Build your first Node.js website, Part 3

Ruth Willenborg

February 25, 2015
(First published July 29, 2014)

Do you want to build a dynamic website with Node.js but are unsure where to start? This three-part article introduces you to Node.js development step by step, with no software installation required. Using only a web browser, you create a Node.js application, write server-side JavaScript, display dynamic data on HTML pages, and store data in a Redis database.

To view the introductory video **Build your first Node.js website, Part 3** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

This is the third part of a three-part article that introduces you to Node.js development step-by-step, with no software installation required. In Part 1 and Part 2, you used IBM Bluemix DevOps Services and IBM Bluemix™ to bring up a web server and a simple HTML front end, and to get data flowing from the server to the browser. All that's left is to persist the data into a database.

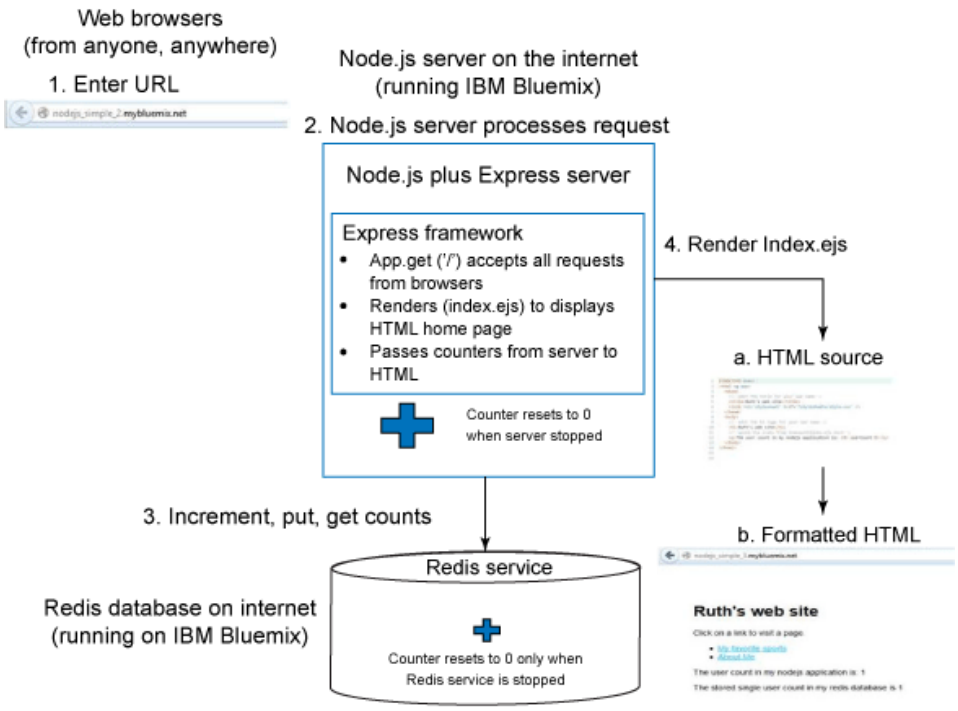
Bluemix gives you ready access to great database services, with no installation or management required. For storing a counter, my many choices included multiple relational databases and NoSQL options such as MongoDB, Cloudant, and Redis.

> *“ The Redis service's lifecycle is independent of the application's, so data stored in the service persists across application restarts. ”*

This part starts with a new Bluemix DevOps Services project, so it is not necessary to complete the exercises in Part 1 and Part 2 before proceeding.

The sample application stores data into a separate Redis service managed by Bluemix. This service's lifecycle is independent of the application's, so data stored in the service persists across application restarts.

To access a Bluemix service, a fair amount of setup is required in the code. You'll use a starter project with all the Redis service setup in place. The new application flow looks like this:



What you'll need

- A Bluemix account.

Run the app
Get the code

Step 1: Create and deploy your copy of the HTML-database application

1. Log in to Bluemix DevOps Services.
2. Click this article's **Get the code** button above.
3. In Bluemix DevOps Services, click the **FORK PROJECT** button on the menu to create your own copy of the code, provide a unique name, and click the **CREATE** button. Click the **EDIT CODE** button to begin working with the application.
4. Take a quick look at the relevant files:
   - In package.json, notice the additional dependencies for Redis: `"redis": "*"`.
   - In manifest.yml, see the specifications for creating the Redis service:

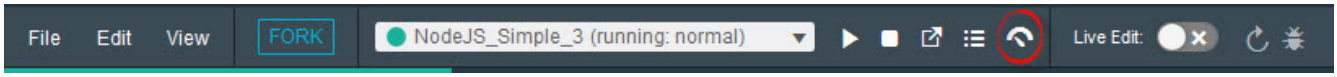
     ```
     services:
       redis-5a659:
         label: redis
         provider: core
         version: '2.6'
         plan: 100
     ```

   - In app.js, search for `redis` to see all the changes that were made to interface to the Redis service.
5. Deploy the application using the same process you used for the previous two parts of this article.

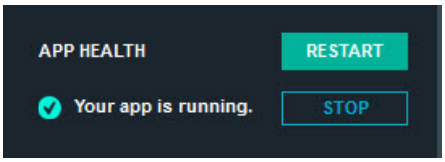
6. Click the URL link in the runbar to open the application in a new browser tab. In the running application, you can see two user counts that both start at 1: the user count in the Node.js server (as configured in Part 2) and a new count that's stored in the database.
7. Refresh the browser page and watch the counts increment with each refresh. Now — after the first deployment — the user count in the server and the user count in the database remain identical.

Step 2: Observe database persistence across restarts

1. To manage your application, go to your application dashboard in Bluemix.



2. On the application dashboard, click the **Stop** button. After the application is stopped, click the **Restart** button



3. Follow the Routes link back to the restarted application. Notice that the user count in the Node.js application is reset to 1 but that the count from the database has incremented from the number that the page displayed before you stopped and restarted the application. This happens because the Redis service has a separate lifecycle from the Node.js application.

Step 3: Work with the Redis service on Bluemix

You use the Bluemix dashboard to work directly with Bluemix services:

1. Click the gear-shaped button near the top right, and then click **Delete App**.
2. In the **Are you sure...** dialog box, select the check box to delete the Redis service (redis-5a659) and click **OK**. By deleting the service along with the application, you delete the database counter that was persisted across application restarts.
3. Back in Bluemix DevOps Services, use the sideways triangle button to redeploy the application. Because you deleted the Redis service, this redeployment creates a new instance of the service.
4. Browse to the running application and observe that the user count stored in the Redis database is also reset to 1.

Step 4: Add data to the database

Now you'll update the code to store a counter that increments by two into the Redis service, and display that counter on the HTML page. Because calls to the Redis service are asynchronous, you must insert the new calls into a nested structure to ensure proper ordering. (Another option is to add explicit async libraries — a topic that's outside of this article's scope.)

1. Open app.js and find the `client.incrby` line. This statement increments a `DBuserCountBy1` variable in Redis and returns the updated value in the `replyBy1` variable.

2. Add a line to increment a database variable, `DBuserCount2`, by two and return the value as `replyBy2`:
```
client.incrby("DBuserCountBy2", 2, function (err, replyBy2);
```

   To ensure that the calls to Redis are synchronized, this statement must be nested inside the increment of the `DBuserCountBy1`:
```
client.incrby("DBuserCountBy1", 1, function (err, replyBy1) {
   client.incrby("DBuserCountBy2", 2, function (err, replyBy2) {
```

   Be sure to add the closing `});` also.
3. Update the `res.render` statement to pass the `DBuserCountby2` that's returned in the `replyBy2` variable from the Redis function call:
```
res.render('index', {userCount: userCount, DBuserCountBy1: replyBy1, DBuserCountBy2: replyBy2});
```

   The app.get section of your code should now look like this:
```
app.get('/', function(req, res) {
  userCount = userCount + 1;
  // Increment the value in the database and render the results
  client.incrby("DBuserCountBy1", 1, function (err, replyBy1) {
   client.incrby("DBuserCountBy2", 2, function (err, replyBy2) {
    res.render('index', {userCount: userCount, DBuserCountBy1: replyBy1, DBuserCountBy2: replyBy2});
    });
  });
});
```
4. Update the index.ejs HTML page to render to `DbuserCountBy2`:
```
<p>The stored user count by two in my redis database is <%= DBuserCountBy2 %></p>.
```
5. Redeploy the application and verify that all the counts are behaving as expected.

Conclusion

In this three-part article, you worked step-by-step through bringing up a Node.js server, adding a web front end with both static pages and dynamic content from the server, and then storing the dynamic data in a database. You saw how the lifecycle of the Redis database service is independent of the lifecycle of the application, enabling the user count to persist as the application is stopped and started. The entire example is done with just a web browser accessing IBM Bluemix DevOps Services and Bluemix— no installation required.

Acknowledgments

No children were harmed in the writing of this article, though several were challenged to expand from users of technology to creators of their own technology. My thanks to my novice helpers: my son Tucker, and Carolyn Norton and her son, Patrick. And my thanks to Mike McKay and Patrick Mueller, my local Node.js gurus.

Related topics

- See IBM Bluemix in action
- Getting started with Redis service
- The SDK for Node.js runtime
- Redis
- Node.js
- Redis