Objective:

Build the database, seed the data, and configure the Project to use it.

Supplemental Material:

Docker Compose file Reference:	https://docs.docker.com/compose/compose-file/
MongoDB Shell Docs:	https://docs.mongodb.com/manual/mongo/
mongodump Reference:	https://docs.mongodb.com/manual/reference/program/mongodump/
GitLab CI Docs:	https://gitlab.camosun.bc.ca/help/ci/README.md

Preparation:

Background:

In the last lab, you designed your Data Model. You created a Schema for your data and compiled it into a Model that your Application can use to interact with a MongoDB database. In this lab, you will concentrate on building the database itself, seeding it with data, and configuring your Project to use it.

BEFORE YOU BEGIN THE TASKS:

- As always, you should start a new git 'dev' branch from your master branch before starting the tasks in these labs.
- A good R.O.T. is to commit your work at the end of each task.

Tasks:

Task 1 - Adding the Data to the Database (Local Development)

Now that your Project's Schema is complete and compiled, you are ready to add your Project's data to the database (adding initial data is called *seeding*).

1. Stop and remove any running containers.

2. First, you need to add an environment variable to the `db:` service in your *docker-*

compose.yml file. Add the following lines to this service:

```
environment:
  - MONGO_INITDB_DATABASE=<your database name>
```

This sets up the database that should be initialized with data. Replace `<your database name>` with the name of your database (which should match your project's name).

3. Next, you are going to mount a host folder to a folder in the container for the `db:` service. Add the following lines to this service:

```
volumes:
  - ./mongo-seed:/docker-entrypoint-initdb.d
```

This maps a `mongo-seed` subdirectory to the `/docker-entrypoint-initdb.d` directory in the container. This 'special' directory will run any Shell or JavaScript scripts that are inside it automagically after the MongoDB database comes up. For those that are curious, this is the exact same method we used for ICS 199 to seed the MySQL database for your project in that course.

4. Create a `mongo-seed` subdirectory in your project folder, if you haven't already. Inside this folder, create a new, empty text file and name it *seed-data.js*. Open up this file in a text editor.

5. Using Camosun Airways as an example, the JavaScript object to be stored in the database is the one from the last Lab in Task 5 that the Schema was designed for. Here it is again:

```
flight: {
      origin: 'Victoria',
      destination: 'Vancouver',
      days: ['daily'],
      flightTimes: [{
        depart: '7:00am',
        arrive: '7:30am',
        flightNum:1,
        Aircraft: 'Dash-8 Q400',
        Fare: '$300'
      },{
        depart: '8:00am',
        arrive: '8:30am',
        flightNum:2,
        Aircraft: 'Cessna C172',
        Fare: '$200'
      }],
      flightDates: [
        'December 12, 2017',
        'December 13, 2017',
        'December 14, 2017'
      ]
}
```

The MongoDB Shell, similar to the MySQL prompt for MySQL or the SQL*Plus environment for Oracle, allows you to run CLI commands on a database. You can find information about it at https://docs.mongodb.com/manual/mongo/. The Shell provides a `save` command that can be used to save a JavaScript object to the database:

```
db.flights.save({
       origin: 'Victoria',
       destination: 'Vancouver',
       days: ['daily'],

       // the rest of the object ...

});
```

The `flight:` at the beginning was removed since it was the property of the parent object in the controller. You start database manipulation commands in the Mongo Shell using `db`. This is then followed by the **collection name**. This is the plural of the model name you used when you compiled your Schema in the last lab. It is important you get that right! For Camosun Airways, the model is `flight` so the collection name is `flights`. Finally, this is followed by the `save` command. You have to add round brackets around the JavaScript object since it is an argument to the `save` command.

6. The Script is almost done but there are a few changes that need to be made. Looking back at the Schema from the last lab for Camosun Airways, the `flightDates` should be Date objects. That is easily rectified:

```
flightDates: [
    new Date('December 12, 2017'),
    new Date('December 13, 2017'),
    new Date('December 14, 2017')
]
```

7. Finally, there is one other change that needs to be done. Subdocuments don't automatically get an `_id` path when added to a database. They have to be manually added. For the `flightTimes` subdocuments:

```
flightTimes: [{
    _id: ObjectId(),
    depart: '7:00am',
    arrive: '7:30am',
    flightNum:1,
    Aircraft: 'Dash-8 Q400',
    Fare: '$300'
  },{
    _id: ObjectId(),
    depart: '8:00am',
    arrive: '8:30am',
    flightNum:2,
    Aircraft: 'Cessna C172',
    Fare: '$200'
  },{
    _id: ObjectId(),
    depart: '9:00am',
    arrive: '9;30am',
    flightNum:3,
    Aircraft: 'Dash-8 Q400',
    Fare: '$300'
}],
```

`ObjectId()` will generate a unique identifier for each subdocument.

****Following the above steps, do something similar for your project.**

8. Save and close your *seed-data.js* file. With the changes you made in *your docker-compose.yml* file and the addition of the *seed-data.js* file in the `mongo-seed` subfolder, the next time you bring up your containers, the database should get automatically populated with your data.

Task 2 - Verify the Data Has Been Added and Adding more Documents

1. If you haven't already, bring up your project's containers. To verify that your document was added to the database in Task 1, you will need to run an interactive MongoDB Shell. Log into the MongoDB container:

   ```
   winpty docker exec -it mongo bash
   ```

   At the root prompt, type `mongo` and hit enter to start the MongoDB Shell.

2. Once at the prompt, you can type `help` to see a list of commands available to you. The first command we'll issue is to list all the databases in the container. Type the following:

   ```
   show dbs
   ```

   You should see your database listed as well as admin and local. Both these databases are created automatically with each MongoDB Instance. The admin database is for access control and the local database is for replication (data redundancy for production deployments).

3. To operate on your database, you use the `use` command. For example, for Camosun Airways:

   ```
   use camosun-airways
   ```

4. The first thing you can do to verify the data was added, is to ensure your collection exists. Remember that the collection name is the plural of the model name. In the case of Camosun Airways, it is `flights`. You can use the `show collections` command to see if your collection is in the database:

   ```
   show collections
   ```

   And you should see your collection listed.

5. To see all the documents in this collection, you use the `find` command. This is a database operation and requires using both `db` and the collection name in front of it (similar to the `db.flights.save` used in Task 1). Enter the following command:

   ```
   db.<your collection name>.find().pretty()
   ```

   The `pretty()` at the end is an operation that 'pretty' prints it to the console in a more human readable format. You should see your document listed in the console. The `save`

operation you did earlier actually converted your JavaScript Object into JSON. To make things even more confusing, your document is actually stored as BSON in the database. The following diagram may help you:

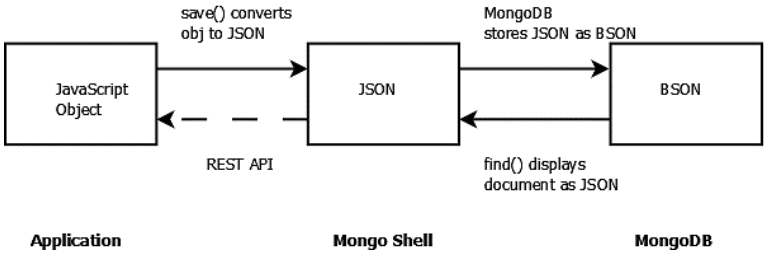

Figure 1: Data Transformations During the Lifetime of the Application

The last piece in the puzzle is taking the JSON returned by queries and using that data in your application. That's where the REST API comes in (next Lab).

6. Once you have verified that your data has been stored correctly, it is time to add more sample data to your database. Open your *seed-data.js* file.

7. To add another document in the same collection, just add another `db.<collection name>.save` command in your *seed-data.js* file. If you have multiple Schemas for your project, use the `db.<collection name>.save` command and change the collection name. **In any case, you should have at least four documents in your database**.

   ****Add more documents to your database so that you have at least four.** Test to make sure they were added correctly by bringing up the containers and use Mongo Shell commands to verify.**

Task 3 - Building the Production Database

At this point, you should have a fully seeded database running in a MongoDB container on your local development machine. The issue now is, how to get that data sent to the deployment server. You added instructions in Task 1 to your *docker-compose.yml* file to seed your database, but the deployment server doesn't use that file. In addition, the Dokku MongoDB container can't use the *seed-data.js* file. We can, however, use the CI/CD file, *.gitlab-ci.yml* to automate seeding the data in the database on the deployment server by using a database dump.

1. The first thing you need to do is make a database dump. You'll put this dump in the `mongo-seed` subfolder of your project. In a Bash prompt in your MongoDB container, navigate to the `/docker-entrypoint-initdb.d` directory. If you take a directory listing of this directory inside the container, you should already see your *seed-data.js* file.

2. At a bash prompt in your MongoDB container, enter the following:

   ```
   mongodump --archive=dump.gz --gzip --db <your database name>
   ```

   The database name should be the same name as your project. This command will create a `dump.gz` file in your `mongo-seed` directory since the container's `/docker-`

`entrypoint-initdb.d` folder is mapped to `mongo-seed`. **Note that if you make any changes to your *seed-data.js* file from this point forward, you will need to re-do the database dump**.

3. With that out of the way, it is time to modify the GitLab CI file to add the deployment of your database. Open your project's *.gitlab-ci.yml* file in a text editor.

4. You are going to create a new section in this file. Below the `deploy_app_to_dokku:` section, without any indentation, add the following line:

   ```
   deploy_db_to_dokku:
   ```

5. Indented by two spaces, add the following:

   ```
   stage: deploy-db
   ```

   This is the stage name for reference to the stages section.

6. In the `stages:` section, above the `deploy_app_to_dokku:` section, add the following below the `- deploy-app` line:

   ```
   - deploy-db
   ```

   The `stages:` section is for clarifying the order each stage should run in.

7. Back in the `deploy_db_to_dokku:` section, add the following (it should be indented so it is inline with what you added in Step 5):

   ```
   tags:
     # use a deploy runner
     - deploy
   ```

   This ensures that the proper runner is used for deployment.

8. Next, is the `script:` section. Add the following (it should be indented so it is inline with what you added in the last Step):

```
script:
  # Unlink Service - the conditional is if this is being run on first deploy
  - 'ssh dokku@$DEPLOY_SRV mongo:unlink ${CI_PROJECT_NAME}-DB $CI_PROJECT_NAME || true'
  # Destroy DB - the conditional is if on first run, the DB hasn't been created yet
  - 'ssh dokku@$DEPLOY_SRV -- --force mongo:destroy ${CI_PROJECT_NAME}-DB || true'
  # Create/Re-create DB
  - ssh dokku@$DEPLOY_SRV mongo:create ${CI_PROJECT_NAME}-DB
  # Import dump
  - ssh dokku@$DEPLOY_SRV mongo:import ${CI_PROJECT_NAME}-DB < ./mongo-seed/dump.gz
  # link DB container to app
  - ssh dokku@$DEPLOY_SRV mongo:link ${CI_PROJECT_NAME}-DB $CI_PROJECT_NAME
```

`$CI_PROJECT_NAME` is a pre-defined variable provided by GitLab Runner. It is automatically set to your project's name. The `${CI_PROJECT_NAME}-DB` sets the database name on the deploy server. It simply is your project name with -DB appended to it.

Each of these lines (besides the comments) ssh into the deploy server and give Dokku

commands. They are discussed below:

- The first ssh line unlinks the mongo container from the app. We can never have a command fail in a CI script otherwise it stops working. Since it is possible this command will run before a mongo container has even be created, we need to add the `|| true` so that if the command fails, the script will keep running. We need to wrap the entire line in single quotes because `|` is a special character.

- The second ssh line destroys the database if it already exists. If it doesn't, we can't have this command fail so like the first line, we add a `|| true` at the end.

- The third ssh line creates the MongoDB container for your project on the deploy server. This should never fail!!

- The fourth ssh line is where the magic happens. This imports the dump you created at the beginning of this task to seed the data.

- The fifth ssh line links the mongo container with your app; it's ready to go!

9. The last thing you need to add to the deploy_db_to_dokku: section is the following:

   ```
   only:
     - master
   ```

   The `only:` should be inline with `script:`, `tags:`, and `stage:`. This ensures that this stage only runs on the master branch.

10. Save and close your *.gitlab-ci.yml* file. Commit all your changes on your dev branch (you did branch didn't you??). Switch to your master branch and merge in your dev branch. Push your master branch to GitLab.

11. Monitor the CI/CD Job on GitLab for your project. Look at the log. Ensure that the database deployment works and that it gets properly seeded!

Task 4 - Configuring Your Application to use the appropriate Database

At this point, you should have successfully deployed your Application, along with its database, to the deployment server. The next step is to have your Application use the appropriate database based on where it is running. You've already configured it to use a local MongoDB instance when developing locally, but you haven't configured it to use the deployed database when it is running on the deploy server.

1. This Task will require you to make changes to *db.js*. Open this file now.

2. The key to connecting to a database is the connection string or database URI. You already have the one for the local MongoDB Instance in *db.js*; you just need to test if the Application is running on the deploy server and if so, change the `dbURI` to connect to the

production database. Since the `dbURI` might change, change its declaration from `const` to `let` in *db.js*:

```
let dbURI = 'mongodb://mongo/<your project name>';
```

3. Next, you are going to test an environment variable. Recall that in your docker-compose.yml file, you set `NODE_ENV` environment variable to `development`. On the deployment server, it defaults to `production`. We can therefore test this variable:

   ```
   if (process.env.NODE_ENV === 'production') {
   ```

   `process.env` is a Node.js JavaScript object that contains all the Application's environment variables.

4. If this statement is true, then the Application must be running on the deploy server. Luckily, it sets an environment variable, called `MONGO_URL` with the database URI (if you look at the CI/CD logs on GitLab for the database deployment, you will see it there). Therefore, we can use it in the `if` statement:

   ```
         dbURI = process.env.MONGO_URL;
   }
   ```

5. Save and close *db.js*. Commit, merge, bump version, tag, and push (if you don't remember how to do some of these steps, refer to Lab 6, Task 5, Step 4). How will you know if your Application connected to the database? That's the next lab, when you start building your Data API.