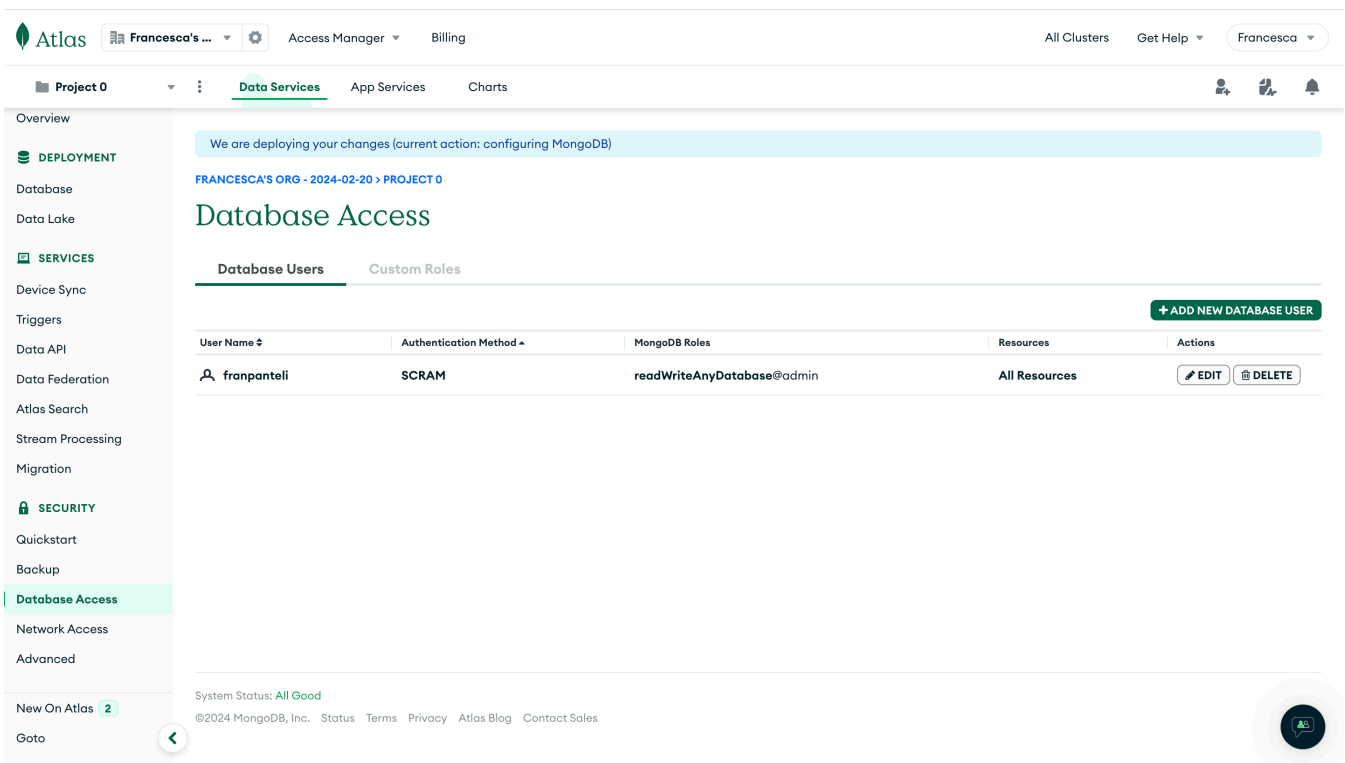
- ***MongoDB***
  - -> this is a database application that stores JSON documents (or records)
  - -> the data stored is for use in the application
  - -> JSON files are java objects
  - -> SQL is another type of database
  - -> MongoDB is a non-relational, "NoSQL" database
    - -> all data is stored within one record
    - -> instead of across many present tables as in an SQL database
- ***Mongoose***
  - -> this is an npm package which is used to interact with MongoDB
  - -> you can use js objects instead of JSON
  - -> this makes it easier to work with MongoDB
  - -> you can also create schemas <- blueprints for your documents
    - -> this allows you to avoid saving the wrong type of data and causing bugs later
- ***This section of the course***
  - -> this is about how to work with persistent data
  - -> how to set up a model, save, delete and find documents in the database
  - -> there are 12 parts to this
- ***Install and Set Up Mongoose***
  - ***Course content***
    - -> clone their GitHub repo for the starter code
    - -> you can also use their Gitpod starter code
    - -> setting up a MongoDB Atlas database
    - -> then importing the required packages to connect to it
    - -> you have to follow the tutorial at https://www.freecodecamp.org/news/get-started-with-mongodb-atlas/ for this
      - -> using MongoDB to store data
      - -> using a service called MongoDB Atlas
      - -> you create a MongoDB Atlas account
      - -> then you create a new cluster
        - -> you fill in information
        - -> then you create a cluster
        - -> then you create a new one for the database



        - -> then under security you are allowing it access from all IP addresses
        - -> under DEPLOYMENT you can connect to the cluster by clicking on Database
        - -> our cluster URI is mongodb+srv://franpanteli:<password>@franpanteli.aqokq0o.mongodb.net/?retryWrites=true&w=majority
            - -> you also need to add in the cluster name
            - -> you are filling your password into that URI, and with the cluster name
            - -> adding the name of our database before the query string ? retryWrites=true&w=majority
        - -> this is making a database
    - -> connecting to an existing database
        - -> you create a cluster and a database
        - -> you can then connect this to a new application
        - -> on the left side under DEPLOYMENT > Database > Browse Collections <- this shows a list of existing databases and collections
        - -> you can copy the database name you want to connect to and replace <db-name> with it in the URI string for your cluster
            - -> URI's aren't always online, URL's are
            - -> you can add the URI of your cluster fo the application and connect to your database
- ***Task / challenge***
    - -> after cloning their boilerplate code and going through their setup tutorial, you complete the task challenge
    - -> after going through the tutorial, mongoose@^5.11.15 will have been added to the project's package.json file -> as an npm package
    - -> require mongoose in myApp.js
    - -> then create a .env file and adds a MONGO_URI variable to it
        - -> the value of this should be the MongoDB Atlas database URI which we previously generated
        - -> the syntax of this should be MONGO_URI='VALUE'
    - -> then connect to the database by calling the connect method within the myApp.js file
        - -> this should be done with this syntax -> mongoose.connect(<Your URI>, { useNewUrlParser: true, useUnifiedTopology: true });
- ***Create a Model***
    - ***-> Schemas***
        - -> CRUD <- Create
        - -> each schema map to a MongoDB collection
        - -> this defines the shape of the documents within that collection
        - -> these are building blocks for Models
        - -> these can be nested to create complex models
        - -> models allow you to create documents (instances or your object)
    - ***-> Gitpod is a server***
        - -> interactions with the database happen with handler functions
        - -> these functions are executed when some event happens
            - -> e.g some user interaction sends an API request
    - ***-> the done() function***
        - -> this is a callback
        - -> this tells us that we can proceed after completing an operation -> inserting / searching / updating / deleting
        - -> this follows the Node convention
        - -> this should be called as done(null, data) on success or done(err) on error
    - ***-> you can encounter errors***

        - -> an example of this is below
            - /* Example */
            -
            - const someFunc = function(done) {
            -   //... do something (risky) ...
            -   if (error) return done(error);
            -   done(null, result);
            - };
            - -> error functions in the code
    - ***-> Task***
        - -> create a person schema called personSchema
        - -> the shape should be
            - -> a required name field type of string
            - -> an age field of type Number
            - -> a favouriteFoods field of type [String]
        - -> use the Mongoose basic schema types
        - -> you can also add more fields
        - -> use validators like required or unique and set default values
        - -> then create a model from the personSchema and assign it to the variable person
        - -> we are creating a schema (a set of rules about a specific document)
        - -> in this case it's the schema of a person called personSchema -> and we are setting a model for it equal to the variable called Person
        - -> we are doing this in the myApp.js file
- ***Create and Save a Record of a Model***
    - -> creating and saving the record of a model
    - ***-> in the createAndSavePerson function***
        - -> we are creating a document instance using the Person model constructor
        - -> in other words, we are using the person model to create and save a 'person'
    - -> we are passing to the constructor an object having the fields name, age and favouriteFoods for the person which we are creating
        - -> the syntax you use when you define the person has to be the same as the syntax which defines that document of person
    - ***-> then save the person***
        - -> this is done using document.save() on the returned document instance
        - -> pass to it a callback using the Node convention
    - ***-> this is a common pattern for the CRUD methods***
        - -> these take a callback function like this as the last argument
        - -> example
            - /* Example */
            - // ...
            - person.save(function(err, data) {
            -   //   ...do your stuff here...
            - });
- ***Create Many Records with model.create()***
    - ***Using the Model.create() method***
        - -> creating many instances of a model
        - -> this can be used when seeding a database with initial data
        - -> Model.create() <- this takes an array of objects as the first argument and saves them all in the database
            - -> the syntax of this argument is [{name: 'John', ...}, {...}, ...]
    - ***Task***
        - -> we are changing the createManyPeople function

        - -> we are using Model.create() function
        - -> the argument of this is an array of people
        - -> rather than using this function to create one person
        - -> you can reuse the model you created from the previous part of the question
- ***Use model.find() to Search Your Database***
    - ***-> Model.find***
        - -> this accepts a query document <- a JSON object as the first argument
        - -> the second argument to this is a callback
        - -> this returns an array of matches <- we are searching a database
        - -> there are a wide number or search options this can be used for
    - ***-> Task***
        - -> modify the findPeopleName function
        - -> we are finding all the people having a given name, using Model.find() -> [Person]
        - -> the search key we are using for this is personName
- ***Use model.findOne() to Return a Single Matching Document from Your Database***
    - -> we are returning a single matching document from the database
    - ***-> Model.findOne()***
        - -> this behaves like Model.find()
        - -> this only returns one document in the search, rather than multiple items
        - -> we are searching a MongoDB database
        - -> this is useful when searching by properties that you have declared as unique (haven't been repeated elsewhere)
    - ***-> Task***
        - -> modify the findOneByFood function
        - -> we are finding one function which has a certain food in the person's favourite's
        - -> to do this, we are using Model.findOne() -> Person
        - -> to do this, we are using the function argument food as the search key
- ***Use model.findById() to Search Your Database By _id***
    - ***-> model.findById()***
        - -> we are searching the MongoDB database by ID
        - -> each of the documents we save to the database has an id field -> we are searching by these IDs for a specific number
        - -> these are unique alphanumeric keys
        - -> this method is used to do this
    - ***-> Task***
        - -> modify the findPersonById
        - -> we are finding the only person having a given _id using Model.findById() -> Person
        - -> the search key we are using for this is the function argument personId
- ***Perform Classic Updates by Running Find, Edit, then Save***
    - ***-> Model.update()***
        - -> if you have a MongooseDB database and want to change one of the documents which is stored in it
            - -> to be able to use it <- e.g to send it back in a server response
            - -> there is a method for doing this called Model.update()
            - -> this is bound to the low level mongo driver
            - -> this can bulk edit many documents matching certain criteria
            - -> this only sends a status message back
                - -> there is the user (client), the MongoDB server which stores the data and the server which we are writing the code for
                - -> we are telling the server to change the information in the database -> without it sending that data back to us (just a status message about it having changed)
                - -> we are on the server side telling the information on the database to change -

like a remote control
        - -> this makes model validations difficult, because it directly calls the mongo driver
    - ***-> Task***
        - -> modify the findEditThenSave function
        - -> we are finding a person by _id
        - -> we are adding "hamburger" to the list of the person's favoriteFoods
        - -> you can search for them using any of the methods from the three previous questions
        - -> you can use Array.push() for this <- to update the database
        - -> then in the find callback `save()` the updated `Person`
        - -> you have to had declared that favourite Foods as an array in the schema
            - -> you need to tell it that the type it an array
            - -> if you don't, it defaults to Mixed type
            - -> you have to manually mark it as edited using document.markModified('edited-field')
            - -> refer to their Mongoose article for more information
- ***Perform New Updates on a Document Using model.findOneAndUpdate()***
    - ***-> findByIdAndUpdate()***
        - -> using this to perform new updates on a document
        - -> the method used in the previous question to update a document in the database was an older method
        - -> this method is for newer versions of Mongoose
        - -> more advanced features are pre/post hooks and validation -> these work differently with this approach
        - -> the method used to update a MongoDB database will be different, depending on what you want to update
        - -> we are searching for an ID and updating the information in the document which is stored at that ID in the MongoDB database
    - ***Task***
        - -> modify the findAndUpdate function
        - -> we are finding a person by Name and setting the person's age to 20
        - -> then using the function parameter personName as the search key
        - -> you should return the updated document
            - -> you need to pass the options document { new: true } as the third argument to findOneAndUpdate()
            - -> these methods return the unmodified object as a default
- ***Delete One Document Using model.findByIdAndRemove***
    - ***model.findByIdAndRemove***
        - -> there were two previous update methods we used
            - -> to update the documents in the MongoDB database
            - -> you can search for the document by its ID in the database
            - -> or you can search for one of the documents which has some unique data which you enter into the search
            - -> findByIdAndRemove and findOneAndRemove <- these are the two methods you can use to do this, but to remove the data in the MongoDB database
                - -> either based off of the ID of the document which you are changing, or some unique piece of information about it
            - -> we are passing the removed document into the database
            - -> we are using personID, in this example <- to search for the document which we want to delete from the MongoDB database
        - ***Task***
            - -> modify the removeById function

        - -> we are deleting one person by their ID from the database
        - -> we are either using findByIdAndRemove() or findOneAndRemove()
- ***Delete Many Documents with model.remove()***
    - ***Model.remove()***
        - -> Model.remove() <- this is what we use to delete all the documents matching given criteria
            - -> we are still dealing with a MongoDB database
            - -> instead of deleting a single item from the database, we are deleting all of the items which match the search input parameters which we want
    - ***Task***
        - -> modify the removeManyPeople function
        - -> we are deleting all of the people whose name is within the variable nameToRemove
        - -> we are using Model.remove() to remove this array of elements from the database
        - -> we are passing this to a query document with the name field set, and a callback
        - -> Model.remove() sends a JSON object
            - -> it doesn't return the information which we removed from the dataset
            - -> it returns a JSON object containing the outcome of the operation and the number of items affected
            - -> we are then using a done() callback to make sure that the function worked
- ***Chain Search Query Helpers to Narrow Search Results***
    - -> chaining search query helpers to narrow search results
    - ***The Model.find() method needs a callback function in the argument***
        - -> for the previous JavaScript functions, a second argument was used which controlled error handling
        - -> this was the second function -> for example if the element we were searching the database for didn't exist then this function would log an error message to the console and the function would stop there
        - -> you need the second (error handling) function in the argument or the query isn't executed
        - -> this allows you to use a chaining syntax
    - ***The database search is executed when you chain the method .exec()***
        - -> the callback has to be passed to the last method
        - -> the second argument which was used when defining the previous functions was concerned with error handling
        - -> you can have many of these functions chained together -> and they are ended by the fully chained method .exec()
        - -> the callback has to be passed to this last method
        - -> there are many query helpers
    - ***Task***
        - -> we are modifying the queryChain function
        - -> we are finding people who like the food given by the variable foodToSearch
        - -> we are then sorting by name, limiting the results to two documents and hiding their age
        - -> we are chaining .find(), .sort(), select() and .exec()
        - -> then passing the done(err, data) callback to exec()