres`	`string`	Search within genres
`genres`	`stringFacet`	Genre count and filtering

Table 5.1: Field mappings for the search index

Notice that the `title` field appears twice, and that is intentional. The same field can have multiple mappings, and we need both `string` and `autocomplete` on the title to support both use cases. The `genres` field also gets two mappings; `string` makes it searchable as text, while `stringFacet` makes it available for grouping and counting in facet queries.

Author names live in a nested array of objects as authors is a list of `BookAuthor`, each with a `Name` property. To index a field inside an array of nested objects, you will use the `document` mapping type on the parent field and declare the child fields inside it.

You will use static mapping (`"dynamic": false`) for the search index you're about to create. This ensures that only the required fields are indexed, which keeps the index small and performant. A dynamic index would index every field on every document, which is convenient for prototyping but wasteful in production when you have fields you never intend to search.

At this point, you have a clear implementation map: what to search, how to map fields, and where each change belongs. Next, you will translate that plan into the actual MongoDB Search index definition and service-layer code.

Creating a search index and adding search to the application

This section turns the design decisions from the previous section into working code. You will first review the full JSON index definition, then implement index creation and keyword search methods in the service layer.

Before writing C#, review the full index definition as a JSON object. This is the structure you will pass to the MongoDB Search API:

```
{
  "mappings": {
    "dynamic": false,
    "fields": {
      "title": [
        {
          "type": "string",
          "analyzer": "lucene.english"
        },
        {
          "type": "autocomplete",
          "tokenization": "edgeGram",
          "minGrams": 2,
          "maxGrams": 15
        }
      ],
      "authors": {
        "type": "document",
        "fields": {
          "name": {
            "type": "string"
          }
        }
      },
      "genres": [
        {
          "type": "string"
        },
```

```
        {
          "type": "stringFacet"
        }
      ]
    }
  }
}
```

The `title` field has two entries in an array, which is how the single field is given multiple mappings. The `string` mapping uses the `lucene.english` analyzer, which handles English stemming so that searching for `libraries` and `library` produces matching results. The `autocomplete` mapping uses `edgeGram` tokenization, which generates tokens for each prefix of a word so that typing `adv` matches `adventure`.

The `authors` field uses the `document` mapping type, which tells MongoDB Search that this field contains nested objects. Inside it, `name` is declared as a `string` field, which users will actually search. When `$search` sees a path such as `authors.name`, it looks inside each element of the authors array and checks the `name` property on each one.

The `genres` field gets two mappings in an array: `string` makes it searchable as text, and `stringFacet` makes it available for grouping and counting in `$searchMeta` queries. Both mappings are needed if you want to search by genre and generate facets from the same field.

Adding EnsureSearchIndexAsync to DatabaseService

Now, let's translate that index definition into C# and create the index at application startup. You will create a new `EnsureSearchIndexAsync` method in the existing `DatabaseService` class. This method will check whether the search index already exists before trying to create it, which is important since this method is called every time the application starts. If the index already exists, the method returns immediately. If it does not, it creates the index and waits until it is queryable before returning.

To implement this, open the `DatabaseService.cs` file and update the `EnsureSearchIndexAsync` method with the following code:

```
public async Task EnsureSearchIndexAsync()
{
    const string indexName = "fulltextsearch";

    var cursor = await _booksCollection.SearchIndexes.ListAsync();
```

```
    var existing = await cursor.ToListAsync();
    if (existing.Any(i => i["name"].AsString == indexName))
        return;

    var indexDefinition = new BsonDocument
    {
        { "mappings", new BsonDocument
            {
                { "dynamic", false },
                { "fields", new BsonDocument
                    {
                        { "title", new BsonArray
                            {
                                new BsonDocument { { "type", "string" }, {
"analyzer", "lucene.english" } },
                                new BsonDocument { { "type",
"autocomplete" }, { "tokenization", "edgeGram" }, { "minGrams", 2 }, {
"maxGrams", 15 } }
                            }
                        },
                        { "authors", new BsonDocument
                            {
                                { "type", "document" },
                                { "fields", new BsonDocument
                                    {
                                        { "name", new BsonDocument { {
"type", "string" } } }
                                    }
                                }
                            }
                        },
                        { "genres", new BsonArray
                            {
                                new BsonDocument { { "type", "string" } },
                                new BsonDocument { { "type", "stringFacet"
} }
                            }
                        }
```

```
                    }
                }
            }
        }
    };

    var model = new CreateSearchIndexModel(indexName, indexDefinition);
    await _booksCollection.SearchIndexes.CreateOneAsync(model);

    _logger.LogInformation(
        "Search index '{IndexName}' creation started. Waiting for it to
become queryable...",
        indexName);

    while (true)
    {
        await Task.Delay(TimeSpan.FromSeconds(2));
        cursor = await _booksCollection.SearchIndexes.ListAsync();
        var indexes = await cursor.ToListAsync();
        var index = indexes.FirstOrDefault(i => i["name"].AsString ==
indexName);
        if (index != null && index.GetValue("queryable", false).AsBoolean)
            break;
    }

    _logger.LogInformation("Search index '{IndexName}' is ready.",
indexName);
}
```

There are a few things worth noting here. First, `_booksCollection.SearchIndexes` is a completely separate API from `_booksCollection.Indexes`, which manages regular MongoDB indexes. Search indexes have their own management API, their own build lifecycle, and their own status model.

The call to the Search management API will return something that resembles the JSON shown below. The code above checks fields such as `status` and `queryable` in that document rather than fields in the documents from the library database.

```
{

...
  "name": "searchindex",

  "type": "search",

  "status": "READY",

  "queryable": true,

  ...
}
```

You are polling on the `queryable` field rather than the `status` field. You might expect to check for a `status` value of `"READY"`, and while that works, `queryable` is the more precise signal. An index can be in a state where status is `"READY"`, but it is still being prepared to serve queries. Polling `queryable` directly is the right approach.

Third, the index definition must be passed as a `BsonDocument`. There is no typed API for declaring field mappings, analyzers, and tokenization strategies, so this part of the index management surface requires raw BSON.

The final step is to run this method during startup so that search is ready before users access the catalog page. Call `EnsureSearchIndexAsync` from `Program.cs` after `app.Build()` and before `app.Run()`:

```
var dbService = app.Services.GetRequiredService<DatabaseService>();
await dbService.EnsureSearchIndexAsync();
```

Because `EnsureSearchIndexAsync` blocks until the index becomes queryable, the application will not start accepting requests until the index is ready to serve queries. As a result, users will never encounter a search page that silently returns zero results because the index is still building. For a first-time deployment, this can add a minute or two to startup time, which is a reasonable trade-off compared to a confusing, empty user interface.

Completing SearchAsync and SearchCountAsync in BookService

With the index taken care of, let's implement the search queries in `BookService`. Unlike index management, the actual query pipeline can use the MongoDB C# driver's typed fluent API, which is cleaner than constructing raw `BsonDocument` pipelines by hand.

1. Open `BookService.cs` and add the following `using` directive at the top of the file. This makes the MongoDB Search typed builders used in the next methods available for use:

```
using MongoDB.Driver.Search;
```

2. Next, update these two methods in `BookService`, each with a specific role. `SearchAsync(...)` retrieves the current page of ranked search results. `SearchCountAsync(...)` retrieves the total match count used by pagination:

```
public async Task<List<Book>> SearchAsync(string query, int page,
int pageSize)
{
    var searchDef = Builders<Book>.Search.Text(Builders<Book>.
SearchPath.Multi("title", "authors.name", "genres"), query);
    return await _booksCollection
        .Aggregate()
        .Search(searchDef, indexName: "fulltextsearch")
        .Skip((page - 1) * pageSize)
        .Limit(pageSize)
        .ToListAsync();
}

public async Task<long> SearchCountAsync(string query)
{
    var searchDef = Builders<Book>.Search.Text(
        Builders<Book>.SearchPath.Multi("title", "authors.name",
"genres"),
```

```
            query);
    var result = await _booksCollection
        .Aggregate()
        .Search(searchDef, indexName: "fulltextsearch")
        .Count()
        .FirstOrDefaultAsync();

    return result?.Count ?? 0;
}
```

`Builders<Book>.Search.Text()` builds a `$search` stage using the text operator. You pass the user's query string and a multi-field path that tells MongoDB Search to look across `title`, `authors.name`, and `genres`. `SearchPath.Multi()` accepts string field names, so you use `"title"` rather than a lambda expression here because `Multi()` works with `SearchPathDefinition<Book>`, and string paths satisfy that implicitly without any ambiguity. Results come back ranked by relevance automatically, so you do not need to add an explicit `$sort` stage.

`SearchCountAsync` uses the same search definition but adds a `$count` stage instead of `$skip` or `$limit`. The driver's `.Count()` extension returns a single `AggregateCountResult` document, whose `Count` property contains the total number of matched documents. This is used to drive the pagination controls in `BooksCatalogue.razor`.

Notice that both methods specify `indexName:` explicitly. If omitted, MongoDB Search looks for a default index named `"default"`. Being explicit makes the code predictable and reduces debugging time if you're working with multiple indexes.

With these two methods complete, `BookService` can now return both paged results and total counts for any search term. The next step is connecting that functionality to the Blazor page.

Wiring up BooksCatalogue.razor

The service layer now supports keyword search, but users cannot benefit from it until the UI calls those methods. In this section, you will connect the existing search input in `BooksCatalogue.razor` to `BookService`, execute searches as users type, and render the resulting list.

1. Open `BooksCatalogue.razor` and inject `BookService` alongside the existing `@inject` directives at the top of the file. This makes the methods in `BookService` available to the component to be called:

```
@inject BookService BookService
```

2. Then update the search input so it triggers updates on each keystroke and calls a handler after binding updates:

```
<input type="search"
       @bind="searchQuery"
       @bind:event="oninput"
       @bind:after="OnSearchAsync"
       placeholder="Search books..." />
```

3. Next, add markup that displays results when matches exist, and a friendly message when no matches are found:

```
@if (searchResults is not null)
{
    <p>@totalResults result(s) found</p>
    <ul>
        @foreach (var book in searchResults)
        {
            <li>@book.Title — @string.Join(", ", book.Authors.
Select(a => a.Name))</li>
        }
    </ul>
}
else if (hasSearched)
{
    <p>No books found. Try a different search term.</p>
}
```

4. Add the following to the top of the @code block alongside the existing variables:

```
private string searchQuery = string.Empty;
    private List<Book>? searchResults;
    private long totalResults;
    private bool hasSearched;
    private CancellationTokenSource? _cts;
```

5. Finally, update `OnSearchAsync()` to debounce input and call the new service methods by adding the following code inside the method:

```
_cts?.Cancel();
        _cts = new CancellationTokenSource();

        if (string.IsNullOrWhiteSpace(searchQuery))
        {
            searchResults = null;
            hasSearched = false;
            await LoadBooksAsync();
            return;
        }

        try
        {
            await Task.Delay(300, _cts.Token);
            searchResults = await BookService.
SearchAsync(searchQuery, currentPage, pageSize);
            totalResults = await BookService.
SearchCountAsync(searchQuery);
            hasSearched = true;
        }
        catch (OperationCanceledException)
        {
            // User typed another character before the delay
finished — discard
        }
    }
```

`@bind:event="oninput"` tells Blazor to update `searchQuery` on every keystroke instead of waiting until the `input` field loses focus. `@bind:after="OnSearchAsync"` then executes `OnSearchAsync` after each update. Inside the handler, `CancellationTokenSource` implements debouncing, which means delaying execution briefly so that the search runs only after typing has paused. Each new keystroke cancels the previous delay, so the query runs after roughly 300 milliseconds of inactivity.

When `searchQuery` is cleared, the component falls back to `LoadBooksAsync()` which is the existing method that loads the unfiltered book list. This way, the catalog view is restored when the search is dismissed.

With that, you now have a working keyword search across titles, author names, and genres. Run the application and test the search functionality before moving on. Try searching for a title, an author, or a genre, and all three should return results.

Adding autocomplete

Keyword search is now in place, so the next improvement is helping users find titles faster with as-you-type suggestions. In this section, you will first add an autocomplete method to `BookService`, then wire the suggestions dropdown into `BooksCatalogue.razor` so users can select a suggestion and immediately run a full search.

Adding GetAutocompleteSuggestionsAsync to BookService

This part implements the service method that returns a short list of title suggestions as a user types. The `using MongoDB.Driver.Search;` directive added earlier also provides `SearchAutocompleteTokenOrder`, which you will use here.

Add the following method to `BookService` to query the `title` field with the autocomplete operator and return just the suggested title strings:

```
public async Task<List<string>> GetAutocompleteSuggestionsAsync(string
query)
{
  var searchDef = Builders<Book>.Search.Autocomplete(
            "title",
            query,
            SearchAutocompleteTokenOrder.Any);

    return await _booksCollection
        .Aggregate()
        .Search(searchDef, indexName: "fulltextsearch")
        .Limit(5)
        .Project<string>(Builders<Book>.Projection.Expression(x =>
```

```
x.Title))
        .ToListAsync();
}
```

The `Builders<Book>.Search.Autocomplete()` method builds a `$search` stage using the `autocomplete` operator. The first argument is the indexed field path (`"title"`), and the second is the user's query text. `SearchAutocompleteTokenOrder.Any` allows token matches in any order, so inputs such as `dark mystery` and `mystery dark` can produce the same suggestion set.

You can cap the results at five with `.Limit(5)` because having more than that makes the dropdown cluttered. The `.Project<string>()` call returns only titles as a `List<string>`, which is preferable here because the UI needs plain text labels, not partial `Book` objects that would increase payload size and require additional mapping work.

Updating BooksCatalogue.razor to display autocomplete suggestions

With the service method ready, you can now render and use suggestions in the UI. The following updates add a dropdown, load suggestions inside `OnSearchAsync`, and handle suggestion clicks so users move from partial input to a full result set in one step.

1. Add the following `suggestions` dropdown to `BooksCatalogue.razor`, just above the results list:

```
@if (suggestions.Any())
{
    <ul class="suggestions">
        @foreach (var suggestion in suggestions)
        {
            <li @onclick="() => SelectSuggestionAsync(suggestion)">@
suggestion</li>
        }
    </ul>
}
```

2. Add the `suggestions` variable and `SelectSuggestionAsync` method to the `@code` block, and update `OnSearchAsync` to also fetch suggestions:

```
@code {
    private List<string> suggestions = [];
```

```
    private async Task OnSearchAsync()
    {
        _cts?.Cancel();
        _cts = new CancellationTokenSource();

        if (string.IsNullOrWhiteSpace(searchQuery))
        {
            searchResults = null;
            suggestions = [];
            hasSearched = false;
            await LoadBooksAsync();
            return;
        }

        try
        {
            await Task.Delay(300, _cts.Token);
            suggestions = await BookService.
GetAutocompleteSuggestionsAsync(searchQuery);
            searchResults = await BookService.
SearchAsync(searchQuery, currentPage, pageSize);
            totalResults = await BookService.
SearchCountAsync(searchQuery);
            hasSearched = true;
        }
        catch (OperationCanceledException) { }
    }

    private async Task SelectSuggestionAsync(string title)
    {
        searchQuery = title;
        suggestions = [];
        searchResults = await BookService.SearchAsync(title, 1,
pageSize);
        totalResults = await BookService.SearchCountAsync(title);
        hasSearched = true;
    }
}
```

When a suggestion is clicked, `SelectSuggestionAsync` sets `searchQuery` to the selected title, clears the dropdown, and initiates a new search from page 1. The suggestion list disappears, and the user immediately sees the complete matching result set.

At this stage, users can discover titles with minimal typing. Next, you will add facets so users can refine broad result sets by genre.

Adding facets for further refinement

Search results are useful, but facets make them actionable. Instead of presenting a flat list of thirty books that match `space`, facets group the results into meaningful categories, such as 18 Sci-Fi books, 7 Non-Fiction books, and 5 Children's books, allowing users to quickly narrow down the results. This difference turns searching from merely finding matches to effectively browsing them.

The concept of facets was introduced in *Chapter 4*, *Search Your Data with MongoDB Search*. In this section, you will learn how to implement them to categorize and filter search results.

$searchMeta or $search with facets?

MongoDB Search provides you with two ways to retrieve facet data:

- `$searchMeta` returns only facet metadata, which includes the count for each facet bucket, and does not return matching documents. This option is appropriate when you want to load facet data in a separate request, simplifying each query and making its response easy to deserialize.
- `$search` with a facet collector returns both documents and facet metadata in a single query. This reduces the number of round trips; however, it produces an unusual response shape that requires additional parsing.

For Leafy Library, you will use `$searchMeta` and make two separate calls: one for search results (which is already present) and one for facet counts. This requires a little more work at the network level, but the code remains clean, and each method performs a single function.

Adding GetGenreFacetsAsync to BookService

The facet query uses a raw `BsonDocument` pipeline rather than the typed fluent API. The `$searchMeta` stage with facet collectors returns a response shape that requires some `BsonDocument` parsing to extract the bucket data. For a chapter on search, the extra complexity of the fully typed facet builder would obscure the point. Therefore, the raw approach is more direct.

In this section, you will build the pipeline and pass the query to the database to return the genre facets relevant to the users' search.

Add the following method to `BookService` to execute a facet query and transform buckets into a dictionary that the UI can render:

```
public async Task<Dictionary<string, int>> GetGenreFacetsAsync(string
query)
{
    var pipeline = new BsonDocument[]
    {
        new BsonDocument("$searchMeta", new BsonDocument
        {
            { "index", "fulltextsearch" },
            { "facet", new BsonDocument
                {
                    { "operator", new BsonDocument
                        {
                            { "text", new BsonDocument
                                {
                                    { "query", query },
                                    { "path", new BsonArray { "title",
"authors.name", "genres" } }
                                }
                            }
                        }
                    },
                    { "facets", new BsonDocument
                        {
                            { "genreFacet", new BsonDocument
                                {
                                    { "type", "string" },
                                    { "path", "genres" },
                                    { "numBuckets", 10 }
                                }
                            }
                        }
                    }
```

```
                }
            }
        })
    };

    var result = await _booksCollection
        .Aggregate<BsonDocument>(pipeline)
        .FirstOrDefaultAsync();

    if (result == null) return [];

    var buckets = result["facet"]["genreFacet"]["buckets"].AsBsonArray;

    return buckets.ToDictionary(
        b => b["_id"].AsString,
        b => b["count"].ToInt32());
}
```

The `$searchMeta` stage returns a single document. Inside it, the facet field contains one entry per named facet that you defined. Inside `genreFacet`, the buckets array holds one object per unique genre found in the matching results, each with an `_id` (the genre name) and a `count` (how many matching books have that genre). This is flattened into a `Dictionary<string, int>` that is easy to iterate over in the UI.

Updating BooksCatalogue.razor to display genre facets

Now that `BookService` can return facet counts, you can render those counts on the catalog page and present them as filter controls.

1. Update `OnSearchAsync` in the `@code` block to also fetch facets:

   ```
   genreFacets = await BookService.GetGenreFacetsAsync(searchQuery);
   selectedGenre = null;
   ```

2. Add the facets sidebar to `BooksCatalogue.razor`, wrapping the existing results in a layout div:

   ```
   <div class="search-layout">
       <aside class="facets">
           <h3>Filter by genre</h3>
   ```

```
        @if (genreFacets.Any())
        {
            <ul>
                @foreach (var (genre, count) in genreFacets)
                {
                    <li>
                        <button @onclick="() =>
ApplyGenreFilterAsync(genre)"
                                class="@(selectedGenre == genre ?
"active" : "")">
                            @genre <span>(@count)</span>
                        </button>
                    </li>
                }
            </ul>
        }
    </aside>

    <main class="results">
        @* existing results markup goes here *@
    </main>
</div>
```

3. Add the `genreFacets` and `selectedGenre` fields to the code block. You will add `ApplyGenreFilterAsync` in the next subsection, after implementing the corresponding service method:

```
private Dictionary<string, int> genreFacets = [];
private string? selectedGenre;
```

At this point, users can see facet counts per genre alongside search results. The remaining step is to make those facet buttons apply an actual filtered search.

Adding SearchInGenreAsync to BookService

Now that you have implemented facet counts, the next step is to make them interactive. In this section, you'll implement the ability to filter results by genre. When a user clicks the genre button, the application should rerun the search and narrow down the results to only the selected genre. To achieve this, you will use a compound query that combines the text search with a filter clause.

To implement this functionality, add the following method to `BookService`:

```
public async Task<List<Book>> SearchInGenreAsync(string query, string
genre, int page, int pageSize)
{
    var searchDef = Builders<Book>.Search.Compound()
       .Must(Builders<Book>.Search.Text(
    Builders<Book>.SearchPath.Multi("title", "authors.name", "genres"),
    query))
.Filter(Builders<Book>.Search.Text("genres", genre));

    return await _booksCollection
        .Aggregate()
        .Search(searchDef, indexName: "fulltextsearch")
      .Match(b => b.Genres != null && b.Genres.Contains(genre))
        .Skip((page - 1) * pageSize)
        .Limit(pageSize)
        .ToListAsync();
}
```

The compound operator combines multiple conditions. The `must` clause defines the core text match that every result must satisfy. The `filter` clause narrows those results to the selected genre without affecting their relevance scores. In other words, a Sci-Fi book that matches the search query retains the same relevance score regardless of whether a genre filter is applied, and filtering narrows results without reordering them.

`Builders<Book>.Search.Compound()` builds the compound definition through chained `.Must()` and `.Filter()` methods that accept arrays of search definitions. The genre filter uses a text search against the `"genres"` path, which works because we gave genres a string mapping in the index.

With `SearchInGenreAsync` complete, the backend now supports genre-specific filtering for any search term.

Adding ApplyGenreFilterAsync to BooksCatalogue.razor

This final UI step connects facet button clicks to the new `SearchInGenreAsync` service method.

Add the `ApplyGenreFilterAsync` method to the `@code` block in `BooksCatalogue.razor`:

```
private async Task ApplyGenreFilterAsync(string genre)
{
    selectedGenre = genre;
```

```
    searchResults = await BookService.SearchInGenreAsync(searchQuery,
genre, 1, pageSize);
}
```

`ApplyGenreFilterAsync` records the currently selected genre and refreshes the result list using the filtered query, so facet selection becomes immediately visible in the UI.

At this point, the application supports keyword search, autocomplete, and genre facets, all working together. However, before you test the application, let's implement error handling to ensure that the application responds gracefully when things go wrong.

Error handling during index creation

Search introduces several failure scenarios that are worth handling explicitly. These are less common in regular CRUD workflows, where operations usually fail loudly with validation or command errors. Search can also fail in ways that resemble 'empty data,' so handling these cases prevents confusing behavior for users and developers.

How EnsureSearchIndexAsync already protects us

One of the most common issues that you might encounter occurs immediately after creating a search index. Running a query against the index before it has finished building returns zero results. There's no error or exception; it simply appears as if there are no books in the database.

What's actually happening is that the index is still in the BUILDING state. MongoDB Search is processing your documents in the background. Until the index reaches a queryable state, `$search` queries return nothing.

You already addressed this issue in the *Adding EnsureSearchIndexAsync to DatabaseService* section by implementing `EnsureSearchIndexAsync` to poll the queryable field on the index document and block until it becomes true. By calling this at startup, before the application starts serving requests, you avoid reaching the catalog page while the index is still building. This solves the problem at the infrastructure layer, rather than scattering the solution throughout query code.

You also check for the index's existence before attempting to create it. This makes `EnsureSearchIndexAsync` safe to call on every startup, avoiding the accumulation of duplicate indexes or throwing exceptions on subsequent runs.

Catching MongoCommandException in BookService

Even when the index is ready, network issues or unexpected states might cause `$search` queries to throw a `MongoCommandException`. To handle this, catch the exception at the service layer so it does not bubble up later as unhandled errors.

To do this, wrap the pipeline execution in `SearchAsync` with a `try/catch`:

```
public async Task<List<Book>> SearchAsync(string query, int page, int
pageSize)
{
    try
    {
        var searchDef = Builders<Book>.Search.Text(
    Builders<Book>.SearchPath.Multi("title", "authors.name", "genres"),
    query);

        return await _booksCollection
            .Aggregate()
            .Search(searchDef, indexName: "fulltextsearch")
            .Skip((page - 1) * pageSize)
            .Limit(pageSize)
            .ToListAsync();
    }
    catch (MongoCommandException ex) when (ex.Message.Contains("index not
found"))
    {
        _logger.LogWarning("Search index not found when running query:
{Message}", ex.Message);
        return [];
    }
    catch (MongoCommandException ex)
    {
        _logger.LogError(ex, "Search query failed unexpectedly");
        return [];
    }
}
```

The first `catch` uses a `when` filter to specifically handle the case where the index is missing, returning an empty list and logging a warning rather than crashing. The second `catch` handles any other `MongoCommandException` as a genuine error worth logging at the error level.

In both `catch` blocks, the method returns an empty list. This provides the Blazor component with a safe, predictable value to render and avoids unhandled exception pages.

Updating BooksCatalogue.razor to handle search errors

After handling exceptions in the service layer, the UI should present a clear outcome state when no results are available, whether that's due to a true zero-match query or a handled search exception.

Update the results section in `BooksCatalogue.razor` to handle empty results gracefully after an error:

```
@if (searchResults is { Count: > 0 })
{
    <p>@totalResults result(s) found</p>
    <ul>
        @foreach (var book in searchResults)
        {
            <li>@book.Title — @string.Join(", ", book.Authors.Select(a =>
a.Name))</li>
        }
    </ul>
}
else if (hasSearched)
{
    <p>No books found. Try a different search term.</p>
}
```

The `No books found. Try a different search term.` message now covers both genuine zero-result searches and the empty-list case from a caught `MongoCommandException`. In both situations, the current query produces no displayable results, so the message remains accurate for users.

Testing the new search feature

With the implementation complete, this section validates the feature set end to end. You will first verify the primary user flow and then run targeted edge-case checks.

The golden path

Use the following sequence to validate the core experience:

1. Start the application and watch the startup logs. Confirm that you see index creation or index-exists confirmation, followed by confirmation that the index is queryable.
2. Open the catalog and search for a known title. After the 300 ms debounce delay, verify that ranked results appear and the closest match is near the top.
3. Type a few characters of a known title. Verify that autocomplete suggestions appear before the full input is completed.

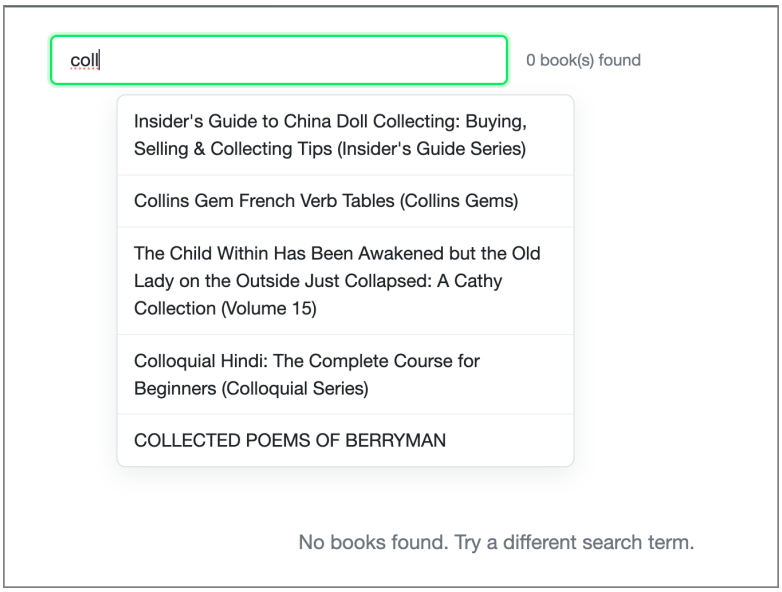


Figure 5.1 – Fully implemented search showing autocomplete

4. Search for a known author name. Verify that matching books appear, confirming that the `authors.name` mapping works.
5. Search for a term that spans multiple genres, then click a genre facet. Verify that results narrow to that genre and counts align with displayed items.

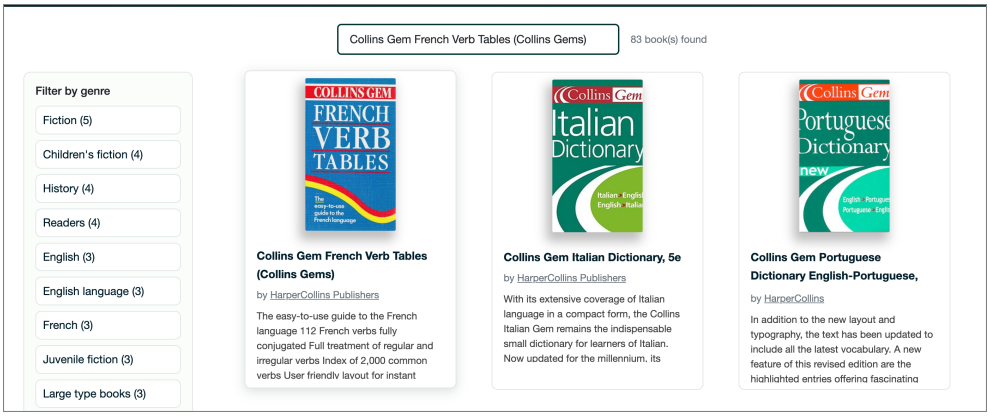


Figure 5.2 – Fully implemented search showing results and genres

If all of these checks pass, the primary flow is working as expected.

Edge cases worth testing

Run these additional checks to verify behavior under less ideal inputs and operational states.

- **Misspelled search terms**: Deliberately type a misspelled title, for example, `Haary Potter` instead of `Harry Potter`. With our current index, this will return no results because you haven't enabled fuzzy matching. If you want typo tolerance, add a fuzzy option to the text operator in the search definition:

  ```
  var searchDef = Builders<Book>.Search.Text(
      Builders<Book>.SearchPath.Multi("title", "authors.name",
  "genres"),
      query,
      fuzzy: new SearchFuzzyOptions());
  ```

  The default `SearchFuzzyOptions` allows for one character of edit distance, which is usually enough to catch common typos without returning too many false positives.

If you want typo tolerance, add `fuzzy: new SearchFuzzyOptions()` to the text operator in `SearchAsync` and `SearchCountAsync`. With it enabled, the misspelling test above will return results instead.

- **No results**: Search for a term that is unlikely to match anything. The `No books found` message should appear instead of a blank page or a spinner that never goes away.

- **Empty or short search query**: The guard clause in `OnSearchAsync` clears the results and falls back to the catalogue view, so no search query is fired. Verify that no network request is made and no loading state appears.
- **Startup with a building index**: Under normal behavior, this should not be reachable because startup is blocked until the index becomes queryable. To simulate the scenario for testing, temporarily remove or bypass the polling loop in `EnsureSearchIndexAsync`, then confirm early search queries return no results until indexing completes.

Debugging when results are not what you expect

Search does not always behave the way you expect the first time. Here is a systematic approach to the most common problems.

Results you expect are missing

Check the index definition first. If dynamic is false and the field you are searching is not listed in the index, it will not be searched. Open the Atlas UI or Compass, locate your index, and review the field mappings. Also, verify that you passed the path correctly in the search definition; a typo in `authors.name` will silently return no matches for that field.

Check the analyzer next. The `lucene.english` analyzer on the title means a search for `history` will match documents containing `histories` and `historical`. If you are not seeing expected matches, try searching for the exact word that appears in the field to confirm whether the index is working, then adjust from there.

Too many irrelevant results

This is often a sign of a dynamic index. If you accidentally created the index with `dynamic: true`, it indexes every field on every document, which can cause unexpected matches against fields you did not intend to search. Switch to static mapping and list only the fields you care about.

Autocomplete not returning suggestions

Verify that the index definition has an autocomplete entry in the title field array, separate from the string entry. If the `autocomplete` mapping is missing, the `autocomplete` operator returns nothing; it does not fall back to text search. Update the index definition to add the mapping, wait for the index to finish rebuilding, and test again.

Facet counts look wrong

Check that genres has both `string` and `stringFacet` mappings. A `string` mapping makes the field searchable as text. A `stringFacet` mapping produces facet buckets but does not make the field searchable. If you want both, you need both mappings in an array just like you declared them in the index definition.

Using the MongoDB Search tester

The Atlas UI includes a Search tester that is useful for isolating whether an issue is in index/query behavior or in application code.

1. Open your Atlas cluster and navigate to the books collection.
2. Open the Search tab and select the index used by your application.
3. Run representative $search queries directly in the tester.
4. Compare raw tester output with what your C# pipeline returns.

This workflow helps you debug faster. If the query works in Atlas but not in the application, focus on C# query construction, projection, or UI-state handling.

If the query fails in Atlas too, focus on index mappings, analyzers, or query operator configuration.

Summary

You've come a long way in this chapter, and Leafy Library now has a proper search experience.

The chapter began by scoping the feature, identifying which fields to search, which to support autocomplete on, and which to expose as facets. Those decisions informed the index definition, which shaped every implementation step that followed.

You created a static search index with separate field mappings for text search, autocomplete, and faceting across title, author name, and genre. Each feature was then implemented in `BookService` using the MongoDB C# driver's typed fluent aggregation API. The typed builders `Builders<Book>.Search.Text()`, `Builders<Book>.Search.Autocomplete()`, and `Builders<Book>.Search.Compound()` keep the search pipeline code readable and refactoring-safe compared to raw `BsonDocument` pipelines, though index management itself still requires `BsonDocument` since there is no typed API for declaring field mappings and analyzers.

Next, you used `$searchMeta` to retrieve facet counts, and a compound query with a filter clause to apply genre filtering without affecting relevance ranking. You also added `EnsureSearchIndexAsync` to `DatabaseService` to handle index creation idempotently during startup, with a polling loop that blocks until the index is queryable. This approach solves the silent-empty-results problem before it can reach the user. Finally, you tested the implementation end to end, checking the golden path, stress-testing edge cases, and systematically debugging search results that did not behave as expected.

In the next chapter, you will explore best practices for MongoDB Search, including synonyms, tuning relevance scoring, and optimizing index performance for production workloads. This will help Leafy Library's search, and your own applications, improve from good to great.

6

Best Practices for MongoDB Search

In the previous chapter, you built a fully working search experience for Leafy Library. Users can type a query into the search box, get back a ranked list of matching books, see autocomplete suggestions as they type, and filter by genre using facets. That was no small feat, and it works!

However, "works" and "production-ready" are not quite the same thing. The search index currently includes everything it needs to, but also some things it does not. Queries return full documents when most of the time only a handful of fields are needed. The relevance ranking is decent, but a match on a book's description carries the same weight as a match on its title, which is not the desired behavior. If a user types `sci-fi` instead of `Science Fiction`, they will get nothing back at all, even though the library is full of science fiction books.

These are the kinds of gaps that appear in production once real users start using a feature. This chapter addresses each of them. It covers how to structure search indexes for efficiency, how to shape queries to run faster, how to understand and tune relevance so the best results rise to the top, and how to use synonyms to bridge the gap between what users type and what the data actually says.

The following topics are covered in this chapter:

- Optimizing your search indexes in MongoDB Search
- Optimizing query search
- Analyzing search results and tuning relevance to improve results quality
- Using synonyms to improve the relevance of search results

Technical requirements

This chapter is primarily conceptual to show best practices. The code snippets are included to illustrate key ideas in context, not as a required implementation exercise.

If you want to run the examples yourself, you will need the following:

- A MongoDB deployment (MongoDB Atlas or local) with the `books` collection loaded (the data is available in the `Data` folder of the sample repository).
- .NET 8 SDK or later.
- MongoDB C# Driver package (`MongoDB.Driver`).
- `mongosh` for running JSON-based aggregation examples.

The supporting code snippets for this chapter are available in the book's repository: `https://mdb.link/building-modern-data-applications-with-mongodb-codesnippets-repo`.

> If you would like a complete overview of the Leafy Library application architecture and implementation, refer to *Chapter 11*, *Appendix*.

Optimizing your search indexes in MongoDB Search

When you created the Leafy Library search index in *Chapter 5*, *Add MongoDB Search to the Application*, you made some solid choices: static mapping, a handful of targeted fields, and the right analyzers for the fields that matter. Now that the feature is live and the catalog can potentially grow, it's worth stepping back and thinking more deliberately about what a performant search index actually looks like, and why the decisions made at index-definition time affect every query that is run against it.

Why index structure matters for performance

A MongoDB Search index is not simply a lookup table. For every document in the collection, MongoDB Search processes the indexed fields through the configured analyzers, generates tokens, and stores them in an inverted index alongside relevance metadata. When a query arrives, MongoDB Search performs the reverse: it tokenizes the query, looks up matching tokens across all indexed documents, and scores each match.

The size and complexity of that inverted index are directly proportional to the number of fields indexed and the volume of text those fields contain. A bloated index means more data to read on every query, more memory pressure on the server, and longer index rebuild times whenever the definition changes. A lean, focused index means faster queries, cheaper updates, and predictable behavior as the catalog grows.

Static versus dynamic indexing

In *Chapter 5*, `dynamic: false` was chosen for the Leafy Library index, with a brief explanation of why dynamic indexing is not well-suited for production. The following discussion should make that argument more concrete.

A dynamically mapped index tells MongoDB Search to automatically index every supported field in every document. This sounds convenient, but consider what it means for a `Book` document. The `description` field might contain hundreds of words. The `isbn` field is a string that is never intended to be searched. There are also timestamps, internal IDs, and status flags that do not belong in a search index. A dynamic index will faithfully index all of them, generating tokens and storing metadata for fields that will never appear in a query.

The following example illustrates the difference in practice. A naive dynamic index definition looks like this:

```
{
  "mappings": {
    "dynamic": true
  }
}
```

The following is the focused static index defined in *Chapter 5*:

```
{
  "mappings": {
    "dynamic": false,
    "fields": {
      "title": [
        { "type": "string", "analyzer": "lucene.english" },
        { "type": "autocomplete", "tokenization": "edgeGram", "minGrams":
2, "maxGrams": 15 }
      ],
      "authors": {
```

```
        "type": "document",
        "fields": {
          "name": { "type": "string" }
        }
      },
      "genres": [
        { "type": "string" },
        { "type": "token" }
      ]
    }
  }
}
```

The static definition indexes exactly five logical elements: title text, title autocomplete tokens, author names, genre text, and genre facet data. Nothing else is included. For a library catalogue, this is all that is needed.

The general rule is: if you would never write a `$search` query against a field, do not index it. Dynamic indexing is perfectly reasonable for exploration and prototyping, but for any feature you intend to deploy and scale, static mapping is often the right choice.

Choosing the right analyzer for each field

Analyzers control how text is broken into terms, both at index time and query time. Using the wrong analyzer is one of the most common causes of missed results in MongoDB Search, and the problem is almost always invisible because the query does not throw an error; it just returns fewer results than it should.

The following table summarizes the analyzers used in Leafy Library and the reasons for each choice:

Field	Analyzer	Why
`title` (string mapping)	`lucene.english`	Handles English stemming: `libraries` and `library` both match.
`authors.name`	`lucene.standard`	General-purpose tokenization; names do not need language-specific stemming.
`genres` (string mapping)	`lucene.standard`	Genre values such as `Science Fiction` should be tokenized and lowercased.

Table 6.1 – Analyzers used in the Leafy Library sample application

One important aspect is that the analyzer used at index time and the analyzer used at query time should match. By default, MongoDB Search uses the same analyzer for both, which is the desired behavior. However, if you configure a custom query-time analyzer that differs from the index-time analyzer, tokens in the index will not match the tokens generated from the query, and results will silently disappear.

Keeping large text fields out of the index

Leafy Library's book model has a `description` field that can contain a paragraph or two of text. Adding it to the index could significantly increase the index size. Unless you are specifically building a full-text search over descriptions, which this application is not, at least not yet, it contributes more noise than signal to the results.

A common temptation is to index everything while the index definition is open. Resist this urge. Add fields to the index only when you have a specific query use case for them. Index size grows quickly and shrinks slowly; rebuilding an index after removing fields is a non-trivial operation.

That said, when you do add a content-heavy field to a search index, it is worth carefully considering not just which analyzer to use, but whether that same analyzer should also apply at query time. The following section examines when and how to configure them separately.

Override the query-time analyzer with searchAnalyzer

Chapter 4, *Search Your Data with MongoDB Search,* established that MongoDB Search applies the same analyzer at both index time and query time by default, and that diverging from this can break results. This default is the right choice for most fields, but there is a specific situation where a different search analyzer is not just acceptable but correct: when the index-time analyzer performs a cleanup step on stored content that user queries will never need.

The most common example is a field whose stored content contains markup or special characters that must be cleaned before indexing. User-typed queries will never contain that markup, so there is no reason to apply the strip step at search time.

Consider the `description` field in Leafy Library. To add it to the search index, you would need to account for the fact that descriptions in the catalogue came from a content management system and may contain HTML tags. A `description` value might look as follows:

```
<p>A classic tale of <em>adventure</em> and mystery, set in the English
countryside.</p>
```

The goal is to index the words, not the tags. At index time, the HTML is stripped before tokenization. When a user types `adventure mystery` into the search box, there is no HTML to strip. Applying that step at search time would be wasted work.

This is what `searchAnalyzer` is for. In the next example, you will see how to define two custom analyzers for the same field: one for indexing, with the HTML strip step, and one for searching, without it. The same pattern can apply in other cleanup-oriented scenarios as well, but the rule never changes. The query-time analyzer must still produce tokens that are compatible with those written into the index. Because the only difference between the two pipelines here is a step that has no effect on clean text, the tokens they produce are identical for any user input, which means the compatibility rule from *Chapter 4* is still satisfied.

First, you define both analyzers alongside the mappings in the index definition:

```
{
  "analyzers": [
    {
      "name": "description_indexer",
      "charFilters": [{ "type": "htmlStrip" }],
      "tokenizer": { "type": "standard" },
      "tokenFilters": [
        { "type": "lowercase" },
        { "type": "stemmer", "language": "english" }
      ]
    },
    {
      "name": "description_searcher",
      "charFilters": [],
      "tokenizer": { "type": "standard" },
      "tokenFilters": [
        { "type": "lowercase" },
        { "type": "stemmer", "language": "english" }
      ]
    }
  ],
  "mappings": {
    "dynamic": false,
    "fields": {
```

```
      "description": {
        "type": "string",
        "analyzer": "description_indexer",
        "searchAnalyzer": "description_searcher"
      }
    }
  }
}
```

The key line is `"searchAnalyzer": "description_searcher"`. Without it, MongoDB Search would apply `description_indexer` at query time as well, including the HTML strip step on the user's plain-text input. That step is harmless on clean text, but it makes the index definition harder to read and reason about. Being explicit is always preferable.

In C#, both analyzers go into a top-level analyzers array alongside the mappings:

```
var indexDefinition = new BsonDocument
{
    { "analyzers", new BsonArray
        {
            new BsonDocument
            {
                { "name", "description_indexer" },
                { "charFilters", new BsonArray { new BsonDocument("type",
"htmlStrip") } },
                { "tokenizer", new BsonDocument("type", "standard") },
                { "tokenFilters", new BsonArray
                    {
                        new BsonDocument("type", "lowercase"),
                        new BsonDocument { { "type", "stemmer" }, {
"language", "english" } }
                    }
                }
            },
            new BsonDocument
            {
                { "name", "description_searcher" },
                { "charFilters", new BsonArray() },
                { "tokenizer", new BsonDocument("type", "standard") },
```

```
                { "tokenFilters", new BsonArray
                    {
                        new BsonDocument("type", "lowercase"),
                        new BsonDocument { { "type", "stemmer" }, {
"language", "english" } }
                    }
                }
            }
        },
        { "mappings", new BsonDocument
            {
                { "dynamic", false },
                { "fields", new BsonDocument
                    {
                        { "description", new BsonDocument
                            {
                                { "type", "string" },
                                { "analyzer", "description_indexer" },
                                { "searchAnalyzer", "description_searcher" }
                            }
                        }
                    }
                }
            }
        }
};
```

The compatibility rule from *Chapter 4* applies here: terms produced by `description_searcher` must match terms in the index built by `description_indexer`. In this case, they do, because both pipelines produce the same token shapes for plain-text input. The only difference is that `description_indexer` also handles HTML-tagged content by stripping the tags first.

As a general rule, use `searchAnalyzer` when the index-time pipeline does something to the stored content that is unlikely to be reflected in user queries. Stripping markup is the clearest example. What you should not do is use a `searchAnalyzer` that applies different stemming or tokenization rules from the index analyzer because that will produce query terms that do not exist in the index, causing results to disappear silently.

Updating an index definition

When you need to update the search index (to add a field, change an analyzer, or add synonym mappings, covered later in this chapter), use `SearchIndexes.UpdateOneAsync()`. The process is the same as creating the index, except you pass the updated definition to an update call rather than a create call.

In `DatabaseService.cs`, you can use something like the C# code below to update your search index:

```
public async Task UpdateSearchIndexAsync(BsonDocument updatedDefinition)
{
    const string indexName = "fulltextsearch";
    await _booksCollection.SearchIndexes.UpdateOneAsync(indexName,
updatedDefinition);
}
```

MongoDB Search rebuilds the index in the background after an update. During this process, the existing index can continue to serve queries while the rebuild happens, so there is no downtime. However, wait until the index returns to a `queryable: true` state before assuming the new definition is active. If you need to wait for the rebuild to complete, you can use the same polling approach from the *Adding EnsureSearchIndexAsync to DatabaseService* section in *Chapter 5*.

Optimizing query performance

With a well-structured index in place, the next layer of improvement is the queries themselves. An efficient query does only the work it needs to: it fetches only the fields your application will use, stops scoring documents as soon as it has enough results, and keeps non-scoring conditions in the right clause so MongoDB Search can optimize them aggressively.

In this section, you will see how you can tighten the query pipeline from end to end. The examples focus on three practical levers you can apply immediately in Leafy Library: projecting only the fields the UI needs, limiting the result set as early as possible, and keeping non-scoring constraints out of the scoring path.

Project only the fields you need

By default, the `$search` aggregation stage returns complete documents: every field, including fields your application never examines. For the Leafy Library results page, only the title, authors, and genres fields are needed. Returning the full document, including description, isbn, reviews, and any other embedded data, is wasteful.

Adding a `$project` stage after `$search` solves this. In C#, the fluent aggregation API makes this straightforward:

```
public async Task<List<Book>> SearchAsync(string query, int page, int
pageSize)
{
    var searchDef = Builders<Book>.Search.Text(
        Builders<Book>.SearchPath.Multi("title", "authors.name",
"genres"),
        query);

    return await _booksCollection
        .Aggregate()
        .Search(searchDef, indexName: "fulltextsearch")
        .Limit(pageSize * page)
        .Project<Book>(Builders<Book>.Projection
            .Include(b => b.Id)
            .Include(b => b.Title)
            .Include(b => b.Authors)
            .Include(b => b.Genres))
        .Skip((page - 1) * pageSize)
        .ToListAsync();
}
```

The `$project` stage tells MongoDB to stop carrying unused fields through the rest of the pipeline. For a book with a long description and embedded review data, this can significantly reduce the amount of data transferred between MongoDB and your application. Note that `$limit` appears before `$project` in the pipeline. That ordering matters because it caps the number of documents that even reach the projection stage, so MongoDB only reshapes the subset of results your application might actually return. The following section explains why that ordering matters.

Place $limit early in the pipeline

One of the most impactful optimizations for search queries is placing `$limit` as early as possible in the pipeline, immediately after `$search` whenever the rest of the pipeline does not need to inspect or reorder the entire matching set.

Without an early `$limit`, MongoDB Search scores every document that matches the query, applies all subsequent pipeline stages to all of them, and discards the excess only at the very end. For a catalog with thousands of books, a query for a common term such as `adventure` could match hundreds of documents. If the results page shows only ten at a time, that represents a great deal of unnecessary work.

The following rule of thumb applies:

- When sorting by relevance score (the default MongoDB Search behavior), place `$limit` directly after `$search`. Where appropriate, add pagination after this stage using `$skip` or the pagination token facility.
- When sorting by a field other than relevance score, the `$search.sort` stage must come before `$limit` because the full sorted set is required to determine which documents make the cut.

For Leafy Library, results are ranked by relevance, so `$limit` goes directly after `$search`:

```
return await _booksCollection
    .Aggregate()
    .Search(searchDef, indexName: "fulltextsearch")
    .Limit(pageSize * page)     // cap early: MongoDB Search stops scoring
at this point
    .Project<Book>(...)
    .Skip((page - 1) * pageSize)
    .ToListAsync();
```

The pipeline limits to `pageSize * page` rather than `pageSize` alone so that the `$skip` stage has the documents it needs for pagination. For example, if the user is on Page 3 with 10 results per page, the top 30 documents are required before skipping the first 20.

Using `$skip` is recommended only at a low offset value. If the user requires unlimited deep pagination, use the pagination sequence token instead.

Use filter instead of must for non-scoring conditions

When a user selects a genre facet in Leafy Library, the search results are narrowed to books in that genre. In *Chapter 5*, this was implemented using a compound query with a filter clause, which was the right choice, and understanding why is important because the distinction matters at scale.

The must clause in a compound query is a scoring condition. A document that satisfies a must clause receives a positive contribution to its relevance score. The filter clause is different. Although documents must satisfy it to appear in results, it does not contribute to the score at all. MongoDB Search can therefore apply filter conditions more aggressively, treating them as hard constraints rather than relevance signals.

For genre filtering, filter is exactly the right choice. Whether a book belongs to the `Mystery` genre does not make it more relevant to the user's search query; it simply determines whether the book is in scope. Using the must clause here would subtly inflate the scores of genre-matched documents in ways that might not reflect what the user actually wants.

The following example shows what a genre-filtered search looks like in `BookService.cs`:

```
public async Task<List<Book>> SearchInGenreAsync(string query, string
genre, int page, int pageSize)
{
    var searchDef = Builders<Book>.Search.Compound()
        .Must(Builders<Book>.Search.Text(
            Builders<Book>.SearchPath.Multi("title", "authors.name"),
            query))
        .Filter(Builders<Book>.Search.Text(
            Builders<Book>.SearchPath.Single("genres"),
            genre));

    return await _booksCollection
        .Aggregate()
        .Search(searchDef, indexName: "fulltextsearch")
        .Limit(pageSize * page)
        .Project<Book>(Builders<Book>.Projection
            .Include(b => b.Id)
            .Include(b => b.Title)
            .Include(b => b.Authors)
            .Include(b => b.Genres))
        .Skip((page - 1) * pageSize)
        .ToListAsync();
}
```

The must clause ensures the query term appears somewhere relevant, such as in the title or author name. The filter clause scopes results to the selected genre without affecting the relevance signal. The result is a clean, efficient query that is straightforward to reason about.

Analyzing search results and tuning relevance to improve result quality

Efficient queries and lean indexes address speed. However, a fast search that returns the wrong results in the wrong order is not genuinely useful. This section addresses the other dimension of quality: ensuring that the most relevant results appear at the top.

To achieve that, it is necessary to understand how MongoDB Search determines what "relevant" means.

How MongoDB Search scores documents

Every document returned by a `$search` query has a relevance score, a number that MongoDB Search calculates based on how well the document matched the query. By default, results are sorted by this score, with the highest scores first.

The scoring model that MongoDB Search uses is based on BM25, a well-established algorithm in information retrieval. You don't need to understand the mathematics to use it effectively, but understanding the key factors can help you make better decisions when tuning:

- **Term frequency**: If the query term appears multiple times in a field, that document scores higher than one where the term appears only once. For instance, a book titled `The Mystery of the Mystery Island` will score higher for `mystery` than one titled `Mystery at Sea`.
- **Inverse document frequency**: If the query term is rare across the entire collection, matching documents score higher. A search for `Agatha` is more distinctive than a search for `the`, so matching documents receive a larger score boost.
- **Field length**: Shorter fields score higher for the same match. A title with four words where one matches is considered a stronger signal than a description with four hundred words where one matches.

These factors combine to produce a single score per document. As a result, documents that match in specific, distinctive, short fields tend to appear at the top of the ranked list.

Exposing relevance scores for debugging

Before you begin tuning, it's helpful to be able to see the scores that MongoDB Search is assigning. You can expose the relevance scores by adding a `MetaSearchScore` projection to the same projection definition that already includes the fields you want to return. The cleanest approach is to define a lightweight result model that includes a score field:

```
public class BookSearchResult
{
    public ObjectId Id { get; set; }
    public string Title { get; set; } = string.Empty;
    public List<BookAuthor> Authors { get; set; } = [];
    public List<string> Genres { get; set; } = [];
    public double SearchScore { get; set; }
}
```

Then update the projection in a debug method to project to this type:

```
public async Task<List<BookSearchResult>> SearchWithScoresAsync(string
query)
{
    var searchDef = Builders<Book>.Search.Text(

        Builders<Book>.SearchPath.Multi("title", "authors.name",
"genres"),
        query);

    return await _booksCollection
        .Aggregate()
        .Search(searchDef, indexName: "fulltextsearch")
        .Limit(20)
        .Project<BookSearchResult>(Builders<Book>.Projection
            .Include(b => b.Id)
            .Include(b => b.Title)
            .Include(b => b.Authors)
            .Include(b => b.Genres)
            .MetaSearchScore("searchScore"))
        .ToListAsync();
}
```

`MetaSearchScore("searchScore")` adds `{ "searchScore": { "$meta": "searchScore" } }` to the projection, which tells MongoDB to include the relevance score in the output document under the key `searchScore`. The driver maps this to the `SearchScore` property on `BookSearchResult`.

It can be useful to call this method during development whenever a query is returning unexpected results. Seeing the actual scores makes the ranking transparent. If enabled, you can also use `searchScoreDetails` to look deeper into scoring. For a deeper dive into how `searchScoreDetails` works and how to interpret its output, see the Foojay article, *MongoDB Search Score Breakdown* at `https://mdb.link/devrel.book.building-modern-data-applications-with-mongodb-searchscoredetails-blog`. You can also use the MongoDB Search tester in the Atlas UI to run queries and inspect scores interactively without writing any code.

Boosting fields to reflect their importance

By default, a match on the title field and a match on the genres field carry roughly the same weight in the relevance score, accounting for field length differences. In practice, if a user searches for `adventure`, a book titled `Adventures of Sherlock Holmes` should almost certainly rank higher than a book in the adventure genre whose title has nothing to do with the query.

You can express this preference using boost, a score modifier that multiplies the contribution of a match by a constant. Higher values mean that matches on that field matter more. The following example shows what this looks like for the Leafy Library search, using a compound query with should clauses:

```
public async Task<List<Book>> SearchAsync(string query, int page, int
pageSize)
{
    var searchDef = Builders<Book>.Search.Compound()
        .Should(Builders<Book>.Search.Text(
            Builders<Book>.SearchPath.Single("title"),
            query,
            score: Builders<Book>.SearchScore.Boost(3.0)))
        .Should(Builders<Book>.Search.Text(
            Builders<Book>.SearchPath.Single("authors.name"),
            query,
            score: Builders<Book>.SearchScore.Boost(2.0)))
        .Should(Builders<Book>.Search.Text(
            Builders<Book>.SearchPath.Single("genres"),
            query));
```

```
    return await _booksCollection
        .Aggregate()
        .Search(searchDef, indexName: "fulltextsearch")
        .Limit(pageSize * page)
        .Project<Book>(Builders<Book>.Projection
            .Include(b => b.Id)
            .Include(b => b.Title)
            .Include(b => b.Authors)
            .Include(b => b.Genres))
        .Skip((page - 1) * pageSize)
        .ToListAsync();
}
```

The single text operator has been replaced with a compound query containing three should clauses, one per field. Title matches are boosted by `3.0`, author name matches by `2.0`, and genre matches use the base score with no boost.

Should clauses work differently from must clauses. A document does not need to satisfy all should clauses to appear in the results, but it needs to match at least one. Satisfying more should clauses, or satisfying a higher-weighted one, pushes the document higher in the ranked list. This allows you to express a nuanced preference: a title match is the strongest signal, an author match is a good signal, and a genre match is a weak signal. Results that match all three will rank highest.

Compound query strategies for relevance control

The compound query operator is the primary tool for relevance tuning in MongoDB Search. It provides four distinct clause types, each with a different relationship to scoring:

Clause	Must appear?	Contributes to score?	Use case
`must`	Yes	Yes	Core search terms; a document must match to appear
`should`	No	Yes	Nice-to-have signals; boosts documents that also match
`filter`	Yes	No	Hard constraints; scopes results without affecting rank
`mustNot`	No (must not appear)	No	Exclusions; removes documents from results entirely

Table 6.2 – Compound clauses and their impact

A well-designed search query for a real application typically uses at least two of these clause types. For Leafy Library, the following compound query performs a genre-filtered search with boosted fields:

```
var searchDef = Builders<Book>.Search.Compound()
    .Should(Builders<Book>.Search.Text(
        Builders<Book>.SearchPath.Single("title"),
        query,
        score: Builders<Book>.SearchScore.Boost(3.0)))
    .Should(Builders<Book>.Search.Text(
        Builders<Book>.SearchPath.Single("authors.name"),
        query,
        score: Builders<Book>.SearchScore.Boost(2.0)))
    .Should(Builders<Book>.Search.Text(
        Builders<Book>.SearchPath.Single("genres"),
        query))
    .Filter(Builders<Book>.Search.Text(
        Builders<Book>.SearchPath.Single("genres"),
        selectedGenre));
```

The three should clauses handle relevance, while the filter clause handles scope. Neither interferes with the other. The result is a search query that ranks results sensibly and respects the user's genre selection, without any tuning compromising the filtering, or vice versa.

Using synonyms to improve the relevance of search results

There is a category of relevance problem that boosting and compound queries cannot solve: the case where the user's query and the data simply do not share vocabulary. A Leafy Library user who searches for `sci-fi` will get no results at all, not because the search logic is wrong, but because every science fiction book in the catalog uses `Science Fiction` as its genre value, not `sci-fi`. The index is correct, and the query is correct. The terminology simply does not match.

Synonyms are the answer to this class of problem.

What synonyms do

A synonym mapping tells MongoDB Search that certain terms should be treated as equivalent at query time. When a user searches for `sci-fi`, MongoDB Search expands the query to also look for `Science Fiction` and `SF`, without requiring a single document in the collection to be changed.

This is purely a query-time operation. The documents themselves are not modified, and the index structure does not change, except for a new synonym configuration reference. It is a clean, non-destructive way to bridge vocabulary gaps.

MongoDB Search supports two types of synonym mapping, and understanding the difference is important.

Equivalent synonyms treat a set of terms as interchangeable in both directions. A query for any term in the set matches documents containing any other term in the set. For example:

```
{
  "mappingType": "equivalent",
  "synonyms": ["sci-fi", "science fiction", "SF"]
}
```

With this mapping, searching for “sci-fi” matches documents containing “Science Fiction”, and searching for “Science Fiction” also matches documents containing “sci-fi”. The relationship is fully symmetric.

Explicit synonyms are one-directional. A query for an input term is expanded to include the synonyms, but the reverse is not true. For example:

```
{
  "mappingType": "explicit",
  "input": ["whodunit"],
  "synonyms": ["mystery"]
}
```

Searching for `whodunit` will expand to include `mystery`, but searching for `mystery` will not automatically find documents containing `whodunit`. Use explicit synonyms when you want to support colloquial or abbreviated input terms without broadening all searches for the canonical term.

For Leafy Library, equivalent synonyms are the right choice across the board. Genre names have multiple valid forms. `sci-fi` and `Science Fiction` are the same thing, full stop.

Creating a synonym source collection

Synonyms in MongoDB Search are stored in a MongoDB collection in the same database as the collection being searched. The synonym documents live in that collection and are referenced by the search index configuration.

In Leafy Library, a `SeedSynonymsAsync` method in `DatabaseService` handles collection creation and seeding at startup. It checks whether the collection already exists before inserting, so it is safe to call every time the application starts:

```
public async Task SeedSynonymsAsync()
{
    var synonymsCollection = _database.
GetCollection<BsonDocument>("synonyms");
    var count = await synonymsCollection.
CountDocumentsAsync(FilterDefinition<BsonDocument>.Empty);
    if (count > 0)
        return;
    var synonymDocuments = new List<BsonDocument>
    {
        new BsonDocument
        {
            { "mappingType", "equivalent" },
            { "synonyms", new BsonArray { "sci-fi", "science fiction",
"SF" } }
        },
        new BsonDocument
        {
            { "mappingType", "equivalent" },
            { "synonyms", new BsonArray { "mystery", "detective fiction",
"crime fiction", "whodunit" } }
        },
        new BsonDocument
        {
            { "mappingType", "equivalent" },
            { "synonyms", new BsonArray { "biography", "memoir", "life
story" } }
        },
        new BsonDocument
```

```
        {
            { "mappingType", "equivalent" },
            { "synonyms", new BsonArray { "fantasy", "high fantasy", "epic
fantasy" } }
        }
    };
    await synonymsCollection.InsertManyAsync(synonymDocuments);
    _logger.LogInformation("Synonym collection seeded with {Count}
mappings.", synonymDocuments.Count);
}
```

In `Program.cs`, `SeedSynonymsAsync` runs before `EnsureSearchIndexAsync`, because the index configuration references the synonyms collection by name, and that collection must exist first:

```
var dbService = app.Services.GetRequiredService<DatabaseService>();
await dbService.SeedSynonymsAsync();
await dbService.EnsureSearchIndexAsync();
```

This code block is an example of how you can ensure the collection is available so that when the search index is created, it doesn't fail to create it because the collection doesn't exist. The key idea is that the collection must exist before the search index references it.

Configure synonym mappings in the search index

With the synonyms collection in place, you can instruct the search index to use it. This means updating the index definition to include a `synonymMappings` configuration that references the collection by name.

The following example shows the updated full index definition:

```
{
  "mappings": {
    "dynamic": false,
    "fields": {
      "title": [
        { "type": "string", "analyzer": "lucene.english" },
        { "type": "autocomplete", "tokenization": "edgeGram", "minGrams":
2, "maxGrams": 15 }
      ],
      "authors": {
        "type": "document",
```

```
        "fields": {
          "name": { "type": "string" }
        }
      },
      "genres": [
        { "type": "string" },
        { "type": "stringFacet" }
      ]
    }
  },
  "synonymMappings": [
    {
      "name": "genre-synonyms",
      "analyzer": "lucene.standard",
      "source": {
        "collection": "synonyms"
      }
    }
  ]
}
```

The `synonymMappings` array can contain multiple synonym source configurations. To create separate synonym sets for titles and genres, a second entry with a different name can be added. For now, a single mapping that covers genre vocabulary is sufficient.

The `analyzer` field in the synonym mapping specifies how the synonym terms themselves are tokenized. The `lucene.standard` analyzer is used here, which lowercases and tokenizes on whitespace and punctuation. This means that `Science Fiction` becomes two tokens, `science` and `fiction`, which is exactly how it is indexed, so the expansion will match correctly.

Now update `EnsureSearchIndexAsync` in `DatabaseService.cs` to use the new index definition, replacing the existing `indexDefinition` variable with the updated version that includes `synonymMappings`:

```
var indexDefinition = new BsonDocument
{
    { "mappings", new BsonDocument
        {
            { "dynamic", false },
```

```
            { "fields", new BsonDocument
                {
                    { "title", new BsonArray
                        {
                            new BsonDocument { { "type", "string" }, {
"analyzer", "lucene.english" } },
                            new BsonDocument
                            {
                                { "type", "autocomplete" },
                                { "tokenization", "edgeGram" },
                                { "minGrams", 2 },
                                { "maxGrams", 15 }
                            }
                        }
                    },
                    { "authors", new BsonDocument
                        {
                            { "type", "document" },
                            { "fields", new BsonDocument
                                {
                                    { "name", new BsonDocument { { "type",
"string" } } }
                                }
                            }
                        }
                    },
                    { "genres", new BsonArray
                        {
                            new BsonDocument { { "type", "string" } },
                            new BsonDocument { { "type", "stringFacet" } }
                        }
                    }
                }
            }
        }
    },
    { "synonymMappings", new BsonArray
        {
```

```
            new BsonDocument
            {
                { "name", "genre-synonyms" },
                { "analyzer", "lucene.standard" },
                { "source", new BsonDocument { { "collection", "synonyms"
} } }
            }
        }
    }
};
```

If the search index already exists, it must be updated rather than created. The following example shows `EnsureSearchIndexAsync` adjusted to handle both cases:

```
public async Task EnsureSearchIndexAsync()
{
    const string indexName = "fulltextsearch";

    var cursor = await _booksCollection.SearchIndexes.ListAsync();
    var existing = await cursor.ToListAsync();
    var existingIndex = existing.FirstOrDefault(i => i["name"].AsString ==
indexName);

    var indexDefinition = BuildSearchIndexDefinition();

    if (existingIndex is null)
    {
        var model = new CreateSearchIndexModel(indexName,
indexDefinition);
        await _booksCollection.SearchIndexes.CreateOneAsync(model);
        _logger.LogInformation("Search index '{IndexName}' creation
started.", indexName);
    }
    else
    {
        await _booksCollection.SearchIndexes.UpdateOneAsync(indexName,
indexDefinition);
        _logger.LogInformation("Search index '{IndexName}' updated.",
indexName);
```

```
    }

    _logger.LogInformation("Waiting for search index to become
queryable...");
    while (true)
    {
        await Task.Delay(TimeSpan.FromSeconds(2));
        cursor = await _booksCollection.SearchIndexes.ListAsync();
        var indexes = await cursor.ToListAsync();
        var index = indexes.FirstOrDefault(i => i["name"].AsString ==
indexName);
        if (index != null && index.GetValue("queryable", false).AsBoolean)
            break;
    }

    _logger.LogInformation("Search index '{IndexName}' is ready.",
indexName);
}

private static BsonDocument BuildSearchIndexDefinition()
{
    // returns the full index definition BsonDocument as shown above
}
```

The index definition has been extracted into a separate `BuildSearchIndexDefinition` method to keep the code readable, and the startup logic has been updated to either create or update the index depending on whether it already exists. With that in place, the application will always start with the latest index definition, including the synonym mapping.

Verifying synonym behavior

Once the configuration is in place, synonym expansion is transparent to the application. A search for `sci-fi` returns science fiction books ranked by relevance, exactly as `Science Fiction` would. The following queries are worth running to confirm that the mapping is working correctly:

- `detective fiction` should match mystery and crime novels.
- `memoir` should return biographies.
- `Science Fiction` should continue to work as before. Synonym expansion does not affect queries that already match without it.

If a synonym is not working, the most likely cause is an analyzer mismatch. The synonym terms in the collection and the field being queried both need to be processed by compatible analyzers. The genres field uses `lucene.standard`, and the synonym mapping also specifies `lucene.standard`, so tokens generated from the synonym terms and tokens in the index both pass through the same pipeline. If you ever change a field's analyzer, check that any synonym mappings referencing that field are updated to match.

One more point worth noting: synonym mappings apply only to fields indexed as string type. The token mapping on genres is used for counting and grouping, not for text search, so synonyms have no effect on facet behavior.

Summary

This chapter has covered four areas where deliberate choices at the index, query, and configuration level make the difference between a search feature that works and one that is production-ready. Leafy Library served as the running example throughout, but the patterns apply to any MongoDB Search implementation.

First, the search index was hardened by committing fully to static field mapping and choosing the right analyzer for each field. Static mapping keeps the index lean and predictable; the right analyzer ensures that terms at index time and query time match, which is what makes searches return what users expect. The chapter also showed how to use `searchAnalyzer` to deliberately separate the index-time and search-time pipelines for fields whose stored content requires extra processing that user input does not.

Queries were then optimized by projecting only the fields the application uses, placing `$limit` early in the pipeline so MongoDB Search stops processing documents once enough results have been collected, and routing non-scoring genre filters through filter clauses rather than must.

Next, relevance scores were exposed and used to understand why results are ranked as they are. You can enable `searchScoreDetails` and the `scoreDetails: true` parameter to inspect the scoring detail for each result. Field-level boosting and compound query strategies were then applied to make the ranking better reflect intent: title matches first, author name matches second, and genre matches last.

Lastly, synonyms were added to bridge the vocabulary gap between user input and document content, so a search for `sci-fi` now finds science fiction books, `memoir` finds biographies, and `detective fiction` finds the mystery shelf.

The theme running through all of these improvements is the same: be intentional. Index only what you will query. Return only what you will use. Weight what matters more, more heavily. Map the terms your users use, not just the terms your data uses.

In the next chapter, we will turn our attention to a different kind of search altogether: one that understands meaning rather than matching words. We will explore MongoDB's vector search capabilities and how they enable semantic search, finding results that are conceptually relevant even when they share no vocabulary with the query.

Additional resources:

The following resources provide additional information to strengthen your existing knowledge of MongoDB Search:

- *MongoDB Search Overview*: `https://mdb.link/building-modern-data-applications-with-mongodb-search-docs`.
- *MongoDB Text Search: B-Tree vs Inverted Index, Use Search Instead (Part 1)*, by *Erik Hatcher*: `https://mdb.link/building-modern-data-applications-with-mongodb-search-series`.
- *MongoDB Search Score Breakdown*, by *Erik Hatcher*: `https://mdb.link/devrel.book.building-modern-data-applications-with-mongodb-searchscoredetails-blog`.

Search Fundamentals

Learn the fundamentals of MongoDB Search, including indexing, lexical search, and query optimization. Build efficient full-text search experiences in MongoDB applications.

`https://mdb.link/badge-search-press-donet`

Part 3:

Semantically Searching Your Data

In the third part of this book, you will move from understanding semantic search concepts to implementing and operating them in a real application.

You will begin by building a clear foundation in MongoDB Vector Search, including how semantic search differs from keyword search, how embeddings represent meaning, how vector indexes are configured, and how similarity-based queries are interpreted. You will then apply that foundation directly to Leafy Library by implementing semantic search end to end, from embedding configuration and generation to vector index creation and vector query integration. Finally, you will move into production decision making, learning how to tune and evaluate vector search using a practical framework based on quality, latency, and cost while making informed choices about models, indexing strategy, and scale.

By the end of this part, you will not only understand how vector search works but also know how to implement it confidently and run it effectively as your application and data grow.

This part of the book includes the following chapters:

- *Chapter 7, Search Your Data Semantically with MongoDB Vector Search*
- *Chapter 8, Add Semantic Search to the Application*
- *Chapter 9, Best Practices for MongoDB Vector Search*

7

Search Your Data Semantically with MongoDB Vector Search

Search is a fundamental capability in modern applications. Users expect to be able to type a question, describe what they need, or paste a fragment of text and quickly get back useful results. Increasingly, they also expect the system to understand their intent, not just the exact words they typed.

Consider the Leafy Library application, a user might search for `books about resilience after loss` even if none of the book descriptions use that exact phrase. Another reader might search for `stories about starting over` and expect novels about grief, recovery, hope, and rebuilding life to appear. These are not exact keyword lookups; they are meaning-based searches.

This is where semantic search with MongoDB Vector Search becomes valuable.

In this chapter, MongoDB Vector Search is introduced from the ground up, with a focus on the underlying concepts. *Chapter 8*, *Add Semantic Search to the Application*, covers how to implement MongoDB Vector Search, while *Chapter 9*, *Best Practices for MongoDB Vector Search*, examines the best practices for using it effectively. By the end of this chapter, you will have a solid understanding of what vector search is, how it works, and when it is the right tool to use.

Chapter 4, *Search Your Data with MongoDB Search*, introduced MongoDB Search as the tool for full-text and relevance-based lexical search. This chapter introduces the companion capability for semantic understanding. Together, these features allow you to support both literal matching and intent-based retrieval in intelligent data applications across multiple media types, not just text.

The following topics are covered in this chapter:

- What is semantic search?
- Generating embeddings for your data
- Creating a search index with MongoDB Vector Search
- Additional features: Filtering, index types, and search tuning
- How MongoDB Vector Search fits into intelligent data applications

> If you would like a complete overview of the Leafy Library application architecture and implementation, refer to *Chapter 11*, *Appendix*.

What is semantic search?

Semantic search is a way of finding information based on meaning rather than exact word overlap. In a traditional keyword search system, results are returned because they contain terms that match the query. This approach works well when the user knows the right words to use and the content uses those same words. However, it starts to break down when users search using terms that are not directly present in the search corpus.

Imagine searching Leafy Library for `hopeful books about rebuilding your life.` The most relevant books in the catalogue might be described as stories of grief, recovery, second chances, or personal transformation. A purely lexical search can miss or under-rank those results if the wording in the query and the wording in the stored descriptions do not overlap enough. A semantic search system has a much better chance of recognizing that these descriptions address the same underlying idea, even though they use different language.

This is the core promise of semantic search: it captures intent, topic, and conceptual similarity.

Why keyword search is not always enough

Keyword search is still extremely useful. In fact, many applications should retain it. There are plenty of cases where literal term matching is exactly what you want:

- Searching for a product SKU
- Looking up an exact book title
- Finding a known author name
- Matching a tag, code, or identifier

Modern, intelligent data applications, however, often need more than this. Many users interact with these applications using natural language, and sometimes through modalities that standard text search does not cover, such as image, audio, and video. These applications also need to support queries such as:

- `Books about recovering from grief`
- `Articles similar to this support ticket`
- `Help me find documents related to expense policy`
- `Show me products like this one`

These are not exact-match problems. They are meaning-matching problems.

Semantic search addresses these problems because it can recognize relationships between concepts, synonyms, paraphrases, and related phrasing. It is also far more natural for users because they can search in the language they would normally use rather than trying to guess which literal keywords are present in the database.

Semantic search compared with lexical search

It is helpful to think of lexical and semantic search as solving different retrieval problems. Lexical search asks:

- Which documents contain these words?
- How often do the words appear?
- How important are those words within the document and collection?

Semantic search asks:

- Which documents are closest in meaning to this query?
- Which documents express the same idea in different words?
- Which records are conceptually related, even if the wording differs?

Neither approach is universally better than the other; they are complementary.

In fact, some of the best search experiences combine both approaches. Lexical search is well suited to cases where the user knows exactly which term they want, while semantic search performs better when the user describes a need, a theme, or an idea. In Leafy Library, for example, a user looking for a known title such as `The Midnight Library` is conducting a very different kind of search than a user asking for `books about regret and second chances.` Both are valid, but they benefit from different retrieval strategies.

Examples of semantic search in real applications

Semantic search is useful across many kinds of intelligent data applications:

- In a customer support system, users can ask a question naturally and retrieve the most relevant help article.
- On a media or publishing platform, readers can discover related stories even when headlines use a different language.
- In e-commerce, users can search for products using descriptive phrases such as `comfortable shoes for standing all day`.
- In internal knowledge systems, teams can retrieve policies, notes, and documentation based on meaning instead of file names or exact phrases.
- In recommendation scenarios, an existing document, product, or message can serve as the search input to find similar items.

This flexibility is a major reason vector search has become such an important part of modern application design. One way to think about semantic search is this: instead of searching the surface form of language, you are searching for a mathematical representation of meaning. That representation is the subject of the next section.

Capturing semantic meaning: The role of vector embeddings

At the heart of semantic search is the idea that language can be represented numerically.

That may sound surprising at first. Human language is rich, ambiguous, and highly contextual. Yet, modern machine learning models are able to transform text into arrays of numbers that preserve useful information about its meaning. These numeric representations are called embeddings.

An analogy can help here. Imagine a vast map where every sentence, paragraph, or document is represented as a point. Two points that sit close together express similar meaning, while two points that are far apart express very different ideas.

If one book description is about losing a loved one and learning to live again, and another is about rebuilding life after a major change, those two points might sit near each other on the map. A gardening guide or a space opera would likely sit much farther away.

This is not a literal map that can be drawn in two dimensions, because the real space might contain hundreds or thousands of dimensions. The principle, however, is the same: near means similar, and far means different.

That is what makes vector search possible.

Without embeddings, a database cannot perform semantic search because it has no mathematical representation of meaning to compare. Once embeddings exist, a user query can also be converted into a vector, and the system can retrieve the stored vectors that are closest to it.

This allows the system to do things that keyword search struggles with:

- Match paraphrases and variations in how the same idea is expressed.
- Understand related language and recognize terms or phrases with similar meanings.
- Tolerate wording differences without requiring an exact match between the query and the content.
- Retrieve conceptually similar content, including images, audio, and video.
- Support query-by-example, where a document is used to find similar documents.

Embeddings are therefore not an optional enhancement, they are the foundation of semantic retrieval.

One important design consideration is what to embed. You need to consider which fields in your document capture the meaning of your data. This could be song lyrics or words transcribed from audio, information drawn from a title, a movie or book cover, or many other examples from your own applications. You might generate embeddings for:

- An entire product description.
- A support article section.
- A single paragraph from a large document.
- A movie synopsis.
- A message in a support conversation.

For example, in Leafy Library, you might choose to embed the book summary, or perhaps a combination of the summary, genre, and a curated description. This decision affects the kind of similarity the system learns to retrieve. Because MongoDB Vector Search supports more than just text, you could also choose to generate embeddings for book covers, allowing users to search by describing something they see in a cover image.

The choice affects search quality. If embeddings are too broad, you may lose detail. If they are too narrow, you might lose context. This is one reason why a strategy for focused embeddings becomes important in real implementations. This topic is revisited in *Chapter 9* when discussing best practices. For now, the main point is that embeddings are not only about converting text or other media types into numbers, but they are also about deciding what information you want the system to retrieve.

Generating embeddings for your data

Once you understand that embeddings are required for semantic search, the next question is obvious: where do those embeddings come from? They are created by embedding models.

An embedding model is a machine learning model trained to convert text into vectors that preserve semantic relationships. You provide it with text as input, and it returns a list of numbers. Those numbers are the embedding.

MongoDB Vector Search does not produce the vector representation itself. Instead, it works with vectors generated by embedding models, and then stores, indexes, and queries them efficiently.

What embedding models do

Modern embedding models are trained on huge amounts of text so that they can learn statistical and semantic relationships between words, phrases, and documents. They learn that certain ideas commonly appear together, that some phrases are interchangeable in meaning, and that context changes interpretation.

For example, a well-trained embedding model can learn that:

- `forgot my password` and `cannot log in` are related.
- `book recommendations` and `what should I read next` are close in intent.
- `apple` in a technology article differs from `apple` in a recipe article.

This contextual awareness is one reason why vector search has become so effective compared to older, more limited text-matching techniques.

MongoDB supports vectors generated by embedding models from many providers, including Voyage AI, OpenAI, Hugging Face, and Fireworks AI, among others. This breadth matters because model quality directly affects search quality. A stronger model generally produces embeddings with better semantic structure, which in turn improves retrieval quality.

When developers talk about vector search, they sometimes focus heavily on the index or the query stage. However, the quality of the embeddings is just as important. If the embeddings are poor, even the best vector search engine cannot produce strong results. Equally important is consistency because the model used to embed your corpus must be the same model, or the same family of models, used to embed queries at search time. Embeddings from different models occupy different vector spaces, so comparing them produces meaningless results.

Manual embedding generation

Traditionally, generating embeddings has been an explicit step in the application or data pipeline.

The workflow looks something like this:

1. Identify the content you want to search semantically.
2. Generate an embedding vector for that content using an embedding model.
3. Receive the embedding vector.
4. Store that vector within the original document in MongoDB.
5. Generate a vector for each user query using the same model.
6. Run a vector search query against the stored vectors.
7. Receive and process the response for return to the user.

This approach is flexible and powerful. It gives you full control over preprocessing, model choice, re-embedding strategy, and update workflows.

For example, if you were building semantic search for Leafy Library, you might decide to generate an embedding from the book description, or from a combination of the title, summary, and genre. Each book document would then store that vector in a dedicated field such as `embedding`, `plotEmbedding`, or `searchEmbedding`. An example of a book document with vector embeddings stored appears later in this chapter, illustrating how a vector can simply be stored as a field alongside the existing data.

When a user searches for `hopeful novels about starting over`, your application generates a query embedding using the same model, and MongoDB Vector Search returns the books whose stored vectors are closest.

Manual versus automatic workflows

Neither approach is universally correct.

Auto embeddings are a good option when:

- You want the fastest path to adoption.
- Your use case fits the supported workflow well.
- You prefer a more managed experience.
- You want to reduce custom pipeline code.

Manual embeddings are attractive when:

- You need full control over how text is prepared.
- You have domain-specific content or custom chunking needs.
- You need stricter versioning or lifecycle control.
- You want to experiment with different embedding models.

This is an important pattern across modern applications. Managed features accelerate delivery, while custom pipelines offer maximum control. The right choice depends on the requirements of your application and the maturity of your team.

Auto embeddings with Voyage AI

As of May 2026, MongoDB also offers auto embeddings with Voyage AI. This is an important development because it simplifies one of the most intimidating parts of the semantic search workflow. Instead of building and maintaining your own embedding generation pipeline, you can let MongoDB automatically generate embeddings for your data.

This changes the developer experience significantly. With a manual workflow, you need to think about:

- How documents are sent to the model.
- When embeddings are refreshed.
- How updates and inserts are handled.
- How vector fields are stored and kept consistent.
- How query embeddings are generated at runtime.

With auto embeddings, much of that operational burden is reduced. You still need to understand the conceptual pieces, but you can adopt vector search faster and with less custom infrastructure. This is especially helpful for teams new to semantic search or looking to focus on product outcomes rather than vector search infrastructure.

Why the same model matters

Embeddings are only meaningfully comparable when they are produced by the same embedding model or by models designed to work together. If stored documents are embedded using one model and a query is embedded using a different model, the resulting vectors are mapped to different semantic spaces. This mismatch can degrade the quality of results, leading to incorrect matches or no results being retrieved at all.

This is why semantic search workflows use one consistent embedding model for both indexed data and incoming queries.

As simple as this sounds, it is one of the most important operational rules in vector search.

Storing embeddings in MongoDB

Once generated, embeddings are stored within the document in MongoDB. This creates a clean architecture because the source data and its semantic representation live together. This arrangement has several benefits:

- Your application has a single source of truth.
- You can combine semantic retrieval with metadata filters.
- You can project both original fields and vector search results in a single query flow.
- You avoid managing a separate external vector database for many use cases.

This close integration is one of the strengths of MongoDB Vector Search. It allows semantic retrieval to become part of the same document-oriented workflow you are already using elsewhere in the application.

The following example shows a book document from the Leafy Library application that includes these embeddings:

```
{
  "_id": "0002005018",
  "title": "Clara Callan: A novel",
  "authors": [
    {
      "_id": {
        "$oid": "64cc2db4830ba29148da4c3b"
      },
      "name": "Richard Bruce Wright"
    }
  ],
  "genres": [
    "Women Teachers",
    "Young Women",
    "Actresses",
    "Sisters"
```

```
  ],
  "pages": 414,
  "year": 2001,
  "synopsis": "Giller Prize Winner 2001. Richard B. Wright. A Phyllis
Bruce Book.",
  "cover": "https://images.isbndb.com/covers/19141603482188.jpg",
  "totalInventory": 5,
  "available": 18,
  "binding": "Hardcover",
  "language": "en",
  "publisher": "HarperFlamingoCanada",
  "longTitle": "Clara Callan: A novel",
  "embedding": [
    0.043669462,
    0.036852073,
    -0.01699437,
    0.010420907,
    -0.025894804,
    0.031142129,
    0.018640816,
    -0.061661635,
…],
  ]
}
```


As you can see, the document contains a field named `embedding`, which is an array of numerical representations of the data chosen for vectorization. Critically, this field resides inside the same document as the source data from which the embeddings were generated. This is what makes MongoDB Vector Search so powerful: all the data is kept together in a single, searchable document.

Creating a search index with MongoDB Vector Search

Once embeddings exist in your documents, the next step is to make them searchable. That is the role of the vector search index.

Just as MongoDB uses indexes to make ordinary queries fast, MongoDB Vector Search uses a dedicated index to make nearest-neighbor retrieval efficient. Without that index, every query would require an expensive scan and comparison across the full set of stored vectors.

A traditional index is designed around exact matching, sorting, or range-based lookups, while a vector index is designed around similarity.

When you run a vector search query, the system is not looking for a literal value or a simple range. It compares one vector against many stored vectors to find the nearest matches. The index exists to make this process fast enough for real applications.

This is one reason vector search is conceptually different from standard create, read, update, and delete (CRUD) queries or even full-text search. The retrieval logic is geometric rather than textual.

What the index needs to know

A vector search index definition typically needs to capture several key facts about the data:

- Which field contains the embedding vector
- What the vector dimensionality is
- Which similarity metric should be used
- Which index algorithm should be used

Dimensionality is especially important. If your embedding model returns vectors of a certain size, the index must be configured to expect vectors of that same size. This illustrates how model choice and index configuration are connected; they are not independent concerns.

In MongoDB Vector Search, vector indexes are not incidental. They are part of your application design. Before creating one, you should be clear about the following:

- Which content is being represented semantically
- Which field stores the vector
- Whether the index serves search, recommendation, similarity lookup, or all three
- What scale and latency requirements the application must meet

This is similar to the search indexes discussion in *Chapter 4*. The index definition shapes what retrieval experience the application can provide.

Suppose the Leafy Library collection stores a summary embedding for each book. Without the index, those queries are not practical at scale. With the index, they become a core application capability. This is the pattern to keep in mind throughout this chapter: the embedding captures meaning, and the index makes that meaning searchable.

It helps to think of the vector search index as part of a larger workflow rather than in isolation. First, the source content is selected, and embeddings are generated for it. These embeddings are then stored in MongoDB, after which a vector search index is created on the embedding field. When a user submits a query, it is converted into an embedding using the same model, and a similarity search is performed to retrieve the most relevant matching documents.

This end-to-end view makes it easier to understand where issues can arise. Poor results may be caused by weak embeddings, the wrong text being embedded, poor data processing, an index configuration mismatch, or the query itself. The vector index is critical, but it is only one part of the full semantic retrieval chain.

Chapter 8 covers an actual vector search implementation. However, before reaching that point, what matters here is understanding what the index is doing conceptually. It is not merely storing vectors; it is enabling fast semantic retrieval over those vectors. That distinction is useful when you later make decisions about performance, cost, scale, and search quality.

Performing a vector search query

Once your data has embeddings and the vector index is in place, you can perform a vector search query. Conceptually, this is straightforward: take the query, turn it into an embedding, compare it to the indexed vectors, and return the closest matches. Under the hood, however, this represents a meaningful shift from how most developers first think about querying data.

In a vector search workflow, the query text is first passed through the same embedding model used to create the stored vectors. The result is a query embedding.

MongoDB Vector Search then compares that query vector to the vectors stored in the collection and returns the nearest ones. These results may be highly relevant even though none of them contain the exact query wording.

Instead of asking whether strings are equal, the system asks how similar two vectors are. Different formulas can be used for this comparison, such as cosine similarity, dot product, or Euclidean distance. These are covered later in the chapter, but the key idea is that the database identifies the stored vectors that are closest to the query vector and returns the nearest matches.

This is why vector embeddings are central to MongoDB Vector Search. They turn meaning into something that can be indexed, compared, and retrieved efficiently.

A similarity measure is used to determine closeness between vectors. You don't need to be a mathematician to use these effectively, but you should understand their role. These formulas define what "near" means in vector space.

For text embeddings, cosine similarity is often an intuitive choice because it emphasizes directional similarity rather than raw magnitude. In practical terms, it helps you focus on semantic relationships. The important point for this chapter is not the exact formula. Instead, note that vector search requires a mathematical definition of similarity, and the chosen metric influences how results are ranked.

A vector search query typically returns the top N nearest matches, often along with a similarity score. That score can be useful in several ways:

- To rank results.
- To suppress weak matches below a threshold.
- To blend vector results with other retrieval signals.
- To inspect how semantically close the returned documents are.

This is especially valuable in applications where search quality is crucial. For example, a support assistant may want to avoid returning weak semantic matches that risk misleading the user. A discovery experience, by contrast, might comfortably show a broader set of loosely related content.

Interpreting results carefully

A vector search result does not state `this is the same document`. Instead, it indicates `this document is close in semantic space`. Some applications require very tight similarity, while others aim for broad thematic relatedness. The meaning of a good result depends on the use case, the embedding model, the content being embedded, and the threshold or ranking strategy you choose.

This is why implementation and best practices are addressed in later chapters. Understanding the query flow conceptually is the first step; tuning it for real-world quality is the next.

Additional features: Filtering, index types, and search tuning

The core vector search flow is powerful on its own, but real applications usually need more than simple nearest-neighbor retrieval.

MongoDB Vector Search includes additional capabilities that make it more practical for production use. These include combining vector similarity with traditional filters, choosing between index types, and tuning search behavior for scale and performance.

Filtering with semantic search

One of the most useful capabilities for powerful vector search is combining vector retrieval with structured filtering. Imagine a user who wants to find books semantically related to `coping with change` but only in English and only currently available in their local branch. Or consider a research platform that wants articles about renewable energy policy but only from the last three years and only within a specific subject area.

In these cases, semantic relevance alone is not enough. The application needs both semantic retrieval and structured constraints.

This is where MongoDB's integrated document model becomes particularly valuable. A vector search workflow can be combined with filters over ordinary document fields, narrowing down results to what is actually usable in the current application context.

This combination often produces much better user experiences than semantic search alone.

In Leafy Library, this could mean retrieving books that are semantically close to `comforting fiction after bereavement` while still limiting results to English-language titles that are currently available.

This is not merely a technical implementation detail. Index choice influences:

- Query latency
- Infrastructure cost
- Scalability under load
- Result quality characteristics

That is why it's important to understand index selection early. Before implementing vector search, you should already understand that retrieval quality is not determined solely by the query. It also depends on the index strategy and the surrounding architecture.

Flat versus HNSW indexes

MongoDB Vector Search supports different index approaches, including flat and **Hierarchical Navigable Small World (HNSW)**. At a high level, the choice between them is a trade-off between exactness and scalability.

Flat indexes

Flat indexes were introduced to solve a specific multitenancy problem in vector search. Traditional HNSW indexes are well suited to large, shared vector spaces where approximate nearest-neighbor search is the right trade-off. However, many AI applications do not query a global vector pool. They query one tenant at a time, often using a strict filter such as `tenantId`, and each tenant may have only a few thousand vectors. With that shape of workload, HNSW's global graph can become more of a burden than a benefit: filtered accuracy can drop, graph traversal overhead adds cost even when the result set is small, large tenants can create noisy-neighbor effects, and latency can vary significantly between tenants.

Flat indexes address this through a different strategy: exhaustive search over the already-filtered tenant subset. Instead of navigating a global graph, MongoDB scans the exact candidate set for that tenant and returns precise nearest neighbors from that bounded partition. In practice, this approach aligns better with highly selective multitenant queries, where recall quality, stable latency, and predictable behavior matter more than **approximate nearest neighbor (ANN)** shortcuts over massive, shared data. The result is a more consistent experience across tenants, lower memory and maintenance overhead compared to graph-heavy indexing, and reduced write-time complexity associated with global graph updates.

The practical model is straightforward: if your application has many tenants, each tenant has a relatively small vector footprint, and queries are always scoped to one tenant, a flat index is the better default. If you are searching across large, shared corpora or very large tenant partitions, HNSW remains the right choice. Adopting a flat index is operationally simple because it is an index configuration change, not a query rewrite. You continue using the same vector search query interface, but choose an indexing method that matches multitenant reality rather than forcing a one-size-fits-all ANN path.

HNSW indexes

HNSW is an approximate nearest-neighbor technique. It is designed to retrieve highly similar matches much faster at scale by navigating a structured graph of vector relationships.

HNSW is suitable when the dataset is large and latency matters, or when you want strong retrieval quality with far better scalability.

Approximate does not mean poor. In practice, approximate nearest-neighbor methods are often the standard way to make vector search viable in real applications with large collections.

The key is understanding the trade-off between recall, speed, and resource usage.

Hybrid retrieval patterns

Another important concept is hybrid retrieval.

Hybrid retrieval combines multiple signals when finding results. For example, a hybrid approach might combine the following:

- Lexical search for exact term relevance
- Vector search for semantic similarity
- Business rules or popularity signals for ranking

This is often a very strong pattern in production systems. A user searching for `beginner gardening books` may benefit from results that match the exact phrase, are semantically related, and are also highly rated or currently in stock.

Even if you do not implement hybrid retrieval immediately, it is useful to recognize that vector search is often one piece of a broader retrieval strategy.

Rerankers

Related to this is the idea of reranking. In some retrieval flows, the first step is to retrieve a strong set of candidate results using vector search, lexical search, or a hybrid of both. A second step can then rerank those results to improve the final ordering. This can be useful when you want to push the most relevant results closer to the top without changing the broader retrieval strategy. We don't need to discuss reranking in detail in this chapter; however, it is worth knowing that it is now part of the wider Vector Search story and is especially relevant once you start refining search quality in production.

Rerankers are examined in more detail in *Chapter 9*, where they are discussed in the context of improving search performance.

Quantization

Vectors are expensive to store. A single embedding at 1536 dimensions using 32-bit floats takes 6 KB. Multiplied across a million documents, that amounts to approximately 6 GB for the embeddings alone. Quantization addresses that cost directly.

Quantization is a compression technique that reduces the size of stored vectors by representing them with lower-precision data types. The goal is to trade a small amount of accuracy for significant gains in memory efficiency and query speed.

The default value is `none`, which means vectors are stored at full precision as 32-bit floats. This provides maximum precision but also incurs the highest storage costs.

Scalar quantization compresses each 32-bit float component down to an 8-bit integer, delivering a 4x reduction in storage size with minimal recall impact in most cases.

And finally, binary quantization reduces each dimension to a single bit, simplifying each value to a binary question: is this value positive or negative (1 or 0)? This is the most aggressive option, delivering a 32x reduction in storage size, but with a meaningful accuracy trade-off. *Chapter 9* covers binary quantization in more detail, including how MongoDB handles that accuracy trade-off.

Tuning and thresholding

One final concept worth introducing here is thresholding. Not every semantic match is strong enough to display to a user. Depending on your application, you may want to discard weak matches, rerank the result set, or combine semantic scores with other decision logic. For example:

- A customer support bot may need conservative thresholds to avoid surfacing poor answers.
- A recommendation shelf may tolerate looser similarity because discovery is the goal.
- An internal knowledge assistant may blend semantic closeness with recency or author trust.

These are best-practice concerns examined in *Chapter 9*, but they are worth noting here because they highlight an important principle: vector search is not just about retrieving similar content. It is about designing the right retrieval behavior for the product.

How MongoDB Vector Search fits into intelligent data applications

Before closing this chapter, it's worth stepping back to consider the bigger picture. MongoDB Vector Search is not just a feature for building search interfaces. It is a building block for a broader class of intelligent data applications. Whenever an application needs to retrieve information based on meaning, intent, or relatedness, vector search becomes relevant. This includes the following use cases:

- Semantic search experiences
- **Retrieval-augmented generation** (**RAG**) workflows
- Recommendation systems
- Similar-item discovery
- Knowledge base assistants
- Support automation
- Document classification and grouping workflows

In many of these systems, vector search acts as the retrieval layer that connects user intent with stored knowledge.

This is especially important in AI-assisted applications. Large language models are powerful, but they are not databases. They require a retrieval mechanism when current, proprietary, or domain-specific data must be brought into the conversation. Vector search is one of the key tools that makes this possible.

If Leafy Library later evolves into a conversational assistant that helps readers discover books, answers catalog questions, or explains why a title was recommended, vector search becomes an important part of that architecture. It gives the assistant a way to retrieve the right books and descriptions based on meaning before any generated response is produced.

MongoDB's advantage here is that the semantic retrieval layer sits close to the application data itself. This means you can work with operational data, metadata, structured filters, and semantic search together, rather than stitching multiple disconnected systems into a single experience.

This does not eliminate architectural decisions. You still need to think carefully about what to embed, how often to refresh embeddings, how to evaluate relevance, and how to tune performance. However, MongoDB does provide you with a strong, integrated platform for building these capabilities.

That is why this topic matters in the context of intelligent data applications. It is not only about search; it is about retrieval as a core capability.

Summary

In this chapter, you explored MongoDB Vector Search and the core concepts that power semantic search. You learned how semantic search differs from traditional lexical search and why understanding user intent is essential for building modern search experiences. The chapter introduced vector embeddings as numerical representations of meaning, explained how embedding models, such as Voyage AI, generate those vectors, and highlighted why using the same embedding model consistently is critical for maintaining retrieval quality.

The chapter also examined the overall semantic search workflow: generating embeddings, storing them within your documents, creating a vector search index, embedding incoming queries, and retrieving the nearest matches. Along the way, important supporting concepts were introduced, such as filtering, flat versus HNSW index types, hybrid retrieval, and the growing importance of managed workflows such as auto embeddings.

At this point, you should understand not only what MongoDB Vector Search is, but also why it matters and how it fits into the architecture of an intelligent data application.

In *Chapter 8*, the focus moves from concept to implementation, with a working vector search flow built from the ground up. *Chapter 9* then covers best practices, including chunking strategy, model selection, result quality, and performance tuning.

Additional resources

The following resources provide additional information to help you learn more about MongoDB Vector Search:

- *MongoDB Vector Search Overview*: `https://mdb.link/building-modern-data-applications-with-mongodb-vector-search-docs`.
- *MongoDB AI Learning Hub*: `https://mdb.link/building-modern-data-applications-with-mongodb-ai-learning-hub`.

Vector Search Fundamentals

Learn how to use MongoDB Atlas Vector Search to build AI-powered search experiences using indexing, embeddings, and retrieval strategies.

`https://mdb.link/badge-vector-search-press-donet`

Vector Search Performance

Learn how to diagnose, optimize, monitor, and tune MongoDB Vector Search performance in production.

https://mdb.link/badge-vector-search-perf-donet

Voyage AI with MongoDB

Learn how to create, store, and query vector embeddings in MongoDB, use auto-embedding and Voyage AI for vector search and reranking, and optimize search quality, latency, and cost.

https://mdb.link/building-modern-data-applications-with-mongodb-voyageai-badge

8

Add Semantic Search to the Application

In *Chapter 7*, *Search Your Data Semantically with MongoDB Vector Search*, you explored the concepts behind semantic search, including embeddings, vector similarity, and how MongoDB Vector Search retrieves results based on meaning rather than exact wording. In this chapter, you'll put those concepts into practice by implementing them in the Leafy Library application.

The Catalog UI already includes a search bar, but it is not yet wired to any backend search logic. This gives you a clean implementation point: the existing UI input can be connected directly to semantic retrieval from day one. If a user searches for `books about starting over after loss`, the app should understand the intent behind that phrase and return semantically relevant titles, even when the wording in the book metadata is different.

In this chapter, you will implement that experience end to end. More importantly, each implementation step is tied to a clear purpose in the final workflow. You will configure embeddings so the application can call an embedding provider consistently, update the data model so vectors can be stored safely, create a MongoDB Vector Search index so those vectors are queryable, and then connect the query pipeline to the `$vectorSearch` stage so users can search by meaning.

As you move through each phase, the focus will be on both how and why. Before each major code change, the chapter provides context for the problem the change solves. After each major code listing, the key parts of the implementation are explained, along with how they contribute to the final search experience. This chapter also covers startup flow and data-shaping decisions that make the feature practical in a real application, including generating embeddings only when a book does not already have one.

By the end of this chapter, your Leafy Library application will have its existing search bar fully connected to semantic retrieval backed by MongoDB Vector Search, and you will understand the reasoning behind each part of that implementation.

The following topics are covered in this chapter:

- Configuring the application for embeddings
- Implementing the embedding service

Technical requirements

To follow this chapter, you will need:

- The Leafy Library repository cloned locally.
- .NET 10 or later installed.
- A MongoDB cluster with a `leafy_library` database.
- Access to an embedding provider (such as Voyage AI in Atlas, OpenAI, or Azure OpenAI) with valid API credentials.

Let's get started.

> If you would like a complete overview of the Leafy Library application architecture and implementation, refer to *Chapter 11*, *Appendix*.

Configuring the application for embeddings

In this section, you will make two foundational changes before generating or querying embeddings. First, you will configure embedding settings so the application can call the provider correctly at runtime. Next, you will update the `Book` model so each document can store its embedding vector in a field that the `$vectorSearch` stage can query.

Set up embedding configuration and models

To get started, you will configure the application and set up the model required for embedding generation. Before implementing the feature, make sure you are on the `start` branch of the Leafy Library repository, because this chapter is designed to move that baseline to the `with-vector-search` state.

> If you have completed *Chapter 5*, *Add MongoDB Search to the Application*, make sure the search bar in your application is not connected to the keyword search implementation to avoid confusion later on.

Next, you need to create a model class named `EmbeddingSettings` to represent the embedding settings at runtime. This follows the standard C# practice of dependency injection.

In the `Models` folder, create `EmbeddingSettings.cs` and replace the existing code with the following:

```
namespace Leafy_Library.Models;

public class EmbeddingSettings
{
    public string Provider { get; set; } = string.Empty;
    public string Endpoint { get; set; } = string.Empty;
    public string ApiKey { get; set; } = string.Empty;
    public string Model { get; set; } = string.Empty;
    public int Dimensions { get; set; }
}
```

A few details are worth noting here:

- `Endpoint` is required because the embedding request will be sent to a configurable API URL.
- `Dimensions` are not hardcoded in the model, so it can be aligned directly with the selected embedding model and index definition.
- `Provider` remains available for future extension, even though the implementation flow in this chapter uses a single endpoint request pattern.

Now add the corresponding configuration to your `appsettings.json` and `appsettings.Development.json`:

```
{
  "Embedding": {
    "Provider": "voyageai",
    "Endpoint": "https://ai.mongodb.com/v1/embeddings",
    "ApiKey": "<your-api-key>",
    "Model": "voyage-3",
    "Dimensions": 1024
  }
}
```

Replace the placeholder values with your actual embedding provider credentials and settings. The `Dimensions` value must match the output dimension of your selected embedding model. For this chapter, you will use Voyage AI and can create an API key directly within the MongoDB Atlas UI.

As with any cloud service, these steps can change over time. However, at the time of writing, you can generate a Voyage AI API key and add it to the `LeafyLibrary` app settings by following these steps:

1. Log in to MongoDB Atlas at `https://mdb.link/building-modern-data-applications-with-mongodb-atlas-login`.
2. Go to the **AI Models** page in the Atlas UI.
3. If it is not visible, select your desired organization from the **Organizations** menu in the navigation bar.

4. At the project level, click **AI Models** under the **Services** header in the navigation bar.

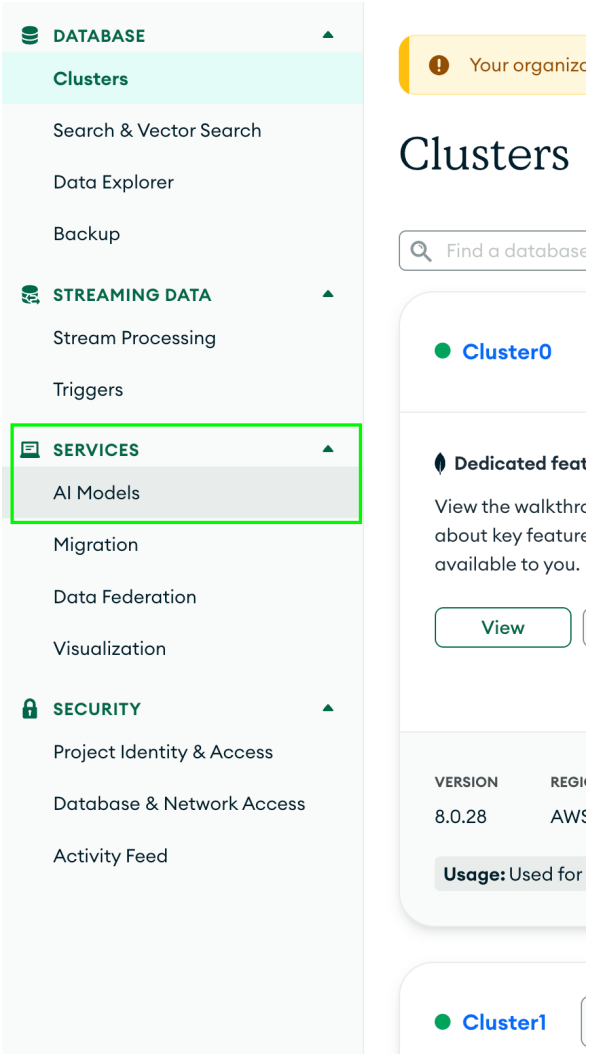


Figure 8.1 - The AI Models section under Services in the Atlas UI

5. Click **Create model API Key**.

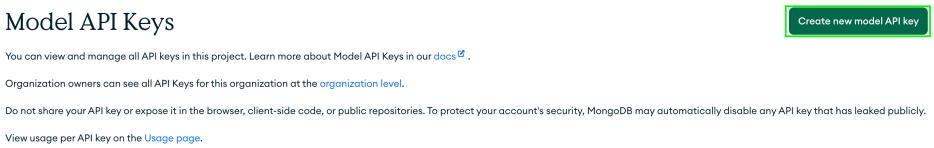


Figure 8.2 - Create model API Key button in the Atlas UI

6. Enter a name for your key.
7. Click **Create**.

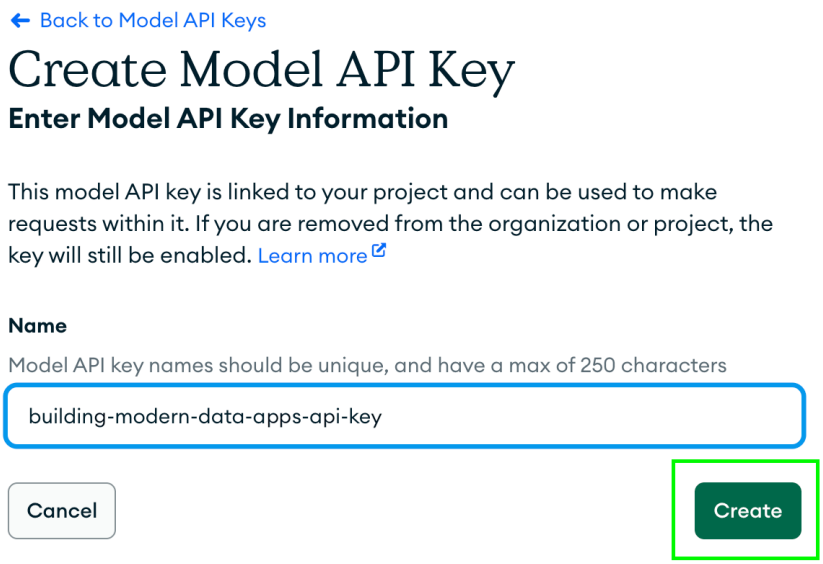


Figure 8.3 - Create button in the Atlas UI in the flow of creating a new model API key

8. Save the API key somewhere safe and copy it to the clipboard for use in the application.
9. Add it to the **ApiKey** entry in your **appsettings** files.

> If you need to view your API key again, see the *Model API Keys* documentation under the *View an API key* section: `https://mdb.link/building-modern-data-applications-with-mongodb-view-voyage-api-key`)

Update the Book model to store vectors

Next, make sure that book documents map correctly to the vector field that MongoDB Vector Search will query. You want the properties in your model class to map to the fields in the documents that you care about in your application, so ensure that the new vector field is present.

In `Models/Book.cs`, add the following new property to represent the embedding field:

```
[BsonElement("embeddings")]
[BsonIgnoreIfNull]
public double[]? Embedding { get; set; }
```

The vector index and query stage will target the embeddings path, so the field must be available in the `Book` document. Add the `BsonElement` attribute so that the driver can map between them automatically.

Implementing the embedding service

With configuration and models in place, the next step is to build the service that generates embeddings from text. As discussed in *Chapter 7*, MongoDB Vector Search supports multiple forms of content, including images, video, and audio. However, Leafy Library uses only text-based queries and generates embeddings only on text fields. This keeps the implementation focused, easier to validate, and fully aligned with how users search in the current application experience.

In the next few steps, you will build an `EmbeddingService` that reads provider settings from configuration, sends requests to the embedding endpoint, and parses the returned embedding vectors so they can be used in the vector search workflow.

Build the embedding service

Now it is time to generate vectors from text. In the `Services` folder, create a file named `EmbeddingService.cs`. You will build it in three logical parts, so each responsibility is easier to follow. Start with the class dependencies and constructor:

```
using System.Net.Http.Headers;
using System.Text;
using System.Text.Json;
using Leafy_Library.Models;
using Microsoft.Extensions.Options;

namespace Leafy_Library.Services;

public class EmbeddingService
{
    private readonly HttpClient _httpClient;
```

```
    private readonly EmbeddingSettings _settings;

    public EmbeddingService(HttpClient httpClient,
IOptions<EmbeddingSettings> settings)
    {
        _settings = settings.Value;
        _httpClient = httpClient;
        _httpClient.DefaultRequestHeaders.Authorization =
            new AuthenticationHeaderValue("Bearer", _settings.ApiKey);
    }
```

This part connects the service to dependency injection and ensures every request includes the API key. Setting the authorization header once in the constructor keeps request code cleaner and reduces repetition.

Next, add the start of `GetEmbeddingAsync` to validate input, handle provider-specific payload differences, and send the HTTP request:

```
public async Task<double[]> GetEmbeddingAsync(string text)
    {
        ArgumentException.ThrowIfNullOrEmpty(text);

        // OpenAI/Azure OpenAI use "dimensions" while some providers (e.g.
Voyage) use "output_dimension".
        var provider = _settings.Provider?.Trim().ToLowerInvariant() ??
string.Empty;
        object requestBody = provider switch
        {
            "voyageai" or "voyage" => new
            {
                model = _settings.Model,
                input = new[] { text },
                output_dimension = _settings.Dimensions
            },
            _ => new
            {
                model = _settings.Model,
                input = new[] { text },
                dimensions = _settings.Dimensions
```

```
            }
        };

        var content = new StringContent(
            JsonSerializer.Serialize(requestBody),
            Encoding.UTF8,
            "application/json");

        var response = await _httpClient.PostAsync(_settings.Endpoint,
content);
        response.EnsureSuccessStatusCode();
```

This part is where provider flexibility is implemented. The provider value decides whether the payload uses `output_dimension` or `dimensions`, so one service can support different providers without changing calling code elsewhere in the application.

Finally, add the response parsing logic to extract the embedding array and return it as double values:

```
        var json = await response.Content.ReadAsStringAsync();
        using var doc = JsonDocument.Parse(json);

        var embeddingArray = doc.RootElement
            .GetProperty("data")[0]
            .GetProperty("embedding");

        var embedding = new double[embeddingArray.GetArrayLength()];
        var i = 0;
        foreach (var value in embeddingArray.EnumerateArray())
        {
            embedding[i++] = value.GetDouble();
        }

        return embedding;
    }
```

This final part translates the provider response into the numeric array your vector search flow needs. At this point, the service provides a reusable way to turn user query text into embeddings that can be passed directly into the `$vectorSearch` stage later in the chapter.

Create the vector index and generate embeddings

Now it's time to create the MongoDB Vector Search index and populate embeddings for the book collection. Although you can now generate vectors, MongoDB still requires a Vector Search index that can query vector similarity efficiently.

In `Services/DatabaseService.cs`, you will add two responsibilities that work together during startup: ensuring the vector search index exists and is queryable, and generating and persisting any missing embeddings for books.

This provides a safe initialization flow where the application can repeatedly start without recreating existing resources.

1. Add a new local private variable to hold the `Dimensions` value from the configuration:

```
private readonly int _embeddingDimensions;
```

2. Update the constructor to match the following:

```
public DatabaseService(
    IOptions<MongoDbSettings> settings,
    IOptions<EmbeddingSettings> embeddingSettings)
{
    // existing setup omitted
    _embeddingDimensions = embeddingSettings.Value.Dimensions;
}
```

   This `dimensions` value is shared by both index creation and embedding generation. Keeping it in one place helps us avoid mismatches between stored embedding length and index definition.

3. Now, add a new method for creating the vector search index:

```
public async Task EnsureVectorSearchIndexAsync()
{
    const string indexName = "vectorsearch";

    using var cursor = await Books.SearchIndexes.
ListAsync(indexName);
    var indexes = await cursor.ToListAsync();
```

```
    if (indexes.Any(i => i["name"] == indexName))
    {
        return;
    }

    var definition = new BsonDocument
    {
        { "fields", new BsonArray
            {
                new BsonDocument
                {
                    { "type", "vector" },
                    { "path", "embeddings" },
                    { "numDimensions", _embeddingDimensions },
                    { "similarity", "cosine" }
                }
            }
        }
    };

    var model = new CreateSearchIndexModel(indexName,
SearchIndexType.VectorSearch, definition);
    await Books.SearchIndexes.CreateOneAsync(model);

    Console.WriteLine($"Waiting for '{indexName}' index to be
ready...");

    while (true)
    {
        using var statusCursor = await Books.SearchIndexes.
ListAsync(indexName);
        var statusList = await statusCursor.ToListAsync();
        var index = statusList.FirstOrDefault(i => i["name"] ==
indexName);

        if (index is not null && index["queryable"].AsBoolean)
        {
```

```
            break;
        }

        await Task.Delay(1000);
    }
}
```

This method makes index initialization idempotent. It first checks whether the `vectorsearch` index already exists. If it does, the method returns immediately. If it doesn't exist, the method creates it and waits until queryable is true, so later search requests don't run against an index that is still initializing.

4. Then add embedding generation:

```
public async Task GenerateEmbeddingsAsync(EmbeddingService
embeddingService)
{
    var filter = Builders<Book>.Filter.Exists(b => b.Embedding,
false)
        | Builders<Book>.Filter.Eq(b => b.Embedding, null);

    var booksWithoutEmbeddings = await Books
        .Find(filter)
        .ToListAsync();

    if (booksWithoutEmbeddings.Count == 0)
        return;

    Console.WriteLine($"Generating embeddings for
{booksWithoutEmbeddings.Count} books...");

    foreach (var book in booksWithoutEmbeddings)
    {
        var text = embeddingService.BuildEmbeddingText(book);
        var embedding = await embeddingService.
GetEmbeddingAsync(text);

        var update = Builders<Book>.Update.Set(b => b.Embedding,
embedding);
```

```
            await Books.UpdateOneAsync(b => b.Id == book.Id, update);
        }

        Console.WriteLine("Embedding generation complete.");
    }
```

This method applies the same startup-safe pattern to data. The filter targets only documents where `Embedding` is missing or null, preserving existing vectors. Each missing document is converted into embedding text, sent to the embedding service, and updated in place.

Together, `EnsureVectorSearchIndexAsync` and `GenerateEmbeddingsAsync` provide a repeatable startup process. The first method prevents duplicate index creation, and the second method prevents unnecessary re-generation of existing embeddings while still filling gaps for new or previously unprocessed books.

Wire everything up in Program.cs

Now, you need to connect these pieces to application startup to ensure the environment is ready before users begin searching. This ensures the embedding service is available through dependency injection and that index and data initialization run automatically at application start.

In `Program.cs`, add the following either before or after the existing service registration code:

```
builder.Services.Configure<EmbeddingSettings>(
    builder.Configuration.GetSection("Embedding"));

builder.Services.AddHttpClient<EmbeddingService>();
```

These registrations are important for two reasons. `Configure<EmbeddingSettings>` binds the `Embedding` section into a strongly typed settings object, while `AddHttpClient<EmbeddingService>` provides a managed `HttpClient` instance for outbound embedding requests.

Then update the startup initialization:

```
var dbService = app.Services.GetRequiredService<DatabaseService>();

var embeddingService = app.Services.
GetRequiredService<EmbeddingService>();
await dbService.EnsureVectorSearchIndexAsync();
await dbService.GenerateEmbeddingsAsync(embeddingService);
```

The order of operations matters here. `EnsureVectorSearchIndexAsync` runs first so the index exists before semantic retrieval is used. Then, `GenerateEmbeddingsAsync` fills only missing vectors in existing documents. This keeps startup deterministic and avoids querying an unready index.

Integrate vector search into BookService

With the index ready and embeddings generated, the next step is to wire the application search functionality to use vector similarity. This is where search-bar text is converted into embeddings and used for semantic retrieval.

Update `Services/BookService.cs` to inject `EmbeddingService`:

```
private readonly IMongoCollection<Book> _books;
private readonly EmbeddingService _embeddingService;

public BookService(DatabaseService db, EmbeddingService embeddingService)
{
    _books = db.Books;
    _embeddingService = embeddingService;
}
```

Injecting `EmbeddingService` here keeps the embedding generation logic inside the service layer, so the UI and calling code stay focused on user flow rather than provider details.

Then replace the existing `SearchAsync` method with a vector search pipeline:

```
public async Task<List<Book>> SearchAsync(string query, int page = 1, int
pageSize = 20)
{
    var queryEmbedding = await _embeddingService.GetEmbeddingAsync(query);

    var vectorSearchLimit = (page * pageSize) + pageSize;
    var options = new VectorSearchOptions<Book>
    {
        IndexName = "vectorsearch",

        NumberOfCandidates = 100

    };
```

```
    return await _books.Aggregate()

        .VectorSearch(

            book => book.Embedding,

            new QueryVector(queryEmbedding.Select(v => (double)
v).ToArray()),

            limit: vectorSearchLimit,

            options)

        .Skip((page - 1) * pageSize)

        .Limit(pageSize)

        .ToListAsync();

}
```

This method converts the user query into an embedding and sends it through the vector pipeline with pagination controls. `NumberOfCandidates` controls how many potential matches are evaluated before the final page is returned, which helps balance relevance quality and query cost.

And update `SearchCountAsync` similarly:

```
public async Task<long> SearchCountAsync(string query)
{
    var queryEmbedding = await _embeddingService.GetEmbeddingAsync(query);
    var options = new VectorSearchOptions<Book>
    {
        IndexName = "vectorsearch",
        NumberOfCandidates = 200
    };
    var results = await _books.Aggregate()
        .VectorSearch(
            book => book.Embedding,
```

```
            new QueryVector(queryEmbedding.Select(v => (double)
v).ToArray()),
            limit: 200,
            options)
        .ToListAsync();
    return results.Count;
}
```

`SearchCountAsync` follows the same embedding-first flow so counting and retrieval use the same semantic basis. Together, these methods move the existing search experience to the `$vectorSearch` stage rather than lexical matching.

Validate the feature end to end

Let us now run the application and confirm that semantic search is working.

If the library data is not already available because this is the first time you are running Leafy Library, it can be found in the `Data` folder of the repository as JSON files for each collection. You can use the `mongorestore` command-line tool or MongoDB Compass to import the data.

When the application starts, watch the console output. You should see the index and embedding setup complete without errors. On the first run with existing books, you should also see a message indicating that embeddings are being generated.

On the home page, try queries that are intent-based rather than exact title matches, such as:

- `books about magic`
- `stories about dragons`
- `hopeful fiction about rebuilding life`

Relevant books should now appear even when those exact phrases are not present in the title. This is the key signal that semantic retrieval is active.

If results appear empty or unstable, check the following first:

- The `Embedding:Dimensions` value matches the model output dimensions.
- The vector index `numDimensions` value matches the stored vector length.
- The `Embedding:Endpoint` and `Embedding:ApiKey` values are valid.
- The `embeddings` field exists on documents after startup generation.

With that, you have a working search bar that allows you to search your documents semantically.

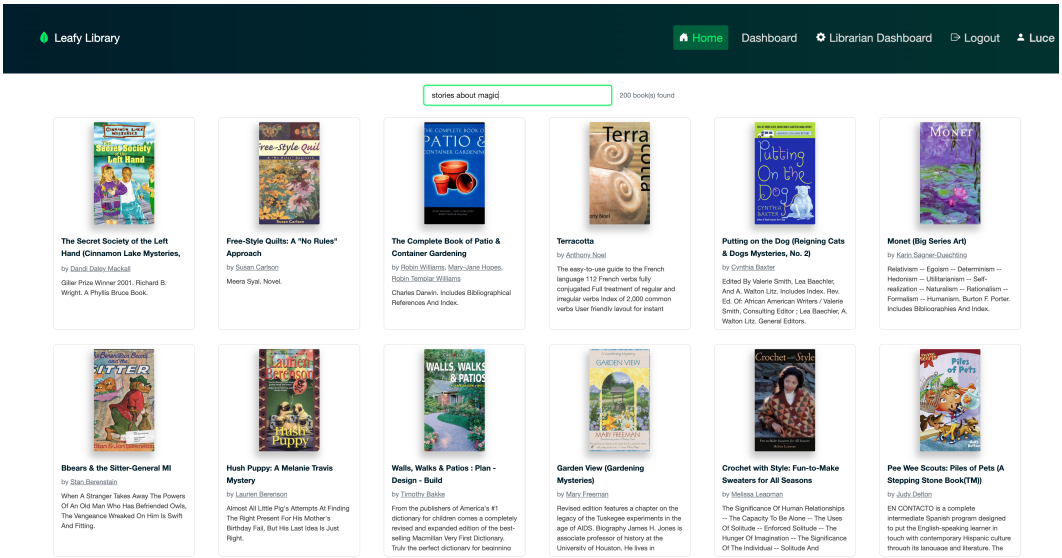


Figure 8.4: Results from semantically searching for stories about magic

Summary

In this chapter, you connected the Leafy Library application's existing search bar to semantic search powered by MongoDB Vector Search. You configured embedding settings, implemented an embedding service, created a vector index, generated embeddings for existing data, and switched the search pipeline to `$vectorSearch`. You also wired startup initialization so that index readiness and vector generation happen automatically in a safe, repeatable sequence.

Most importantly, you implemented the concepts from *Chapter 7* into practical code that users can experience immediately through improved search relevance.

In the next chapter, you will explore the best practices for production vector search workloads, including retrieval strategy, candidate sizing, filtering, and quality tuning. Leafy Library will serve as a conceptual framework for discussing these patterns.

9

Best Practices for MongoDB Vector Search

In *Chapter 8*, *Add Semantic Search to the Application*, you implemented semantic search in the Leafy Library application. In this chapter, the focus shifts from feature-building to the decisions behind it.

A vector search system can look excellent in testing, but it may become inconsistent in production when data volume grows, traffic patterns change, and costs start to matter as much as relevance. Getting results back is not enough. The right results must be returned quickly and at a sustainable cost.

In this chapter, you will learn practical, field-tested guidance for MongoDB Vector Search. You will examine how to reason about quality, latency, and cost, how embedding model decisions shape everything downstream, how index size affects both performance and budget, and when dedicated Search Nodes become the right architectural choice.

You can think of this chapter as an operating playbook for semantic search in production. The goal is to help you reason clearly about trade-offs and architecture choices before and during implementation.

The following topics are covered in this chapter:

- Vector search performance considerations
- Selecting an embedding model
- Managing index size
- When and why to use Nodes

By the end of this chapter, you will be able to tune MongoDB Vector Search intentionally instead of relying on defaults, and you will have a decision framework that can be applied as your data and usage grow.

Technical requirements

To follow this chapter effectively, you should have:

- Familiarity with basic vector search concepts and terminology.
- Awareness of embedding workflows, either manual embedding generation or Automated Embedding in Atlas.
- Basic familiarity with the MongoDB aggregation framework and `$vectorSearch`.

You don't need to complete *Chapter 8* to benefit from this chapter. Leafy Library is only used as an example to explain decisions and trade-offs.

> If you would like a complete overview of the Leafy Library application architecture and implementation, refer to *Chapter 11*, *Appendix*.

Vector search performance considerations

When teams first adopt semantic search, they often optimize one dimension in isolation, usually relevance or speed, and then are caught off guard by cost or instability later. A more reliable approach is to evaluate every major choice against three dimensions at once: quality, latency, and cost.

For Leafy Library, this trade-off is concrete. Aggressively increasing recall may give users excellent recommendations but at the cost of slower responses. Optimizing only for speed may produce results that feel arbitrary. Optimizing only for cost risks shipping an experience that technically works but does not earn user trust.

The key is learning to tune the system as a set of trade-offs rather than as a single objective function.

Understand ANN and ENN in practical terms

MongoDB Vector Search supports both **approximate nearest neighbor (ANN)** and **exact nearest neighbor (ENN)** search modes. ANN is designed for production-scale latency, whereas ENN is computationally intensive but invaluable for validation and querying smaller datasets. In

practical tuning workflows, ANN typically serves live user queries, while ENN is used to validate how closely ANN results match an exact nearest-neighbor result set during testing and tuning.

MongoDB recommends using ANN when you have a tuned `numCandidates` value and using ENN when you need exact recall baselines or when querying smaller data volumes, for example, under 10,000 documents, where exhaustive comparison is still practical.

For ANN queries, the `numCandidates` value controls how many nearest neighbors are considered before returning the results, while the limit is the number of final results returned to the application. MongoDB recommends using an initial test configuration where `numCandidates` is at least 20 times the limit, then tuning from that starting point based on quality and latency. For instance, if you want to return 10 final results to the user (`limit: 10`), start by evaluating 200 candidates during ANN search (`numCandidates: 200`), then adjust based on your own data distribution, relevance goals, and latency targets.

The following `$vectorSearch` stage shows this configuration in a minimal aggregation snippet:

```
{
  $vectorSearch: {
    index: "books_vector_index",
    path: "embeddings",
    queryVector: userQueryEmbedding,
    numCandidates: 200,
    limit: 10
  }
}
```

This query configuration uses a common ANN tuning strategy, sometimes called over-requesting. It asks MongoDB Vector Search to evaluate a larger candidate set (200) before returning the final top results (10).

The following example shows the same ANN pattern in C# using the `fluent .VectorSearch()` API:

```
public async Task<List<BookSearchResult>> SearchBooksAsync(float[]
queryVector)
{
  var options = new VectorSearchOptions<Book>
  {
    IndexName = "books_vector_index",
    NumberOfCandidates = 200
```

```
    };

    return await _booksCollection.Aggregate()
      .VectorSearch(b => b.Embedding, queryVector, 10, options)
      .Project(b => new BookSearchResult
      {
        Id = b.Id,
        Title = b.Title,
        Genres = b.Genres
      })
      .ToListAsync();
}
```

This C# version keeps everything in the fluent driver pipeline, including the vector stage. In practice, this approach keeps your query code readable and avoids manual stage assembly for common ANN query patterns.

If your embedding pipeline currently returns `double[]`, convert it to `float[]` at query time, or standardize it upstream so that the vector type is consistent with your driver query path. This is important because larger candidate sets generally increase recall quality, especially as your corpus grows or as quantization compresses vectors. The trade-off is latency and compute. Treat this value as a controlled tuning dial, not a fixed magic number.

Use pre-filtering to cut candidate space before scoring

Many vector search workloads include obvious constraints, such as tenant, language, catalog region, access level, document status, or content type. Apply these as pre-filters within `$vectorSearch`, provided the corresponding fields are indexed as filter fields in the vector search index definition. Doing so reduces the candidate space before similarity scoring runs, which can improve performance and reduce noisy cross-segment matches.

The following example applies tenant, language, and availability filters directly in the `$vectorSearch` stage before nearest-neighbor scoring runs:

```
{
  $vectorSearch: {
    index: "books_vector_index",
    path: "embeddings",
    queryVector: userQueryEmbedding,
    numCandidates: 300,
```

```
    limit: 15,
    filter: {
      tenantId: "leafy-library-uk",
      language: "en",
      isAvailable: true
    }
  }
}
```

This query applies business constraints directly in the vector stage, so irrelevant partitions are excluded before nearest-neighbor ranking runs.

In C#, you can keep pre-filter logic fully typed by passing a driver filter into `VectorSearchOptions`:

```
public async Task<List<Book>> SearchAvailableBooksAsync(float[]
queryVector)
{
  var typedFilter = Builders<Book>.Filter.And(
    Builders<Book>.Filter.Eq(b => b.TenantId, "leafy-library-uk"),
    Builders<Book>.Filter.Eq(b => b.Language, "en"),
    Builders<Book>.Filter.Eq(b => b.IsAvailable, true));

  var options = new VectorSearchOptions<Book>
  {
    IndexName = "books_vector_index",
    NumberOfCandidates = 300,
    Filter = typedFilter
  };

  return await _booksCollection.Aggregate()
    .VectorSearch(b => b.Embedding, queryVector, 15, options)
    .ToListAsync();
}
```

This approach avoids hand-writing BSON filter definitions and lets the C# driver map field names from your model safely. This benefit becomes especially valuable as your filter logic grows.

Pre-filtering improves both relevance and efficiency in multi-tenant or segmented datasets. It is one of the highest-impact techniques for keeping ANN quality stable as collections become larger and more heterogeneous.

Measure performance as a loop, not a one-time setup

A reliable vector search evaluation process is iterative because index settings, model choices, and quantization changes can all shift both relevance quality and latency. The workflow below gives you a repeatable way to measure those changes and make tuning decisions with evidence.

A mature tuning workflow is iterative:

1. Define a benchmark set of representative queries and expected relevant results, then choose evaluation metrics upfront, such as Precision@k, Recall@k, and NDCG@k.
2. Run ENN search on a representative sample with query judgments to establish a quality reference.
3. Run ANN variants with different `numCandidates` values.
4. Compare overlap with ENN results and track latency percentiles.
5. Repeat after any index, embedding model, or quantization change.

In practice, *Step 2* works best as a small evaluation workflow. Start with a judged query set, a table in which each query has known relevant documents and, if useful, graded relevance labels such as high, medium, and low relevance. Then run ENN for those same queries on a representative sample of your corpus, using the same filters and result limits you expect in production. This gives you a high-confidence reference ranking for that sample, which you can use to score ANN runs consistently with Precision@k, Recall@k, and NDCG@k.

In these metric names, **@k** denotes the "top-k cutoff," where **k** is the number of highest-ranked results you evaluate, for example, top 5, top 10, or top 20.

Precision@k measures how many of the top-k results are actually relevant. Recall@k shows how much of the total relevant set you are recovering within the top-k window. NDCG@k adds ranking sensitivity, rewarding systems that place the most relevant results near the top rather than merely somewhere in the list.

MongoDB generally characterizes strong ANN behavior as high overlap with ENN results while maintaining practical latency. In production, your target overlap and latency **service-level objectives** (**SLOs**) should come from product expectations, not generic benchmarks.

For the Leafy Library application, that means you might accept a slightly lower overlap for broad discovery queries if your response times remain smooth under load but require higher overlap for intent-sensitive recommendations involving grief, trauma, or age-range content, where ranking quality is critical.

Selecting an embedding model

If vector search quality is the output, selecting the right embedding model is one of the most important inputs.

A vector index can be perfectly configured and still produce underwhelming results if the embeddings do not represent your domain language well. Conversely, a strong model can often outperform heavier infrastructure tuning because it provides the index with better semantic geometry from the start. In other words, you cannot indefinitely compensate for weak embeddings through tuning.

Before comparing model benchmarks, first define the retrieval scenario you want to optimize for:

- Are queries short and vague, or long and descriptive?
- Are documents short snippets, long passages, or of mixed lengths?
- Does your domain primarily use general language, technical jargon, legal language, medical terminology, or internal acronyms?
- Is the workload monolingual, multilingual, or code-heavy?
- What type of data are you working with? MongoDB Vector Search fully supports many types of content beyond text, such as audio, video, and images. This will influence which model you select.

MongoDB guidance emphasizes that model choice affects both retrieval quality and index design constraints, including vector dimensions. Higher dimensions can capture more nuance, but they increase memory footprint and index costs.

For Leafy Library, the work consists mostly of natural-language summaries, genres, and reader-intent queries. That profile often favors models with strong semantic retrieval on narrative text rather than models optimized for code or image-text workloads.

When choosing embedding models for MongoDB Vector Search, evaluate at least the following dimensions:

- Retrieval quality for your query and document pairs.
- Embedding dimensions and the resulting index memory footprint.
- Token context capacity for your document chunking strategy.
- Inference cost and throughput constraints.
- Latency and availability profile of your embedding pipeline.

A model with stronger benchmark scores but very high dimensions can create downstream pressure on memory and search tiers. A slightly smaller model with better operational fit might produce better overall product outcomes.

Every model decision propagates into later sections of this chapter. Higher dimensions usually increase the index footprint, which affects memory planning. Larger vectors can also increase query-time candidate-processing costs, which affects latency tuning. That means model choice is not only a retrieval-quality decision, but also an infrastructure and budget decision.

A core operational rule remains unchanged: embeddings are comparable only when they come from the same semantic space. In practice, this means indexing documents and encoding query text with the same model configuration, or with explicitly compatible paired models.

If you migrate models, treat it as a controlled re-embedding event. Partial migrations, in which old and new embeddings coexist in the same index without a deliberate strategy, typically degrade ranking quality in subtle ways.

To make this concrete, here is a practical way to think about compatible model choices using Voyage AI examples:

- `voyage-4-large`: A strong default candidate when retrieval quality is your primary objective and you can afford a higher vector footprint.
- `voyage-4-lite`: A practical candidate when you need a lower-cost profile with good semantic quality for production traffic.

For Leafy Library, a sensible evaluation pattern is to test both of these models against the same judged query set, then compare recall quality and latency at equal `numCandidates` settings.

This comparison is important because it prevents a common trap: selecting a model based only on benchmark reputation, then discovering later that its operational profile forces expensive infrastructure changes. Testing both retrieval quality and operating behavior early keeps the decision grounded.

One important nuance is that, for retrieval workflows, you can embed documents and queries differently by using "`input_type: document`" for corpus text and "`input_type: query`" for user input, while still using the same base model. This is a compatible pairing pattern because both embeddings are generated by the same model family with retrieval-specific prompting behavior.

By contrast, using two different model IDs for document and query embeddings is safe only when the model provider explicitly documents that those models share a compatible embedding space. If that guarantee is absent, assume they are incompatible.

Do not mix model outputs in a single active retrieval index unless you are intentionally implementing a migration strategy with separate index versions.

You should also map model output dimensions directly to the index definition. If the embedding model changes and has different dimensions, update the vector index definition and re-embed all data before the cutover.

To make this easier to apply, consider these three real-world scenarios:

- **Safe pattern**: You embed your catalog with `voyage-4-large` using `input_type: document` and embed user queries with voyage-4-large using `input_type: query`. This is compatible because both sides come from the same base model family.
- **Risky pattern**: You keep documents on `voyage-4-large`, but query embedding switches to a different model because it is cheaper. Unless the provider specifically guarantees shared embedding space compatibility, ranking quality can drift in unpredictable ways.
- **Safe migration pattern**: You build a new index version (for example, `books_vector_v2`) with the new model, re-embed the corpus, and validate quality and latency before switching traffic from `books_vector_v1`.

A core operational rule remains unchanged: embeddings are comparable when they come from the same semantic space. In practice, this means indexing documents and encoding query text with the same model configuration or with explicitly compatible paired models.

When migrating models, treat the transition as a controlled re-embedding event. Partial migrations, where old and new embeddings coexist in the same index without a deliberate strategy, typically degrade ranking quality in subtle ways.

In practice, use versioned index names during migration, such as `books_vector_v1` and `books_vector_v2`, and switch traffic only after quality and latency checks pass on the new index.

Build a model selection harness before committing

Before selecting a production model, it helps to run a lightweight, but structured, evaluation harness. The goal is to compare candidate models under the same conditions so that quality, latency, and cost trade-offs are visible before you commit.

A simple but disciplined harness can prevent expensive mistakes:

1. Define a representative evaluation set (queries, relevant documents, and hard negatives).
2. Generate embeddings with each candidate model.
3. Index each model's output in isolated test indexes.
4. Run identical query suites and capture quality and latency results.
5. Compare operational cost projections based on dimension count and index size.

Where possible, keep `limit`, `numCandidates`, and filter conditions identical across all candidates during evaluation. Otherwise, you risk comparing model behavior and query-tuning behavior simultaneously, which makes decisions unreliable.

The following configuration example shows a practical structure for capturing candidate models, query-set size, and evaluation metrics in one place:

```
{
  "modelEvaluation": {
    "candidates": ["voyage-4-large", "voyage-4.5-lite"],
    "querySetSize": 250,
    "judgedRelevance": true,
    "metrics": ["overlapAt10", "mrr", "p95Latency", "costPer1kQueries"]
  }
}
```

This defines what to evaluate before choosing a production embedding model: candidate models, query sample size, judged relevance, and success metrics.

Metric choice should reflect product intent because different metrics reward different behavior. If the primary risk is missing important results, such as in safety-sensitive recommendations or narrow-intent queries, prioritize recall-oriented metrics such as Recall@k and overlap against ENN. If your primary risk is surfacing too many weak matches on the first screen, prioritize precision-oriented metrics such as Precision@k. If result-ordering quality matters, prioritize ranking metrics such as NDCG@k or **mean reciprocal rank (MRR)**. Then evaluate those quality metrics together with p95 latency and cost per 1,000 queries, so the winning model is not only accurate in isolation but also practical to run at scale.

Model choice is not a philosophical preference; it is an empirical decision. A small evaluation harness provides evidence, helps avoid costly re-embedding churn later, and creates a repeatable process for future upgrades.

Account for automated embedding and manual workflows

MongoDB now supports Automated Embedding workflows with Voyage AI models in Atlas. This feature can simplify operational overhead for teams that want faster adoption with fewer moving parts.

Manual embedding workflows are still relevant when you need strict control over preprocessing, custom chunking, batching strategy, or provider flexibility. The best practice is not to universally choose one approach, but to select based on your operational constraints and governance requirements.

For Leafy Library, either approach can work. If you value speed of delivery and low operational burden, Automated Embedding is attractive. However, if custom text shaping and strict model lifecycle control are needed, manual embeddings might remain the better fit.

Once you choose the model and embedding workflow, predicting and optimizing index size behavior becomes much easier, which is the focus of the next section.

Managing index size

As vector workloads grow, index size becomes one of the most important drivers of cost and query stability. Large indexes can still perform well, but only if memory planning, quantization choices, and indexing scope are intentional. Without that discipline, teams may encounter rising latency, memory pressure, and sudden tier upgrades that could have been avoided. This section focuses on keeping vector index growth under control without sacrificing relevance quality.

Understand what drives vector index footprint

In MongoDB Vector Search, index size is primarily driven by the number of indexed vectors, vector dimensions, numeric precision and quantization strategy, and additional HNSW graph metadata.

Index size is roughly proportional to the vector count multiplied by per-vector storage cost, with graph overhead layered on top. This relationship explains why both model dimensions and quantization choice have a significant impact on operating cost. It also illustrates why model selection and index sizing should never be treated as separate decisions. If you choose a larger embedding profile, you are implicitly choosing a larger memory requirement unless quantization or other controls offset it.

MongoDB Vector Search supports two approaches to quantization: *automatic quantization of float embeddings*, where you store the original float vectors and MongoDB compresses them during indexing, and *ingestion of pre-quantized vectors*, where you compress the vectors before writing them and store the compact form directly. Automatic quantization is simpler operationally, while pre-quantized vectors can reduce both index memory and stored vector size because the full-precision source vectors are not retained in the retrieval path.

In practice, quantization is especially valuable for larger vector collections, such as those exceeding 100,000 vectors, where memory reduction can materially improve efficiency.

The two common quantization modes are:

- **Scalar quantization**: Strong memory reduction with moderate quality trade-offs.
- **Binary quantization**: Far stronger memory reduction with potentially larger recall trade-offs depending on the workload.

According to MongoDB documentation (`https://mdb.link/building-modern-data-applications-with-mongodb-vector-quantization-docs`), scalar quantization can reduce the RAM cost for vectors to roughly one-fourth of the original (1/3.75), while binary quantization can reduce RAM cost to approximately one twenty-fourth (1/24), with additional behavior such as rescoring against the original float values for candidate refinement.

The following index definition shows scalar quantization enabled alongside filter fields used for pre-filtered vector queries:

```
{
  "fields": [
    {
      "type": "vector",
      "path": "embeddings",
      "numDimensions": 1024,
      "similarity": "cosine",
      "quantization": "scalar"
    },
    {
      "type": "filter",
      "path": "tenantId"
    },
    {
```

```
      "type": "filter",
      "path": "language"
    }
  ]
}
```

This vector index definition enables scalar quantization and includes the filter fields needed for pre-filtered vector queries.

The following example shows the same index setup using the C# driver when creating or updating a vector index:

```
var indexDefinition = new BsonDocument
{
  {
    "fields", new BsonArray
    {
      new BsonDocument
      {
        { "type", "vector" },
        { "path", "embeddings" },
        { "numDimensions", 1024 },
        { "similarity", "cosine" },
        { "quantization", "scalar" }
      },
      new BsonDocument { { "type", "filter" }, { "path", "tenantId" } },
      new BsonDocument { { "type", "filter" }, { "path", "language" } }
    }
  }
};

var model = new CreateSearchIndexModel(
  "books_vector_index",
  SearchIndexType.VectorSearch,
  indexDefinition);

await _booksCollection.SearchIndexes.CreateOneAsync(model);
```

This is one of the places where BSON is still expected by the driver API today. Even so, using `CreateSearchIndexModel` and `SearchIndexType.VectorSearch` keeps the intent explicit and avoids strongly typed command execution.

Quantization is one of the fastest paths to reducing memory pressure and improving cost efficiency for large collections. It also interacts with query tuning: quantized vectors often need higher `numCandidates` values to maintain comparable recall, so you should tune query parameters after enabling quantization.

Right-size dimensions through evidence

It is tempting to assume that higher dimensions always produce better results. In reality, quality gains often plateau while memory usage and costs continue to rise.

A practical approach is to test multiple dimension configurations through your model harness, including Voyage model variants with larger or smaller embeddings and Matryoshka-style embedding families that support useful truncation lengths from the same output. Then, choose the smallest dimension that consistently meets your relevance goals. This approach preserves quality while improving index density and search node usage.

For the Leafy Library application, if two model configurations produce similar top-10 usefulness, the lower-dimension option often provides better long-term operating characteristics.

Do not index vectors for fields that are never used in semantic retrieval. While this sounds obvious, index sprawl usually occurs gradually, adding one extra experimental field at a time.

Apply clear inclusion rules:

- Index vectors only for content that participates in retrieval.
- Keep archival or diagnostic embeddings in separate collections if needed.
- Version embedding fields explicitly when testing migrations.
- Retire unused vector fields after rollout decisions.

This discipline keeps index growth predictable and prevents hidden cost accumulation.

Any major index definition change, especially quantization changes or field reconfiguration, should be treated as a lifecycle event with rollout planning.

Define in advance:

- How long reindexing may take at your scale.
- Which environment validates index quality before production cutover.
- Which metrics gate completion.
- Which rollback path exists if quality regresses.

For teams that skip this planning, index updates become high-stress events. For teams that treat them as normal operations, vector search remains maintainable even as data volume expands.

When and why to use Search Nodes

As your vector workload matures, infrastructure architecture becomes as important as query and index tuning. MongoDB Atlas supports deployment models where search workloads run on dedicated Search Nodes. For production vector search, this is often the architecture that allows you to maintain quality and latency stability without overprovisioning the primary database tier.

It is important to separate product availability from deployment topology terminology. MongoDB Vector Search is available beyond Atlas, including MongoDB Enterprise deployments (with Kubernetes Operator support) and MongoDB Community deployments in current supported versions. However, the term Search Nodes is Atlas-specific language. In self-managed environments, the equivalent pattern is a dedicated `mongot` infrastructure that is separated from primary `mongod` workload hosts.

Search Nodes is not a default requirement for every project. They are a scaling and isolation strategy that becomes increasingly valuable as index size, concurrency, and performance expectations rise.

Why Search Nodes matter for vector workloads

MongoDB guidance emphasizes that vector search performs best when vector indexes are held in memory. Dedicated Search Nodes help by isolating search processing from core database workloads and allowing independent sizing for search resources.

If you are running outside Atlas, apply the same principle by isolating `mongot` resources from transactional database resources where possible. The architecture goal is the same even when the operational tooling differs.

This approach provides three practical benefits:

- Better workload isolation between transactional operations and vector search.
- Independent scaling of search CPU and RAM.
- More predictable query latency under mixed workload pressure.

For Leafy Library, this matters when semantic search becomes a high-traffic user path. Without isolation, bursts in recommendation or discovery traffic can compete with ordinary CRUD operations and create an unpredictable user experience.

As a starting planning rule, MongoDB recommends sizing node RAM to at least 10 percent more than the total vector index size.

It also notes that CPU availability strongly influences query concurrency and latency. In some cases, a tier with sufficient RAM but low CPU can underperform compared to a configuration with better CPU concurrency characteristics.

The practical takeaway is straightforward: size for both memory fit and concurrency headroom.

For self-managed environments, apply this same rule to dedicated search infrastructure sizing, accounting for memory to fit the index and CPU for concurrency, even though you are not selecting Atlas Search Node tiers.

When migrating to dedicated Search Nodes or changing search tiers, Atlas handles index build and transition behavior, but you should still plan for index lifecycle effects.

Scaling events can trigger index rebuild or copy behavior depending on the deployment context. Therefore, you should treat them as controlled operations with observability and rollout windows, especially for business-critical search flows.

In operational terms, this means:

- Schedule scaling changes during lower-risk windows.
- Monitor query latency and index queryable status during transitions.
- Keep a rollback plan if quality or latency deviates.

For Leafy Library, moving to dedicated Search Nodes is usually justified when semantic search becomes central to discovery and personalization rather than being an optional enhancement.

Summary

This chapter focused on production best practices for MongoDB Vector Search. You began by exploring vector search tuning as a three-way balance between quality, latency, and cost, learning how ANN and ENN modes serve different roles. You also learned how to tune `numCandidates`, and how pre-filtering can improve both relevance and performance in segmented datasets.

Next, you explored embedding model selection as a foundational design decision, including how model dimensions, retrieval quality, and operational constraints interact, and why a model evaluation harness is essential before committing to production. The chapter also addressed index size management, including quantization strategies, index scope discipline, and lifecycle planning for index updates. These practices help keep vector workloads cost-effective as data grows.

Finally, you examined Search Nodes and when dedicated search infrastructure becomes the right choice, covering workload isolation, memory and CPU sizing principles, and operational practices for stable performance at scale.

At this point, you have a practical framework for running MongoDB Vector Search sustainably in real applications. Whether you are planning a new build or refining an existing one, these practices help you make quality, latency, and cost decisions with confidence.

Additional resources

If you want to learn more about MongoDB Vector Search best practices as well as other useful related content, including our MongoDB Learn content, the following resources are useful:

- *MongoDB Vector Search Overview*: `https://mdb.link/building-modern-data-applications-with-mongodb-vector-search-docs`.
- *MongoDB AI Learning Hub*: `https://mdb.link/building-modern-data-applications-with-mongodb-ai-learning-hub`.

Vector Search Fundamentals

Learn how to use MongoDB Atlas Vector Search to build AI-powered search experiences using indexing, embeddings, and retrieval strategies.

`https://mdb.link/badge-vector-search-press-donet`

Vector Search Performance

Learn how to diagnose, optimize, monitor, and tune MongoDB Vector Search performance in production.

https://mdb.link/badge-vector-search-perf-donet

Voyage AI with MongoDB

Learn how to create, store, and query vector embeddings in MongoDB, use auto-embedding and Voyage AI for vector search and reranking, and optimize search quality, latency, and cost.

https://mdb.link/building-modern-data-applications-with-mongodb-voyageai-badge

Part 4:
Putting It All Together

In this fourth and final part of the book, you will consolidate everything you have built and turn it into a practical blueprint that you can apply to your own work. You will begin by distilling the key technical and architectural lessons from the book, bringing together aggregation, search, and semantic retrieval as a coherent approach to building intelligent data applications. You will then move to the *Appendix*, which provides a detailed reference for Leafy Library. There, you will explore the dataset, import it into your MongoDB environment, understand the application structure and architecture, review visual walkthroughs of the application, and run the project end to end on your machine.

By the end of this part, you will have a clear summary of the concepts and techniques covered throughout the book, along with a complete hands-on reference implementation that you can study, run, extend, and adapt for your own applications.

This part of the book includes the following chapters:

- *Chapter 10, Key Takeaways and Next Steps*
- *Chapter 11, Appendix*

10

Key Takeaways and Next Steps

We have covered a lot of ground together in this book. By this point, you have seen how retrieval has evolved from shaping data with aggregation pipelines to building practical search experiences, and finally to semantic and hybrid strategies that better reflect real user intent.

This progression is the central thread of Leafy Library. You started by building clear query structures and more advanced queries in a user dashboard, then added lexical relevance, expanded into semantic understanding, and finally tuned your solution for production constraints such as quality, latency, and cost.

This chapter serves as a consolidation point. It steps back to extract the patterns that matter most once the tutorials are finished and extends that discussion into the next choices you are likely to face in a real application.

In this chapter, you will review the strongest takeaways from aggregation, MongoDB Search, MongoDB Vector Search, and hybrid retrieval, turning those lessons into a practical decision framework for choosing lexical, vector, or hybrid approaches. You will then use that foundation to go further into topics that were only introduced briefly earlier, such as rerankers for final-result tuning. You will also examine where other products, such as MongoDB Charts and Stream Processing, can extend the functionality of your applications, and see how MongoDB supports **Entity Framework (EF)** Core, which is commonly used by C# developers.

The goal is simple. You should leave this chapter with a clear, reusable playbook that helps you summarize what you learned, decide what to apply next, and identify where deeper investment will pay off beyond Leafy Library.

This chapter will cover the following topics:

- Core retrieval patterns in practice
- Designing and refining your retrieval strategy
- Taking it further: Charts, multimodal search, and event-driven updates
- Special note for .NET teams: EF Core support

Technical requirements

To get the most value from this chapter, you should have completed the earlier chapters, especially those in *Part 2*, *Searching Your Data*, and *Part 3*, *Semantically Searching Your Data*. You should also have access to your Leafy Library codebase and implementation notes, as well as a list of your own project goals, such as improved relevance or lower latency. Ideally, you should also have access to your MongoDB metrics to validate the guidance against real observations.

The sample book repository used throughout this book is available at `https://mdb.link/building-modern-applications-with-mongodb-repo-mainbranch`.

Core retrieval patterns in practice

The aggregation framework, MongoDB Search, MongoDB Vector Search, and hybrid search each solve different parts of the retrieval problem, but their real value becomes clearer when considered together. The following sections revisit these patterns and highlight the principles that are most important when applying them in production. Together, these principles provide the foundation for the decision framework that follows, where you will learn how to choose the most appropriate retrieval strategy for a given use case.

Aggregation framework

The aggregation framework has been one of the quiet pillars of this book. Even when the focus was on search, autocomplete, facets, vector retrieval, or troubleshooting, the underlying execution pattern was still an aggregation pipeline. That consistency matters because it means you can combine retrieval, filtering, shaping, and post-processing in one composable query model.

For Leafy Library, this provides three practical benefits. First, complex retrieval logic can be kept in the database layer rather than moving raw documents into memory for processing. Second, feature behavior can be composed incrementally with staged operators, starting with retrieval, followed by projection, pagination, and optional metrics. And third, query behavior can be reasoned about in a structured way because each stage has a single responsibility and a clear effect on output.

When developers first adopt MongoDB, they often think of aggregation as an advanced feature used only for analytics. In practice, the chapters in this book show that aggregation can also power everyday application queries by giving you one place to combine retrieval, filtering, ranking support, reshaping, and response preparation. In modern application engineering, aggregation is not an optional advanced tool, it is often the core execution model for production-grade read paths.

A good way to frame this distinction is to separate architecture from retrieval. Aggregation establishes architecture-level consistency while the search capabilities provide retrieval-level intelligence.

As your application grows, maintaining correct and manageable retrieval workflows becomes more difficult when they are spread across many layers. Aggregation keeps these workflows explicit, testable, and explainable. This clarity is especially valuable when product teams ask for new filters or ranking constraints, when you need to debug missing results under time pressure, or when you are balancing response quality against latency and cost constraints.

If you take one practical habit forward from this topic, it should be to design your query workflows as readable pipelines with an intentional stage order, because clear stage sequencing makes query intent easier to review, defects easier to isolate, and later changes less likely to introduce hidden regressions.

Throughout this book, aggregation has served as the execution backbone for these workflows, providing a structured and maintainable way to combine retrieval, filtering, and data shaping without introducing ad hoc logic across different application layers.

MongoDB Search

MongoDB Search gave Leafy Library strong lexical retrieval capabilities, including full-text matching, autocomplete, facets, and relevance tuning. This was a major milestone because it moved the application from browsing to true search behavior.

The most important lesson was not any single operator. It was that quality search behavior comes from good index design and query intent alignment, not from one magical query setting.

In *Part 2*, *Searching Your Data*, the strongest production pattern was intentionality. Static mappings kept indexed fields deliberate and growth predictable, while careful analyzer choices protected token compatibility between indexing and querying. You also saw that autocomplete works best when it is scoped to real user-intent paths rather than applied broadly, and that facet behavior is only reliable when facet fields are explicitly mapped with the correct facet types.

Equally important, query shape had to remain disciplined. Projecting only the fields needed by the interface and applying limits early kept search behavior efficient and easier to reason about as the application evolved. These patterns are less flashy than new feature launches, but they are exactly what separates a demonstration search from a trustworthy product search experience.

A useful mindset from this part of the book is that relevance is a matter of product quality. If users repeatedly retype or reformulate search queries, it is not merely a technical quirk. It is a signal that retrieval quality, query interpretation, or result presentation needs attention. MongoDB Search gives you the controls to improve this behavior, but the improvement process is iterative and measurable.

Figure 10.1 – MongoDB Search feedback loop from user intent to confidence

Figure 10.1 captures a practical reality from the MongoDB Search chapters. Relevance quality is not a one-time setup step. It is a loop, and user behavior is part of the signal. In practice, MongoDB Search became reliable when treated as a product capability rather than a query feature, with the biggest gains coming from deliberate index design, analyzer choices, and disciplined query shaping.

MongoDB Vector Search

MongoDB Vector Search added semantic retrieval to the Leafy Library. Semantic retrieval allowed the application to answer intent-level queries even when document wording did not closely match user phrasing. This is where search started to feel intelligent to end users.

From the chapters in *Part 3*, *Semantically Searching Your Data*, the most valuable lessons centered on embeddings as foundational to retrieval quality. Weak embeddings cannot be tuned away by index tweaking alone. The key practices were to keep query and document embeddings in a compatible semantic space by using the same model family, to treat **approximate nearest neighbor (ANN)** settings as trade-off controls rather than fixed constants, to apply filters intentionally for segmentation before similarity scoring, and to evaluate quality, latency, and cost together rather than optimizing for a single metric. Semantic retrieval is powerful, but it is also easier to mis-tune when teams treat it as a single-metric system.

Model choice is an architectural choice

One subtle but important insight from this part of the book is that model selection affects much more than relevance. It also affects vector dimensions, index footprint, infrastructure cost, and operational complexity.

That is why the best practice was to use a repeatable evaluation harness before finalizing model and index decisions. *Table 10.1* summarizes the vector search operating lens applied throughout the discussion.

Dimension	Primary question	Typical tuning considerations	Experience, if ignored
Quality	Are top results semantically correct?	Embedding model, `numCandidates`, evaluation set	Users get plausible but wrong results
Latency	Are responses fast enough under real load?	Candidate count, filter strategy, workload isolation	Good results arrive too slowly
Cost	Is retrieval sustainable at scale?	Vector dimensions, index sizing, node strategy	Quality is unaffordable in production

Table 10.1 – Vector Search operating framework for production decisions

You can use this framework as a recurring review checklist when tuning semantic retrieval in any application, not just Leafy Library. MongoDB Vector Search unlocks meaning-based retrieval, but its success in production depends on balancing quality, latency, and cost together, and on treating embedding and model selection as architectural decisions.

Hybrid search

Hybrid search combines lexical retrieval and semantic retrieval in a single workflow. Hybrid search is often the strongest practical default because user queries vary. Some queries are exact and term-driven, while others are vague, thematic, or intent-heavy. A single retrieval mode rarely handles both equally well.

For Leafy Library, hybrid behavior was a natural progression where lexical logic handled exact titles, known names, and explicit metadata with strong precision, while vector logic captured intent-based phrasing and conceptual similarity. Together, these two approaches produce better first-page relevance than either approach in isolation.

Hybrid search preserves precision signals from lexical matches, adds recall and intent coverage from semantic proximity, and reduces brittle behavior when users phrase their needs in unexpected ways. This is especially useful in product experiences where discovery and precision need to coexist.

Hybrid search is not simply a matter of running both modes and hoping for the best. It still requires clear weighting logic, evaluation criteria, and fallback behavior. In other words, hybrid search raises your search ceiling, but only when it is tuned intentionally. For this reason, hybrid search is often the strongest default for mixed intent, because lexical and semantic signals complement each other when weighting and evaluation are handled intentionally.

Designing and refining your retrieval strategy

Once you understand how these retrieval modes behave individually, the next step is to turn that knowledge into a practical tuning workflow. Start by choosing the retrieval approach that best matches the user intent in front of you and evaluate whether it meets the quality bar for the feature. Then, if the candidate set is broadly correct but the final ordering still needs work, refine that first-stage retrieval with reranking. The following sections walk through that progression in order.

Decision framework: Vector, full-text, or hybrid

The preceding sections brought together the key concepts and patterns covered throughout the book. With that context in place, let us now turn those lessons into an actionable framework. The goal is not to memorize rules, but to choose a retrieval strategy quickly and defensibly when starting a new feature or tuning an existing one.

A practical starting question is: *what type of user intent dominates this feature?* If intent is mostly exact and term-specific, start with lexical search. If intent is mostly descriptive and conceptual, start with Vector Search. If intent is mixed or uncertain, start with hybrid.

Table 10.2 provides a compact strategy overview that you can reuse with your own teams.

Retrieval Strategy	Best fit	Strength	Trade-off	Good first use cases
MongoDB Search	Known-term and structured queries	High precision for exact terms	Misses semantically related phrasing	SKU or name lookup, exact title or author search
MongoDB Vector Search	Meaning-driven and exploratory queries	Captures intent and paraphrases	Requires careful embedding and cost tuning	Recommendations, natural-language discovery
Hybrid	Mixed intent and broad user behavior	Balances precision and semantic coverage	More tuning complexity	Consumer search experiences, content discovery

Table 10.2 – Retrieval strategy decision framework for full-text, vector, and hybrid modes

When deciding in a real project, apply the following steps:

1. Identify query categories from real logs or expected product behavior.
2. Define quality success criteria for top results, not only aggregate metrics.
3. Choose an initial strategy from *Table 10.2*.
4. Run a focused evaluation set and compare user-visible outcomes.
5. Promote to hybrid if either precision or intent coverage remains weak.

This process keeps decisions grounded in behavior rather than preference, but it works only when testing is treated as a continuous discipline rather than a one-time checkpoint.

A strong retrieval strategy starts with choosing a lexical, vector, or hybrid strategy based on dominant user intent and validating that choice with a small, repeatable evaluation process before scaling it. In practice, this means tracking ranking quality with metrics such as **Normalized Discounted Cumulative Gain (NDCG)**, which rewards placing highly relevant results near the top of the ranked list and penalizes relevant results that appear lower down. Use NDCG alongside measures such as Precision@K and Recall@K, then iterate until retrieval quality is consistently strong for your real query mix.

Rerankers for further tuning

Rerankers are a powerful next step once your first-stage retrieval is working. A reranker takes an initial candidate set, often produced by lexical, vector, or hybrid retrieval, and reorders those candidates using a stronger relevance model. Crucially, the model evaluates not just the query in isolation but the query and each candidate document together. Earlier chapters provided only enough coverage to place rerankers within the broader retrieval landscape. Here, it is worth examining them in depth as a practical next-stage design choice, because rerankers tend to become relevant as soon as a team has a decent first-stage search experience and wants to improve the top results without rebuilding the entire retrieval stack.

First-stage retrieval is usually optimized for speed and broad relevance. Reranking is optimized for precision in the final presented results.

In practical terms, rerankers are useful when result sets are mostly good but top positions are unstable or off-target, when business-critical workflows require stronger confidence in the first few results, or when you need to combine multiple signals such as lexical score, semantic proximity, freshness, and business constraints.

That distinction matters because rerankers do not replace retrieval. Their job is to improve ordering within a candidate set that is already good enough to work with. If your first-stage query is returning obviously wrong or incomplete candidates, the problem lies in retrieval, indexing, filtering, or embeddings. If your candidate set is broadly correct but the top few results still feel inconsistent, reranking becomes a sensible next investment.

A common pattern looks like this:

1. Retrieve the top N candidates using lexical, vector, or hybrid query logic.
2. Apply a reranking model to those N candidates.
3. Return the top K reranked results to the user.

This pattern controls computational cost by reranking only a small candidate set. In practice, it also gives teams a cleaner experimentation path. You can keep a stable first-stage retrieval design, apply reranking to a limited candidate window, and compare whether the top results feel more trustworthy, without having to redesign every search endpoint at once.

This approach also makes the cost trade-off measurable. A reranker typically increases compute per query, so the decision to add one should be based on evidence that precision gains justify the additional cost.

Treat reranking as an optimization layer, not a substitute for good retrieval foundations. If the candidate set quality is weak, rerankers can only do so much. Strong reranking starts with strong first-stage retrieval.

First-stage retrieval might already be good enough for your application. Before adding reranking in production, use a strong testing suite to compare baseline and reranked results using measurable metrics such as NDCG, Precision@K, Recall@K, response latency, and cost per query. If the quality improvement is small relative to the compute and latency impact, retaining the simpler first stage can be the better engineering decision.

Rerankers work best as a precision layer on top of an already solid retrieval stage, and they should be adopted only when testing demonstrates that measurable relevance gains are worth the additional compute and latency cost.

Taking it further: Charts, multimodal search, and event-driven updates

At this point, you have a solid search and retrieval foundation. The next question is what else you should be aware of to help you build modern data applications in C# with MongoDB.

MongoDB Charts as a visualization layer

Although MongoDB Charts did not appear in the earlier chapters, it is an important capability to be aware of. Once you have search, facets, recommendations, or semantic retrieval in place, you will gradually want better ways to present your data and tell clearer stories through visuals.

That is where Charts becomes practical. It gives you a native MongoDB capability to build rich visual representations of your data without generating every graph in application code or wiring in separate third-party graphing libraries.

For Leafy Library, a practical example is moving from custom HTML-based graphs in the user dashboard to MongoDB Charts-backed visuals. This shift can reduce front-end charting overhead, broaden the available chart styles, and make it easier to add new views as product questions evolve. For example, you can use it to create visuals such as reading trends by genre, author popularity over time, borrowing activity by audience segment, and curated discovery dashboards that combine catalog and user interaction data.

Good search and recommendation experiences are not only about query tuning. They are also about presenting information in ways that readers, product teams, and stakeholders can quickly understand. That is the deeper reason Charts belongs in this chapter even though it was not part of the earlier build steps. Once an application is live, visualization becomes part of how a team communicates product value. Charts give teams a fast way to design polished dashboards, reuse chart layouts, and publish new views without building custom graph components first.

It is also worth noting that MongoDB can store GeoJSON location data that MongoDB Charts can visualize in map-based views. If your application includes location fields, this makes geographic dashboards and spatial reporting possible. Leafy Library might not need this immediately, but understanding that geographic views are available is part of seeing the full capability of Charts.

Multimodal Vector Search beyond text

Another high-value extension is to use Vector Search with more than text embeddings. If your catalog includes book covers, product photos, or other media, you can generate image embeddings and store them alongside your existing metadata so that retrieval can use visual similarity in addition to language cues.

For example, in a travel discovery application, multimodal retrieval could help users find destinations, hotels, or activity listings with a visual style similar to an uploaded image, then combine that visual match with text-based constraints such as location, budget, or amenities.

A practical rollout pattern is to start with one new media type, keep the index scope focused, and evaluate quality before scaling. In many teams, that means first embedding a single image field, storing vectors in a dedicated field, and then testing hybrid combinations in which lexical constraints and image similarity each contribute to final candidate quality.

The key idea is that Vector Search is not limited to plain-text use cases. Once the embedding pipeline supports additional media types, the retrieval layer can become genuinely multimodal while maintaining the same core evaluation discipline around quality, latency, and cost.

voyage-4-context for RAG applications

Another practical extension is model-focused tuning for **retrieval-augmented generation** (**RAG**) workloads. If your team is evaluating `voyage-4-context` for embeddings, treat it as a retrieval design decision rather than a drop-in model swap.

In practice, this means building or reusing a RAG-specific evaluation set, testing whether `voyage-4-context` improves retrieval quality for your domain queries, and comparing that quality lift against latency and cost impact. The same framework from earlier sections still applies: measure NDCG, Precision@K, and Recall@K on real query distributions before committing to a production rollout.

For Leafy Library, this could be useful for conversational discovery or assistant-style features where the retrieval stage feeds generated answers. The core guidance is to adopt `voyage-4-context` only when measurable retrieval gains are clear for your content and user intent patterns.

Triggers and stream processing for retrieval freshness

Once retrieval quality improves, freshness becomes the next challenge. User interactions change what should be surfaced, inventory status changes what can be shown, and content updates can invalidate previously strong candidates.

This is where event-driven patterns help. MongoDB Change Streams provide the underlying real-time feed of inserts, updates, and deletes. Atlas Triggers sit close to that change feed and are useful for immediate reaction work, such as updating a derived field, enqueueing a follow-up task, or kicking off selective re-embedding after an important document edit. MongoDB Atlas Stream Processing handles the broader continuous-transformation layer, where many events can be filtered, enriched, aggregated, and converted into retrieval-ready signals over time. Together, these products let you keep derived fields, recommendation signals, or profile vectors current without relying solely on scheduled batch jobs.

For Leafy Library, that could mean updating reader-preference signals when borrowing behavior changes, refreshing popularity features used by ranking logic, or triggering selective re-embedding workflows when key metadata fields are edited.

A focused rollout is the best way to put this event-driven approach into practice. Start with one high-impact event path, verify its correctness and latency, and then expand from there. Event-driven systems are powerful, but they work best when the initial rollout is narrow, observable, and tied to a measurable retrieval outcome.

As your application matures, the next gains often come from presenting data more clearly, expanding retrieval beyond text, and keeping ranking signals current as behavior changes. MongoDB Atlas Charts, multimodal Vector Search, and event-driven update patterns each extend the system in a different direction, but together they help move a capable search feature toward a more complete product experience.

Suggested enhancement roadmap for Leafy Library

To continue evolving the sample application, consider the following strong candidates for further development:

- Add a hybrid retrieval endpoint and compare it with the existing lexical-only and vector-only routes.
- Add a reranking stage for top candidates and measure first-page precision changes.
- Replace one HTML dashboard graph with a MongoDB Atlas Charts visualization and compare clarity, flexibility, and maintenance effort.
- Add cover-image embeddings and test multimodal retrieval quality alongside text-based retrieval.
- Add one event-driven update path (Atlas Trigger or MongoDB Atlas Stream Processing) to keep a retrieval signal fresh.
- Add a lightweight relevance evaluation harness to support release-safe tuning.
- Evaluate EF Core integration options for teams that prefer EF-centered workflows in adjacent modules.

The final takeaway is that momentum after completing this book comes from combining continued learning, visible validation, and a focused enhancement roadmap. This can help you translate the knowledge gained across these chapters into measurable product progress.

Special note for .NET teams: EF Core support

If your team uses EF Core as a primary **object-document mapper (ODM)** or data-access abstraction, you can work with MongoDB through EF Core support. This gives you a familiar on-ramp when you want to evaluate or adopt MongoDB without abandoning established .NET development patterns on day one.

At a high level, the experience is what many .NET developers expect: you work with entity classes, use a context-based access pattern, and write LINQ-oriented queries within the supported feature set. This can reduce migration friction for relational-first codebases and help teams experiment incrementally instead of forcing an all-at-once rewrite.

Use EF Core support when familiarity and team velocity are your priority, and use MongoDB-native driver patterns directly for advanced search, aggregation-heavy workflows, and vector retrieval scenarios where you need full access to MongoDB-specific capabilities. Because the MongoDB EF Core Provider is built on top of the MongoDB C# Driver, you can still access those additional features without installing a separate driver package.

Summary

This chapter closes the loop on the full journey from aggregation and retrieval fundamentals to production-ready search design.

In this chapter, you revisited how the aggregation framework enables composable query workflows, how MongoDB Search provides strong lexical relevance and faceting behavior, and how MongoDB Vector Search expands retrieval into semantic understanding. It also examined why hybrid search is often the practical default for real-world, mixed-intent queries and introduced a reusable decision framework for choosing between lexical, vector, and hybrid strategies based on product behavior.

From there, you examined rerankers as a precision layer for tuning top results, explored MongoDB Atlas Charts as a native visualization layer for product data, and saw how multimodal Vector Search can extend discovery beyond text. You also learned how Atlas Triggers and MongoDB Atlas Stream Processing provide event-driven approaches for keeping retrieval signals fresh, along with EF Core support for .NET teams that prefer a familiar data-access approach.

Most importantly, this chapter translated the content of the entire book into a practical next-steps playbook. You now have the concepts, implementation patterns, and decision tools needed to build modern data applications that do not merely store information, but help users find the right information with confidence.

That is the real goal this book has been working toward, since the first chapter.

Additional resources

This final section serves as a practical continuation map to guide your learning after finishing the book. Use the following checklist to map chapter topics to your ongoing learning and validation activities.

Topic Area	Recommended next step	Skill badge
Aggregation framework	Build two additional pipelines from your own domain data	Fundamentals of Data Transformation – `https://mdb.link/badge-aggregation-press-donet`

Topic Area	Recommended next step	Skill badge
MongoDB Search fundamentals and tuning	Reproduce lexical search tuning on a second dataset	Search Fundamentals – https://mdb.link/badge-search-press-donet Retrieval Evaluation – https://mdb.link/building-modern-data-applications-with-mongodb-retrieval-evaluation-badge
MongoDB Vector Search	Run semantic retrieval evaluation on your own query set	Vector Search Fundamentals – https://mdb.link/badge-vector-search-press-donet Vector Search Performance – https://mdb.link/badge-vector-search-perf-donet Voyage AI with MongoDB – https://mdb.link/building-modern-data-applications-with-mongodb-voyageai-badge

Table 10.3 – Recommended next steps and skill badge links for different topic areas

For deeper implementation guidance and current platform updates, consult the following resources:

- *MongoDB Search Overview*: (https://mdb.link/building-modern-data-applications-with-mongodb-search-docs).
- *MongoDB Vector Search Overview*: (https://mdb.link/building-modern-data-applications-with-mongodb-vector-search-docs).
- *MongoDB Search and MongoDB Vector Search Changelog*: (https://mdb.link/building-modern-data-applications-with-mongodb-search-release-notes).
- *MongoDB .NET/C# Driver Release Notes*: (https://mdb.link/building-modern-data-applications-with-mongodb-driver-release-notes).
- *MongoDB Entity Framework Core Provider*: (https://mdb.link/building-modern-data-applications-with-mongodb-efcore-docs).
- *MongoDB Atlas Charts*: (https://mdb.link/building-modern-data-applications-with-mongodb-charts-docs).

- *MongoDB Change Streams*: (`https://mdb.link/building-modern-data-applications-with-mongodb-changestream-docs`).
- *MongoDB Atlas Triggers*: (`https://mdb.link/building-modern-data-applications-with-mongodb-triggers-docs`).
- *MongoDB Atlas Stream Processing*: (`https://mdb.link/building-modern-data-applications-with-mongodb-streamprocessing-docs`).

Because search and AI-adjacent features evolve quickly, always verify version details and availability against the current official documentation before applying this guidance in production.

11

Appendix

This appendix serves as a practical reference for the Leafy Library application used throughout the book. You can use it as a quick setup guide when you want to run the project or as an architecture map to understand where the code fits within the overall application.

Across the chapters, we discuss aggregation pipelines, MongoDB Search, and MongoDB Vector Search in the context of this same application. Here, you can find the implementation details gathered in one place, so you can move from chapter concepts to hands-on execution with less friction.

The topics covered are as follows:

- Quick start – Set up Leafy Library in 5 minutes
- Leafy Library at a glance
- Architecture overview
- Model walkthrough
- Exploring the repository structure
- Running the application locally
- Importing sample collections from the Data folder
- Troubleshooting common startup issues

Technical requirements

Before running Leafy Library, make sure you have:

- .NET SDK 10.0
- A MongoDB deployment (local MongoDB instance or Atlas cluster)

- Git
- The Leafy Library repository: `https://mdb.link/devrel.book.building-modern-data-applications-with-mongodb-repo`

Quick start – Set up Leafy Library in 5 minutes

If you want the shortest path to a working Leafy Library instance, follow these steps first, then return to the detailed sections later. The sections that follow provide deeper context on the architecture, repository structure, branch states, data shape, and startup behavior.

1. If you plan to adapt the code and push your own commits later, forking the repository is recommended before cloning your fork; otherwise, clone the upstream repository directly to run the sample without publishing changes.

   ```
   git clone <your-fork-url>
   cd "Leafy Library"
   ```

2. Set a valid MongoDB connection string and JWT secret in `appsettings.Development.json`:

   - `MongoDb:ConnectionString`
   - `MongoDb:DatabaseName` (typically `leafy_library`)
   - `Jwt:Secret` (use a strong secret, minimum 32 characters)
   - Embedding settings if you plan to run semantic search chapters

   If you need to generate a suitable JWT secret, a simple option on macOS or Linux is:

   ```
   openssl rand -base64 48
   ```

   On Windows PowerShell, you can use:

   ```
   [Convert]::ToBase64String((1..48 | ForEach-Object { Get-Random
   -Maximum 256 }))
   ```

   Copy the generated value into `Jwt:Secret`.

3. Import the sample collections from the `Data` folder:

   ```
   mongoimport --uri "mongodb://localhost:27017" --db leafy_library
   --collection authors --file "Data/leafy_library.authors.json"
   --jsonArray
   mongoimport --uri "mongodb://localhost:27017" --db leafy_library
   --collection books --file "Data/leafy_library.books.json"
   ```

```
--jsonArray
mongoimport --uri "mongodb://localhost:27017" --db leafy_library
--collection issueDetails --file "Data/leafy_library.issueDetails.
json" --jsonArray
mongoimport --uri "mongodb://localhost:27017" --db leafy_library
--collection reviews --file "Data/leafy_library.reviews.json"
--jsonArray
```

Update the `–uri` value with your own connection string for each command.

4. Run the application:

```
dotnet run
```

5. Open the URL printed in the terminal and log in with any username. The first user is promoted to admin automatically.

If you want deeper context after this quick setup:

- See the *Branch guide* section for when to use `main`, `start`, `with-text-search`, or `with-vector-search`.
- See the *Importing sample collections from the Data folder* section for alternative ways to import the sample data using MongoDB Atlas or MongoDB Compass.
- See the *Troubleshooting* section for a checklist to follow if the application fails to start or the sample data cannot be imported.

Leafy Library at a glance

Leafy Library is a Blazor Server application built with .NET 10 and MongoDB. It models a library management workflow where users can browse books, reserve books, borrow books, return books, and write reviews. It also includes search capabilities that evolve through this book.

The Leafy Library application provides a catalog experience for searching and exploring books and authors, authenticated flows for borrowing and reservations, and admin flows for lending reserved books and processing returns. As you progress through the chapters in this book, the same application is extended with full-text and semantic retrieval capabilities backed by MongoDB indexes.

The running scenario matters because each chapter adds or refines one part of this same system. That means the architecture decisions made in one chapter influence what becomes possible in later chapters.

Architecture overview

Leafy Library follows a layered application structure where responsibilities are split cleanly.

UI and interaction layer

The user interface is implemented with Blazor Server components under the `Components` folder, including pages for catalog browsing, book and author details, login, the user dashboard, and the admin dashboard. This layer is responsible for presenting data and collecting user actions, while data access and persistence concerns remain outside the UI.

API layer

The `Controllers` folder exposes REST-style endpoints for books, authors, reviews, reservations, and issue details (borrows and returns). These controllers coordinate incoming requests and delegate business logic to services.

Service layer

The `Services` folder contains most of the core application behavior, including data access and query orchestration, borrow and return workflows, reservation lifecycle handling, JWT support, and startup responsibilities such as search index creation, vector index creation, and embedding backfill for books that do not yet have vectors. This is also where most chapter implementation work takes place.

Data layer (MongoDB)

The application uses MongoDB collections for books, authors, reviews, users, and issue details. The schema and access patterns reflect the same modeling ideas discussed throughout the book, such as embedding frequently displayed related data, keeping loan and reservation records in a unified collection pattern, and handling search-oriented index checks at startup.

Startup lifecycle

On startup, the application configures services and middleware, then ensures required search resources exist before serving traffic:

1. Full-text search index check and creation (if missing).
2. Vector search index check and creation (if missing).
3. Embedding generation for books that do not yet have embedding values.

This startup behavior is useful for chapter workflows because you do not need to manually create indexes every time you run the app.

Visualizing the Leafy Library architecture

Figure 11.1 presents a high-level view of the Leafy Library runtime architecture. It shows the browser entry point, the ASP.NET Core host, including the Blazor UI and REST API, the shared services layer, embedding provider integration, and the MongoDB boundary where both the application collections and search indexes live.

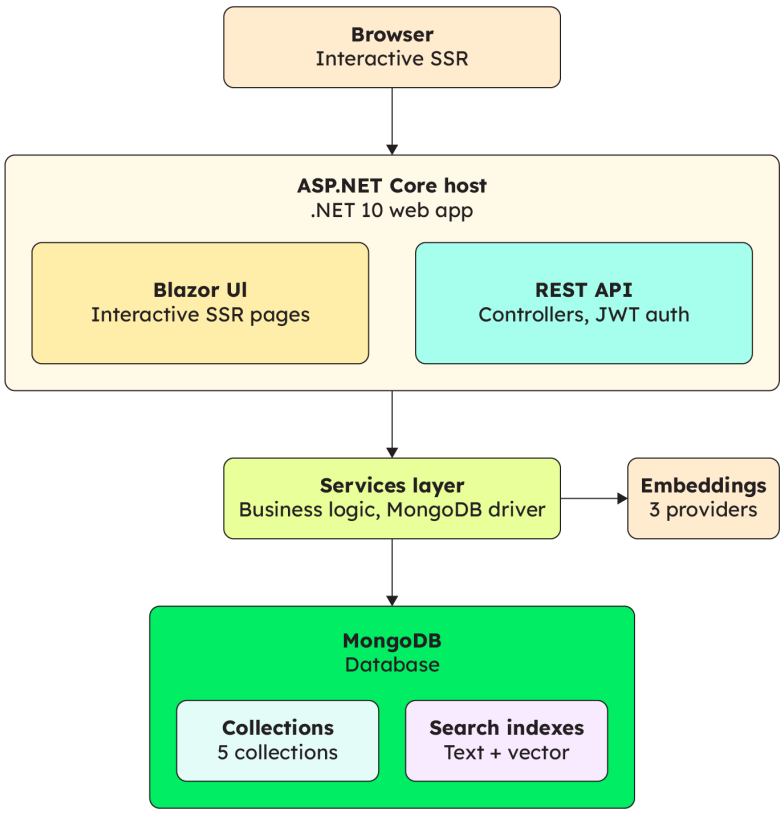


Figure 11.1 – Leafy Library architecture

Model walkthrough

The model layer in Leafy Library does more than map collections one-to-one. It combines domain models, response/projection models, aggregation output models, and configuration models so each layer can work with data in the shape it actually needs.

Domain models mapped to MongoDB collections

The core collection-backed models are `Book`, `Author`, `Review`, `User`, and `IssueDetail`.

- `Book` maps to the `books` collection and includes search-ready fields such as `title`, `genres`, `synopsis`, and `embeddings`, as well as embedded structures, such as `authors` and `reviews`.
- `Author` maps to `authors` and includes name normalization support through `sanitizedName` plus aliases and related book references.
- `Review` maps to reviews and stores text, rating, author name, and timestamp values.
- `User` maps to users and stores `username` and `isAdmin`.
- `IssueDetail` maps to `issueDetails` and uses a single-collection approach for both reservations and borrowed book records.

Embedded helper models and schema patterns

Several helper models express embedded document patterns directly in code.

- `BookAuthor` is a lightweight author reference embedded in each book (extended reference pattern).
- `IssueDetailBook` and `IssueDetailUser` embed minimal book and user details in `issueDetails` records.
- `BookAttribute` holds flexible key-value metadata for books.

This structure keeps read paths simple for common UI views because related display values are available without additional joins.

Custom serializers

Two custom serializers improve resilience against uneven imported data.

- `FlexibleStringSerializer` converts mixed BSON value types into `string` values for book attributes.
- `FlexibleTimestampSerializer` handles review timestamps stored either as epoch numbers or BSON datetime values.

These serializers are especially useful when chapter data imports come from different export histories.

Read/projection and aggregation output models

Not every model is a persisted collection entity. Some models represent computed outputs returned by service methods.

- `AuthorResponse` and `AuthorBookReference` are response models used when author endpoints return resolved book references.
- `GenreCount` represents grouped search facet output.
- `MonthlyLoanCount`, `LoanSummary`, and `LoanSummaryStats` represent dashboard-oriented aggregation outputs.
- `PagedResult` is a reusable wrapper for paginated API responses.

These types keep controller and component code cleaner because query shaping stays explicit and strongly typed.

Configuration models

`MongoDbSettings`, `JwtSettings`, and `EmbeddingSettings` bind configuration from application settings into strongly typed objects consumed by services.

This makes startup configuration safer and easier to reason about, especially when moving between local development, chapter branches, and production-like environments.

Exploring the repository structure

The Leafy Library project root is organized around clear responsibilities, with each folder and configuration file serving a specific role:

- `Components` contains the Blazor UI pages and shared UI elements.
- `Controllers` contains the API endpoints.
- `Services` contains the application's business logic and MongoDB interactions.
- `Models` defines the domain and configuration types.
- `Data` contains the JSON files used to import the sample collections.
- `appsettings.json` and `appsettings.Development.json` provide runtime configuration and local development overrides.
- `Program.cs` brings these components together through dependency injection, authentication, middleware, and startup orchestration.

If you are following the chapter code changes, most modifications will take place in the `Services` and `Controllers` folders, `Program.cs`, and selected pages in `Components`.

Branch guide

The repository is organized into tutorial-friendly branches that represent different progression points:

- `main`: A production-ready baseline snapshot. This is the best branch to use when you want the completed baseline state.
- `start`: A minimal starting point for chapter-by-chapter build-up. This is the best branch to use when you want to follow the foundational implementation steps in the book.
- `with-text-search`: Includes the MongoDB Search full-text search implementation. Use this branch when you are working through the MongoDB Search-focused chapters.
- `with-vector-search`: Includes the vector embedding and semantic search implementation with MongoDB Vector Search. Switch to this branch when you are working on semantic retrieval chapters.

Each of these branches has a different intended purpose and helps set you up for success when implementing the features as you read through the book or using the repository as a reference when implementing the features in your own applications.

Running the application locally

This section explains how to run the application locally, so you get up and running as smoothly as possible:

1. If you expect to customize the project and maintain your own commit history, fork the repository on GitHub and clone your fork:

   ```
   git clone <your-fork-url>
   cd "Leafy Library"
   ```

   If you are only reading along and do not need to push changes, cloning the upstream repository directly is also fine.

2. Update your runtime settings in `appsettings.json` (or `appsettings.Development.json` for local secrets):

   - `MongoDb:ConnectionString`
   - `MongoDb:DatabaseName` (typically `leafy_library`)
   - `Jwt:Secret` (use a strong secret, minimum 32 characters)
   - Embedding settings only if you plan to run semantic chapters

3. If you need to generate a strong JWT secret for local development, you can use the following on macOS or Linux:

```
openssl rand -base64 48
```

On Windows PowerShell, you can use:

```
[Convert]::ToBase64String((1..48 | ForEach-Object { Get-Random
-Maximum 256 }))
```

To keep your local configuration secure and avoid committing environment-specific values to the repository, follow these practices:

- Keep placeholder values in `appsettings.json`.
- Store real credentials in `appsettings.Development.json` or environment variables.
- Never commit real secrets.
- Run the app using either the CLI with `dotnet run` or an IDE of your choice.

After launching the application, open the URL shown in the console output and log in using any username. If a user with that username does not already exist, a new user is created. The first created user is automatically promoted to admin. This first-login behavior makes testing simpler during chapter walkthroughs because you can access admin pages without extra manual role setup.

Importing sample collections from the Data folder

Many chapter examples assume data is already present in MongoDB. If your database is empty, import the sample JSON files from the `Data` folder first.

The `Data` folder contains collection exports such as:

- `leafy_library.authors.json`
- `leafy_library.books.json`
- `leafy_library.issueDetails.json`
- `leafy_library.reviews.json`

You can import the sample collections using either the `mongoimport` command-line tool or MongoDB Compass, depending on your preferred workflow.

Option A: Importing with mongoimport (CLI)

Run the following commands from the root of the Leafy Library project:

- Local MongoDB database example:

```
mongoimport --uri "mongodb://localhost:27017" --db leafy_library
--collection authors --file "Data/leafy_library.authors.json"
--jsonArray
mongoimport --uri "mongodb://localhost:27017" --db leafy_library
--collection books --file "Data/leafy_library.books.json"
--jsonArray
mongoimport --uri "mongodb://localhost:27017" --db leafy_library
--collection issueDetails --file "Data/leafy_library.issueDetails.
json" --jsonArray
mongoimport --uri "mongodb://localhost:27017" --db leafy_library
--collection reviews --file "Data/leafy_library.reviews.json" –
jsonArray
```

  These commands will populate your local MongoDB instance (assuming it is running on the default port) with the collections and data required to have a functioning library with data.

- MongoDB Atlas example:

```
mongoimport --uri "mongodb+srv://<username>:<password>@<cluster>.
mongodb.net" --db leafy_library --collection authors --file "Data/
leafy_library.authors.json" --jsonArray
mongoimport --uri "mongodb+srv://<username>:<password>@<cluster>.
mongodb.net" --db leafy_library --collection books --file "Data/
leafy_library.books.json" --jsonArray
mongoimport --uri "mongodb+srv://<username>:<password>@<cluster>.
mongodb.net" --db leafy_library --collection issueDetails --file
"Data/leafy_library.issueDetails.json" --jsonArray
mongoimport --uri "mongodb+srv://<username>:<password>@<cluster>.
mongodb.net" --db leafy_library --collection reviews --file "Data/
leafy_library.reviews.json" –jsonArray
```

  The `--jsonArray` flag matters because the sample files are array-formatted JSON exports. Without `--jsonArray`, `mongoimport` will not parse them as expected.

Option B: Importing with MongoDB Compass

If you prefer a GUI-based workflow, use MongoDB Compass to import the sample collections by following these steps:

1. Open MongoDB Compass and connect to your MongoDB deployment.
2. Open or create the `leafy_library` database.
3. Create each target collection: `authors`, `books`, `issueDetails`, and `reviews`.
4. Use the **Import Data** option from the **Collection** menu for each collection, and select the matching JSON file from the `Data` folder.

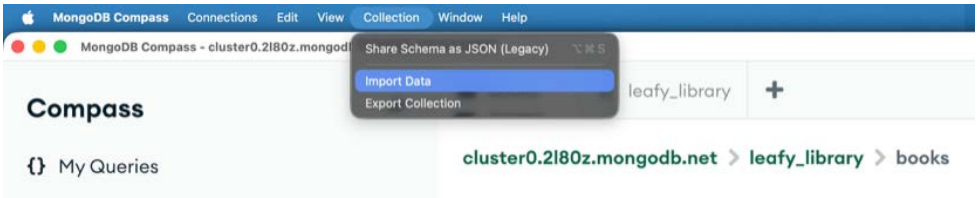


Figure 11.2 – Compass Collection menu with Import Data on macOS

5. Confirm JSON format and select the **Import** button to complete the import.

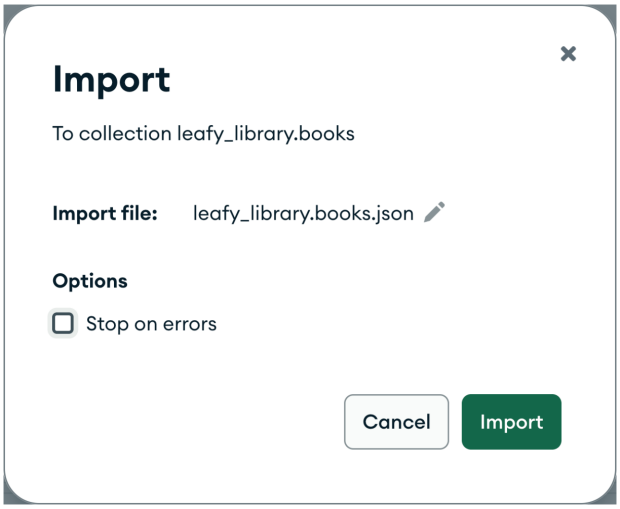


Figure 11.3 – Compass import dialog box for leafy_library.books.json

MongoDB Compass also lets you inspect the imported documents immediately.

6. After completing the import, run the application and confirm that you can browse the populated catalog.

How import and startup behavior work together

A common point of confusion is where data setup ends, and index setup begins. When starting with an empty MongoDB deployment, you are responsible for importing the data. After that, the application startup flow handles search and vector index checks, and generates embeddings for books that are missing embedding values.

Hence, the practical sequence is:

1. Import the base collections from JSON files in the `Data` folder.
2. Run the application.
3. Let the startup create any missing indexes and backfill missing embeddings on the relevant branches.

This sequencing keeps setup predictable and aligns with the implementation shown across chapters.

Troubleshooting common startup issues

If the application does not behave as expected, check the following:

- Verify that the connection string is valid and points to the intended MongoDB deployment.
- Verify that the `DatabaseName` matches the database name where you imported the data, typically `leafy_library`.
- Ensure that the JWT secret is configured and meets the minimum length requirement.
- Confirm that the collections were imported into the same database used by the application.
- Ensure that the embedding dimensions match both the model output and vector index dimensions.
- If you are working through the semantic retrieval chapters, verify that the embedding provider API key and endpoint are valid.

If the issue appears to be branch-related:

- Confirm your current branch before running the chapter code.
- Ensure that the chapter you are following matches the capabilities available in that branch.

Summary

This appendix is designed to remove setup ambiguity so you can focus on the learning outcomes in each chapter. When in doubt, return here to verify your branch and data state before continuing with the chapter.

Once the application is running with the sample data imported, you can more easily reproduce, understand, and extend the examples in your own projects.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

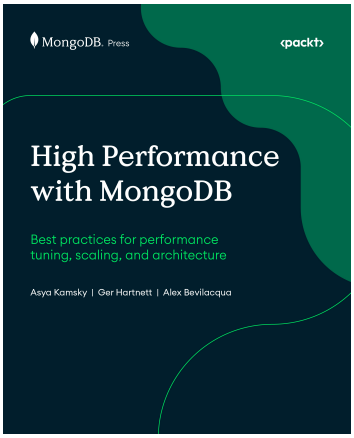
If you enjoyed this book, you may be interested in these other books by Packt:

Building AI Intensive Python Applications

Rachelle Palmer, Ben Perlmutter, Ashwin Gangadhar, Nicholas Larew, Sigfrido Narváez, Thomas Rueckstiess, Henry Weller, Richmond Alake, Shubham Ranjan

ISBN: 978-1-83620-725-2

- Understand the architecture and components of the generative AI stack
- Explore the role of vector databases in enhancing AI applications
- Master Python frameworks for AI development
- Implement Vector Search in AI applications
- Find out how to effectively evaluate LLM output
- Overcome common failures and challenges in AI development



High Performance with MongoDB

Asya Kamsky, Ger Hartnett, Alex Bevilacqua

ISBN: 978-1-83702-263-2

- Diagnose and resolve common performance bottlenecks in deployments
- Design schemas and indexes that maximize throughput and efficiency
- Tune the WiredTiger storage engine and manage system resources for peak performance
- Leverage sharding and replication to scale and ensure uptime
- Monitor, debug, and maintain deployments proactively to prevent issues
- Improve application responsiveness through client driver configuration

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Building Modern Data Applications with MongoDB and .NET*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.