



# Sysdig



## The DZone Guide to Microservices: Breaking Down the Monolith

An API-First Approach for Microservices on Kubernetes

Akka: The Ideal Actor-Based Framework

Explore the Most Common Patterns for Microservices



**Download My Free PDF**

# Microservices Monitoring Best Practices

DOWNLOAD



BROUGHT TO YOU BY

# Microservices With Apache Kafka and Netflix OSS

This microservice building tutorial continues by walking through Eureka service discovery, the config server and user service, and configuring the email service.

by **Shashank Rastogi** · Mar. 01, 18 · **Microservices Zone**

Sysdig is the container intelligence company. The only unified platform for monitoring, security, and troubleshooting in a microservices-friendly architecture.

---

This is Part 2 of the blog series on building microservices with Netflix OSS and Apache Kafka. The first part on installing Apache Kafka in a Docker container is published here.

Before this, I have also published a blog on Docker, so if anyone wants to take a look at Docker, they can read it here.

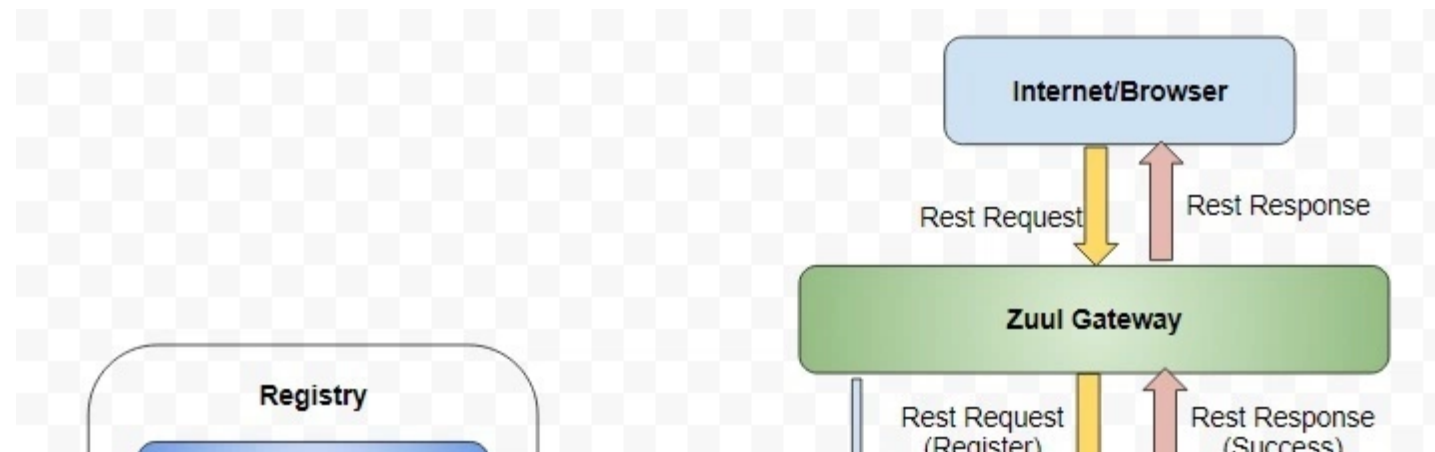
In this blog, we will create a microservice to send an email after a user registers successfully. This is a very simple implementation to explain the Netflix OSS stack and the concept of building a microservice. Therefore, my thrust is not on building a registration module.

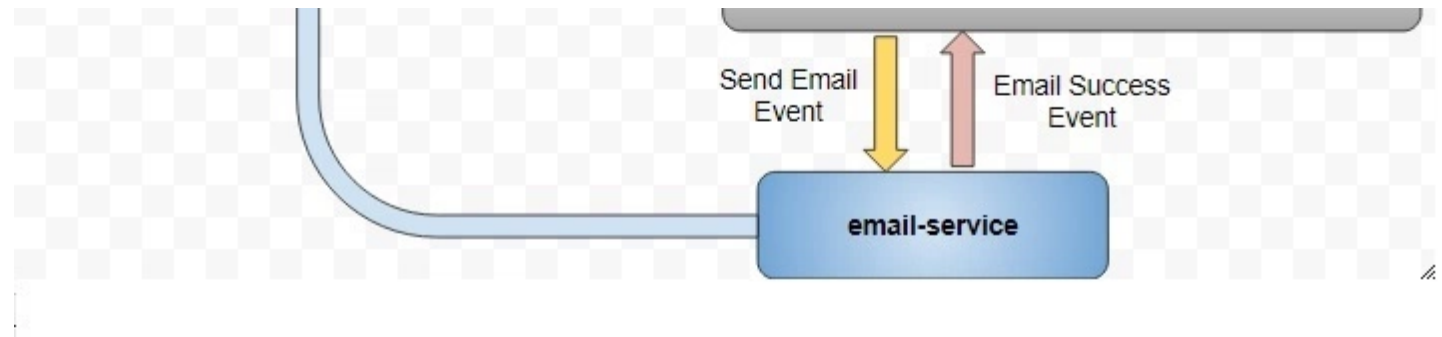
configuration from this server. We are going to use a Github repository for this purpose.

3. **Zuul** - Zuul is the gateway service or load balancer which is used to proxy web requests to the backend services. It provides dynamic routing, monitoring, resiliency, and security.
4. **User Service** - This service will register a new user and will send the message to the Kafka broker.
5. **Email Service** - This service will consume the message from the Kafka broker and send the Registration Success email notification to the user.

Prerequisites for building the microservices project:

- Maven
- Spring tool suite, or any other IDE for Spring Boot development





## Service Registry (Eureka)

One of the first things needed is a central Service Registry which allows service discovery. Implementing a Eureka Server for use as service registry is quite easy and involves the following steps:

Adding *spring-cloud-starter-eureka-server* to the dependencies in pom.xml:

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-eureka-server</artifactId>
4 </dependency>
```

Enable the Eureka Server in a main \*Application file and annotate it with *@EnableEurekaServer*:

```
1 @SpringBootApplication
```

application name as below:

```
1  server:
2    port: 8761
3  spring:
4    application:
5      name: discovery-service
```

In application.yml, define the standard configuration for the dev env as below:

```
1  eureka:
2    client:
3      register-with-eureka: false
4      fetch-registry: false
```

*Note: Setting both register-with-eureka and fetch-registry to false will stop the registry that there are no replica nodes. Don't forget to set them back to true in a production env.*

Build and execute the project from cmd prompt:

- mvn clean install
- Go to the target folder

Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

## Config Server

As is always the case, we will have to support the application in multiple environments, and therefore, a need for a config server to host the configuration files. In this case, we are going to use a Git repository to store the configuration files for each service. When the service is started, it will ask for configurations.

We will create a project in the Git repository with the name **EmailServiceConfigProperties**. The structure of this project would be like this:

```

1  EmailServiceConfigProperties
2      email-service
3          default
4              email-service.yml
-          dev
  
```

```
14         prod
15         user-service.yml
```

This project is available in the Git repository, so do not change the properties for the Kafka broker and Gmail smtp server.

The config server will be built as a discovery service (Eureka) client. So when a service is started, it will take the config server location from the discovery service.

Add the dependencies for the config server and discovery client to the pom.xml, as given below:

```
1     <dependency>
2         <groupId>org.springframework.cloud</groupId>
3         <artifactId>spring-cloud-config-server</artifactId>
4     </dependency>
5     <dependency>
6         <groupId>org.springframework.cloud</groupId>         <artifactId>spring-clo
7     </dependency>
```

Enable the config server using the annotation `@EnableConfigServer` and Eureka discovery using the annotation `@EnableEurekaClient` in the main `*Application` file in the project.



Configure properties like define server/port, etc. In bootstrap.yml, define the server port and application name as below:

```
1  server:
2    port: 8888
3  spring:
4    application:
5      name: config-server
```

*Note: By default, the config server runs on port 8888.*

In application.yml, define the Git repository path of the project storing configuration files as given below:

```
1  spring:
2    profile:
3      active: dev
4  cloud:
5    config:
6      server:
7        git:
8          uri: https://github.com/srast3/spring-kafka-ms.git/
```

- Run `java -jar ConfigServer-0.0.1-SNAPSHOT.jar`

Test ConfigServer and you will see that it returns the configuration properties from the dev folder in the Git repository – `http://localhost:8888/email-service/dev`

Since it is discovered through Eureka service registry, it will also show in the Eureka discovery console:

### System Status

Environment	test	Current time	2018-02-02T11:11:07 +0000
Data center	default	Uptime	01:14
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	5

**EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.**

### DS Replicas

localhost

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-SERVER	n/a (1)	(1)	UP (1) - 192.168.0.3:config-server:8888
EMAIL-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.3:email-service:8082

## User Service

## 6. Add the following dependencies to pom.xml:

```
1  <dependency>
2      <groupId>org.springframework.cloud</groupId>
3      <artifactId>spring-cloud-starter-config</artifactId>
4  </dependency>
5  <dependency>
6      <groupId>org.springframework.cloud</groupId>
7      <artifactId>spring-cloud-starter-eureka</artifactId>
8  </dependency>
9  <dependency>
10     <groupId>org.springframework.boot</groupId>
11     <artifactId>spring-boot-starter-data-jpa</artifactId>
12 </dependency>
13 <dependency>
14     <groupId>org.springframework.kafka</groupId>
15     <artifactId>spring-kafka</artifactId>
16 </dependency>
17 <dependency>
18     <groupId>com.h2database</groupId>
19     <artifactId>h2</artifactId>
20     <scope>runtime</scope>
```

```
5         SpringApplication.run(UserAccountApplication.class, args);
6     }
7 }
```

In bootstrap.yml, we will enable cloud config discovery and add the port and application name.

```
1  server:
2    port: 8081
3  spring:
4    application:
5      name: user-service
6    profiles:
7      active: dev
8    cloud:
9      config:
10       discovery:
11         enabled: true
12         service-id: config-server
```

In the configuration file user-service.yml, define the Kafka broker and H2 database properties.

```
1  spring:
```

```
10     kafka:
11       bootstrap-servers: 192.168.99.100:9092
12       topic:
13         userRegistered: USER_REGISTERED_TOPIC
14     security:
15       basic:
16         enabled: false
```

Now let's discuss the project structure. In the user service project, we need to define an entity which will represent the data structure and will be used to transfer the data.

### **/User.java**

```
1  @Entity
2  @Table(name = "\"User\"")
3  public class User {
4
5      @Id
6      @GeneratedValue(strategy = GenerationType.IDENTITY)
7      private Long id;
8      @NotNull
9      private String userName;
10     @NotNull
```

```
2    }
```

Then there will be a service class which will actually do the dirty work of either registering a user or fetching all users based on the call.

```
1  public interface UserService {  
2      User registerUser(User input);  
3      Iterable<User> findAll();  
4  }
```

Service implementation of this interface will have the logic to send a message to Kafka and save the new user in the database.

Next, we will define a REST controller and will expose the rest endpoints.

### **/UserController**

```
1  @RestController  
2  @RequestMapping(value = "/", produces = MediaType.APPLICATION_JSON_VALUE)  
3  public class UserController {
```

**Create a post method – /register to register a new user:**

```
1      @RequestMapping(method = RequestMethod.POST, path = "/register")
2      public ResponseEntity<User> register(@RequestBody User input) {
3          User result = userService.registerUser(input);
4          return new ResponseEntity<>(result, HttpStatus.OK);
5      }
```

Now, let's characterize the sender configuration which produces the message to Kafka Topic. First, we will define a `KafkaTemplate`. The Kafka template will require `ProducerFactory`, which will set the strategy to produce a `Producer` instance. The `ProducerFactory` will need a `Map` of configuration properties, the most essential of which are `BOOTSTRAP_SERVERS_CONFIG`, `KEY_SERIALIZER_CLASS_CONFIG`, and `VALUE_SERIALIZER_CLASS_CONFIG`.

**/SenderConfig.java**

```
1      @Configuration
2      public class SenderConfig {
3
4          @Value("${spring.kafka.bootstrap-servers}")
5          private String bootstrapServers;
6
7          @Bean
```

```
17     public ProducerFactory<String, User> producerFactory() {  
18         return new DefaultKafkaProducerFactory<>(producerConfigs());  
19     }  
20  
21     @Bean  
22     public KafkaTemplate<String, User> simpleKafkaTemplate() {  
23         return new KafkaTemplate<>(producerFactory());  
24     }  
25  
26     @Bean  
27     public Sender sender() {  
28         return new Sender();  
29     }  
30 }
```

The String message payload is transformed from a User object with the help of JsonSerializer. Finish with the implementation of Sender bean which will use the KafkaTemplate configured above.

### **/Sender.java**

```
1     public class Sender {  
2         private static final Logger LOGGER = LoggerFactory.getLogger(Sender.class);
```



# Email Service

Next, we will build the microservice for sending emails upon successful registration. This microservice will listen to the topic that comes from the user service. For this purpose, we will create a DTO object and will configure the Kafka consumer to transform the incoming payload to it.

As a first step, let's create a Spring project and add the required dependencies to pom.xml. Required dependencies are **Eureka Discovery, JPA, H2, Kafka, and Config Client**.

```
1  <dependency>
2      <groupId>org.springframework.cloud</groupId>
3      <artifactId>spring-cloud-starter-config</artifactId>
4  </dependency>
5  <dependency>
6      <groupId>org.springframework.cloud</groupId>
7      <artifactId>spring-cloud-starter-eureka</artifactId>
8  </dependency>
9  <dependency>
10     <groupId>org.springframework.boot</groupId>
11     <artifactId>spring-boot-starter-data-jpa</artifactId>
12 </dependency>
13 <dependency>
14     <groupId>org.springframework.kafka</groupId>
```

```
24 </dependency>
```

We are going to use Gmail smtp to send the email for this project, and related properties are configured in email-service.yml in the config repository in GitHub.

```
1  mail:
2    host: smtp.gmail.com
3    port: 587
4    username: <userid>
5    password: <password>
6    properties.mail.smtp:
7      auth: true
8      starttls.enable: true
```

Then, we need to configure ConsumerFactory to be able to consume messages from the broker.

### **/ReceiverConfig.java**

```
1  @Configuration
2  @EnableKafka
3  public class ReceiverConfig {
4      @Value("${spring.kafka.bootstrap-servers}")
```

```
13         return props;
14     }
15     @Bean
16     public ConsumerFactory<String, UserDTO> consumerFactory() {
17         return new DefaultKafkaConsumerFactory<>(consumerConfigs(), new StringDese
18             new JsonSerializer<>(UserDTO.class));
19     }
20     @Bean
21     public ConcurrentKafkaListenerContainerFactory<String, UserDTO> kafkaListenerC
22         ConcurrentKafkaListenerContainerFactory<String, UserDTO> factory =
23         new ConcurrentKafkaListenerContainerFactory<>();
24         factory.setConsumerFactory(consumerFactory());
25         return factory;
26     }
27     @Bean
28     public Receiver receiver() {
29         return new Receiver();
30     }
31 }
```

Now, lets create the Kafka listener which will be invoked when a message is received

```
7      public void receive(UserDTO payload) {
8          LOGGER.info("receivedoad='{ }'", payload);
9          emailService.sendSimpleMessage(payload);
10         latch.countDown();
11     }
12 }
```

As done earlier for the user service, we will have to define a service implementation class which will take the incoming request, transform it, and send the email.

### **/EmailServiceImpl**

```
1  @Component
2  public class EmailServiceImpl implements EmailService {
3      @Autowired
4      public JavaMailSender emailSender;
5      @Autowired
6      public MailRepository mailRepository;
7      @Override
8      public void sendSimpleMessage(UserDTO input) {
9          try {
10             Mail newMail = new Mail();
```

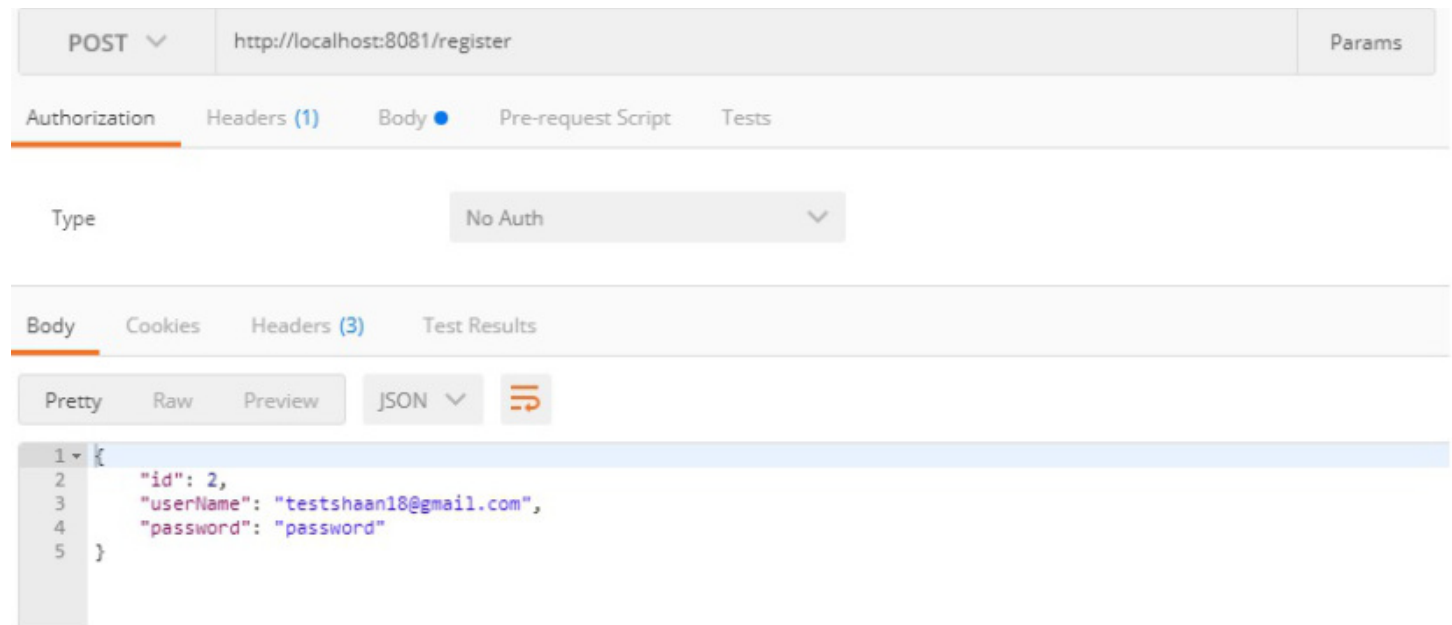
```
20         mailRepository.save(newMail);
21         emailSender.send(message);
22     } catch (MailException exception) {
23         exception.printStackTrace();
24     }
25 }
26 }
```

To test if everything is fine, build and run the project

- Verify Service registry (Eureka) is running (<http://localhost:8761>)
- Verify that the config server (Spring Cloud Config) is running and user-service and email-service configuration are available (<http://localhost:8888/user-service/default>)
- Build the project: `mvn clean install`
- Run `java-jar EmailService-0.0.1-SNAPSHOT.jar`

We also need to start the Docker container for Zookeeper and Kafka. There is a slight change in the command we used to run the Docker container for Kafka in the previous blog:

```
1 docker run -d --name kafka --network kafka-net --publish 9092:9092 --publish 7203:
```



You can also verify the user by calling GET <http://localhost:8081/member> .

Verify that the registration success email was received at your email address.

So this is it! We have successfully created our first microservice project with Netflix OSS, Apache Kafka, and Spring Boot. The complete code for this microservice implementation is available on [GitHub](#).


In the rest of the blog series, I will publish articles on the following topics, so stay tuned:

---

Download our white paper: The Architecture of the Sysdig Container Intelligence Platform. Built for microservices and containers for monitoring, security, troubleshooting

---

Topics: MICROSERVICES , DOCKER , TUTORIAL , NETFLIX OSS

Published at DZone with permission of Shashank Rastogi . [See the original article here.](#)   
Opinions expressed by DZone contributors are their own.

# Get the best of Microservices in your inbox.

Stay updated with DZone's bi-weekly Microservices Newsletter. [SEE AN EXAMPLE](#)

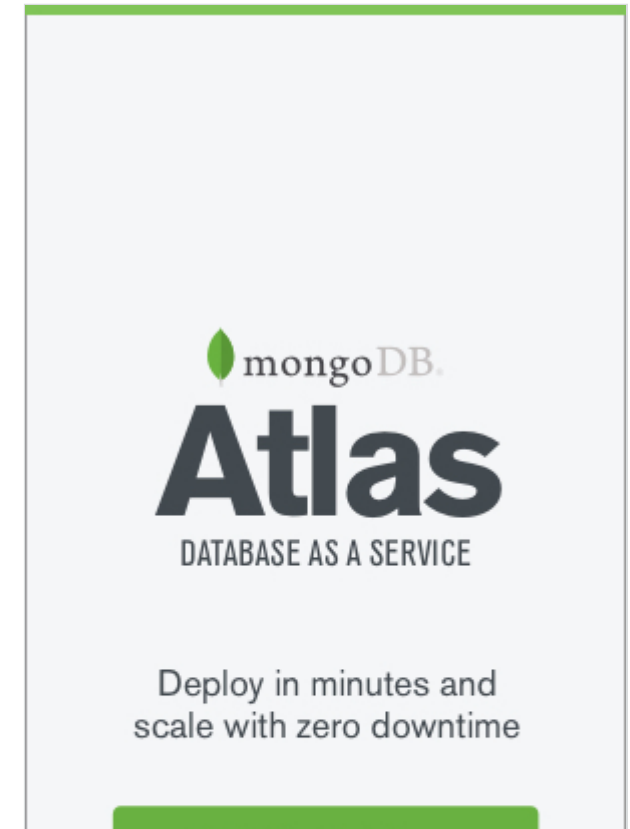
SUBSCRIBE

## Microservices Partner Resources

How to Overcome the Microservices Sprawl

AppDynamics







# Build or Buy? — The Eternal IT Question

To build or not to build, but buy? Here we take a look at the implication and situations in which one might be more beneficial than the other.

by Jay Gordon  MVB · Mar 05, 18 · Cloud Zone

Discover when your data grows or your application performance demands increase, MongoDB Atlas allows you to scale out your deployment with an automated sharding process that ensures zero application downtime.

---

I have been thinking a lot about the idea of “Build or Buy” in regards to systems for IT solutions. What do I mean by “Build or Buy” exactly?

## Build

Build an IT solution based on your team putting together the parts required. Common examples are building a custom database or storage systems for your application. These solutions are typically within the (public or private) cloud or can be hosted on-premise.

**I advocate for both. When what's for sale does not work, then you will have no choice but to build it. Coming to that conclusion is the hard part.**

**— Kelsey Hightower (@kelseyhightower) February 20, 2018**

---

At what point do you determine you've met the limits of what a platform has available in regards to scale and resources? Additionally, when do you determine that a self-hosted solution is no longer as valuable as a Platform-as-a-Service?

## Why Build?

Building a solution for something such as data storage tends to be a common task for many teams in enterprise solutions. There are a number of concerns that major organizations tend to consider when deploying large storage arrays as well that can put pressure on a team:

- How will we back this up?
- Who will provide long-term maintenance?
- Will costs remain reasonable?
- Do we have any specific business or regulatory rules we need to be included in how we store data?

deliver applications. This change to “buy” has also put many other options in the hands of developers:

- Self-service via APIs or GUI based interfaces.
- Self-remedying failure response
- Alerting.
- Automated processes to handle common administration tasks.
- Using a scalable solution (both scaling up and down).

## What's Going To Work For Me?

These are just a few reasons why businesses and developers turn to services to host their applications. Your use case will ultimately require you to put into a plan all of the potential risks using either solution will present. There is a valid point to state that you really should buy until it becomes evident it's time to build. This is one of the reasons to avoid service vendor lock-in when selecting technologies to leverage when building apps.

### Tips:

- Consider open formats like JSON to store your data that translate to many different languages.
- If selecting a service, ensure that the vendor will permit your data to others if your situation changes. (costs, competition, credits)
- Make checklists and document the architecture of your systems regardless of their hosting for

---

MongoDB Atlas is the easiest way to run the fastest-growing database for modern applications — no installation, setup, or configuration required. Easily live migrate an existing workload or start with 512MB of storage for free.

---

**Topics:** DEVOPS, CLOUD COMPUTING, DATABASES, STORAGE, MONGODB

**Opinions expressed by DZone contributors are their own.**