

[Open in app](#)

★ Member-only story

# Elastic Search — Day 4: Advanced Querying!!



Navya Cloudops · Following

7 min read · Feb 10, 2024



Listen

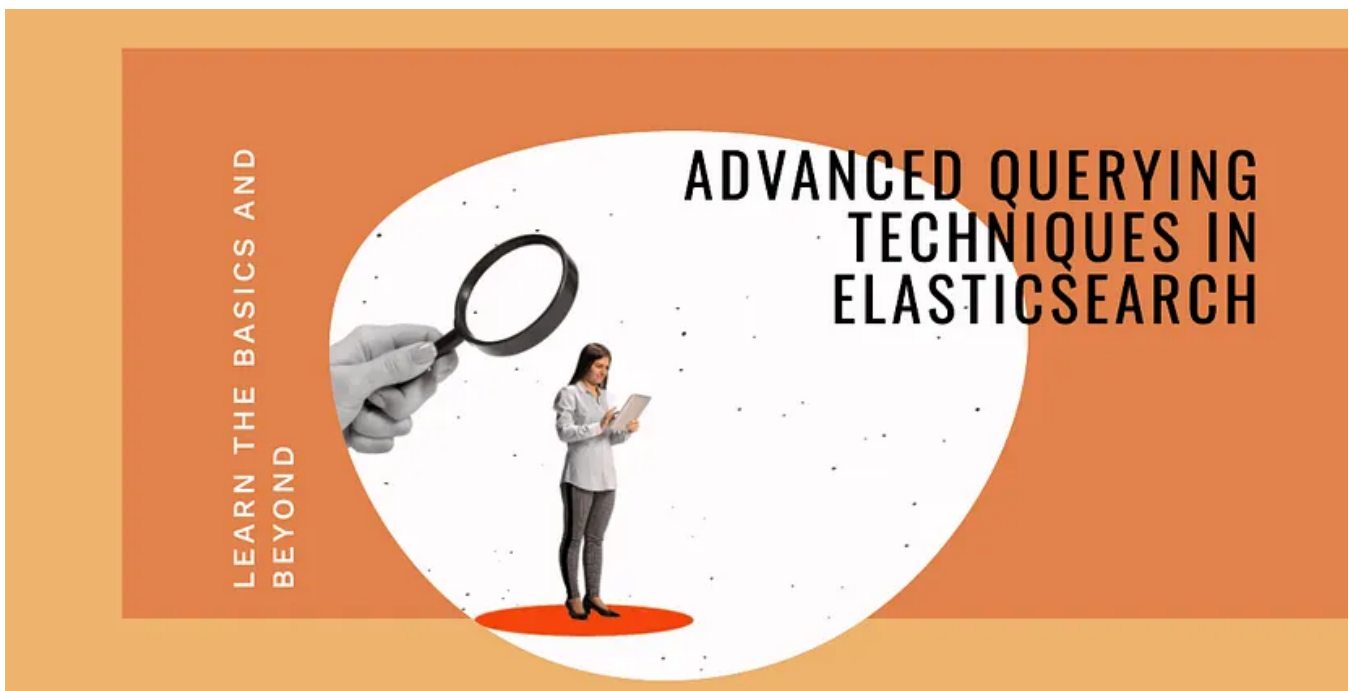


Share



More

Welcome to Day 4 of our 10-day DevOps Elasticsearch course! Today, we're diving deep into Elasticsearch's Advanced Querying capabilities. Elasticsearch's Query DSL (Domain Specific Language) provides powerful tools for crafting complex queries to retrieve exactly the data you need from your Elasticsearch indices. In this session, we'll explore compound queries, full-text search, fuzzy matching, pagination, and sorting.



## Deep dive into Elasticsearch query DSL:

Let's delve deeper into Elasticsearch's Query DSL (Domain Specific Language) and understand its key components and functionalities.

## Understanding Elasticsearch Query DSL

Elasticsearch Query DSL is a powerful mechanism for constructing and executing various types of queries against your Elasticsearch indices. It provides a flexible and expressive syntax to define complex search conditions and retrieve relevant documents efficiently.

### Key Components of Elasticsearch Query DSL:

- 1. Queries:** Queries are the building blocks of Elasticsearch searches. They define the conditions that documents must satisfy to be considered a match. Elasticsearch provides a wide range of queries catering to different use cases, such as term queries, match queries, range queries, etc.
- 2. Filters:** Filters are used to narrow down the result set based on specific criteria without affecting the relevance score of documents. Filters are faster and more efficient than queries since they do not calculate relevance scores.
- 3. Aggregations:** Aggregations allow you to perform analytics and statistical operations on your data. They enable you to extract insights from your documents by computing metrics like counts, averages, sums, etc., and also enable hierarchical, nested, and multi-bucket analyses.
- 4. Sorting:** Sorting allows you to order the search results based on one or more fields. You can specify the sort order (ascending or descending) and prioritize certain fields for sorting.

### Example of Elasticsearch Query DSL:

```
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "Elasticsearch" } },
        { "range": { "views": { "gte": 1000 } } }
      ],
      "must_not": [
        { "term": { "status": "archived" } }
      ],
      "should": [
        { "match": { "category": "DevOps" } }
      ]
    }
  }
}
```

```
    },  
    "sort": [  
      { "date": "desc" }  
    ],  
    "size": 10  
  }  
}
```

In this example:

- We use a bool query to combine multiple conditions using **must**, **must\_not**, and **should**.
- We specify criteria for matching documents based on the title and views fields, while excluding documents with the status “archived”.
- We also include an optional condition where documents matching the “DevOps” category receive a higher relevance score.
- Additionally, we sort the results based on the date field in descending order and limit the number of returned documents to 10.

Elasticsearch Query DSL provides a robust and versatile framework for constructing complex search queries and aggregations.

### Compound queries (**bool**, **must**, **must\_not**, **should**):

Compound queries in Elasticsearch, particularly **‘bool’**, **‘must’**, **‘must\_not’**, and **‘should’**, are fundamental constructs that allow you to build complex search conditions by combining multiple queries together. Let’s delve into each of these compound queries:

#### 1. Bool Query:

The **‘bool’** query is a versatile and powerful compound query that allows you to combine multiple query clauses together. It supports the following sub-clauses:

- **‘must’**: Documents must match all of these clauses to be considered a match.
- **‘must\_not’**: Documents must not match any of these clauses to be considered a match.
- **‘should’**: These clauses are optional. If any of them match, they contribute to the overall relevance score.

- **‘filter’**: Similar to **‘must’**, but does not calculate the relevance score. It’s more efficient for conditions that should simply match or not match.

Example:

```
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "Elasticsearch" } },
        { "range": { "views": { "gte": 1000 } } }
      ],
      "must_not": [
        { "term": { "status": "archived" } }
      ],
      "should": [
        { "match": { "category": "DevOps" } }
      ]
    }
  }
}
```

## 2. Must Query:

The **‘must’** query requires that all specified conditions must match for a document to be considered a match. It’s similar to the logical “AND” operation.

Example:

```
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "Elasticsearch" } },
        { "range": { "views": { "gte": 1000 } } }
      ]
    }
  }
}
```

### 3. Must Not Query:

The **'must\_not'** query specifies conditions that must not be satisfied by a document to be considered a match. It's useful for excluding certain documents from the result set.

Example:

```
{
  "query": {
    "bool": {
      "must_not": [
        { "term": { "status": "archived" } }
      ]
    }
  }
}
```

### 4. Should Query:

The **'should'** query defines optional conditions. If any of the **'should'** clauses match, they contribute to the overall relevance score. It's similar to the logical "OR" operation.

Example:

```
{
  "query": {
    "bool": {
      "should": [
        { "match": { "category": "DevOps" } },
        { "match": { "category": "Data Engineering" } }
      ]
    }
  }
}
```

Compound queries in Elasticsearch, such as **'bool'**, **'must'**, **'must\_not'**, and **'should'**, offer powerful mechanisms for constructing complex search conditions.

**Full-text search and fuzzy matching:**

These are essential features of Elasticsearch that enhance the search capabilities and enable users to find relevant documents even in cases of misspellings or variations in the search terms.

### Full-Text Search:

Full-text search refers to the ability to search for documents based on the textual content within them. Elasticsearch provides various query types and analyzers that enable efficient full-text search capabilities.

#### Example:

Suppose you have a collection of documents containing product descriptions. You can use the **'match'** query to perform a full-text search:

```
{
  "query": {
    "match": {
      "description": "powerful and reliable search engine"
    }
  }
}
```

This query will search for documents where the “description” field contains the phrase “powerful and reliable search engine”.

### Fuzzy Matching:

Fuzzy matching allows Elasticsearch to find documents that are similar to a given term, even if there are spelling mistakes or slight variations in the term. It's particularly useful for handling typos or misspellings in user queries.

#### Example:

Let's say you want to search for documents containing the term “Elasticsearch”, but there might be variations in how it's spelled:

```
{
  "query": {
    "fuzzy": {
      "title": {
        "value": "elasticserch",

```

```
    "fuzziness": "AUTO"  
  }  
}  
}
```

In this example, the ‘**fuzziness**’ parameter is set to “AUTO”, which allows Elasticsearch to determine the maximum edit distance for fuzzy matching based on the length of the term. Elasticsearch will then find documents where the “title” field closely resembles “elasticserch”.

Full-text search and fuzzy matching are powerful features of Elasticsearch that enable users to perform flexible and accurate searches across large volumes of textual data.

### Handling pagination and sorting:

They are important aspects of managing search results effectively, especially when dealing with large datasets. Elasticsearch provides features to facilitate pagination and sorting, allowing users to navigate through search results efficiently.

### Pagination:

Pagination involves breaking search results into manageable chunks or pages, typically to improve performance and user experience by loading content incrementally.

### Example:

Suppose you have a search query that returns a large number of results, and you want to display them in batches of 10 documents per page. You can use the ‘**from**’ and ‘**size**’ parameters to implement pagination:

```
{  
  "from": 0,  
  "size": 10,  
  "query": {  
    "match": {  
      "category": "Technology"  
    }  
  }  
}
```

In this example, **‘from’** specifies the starting index of the first document to retrieve (0-based index), and **‘size’** indicates the maximum number of documents to return for each page.

### Sorting:

Sorting allows you to order search results based on specific fields or criteria. Elasticsearch supports sorting in ascending or descending order.

### Example:

Let's say you want to sort search results based on the “price” field in descending order:

```
{
  "query": {
    "match_all": {}
  },
  "sort": [
    { "price": { "order": "desc" } }
  ]
}
```

In this example, search results will be sorted based on the “price” field in descending order, meaning documents with higher prices will appear first.

### Combined Example:

You can combine pagination and sorting to provide a more refined search experience:

```
{
  "from": 0,
  "size": 20,
  "query": {
    "match": {
      "category": "Books"
    }
  },
  "sort": [
    { "publish_date": { "order": "desc" } }
  ]
}
```



```
]
}
```

This example retrieves the first 20 documents matching the “Books” category and sorts them based on the “publish\_date” field in descending order.

Here are some **scenario-based interview questions** covering advanced querying!

1. You’re building a search feature for an e-commerce platform. Users should be able to search for products by name, category, and price range. How would you utilize compound queries to fulfill this requirement?
2. In a social media application, users can search for posts by text content, but they should not see posts that they’ve already liked. How would you use compound queries to achieve this?
3. A user is searching for articles related to “machine learning” on a news website. How would you implement a full-text search query to retrieve relevant articles from Elasticsearch?
4. In a document management system, users frequently make typographical errors when searching for documents. How could fuzzy matching be applied to accommodate these errors and improve search accuracy?
5. You’re developing a search functionality for a blog platform. Users should be able to see 10 blog posts per page and navigate through pages. How would you implement pagination using Elasticsearch?
6. In a job portal application, search results should be sorted based on the relevance of the job posting and the posting date. How would you configure sorting to achieve this?
7. You’re tasked with optimizing the performance of a search feature in an application that experiences high traffic. What strategies would you employ in Elasticsearch to improve search responsiveness and efficiency?
8. A user complains that search results seem inaccurate or irrelevant. How would you troubleshoot and debug potential issues with search queries in Elasticsearch?

## Conclusion:

In Day 4, we've explored the depths of Elasticsearch's Advanced Query DSL. By leveraging compound queries, full-text search, fuzzy matching, pagination, and sorting, you can harness the full power of Elasticsearch to build highly performant and relevant search experiences for your applications.

Stay tuned for Day 5, where we'll delve into Elasticsearch Aggregations.

[Interview](#)[Elasticsearch](#)[DevOps](#)[Learning](#)[Training](#)[Following](#)

**Written by Navya Cloudops**

514 Followers

More from Navya Cloudops



N Navya Cloudops

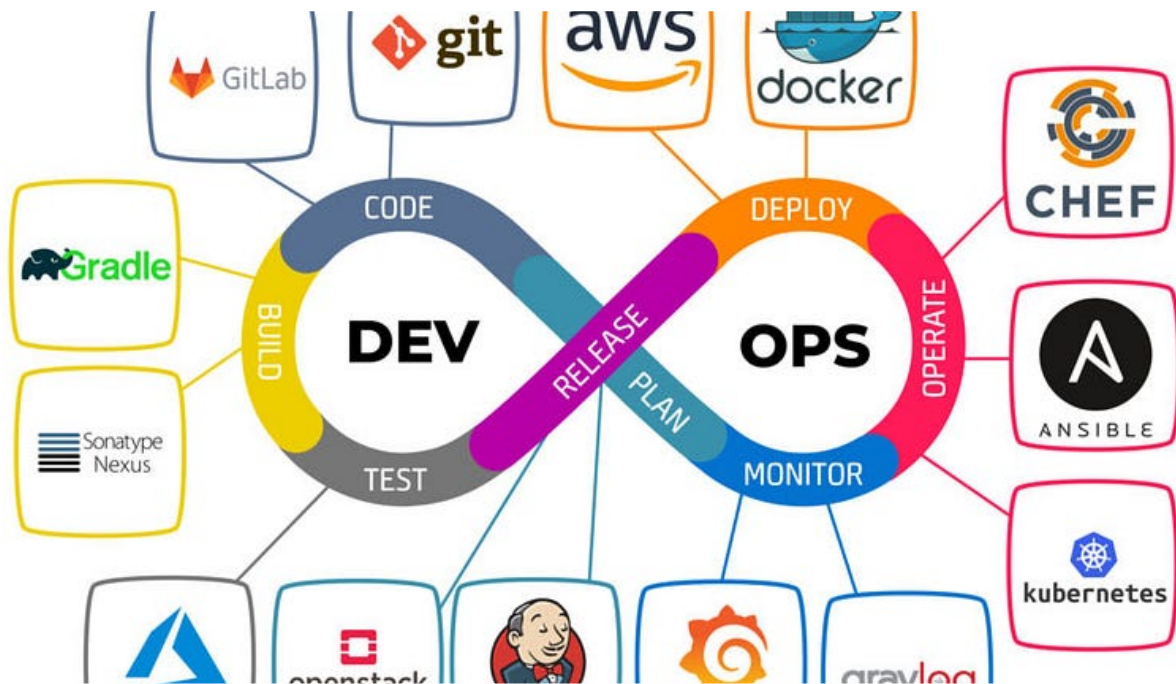
## Real-time Interview Questions for 4 years Experience!!

Here are some scenario based interview questions with respect to 4 years experience for a position in CITI Bank.

★ · 3 min read · Jan 24, 2024

👏 88    💬

🔖<sup>+</sup>    ⋮



N Navya Cloudops

## DevOps Zero to Hero in 30 days!!

Here's a 30-day DevOps course outline with a detailed topic for each day!

4 min read · Jul 12, 2023

👏 26    💬 1

🔖<sup>+</sup>    ⋮



N Navya Cloudops

## DevOps Zero to Hero—Day 14: Release Management!!

Welcome to Day 14 of our 30-day course on Software Development Best Practices! In today's session, we'll delve into the critical aspect of...

7 min read · Jul 26, 2023



1



N Navya Cloudops

## DevOps Zero to Hero—Day 20: Deployment Strategies

Welcome to Day 20 of our comprehensive 30-day course on Application Deployment! In this segment, we will delve into various deployment...

8 min read · Aug 3, 2023



See all from Navya Cloudops

### Recommended from Medium



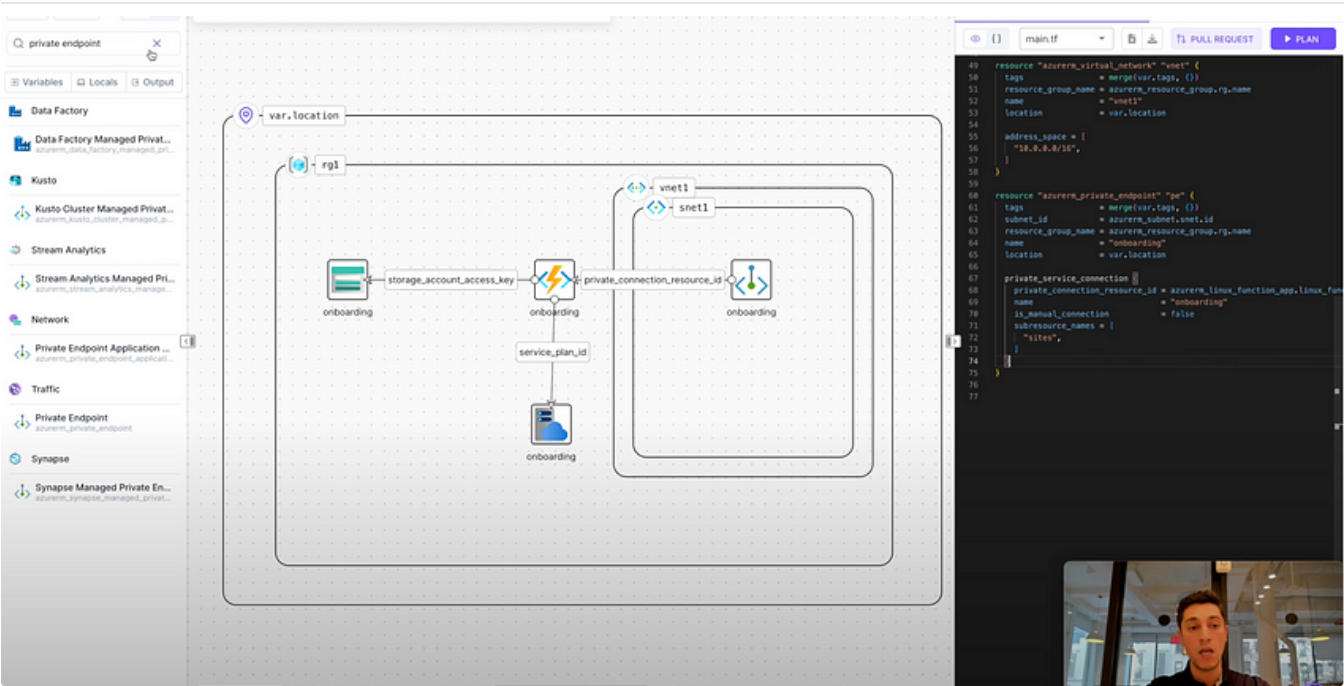
Chameera Dulanga in Bits and Pieces


### Best-Practices for API Authorization

4 Best Practices for API Authorization

9 min read · Feb 6, 2024





 Mike Tyson of the Cloud (MToC)

Complete Terraform Tutorial

Your journey in building a cloud infrastructure from scratch and learning Terraform with Brainboard starts here.

25 min read · Feb 8, 2024

 248 

Lists



Self-Improvement 101  
20 stories · 1364 saves



How to Find a Mentor  
11 stories · 422 saves

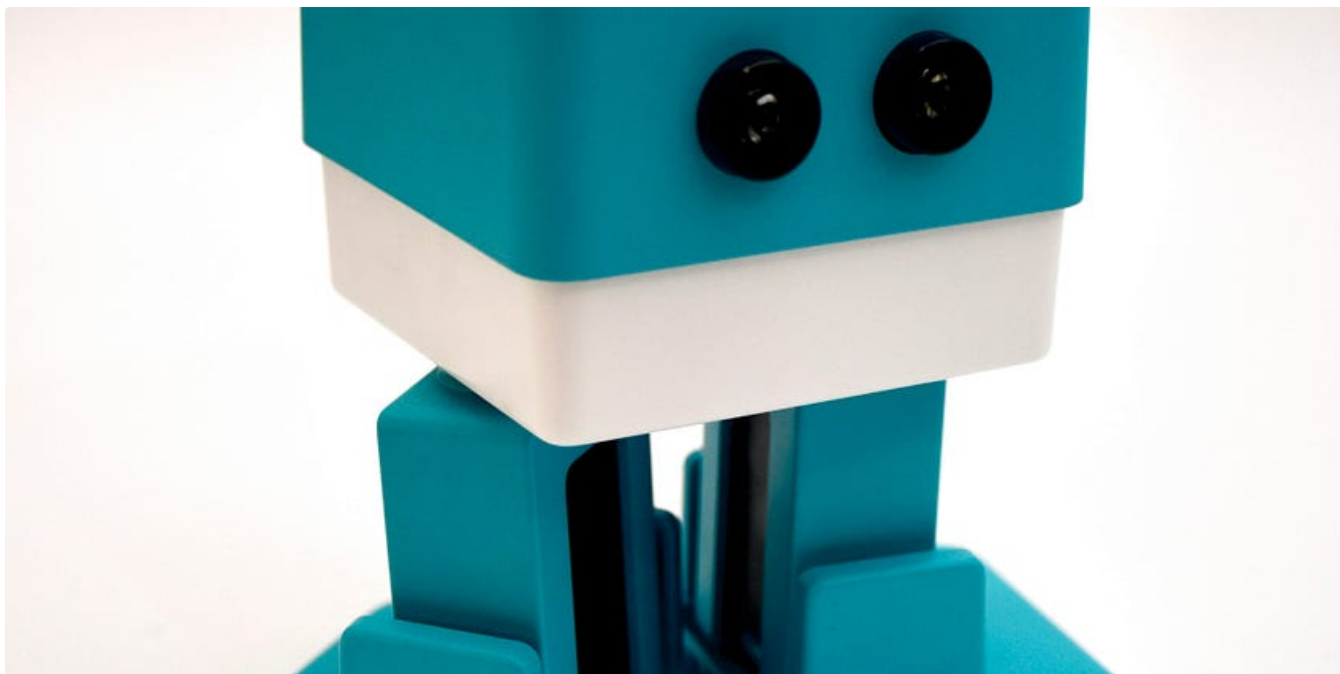


Good Product Thinking  
11 stories · 470 saves



The New Chatbots: ChatGPT, Bard, and Beyond  
12 stories · 307 saves





 Akhilesh Mishra

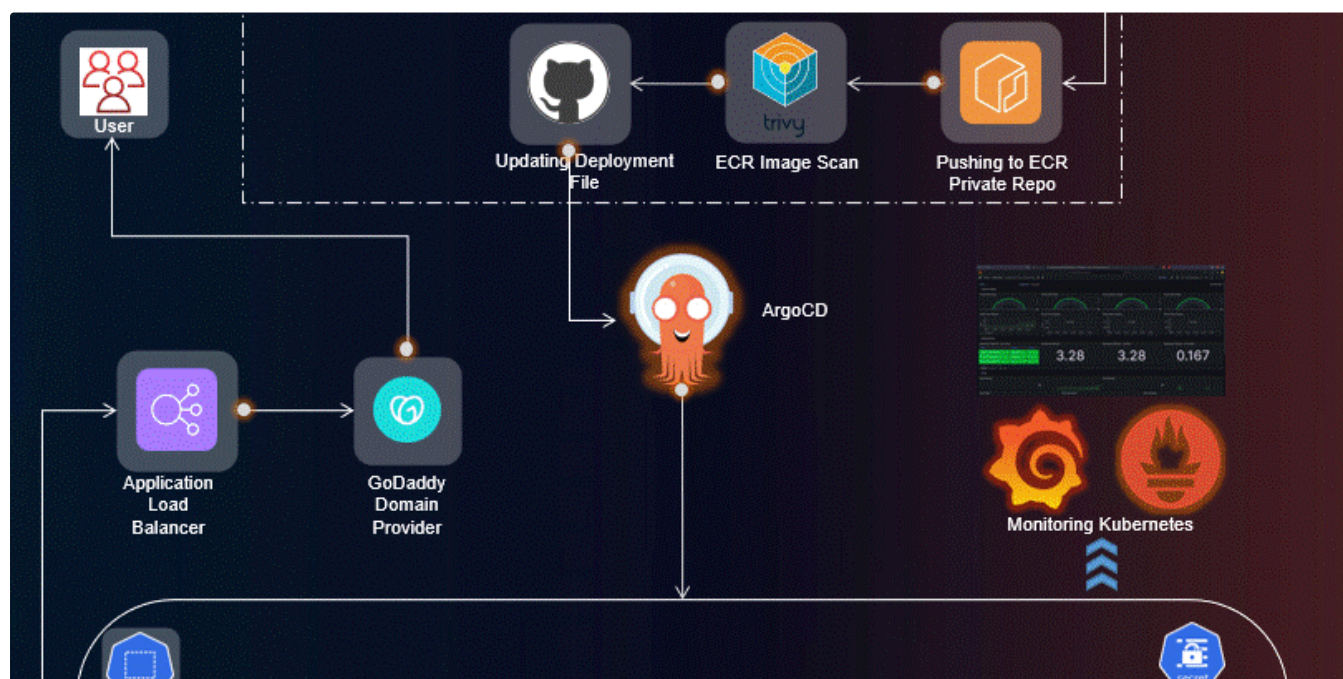
## You should stop writing Dockerfiles today— Do this instead

Using docker init to write Dockerfile and docker-compose configs

5 min read · Feb 8, 2024

 2.1K  20



 Aman Pathak in Stackademic

## Advanced End-to-End DevSecOps Kubernetes Three-Tier Project using AWS EKS, ArgoCD, Prometheus...

Project Introduction:

23 min read · Jan 18, 2024



1.5K



15



Dolan Miu

### Why have 100% Test Coverage

100% test coverage is somewhat of a taboo phrase in software. It's "unachievable" with "diminishing returns" they would say. Non-developers...

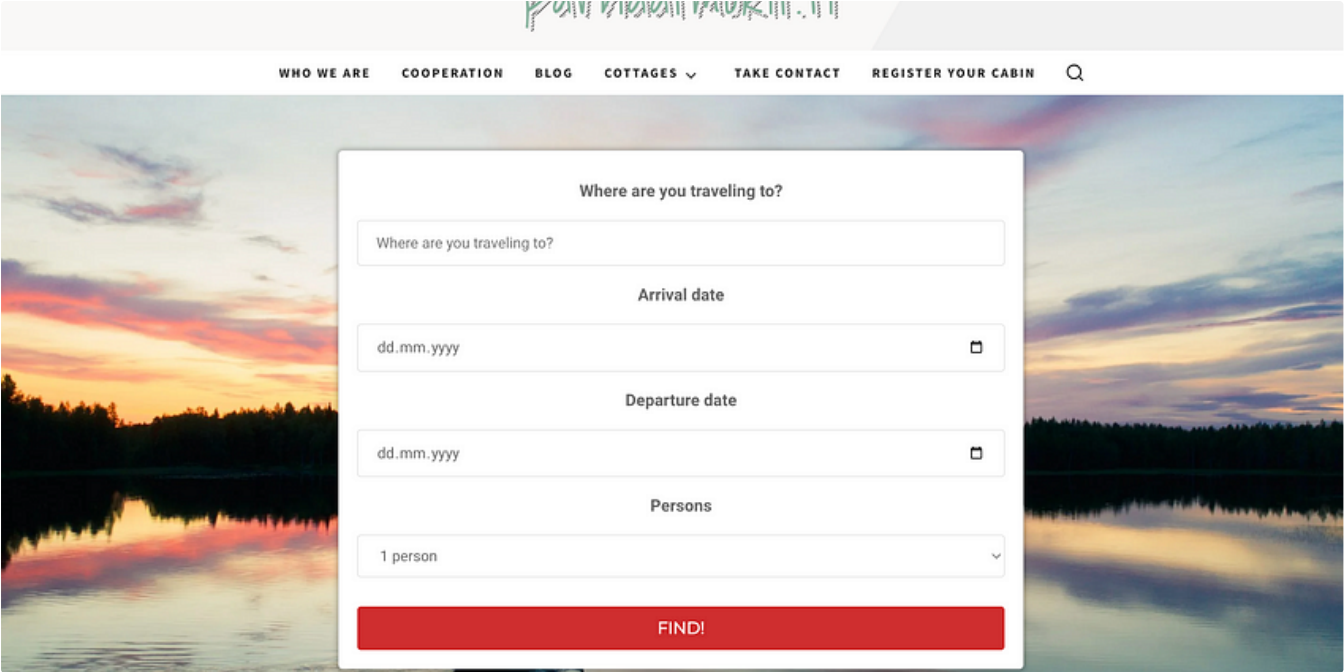
4 min read · 3 days ago



53







 Artturi Jalli

# I Built an App in 6 Hours that Makes \$1,500/Mo

Copy my strategy!

🌟 · 3 min read · Jan 23, 2024

 9.2K    121

See more recommendations