Guided Exercise: Remote Debugging Containers

Use VSCodium to debug a Node.js application running inside a container.

Outcomes

You should be able to:

- Start a container that exposes the Node.js debug port.
- Attach the VSCodium debugger to an application.
- Use the debugger to step through code execution and find a bug.
- Mount the application code into the container.

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the exercise materials exist in the classroom.

```
[student@workstation ~]$ lab start troubleshooting-debugging
```

Instructions

1. Build and run a container image for the application.

   Go to the application directory.

   ```
   [student@workstation ~]$ cd ~/DO188/labs/troubleshooting-debugging/nodebug
   no output expected
   ```

   Examine the NPM scripts in the `package.json` file.

   ```
   [student@workstation nodebug]$ cat package.json
   ...output omitted...
     "scripts": {
       "start": "node index.js",
       "debug": "nodemon --inspect=0.0.0.0 index.js"
     },
   ...output omitted...
   ```

   The `start` script starts the application. The `debug` script starts the application with the `--inspect` option to enable debugging.

   The Node.js debug mode binds to `127.0.0.1` by default, which means it is available only inside the container. To connect from outside of the container, it must bind to `0.0.0.0`.

   Build a container image for the application.

   ```
   [student@workstation nodebug]$ podman build -t nodebug .
   ...output omitted...
   Successfully tagged localhost/nodebug:latest
   ```

   Create a container that forwards traffic on the `8080` and `9229` ports to the same ports inside of the container. Override the container start command so that it runs the `debug` npm script. Run the container in the background by using the `-d` option and remove the container when it stops by using the `--rm` option.

   ```
   [student@workstation nodebug]$ podman run -d --rm \
     --name nodebug -p 8080:8080 -p 9229:9229 \
     nodebug npm run debug
   ...output omitted...
   ```

   Confirm that the application started in debug mode.

   ```
   [student@workstation nodebug]$ podman logs nodebug
   ...output omitted...
   Debugger listening on ws://0.0.0.0:9229/16a2...4bf6c
   ...output omitted...
   app started on port 8080
   ```

   The application runs on port `8080` and the debugger listens on port `9229`. Podman forwards both of these ports from the host to the container.

2. Attach VSCodium to the running container and make a request to the `/echo` endpoint.

Launch VSCodium with the application project.

```
[student@workstation nodebug]$ codium .
```

If prompted, click **Yes, I trust the authors**.

Click **Run** → **Add configuration** to open a selection menu, and then click **Node.js**.

Replace the default contents with the following launch configuration and save the file.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "attach",
            "name": "Debug Nodebug",
            "port": 9229,
            "address": "localhost",
            "localRoot": "${workspaceFolder}",
            "remoteRoot": "/"
        }
    ]
}
```

The preceding configuration is available in the `~/DO188/labs/troubleshooting-debugging/nodebug/launch-config.json` file.

Click **Run** → **Start Debugging** to launch the configuration and attach the VSCodium debugger to the running application.

In the terminal window, make a request to the running application.

```
[student@workstation nodebug]$ curl localhost:8080/echo?message=hello
no output expected
```

The request does not respond as the hard-coded breakpoint was triggered.

> **NOTE**
>
> In Node.js, the `debugger` statement is a hard-coded breakpoint when running in debug mode. Outside of the debug mode, this statement has no effect.

Notice that the debugger reached the breakpoint in the `index.js` file and the execution context is displayed in the debug panel.

Observe that the local `message` variable uses the `hello` value. The application sets the variable from the `message` query parameter that you supplied to the request.

Continue the execution by clicking the **Continue** icon.

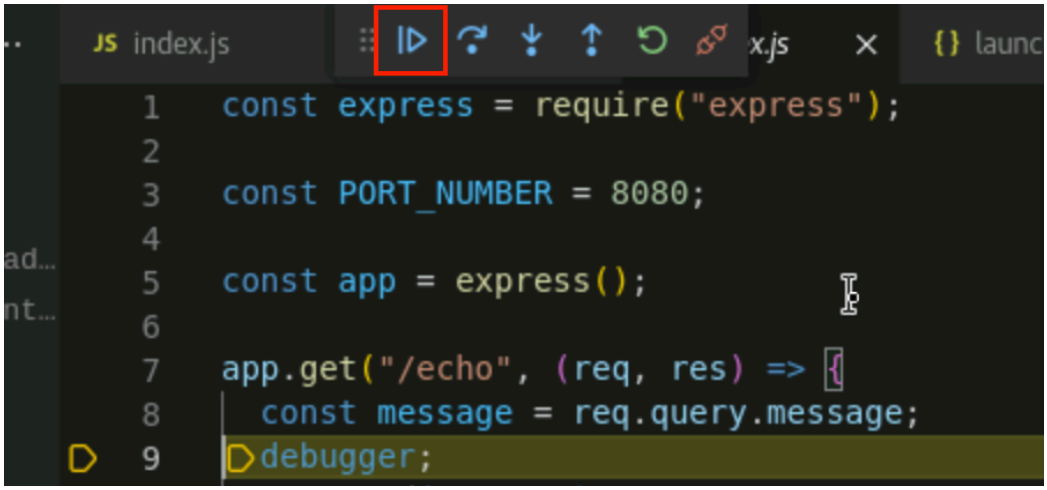

Figure 6.1: Continue program execution

In the terminal window, the `curl` request completes and echoes our message: `hello`.

3. Use the debugger to find and fix a bug in the `/snacks` endpoint.

Attach a breakpoint to the endpoint. Click the red circle next to the line in the file that assigns the `available_snacks` variable.

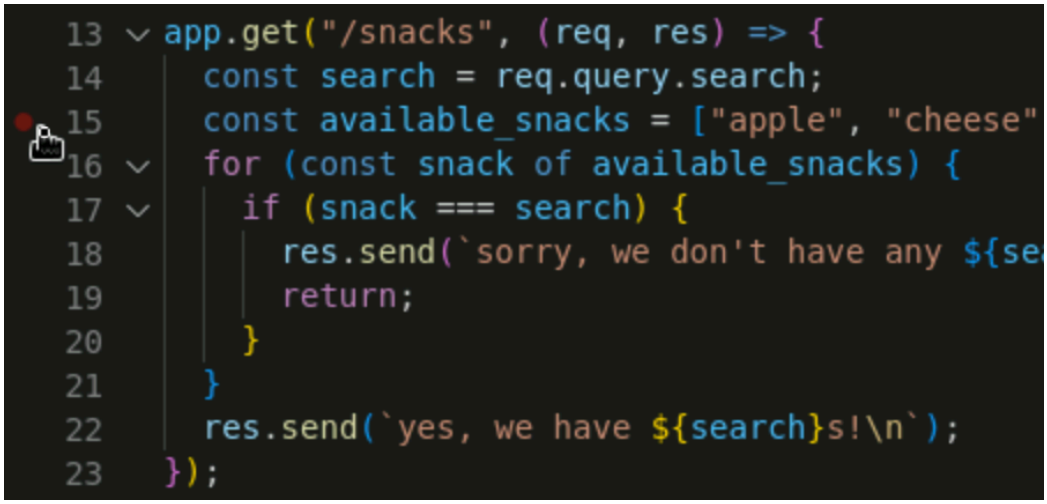

Figure 6.2: Attach a breakpoint

> **NOTE**
> The red circle does not appear until you hover over the space to the left of the line number.

A solid circle persists if you attached the breakpoint successfully.

In the terminal window, make a request to the `/snacks` endpoint.

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
```

Execution pauses because the breakpoint in the snacks endpoint is triggered.

Click the **Step Over** icon to step through the function line-by-line.

Figure 6.3: Step Over icon

Each time you click the button, the highlighted line of code is executed.

Continue to click the button until the request responds that the search has failed and the function returns.

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
sorry, we don't have any apples :(
```

This is a bug because the queried snack, apples, is in the `available_snacks` list.

Try to swap the lines containing `res.send` statements. VSCodium does not allow this because the file is read-only.

This editor buffer does not contain the `index.js` file, but instead an in-memory copy of what VSCodium is debugging.

Open the `index.js` file by clicking **View** → **Explorer** and double-clicking the `index.js` file in the explorer panel.

Change the application response when a snack is found as follows:

```
app.get("/snacks", (req, res) => {
  const search == req.query.search;
  const available_snacks == ["apple", "cheese", "cracker", "lunchmeat", "olive"];
  for (const snack of available_snacks) {
    if (snack ==== search) {
      res.send(`yes, we have ${search}s!\n`);
      return;
    }
  }
  res.send(`sorry, we don't have any ${search}s :(\n`);
});
```

Save the changes to the file.

In the terminal window, make the same request to the snack endpoint as before.

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
```

Step through the execution again and notice that the bug is not fixed. This is because only the application code on the host was updated. The container contents are unchanged.

4. Recreate and debug the container with the application code mounted into the container.

   In the terminal window, stop the container.

   ```
   [student@workstation nodebug]$ podman stop nodebug
   nodebug
   ```

   This also stops the VSCodium debug connection.

   Install the Node.js dependencies on the host machine.

   ```
   [student@workstation nodebug]$ npm install
   ...output omitted...
   ```

   Create a container with the application code mounted into the container.

   ```
   [student@workstation nodebug]$ podman run -d --rm \
     --name nodebug -p 8080:8080 -p 9229:9229 \
     -v .:/opt/app-root/src:Z \
     nodebug npm run debug
   9eba...9c81
   ```

   > **NOTE**
   >
   > Installing the Node.js dependencies in the host machine is necessary because the preceding `-v` option overrides the entire application directory from the container image, which includes runtime dependencies.

   Connect VSCodium to the debug socket by clicking the **Start Debugging** icon.

   In the terminal window, make the same request to the snack endpoint as before.

   ```
   [student@workstation ~]$ curl localhost:8080/snacks?search=apple
   ```

   Step through the execution.

   Notice that the bug is fixed because the updated code on the host was mounted inside the container:

   ```
   [student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
   yes, we have apples!
   ```

   This set up provides faster debugging by not requiring a container rebuild on every code change.

   Disconnect the debugger by clicking **Disconnect** or by closing the VSCodium window.

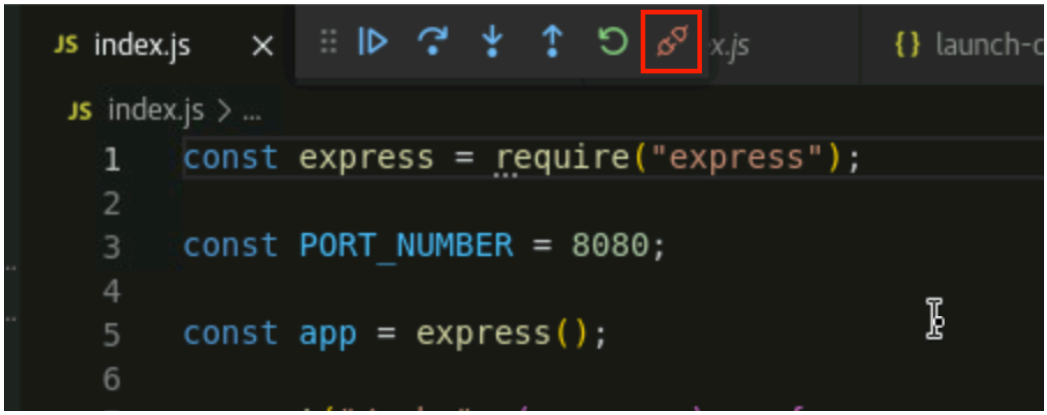

Figure 6.4: Disconnect the debugger

5. Create a container image with the updated code and verify that the bug is fixed.

   Stop and remove the running debug container.

```
[student@workstation nodebug]$ podman rm -f nodebug
nodebug
```

   Remove the `node_modules` directory from the host.

```
[student@workstation nodebug]$ rm -r node_modules
no output expected
```

   Build a container image with the fixed application code.

```
[student@workstation nodebug]$ podman build -t nodebug .
...output omitted...
Successfully tagged localhost/nodebug:latest
...output omitted...
```

   Create a container with the new image. Run the application without debug mode and without exposing the debug port.

```
[student@workstation nodebug]$ podman run -d --rm \
  --name nodebug -p 8080:8080 nodebug
8ef...eb7
```

   Make the same request to the application as before and verify that the response is correct.

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
yes, we have apples!
```

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish troubleshooting-debugging
```