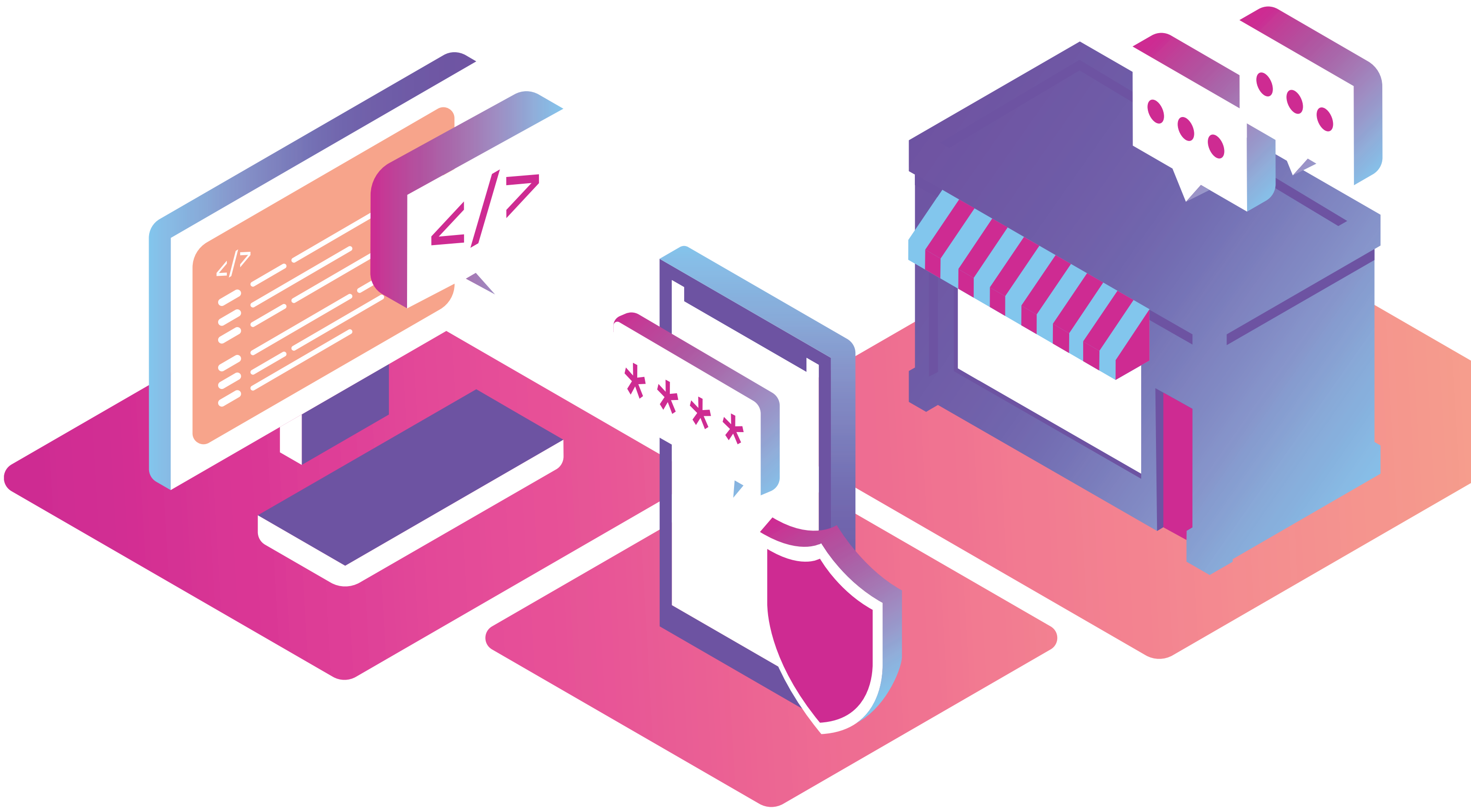
VONAGE | MongoDB Atlas

Using Vonage Communications APIs With MongoDB Atlas



Developing applications is hard. Not only are there the base requirements for the application itself, but there are always common problems to solve, like how to authenticate users, how the database is managed, and where to host the application. In 2023 we are almost spoiled with the number of services that can help solve these problems, but all of this needs to be brought together inside an application. **One solution that has come along is MongoDB Atlas, a suite of products that are designed to help developers build their applications quickly and handle many common problems.**

What do we plan to do?

In the next series of articles, we are going to walk through building an application that utilizes MongoDB Atlas and a suite of Vonage Communications APIs. The demo application will take the form of a simple restaurant website and an associated backend.

We will show:

- How to integrate Vonage Verify API with a user login
- How to use Vonage Messages API to send an order confirmation
- How to use Vonage Meetings API for issue resolution

We will also help developers set up:

- A MongoDB Atlas cluster and associated App Service
- Having a front-end app talk back to MongoDB Atlas via Realm
- User Authentication with MongoDB Users

While we will be breaking down how all of this works over time, feel free to take a quick glance at the source code for the application at the **source code on GitHub**.

Prerequisites

- **MongoDB Account**
- **Vonage Developer Account**
- **Realm CLI**
  - A command line application that makes it easier to manage App Services in MongoDB Realm
- **Node.js 16+**
  - Node.js is an open-source, cross-platform JavaScript runtime environment.

What is MongoDB Atlas?

MongoDB Atlas is a hosted cloud database service that multiple cloud hosting providers can use. This means that you can host your database in various regions and across AWS, Azure, and Google Cloud Platform for multi-cloud availability. Since the databases are hosted in the cloud, scaling up and down can be done on demand. For developers, it frees up a lot of administrative time handling more servers by allowing MongoDB to manage all the infrastructure while developers can focus on their applications.

MongoDB Atlas allows you to set up MongoDB clusters. If you are not familiar with MongoDB, it is a **document-based NoSQL Database** system. Unlike a traditional Relational Database Management System (RDMS), MongoDB stores information as JSON-like documents that can be searched. It has limited relational capabilities and focuses more on lightly-structured data than tabular row/column architectures. Documents are grouped as "collections," replacing the standard table architecture. Documents in a collection can share a common schema, like an RDMS table, but can also change their structure.

```
{
  "_id": ObjectId("6413733ba623c618c2fab2d9"),
  "name": "Hamburger",
  "price": 995
}
```

NoSQL databases, as the name suggests, do not use SQL to query for information. MongoDB uses a JSON-like query syntax to search for documents that match the criteria. For example, instead of using something like SELECT * FROM users WHERE admin = true, you would use the following syntax:

```
db.users.find({
   admin: {
   $eq: true
   }
})
```

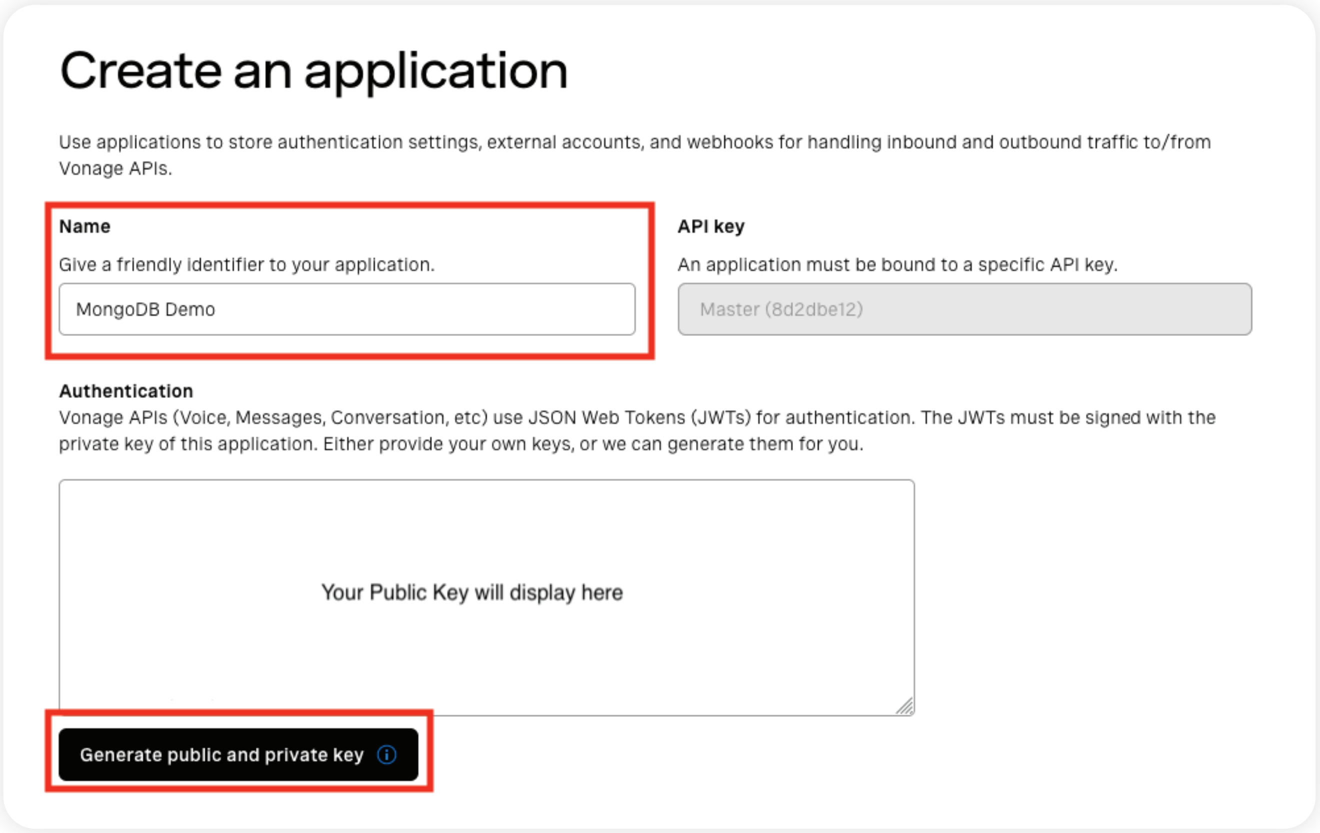
Many developers prefer using a NoSQL database for the freedom of the schemaless document design. There are no major migrations as new "columns" can be added to documents by adding them to new or existing documents. You can define a schema if you want, but it largely only helps the database engine filter through data in larger data sets.

MongoDB Atlas also brings a few additional features that developers can use to build their applications on top of the robust NoSQL database that MongoDB brings. This is part of their "App Service" layer that adds user authentication, a serverless function runtime, an associated API gateway and router, automatic GraphQL and HTTPS data access, and a device data syncing service called MongoDB Realm.

This means that a developer can start developing their application right away without having to piece together a bunch of disparate services and can focus on the business problems that the application solves, not shave the proverbial yak on how to do user authentication or how to deploy code. Atlas and its App Services can do much of that heavy lifting for a developer.

Set up a Vonage Application

Our Messages, Verify, and In-App Messaging APIs are all backed by a Vonage Application, a set of configuration data that can be grouped. Once you have signed into your developer account, go to the Applications page and **create a new application**. Give your application a name like MongoDB Demo, then click Generate Public and Private key. This will create the authentication keys we will use in the SDKs.



Create an application

Use applications to store authentication settings, external accounts, and webhooks for handling inbound and outbound traffic to/from Vonage APIs.

Name

Give a friendly identifier to your application.

MongoDB Demo

API key

An application must be bound to a specific API key.

Master (8d2dbe12)

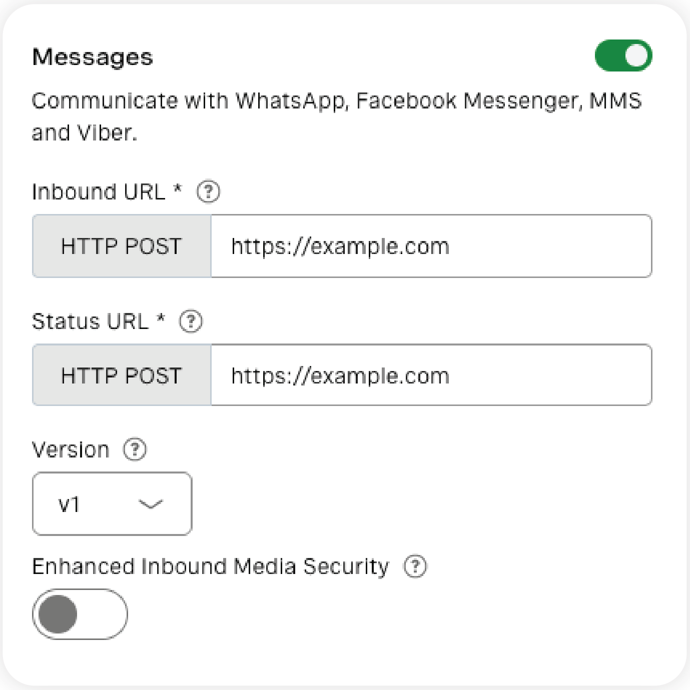
Authentication

Vonage APIs (Voice, Messages, Conversation, etc) use JSON Web Tokens (JWTs) for authentication. The JWTs must be signed with the private key of this application. Either provide your own keys, or we can generate them for you.

Your Public Key will display here

Creating a Vonage application

Now scroll down, and we can turn a few different capabilities. We will need Messages, RTC (In-app voice & messaging), and the Meetings API. Toggle each of those capabilities on. Messages needs a few callback URLs that we will not utilize for the moment, so enter https://example.com for those handfuls of URLs that are required. Meetings can stay blank. Once that's all done, click on "Generate new application."



Messages

Communicate with WhatsApp, Facebook Messenger, MMS and Viber.

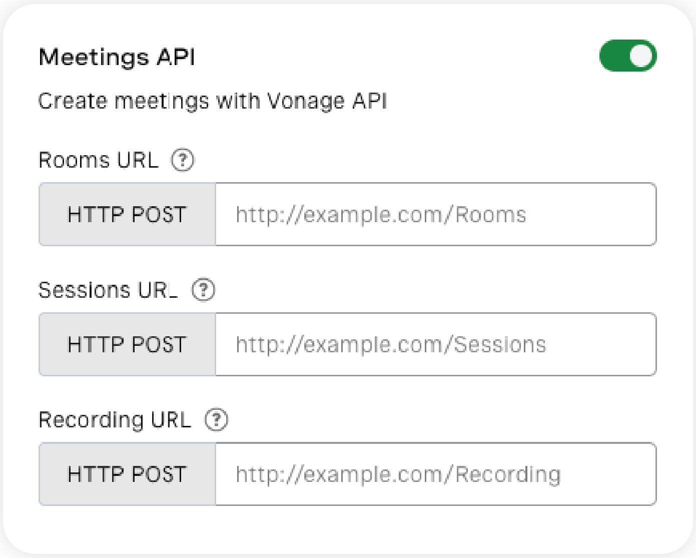
Inbound URL * HTTP POST https://example.com

Status URL * HTTP POST https://example.com

Version v1

Enhanced Inbound Media Security

Messages API Capability



Meetings API

Create meetings with Vonage API

Rooms URL HTTP POST http://example.com/Rooms

Sessions URL HTTP POST http://example.com/Sessions

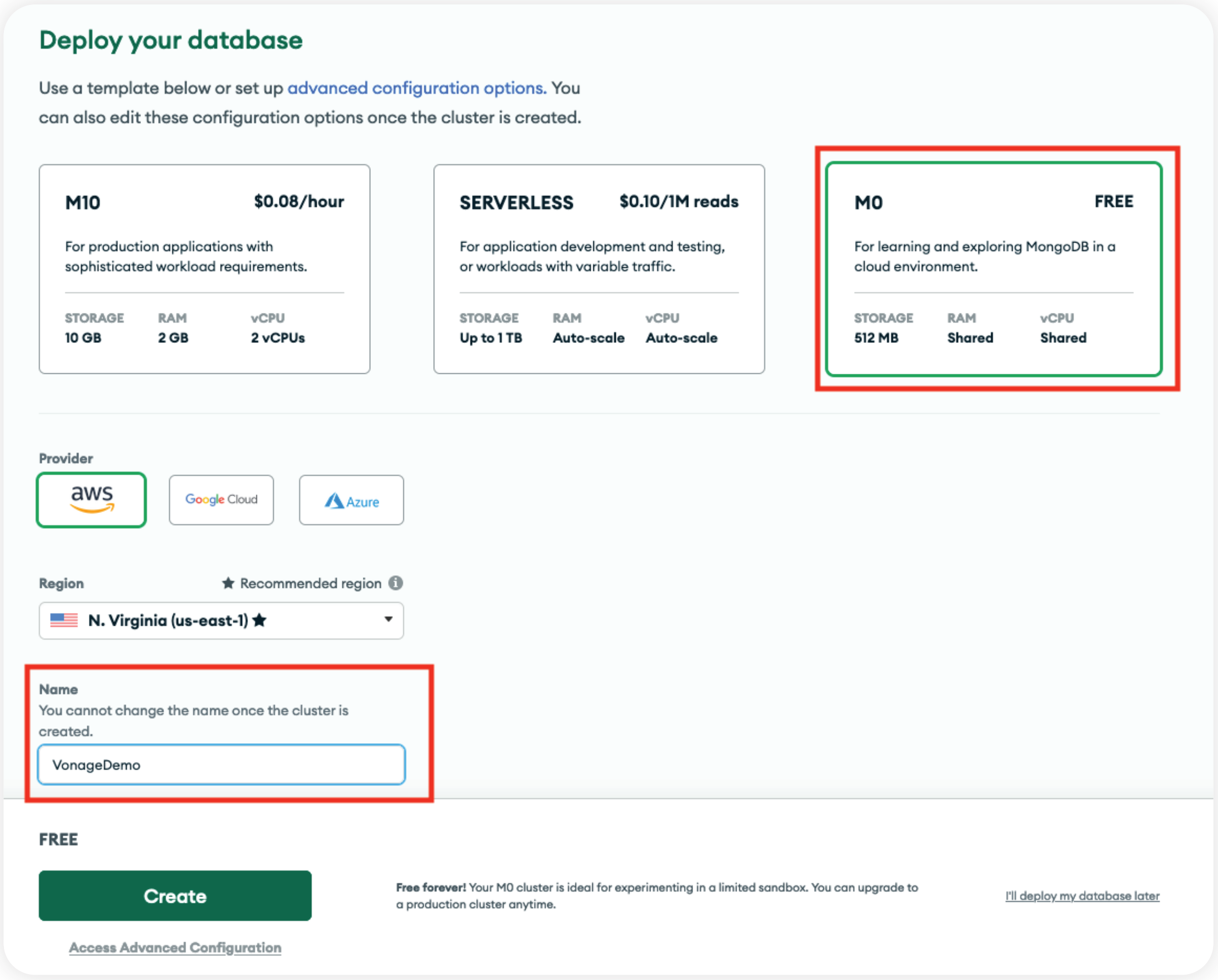
Recording URL HTTP POST http://example.com/Recording

Meetings API Capability

Since we are using the Messages API, we will need to link a telephone number to our application. This will be used for outbound SMS later on. Developer Accounts should have a number already where available, so just click the "Link" button to attach it to this application.

Set up MongoDB Atlas

Now that we have the Vonage side let's set up the database in MongoDB Atlas. When you first log into your account, it will prompt you to deploy your database. We must set up some hosting information as this is a hosted plan. Thankfully the MongoDB Atlas system has a very generous free tier. Just select the M0 free tier to host our database. This is powerful enough for us to play around for our demo. The only other thing you will need to do is add a Name for the database cluster. For this demo, I have just named it *VonageDemo*. If you want, you can change the hosting provider or Region, but for now, you can leave them at the default of "AWS" and "N. Virgina (us-east-1)". Click Create to move on



Deploy your database

Use a template below or set up advanced configuration options. You can also edit these configuration options once the cluster is created.

M10	$0.08/hour	For production applications with sophisticated workload requirements.	STORAGE 10 GB · RAM 2 GB · vCPU 2 vCPUs
SERVERLESS	$0.10/1M reads	For application development and testing, or workloads with variable traffic.	STORAGE Up to 1 TB · RAM Auto-scale · vCPU Auto-scale
M0	FREE	For learning and exploring MongoDB in a cloud environment.	STORAGE 512 MB · RAM Shared · vCPU Shared

Provider: aws (selected) · Google Cloud · Azure

Region: N. Virginia (us-east-1) ★ (★ Recommended region)

Name
You cannot change the name once the cluster is created.
VonageDemo

FREE

Create

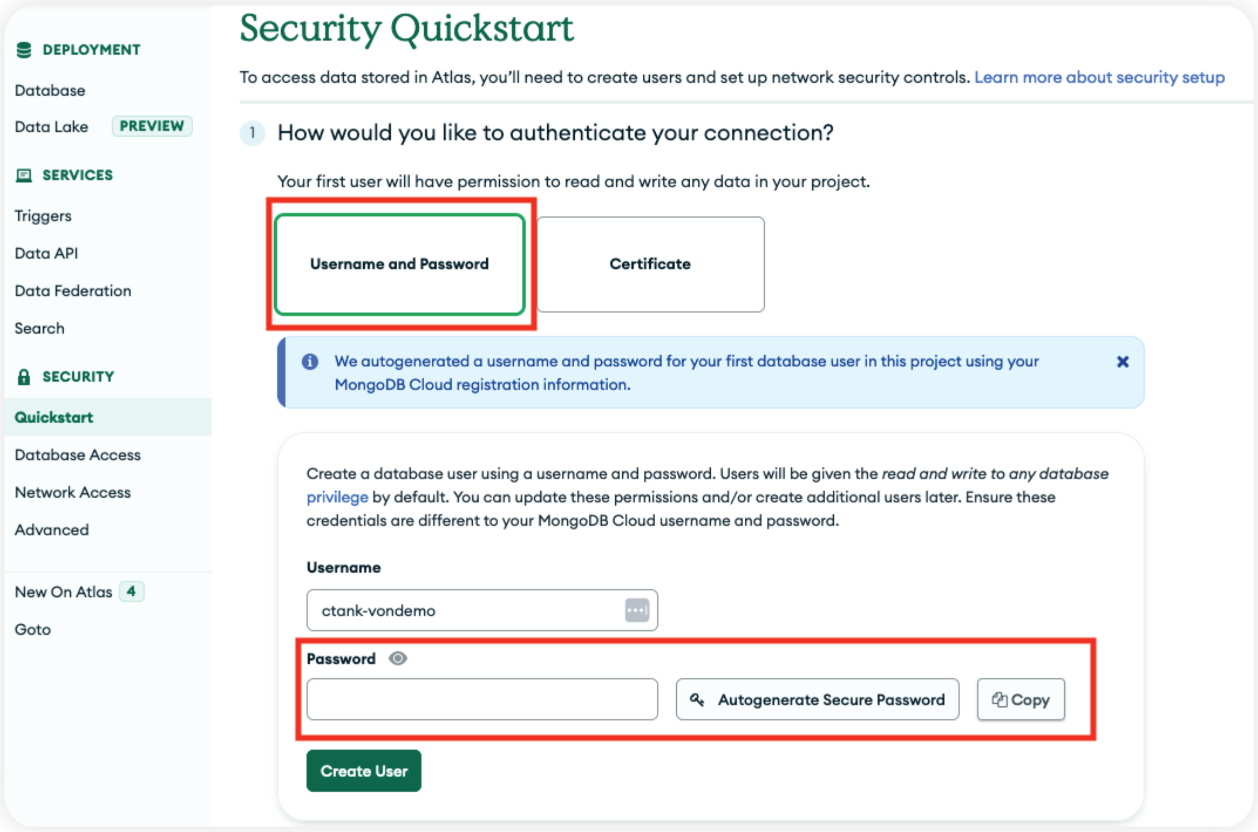
Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

I'll deploy my database later

Access Advanced Configuration

Database Cluster Settings

We will need to set up authentication as we will be accessing the MongoDB cluster over the internet. We can use Username and Password auth for our demo as it is the easiest to get up and running. It will pre-fill a username and password for you. Feel free to change these. Just note down the password for later; we will need that to authenticate to MongoDB. When you are done, click Create User.



Security Quickstart

To access data stored in Atlas, you'll need to create users and set up network security controls. Learn more about security setup

1 How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.

We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information.

Create a database user using a username and password. Users will be given the *read and write to any database* privilege by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username: ctank-vondemo

Password:

MongoDB Cluster Authentication

For security reasons, MongoDB Atlas restricts who can talk to your cluster. For our demo, you can select My Local Environment. The server component of the demo will connect directly to the cluster, so we will need to allow it access to the cluster. By default, it adds your public IP address to the list. This is fine for running the demo locally, but if you are going to deploy this to a public server, you will need to add that server's IP address. If you are hosting the server on another machine, please check with your hosting provider for your public IP address. If you host the demo in a cloud provider like AWS or Google Cloud Platform, you can select Cloud Environment and provide the appropriate details. Click Finish and Close to finish up.

Next, we'll dive into the sample application to see how we can use MongoDB to back our registration process as well as wire in Vonage Verify for additional user security.

What is Vonage Verify API?

Vonage Verify API is a **Two-Factor Authentication** service that Vonage provides as an API. It allows you to make a simple API call to send a code to a user, and then check its validity with another API call. This provides additional security by ensuring that not only does the user know their password, but they have a physical device that they have told us about to receive the code. Since most people have a mobile device, it's pretty safe to assume they will have access to receive the code on it.

Vonage Verify API generates the code and contacts the customer on your behalf. We also will try to contact the customer multiple times in a variety of ways. For example, if you do not try and validate the code from a customer within a certain timeframe, we will try and call the customer with an automated message to provide them with the code. If they still do not enter it, or answer the phone, we will try SMS again. You can control how we try and contact the customer through what are known as Workflows.

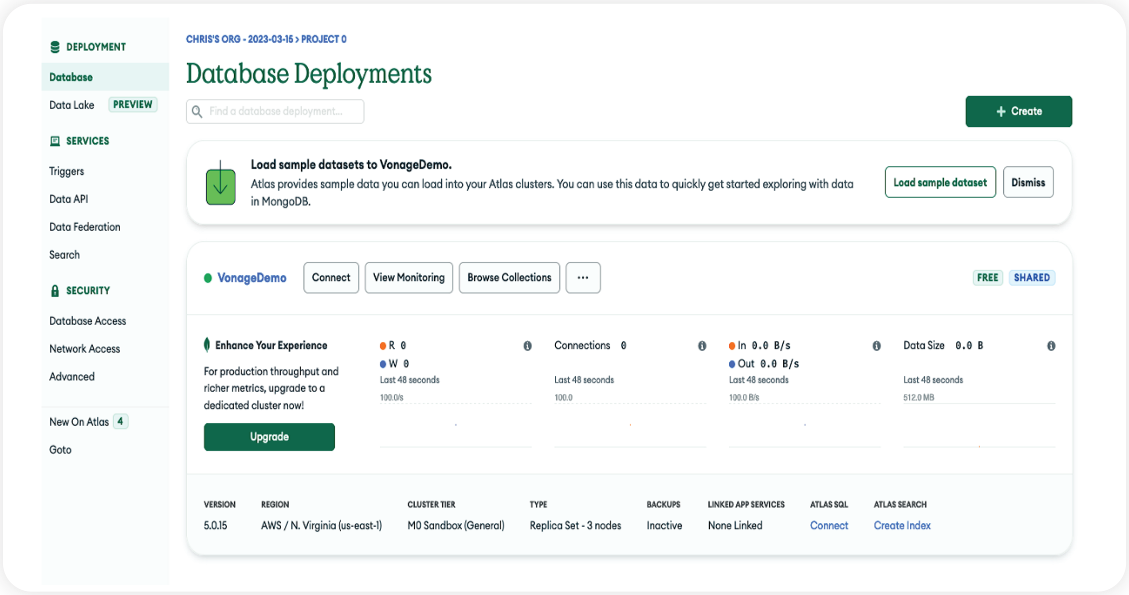
Our standard Verify product supports SMS and Voice to contact the customer. Our newer **Verify V2** product supports SMS, Voice, WhatsApp, Email, and device-based authentication channels with fully customizable workflows. Both products relieve you from having to manually send notifications to your customers and track their responses. nor have to worry about sending complaint messages or getting caught in telephone company spam filters.

You just send an API request to us, we send the code, and you verify it.

Creating a Database

For our demo to work, we need some food for users to order! Adding information into a MongoDB database is a bit different than adding data to a traditional Relational Database Management System like Postgres or MySQL. MongoDB is a document-based system, so instead of creating a database with tables, we will create a database with Collections. These Collections will store documents, which are special JSON-like documents that we can search for and use.

Let's create our first database. From your MongoDB Atlas dashboard, click the Browse Collections button for your cluster.


DEPLOYMENT
Database
Data Lake PREVIEW
SERVICES
Triggers
Data API
Data Federation
Search
SECURITY
Database Access
Network Access
Advanced
New On Atlas 4
Goto

CHRIS'S ORG - 2023-03-15 > PROJECT 0

Database Deployments

Find a database deployment... + Create

Load sample datasets to VonageDemo.
Atlas provides sample data you can load into your Atlas clusters. You can use this data to quickly get started exploring with data in MongoDB.
Load sample dataset | Dismiss

VonageDemo | Connect | View Monitoring | Browse Collections | ··· FREE SHARED

Enhance Your Experience
For production throughput and richer metrics, upgrade to a dedicated cluster now!
Upgrade

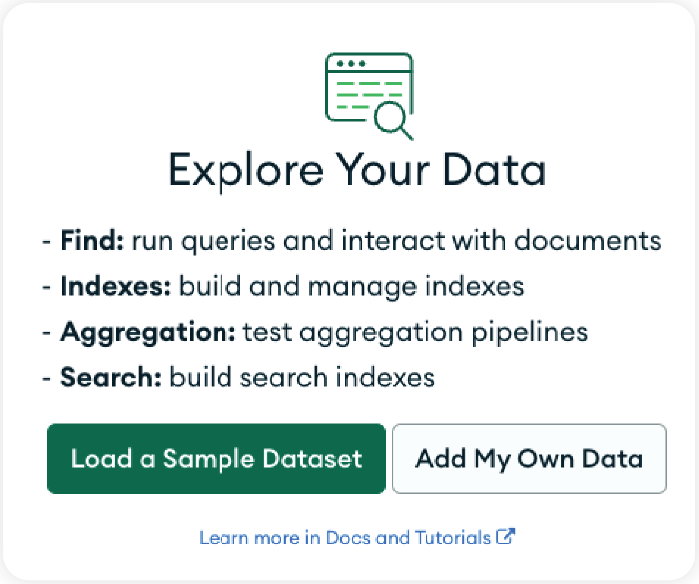
R 0 / W 0 — Last 48 seconds — 100.0/s
Connections 0 — Last 48 seconds — 100.0
In 0.0 B/s / Out 0.0 B/s — Last 48 seconds — 100.0 B/s
Data Size 0.0 B — Last 48 seconds — 512.0 MB

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED APP SERVICES	ATLAS SQL	ATLAS SEARCH
5.0.15	AWS / N. Virginia (us-east-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Connect	Create Index

MongoDB Cluster Authentication

This will bring you to a view of all the information in your cluster. At the moment it is quite bare as we have no databases or information. Let's add a few food items for users to purchase. Click the My Own Data button.

It will then ask for the database name and a collection name. For our demo, we want "restaurant_pos_demo" for the Database name and "inventory" for the Collection name. The demo is already set up to look for this database and collection, so make sure you use these names instead of something custom. Once you have that entered, click the Create button.

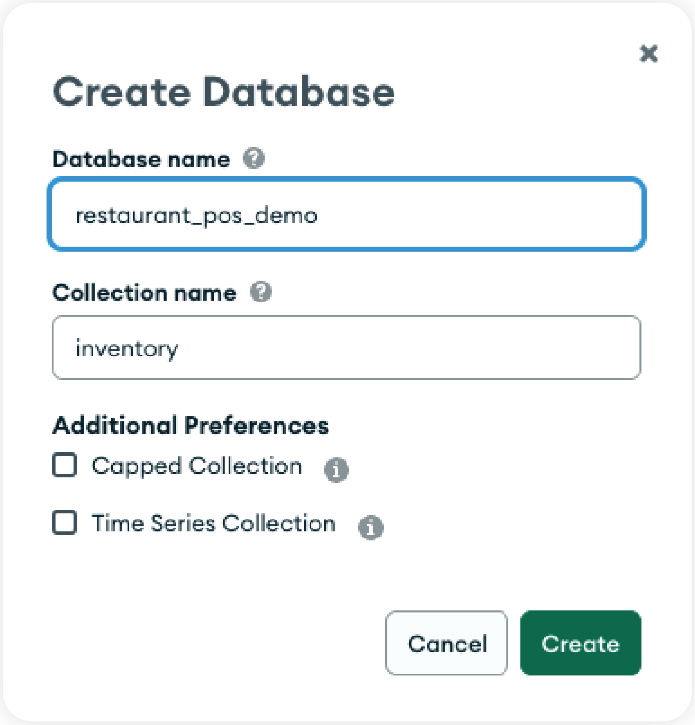

Explore Your Data

- **Find:** run queries and interact with documents
- **Indexes:** build and manage indexes
- **Aggregation:** test aggregation pipelines
- **Search:** build search indexes

Load a Sample Dataset | Add My Own Data

Learn more in Docs and Tutorials

Adding Data


Create Database

Database name: restaurant_pos_demo

Collection name: inventory

Additional Preferences
☐ Capped Collection
☐ Time Series Collection

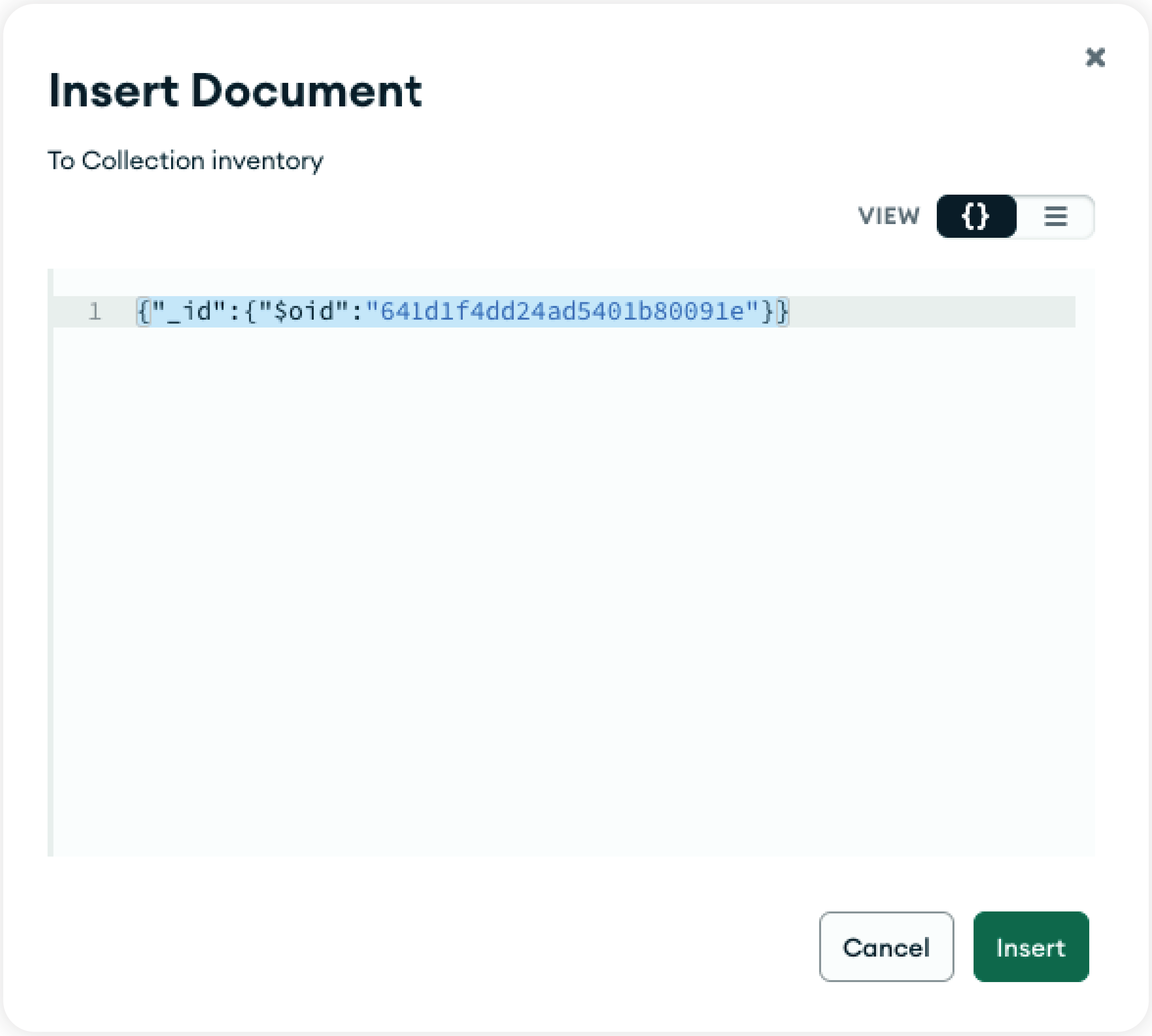
Cancel | Create

Create new Collection

Now we can enter some data. Click the Insert Document button. This will bring up the document editor. While it gives a decent little set of drop-downs for entering information, we can also just paste in some documents. Click the {} button at the top to switch to the text-entry mode, and paste in the small block of JSON for the Hamburger document. Click on Insert, and the inventory item will be saved. Do this again, but the second time paste in the Soda document.

```
// Document 1
{
   "name": "Hamburger",
   "price": 995
}
```

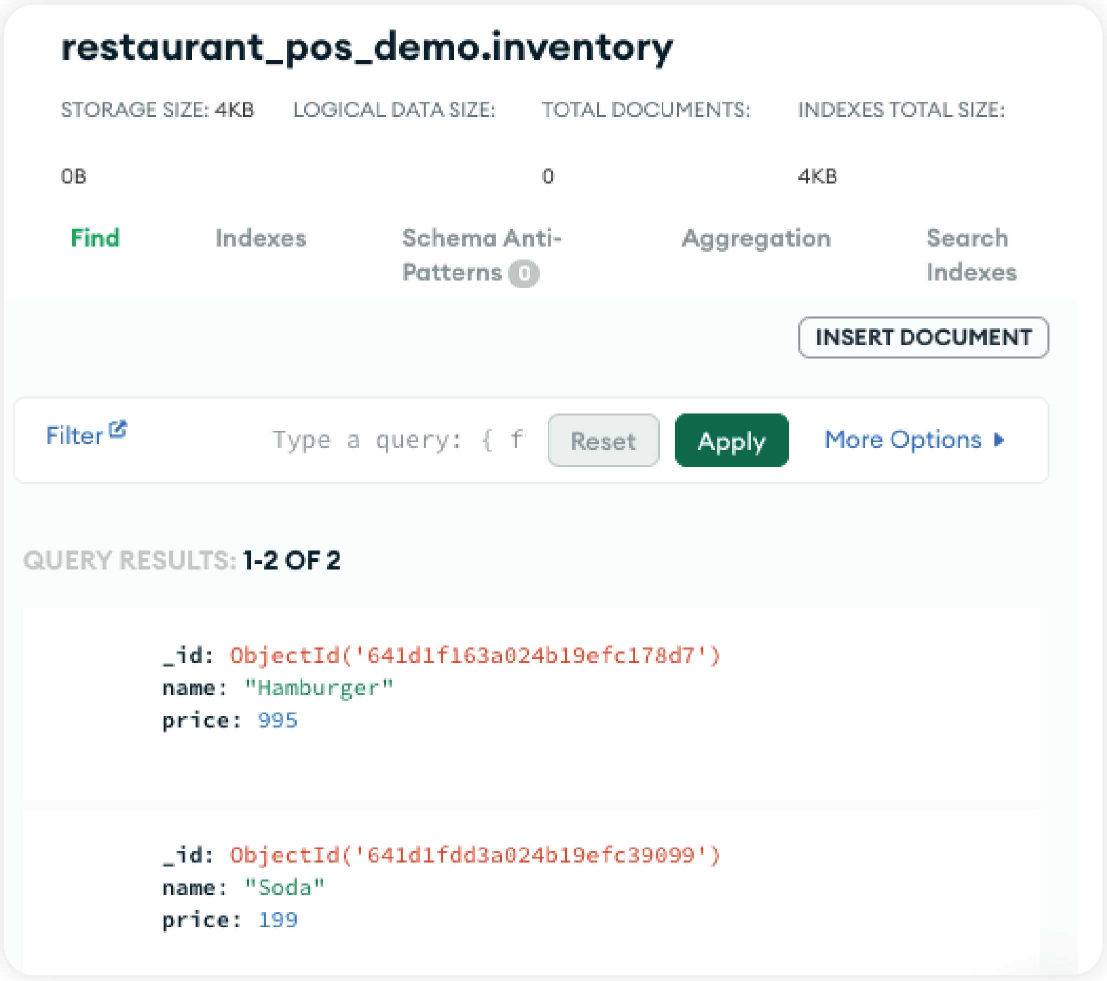
```
/// Document 2
{
   "name": "Soda",
   "price": 199
}
```



```
{"_id":{"$oid":"641d1f4dd24ad5401b80091e"}}
```

Document Editor

Once you have entered the two documents, you can see them in the database view. This editor is a great way to play around with documents and data while you build your database, and can save a lot of time during the development phase debugging data. In a larger production environment, you can run queries to filter out data, but for now, this is a good quick way to enter our data.



```
_id: ObjectId('641d1f163a024b19efc178d7')
name: "Hamburger"
price: 995

_id: ObjectId('641d1fdd3a024b19efc39099')
name: "Soda"
price: 199
```

Documents

Set Up the Demo

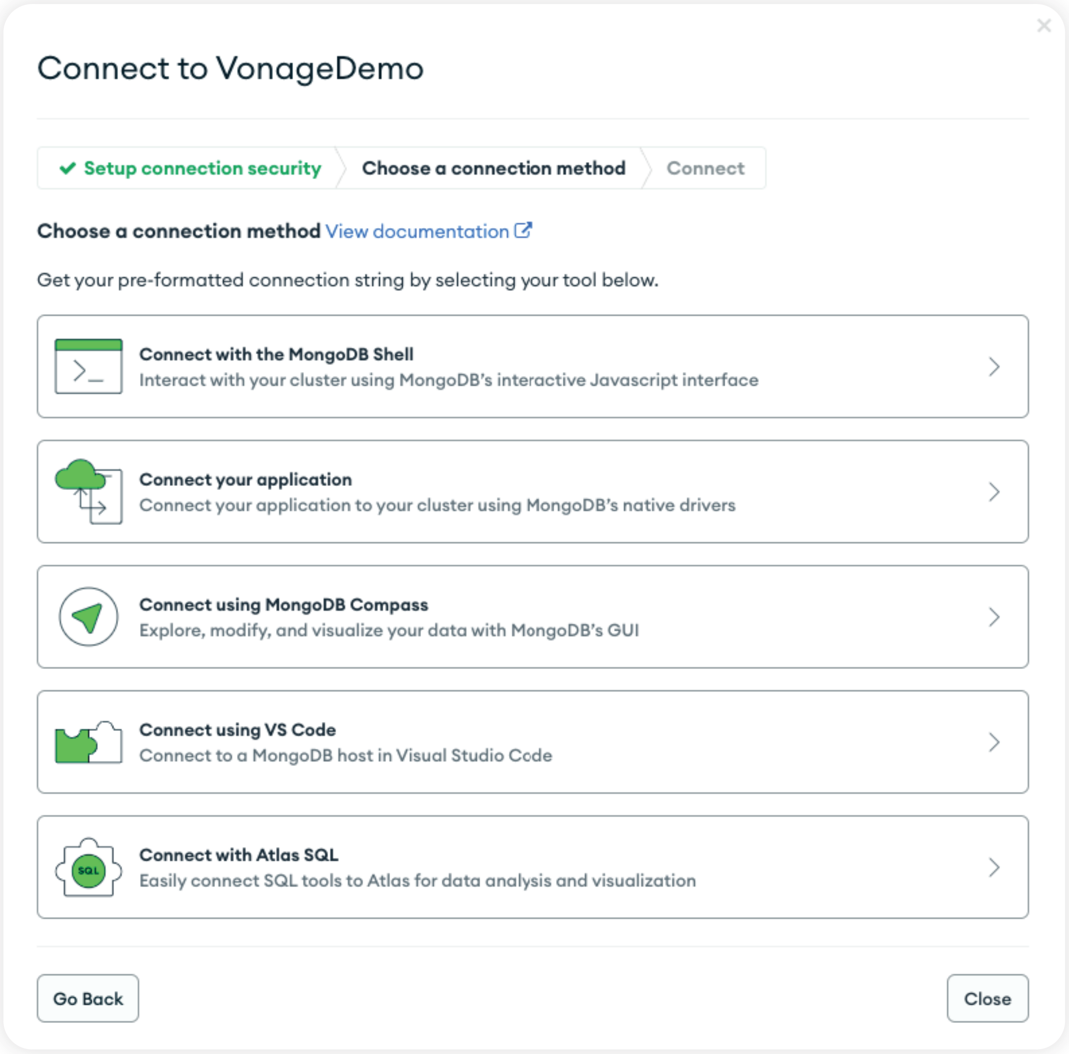
Now that we have some data, we can wire up our demo. Clone the demo from **https://github.com/Vonage-Community/sample-mongodb-vonage-integration-restaurant-demo**. There are two folders, both a "webapp" folder with the source code and an "app-service" folder with some MongoDB configuration we will use later. For now, go into the "webapp" folder and open that up in your favorite editor.

We need to add some configuration details so the application knows how to talk to your MongoDB cluster. Make a copy of the .env.dist file in the repository, and name it .env. This file will have all of the information custom to your install inside of it.

Open up .env and make the following changes:

1. Change "ENABLE_VERIFY" to "1" so that we can see the Vonage Verify API in action
2. Set "VONAGE_API_KEY" to the Vonage API key available on your Vonage Customer Dashboard
3. Set "VONAGE_API_SECRET" to the Vonage API secret available on your Vonage Customer Dashboard
4. Change the "JWT_SIGNING_KEY" to some random value. The string does not really matter, but we will use this later for validating API calls.

We will also need to set the value of "MONGODB_DSN" to the connection string for your cluster. To find this value, go to your MongoDB Atlas dashboard, and click the Connect button for your cluster. In the pop-up, click Connect your application. This will bring you to a screen that has your connection string. Copy that value, and paste it into the value for "MONGODB_DSN" in .env. Make sure to change the <password> part to the password for your cluster.


Connect to VonageDemo

✓ Setup connection security | Choose a connection method | Connect

Choose a connection method View documentation

Get your pre-formatted connection string by selecting your tool below.

Connect with the MongoDB Shell
Interact with your cluster using MongoDB's interactive Javascript interface

Connect your application
Connect your application to your cluster using MongoDB's native drivers

Connect using MongoDB Compass
Explore, modify, and visualize your data with MongoDB's GUI

Connect using VS Code
Connect to a MongoDB host in Visual Studio Code

Connect with Atlas SQL
Easily connect SQL tools to Atlas for data analysis and visualization

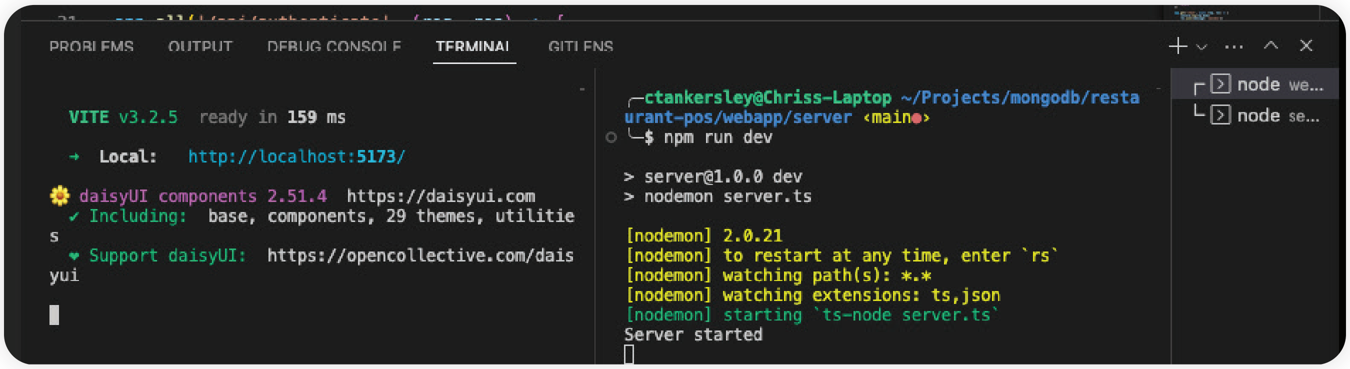
Connecting to the application

Running the Demo

The demo itself is built using Vite, Vue.js, and Typescript. To run the demo, we need to run both the front-end client application and a back-end server application. Open up two command-line terminals.

In the first terminal, in the webapp/ folder run npm ci to install all the dependnencies, and then run npm run dev. If everything goes correctly, you should get a screen saying "Vite <version>" and then a link for Local, probably pointing to http://localhost:5173. Your link may be slightly different if you have other things listening on port 5173.

In the second terminal, navigate to webapp/server. Like the other window run npm ci to install all the dependencies, and then run npm run dev. This screen should show nodemon start and eventually say "Server Started". If you see an error about not being able to connect, check your MongoDB cluster connection string.

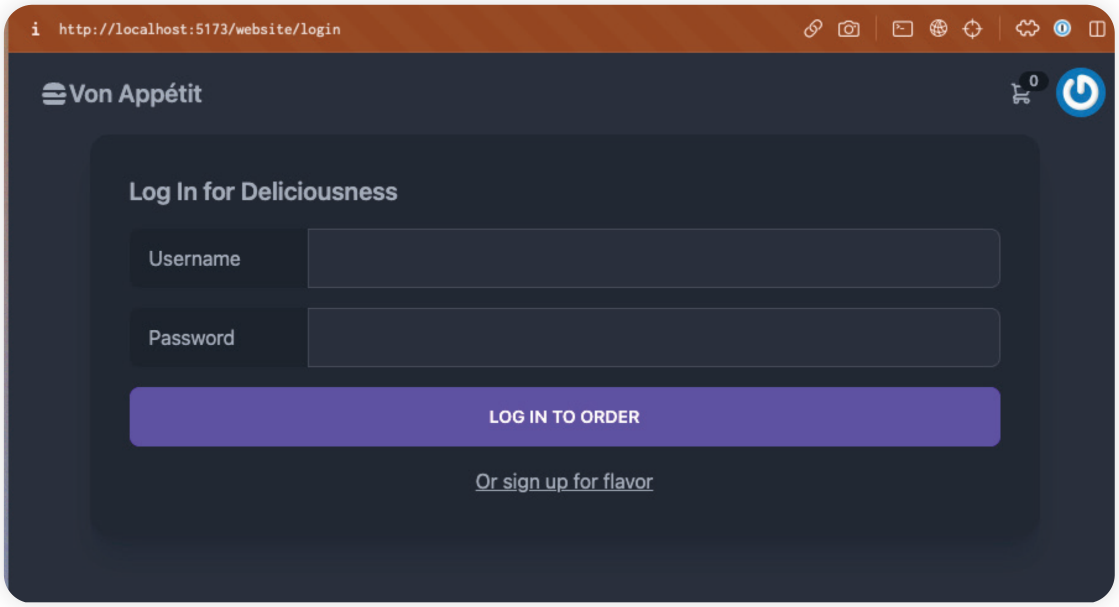


```
  VITE v3.2.5  ready in 159 ms

  ➜  Local:   http://localhost:5173/

🌼 daisyUI components 2.51.4  https://daisyui.com
  ✓ Including:  base, components, 29 themes, utilitie
s
  ❤ Support daisyUI:  https://opencollective.com/dais
yui
```

```
╭─ctankersley@Chriss-Laptop ~/Projects/mongodb/resta
urant-pos/webapp/server ‹main●›
╰─$ npm run dev

> server@1.0.0 dev
> nodemon server.ts

[nodemon] 2.0.21
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node server.ts`
Server started
```

Starting the demo

Open your browser, and navigate to http://localhost:5173/website/login (replace the port number with whatever Vite says it's running for you.). You should be greeted with the following login screen!



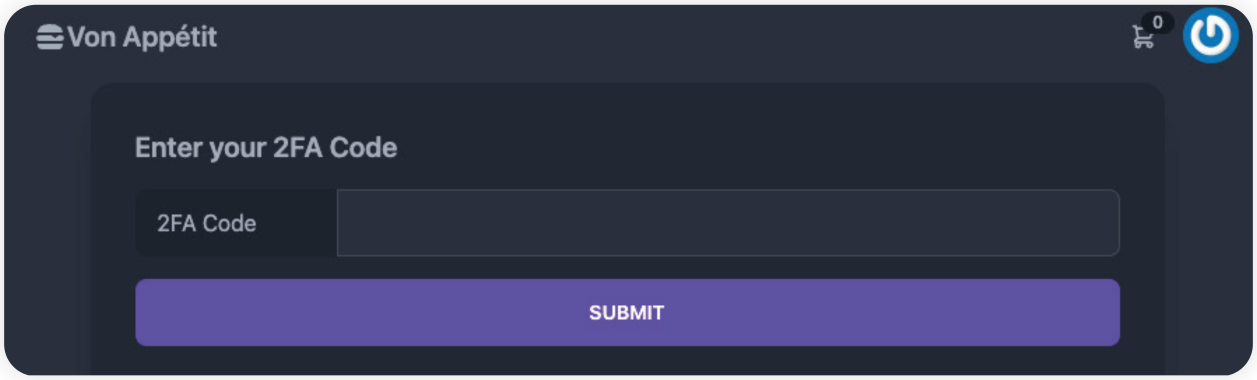
Login Page

Testing out Verify

We currently have no users, so let’s create one. Click the Or sign up for flavor link on the page. Enter a username, password, and mobile telephone number. Your number should include the country code prefix and no dashes. We will send a two-factor authentication code to this mobile number as part of the user login, so make sure to use an actual mobile number, not a Google Voice number. If you are in the US, an example will look like “15556661234”.

Once you have entered your user information click Register.

You should now be able to log in. Enter your username and password that you just registered with. If the authentication was successful, you should be taken to a small form asking you to enter your 2FA Code.



M2FA Form

After a few seconds, you should receive an SMS with a four-digit code. Enter that code into the form and click Submit. If everything works, you will see an order screen with our hamburger and soda!

How does it work?

When the user logs in, our Vue.js client-side application sends the username and password to our backend server, specifically /api/website/authenticate. This route connects directly to our MongoDB cluster and finds the user from a users collection. When we registered a new user, MongoDB automatically created the collection for us and stored a document for the user. We retrieve this document and then compare the password to the stored hashed copy in the document.

The MongoDB Node.js client is a fluent client, which means we can chain together method calls to generate a query. The line:

```
const userRecord = await
client.db('restaurant_pos_demo').collection('users').findOne({
username });
```

Tell the MongoDB client to use our “restaurant_pos_demo” database, search in the “users” collection, and find one document with the "username" that was supplied in the request. Since we stored the password as a bcrypt hash, we can use bcrypt.compare() to check the user-supplied password with the one we stored in the user's document. If they match, the user entered the correct password!

```
// webapp/server/server.ts

app.all('/api/website/authenticate', async (req, res) => {
   const { username, password } = req.body
   const userRecord = await
client.db('restaurant_pos_demo').collection('users').findOne({
username });

   if (userRecord) {
      await bcrypt.compare(password, userRecord.password)
         .then(async (match) => {
            if (match) {
               const token = jwt.sign({user_id: userRecord._id },
process.env.JWT_SIGNING_KEY, { expiresIn: '15m'})
               let verifyId = {request_id: 'abcd'};
               if (process.env.ENABLE_VERIFY === "1") {
                     verifyId = await
vonage.verify.start({number: userRecord.phone, brand:
'Vonage Restaurant'})
                     console.log(verifyId);
               } else {
                      console.log('Verify Disabled');
               }
res.status(200).json({ token, verifyId: verifyId.request_id })
        } else {
              res.status(401).send()
        }
      })
    return
  }

  res.status(401)
  res.send()
  return
})
```

We then generate a temporary JWT to send back to the Vue.js application. Our Vue.js app will use this temporary JWT when the user enters the code on the client-side application. If Verify is enabled in the demo with "ENABLE_VERIFY", we use the **Vonage Node.js SDK** to call the Verify API. We pass the user's telephone number and set the brand to "Vonage Restaurant." When the user receives an SMS message or a voice call, it will be identified as "Vonage Restaurant" when they receive it.

The Vonage Verify API returns a "request ID." We will also send this back to the front-end and use this request ID to check the code from the user. We then send the temporary JWT token and request ID back to the Vue.js app.

Once we verified the user was who they said they were, we changed the Vue.js form to ask for the 2FA code. When the user enters the code, the Vue.js app sends a request to `/api/website/authenticate/verify` with the token, Verify Request ID and the code the user entered.

The JWT contains the user's document ID, so we decode the token and look the user back up in MongoDB. If we find them, we then call the Verify API, but this time we use the `check()` method and send along the request ID and code. The API will return a success if the code matches. If it matches, we generate a real JWT with a longer expiration and return it to the Vue.js application.

```
// webapp/server/server.ts

app.all('/api/website/authenticate/verify', async (req, res) => {
  const { token, verifyId, tfaPin } = req.body
  const decodedToken = jwt.decode(token)
  const userRecord = await
client.db('restaurant_pos_demo').collection('users').findOne({
_id: new ObjectId(decodedToken.user_id) });

  if (userRecord) {
    if (process.env.ENABLE_VERIFY === "1") {
      await vonage.verify.check(verifyId, tfaPin)
        .then(resp => {
          console.log(resp)
          const token = jwt.sign({user_id:
userRecord._id }, process.env.JWT_SIGNING_KEY, { expiresIn: '2h'})
          res.status(200).json({ token })
        })
        .catch(err => {
          console.error("there was an error", err);
        })
      return
    } else {
      const token = jwt.sign({user_id: userRecord._id },
process.env.JWT_SIGNING_KEY, { expiresIn: '2h'})
      res.status(200).json({ token })
    }
  }

  res.status(500)
  res.send()
  return
})
```

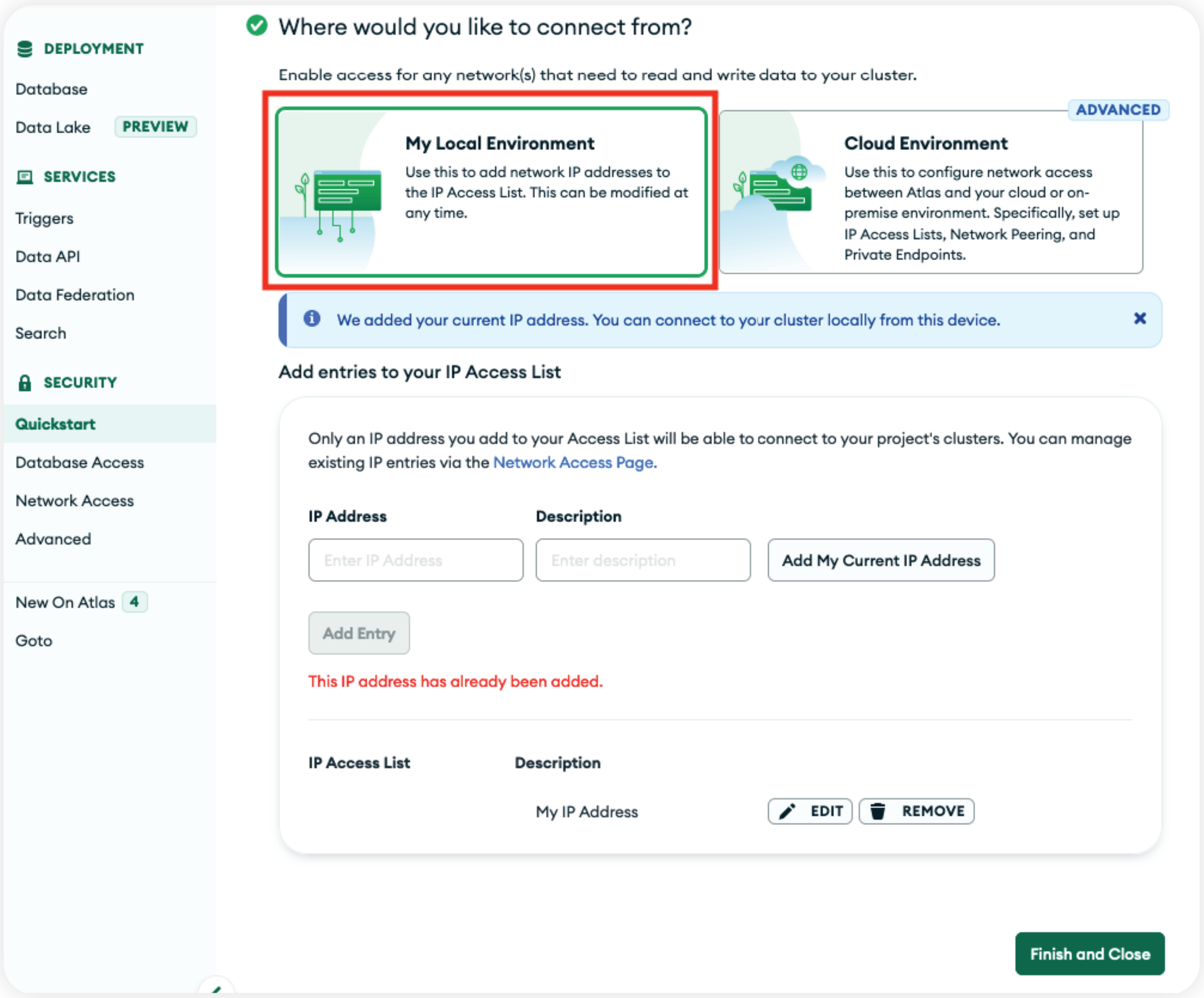
The Vue.js application knows we are fully authenticated once it gets back the proper JWT. It stores this token inside a global store called "authenticationStore", and the rest of the application will use this JWT to authenticate the user for any further API calls.

```
// src/views/Website/Login.vue

const verify = async() => {
  fetch(import.meta.env.VITE_API_URL +
'/api/website/authenticate/verify', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      token: tempJWT.value,
      verifyId,
      tfaPin: tfaPin.value
    })
  })
    .then(resp => resp.json())
    .then(async (json) => {
      console.log(json)
      authStore.setToken(json.token)
      router.push({ name: 'website.order' });

      return
    })
    .catch(err => console.log(err));
}
```

If you already have an authentication step in your application, adding Vonage Verify API is only a few additional lines of code. For our Vue.js app, it meant one additional call to our backend and a new form, and on the server side, we just needed to make the API call to send the code and then a new route to verify the code. Since Vonage handles all the heavy lifting of generating, sending, and checking the code, the impact on our codebase is minimal. The flexibility of MongoDB's document-based storage meant we did not need to run any database migrations and could quickly write the code to insert a new user and do the lookups.

Now that our users can log in, they should order some food!

In the next section, We will look at using MongoDB to store the order and the Vonage SMS API to send an order confirmation. We will also get a peek at using the Vonage Meetings API to quickly add video conferencing to our application for customer service resolutions.



Where would you like to connect from?

Enable access for any network(s) that need to read and write data to your cluster.

My Local Environment
Use this to add network IP addresses to the IP Access List. This can be modified at any time.

ADVANCED
Cloud Environment
Use this to configure network access between Atlas and your cloud or on-premise environment. Specifically, set up IP Access Lists, Network Peering, and Private Endpoints.

We added your current IP address. You can connect to your cluster locally from this device.

Add entries to your IP Access List

Only an IP address you add to your Access List will be able to connect to your project's clusters. You can manage existing IP entries via the Network Access Page.

IP Address	Description	
Enter IP Address	Enter description	Add My Current IP Address

Add Entry

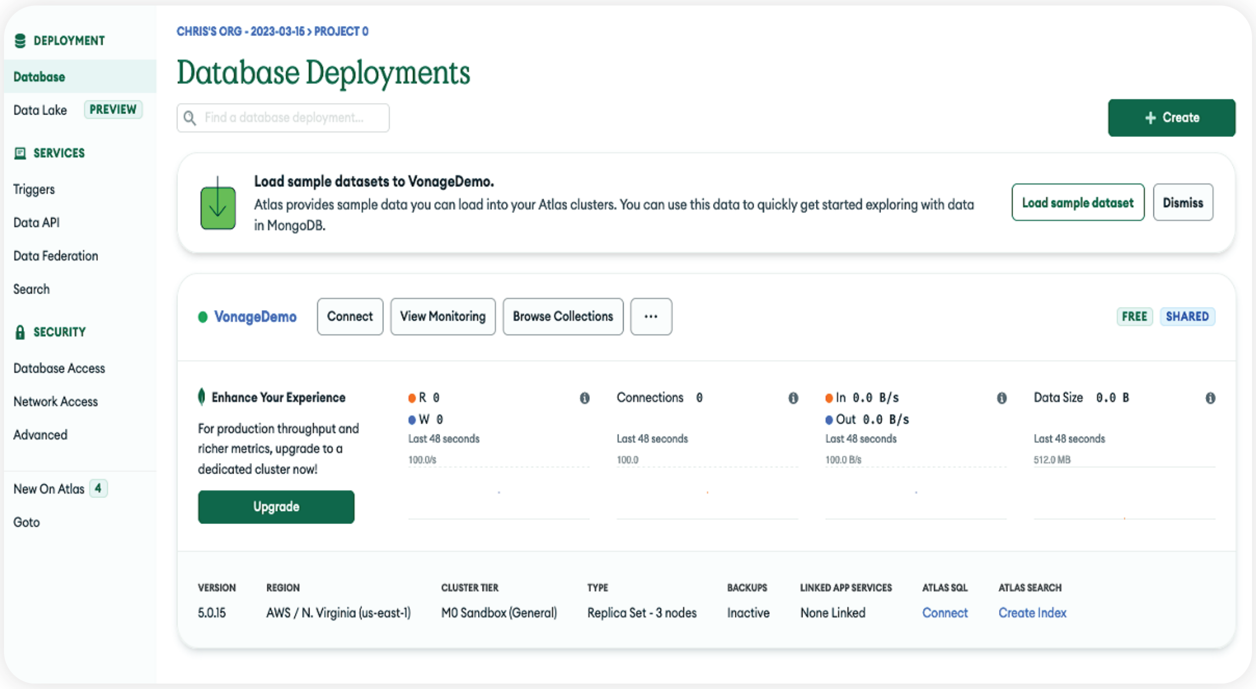
This IP address has already been added.

IP Access List	Description	
	My IP Address	EDIT REMOVE

Finish and Close

MongoDB Security Settings

Your MongoDB Atlas cluster is now all setup! You can administer the cluster through the browser, including viewing the stored documents. The dashboard also has instructions for connecting through the **MongoDB VSCode plugin** to access the database directly in your IDE.



Load sample datasets to VonageDemo.
Atlas provides sample data you can load into your Atlas clusters. You can use this data to quickly get started exploring with data in MongoDB.

Enhance Your Experience
For production throughput and richer metrics, upgrade to a dedicated cluster now!

R 0 / W 0	Connections 0	In 0.0 B/s / Out 0.0 B/s	Data Size 0.0 B
Last 48 seconds	Last 48 seconds	Last 48 seconds	Last 48 seconds
100.0/s	100.0	100.0 B/s	512.0 MB

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED APP SERVICES	ATLAS SQL	ATLAS SEARCH
5.0.15	AWS / N. Virginia (us-east-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Connect	Create Index

MongoDB Dashboard

Let's look at contacting the customer for their order and what we can do when customers need to speak to the restaurant.

How will we do this?

Vonage offers a wide variety of ways for developers to connect to their customers, and one of the simplest ways is through **Vonage Messages API**. This API allows developers to message end users through a variety of channels. At the time of this article, Vonage supports SMS, MMS, WhatsApp, Facebook Messenger, and Viber, but Vonage is continually working on adding more channels. This tutorial will look at sending an SMS, which is usually the easiest way to message a customer. Other channels require more setup and may have additional restrictions.

For the demo, once a user has placed an order, we will send them an SMS notification letting them know that their order has been received. You could expand this in the future to also send notifications on the status of an order or even in-time delivery notifications. Right now, we will send one message so you can see how it is done.

In a perfect world, that would be the last interaction with a customer, but we all know how the world works. What happens if the customer has an issue with the delivery? We could have them call us or even send a text message back with the problem, but what if they could show us the problem? The **Vonage Meetings API** is a quick way to set up a one-to-one video chat without building a video application. We can use it to send a link to the customer and drop them into a pre-built interface, and we barely have to write any code for it.

Sending the Text

Once a user logs in, they should see a Hamburger and a Soda for sale. There is nothing magical going on with this. We have an API endpoint on the server that will query all the available inventory and returns it as a JSON blob. We will then add that to a VueJS variable so that they display.

```
let inventory = ref(Array());

async function getInventory() {
   await fetch(import.meta.env.VITE_API_URL + '/api/inventory')
      .then(resp => resp.json())
      .then(data => {
         inventory.value = []
         data.forEach((dish: {name: string, price: string}) => {
            inventory.value.push(dish)
         })
      })
      .catch(err => console.log(err));
}
```

When the user selects something from the menu, we save that to a VueJS store powered by **Pinia**. Pinia is a plugin for VueJS that makes sharing information across different views easier, so we will store our cart here as we move between the menu page to the order page. If you dug into the authentication code as part 2, you would also see we used it to store the fact the user is authenticated.

Once you select an order and click "Check Out," you will get a confirmation page. Again, nothing is special here as we pull the information from the cart store and display it on the page. The magic happens when we click "Submit Order."

The VueJS code will submit the cart contents to our backend API through a call to fetch(). The server-side code will take our order and save it into MongoDB as a new document in the orders collection.

```
const { items } = req.body
const bearerToken = req.header('authorization').split(' ')[1]
const decodedToken = jwt.decode(bearerToken);
const userRecord = await
client.db('restaurant_pos_demo').collection('users').findOne({
_id: new ObjectId(decodedToken.user_id) });

const orderTime = new Date().toISOString()
const result = await
client.db('restaurant_pos_demo').collection('orders').inser-
tOne({
   items, orderTime, status: 0, lastUpdated: orderTime, user_id:
userRecord._id
});
```

If you are coming from a relational database background, you may notice that we take the items that were sent from the order and just put them into the new order document. We are storing all the relevant item and order information in this document instead of denormalizing the data (where we would rather store just the item ID to link it to the inventory collection). Document-based databases keep all the needed information within the document instead of using foreign keys to reference other documents and collections. You can, as each document has an ID, but it is common.

This is one of the significant advantages of Document-based databases. All of the information for a document can be stored within the document instead, reducing the number of external lookups that need to be done. You may use a series of JOIN operands in a relational database to piece together a row of information from various tables. Still, in MongoDB, this is accomplished through **aggregate pipelines**.

Aggregate pipelines allow you to select and manipulate documents through a series of queries and pipe those results into other aggregate queries. While we are not using them in this example, as we are just storing the inventory information in the order document, you can do quite complex data manipulation with aggregations.

Once the order is saved, we fire an SMS message through the Messages API. Since we are using our **Node.js SDK** is just a single call to vonage.messages.send(). We pass in an SMS object with the text message, the number to send to, and the number we have linked to our application (which we set up in Part 1 and have in our .env file).

```
await vonage.messages.send(
  new SMS(
    'Your order has been submitted',
    userRecord.phone,
    process.env.VONAGE_FROM
  )
);
```

That is all it takes to send an SMS through our Messages API! The user should get a text message on their mobile device in just a few minutes.

Vonage Messages API vs Vonage SMS API

If you have dug around in our **developer documentation**, you may notice that we have two APIs for sending SMS messages. One is the Messages API we just discussed, and the other is our **SMS API**. Why do we have two APIs for the same thing?

The SMS API is one of the original APIs provided by Vonage and was built when SMS was the only text message option. As such, it is purpose-built for not only basic "Send an SMS through an HTTP API" but also more advanced SMS interactions like the **SMPP** protocol, or Short Message Peer-to-Peer protocol. SMPP is a telecom industry protocol that allows a more direct message exchange between applications and providers like Vonage.

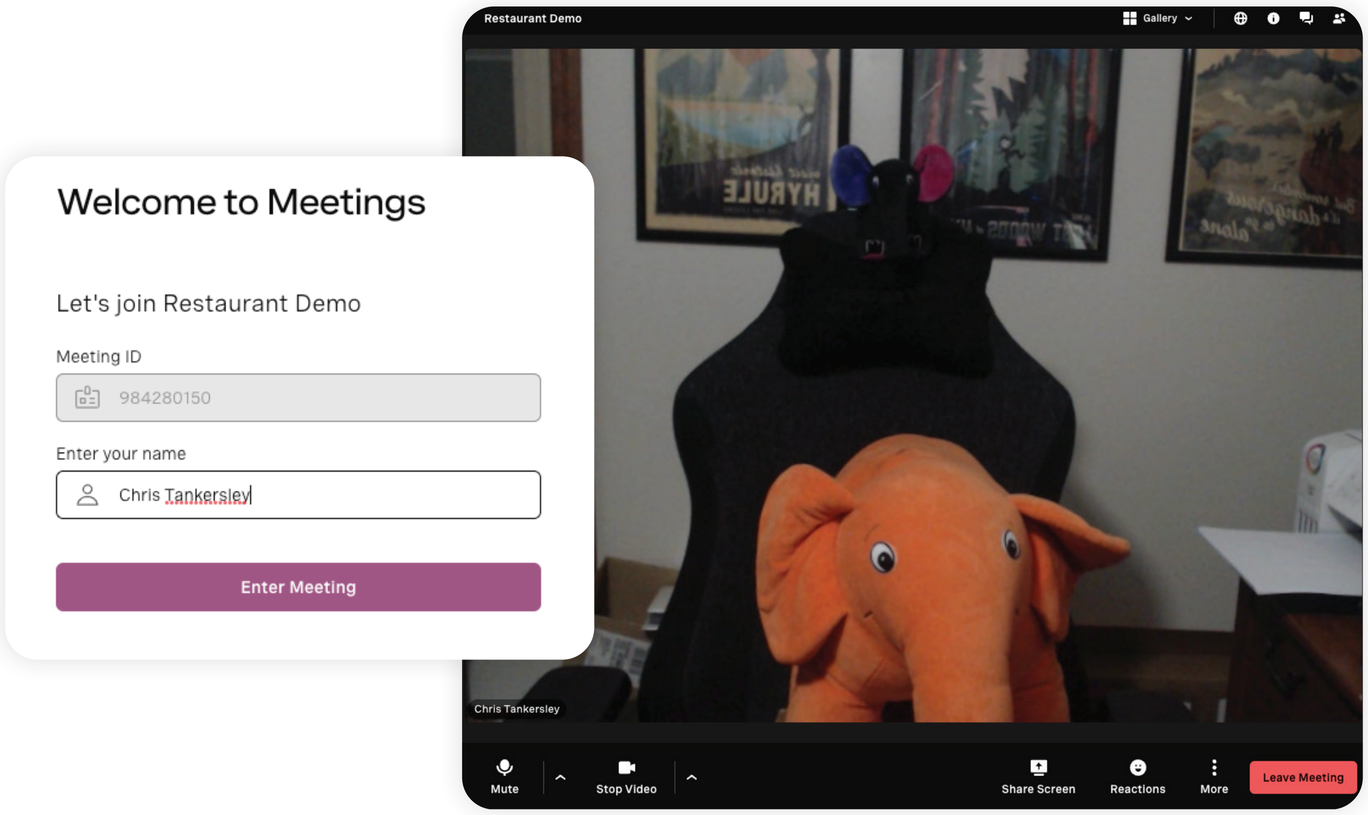
The Messages API is designed for more day-to-day users. It takes the ease-of-use of the original SMS API and extends it to more channels like MMS and WhatsApp. Since it focuses on more general usage, it does not have SMPP access.

We recommend using the Messages API for any new projects. SMS and Messages must still abide by **country-specific SMS restrictions** like 10DLC in the US, so unless you specifically need very low-level SMS sending like SMPP the Messages API is a better choice.

Houston, We Have a Problem

Once the user submits their order, they are brought to an Order Status screen. This displays the order number returned from the MongoDB record we added and could be extended to show the order items themselves. We want to look at the "Video Call" button now, as this is a way for the customer to contact the restaurant.

From the end-user perspective, they can click this button, and a new window will open into a video call. They will enter a meeting room with a nice visual theme, the ability to turn their camera and mic on and off, and all the comforts you would want for a video call. The best part is this works with all major browsers on both desktop and mobile devices.



The Meetings API is still in beta at this article's time, but setting it up is incredibly easy. The first thing we are going to do is set up a theme. This can be done before and only needs to be done once, but you can **create a theme** with your companies logo, color scheme, and branding. To set up a new theme, it takes a single API call.

```
const privateKey =
readFileSync(process.env.VONAGE_PRIVATE_KEY);
const token =
tokenGenerate(process.env.VONAGE_APPLICATION_ID, privateKey);
await
fetch('https://api-eu.vonage.com/beta/meetings/themes', {
   method: 'POST',
   body: JSON.stringify({
      theme_name: 'Restaurant Theme',
      main_color: '#a05683',
      brand_text: 'Vonage Restaurant',
      short_company_url: 'my-restaurant'
   }),
   headers: {
          'Authorization': 'Bearer ' + token,
          'Content-Type': 'application/json'
   }
})
   .then(resp => resp.json())
   .then((data: any) => {
         res.json(data)
   })
   .catch(err => console.error(err))
```

Since we are not using the SDK, we will use the tokenGenerate() method from @vonage/jwt to create a JWT token to talk to the API. We then make a POST call to the Meetings API with our theme name, color, and other information. Check out the **Vonage Meetings API Reference** for all the options. This API call will return a theme ID we will use later in the demo.

Once we have the meeting ID, we send it back to the client so it can be used to open a new window.

```
app.post('/api/website/video-call', async (req, res) => {
   const { orderNumber } = req.body;
   const privateKey =
readFileSync(process.env.VONAGE_PRIVATE_KEY);
   const token =
tokenGenerate(process.env.VONAGE_APPLICATION_ID, privateKey);

   fetch('https://api-eu.vonage.com/beta/meetings/rooms', {
      method: 'POST',
      body: JSON.stringify({
         display_name: 'Restaurant Demo',
         type: 'instant',
         theme_id:
'6ba90e1b-c27a-45e8-9e49-877634c315b0'
      }),
      headers: {
         'Authorization': 'Bearer ' + token,
         'Content-Type': 'application/json'
      }
   })
      .then(resp => resp.json())
      .then(async (data: any) => {
         console.log('guest url: ' + data._links.guest_url.href)
         console.log('host url: ' + data._links.host_url.href)
         const orderRecord = await client.db('restaurant_pos_
demo').collection('orders').updateOne({ _id: new ObjectId(order-
Number) }, { $set: { meetingUrl: data._links.host_url.href}})
            .then(async (document) => {
               res.json({
                  guest_url: data._links.guest_url.href
               })
            });
      })
      .catch(err => console.error(err))
});
```

A single API call is all we need to add to our application to add video conferencing to our application. We did not have to do anything to set up the UI for the video room, and it all uses the WebRTC standard to work on almost any device.

Let's look at using MongoDB Atlas's user authentication, allowing us to offload our user authentication to Atlas for our administrative backend.

Offloading User Authentication

One common area that web applications share is the need to authenticate users. Frameworks help handle some of this, but each application builds similar code to do one thing - confirm a user's credentials. We can use MongoDB Atlas has a built-in system to **authenticate and manage users**.

This system is different from the authentication we do to MongoDB and is a service Atlas provides. You can manage users through Atlas as a third-party (to your application) authentication service. Using Atlas allows you to support many different authentication types securely.

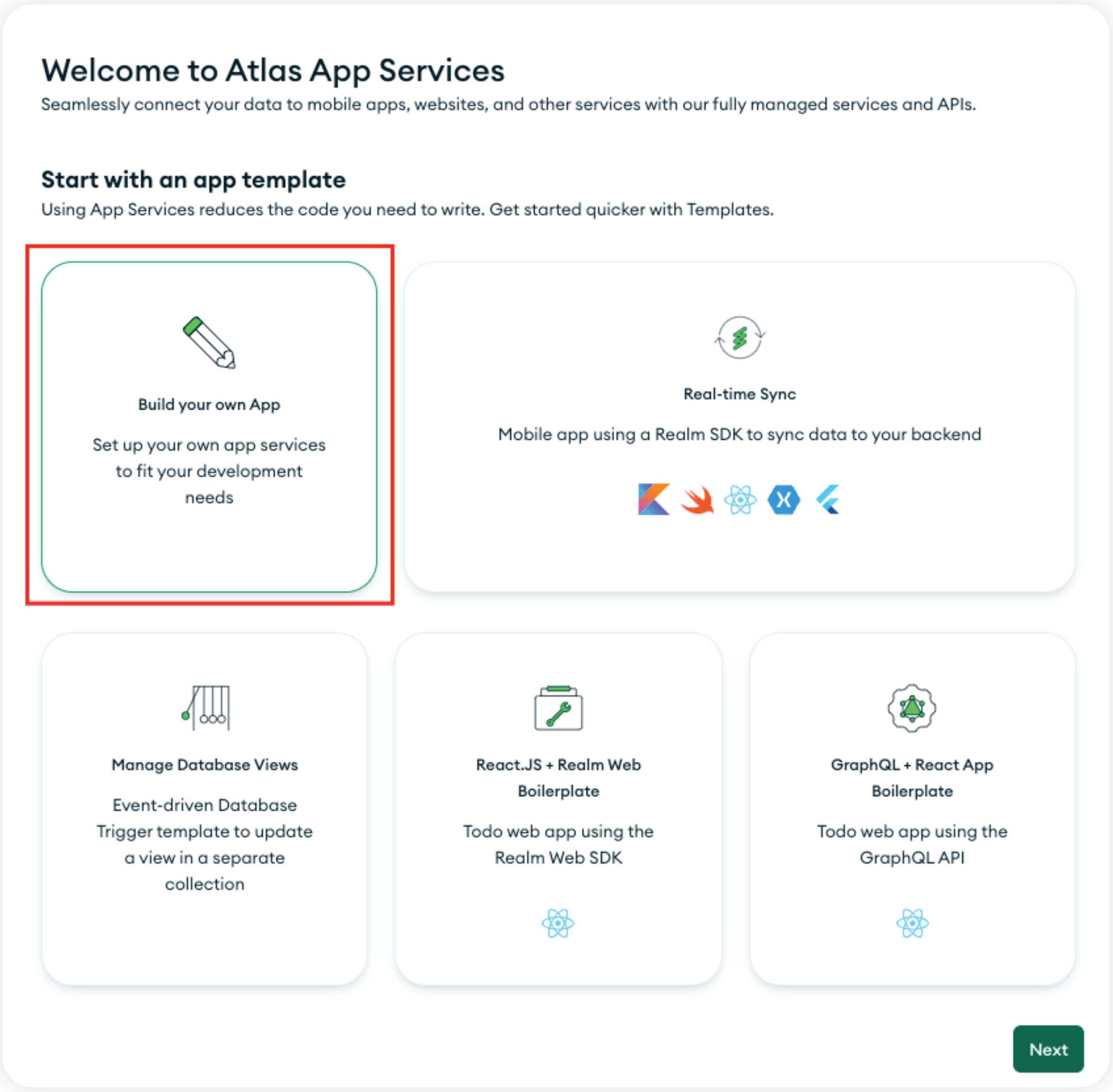
To show this off, the administrative backend of our demo uses Atlas authentication instead of Verify. This backend will allow us to manage the inventory that we show users and orders that have come in. It will also allow us to join any video meetings customers have started. As a bonus, it will enable us to see what happens if we want to embed MongoDB access into our application instead of relying on a backend API.

Setting up Authentication

Atlas supports both a web UI and configuration files for many application-centric features. We will use the web UI for the tutorial for configuration, but you can also use the supplied sample files in the demo repository. These files work with the **Realm CLI** tool, and we have provided them for you to compare to the web UI. If you are just getting started, I recommend using the web UI, but in a managed application, you will want to store the configuration and use the Realm CLI to deploy config changes. The demo includes an `app-service/` folder with sample files you can edit to get started.

Atlas Apps are a combination of configuration details and deployed code. The Apps interface makes it easy to work with the settings and code a developer has offloaded to Atlas services.

For now, let's use the web UI. Once you log into your project, click "App Services" in the top secondary navigation bar. This will bring up a list of configured application projects. If this is the first application you are working with, a window will pop up, taking you to the App Services. Select "Build your own app" for now, as we will handle everything for the tutorial.



Welcome to Atlas App Services

Seamlessly connect your data to mobile apps, websites, and other services with our fully managed services and APIs.

Start with an app template

Using App Services reduces the code you need to write. Get started quicker with Templates.

Build your own App
Set up your own app services to fit your development needs

Real-time Sync
Mobile app using a Realm SDK to sync data to your backend

Manage Database Views
Event-driven Database Trigger template to update a view in a separate collection

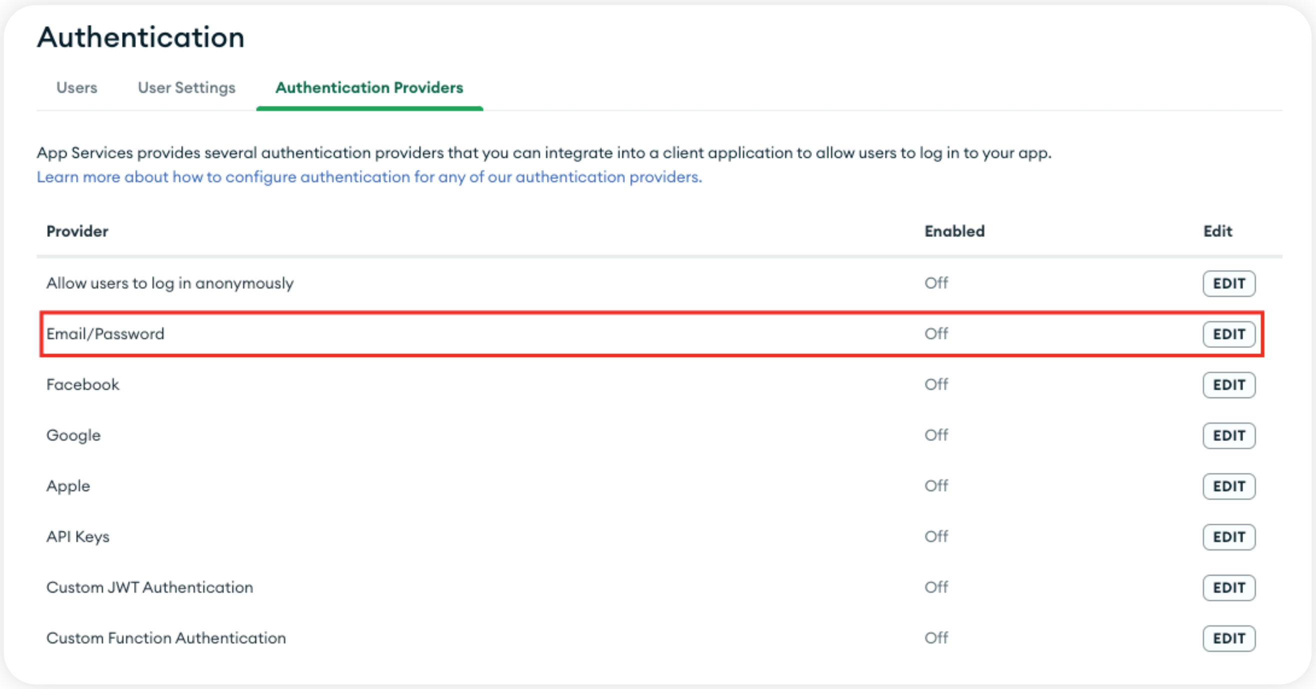
React.JS + Realm Web Boilerplate
Todo web app using the Realm Web SDK

GraphQL + React App Boilerplate
Todo web app using the GraphQL API

New Atlas App dialog

The next screen will have some configuration questions. Our Data Source, which is the cluster we use, should be filled in. Select the one you use for the tutorial if you have multiple clusters. You can also change the name of the application. I will name the application “Frontend” as this application service will handle our JavaScript frontend for the admin pages. Click Create App Service to continue, and then Close Guides to close the jumpstart window.

This brings us to the Apps dashboard for our Frontend application. As you can see, you can do many things with an Atlas App, but for now, we are focused on using user authentication. Under the "Data Access" section of the sidebar, click on Authentication so we can start to set it up.



Authentication

Users | User Settings | **Authentication Providers**

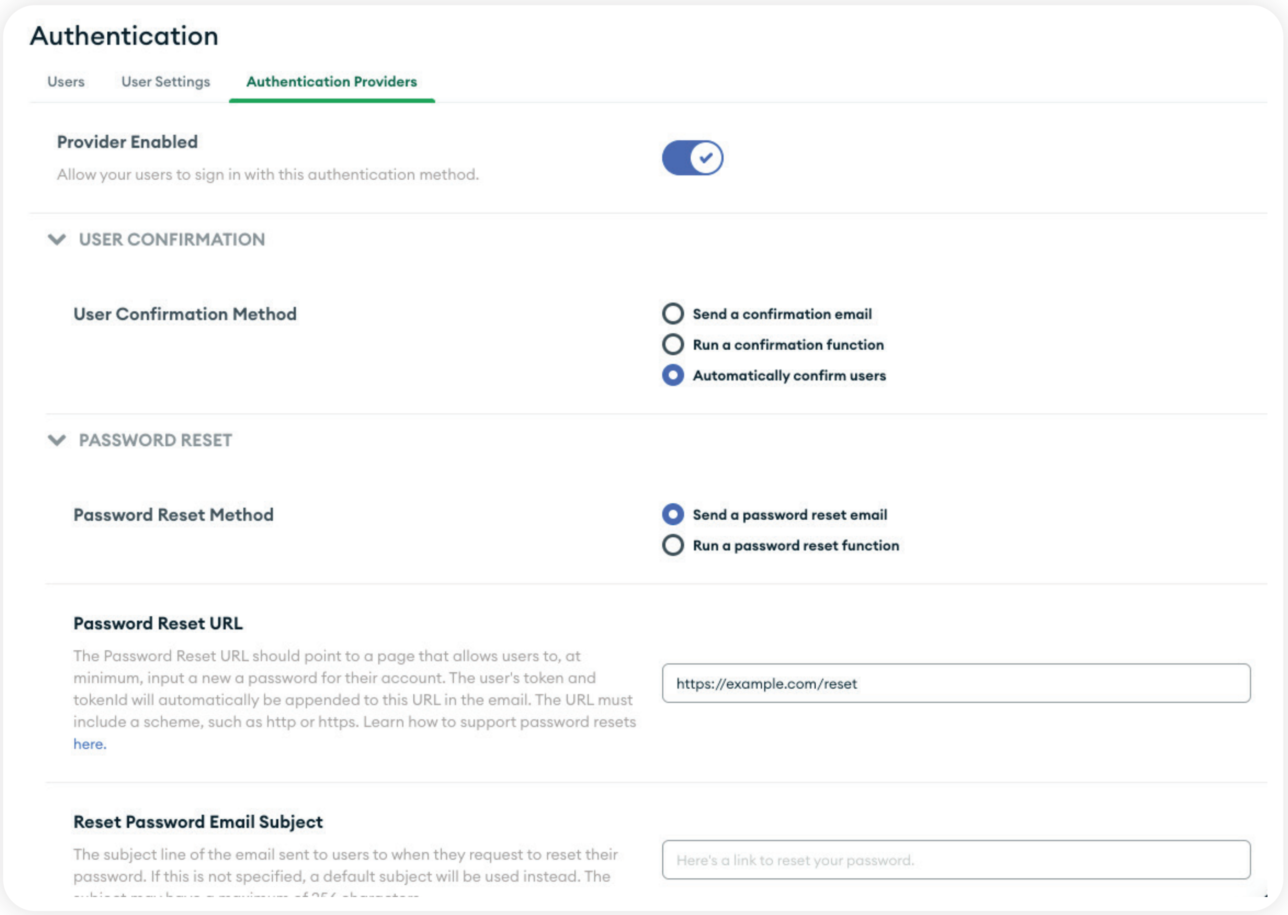
App Services provides several authentication providers that you can integrate into a client application to allow users to log in to your app.
Learn more about how to configure authentication for any of our authentication providers.

Provider	Enabled	Edit
Allow users to log in anonymously	Off	EDIT
Email/Password	Off	EDIT
Facebook	Off	EDIT
Google	Off	EDIT
Apple	Off	EDIT
API Keys	Off	EDIT
Custom JWT Authentication	Off	EDIT
Custom Function Authentication	Off	EDIT

User authentication options

As mentioned before, Atlas supports multiple types of authentication. For now, we will only worry about “Email/Password.” Click on the Edit button to start setting it up.

On the configuration page, toggle “Provider Enabled” to on. For the tutorial, we will automatically confirm new users, so go ahead and also select “Automatically confirm users.” In a production application, you will want the user to verify their email to validate that the email address exists, but we can skip that step for now. While we will not go over implementing it, you must enter a “Password Reset URL.” For now, enter "https://example.com/reset" to satisfy the form. Click on Save Draft when you are finished.


Authentication

Users | User Settings | Authentication Providers

Provider Enabled [✓]
Allow your users to sign in with this authentication method.

USER CONFIRMATION

User Confirmation Method: Send a confirmation email / Run a confirmation function / [✓ Automatically confirm users]

PASSWORD RESET

Password Reset Method: [✓ Send a password reset email] / Run a password reset function

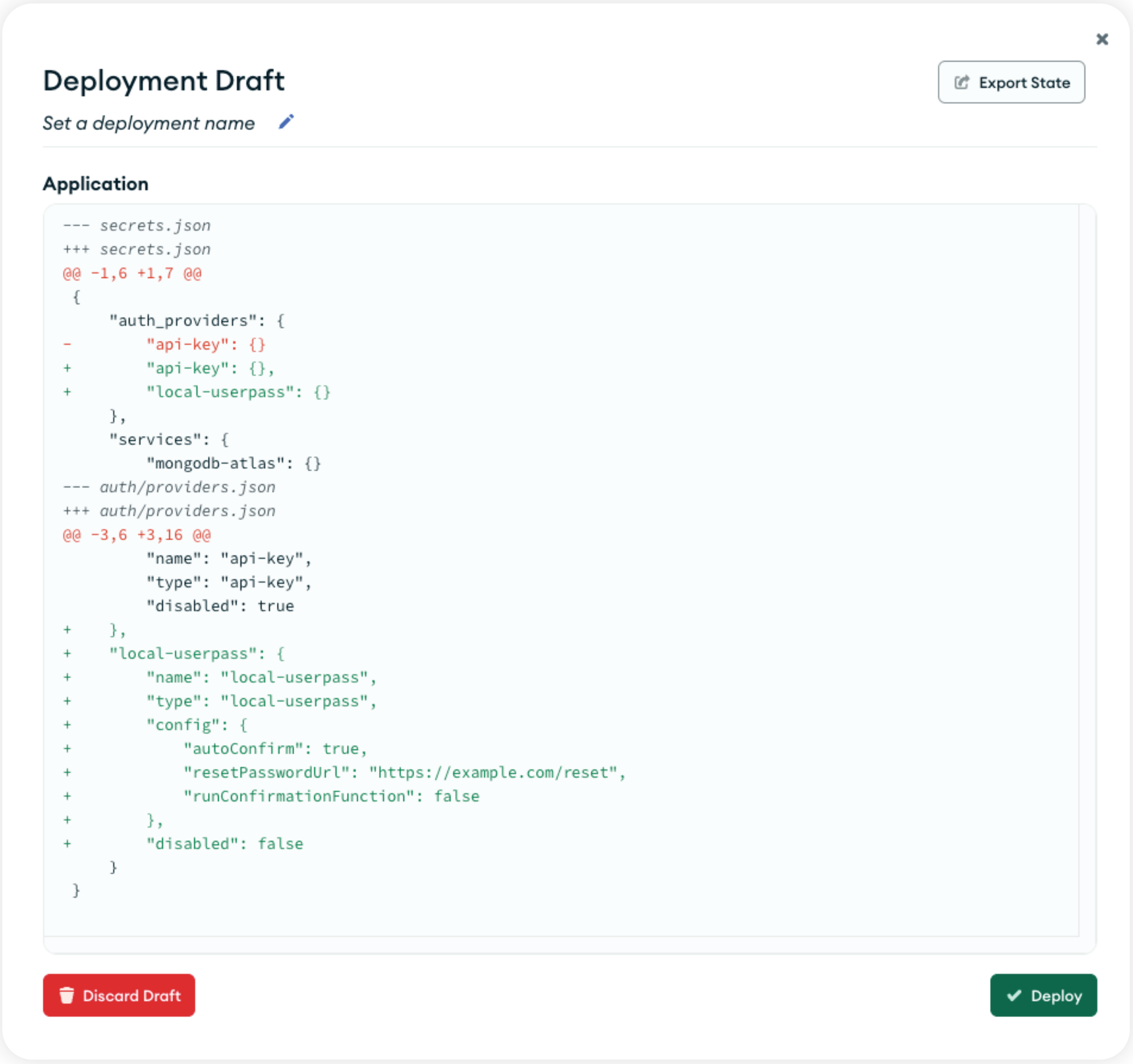
Password Reset URL
The Password Reset URL should point to a page that allows users to, at minimum, input a new a password for their account. The user's token and tokenId will automatically be appended to this URL in the email. The URL must include a scheme, such as http or https. Learn how to support password resets here.
https://example.com/reset

Reset Password Email Subject
The subject line of the email sent to users to when they request to reset their password. If this is not specified, a default subject will be used instead. The
Here's a link to reset your password.

Email/Password options

Wait, "Save Draft?!" If you skip past the panel that comes up, any changes you make in Atlas are considered Draft changes. You can stage a set of different draft stages and deploy them when everything is set up. All of this information is saved in configuration files that can be pushed and pulled using the Realm CLI, and the files mentioned above are stored in `app-service/` as examples.

Once you've made a change, you will see a banner along the top of the page that now says "Changes have been made" with a button to review. Go ahead and click on Review Draft & Deploy. You will see a JSON blob that is a text diff between the old and new settings. This will look very familiar if you have used GitHub's pull request system. Since we just made this change, click on Deploy. These settings will be pushed out to the app service, and we can start to use the authentication.



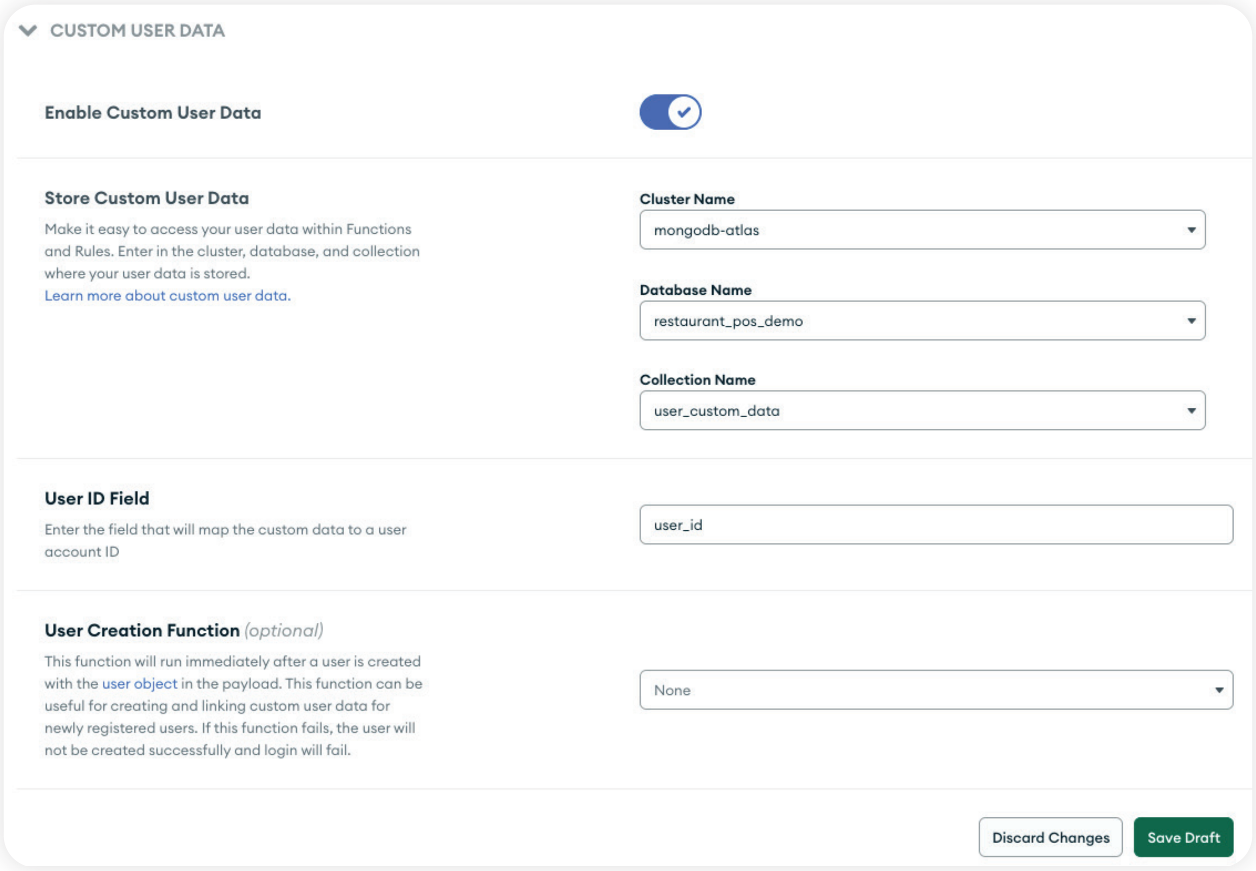
```
--- secrets.json
+++ secrets.json
@@ -1,6 +1,7 @@
 {
     "auth_providers": {
-        "api-key": {}
+        "api-key": {},
+        "local-userpass": {}
     },
     "services": {
         "mongodb-atlas": {}
--- auth/providers.json
+++ auth/providers.json
@@ -3,6 +3,16 @@
         "name": "api-key",
         "type": "api-key",
         "disabled": true
+    },
+    "local-userpass": {
+        "name": "local-userpass",
+        "type": "local-userpass",
+        "config": {
+            "autoConfirm": true,
+            "resetPasswordUrl": "https://example.com/reset",
+            "runConfirmationFunction": false
+        },
+        "disabled": false
     }
 }
```

Deployment Diff dialog

Now we need a user. Click on App Users in the sidebar, and then the Add New User button. Fill in a valid e-mail address and password, then click Create. Creating users like this will not scale, so there are options to create users programmatically through a signup process, but for now, we will use one we make by hand.

At this point, authentication is configured for our application. We could use the MongoDB Realm SDK to authenticate a user, but our current user is nothing more than an e-mail address and password. We cannot store extra information or denote that the user is an administrative user. This is where Custom User Data comes in. We can link a User to a document collection that will house additional user content, like Name, Phone, or even if they are flagged as an admin.

Click on User Settings. This will bring up the configuration page for our user data linking.


CUSTOM USER DATA

Enable Custom User Data ✓

Store Custom User Data

Make it easy to access your user data within Functions and Rules. Enter in the cluster, database, and collection where your user data is stored.
Learn more about custom user data.

Cluster Name: mongodb-atlas

Database Name: restaurant_pos_demo

Collection Name: user_custom_data

User ID Field

Enter the field that will map the custom data to a user account ID

user_id

User Creation Function *(optional)*

This function will run immediately after a user is created with the user object in the payload. This function can be useful for creating and linking custom user data for newly registered users. If this function fails, the user will not be created successfully and login will fail.

None

Custom User Information settings

Toggle "Enable Custom User Data." Then select your cluster and database from the dropdown menus for "Cluster Name" and "Database Name," respectively. For the "Collection Name," select "Create new Collection." This will make an additional text box appear. In this new box, enter user_custom_data and click Create. This will store our custom data in a separate collection from our customer data.

For the "User ID Field," enter user_id. This will act as a foreign key to the user the data is attached to. While we mentioned not doing this in **Part 3**, this is one of those times when doing something like a relational database foreign key makes sense. The table storing user data is fully managed, so we do not get direct access to it, which means we cannot embed this data in the user record nor want to store the user credentials with the user data.

Once that is all done, click on Save Draft and then Review Draft & Deploy to save the new settings.

Once deployed, head back to the Users tab. We want to flag our new user as an admin, so let's create that custom user data. We will need the ID of the user we just created, so copy down that ID for the user. Then head back to Data Services in the top navbar and go into Browse Collections.

We need to make a new collection, so hover over the restaurant_pos_demo database name, and a + will appear to the right of the text. Click that, and then enter user_custom_data as the collection name. Go ahead and click Create to make an empty collection. Once that's created, click on Insert Document, switch over to the {} view, and paste in the following JSON document.

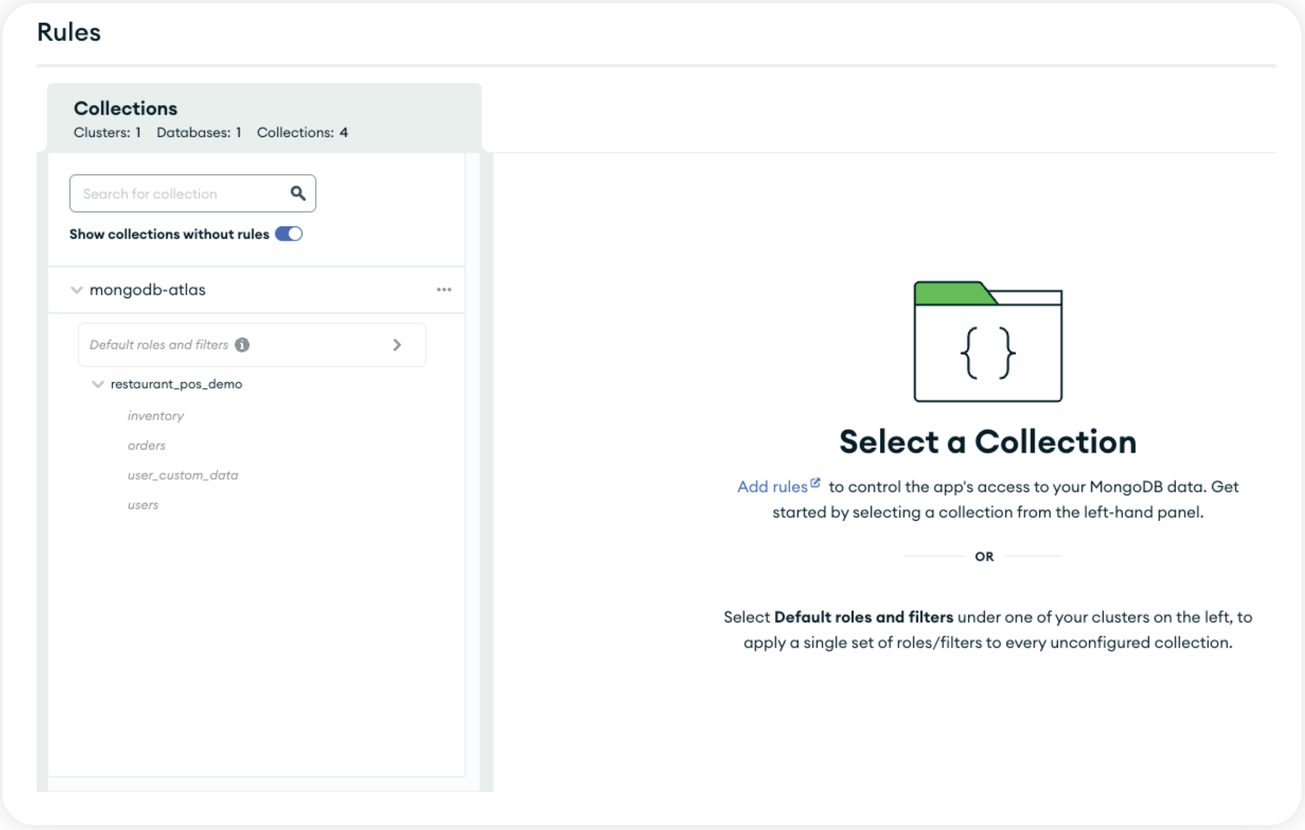
```
{
 "user_id": "<user-id-we-just-copied>",
 "admin":true
}
```

When we get to the code where we log in inside our application, the admin flag will be added to the user when it is returned. You can also arbitrarily add any information to this document for any more user information you may want to track in your application. For our tutorial, we need a boolean admin flag.

Query Security

We will look at one more section while we are in the Atlas web UI. One feature our administrative backend for the tutorial uses is querying the database directly from our client-side application. In many applications, like the customer side of our tutorial, we have a backend API that accesses our data. Atlas allows us to query the database from the browser through a combination of user authentication, which we just set up, and **rules-based data access controls**.

Click over to Rules from the App Services screen under "Data Access." This will take you to the Rules screen, where we can control authenticated users' access. Right now, our application does not do any authorization checking, but adding it is only a few clicks.


Rules

Collections

Clusters: 1 Databases: 1 Collections: 4

Show collections without rules

mongodb-atlas

Default roles and filters

restaurant_pos_demo

inventory

orders

user_custom_data

users

Select a Collection

Add rules to control the app's access to your MongoDB data. Get started by selecting a collection from the left-hand panel.

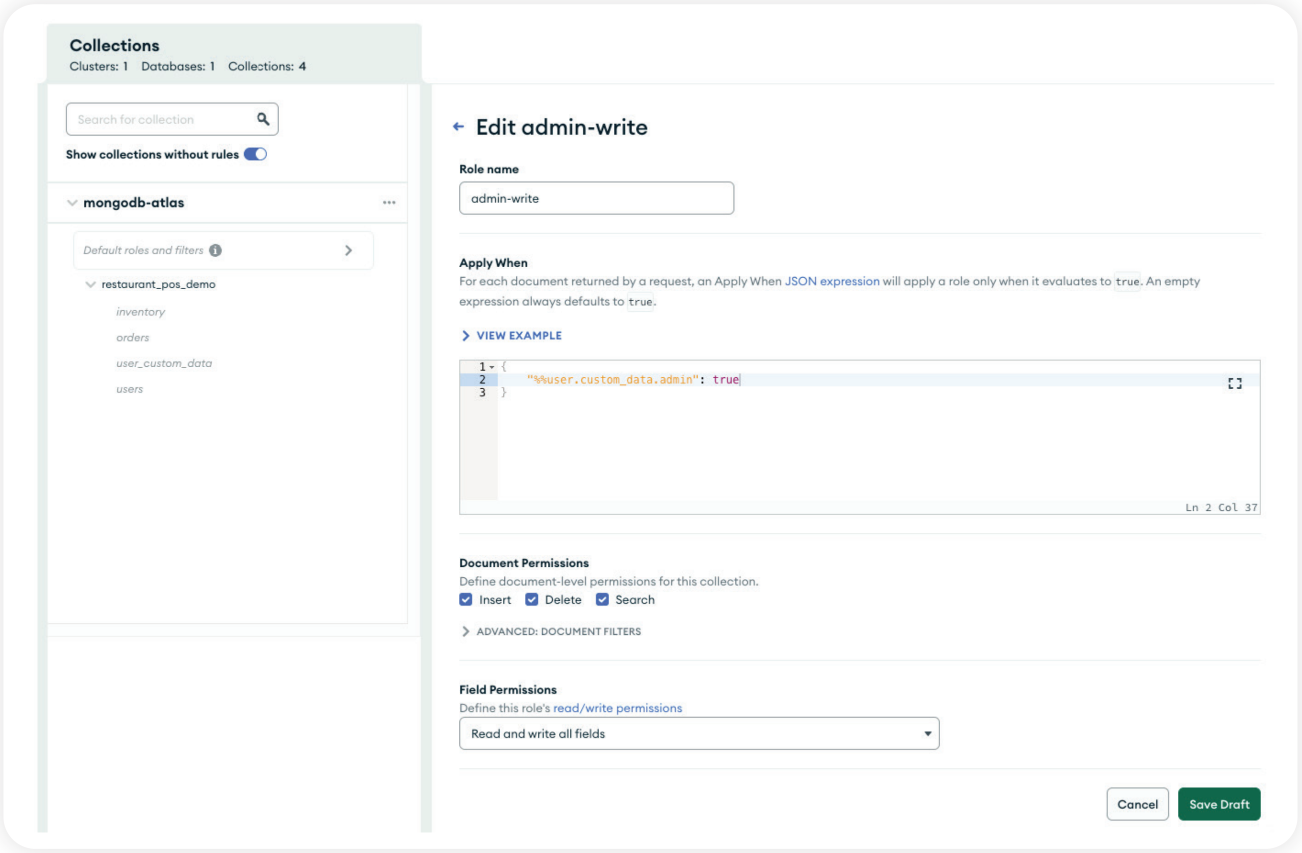
OR

Select **Default roles and filters** under one of your clusters on the left, to apply a single set of roles/filters to every unconfigured collection.

Atlas App Rules configuration

We want to ensure that any user that accesses is an admin, as admins will be the only ones who currently should access this data. For our application, we want only to allow someone with the `admin` flag set to true on their account (see why we went ahead and set that up earlier?). You can impose restrictions on the entire database or per schema. Since we only allow our admin backend to access the database directly, we can add these rules to the database itself. From the Rules screen, click on Default roles and filters just above the database name.

We can set up some preset roles, like deny all or allow all. We want to create a rule that uses our custom data, so go down and click on Skip (start from scratch).



Collections
Clusters: 1 Databases: 1 Collections: 4

Search for collection
Show collections without rules

mongodb-atlas
Default roles and filters
restaurant_pos_demo
inventory
orders
user_custom_data
users

Edit admin-write

Role name
admin-write

Apply When
For each document returned by a request, an Apply When JSON expression will apply a role only when it evaluates to `true`. An empty expression always defaults to `true`.

VIEW EXAMPLE

```
{
    "%%user.custom_data.admin": true
}
```

Ln 2 Col 37

Document Permissions
Define document-level permissions for this collection.
✓ Insert ✓ Delete ✓ Search

ADVANCED: DOCUMENT FILTERS

Field Permissions
Define this role's read/write permissions
Read and write all fields

Cancel Save Draft

New Atlas App dialog

We need to give our role a name, so let's name it "admin-write." We then need to set the rules for when our role will apply. Since we are worried about getting access to data when we are an admin, we can establish a simple rule that ensures that the user has a custom data attribute called `admin` and that it is set to `true`. Copy and paste the block of JSON below into the editor.

```
{
  "%%user.custom_data.admin": true
}
```

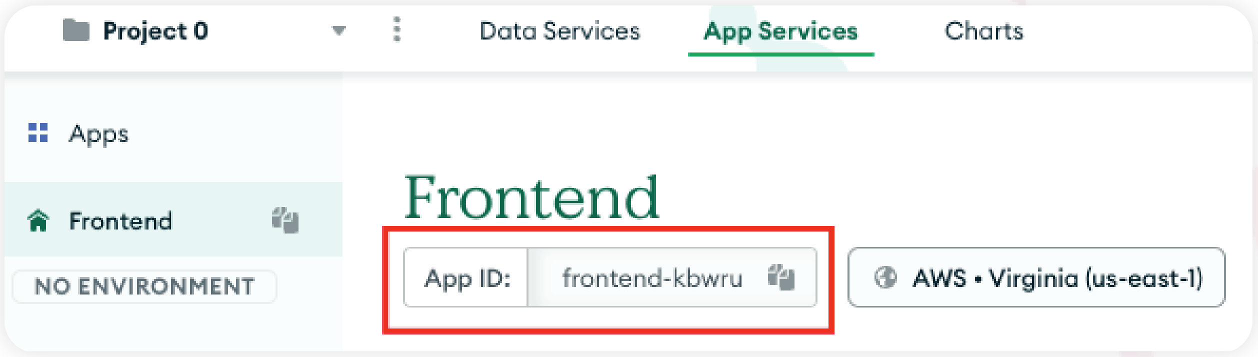
%%user tells the rule system to check the authenticated user. When we authenticate, the information stored in user_custom_data is attached to the user returned and assigned to the custom_data property. You can add any number of rules to help make this as granular as you want in an actual application.

Below this, we can set document permissions. Since we are an admin user, select "Insert," "Delete," and "Search." This will give any admin user full access to all the documents in any collection. Finally, we have the field permissions. You can set access rules down to the specific file for a collection. Right now, select "Read and write all fields."

These two settings will be more useful when you want to do things like all read-only views to specific user roles or restrict fields from roles that only have some access to information. These rules can be used in conjunction with broader Filter rules that restrict what data can even be returned from a query.

Save all these settings and then review and deploy our new access controls.

The last thing we need to do is tell our application which Atlas app to talk to. On the homepage for the Atlas app we are using, near the top is an App ID. Copy that down, and enter it into the .env file for the web app under VITE_REALM_ID.



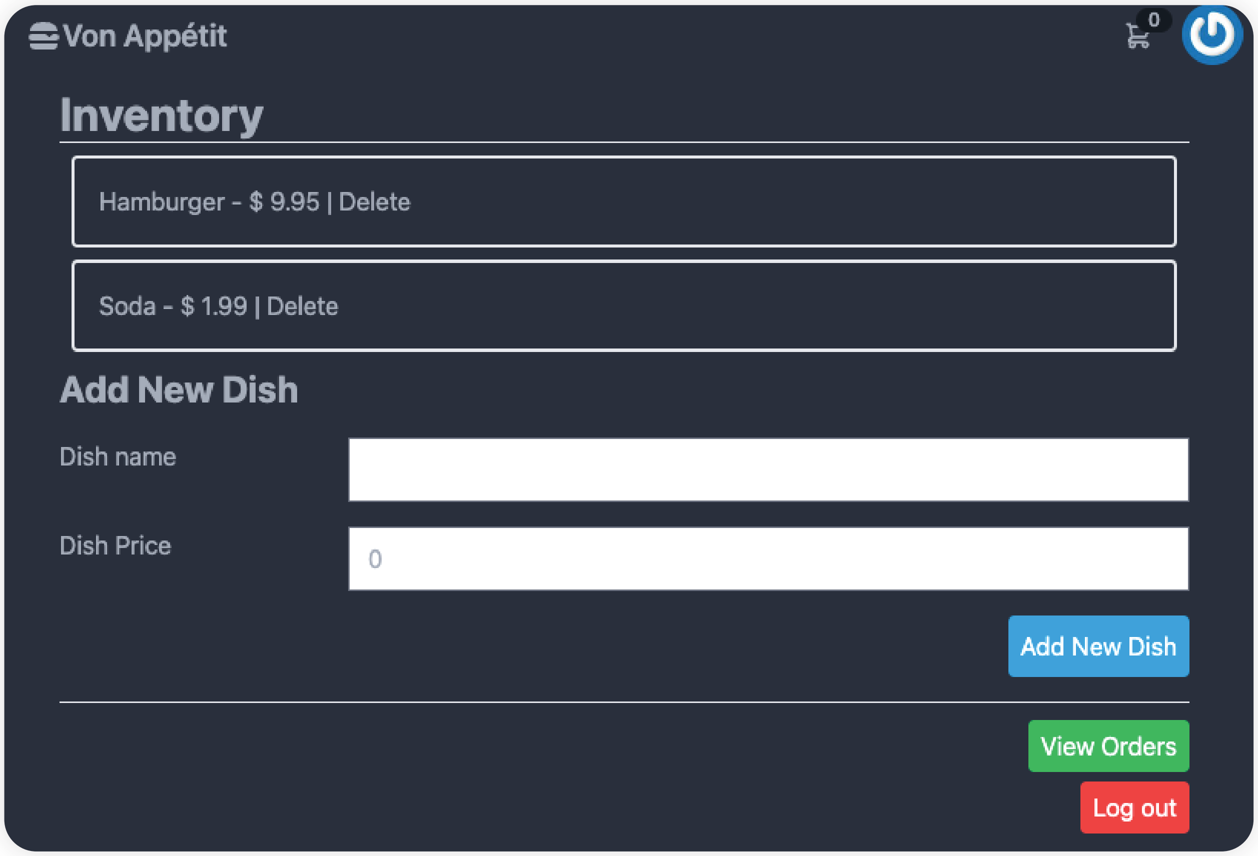
App ID: frontend-kbwru

App ID location

Can we log in yet?

Yes!

Head to http://localhost:5173/login and log in using the email address and password you assigned the user in Atlas. You should be greeted with an inventory screen and the option to add new dishes. If you see this, you are authenticated!



Tutorial Admin Area

MongoDB Atlas has a browser SDK that can be used to contact our cluster and Atlas app. We must take in an email address and password for our application and pass it into the SDK's authentication calls.

```
import { MongoDBRealmError } from 'realm-web';
import { ref } from 'vue'
import { useRouter } from 'vue-router';
import { authenticationStore } from '../stores/authentication-
Store';

const router = useRouter();
const username = ref('')
const password = ref('')
const authStore = authenticationStore()

const login = async () => {
  try {
    await authStore.login(username.value, password.value)
    router.push({ name: 'inventory.home' });
  } catch (error) {
    if (error instanceof MongoDBRealmError) {
      console.log(error.errorCode)
    }
  }
}
```

The VueJS code is relatively minimal. Our **Login.vue component**, we have a pull in an Authentication Store, which like our shopping cart is a wrapper to make it easier to pass logged in user information around. This store will use the SDK to log in. On this page, we only need to watch for the user to log in using the form and call `authStore.login()` with the username and password.

```
import { defineStore } from 'pinia'
import * as Realm from 'realm-web'

const realmApp = new Realm.App({id:
import.meta.env.VITE_REALM_ID})

export const authenticationStore =
defineStore('authenticationStore', {
   state: () => {
      return {
         token: null,
         user: null,
      }
   },
   actions: {
      async login(username, password) {
         const creds =
Realm.Credentials.emailPassword(username, password);
         this.user = await realmApp.logIn(creds)
         return this.user
      },
      setToken(token: string) {
         this.token = token
      },
      logout() {
         this.token = null
      }
   }
})
```

The **Authentication Store** is little more than a wrapper for the MongoDB SDK and some places to keep user information. We create a store using Pinia and create a new Realm.App() object with a link to our App ID we added to our .env file. Inside our authenticationStore object is a login() method called Realm.Credentials.emailPassword(). This generates a set of user credentials we can pass into the app object to authenticate. If the call to realmApp.login() is successful, we get a user back. We store that user off and can pull it from the store at any time.

From this point on, our user is considered authenticated. At any time we can check authenticationStore.user and if one exists, we are authenticated. Since we have logged in to Atlas via the SDK, we can also now access the database directly from the front end. We do this through a **Database Store**. All this store does is hold a connection back to our MongoDB cluster, and uses the logged in user's credentials.

This is powerful as we can perform data lookups directly in the browser instead of relying on our backend API. We can lock down this access to just admin users using the Rules we set up in the Atlas App configuration. If we wanted to throw away all of the MongoDB code in our backend API, we could add additional rules and filters to lock users to see only the data they can access. It's a great way to sketch together an application quickly.

```
import { defineStore } from 'pinia'
import { authenticationStore } from './authenticationStore'

const authStore = authenticationStore()
const dataSource = import.meta.env.VITE_MONGODB_DATA_SOURCE
const databaseName = import.meta.env.VITE_MONGODB_DATABASE

export const mongodbStore = defineStore('mongodbStore', {
  state: () => {
    return {
      restaurantDb:
authStore.user.mongoClient(dataSource).db(databaseName),
    }
  },
  actions: {
    getInventoryCollection() {
      return
this.restaurantDb.collection('inventory')
    }
  }
})
```

The database store is very minimal. We make pulling the database object and collection easier from the Realm connection we established in the Authentication Store. We can then query the database from our VueJS code, like on the **Inventory component**:

```
import { ref } from 'vue';
import { mongodbStore } from '../stores/mongodbStore';

const dbStore = mongodbStore()
let inventory = ref(Array());

async function getInventory() {
  const dishes = await
dbStore.getInventoryCollection().find()
  inventory.value = Array()
  dishes.forEach(dish => {
    if (dish.name) {
      inventory.value.push(dish)
    }
  })
}
```

In our VueJS component, we pull in the database store as `mongodbStore`. We can then use the MongoDB SDK syntax to find documents for us to use. Since we want all of the documents in the inventory collection, we can use `dbStore.getInventoryCollection().find()` to return all the documents we have access to. We can then push those into a VueJS `ref()` object to display on the page.

An essential part of that sequence is "we have access to." The Rules page in the Atlas App can be used to restrict what documents we can see. For example, it's common to tie a document to a user, such as an Author (or, in our case, the person who made an order). You can set up a Filter that would only return that user's orders, even if they did a call to `find()` to return everything. The restrictions and filters set up in the Rules section of the Atlas app will augment any query performed by the browser.

Conclusion

This ends our eBook through MongoDB Atlas and some complimentary Vonage Communications APIs. Hopefully, this series has inspired you to see what each of our platforms provides and how they might be helpful to you. As always, contact our developer advocates if you have any questions.

Happy coding!

About the Author

Chris Tankersley is a husband, father, author, speaker, PHP developer, podcast host, and probably lots of other things he's forgetting to mention. He works for Vonage as a Senior Developer Relations Advocate, helping developers use and integrate Vonage's communication platform into their applications and third-party services. Chris has worked with many different frameworks and languages throughout his fifteen years of programming but spends most of his day working in PHP and TypeScript. He is the author of "Docker for Developers," and helps developers integrate containers into their workflows. He can be found on Twitter at twitter.com/dragonmantank.