Mongoose CRUD Operations Reference

Database analysts usually refer to the query actions that can be performed against a database as CRUD operations, which stands for Create, Read, Update, and Delete. One can create a new page in the database, read back the contents of that page, modify or update it after it already exists, and delete or remove it from the database. However, in Mongoose, we will use the HTTP versions of those words, as shown below, along with reference material for the various versions of these four methods.

CRUD OPERATION	HTTP METHOD
Create	POST
Read	GET
Update	PUT
Delete	DELETE

Read / GET

GET is the method used to read back the contents of a database. Mongoose provides three basic ways to do this (find), plus one advanced method (where).

.find([callback])
Finds all pages in the document and returns them in an array, even if the array is empty or contains a single page.

```
Employee.find((err, emps) => {
   if (err) {
      res.status(500)
      return next(err)
   },
   return res.status(200).send(emps);
})
```

Note: the error is not meant to catch a query that returns no results, but any connection, syntax, or transmission error that may have been encountered, hence the 500 status. If no error is found, this query returns a list of all employees.

.find([filter(s)], [callback])

If filters are passed into .find(), only those pages that match the filters will be returned in the array. The following query returns only those pages that contain an employee in the marketing division whose last name is “Patel,” but it will return *every field* contained in the database.

```
Employee.find(
   {surname: "Patel", division: “Marketing”},
   (err, emps) => {
      if (err) {
         res.status(500)
         return next(err)
      }
   },
   return res.status(200).send(emps);
});
```

.find([filter(s)], [field(s)ToReturn], [callback])
This query adds another argument that specifies which fields should be returned.

```
Employee.find(
   {surname: "Patel", division: “Marketing”},
   {hire_date: true, salary_level: true},
   (err, emps) => {
      if (err) {
         res.status(500)
         return next(err)
      }
   },
   return res.status(200).send(emps);
});
```

If you want to limit the number of records returned (this can greatly reduce the time for the query to execute), you can substitute findOne for find in any of the examples above.
With .findOne():

.where(selector)
This query is powerful and more intuitive, once you get the hang of it, that is. Where queries can be much more complex, for example setting top and bottom limits on numeric or date fields. Calling .where() on a Mongoose model returns a Mongoose query object, not the query that has been run against the database. In order to actually execute the query, the .exec() method has to be called and passed in a callback.

```
Employee.where
```

```
   ("salary_level").gte(4).lte(6).exec(
      (err, emps) => {
         if (err) {
            res.status(500)
            return next(err)
         }
      },
      return res.status(200).send(emps);
   )
```

The beauty of .where queries is that they read almost like English: "Find all employees where the salary level is greater than or equal to four, and less than or equal to six." Just don't forget to execute the query.

Create / POST

POST is the method used to create new pages in a database document. The method's syntax itself is very simple, however care must be taken when creating the request body to make certain it is formatted correctly (matching the criteria set down in the Mongoose schema) and posted in the proper protocol (raw JSON).

```
employeeRouter.post("/", (req, res, next) => {
  const newEmployee = new Employee(req.body)
  newEmployee.save((err, savedEmp) => {
    if(err){
      res.status(500)
      return next(err)
    }
    return res.status(201).send(savedEmp)
  })
})
```

One common operation you may find yourself doing is, before adding a page to a document, checking to make sure that document doesn't already exist. Mongoose does not provide a way to do this, however, there is a library called mongoose-findOrCreate that will take care of this for you.

Update / PUT

One could think of this method as a combination of "read" and "create," but instead of creating a new page, we query the database and send a change to be made using findOneAndUpdate. In the example below the .findOneAndUpdate uses four arguments: 1) the field on which to match the page, 2) the body that is being passed by the JSON request, 3) a switch to require the database to return the *updated* record instead of the record we passed, and 4) the standard callback function to retrieve the recordset.

```
employeeRouter.put("/:employeeId", (req, res, next) => {
  Employee.findOneAndUpdate(
    { _id: req.params.employeeID},
    req.body,
    {new: true},
    (err, updatedEmp) => {
      if(err){
        res.status(500)
        return next(err)
      }
      return res.status(201).send(updatedEmp)
    }
  )
})
```

Another option would be to use a findOne or findById, make the changes to the properties manually, then use .save to save the change. The benefit of doing it this way is that you have more control over the changes being made, but at the expense of having to make two trips to the database (one to retrieve the document, another to save it). Using findByIdAndUpdate combines these two trips into one, but also makes it a little harder to make granular modifications. It also bypasses any model “hooks”, like a "pre-save" hook. But if that isn't a concern to you, findByIdAndUpdate and findOneAndUpdate are great shortcut methods to use.

Delete / DELETE

Similar to the Update/POST section above, you can go about deleting a document from the database by first finding it, then running the .remove() method on the found document. However, Mongoose has provided some helper methods such as .findOneAndRemove() and .findByIdAndRemove().

```
employeeRouter.delete("/:employeeId", (req, res, next) => {
  Employee.findOneAndDelete(
    { _id: req.params.employeeID },
    (err, deletedEmp) => {
      if(err){
        res.status(500)
        return next(err)
      }
      return res.status(200).send(`Successfully deleted item
${deletedEmp.surname} from the database.`)
  })
})
```

Conclusion

The examples above are by no means an exhaustive listing of the query options you can perform on a database. In the future you will need to spend time reading the Mongo and Mongoose documentation for these operations. In order to add the title of DBA to your full-stack resume, you will need to master the many nuances of the query operations.