

Three Design Patterns to Speed Up MEAN and MERN Stack Applications

Effective techniques when using Redis for performance improvement



Table of Contents

Introduction	3
Our demo setup.	4
The master data-lookup pattern	5
Use Cases.	6
Demo	7
Code	8
The cache-aside pattern	11
Use Cases.	12
Demo	13
Code	15
The write-behind pattern	17
Use Cases.	18
Demo	19
Code	20
Consider using Redis as your primary database	26

Together, the MongoDB database, the Express and Angular.js frameworks, and Node.js constitute the **MEAN stack**, a pure JavaScript stack that helps developers create every part of a website or application. Similarly, the **MERN stack** is composed of MongoDB, the Express and ReactJS frameworks, and Node.js. Both stacks have become popular ways to build Node.js applications.

Those stacks work well, which accounts for their popularity. But it doesn't mean the software generated runs as fast as it can – or as fast as it needs to. An application needs to be designed with speed in mind, just as it needs security “baked in.” If you don't design and build software with your attention on performance, those applications can encounter significant bottlenecks when they go to production.

There isn't one magic pill to make software run faster. There never is. But over time, the development community has learned common techniques that work as reliable patterns to solve well understood problems, and application performance certainly is among them. To prevent and resolve performance issues, technical architects and software developers often use Redis – and they do so in different ways depending on the situation.

In this e-book, we share three popular design patterns that developers use with Redis to improve application performance with MEAN and MERN stack applications. We explain each pattern in detail, and accompany it with an overview, typical use cases, and a code example. Our intent is to help you understand when and how to use each pattern.

New to design patterns?

Design patterns are recommended practices to solve recurring design problems in software systems. A design pattern has four parts: a name, a problem description (a particular set of conditions to which the pattern applies), a solution (the best general strategy for resolving the problem), and a set of consequences.

The first popular book about software design patterns was [Design Patterns: Elements of Reusable Object-Oriented Software](#) by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1994). It's still an authoritative resource.

Our demo setup

This e-book uses a **GitHub sample demo that was built using the following tools:**

Frontend	ReactJS (18.2.0)
Backend	Node.js (16.17.0)
Database	MongoDB
Cache and database	Redis stack (using Docker)

Want to learn more?
Consult these references:

- ▶ [Redis stack Docker](#)
- ▶ [MongoDB Atlas](#)
- ▶ [Caching at Scale with Redis](#)

The demonstration application (as shown in image 1) showcases a movie application that uses basic create, read, update, and delete (CRUD) operations.

The movie application dashboard contains a search section at the top and a list of movie cards in the middle. The floating plus icon displays a pop-up when the user

selects it, permitting the user to enter new movie details. The search section has a text search bar and a toggle link between text search and basic (that is, form-based) search. Each movie card has edit and delete icons, which are displayed when a mouse hovers over the card.

You can follow along with the details using the code on GitHub.

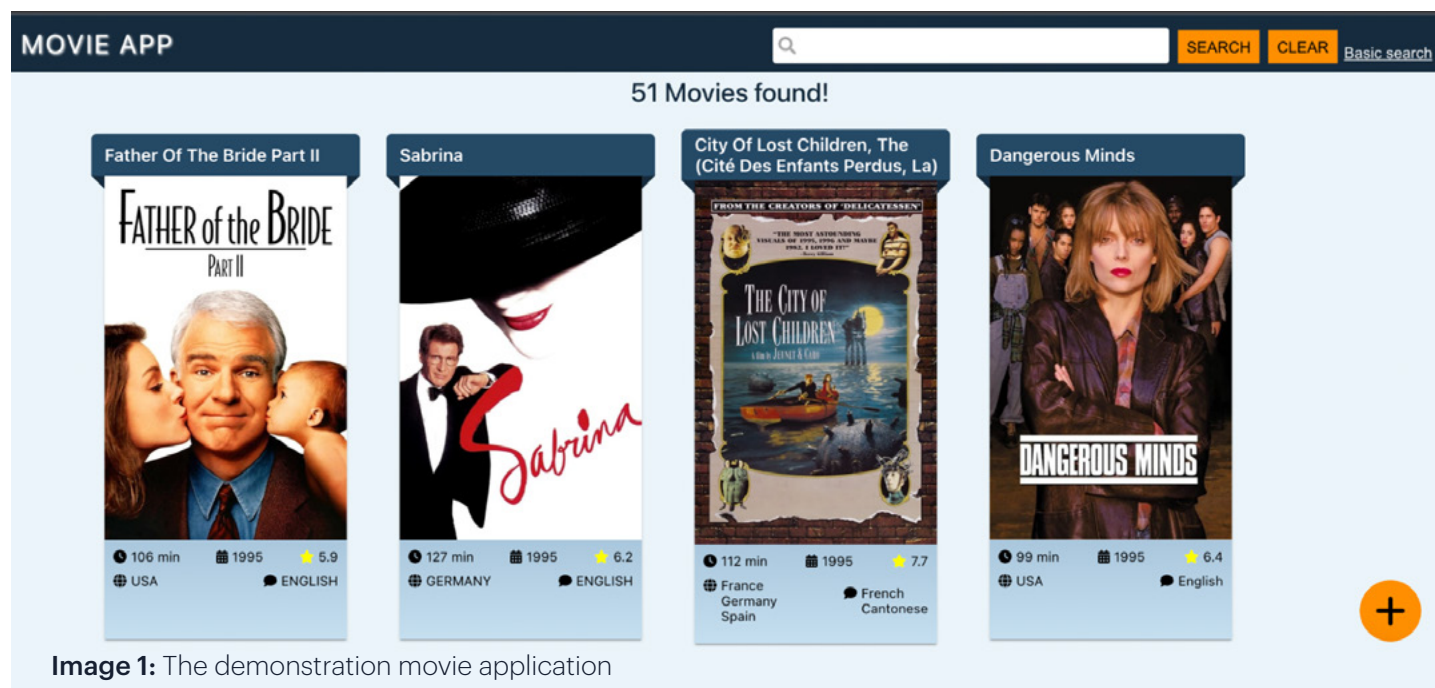


Image 1: The demonstration movie application

Source code for the application used to showcase each pattern

<https://github.com/redis-developer/ebook-speed-mern-frontend.git>
<https://github.com/redis-developer/ebook-speed-mern-backend.git>



The master data-lookup pattern

One ongoing developer challenge is to (swiftly) create, read, update, and (possibly) delete data that lives long, changes infrequently, and is regularly referenced by other data, directly or indirectly. That's a working definition of **master data**, especially when it also represents the organization's core data that is considered essential for its operations.

Master data generally changes infrequently – country lists, genre, and movie languages usually stay the same. That presents an opportunity to speed things up. You can address access and manipulation operations so that data consistency is preserved and data access happens quickly.

From a developer's point of view, **master data lookup** refers to the process by which master data is accessed in

business transactions, in application setup, and any other way that software retrieves the information. Examples of master data lookup include fetching data for user interface (UI) elements (such as drop-down dialogs, select values, multi-language labels), fetching constants, user access control, theme, and other product configuration. And you can do that even when you rely primarily on MongoDB as a persistent data store.

To see how the pattern functions, refer to image 2.

1. Read the master data from MongoDB. Store a copy of the data in Redis upfront. This pre-caches the data for fast retrieval. Use a script or a cron job to repeatedly copy master data to Redis.
2. The application requests Master data.
3. Instead of MongoDB serving the data, the master data will be served from Redis.

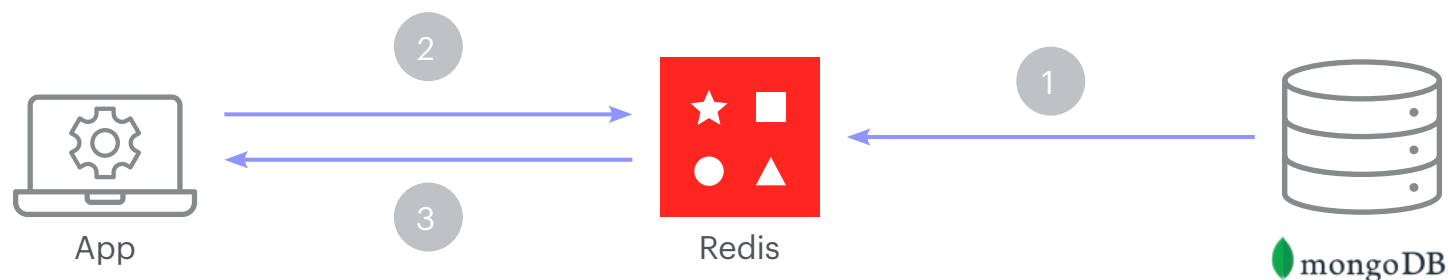


Image 2: To serve master data from Redis, preload the data from MongoDB.

Use Cases

Consider this pattern when you need to...

- ▶ **Serve master data at speed:**

By definition, nearly every application requires access to master data. Pre-caching master data with Redis delivers it to users at high speed.

- ▶ **Support massive master tables:**

Master tables often have millions of records. Searching through them can cause performance bottlenecks. Use Redis to perform real-time search on the master data to increase performance with sub-millisecond response.

- ▶ **Postpone expensive hardware and software investments:**

Defer costly infrastructure enhancements by using Redis. Get the performance and scaling benefits without asking the CFO to write a check.

Demo

Image 3 focuses on a standard way to showcase a UI that is suitable for master data lookups. The developer responsible for this application would treat certain fields as master data, including movie language, country, genre, and ratings, because they are required for common application transactions.

Consider the pop-up dialog that appears when a user who wants to add a new movie clicks the movie application plus icon (image 3). The pop-up includes drop-down menus for both country and language. In this demonstration, Redis loads the values.

Image 3: Pop-up screen to add a new movie

The image shows a 'Movie Detail' pop-up form with a dark blue header and a close button (X) in the top right corner. The form contains several input fields and dropdown menus:

- Title***: A text input field.
- Movie (IMDB) URL***: A text input field.
- Poster (Image) URL***: A text input field.
- Country***: A dropdown menu with a 'Select' option. A blue arrow points to it from a list of countries.
- Language***: A dropdown menu with a 'Select' option. A blue arrow points to it from a list of languages.
- Plot***: A text input field.
- Duration(minutes)***: A text input field with the value '0'.
- Released Date***: A text input field with the format 'dd/mm/yyyy'.
- Rating***: A text input field with the value '0'.

At the bottom right of the form are two buttons: 'SAVE' and 'CANCEL'.

Two dropdown lists are shown on the sides of the form, representing the data loaded from Redis:

- Country List** (left):
 - ✓ Select
 - Australia
 - Canada
 - China
 - France
 - Germany
 - Italy
 - SouthAfrica
- Language List** (right):
 - ✓ Select
 - Algonquin
 - Cantonese
 - Dutch
 - English
 - French
 - German
 - Hungarian

Code

The code snippets in code blocks 1 and 2 display a fetch query of master data from both MongoDB and Redis to load the country and language into the dropdown values.

Previously, if the application used MongoDB, it searched the static database to retrieve the movie's country and language values. That can be time consuming if it's read from persistent storage – and is inefficient if the information is static.

Instead, the “after” views in code blocks 1 and 2 show that the master data can be accessed with only a few lines of code – and much faster response times.

Code block 1. Query to fetch masters from MongoDB and Redis

```
```js
*** BEFORE (MongoDB)***
*** MongoDB regular search query ***
function getMasterCategories() {
 ...
 db.collection("masterCategories").find({
 statusCode: {
 $gt: 0,
 },
 category: {
 $in: ["COUNTRY", "LANGUAGE"],
 },
 });
 ...
}
```
```

```
```js
*** AFTER (Redis) ***
*** Redis OM Node query ***
function getMasterCategories() {
 ...
 masterCategoriesRepository
 .search()
 .where("statusCode")
 .gt(0)
 .and("categoryTag")
 .containOneOf("COUNTRY", "LANGUAGE");
 ...
}
```
```


Want to learn more?
Consult these references:

- ▶ [Redis om Node](#)
- ▶ [Atlas compound query](#)

Code block 2. Query to fetch masters from MongoDB Atlas and Redis

```
```js
*** BEFORE (MongoDB)***
*** Mongodb atlas search query (used in the
demo) ***
function getMasterCategories() {
 ...
 db.collection(_collectionName).
 aggregate([{
 $search: {
 index: "index_master_
categories",
 compound: {
 must: [{
 range: {
 gt: 0,
 path: "statusCode",
 },
 }, {
 text: {
 query: ["COUNTRY",
"LANGUAGE"],
 path: "category",
 },
 },
]
 }
 }
]));
 ...
}
```
```

```
```js
*** AFTER (Redis) ***
*** Redis OM Node query ***
function getMasterCategories() {
 ...
 // "Redis OM Node query";
 masterCategoriesRepository
 .search()
 .where("statusCode")
 .gt(0)
 .and("categoryTag")
 .containOneOf("COUNTRY", "LANGUAGE");
 ...
}
```
```

Testimonial



Search and query
4-5x faster than
ElasticSearch



10M spare parts
search, display,
less than ~25-30ms

GoMechanic needed a fast, accurate search for its spare parts catalog, which has more than 10 million items. The network of car service centers chose Redis to help reduce the time it took for applications to search through its master data.

With Redis, GoMechanic greatly improved customer experience by reducing the time it takes to search through master data.

**“GoMechanic needed a fast data layer to search across
10 million spare parts and power our customer service charts.”**

- Prasenjit Singh, Vice President of Engineering, GoMechanic



Company Profile

GoMechanic operates thousands of car service centers across India

Business Challenges

- Retail customer search of over 10M parts became too slow
- Customer chat grew very fast and growing service team needed access to chat history

Technical Challenges

- Performance of ElasticSearch too slow as parts list grew
- Searching of chat histories using JavaScript became slow and cumbersome

Benefits

- Enhanced customer experience through real time search
 - Customer response improved by 4-5x, eliminating master table bottlenecks
 - Faceted and auto-complete search using fuzzy matching in less than 30ms
 - Chat history now searchable in real time by Customer Service
- Increase ROI with Redisearch
 - 4x the speed, half the cost of ElasticSearch
 - Cost effective to scale globally

The cache-aside pattern

The cache-aside pattern offers guidance for how to load data on demand into a cache from a data store to improve query performance.

The goal of this design pattern is to set up optimal caching, read operations, and data retrieval, with variations that depend on whether you preload data or load it as you go. That requires a significant “what if” decision tree:

- Is there a cache hit? Initially, the application attempts to read data from the cache. If it finds the data and returns the data; the process stops here.
- Is there a cache miss? When the data is not found in the cache, Redis reads the data from the MongoDB database. Then Redis stores the data to the cache so that it is ready for future requests.

To see how the pattern functions, refer to image 4.

1. An application requests data from the backend
2. The backend checks to find out if the data is available in Redis

A. If the data isn't available (a cache miss):

- The Data is fetched from MongoDB database
- Redis stores a copy of the data
- The backend serves the data to the application

B. If the data is already available in Redis (a cache hit):

- The backend serves the cached data directly to the application

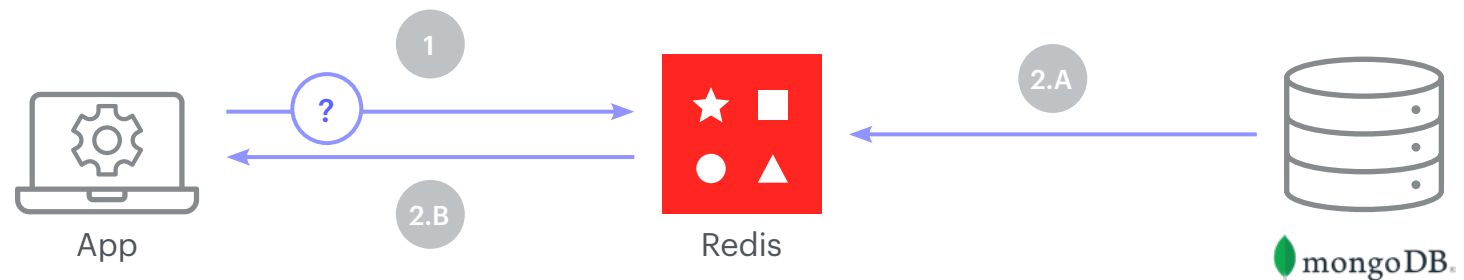


Image 4: Serve data from MongoDB first, then subsequently from Redis

Use Cases

Consider this pattern when you need to...

- ▶ **Query data frequently:**

The cache-aside pattern is helpful when an application regularly reads data from a database – and especially when it accesses the same field. This is a good option when you need immediate performance gain for subsequent data requests.

- ▶ **Fill cache on demand:**

Use this pattern to fill the cache by only the data that the application has actually demanded. This pattern is useful when the cache size needs to be cost-effective and the kind of data to be cached is unclear.

- ▶ **Be cost conscious:**

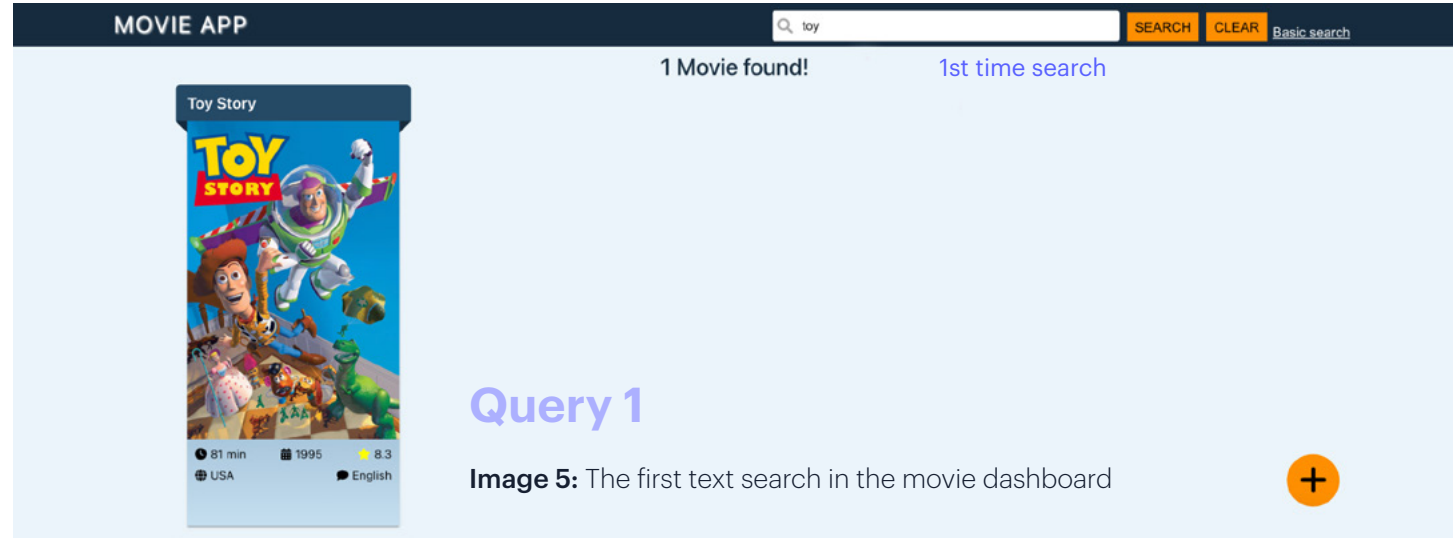
Since cache size is directly related to the cost of cache in the cloud, the smaller the size, the less you pay.

Demo

In the demonstration application's movie dashboard, the developer's assumption is that the user regularly accesses the same records (that is, movie data). The end user types text into the search header to locate a movie. The first time the user performs the search, the results come from the MongoDB database. Redis retrieves the same data for subsequent similar searches – and it happens much faster.

Try this out for yourself.

Perform a text search for “toy” to see the results (*Toy Story*) from the MongoDB database (image 5).



Query 1

Image 5: The first text search in the movie dashboard

Repeat the search: Search for “toy” again to see the results appear from the cache (image 6). It's much faster, isn't it?

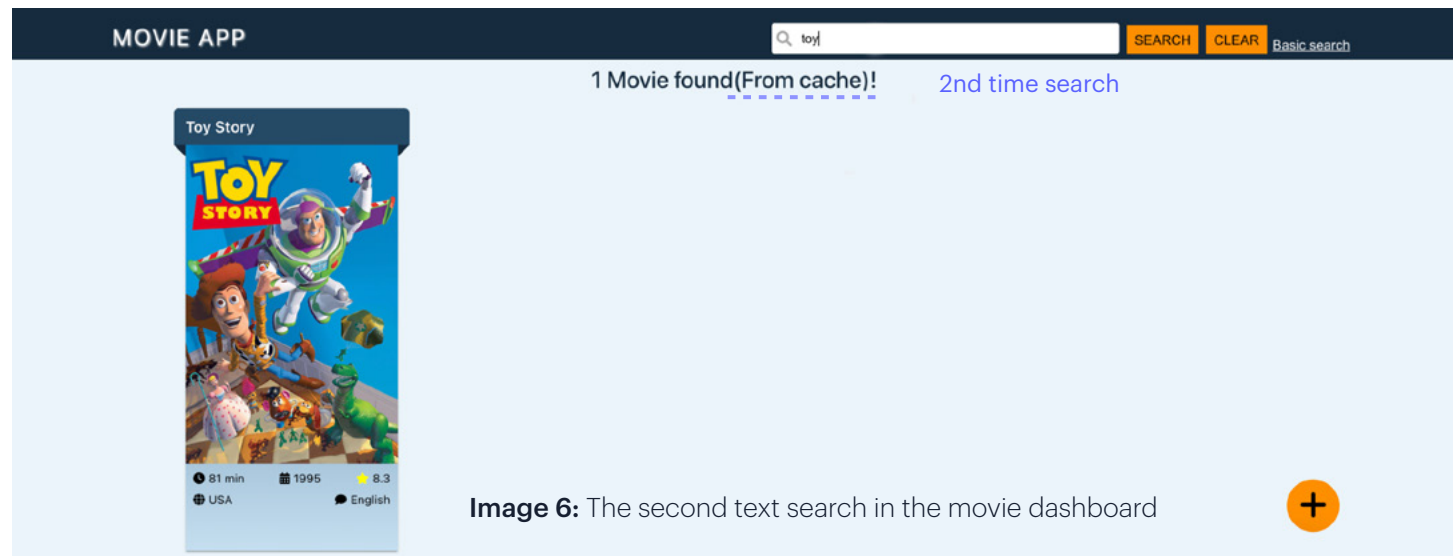


Image 6: The second text search in the movie dashboard

Repeat the exercise to confirm that this works on additional searches.

- Perform a basic form search to see the results from the database (image 7).

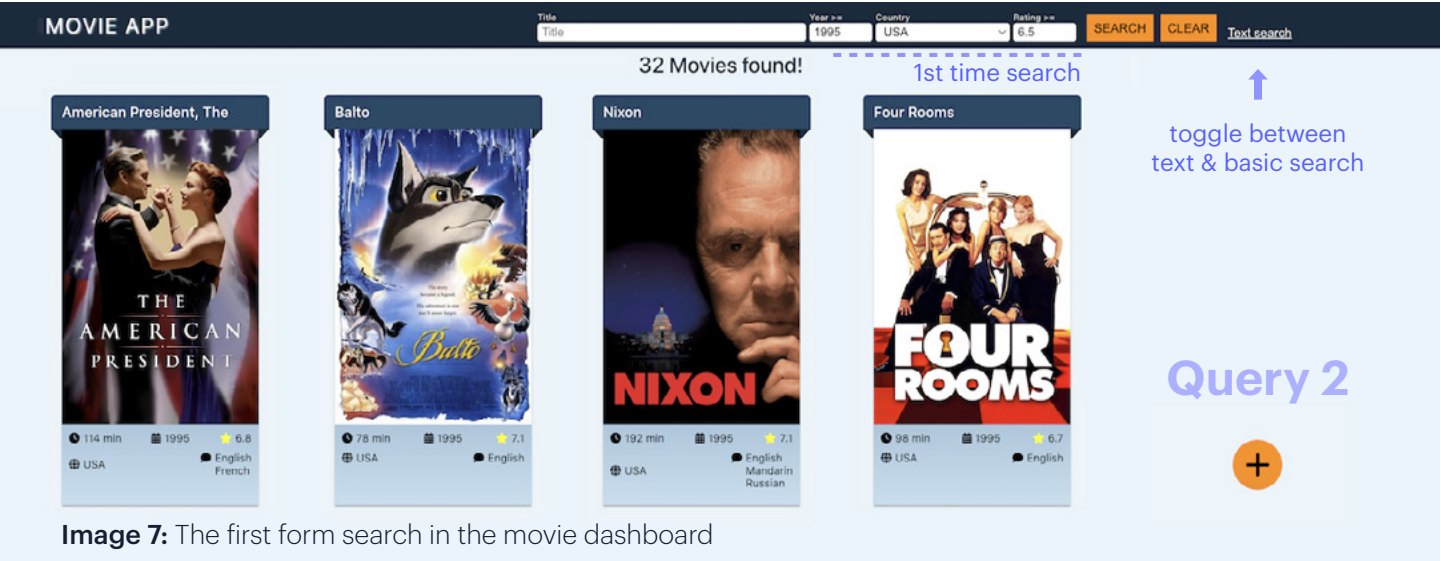


Image 7: The first form search in the movie dashboard

- Perform the same basic form search again to see the speed difference in the cached results (image 8).

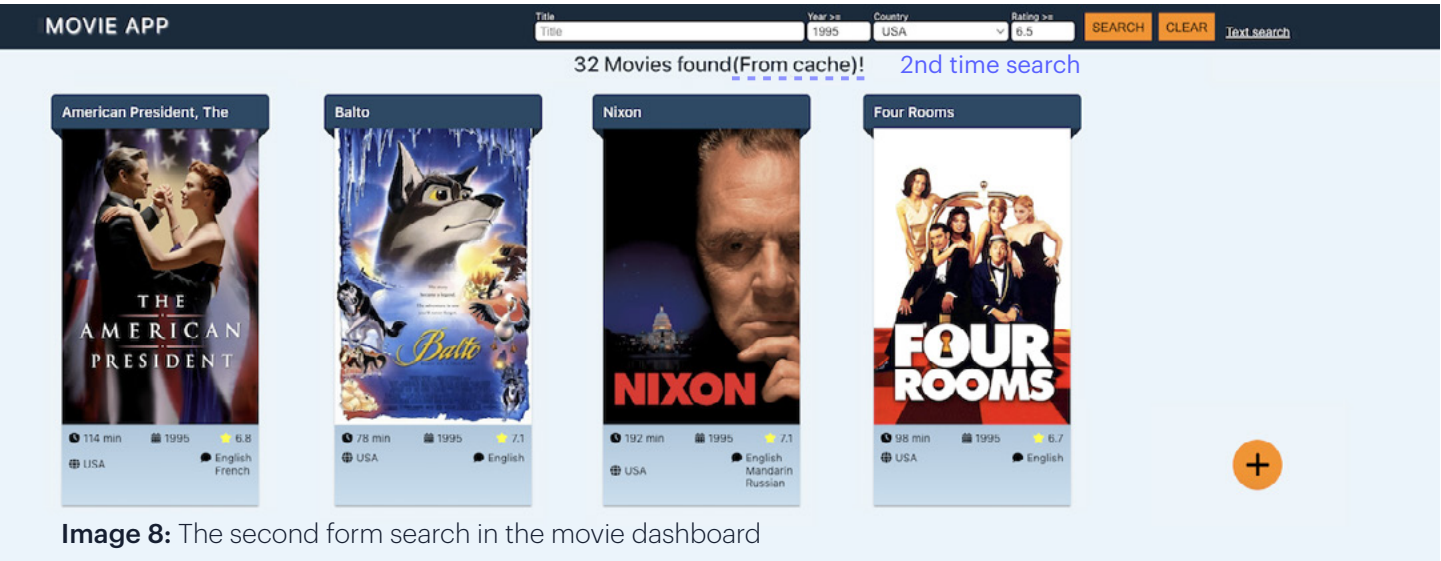


Image 8: The second form search in the movie dashboard

Code

This pseudo code illustrates how a developer might design the application to use the cache-aside pattern (code block 3).

In MongoDB, the application executes a simple call to the database.

Using Redis, the application first checks the contents of the Redis cache, and then serves it up if it finds the data. (“Toy Story, coming right up!”)

Code block 3. Pseudo code for cache-aside pattern

```
```js
*** BEFORE (MongoDB) ***
async function getMoviesByFilters(_
searchFilter) {
 ...
 const moviesList = await
getMoviesFromDb(_searchFilter);
 ...
 return moviesList;
}
```
```

```
```js
*** AFTER (Redis)***
async function getMoviesByFilters(_
searchFilter) {
 ...
 const moviesList = null;
 const cacheData = await
getDataFromRedis(_searchFilter);
 if (cacheData) {
 moviesList = cacheData;
 } else {
 moviesList = await getMoviesFromDb(_
searchFilter);
 setCacheDataToRedis(searchFilter,
moviesList); //async
 }
 ...
 return moviesList;
}
```
```

Here are a few points to consider as you design your own applications.

- **Expiry time of cached data:**

How long do you expect to access this data on a frequent basis? Define the expiry time for the cached data based on your product requirements. You need to balance performance needs with data accuracy. Expiry time cannot be too short (requiring frequent re-caching, negating performance benefits) or too long (causing cached data to become irrelevant compared with database data).

- **Data consistency:**

Pay attention to the possibility of data mismatch between the database and cache during the lifetime of the cached data. For continuous data consistency, choose other patterns, such as write-behind or write-through.

The application needs to explicitly get data from the cache, and to save (or set) data to the cache (code block 4). (For simplicity, this example uses the sha1 algorithm to create a unique key for reasons of clarity; it's faster than sha256 and the example does not require data encryption.)

Code block 4. Cache-aside pattern code to save/set data

```
```js
*** AFTER (Redis) - set data to cache ***

async function setCacheDataToRedis(_filter,
_data) {
 ...
 _filter = JSON.stringify(_filter);

 //create key from search filter
 const keyHash = crypto.createHash("sha1")
 .update(_filter)
 .digest("hex");

 const valueStr = JSON.stringify(_data);
 const expirySec = 30;

 //store search result against that key
 await redisClient.set(keyHash, valueStr,
 {
 EX: expirySec,
 });
 ...
}
```

```
```js
*** AFTER (Redis) - get data from cache ***

async function getDataFromRedis(_filter) {
  ...
  let value = "";
  _filter = JSON.stringify(_filter);

  //create key from search filter
  const keyHash = crypto.createHash("sha1")
    .update(_filter)
    .digest("hex");

  //get cached search result against that
  key
  const valueStr = await redisClient.
  get(keyHash);
  if(valueStr){
    value = JSON.parse(valueStr);
  }
  ...
  return value;
}
```


The write-behind pattern

Here are two related caching techniques that can speed up applications.

1. With the write-through pattern, every time an application writes data to the cache, it also updates the records in the database.
2. Write-behind is similar to write-through caching, except that the cached data is asynchronously updated with the persistent database.

They both treat the cache as the system of record. However, the timing of the write to the system of record is slightly different. With the write-through pattern, the

thread waits until the write to the database is complete. In contrast, the write-behind pattern adds the writing of the data to the system of record to a queue. That permits the application thread to move on more quickly (and to let the cache catch up when it can), but it does mean that there is a short time when the data between the cache and the system of record is inconsistent.

The pattern works as presented in image 9.

1. The application reads and writes data to Redis.
2. Next, Redis syncs any changed data to the MongoDB database asynchronously.

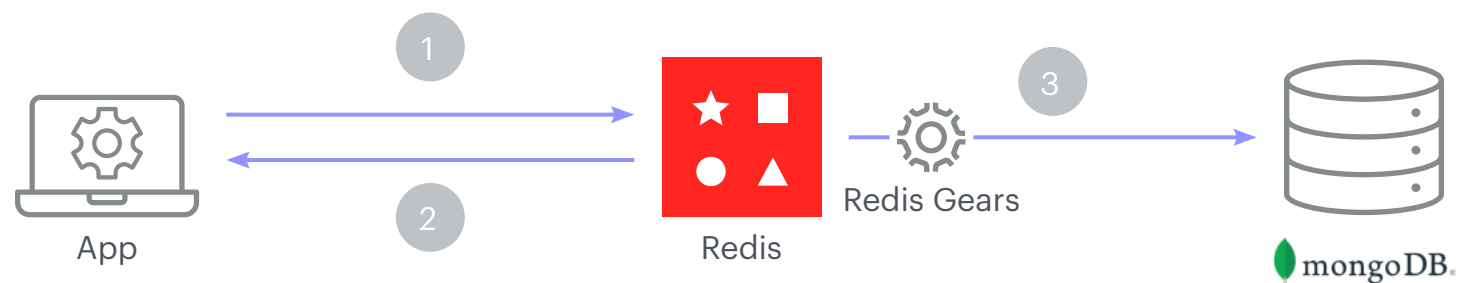


Image 9: Write data to Redis, then RedisGears syncs the data to MongoDB

Use Cases

Consider this pattern when you need to...

- ▶ **Flatten peaks in demand:**

Under stress, an application may need to write data quickly. If your application needs to perform a large number of write operations at high speed, consider this pattern. The Redis and RedisGears modules make sure the data stored in the cache is synced with the database.

- ▶ **Batch multiple writes:**

Sometimes it's expensive to write to a database frequently (for example, logging). In those cases, it can be cost-effective to batch the database writes so that data syncs at intervals.

- ▶ **Offload the primary database:**

Using this pattern can reduce database load when heavy writes operate on Redis by spreading writes out to flatten the peak time of application usage.

Demo

To demonstrate this pattern using the movie application, imagine that the user opens the pop-up to add a new movie (image 10). Instead of the application immediately storing the data in MongoDB, the application writes the changes to Redis. In the background, RedisGears automatically synchronizes the data with the MongoDB database.

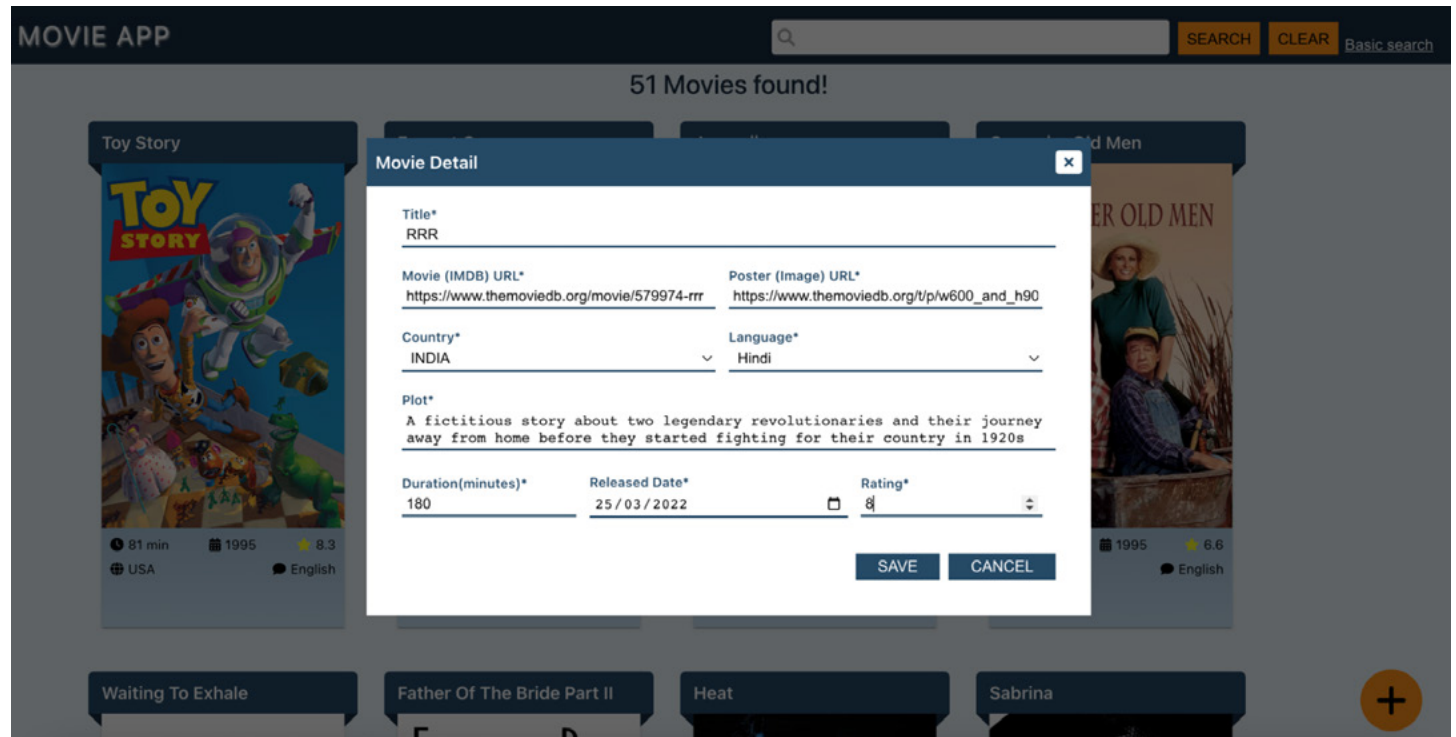


Image 10: Add a new movie

Code

Developers need to load some Python code to the Redis server before using the write-behind pattern (which syncs data from Redis to MongoDB). The Redis server has a RedisGears module that interprets the Python code and syncs the data from Redis to MongoDB.

Loading the Python code is easier than it sounds. Simply replace database details in the Python file and then load the file to the Redis server. Create the Python file (shown in code block 5, and [available online](#)). Then update the MongoDB connection details, database, collection, and primary key name to sync.

Code block 5. Python code for RedisGears to sync Redis data to MongoDB

```
```python
movies-write-behind.py

Gears Recipe for a single write behind

import redis gears & mongo db libs
from rgsync import RGJSONWriteBehind, RGJSONWriteThrough
from rgsync.Connectors import MongoConnector, MongoConnection

change mongodb connection (admin)
mongodb://usrAdmin:passwordAdmin@10.10.20.2:27017/dbSpeedMernDemo?authSource=admin
mongoUrl = 'mongodb://usrAdmin:passwordAdmin@10.10.20.2:27017/admin'

MongoConnection(user, password, host, authSource?, fullConnectionUrl?)
connection = MongoConnection('', '', '', '', mongoUrl)

change MongoDB database
db = 'dbSpeedMernDemo'
```

```
change MongoDB collection & it's primary key
movieConnector = MongoConnector(connection, db, 'movies', 'movieId')

change redis keys with prefix that must be synced with mongodb collection
RGJSONWriteBehind(GB, keysPrefix='MovieEntity',
 connector=movieConnector, name='MoviesWriteBehind',
 version='99.99.99')
...

```

There are two ways to load that Python file into the Redis server: using the Gears command-line interface (CLI) or RG.PYEXECUTE.

- ▶ Find more information about the Gears CLI (code block 6) at [gears-cli](#) and [rgsync](#).

**Code block 6.** Load the Python file to Redis via the Gears command-line interface

```
```sh
# If python file is located at "/users/tom/movies-write-behind.py"
gears-cli --host <redisHost> --port <redisPort> --password <redisPassword> run /users/tom/
movies-write-behind.py REQUIREMENTS rgsync pymongo==3.12.0
```

```

- ▶ Find more information at [RG.PYEXECUTE](#).

Execute RG.PYEXECUTE from the Redis command line (code block 7).

**Code block 7.** Load the Python code to Redis using the RG.PYEXECUTE command

```
```sh
# Via redis cli
RG.PYEXECUTE 'pythonCode' REQUIREMENTS rgsync pymongo==3.12.0
```
```

- ▶ Find more examples at [Redis Gears sync with MongoDB](#).

The RG.PYEXECUTE command can also be executed from the Node.js code like this. (Consult [the sample Node file](#) for more details.)

**Code block 8.** The Node code runs the RG.PYEXECUTE command

```
```js
*** same command via node js code ***
import * as fs from "fs";
import { createClient } from "redis";
import { fileURLToPath } from "url";
import { dirname } from "path";

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
const redisConnectionUrl = "redis://127.0.0.1:6379";
const pythonFilePath = __dirname + "/movies-write-behind.py";
```

```

const runWriteBehindRecipe = async () => {
  const requirements = ["rgsync", "pymongo==3.12.0"];
  const writeBehindCode = fs.readFileSync(pythonFilePath).toString();
  const client = createClient({ url: redisConnectionUrl });
  await client.connect();

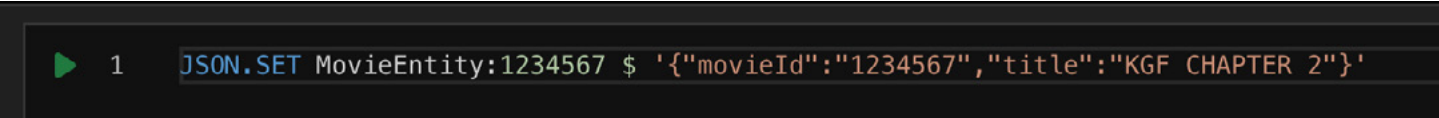
  const params = ["RG.PYEXECUTE", writeBehindCode,
    "REQUIREMENTS", ...requirements];

  try {
    await client.sendCommand(params);
    console.log("RedisGears WriteBehind set up completed.");
  }
  catch (err) {
    console.error("RedisGears WriteBehind setup failed !");
    console.error(JSON.stringify(err, Object.getOwnPropertyNames(err), 4));
  }
  process.exit();
};
runWriteBehindRecipe();
`

```

The next step is to verify that RedisGears is syncing data between Redis and MongoDB.

Insert a key starting with the prefix (that's specified in the Python file) using the Redis CLI (image 11).


 A screenshot of a terminal window showing a Redis CLI command. The command is: `1 JSON.SET MovieEntity:1234567 $ '{"movieId":"1234567","title":"KGF CHAPTER 2"}'`. The command is highlighted in blue and orange.


```

1 JSON.SET MovieEntity:1234567 $ '{"movieId":"1234567","title":"KGF CHAPTER 2"}'

```

Image 11: Inserting JSON to Redis via CLI/RedisInsights

Next, confirm that the JSON is inserted in MongoDB too (image 12).

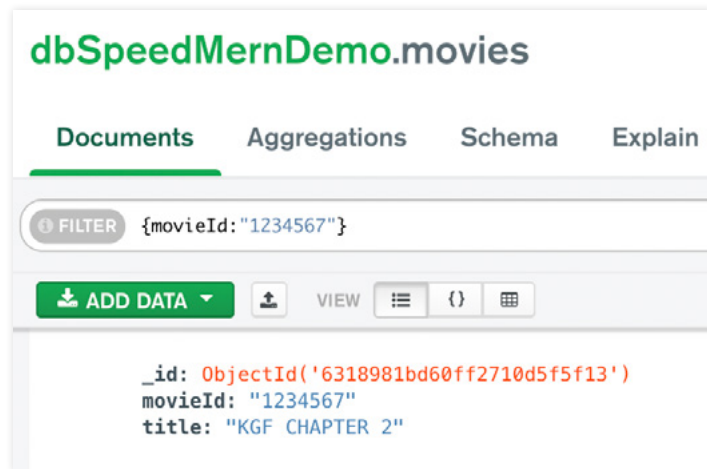


Image 12: Verifying the JSON via MongoDB compass

You also can check [Redis Insight](#) (image 13) to verify that the data is piped in via stream for its consumers (like RedisGears)

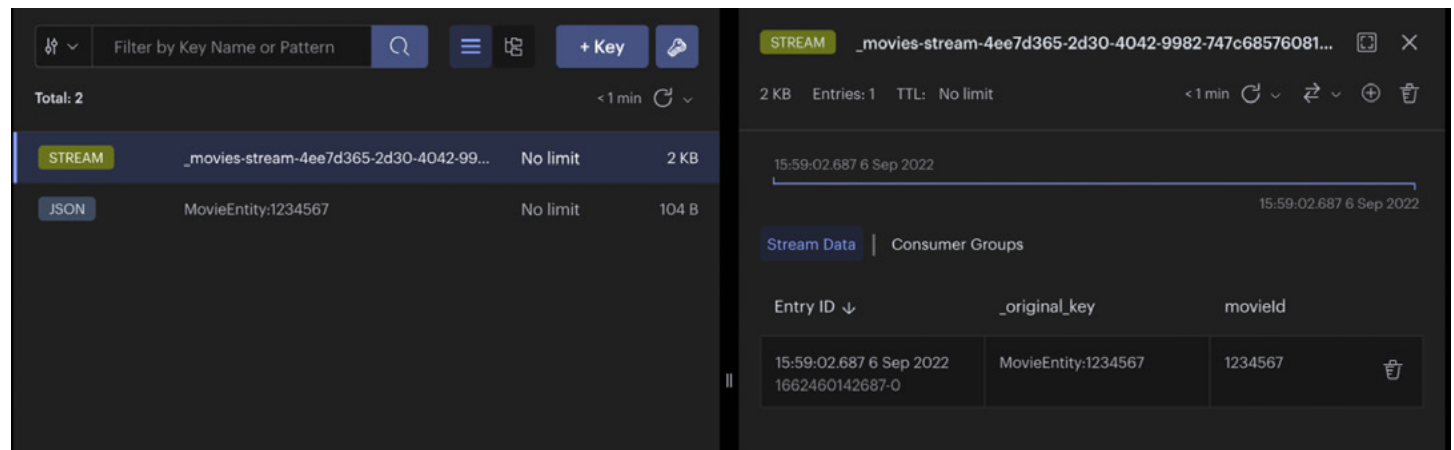


Image 13: Verifying Streams entry in Redis Insight

How does all that work with the demo application? Consider using snippets to insert a movie (code block 9). Once data is written to Redis, Redis Gears automatically synchronizes it to MongoDB.

Code block 9. Insert JSON to MongoDB directly using Redis

```
```js
*** BEFORE (MongoDB) ***
...
// (Node mongo query)
if (movie) {
 // insert movie to MongoDB
 await db.collection("movies")
 .insertOne(movie);
}
...
```
```

```
```js
*** AFTER (Redis) ***
...
// (Redis OM Node query)
if (movie) {
 const entity = repository.
 createEntity(movie);
 // insert movie to Redis
 await moviesRepository.save(entity);
}
...
```
```

Consider using Redis as your primary database

Why stick with MongoDB? You can use Redis Enterprise as a multi-model primary database. Redis Enterprise is a fully managed, highly available, secure, and real-time data platform. It can store data on both RAM or Flash. It also supports Active-Active (multi-zone read and write replicas) on different cloud vendors, providing extreme high availability and scalability. Active-Active offers global scalability while maintaining local speed for database reads and writes.

Using Redis Enterprise can simplify your software architecture, too. You avoid system complexity, management overhead, and application costs.

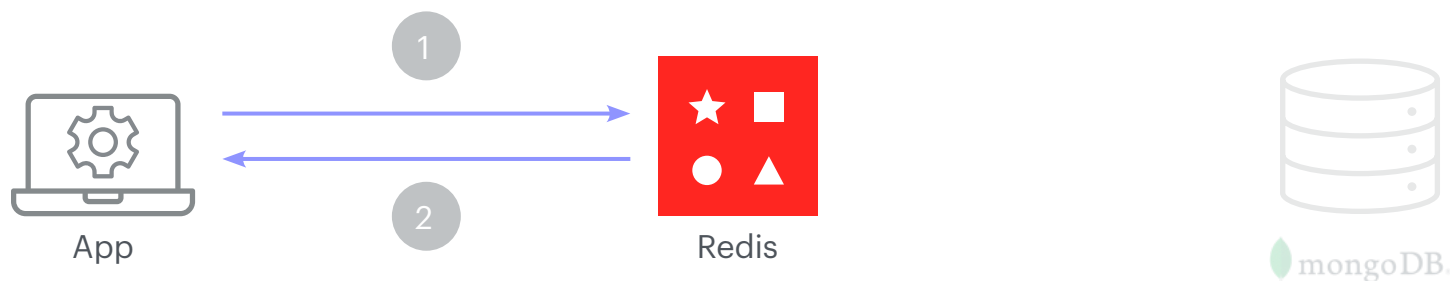


Image 14: Read/write data on Redis (with persistence) instead of MongoDB

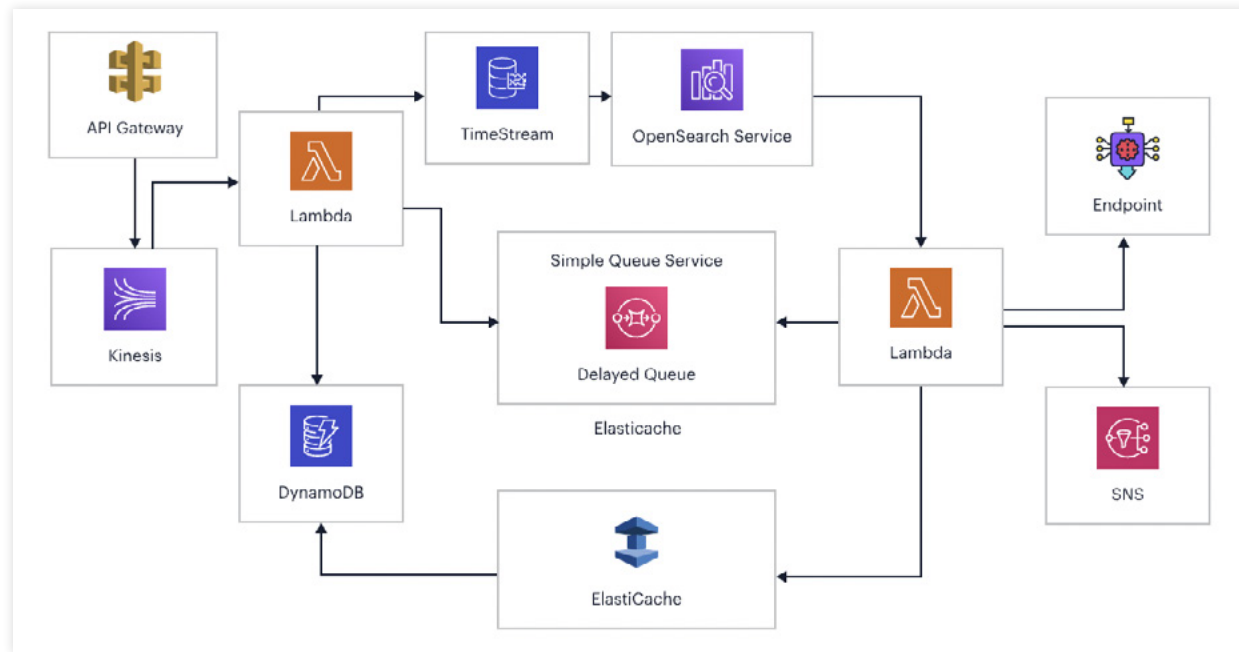


Image 15: Application architecture using multiple components

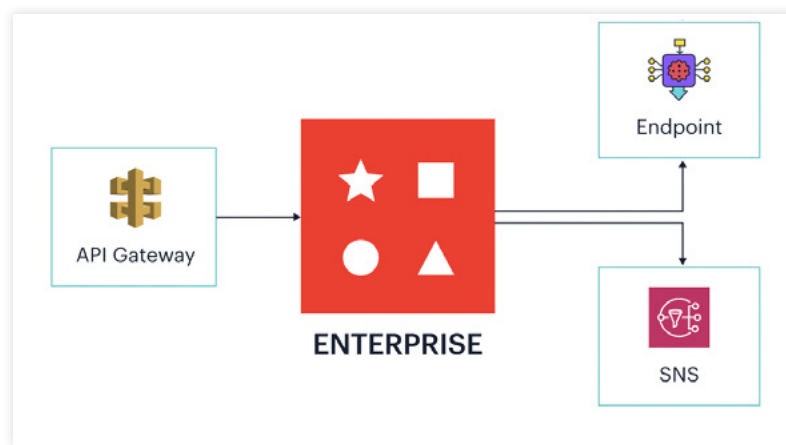


Image 16: Application architecture using Redis

Redis Enterprise has many built-in modular capabilities (image 17), making it a unified, real-time data platform. Redis Enterprise is far more than a document database.

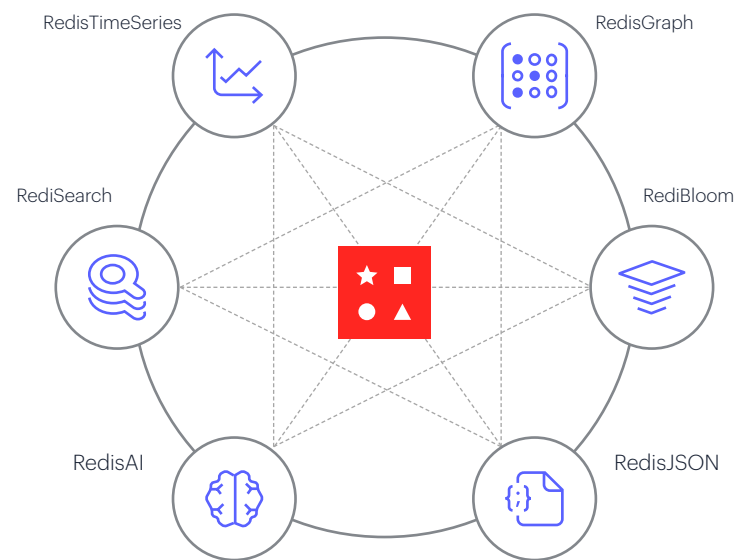
- **RedisJSON:** Persists JSON documents
- **RedisSearch:** Indexes and searches JSON documents
- **RedisGraph:** A fast and easy-to-use graph database
- **RedisBloom:** Provides bloom filters and other probabilistic data structures
- **RedisTimeSeries:** Supports time series data structures
- **RedisGears:** Syncs data to external databases via different patterns (write-behind/ write-through) or executes custom logic.

There's a lot more functionality to explore. Use [RedisInsight](#) to view your Redis data or to play with raw Redis commands in the workbench. Clients like [Node Redis](#) and [Redis on Node](#) help you to use Redis in Node.js applications.

It's easy for you to explore the features, because we make it easy to try... for free. We encourage you to experiment and learn how useful Redis can be.

Try Free →

Image 17: Redis capabilities



About Redis

Data is the lifeline of every business, and Redis helps organizations reimagine how fast they can process, analyze, make predictions, and take action on the data they generate. Redis provides a competitive edge to any business by delivering [open source](#) and [enterprise-grade](#) data platforms to power applications that drive real-time experiences at any scale. Developers rely on Redis to build performance, scalability, reliability, and security into their applications.

Born in the cloud-native era, Redis uniquely enables users to unify data across multi-cloud, hybrid, and global applications to maximize business potential. Learn how Redis can give you this edge at redis.com.