labs-debug-api

Debug along with coach and follow the instructions for exercises

Jump to:

Setup Debugging Profiles

Add debugging configurations to settings.json for Visual Studio Code

1. From menu
   - Windows: **File - Preferences - Settings**
   - Mac: **Code - Preferences - Settings**
2. Search for launch
3. Click on **Edit in settings.json**
4. In the `settings.json` file, find the `launch` key. Within the launch key, there will be another key for `configurations`. Add the `configurations` array as outlined below. This adds configurations for node.js app, React app, and unit tests.

   NOTE: Do not copy paste the snippet to replace your entire settings.json file!! We are only concerned with the `launch` and `configurations` key.

   Windows:

```
"launch": {
"version": "0.2.0",
"configurations": [
    {
        "name": "React App",
        "type": "chrome",
        "request": "launch",
        "url": "http://localhost:3000",
        "webRoot": "${workspaceFolder}"
    },
    {
        "name": "React Unit Tests (create-react-app)",
        "type": "node",
        "request": "launch",
        "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/react-
scripts",
        "args": [
            "test",
            "--runInBand",
            "--no-cache",
            "--env=jsdom"
```

```
        ],
        "cwd": "${workspaceRoot}",
        "protocol": "inspector",
        "console": "integratedTerminal",
        "internalConsoleOptions": "neverOpen"
    },
    {
        "name": "Node Attach by Process ID",
        "type": "node",
        "request": "attach",
        "processId": "${command:PickProcess}",
        "skipFiles": [
            "<node_internals>/**"
        ]
    },
    {
        "name": "Node Launch App",
        "type": "node",
        "request": "launch",
        "skipFiles": [
            "<node_internals>/**"
        ],
        "program": "${workspaceFolder}\\index.js"
    },
    {
        "name": "Node Jest Tests",
        "type": "node",
        "request": "launch",
        "runtimeArgs": [
            "--inspect-brk",
            "${workspaceRoot}/node_modules/jest/bin/jest.js",
            "--runInBand"
        ],
        "console": "integratedTerminal",
        "internalConsoleOptions": "neverOpen",
        "port": 9229
    }
  ]
}
```

Mac:

```
"launch": {
"version": "0.2.0",
"configurations": [
    {
        "name": "React App",
        "type": "chrome",
        "request": "launch",
        "url": "http://localhost:3000",
```

```
        "webRoot": "${workspaceFolder}"
    },
    {
        "name": "React Unit Tests (create-react-app)",
        "type": "node",
        "request": "launch",
        "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/react-scripts",
        "args": [
            "test",
            "--runInBand",
            "--no-cache",
            "--env=jsdom"
        ],
        "cwd": "${workspaceRoot}",
        "protocol": "inspector",
        "console": "integratedTerminal",
        "internalConsoleOptions": "neverOpen"
    },
    {
        "name": "Node Attach by Process ID",
        "type": "node",
        "request": "attach",
        "processId": "${command:PickProcess}",
        "skipFiles": [
            "<node_internals>/**"
        ]
    },
    {
        "name": "Node Launch App",
        "type": "node",
        "request": "launch",
        "skipFiles": [
            "<node_internals>/**"
        ],
        "program": "${workspaceFolder}\\index.js"
    },
    {
        "name": "Node Jest Tests",
        "type": "node",
        "request": "launch",
        "runtimeArgs": [
            "--inspect-brk",
            "${workspaceRoot}/node_modules/.bin/jest",
            "--runInBand"
        ],
          "console": "integratedTerminal",
          "internalConsoleOptions": "neverOpen",
          "port": 9229
    }
  ]
}
```

Project Setup

1. Open up this repo on VS Code.
2. From your terminal, run `npm install`.
3. From your terminal, run `npm run dev` to run the app in development mode. Any new changes you make will automatically restart the server.

Exercises - Debugging using console

console.time / console.timeEnd

Goal: Log the time it takes for the API to get a doctor by id. Documentation on console.time()

1. In `index.js`, find the route for getting a doctor by ID (`/api/v1/doctors/:id`).
2. Add the console.time commands before and after a doctor is found.

```
app.get("/api/v1/doctors/:id", (request, response) => {
  console.time("get doctor by id");
  const doctor = data.doctors.find(
    (doctor) => doctor.id === request.params.id
  );
  console.timeEnd("get doctor by id");

  if (!doctor) {
    return response.status(404).json({ error: "Doctor not found." });
  }

  return response.json(doctor);
});
```

3. Go to your browser and run the endpoint `http://localhost:3000/api/v1/doctors/1`.
4. Your VS Code terminal console should show a time below (your time may be different):

```
get doctor by id: 0.558ms
```

This gives you a baseline starting point to start debugging where your application is slowing down.

console.assert

Goal: Use console.assert to log an error message if the assertion is false.

Documentation on console.assert.

1. In `index.js`, find the route for finding a doctor by id.
2. Add an assertion that the doctor exists before returning the error:

```
app.get("/api/v1/doctors/:id", (request, response) => {
  console.time("get doctor by id");
  const doctor = data.doctors.find(
    (doctor) => doctor.id === request.params.id
```

```
  );
  console.timeEnd("get doctor by id");

  console.assert(doctor, `Doctor ${request.params.id} not found`);

  if (!doctor) {
    console.log(error(`Doctor ${request.params.id} not found`));
    return response.status(404).json({ error: "Doctor not found." });
  }

  return response.json(doctor);
});
```

3. Go to your browser and run the endpoint `http://localhost:3000/api/v1/doctors/1000`.
4. Your VS Code terminal console should show a new console line, *before* the chalk red error.

```
Assertion failed: Doctor 1000 not found
```

5. Go to your browser and run the endpoint `http://localhost:3000/api/v1/doctors/1` (a doctor that exists). Your VS Code terminal console should not see any additional console logs.

console.table

Goal: Figure out what's wrong with our POST /doctors endpoint

1. First let's test the endpoint for adding a new doctor using Postman.
2. Open up Postman.
3. Click on File -> Import -> Choose the file that's in this repository called **Debugging JS.postman_collection.json**. This creates a folder called "Debugging JS" and imports all the POST endpoints in your API into Postman for you to test.
    - creating a new doctor
    - creating an appointment
    - creating a patient
4. Select the Create new doctor request on the left sidebar. You may need to expand the Debugging JS folder to see it.
5. Click on the Body tab to see the payload we're sending to the API.
6. Click on Send to test the request. You should see a response `Successfully added a doctor`.
7. Go to your browser console and visit the endpoint `http://localhost:3000/api/v1/doctors/` to see your list of doctors.

   It looks like our doctor was added to the bottom of the list, but it's missing a name! And it looks like the id is also set to a number, not the string like the other doctors.

   Time to debug! :)

Using console.table

Documentation for console.table

1. In `index.js`, find the route for creating a doctor.

```
app.post("/api/v1/doctors", (request, response) => {
  const nextId = data.doctors.length + 1;
  const doctor = { id: nextId, name: request.body.nane };

  data.doctors.push(doctor);

  return response.status(201).send("Successfully added a doctor");
});
```

2. Log the state of `data.doctors` after the new doctor information was added to it. This is a good time to use `console.table()` to see this data in a better, visual way.

   Use the normal `console.log` as well to see the difference between both approaches.

```
app.post("/api/v1/doctors", (request, response) => {
  const nextId = data.doctors.length + 1;
  const doctor = { id: nextId, name: request.body.nane };

  data.doctors.push(doctor);

  console.log(data.doctors);
  console.table(data.doctors);

  return response.status(201).send("Successfully added a doctor");
});
```

3. Go back to your Postman and test the endpoint again.
4. Your VS Code terminal console should show the data in a nice table format!
5. Now we can clearly see the 2 bugs: id is a number and name is undefined.
6. Let's quickly fix the `id` error by adding a `toString` call.

```
const doctor = { id: nextId.toString(), name: request.body.nane };
```

   We can keep going with console logs, but let's test out our IDE Debugger.

Exercises: Debugging using VS Code IDE

Goal: Learn how to use the VS Code IDE Debugger to troubleshoot the issue with the name property being set to `undefined`.

1. Add a `debugger;` line after creating the new doctor object.

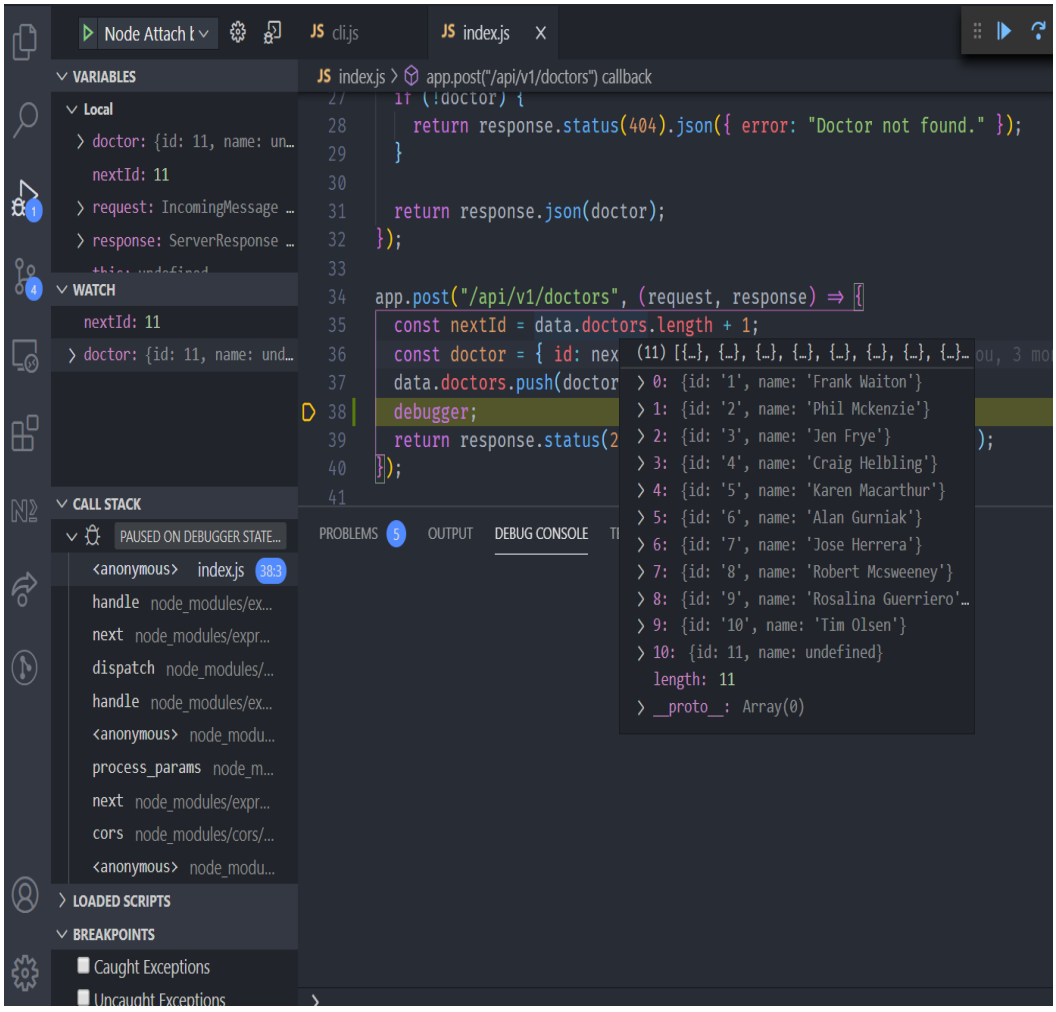
```
app.post("/api/v1/doctors", (request, response) => {
  const nextId = data.doctors.length + 1;
  const doctor = { id: nextId.toString(), name: request.body.nane };

  data.doctors.push(doctor);
```

```
  debugger;

  return response.status(201).send("Successfully added a doctor");
});
```

2. Click on the Debugger icon on the left sidebar menu.
3. Our application is running already so let's start a debugger session with that application node process. Click on the dropdown on the top beside the Run label.
4. Select **Node Attach by Process ID**.
5. You will see a prompt that shows a list of processes. Select the first one (that will usually be the one that's currently running in the folder).
6. Go to Postman and test the endpoint. It will automatically flip you back to VS Code and highlight the `debugger;` line - indicating that the application stopped running at that point.

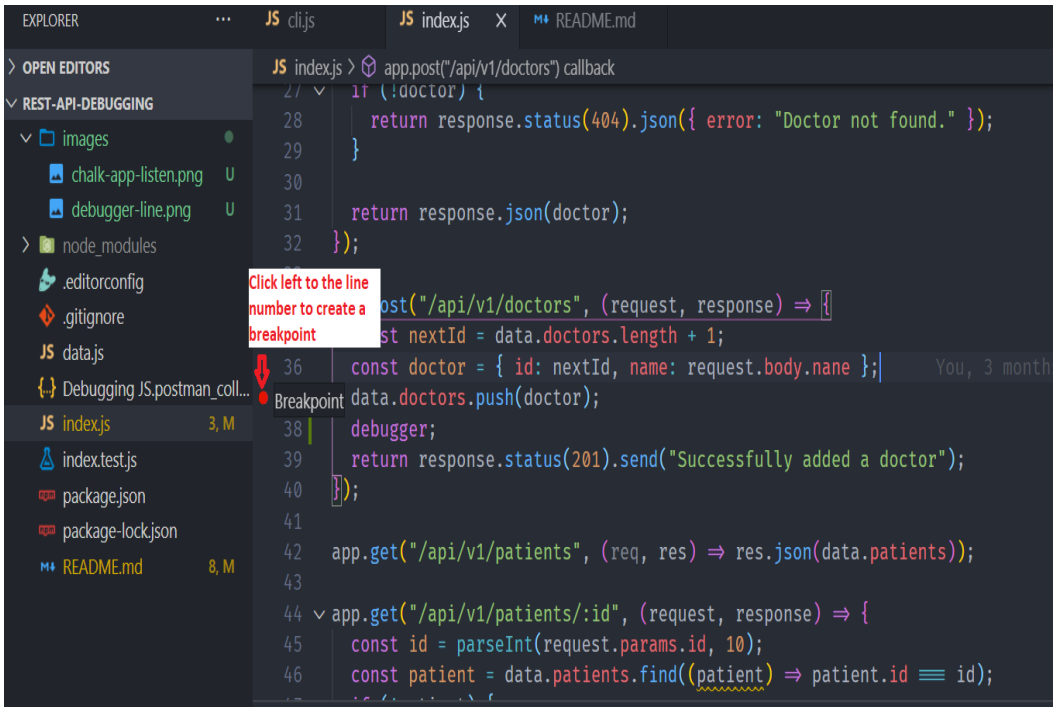


7. On the left sidebar in the Debugger view, we can now see some helpful sections!
    - **Variables** (Local, Closure and Global scopes). Here you'll see the `doctor` object.

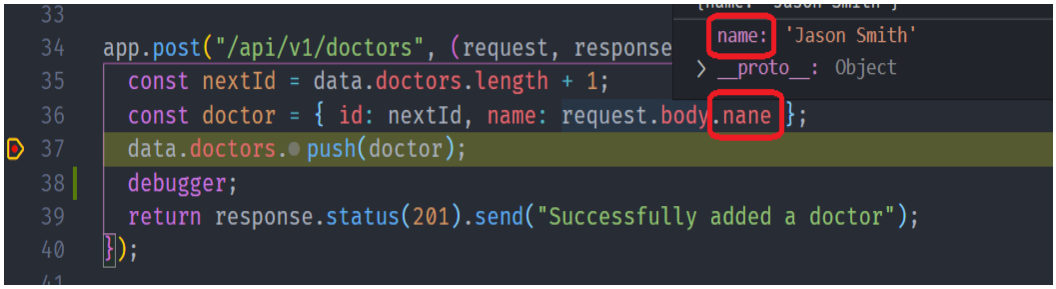
- **Watch** . You can add expressions here to watch for, such as `doctor` to be explicit about what you're investigating.
- Call Stack
- Breakpoints

  Note: Even through debugger line works as a breakpoint, it doesn't show up in the list of breakpoints section

8. Set up breakpoint in VS Code. This should be very similar experience for people who work with IDEs like Eclipse or Visual Studio. You can find the target line then simply click on the left space of the line number to set a breakpoint. When you run the debugger again and do a post call from Postman to that endpoint, it will pause on the line for you to debug, similar to the debugger line in the previous step (except in this way the breakpoint shows in the breakpoints section of the debugger panel). You can check variables, use watch, or hover over to the variables in the code to check the value and go from there.



It is pretty easy from that point to figure out that we have a typo and `request.body.nane` should really be `request.body.name` .



Once fixed, we can verify the endpoint works as expected.

Self-paced - Debugging using the Chrome browser

Goal: Learn how to use the Chrome browser for debugging with breakpoints.

1. We have already set up the npm script to run debugging mode in Chrome. In `package.json`, you should see:

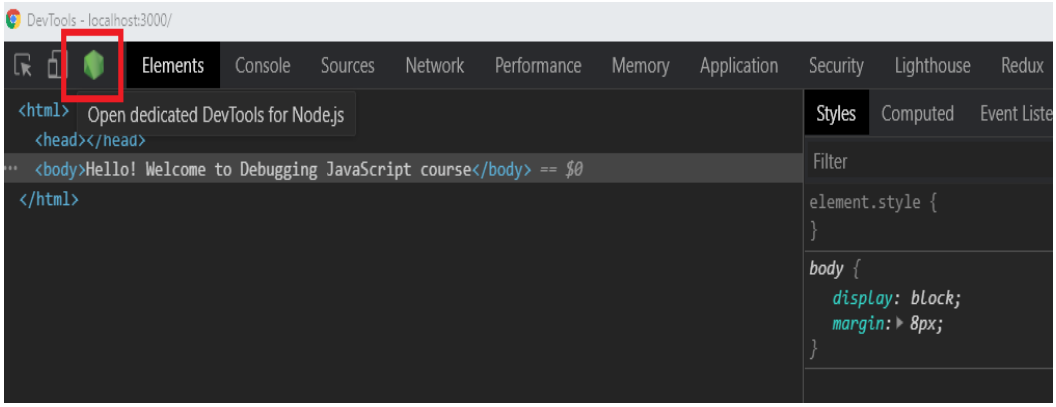
   ```
   "debug": "node --inspect-brk index.js",
   ```

   The `--inspect-brk` flag enables the inspector agent and sets a breakpoint before the user code starts. (Documentation here).

2. In your terminal, run `npm run debug`.

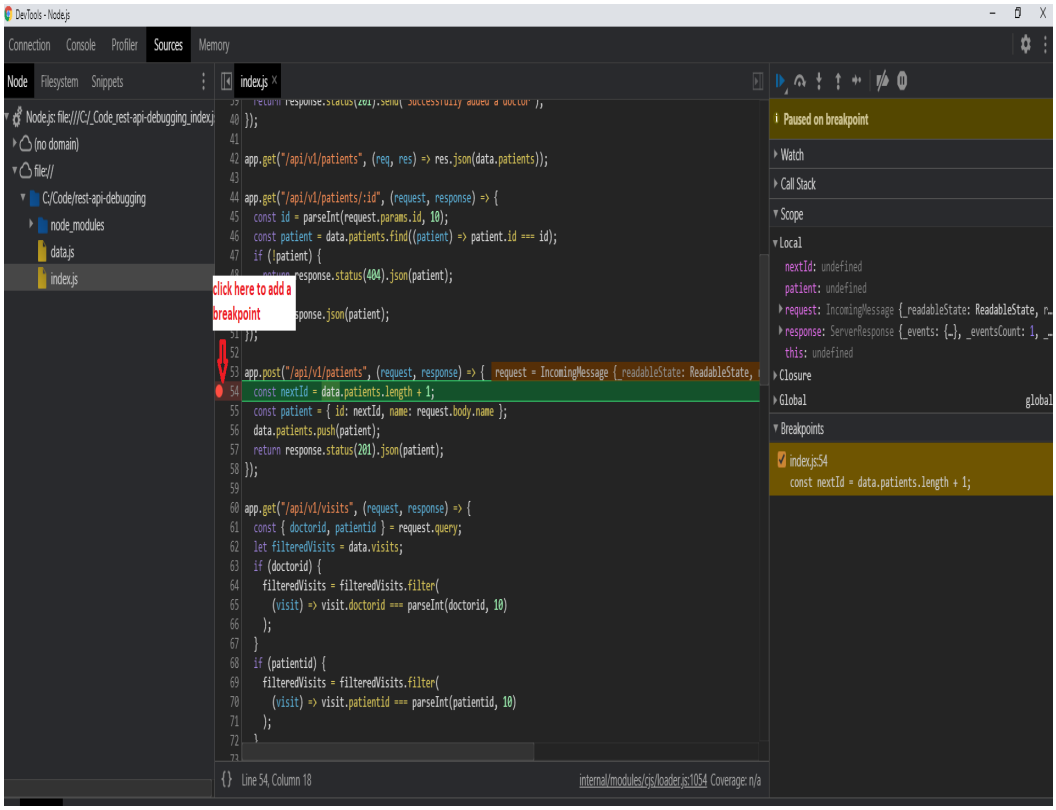
3. You should see a message in your console saying `Debugger listening on ws://___`.

4. Go to your browser and open up the Developer Tools. You should see a new green icon on the toolbar beside 'Elements'. If you hover over it, it says "Open dedicated DevTools for NodeJs". Click on it.



5. A new window should open up with the title "DevTools - Node.js". This works similar to the VS Code Debugger. Click on the Source tab and you will see the source code as well as sections for Watch/Call Stack/Scope/Breakpoints. Select `Node` tab and select the target file.

   To set up a breakpoint, simply click on the space on the left of the line number. Run Postman to create a patient to hit the breakpoint of `/api/v1/patients`.

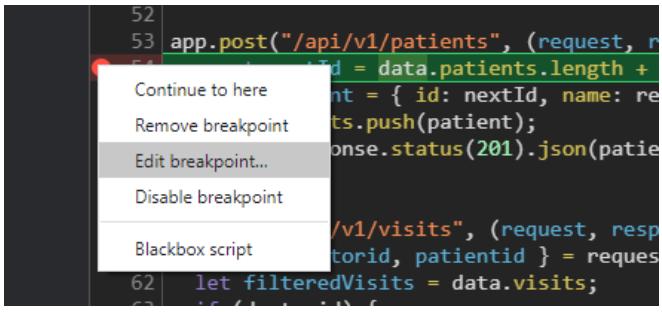


Explore different tabs and menu to see what Chrome DevTools offer.

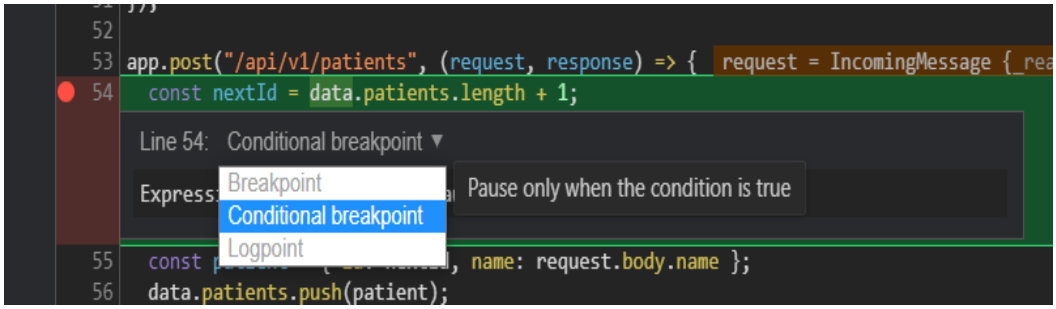
Set up conditional breakpoint

One of the unique features that Chrome DevTools offers would be different types of breakpoints. There are `Conditional breakpoint` and `Logpoint` we can set in addtion to the regular breakpoint. Logpoint will log message to your console instead of pauseing on the breakpoint. Conditional breakpoint will only pause when condition is true.

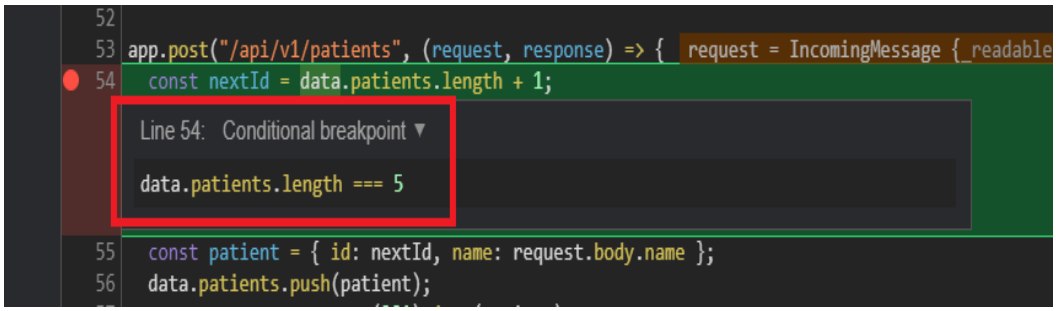
After we click on the source code line to set up a breakpoint, right-click on the breakpoint and select `Edit breakpoint`



Select `Conditional breakpoint` from the drop down



Provide the following expression to only pause when patients list has a length of 5



In Postman, send patient creation request multiple times. Note that code pauses on line 54 only when there are already 5 paitents in the list.