# **NeDB Example**

The focus of this example is to explore using NeDB [1] and its integration with a sample web application.

Contents

[**NeDB Example** 1](#_Toc39087552)

[**1.** **Introduction to NeDB** 1](#_Toc39087553)

[**1.1.** **Installation** 1](#_Toc39087554)

[**2.** **Standalone NeDB** 1](#_Toc39087555)

[**2.1.** **Insert documents** 1](#_Toc39087556)

[**2.2.** **Retrieve documents** 3](#_Toc39087557)

[**2.3.** **Delete documents** 4](#_Toc39087558)

[**2.4.** **Update documents** 5](#_Toc39087559)

[**3.** **Using NeDB in a web application** 6](#_Toc39087560)

[**3.1.** **The model** 6](#_Toc39087561)

[**3.2.** **Show all students** 8](#_Toc39087562)

[**3.3.** **Deleting students** 10](#_Toc39087563)

[**3.3.1.** **Add a delete icon to the students page** 10](#_Toc39087564)

[**3.3.2.** **Add a request handler to the controller** 11](#_Toc39087565)

[**3.3.3.** **Add a delete function to the model** 11](#_Toc39087566)

[**3.4.** **Update students** 12](#_Toc39087567)

[**3.4.1.** **Add an edit icon to the students page** 12](#_Toc39087568)

[**3.4.2.** **Add a GET request handler to the controller** 12](#_Toc39087569)

[**3.4.3.** **Add an update page** 13](#_Toc39087570)

[**3.4.4.** **Add a POST request handler to the controller** 14](#_Toc39087571)

[**3.4.5.** **Add an updateStudent method to the model** 14](#_Toc39087572)

[**4.** **Images** 15](#_Toc39087573)

[**5.** **References** 15](#_Toc39087574)

1. **Introduction to NeDB**

NeDB is a lightweight document DBMS written in JavaScript. NeDB runs either in-memory or in embedded mode. Storing data in memory is useful during development as it easily allows us to re-start the database in a pre-determined state and makes it perform fast on smaller data sets. Storing data in embedded mode makes it portable. Embedded mode allows the database to be imported into to the application as a library. NeDB is designed to be partially compatible with MongoDB's JSON-based query API.

## **Installation**

npm install nedb # Put latest version in your package.json

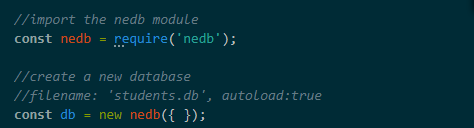
1. **Standalone NeDB**

The focus of this section is to explore implementing create – read – update – delete (CRUD) functionality on the NeDB database alone without the added complexity of a web application.

Once NeDB has been installed it can be used by the application. This example is implemented in one file, db-play.js and can be run from terminal by running: node db-play

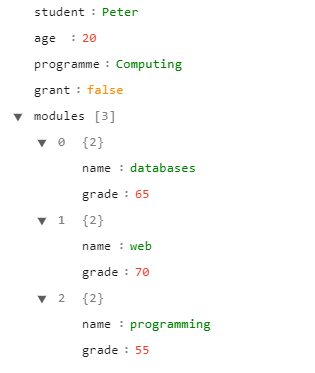
* 1. **Insert documents**

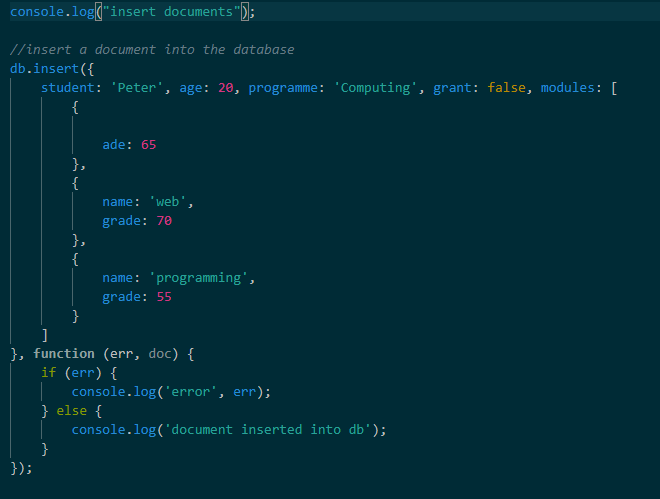
The code below imports the NeDB module and creates a database that persists data to a file students in the root of the project. If you create the database without a file name it runs in in-memory mode (also see the API in [1]).



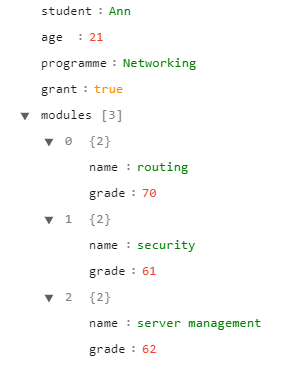
NeDB is a No-SQL datastore that stores documents in JSON format [4].

Use NeDB‘s insert method to add the following new document to the datastore.





Add another student document to the database:



At this point we can run the code: node db-play. We can’t see anything other than the console messages after inserting the documents but this gives us an indication of the correctness of the syntax.

## **Retrieve documents**

Use NeDB’s find method to retrieve documents. The first argument of the find method specifies the criteria that retrieved documents must match. If no criteria are specified all documents are returned as the second object of the error-first callback function.

The second argument is an error first callback function with 2 arguments, err and doc, representing the results of retrieving the data from the database: the first argument is an error, which may occur during database operations, the second argument comprises the retrieved documents.

The code below prints all retrieved documents to the console:

db.find({}, **function** (err, doc) {

    console.log('Showing all documents');

    if (err) {

        console.log('Error retrieving all documents', err);

    } else {

        console.log("Retrieved all documents", doc);

    }

});

Notice that NeDB creates a unique id (\_id) for each document in the datastore.

In order to retrieve specific documents we can use a filter in the find method to specify criteria in that retrieved documents have to fulfil. The method below retrieves the document with a student – value “Ann”.

db.find({student: 'Ann'}, **function** (err, doc) {

    console.log('Retrieving document(s) "Ann"');

    if (err) {

        console.log('Error retrieving Ann: ', err);

    } else {

        console.log(doc);

    }

});

The method below retrieves all documents matching the module name “security”. The module name is specified within the modules array. Use the dot notation to search within a nested document structure (also see queries in MongoDB [2]).

db.find({"modules.name": 'security'}, **function** (err, doc) {

    console.log('Retrieving document(s) with module "security"');

    if (err) {

        console.log('error: ', err);

    } else {

        console.log(doc);

    }

});

When searching for more than one criterion combine the criteria inside the first argument of the find method. The example below retrieves all documents with programme = Computing and grade = 70. Because grade is a property of the modules array we need to prefix it with modules as modules.grade.

db.find({"programme": 'Computing', "modules.grade":70}, **function** (err, doc) {

    console.log('Retrieving document(s) with programme - Computing and grade - 70');

    if (err) {

        console.log('Error retrieving document(s) with programme - Computing and grade - 70: ', err);

    } else {

        console.log(doc);

    }

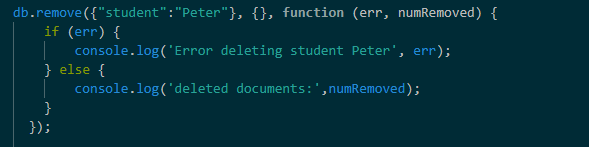
});

## **Delete documents**

NeDB’s remove method is used to delete documents. remove can have 3 arguments: query, options and callback.

* query is the same as used for finding and updating
* options when specified as “multi” allows the removal of multiple documents if set to true. The default value is false.
* callback is optional; the signature contains the database error (err) and the number of deleted document (numRemoved).

In the example below the first argument is of remove is used to identify criteria that removed documents must fulfil. All documents matching the query may be removed. The second argument, multi, which is empty in the example below indicates whether multiple deletions are permissible. The default is false. The third argument is an error-first callback function.



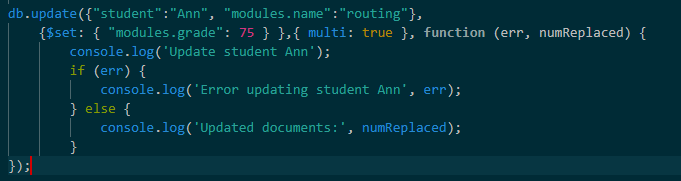
## **Update documents**

NeDB’s update method is used to update documents. update has 4 arguments:

* query identifies the document(s) to be updated,
* update specifies the updates to the document(s),
* options indicates whether multiple documents matching the criteria in query can be updated as well as other options (see API [1]),
* an optional callback function.

The identifying query is similar to the one used in the find and remove methods. The second argument defines the updates to documents inside $set: {}

The example below updates all documents matching student = Ann and modules.name = routing to set the grade of routing to 75. It allows multiple updates.



**NB**: when updating multiple keys combine these inside the one {$set:{}} object, e.g.

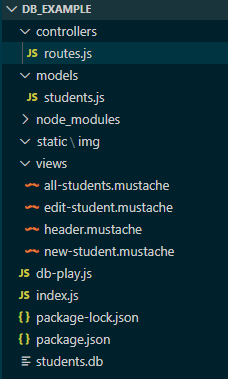
{ $set: { "age": 23, "programme": "Computing" } }

1. **Using NeDB in a web application**

Having explored NeDB we can now integrate it into a web application. This example is a minimal Model-View-Controller (MVC) type web application with the following files:

* index.js is the application entry point that defines the application’s properties,
* views are rendered from Mustache templates located in ./views,
* routes are defined in ./controller/routes.js and
* the operations on the database are implemented in ./model/students.js and
* static assets, such as icons are located in ./static/ .

The application’s structure is visualized below.



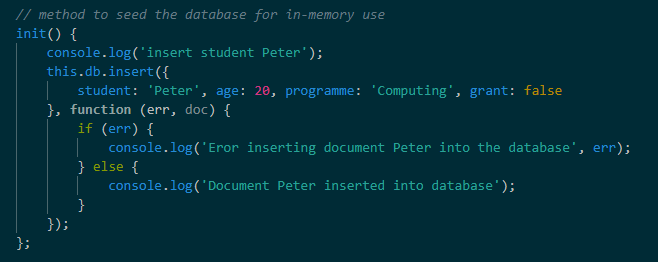
## **The model**

The database is set up in the model part of the application ./models/student.js . The model exports a class DAO. The constructor allows setting up the database in-memory, i.e. without a database file. In-memory use is useful during development because the database is loaded from scratch in the state defined (here by running the init method) rather than carrying all previously inserted documents. The database can also created to run in embedded mode by proving the filename and autoload:true in the constructor. The underlying syntax is the same as in the db-play.js standalone example. The example below sets up the database and some method stubs.

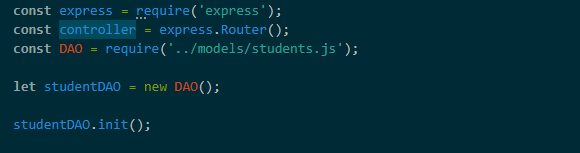


Application functionality is implemented by the methods in this class, which are exposed by the DAO. Each of the methods implements a required operation on the datastore.

The init method shown below seeds the database for use during the development process. The core of the method is an insert operation on the datastore.



The model is instantiated and the methods are called where the functionality is needed, e.g. when the requests are handled in ./controllers/routes.js .

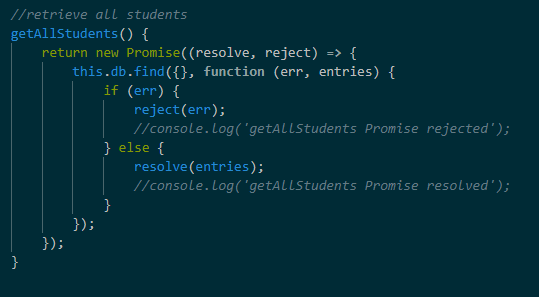


## **Show all students**

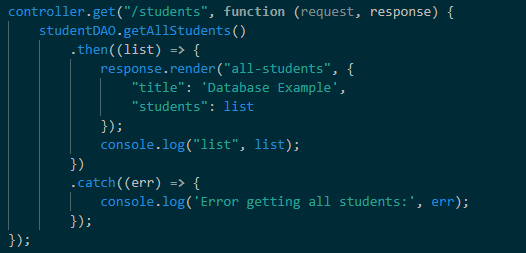
This involves:

* A database operation retrieving all students.
* A request handler that responds with the page showing all students. This is produced by rendering a Mustache template with the data retrieved from the database.
* The Mustache template for the page.

The model exposes a function that retrieves all data: getAllStudents. The core is a find operation on the database without any filters, which retrieves all documents. The data is returned as a Promise object, that, when resolved, returns the retrieved entries. Otherwise it returns the error.



The above method is used when rendering GET requests for the URL: /students. Request handling is set up in the controller part of the application (./controllers/routes.js).



The request handler runs the getAllStudents method. Then, i.e. when the Promise has been resolved or rejected, the Mustache render function is called to render the template all-students.mustache with the data document with the keys title and students. The retrieved data is the value for the key students. “Database Example” is provided as a value for the key “title”. Shown below is an example of the all-students.mustache template.

In order to utilize the data the mustache template should have keys for title and a section (list) students. Each student in the students list has data for the keys name, age, programme and grant. The supplied code contains logs that when uncommented allow you to examine structure of the retrieved documents.

<html lang="en">

<head>

  {{>header}}

</head>

<body class="container">

  <h1 class="display-4">{{title}}</h1>

  <h2 class="display-4">List of Students</h1>

    <div class="text-left" style="margin-bottom: 1em;">

      {{#students}}

      <div class="card">

        <div class="card-body">

          <div class="card-title">

            Student: {{student}}

          </div>

          <div class="card-text">

            <div class="card-title">Programme: {{programme}} </div>

            <div class="card-text">Age: {{age}}</div>

            <div class="card-text">Grant: {{grant}}<div>

          </div>

        </div>

      </div>

      {{/students}}

    </div>

</body>

</html>

## **Delete students**

To implement delete functionality we need to indicate which student is to be deleted. The list of students are shown on the /students page. We can simply add a delete button or icon to each student. Clicking the icon generates a request that triggers the delete operation for this student.

The following modification are made to the code:

* Add a delete icon to each student on the students.mustache template.
* Add a request handler for delete requests to the controller.
* Add a delete method to the model.

### **Add a delete icon to the students page**

This example uses Font Awesome icons [3] that are imported via a link. The link is added to the head section of the template in addition to a boostrap link. Bootstrap is used for styling of the pages.

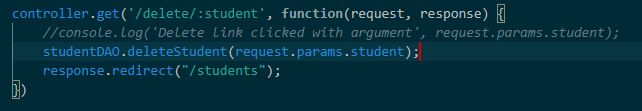


Shown below is a delete icon that is then rendered as a link. The href attribute of the link creates a GET request to the delete URL. The URL includes a property of the student object that can be extracted when the request is handled (see also lab 9 challenge) and which passes the information of which student is to be deleted to the operation.

         <a href="delete/{{student}}"><i class="fa fa-trash fa-lg"></i></a>

### **Add a request handler to the controller**

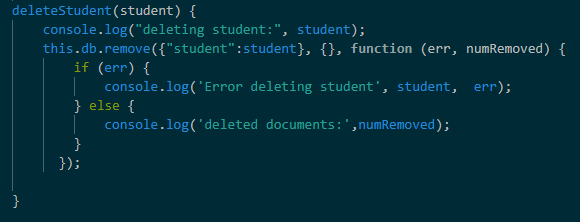
The request to delete/<student\_name> is handled in the controller. In the request handler a property identifying the student is extracted from the request parameters (request.params.student, also see lab 9 challenge) and passed to the deleteStudent method of the model. After the operation has occurred another request to the /students page is generated in order to show the user a new list of students (not showing the deleted student).



### **Add a delete function to the model**

The model exposes a deleteStudent method, which implements the delete functionality on the datastore. The parameter of the function is the name of the student to be deleted.

The implementation uses NeDB’s remove method to delete the student from the database, similar to what was implemented in the standalone database example. NeDB’s remove method uses the passed argument identifying the student as the filter.



## **Update students**

To implement update functionality we need to indicate which student is to be updated. This can be done in a similar way to deleting students by adding a clickable edit icon to the /students page.

Clicking the icon generates a GET request, the response to which contains the update page (/edit/\*). The update page essentially is a form that allows the user to update this student’s record. Submitting the form generates a post request to the /edit URL. Handling the post request updates the data in the database. After this operation the data is read again to show the user a list of all students including the updated student.

The following modification are made to the code:

* Add an edit icon to each shown student on the students.mustache template.
* Add a GET request handler for the edit URL to the controller. The response returns the update page.
* Create the update form.
* Add a POST request handler for /edit to the controller. The response updates the database and shows the modified students page.
* Add an update method to the model.

### **Add an edit icon to the students page**

The edit icon is another Font Awesome icon [3]. The icon is rendered as a link. The href attribute of the link creates a GET request to the /edit/\* URL. The URL includes a property of the student object that can be extracted from the request parameters when the request is handled (see also lab 9 challenge) and which passes the information of which student is to be updated to the database operation.



### **Add a GET request handler to the controller**

When the icon is clicked a get request to the /edit URL is generated, which needs to be handled in the controller. The student property is extracted from the request parameters (request.params.student). The response returns the update page, here edit-student.mustache.

controller.get('/edit/:student', **function**(request, response) {

*//console.log('Edit get link clicked for', request.params.student);*

    response.render("edit-student", {

            "title": "Database Example",

            "student": *request.params.student*

    });

})

The supplied code contains some further additions to pre-set the existing values on the page by looking these up in the datastore and rendering the template with these values. In a larger example this is more easily achieved with sessions.

### **Add an update page**

The core of the update page is a form with input fields for the new values. Submitting the form generates a post request to the update URL which triggers the database operation. This this small example the name of the student cannot be edited as it is used to identify the student.

<html lang="en">

<head>

{{>header}}

</head>

<body class="container">

<h1 class="display-4">{{title}}</h1>

<h2 class="display-4">Edit Student</h2>

<form method="post" role="form" >

  <div class="form-group">

    <label for="student">Student: </label>

<input type="text" class="form-control-plaintext" id="student"

name="student" value="{{student}}" >

  </div>

   <div class="form-group">

    <label for="age">Age</label>

<input type="number" min="5" max="120" class="form-control" id="age"

name="age">

   </div>

   <div class="form-group">

    <label for="programme">Programme</label>

<input type="text" class="form-control" id="programme"

name="programme">

   </div>

   <div class="form-group">

    <label for="grant">Grant</label>

    <input type="radio" name="grant" id="grant" value="true">Yes

    <input type="radio" name="grant" id="grant" value="false">No

  </div>

  <div class="form-group">

    <input type="submit" value="Save student details" class="btn btn-primary">

  </div>

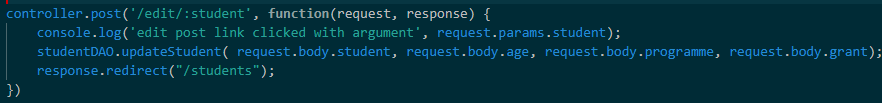
</form>

</body>

</html>

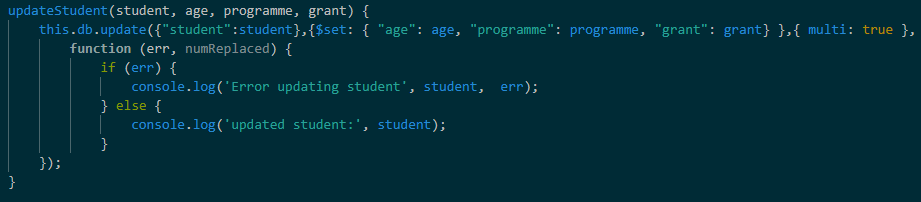
### **Add a POST request handler to the controller**

Add a request handler for POST requests to /edit to the controller. A property identifying the student is extracted from the link and passed to the updateStudent method of the model. After the operation has occurred another request to the /students page is generated in order to show the user the new list of students showing the updated student.

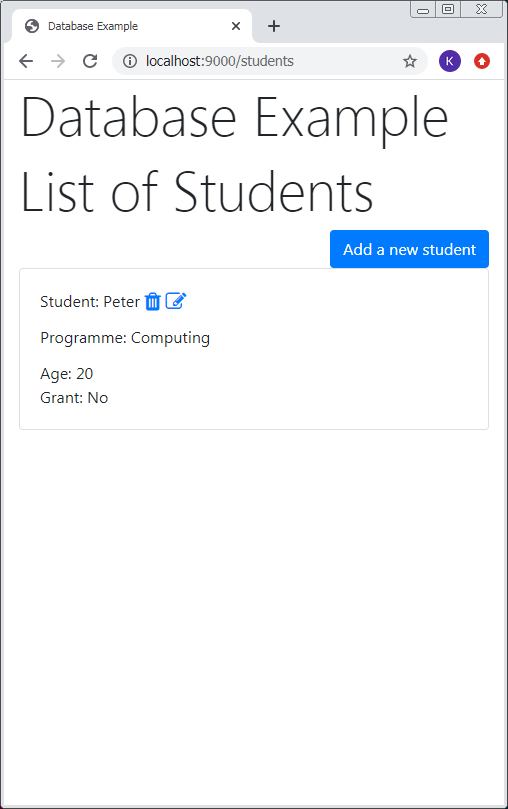
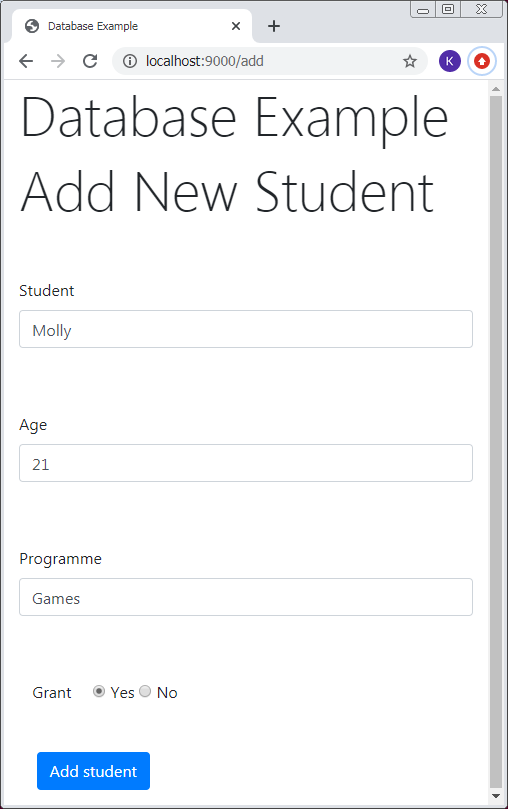
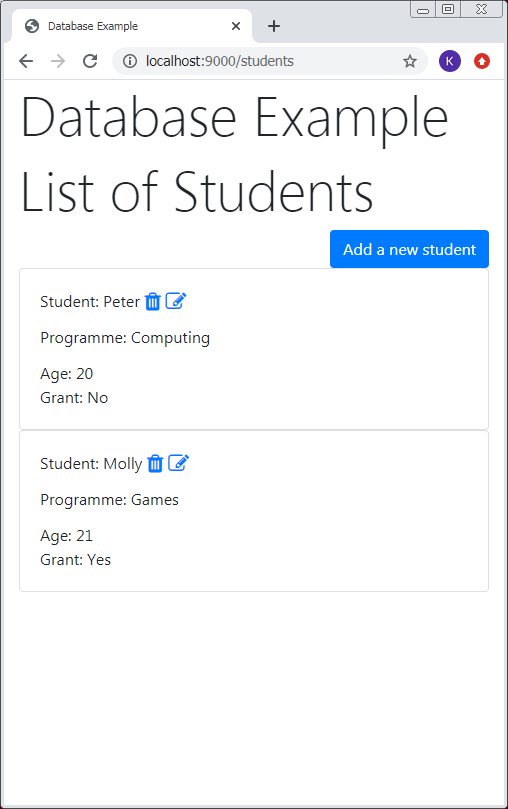
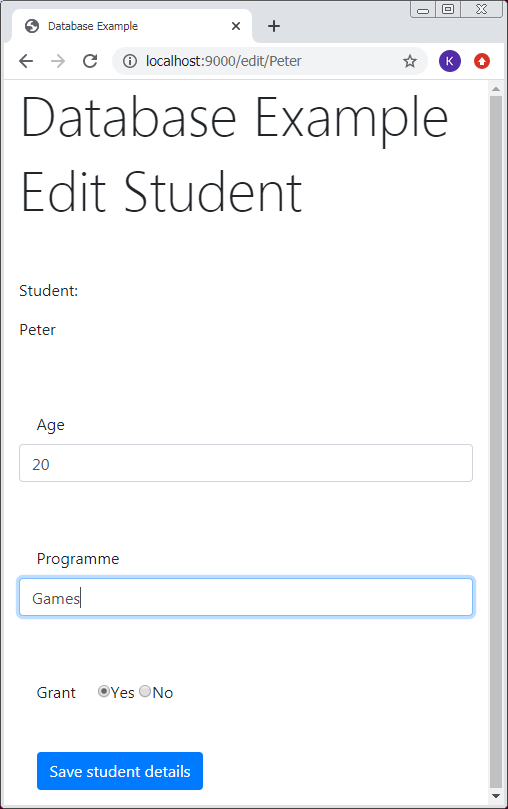
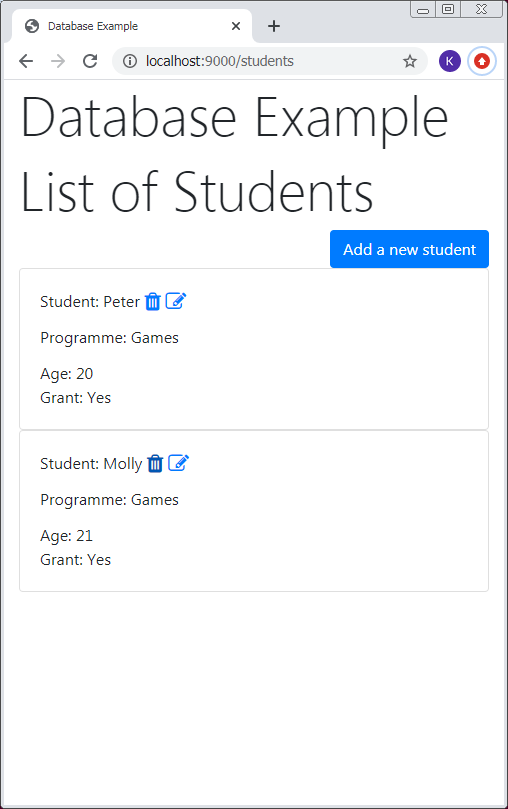
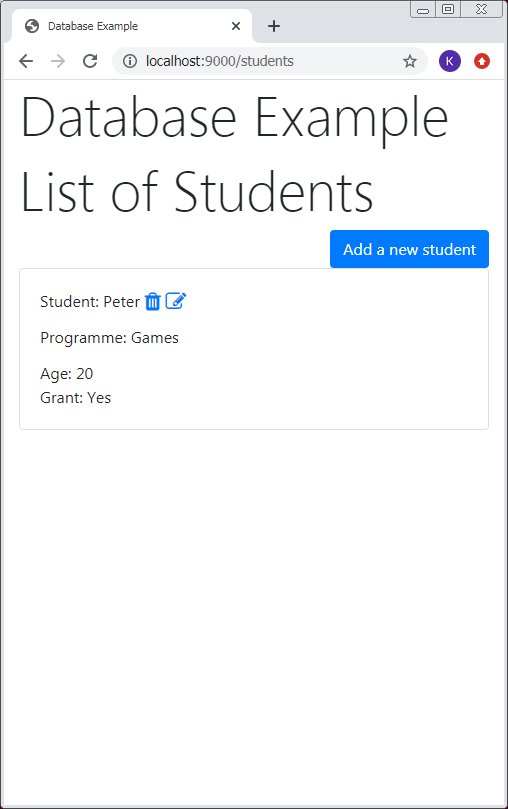


### **Add an updateStudent method to the model**

The model exposes the updateStudent method with parameters necessary for the update operation. This this small example the name of the student cannot be edited as it is used to identify the student. The implementation uses NeDB’s update method to update the student in the datastore, also see the standalone database section.



1. **Images**

# **References**

[1] https://github.com/louischatriot/nedb

[2] https://docs.mongodb.com/manual/core/document/

[3] https://fontawesome.com/

[4] https://en.wikipedia.org/wiki/JSON