Assignment 4
RESTful Web Service
Datanet 2015

Christian Hohlmann Enevoldsen

University of Copenhagen

A brief introduction to RESTful web services is given. Next a RESTful TODO web server is described. Here the design choices, frameworks used and the basics of how to use it is covered. Afterwards the web service's limitations are outlined and finally we wrap up with the tests done in this experiment.

Categories and Subject Descriptors:

General Terms: Programming, Experimentation, API, REST

Additional Key Words and Phrases: HTTP, SERVER, CRUD

1. INTRODUCTION

In these days almost every application uses resources. Most applications these days are clients for a service. The client's responsibility is to present data to the user and to tell its service about changes. The service's responsibility is to create, read, update and delete data (CRUD). One example is the todo we are going to talk about in the following sections. The todo service has a persistent storage of resources such that the client doesn't need to store the todos locally, which results in the application to be very portable. All the data is stored on the service and is accessible through an RESTful API. That means that no matter which client you use or where you use it you will always have your data if you have an internet connection.

2. DESIGN

The following modules/frameworks is used in the Server

—**Node.js**: Non-blocking I/O platform for building the server.
—**Express**: Used for creating routes and the HTTP Server
—**Mongoose**: ODM - object data mapping.

The todo REST server is written in Node.js, and the todo items is stored in MongoDB. Express is used to ease up routing and server creation.

The routes are:

—**/** - for the static index file
—**/todo** - endpoint for the todo resources.

Mongoose is an ODM and it is used to connect to the Mongo Database. With Mongoose comes Schemas. Schemas helps define the object data mapping, and eases up the CRUD. There's only one schema defined in the todo server. The todo Schema has the three properties: id, completed and text.
The id is however not an Integer in this case since MongoDB is designed to use Object ID's instead.

The routes can now easily use the built in functions for CRUD operations on the todo Schema, and safely ignore any kind of database communication since that's covered in Mongoose.

2.1 Using the API

Creating TODO items
To create a new TODO item a POST request is issued with the following details

```
POST /todo/ HTTP/1.1
Host: localhost:3000
content-type: application/json
Content-Length: 33

{
  "todo" {
    "text": "value"
  }
}
```

You can optionally set the completed property. Defaults to false
The response on success is the newly created TODO

```
HTTP/1.1 201 Created
Content-Type: application/json;
Content-Length: 82
Connection: keep-alive

{
  "todo": {
    "text": "hello world",
    "_id": "556f32f03d8f37d8180ab58e",
    "completed": false
  }
}
```

Getting TODO items
To get all todo items a GET request is issued with following details:

```
GET /todo/ HTTP/1.1
Host: localhost:3000
Content-type: application/json
```

On success the response will be:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 234

{
  "todos": [
    {
      "_id": "556f32f03d8f37d8180ab58e",
      "text": "hello world",
      "completed":false
    },
    ...
  ]
}
```

To get a specific todo item a GET request is issued with following details:

```
GET /todo/556f32f03d8f37d8180ab58e HTTP/1.1
Host: localhost:3000
Content-type: application/json
```

On success the response will be:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 82

{
  "todo": {
      "_id": "556f32f03d8f37d8180ab58e",
      "text": "hello world",
      "completed":false
    }
}
```

Updating a TODO item

To update a TODO item a PUT request is issued with the following details

```
PUT /todo/556aeb081eba49a108beffa3 HTTP/1.1
Host: localhost:3000
content-type: application/json
Content-Length: 28

{
  "todo": {
      "completed": true
    }
}
```

On success you get the following response

```
HTTP/1.1 204 No Content
```

The content-length is omitted, since newer RFC specs suggests that.

Deleting a TODO item

To delete a TODO item a DELETE request is issued with the following details

```
DELETE /todo/556aeb081eba49a108beffa3 HTTP/1.1
Host: localhost:3000
content-type: application/json
```

On success you get the following response

```
HTTP/1.1 204 No Content
```

Errors
If a required property is omitted you can expect the following error response

```
HTTP/1.1 422 Unprocessable Entity
Content-Length: 46
Content-Type: application/json

{
  "error": {
    "message": "Property todo requried"
  }
}
```

For bad syntax expect the following response

```
HTTP/1.1 400 Bad Requst
Content-Length: 46
Content-Type: application/json

{
  "error": {
    "message": "Unexpected end of input"
  }
}
```

For bad URI’s you get a 404

```
HTTP/1.1 404 Bad Request
Content-Length: 76
Content-Type: application/json

{
  "error": {
    "message": "The requested todo item does not
        exist"
  }
}
```

3. LIMITATIONS

Even though the service only does some trivial logic and the code base is very small there are some major limitations and some minor, which may be features rather than limitations.

3.1 Limits

It is not possible to limit the number of TODO items returned from a request. Because of that you might get a ton of items in response.

3.2 Authentication

As of now no authentication is required when using the API. This means that the server is vulnerable to attacks and misuse. An attacker can easily delete all items in the database by simply getting the list of all items and afterwards delete one by one.

3.3 Filtering and searching

It is not possible to filter or search for items. For instance; you might only want to list all completed TODO items, however this requires manual data manipulation on the client side right now. Other than that you cannot search for TODO items containing specific phrases. This results in overhead in data transfer that could have been omitted by doing the operations on the server.

3.4 Database

The server relies on MongoDB running locally on the hosting computer. However the configuration can easily be modified to use another database path. Although you might change the database path you cannot use another kind of database without major refactoring.

3.5 Errors

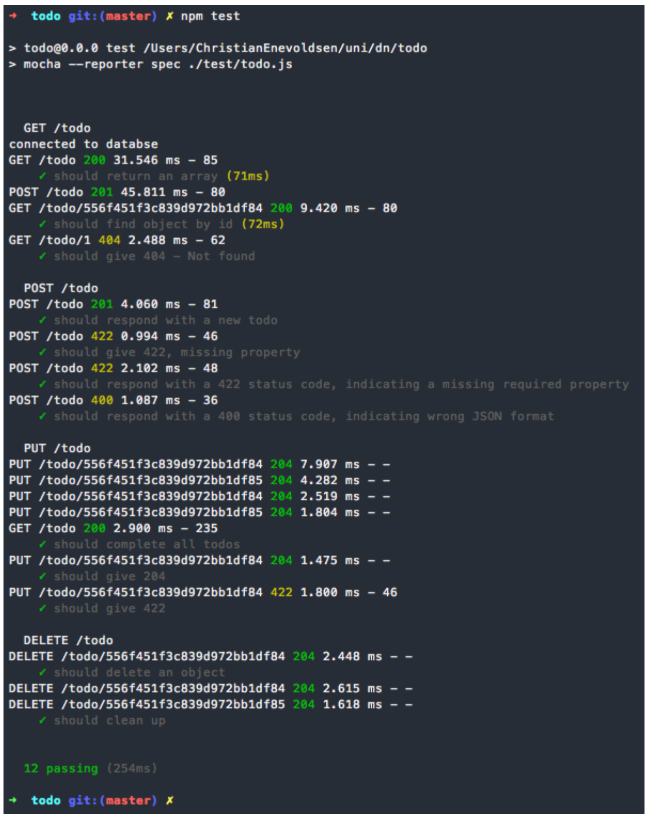
Some errors might have been overlooked - some bugs might occur.

4. TESTS

To make sure that the server is stable and everything works as expected a static webpage is used to as a client to the server. The client is a simple website that enables a user to create, modify, delete and see TODO items. The client uses the RESTful API behind the scenes. Arguably any errors in the server would be visually reflected to the user. An interactional test is useful up to some point, which includes testing expected results and errors.
Unit tests is written to make sure that everything is indeed working as expected. Although the interactional website is a great way to test the server, it could easily be discarded in favour of unit tests.
The unit tests are written with Supertest which is a module that other than being able to do unit tests also is useful for testing asynchronous code. The appendix contains a screenshot of the test results.

APPENDIX

A. TEST



```
➜  todo git:(master) ✗ npm test

> todo@0.0.0 test /Users/ChristianEnevoldsen/uni/dn/todo
> mocha --reporter spec ./test/todo.js



  GET /todo
connected to databse
GET /todo 200 31.546 ms - 85
    ✓ should return an array (71ms)
POST /todo 201 45.811 ms - 80
GET /todo/556f451f3c839d972bb1df84 200 9.420 ms - 80
    ✓ should find object by id (72ms)
GET /todo/1 404 2.488 ms - 62
    ✓ should give 404 - Not found

  POST /todo
POST /todo 201 4.060 ms - 81
    ✓ should respond with a new todo
POST /todo 422 0.994 ms - 46
    ✓ should give 422, missing property
POST /todo 422 2.102 ms - 48
    ✓ should respond with a 422 status code, indicating a missing required property
POST /todo 400 1.087 ms - 36
    ✓ should respond with a 400 status code, indicating wrong JSON format

  PUT /todo
PUT /todo/556f451f3c839d972bb1df84 204 7.907 ms - -
PUT /todo/556f451f3c839d972bb1df85 204 4.282 ms - -
PUT /todo/556f451f3c839d972bb1df84 204 2.519 ms - -
PUT /todo/556f451f3c839d972bb1df85 204 1.804 ms - -
GET /todo 200 2.900 ms - 235
    ✓ should complete all todos
PUT /todo/556f451f3c839d972bb1df84 204 1.475 ms - -
    ✓ should give 204
PUT /todo/556f451f3c839d972bb1df84 422 1.800 ms - 46
    ✓ should give 422

  DELETE /todo
DELETE /todo/556f451f3c839d972bb1df84 204 2.448 ms - -
    ✓ should delete an object
DELETE /todo/556f451f3c839d972bb1df84 204 2.615 ms - -
DELETE /todo/556f451f3c839d972bb1df85 204 1.618 ms - -
    ✓ should clean up


  12 passing (254ms)

➜  todo git:(master) ✗
```