Hapi Basics

In this assignment, you will use Hapi to build a simple RESTful API that accesses resources stored in a PostgreSQL database.

1 Read

Read chapters 1 and 2 from the Harrison text (*Hapi.js in Action*). Note that although the examples in Harrison use (the amazing) SQLite for database management, you will continue to use PostgreSQL and access it using Knex.

2 Prepare

Set up a new project for this assignment.

2.1 Initialize a Node Project

Create a new directory for the project, make it your current working directory, and set up for Node development.

```
mkdir hapi-intro                # Use whatever name you want
cd hapi-intro
yarn init
```

2.2 Install Node modules

Install Hapi, support packages, Knex, and the PostgreSQL driver

```
yarn add @hapi/hapi hapi-pino blipp knex pg
```

2.3 Maybe Install Postman

The Postman API development tool should *already* be installed on all the lab machines. If you're using a lab machine, skip to Section 2.4.

If you are using your own computer, install Postman. It is free to use for most purposes, including in class.

1. In your browser, navigate to the Postman home page.
2. Based on your operating system, click the appropriate button to *Download the free Postman App*.
3. Follow the instructions for your operating system.

2.4 Configure Database Connection

Configure access to the `dvdrental` database. As in previous KNEX labs, create a `knex` object for this purpose. You will create just one source file for this lab; call it `simple-hapi.js`. Add the following code to your source file.

```
const knex = require('knex')({
  client: 'pg',
  connection: {
    host: 'faraday.cse.taylor.edu',
    user: 'readonly',
    password: 'nerds4christ',
    database: 'dvdrental'
  }
});
```

Note that for this introductory lab, were just going to use KNEX and not bother installing OBJECTION. When you implement endpoints that use the database, you can use KNEX directly to retrieve information from the database.

2.5 Set up Basic Hapi Server

Add to your source file the following template for a basic Hapi server.

```
const Hapi = require('@hapi/hapi');

const init = async () => {
  // Create a new Hapi server
  const server = Hapi.server({
    host: 'localhost',
    port: 3000
  });

  // Output endpoints at startup.
  await server.register({plugin: require('blipp'),
                         options: {showAuth: true}});

  // Log stuff.
  await server.register({
    plugin: require('hapi-pino'),
    options: {
      prettyPrint: true
    }
  });

  server.route([
    {
      method: 'GET',
      path: '/',
```

```
      handler: (request, h) => {
        return 'Hello, Hapi';
      }
    },
  ]);

  // Start the server.
  await server.start();
}

// Catch promises lacking a .catch.
process.on('unhandledRejection', err => {
  console.error(err);
  process.exit(1);
});

// Go!
init();
```

2.6 Verify Hapi Server

Execute your new server by running it as a NODE application.

```
node simple-hapi.js
```

The server should respond with a message indicating Hapi is listening on `http://<some host name>:3000`. Verify the operation of the server using Postman.

1. Check that the drop-down list of methods shows `GET`.
2. Enter `localhost:3000` in the address bar.
3. Hit SEND and check that the response body is `Hello, Hapi`.

3 Add Routes

Add routes to your Hapi server as follows.

1. A collection route that lists country resources. It should respond to a `GET` request to the URL

   ```
   localhost:3000/countries
   ```

   with a JSON list containing the `country_id` and `country` name of all the countries in the database. For example:

```
[
  {
    "country_id": 1,
    "country": "Afghanistan"
  },
  {
    "country_id": 2,
    "country": "Algeria"
  },
  "... remaining countries ..."
]
```

2. An element route that fetches information on a single film resource. It should respond to a `GET` request to URLs like

   ```
   localhost:3000/films/172
   ```

   with the `film_id`, `title`, and `rental_rate` of the film with the indicated ID. Of course, this should work for *any* valid film ID. For example,

```
{
    "film_id": 172,
    "title": "Coneheads Smoochy",
    "rental_rate": "4.99"
}
```

   IMPORTANT: because this is an *element* route, make sure the resulting value is a *single* object—not an array *containing* a single object.

3. A collection route that returns the `film_id`, `title`, and `rental_rate` of film resources. It should support two variations.

   (a) It should respond to a `GET` request to

   ```
   localhost:3000/films
   ```

   with an array of objects for *all* films in the database.

   (b) It should also support searching for films whose title matches a pattern. For example, a `GET` request to

`localhost:3000/films?title=Sa%`

should return the `film_id`, `title`, and `rental_rate` of films whose title matches the SQL `LIKE` pattern `Sa%` (that is, all films whose title begins with the letters `Sa` followed by any other characters).

Note that the search string is *not* part of the film resource itself; it is not an attribute of a film but instead, an artifact of our query. That's why we separate it out as a *query parameter* in the URL.

To access this query parameter from within your Hapi handler function, use code like the following.

```
1 const searchKey = request.query.title;
```

For the the previous URL, this code would set `searchKey` to `Sa%`.

4 Submit

Submit your `simple-hapi.js` source file to the course web site, including the handler functions that implements all the routes described in Section 3.