The 3 Stages of Docker Debugging

Course overview

1. **The 3 Stages of Docker Debugging**
    1. Who am I
    2. Our use case
    3. Introducing the problem
    4. Stage 1
    5. Stage 2
    6. Stage 3

The 3 Stages of Docker Debugging

Sections in this chapter:

1. Who am I
2. Our use case
3. Introducing the problem
4. Stage 1
5. Stage 2
6. Stage 3

1-1. Who am I

A knowledge based company

- Courses
- Consulting/Mentoring
- Project Development

Stephen Lau

- Originally from Canada
- Moved to Sweden 4.5 years ago
- Enjoy travelling
- Was commuting to China a couple of years
- Like tackling complex problems

History

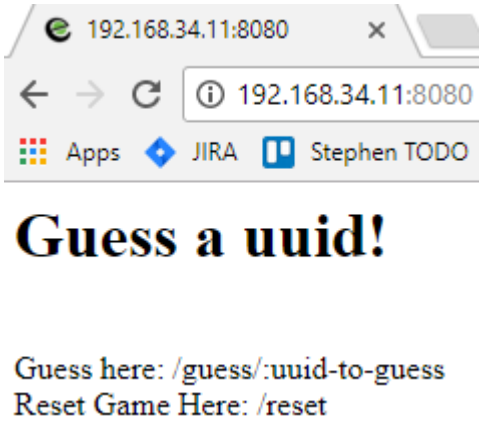
- Started out in firmware engineering
- Worked on the rendering engine for BlackBerry
- Have worked with large companies globally on their cloud computing offerings
- Have worked at all levels of the stack
- Lately even doing some frontend work

1-2. Our use case

Guess a UUID

A simple game*

1-2-1



*** some cheating required...**

Generating the uuid:

1-2-2

```
import uuid from 'uuid/v4';

let uuidToGuess;
function resetUuid() {
  uuidToGuess = uuid();
}

resetUuid();
```

Serving a simple page:

1-2-3

```
app.get('/', (req, res) => {
  res.status(200).send('<h1>Guess a uuid!</h1><br>' +
    'Guess here: /guess/:uuid-to-guess<br>' +
    'Reset Game Here: /reset');
});
```

The routes:

1-2-4

```
app.get('/guess/:uuid', (req, res) => {
  res.status(200).send(
    uuidToGuess === req.params.uuid
    ? '<h1>Correct Guess!</h1>'
    : '<h1>Wrong!</h1>');
});

app.get('/reset', (req, res) => {
  resetUuid();
  res.status(200).send('uuid to guess is now reset!');
});
```

The docker file:


```
FROM node:8

WORKDIR /work
COPY ./ /work/
#RUN npm install
#RUN npm run build
EXPOSE 8080

CMD ["node", "server-build.js"]
```

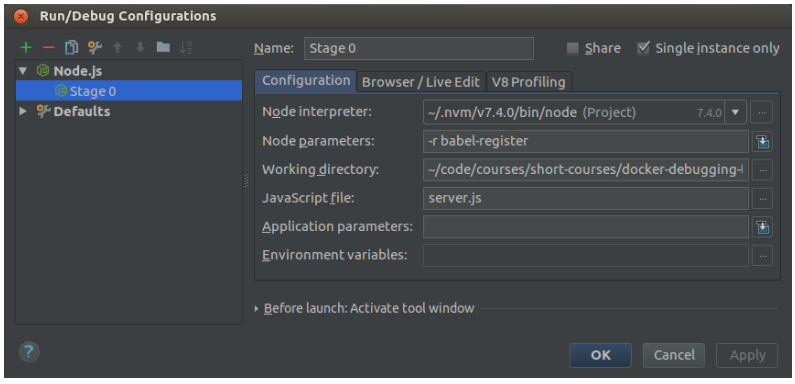
1-3. Introducing the problem

Debugging outside of Docker

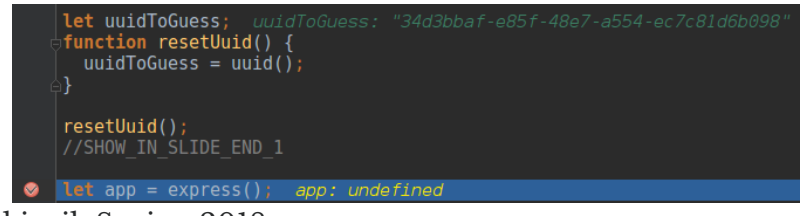
Debug in IDE

- Run app as per normal
- Set breakpoints
- Dockerize app
- Run Docker container
- **Cross fingers!**

Configure debugger:



Set breakpoint:



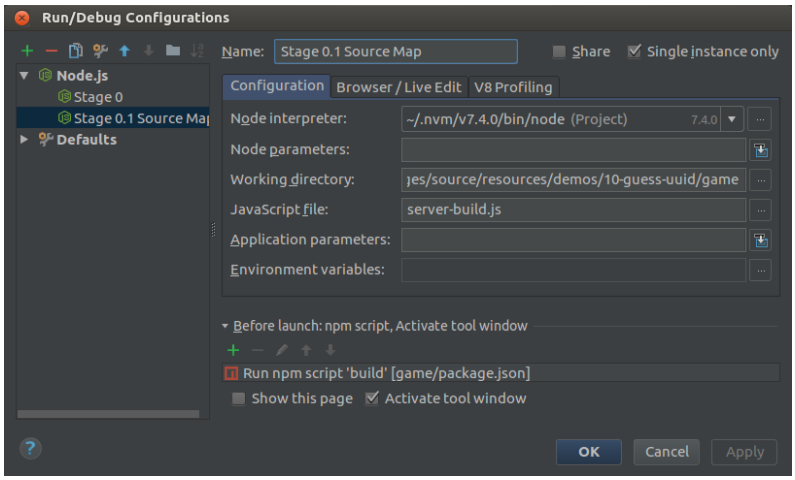
Setup source maps:

1-3-4

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "serve": "babel-node server.js",
  "build": "babel server.js -o server-build.js --source-maps inline"
},
```

Debugger configured with source maps

1-3-5



Demo

1-3-6

Trying the game

Building the Docker image

1-3-7

```
$ docker build . -t game:s0
```

Running the Docker image

1-3-8

```
$ docker run -p8080:8080 game:s0
```

Running Detached

1-3-9

```
$ docker run -d -p8080:8080 game:s0
```

Demo

1-3-10

Our isolated game

Currently we have no way to debug our game while running in docker.

1-3-11

1-4. Stage 1

Docker logging

Modifying our source

1-4-1

```
let uuidToGuess;
function resetUuid() {
  uuidToGuess = uuid();
  console.log(`uuid to guess: ${uuidToGuess}`);
}
```

Building our image

1-4-2

```
$ docker build . -t game:s1
Sending build context to Docker daemon  12.85MB
Step 1/7 : FROM node:8

#... truncated

Removing intermediate container 0c8b21542167
Successfully built 455d9c957892
Successfully tagged game:s1
```

Running our container

1-4-3

```
$ docker run -d -p8080:8080 game:s1
11e228cfbf32c68a20735bb4d719b956591b6e7bb77b78b212dff33f619191de
```

Getting our logs

1-4-4

```
$ docker logs 11e228c
uuid to guess: 55c8f651-0181-40f7-b6a8-07dea150c86f
Server running on port 8080
```

Demo

1-4-5

Winning with logs

Pros:

1-4-6

- Fine for simple examples
- Simple to work with
- Can go back and audit logs

Cons:

1-4-7

- Not really scalable
- We may put logging in the wrong place
- Can lose context (if adding logs)

Disclaimer

1-4-8

Logging is a powerful tool, when used correctly.

1-5. Stage 2

Open a port

Node.js Remote Debugger

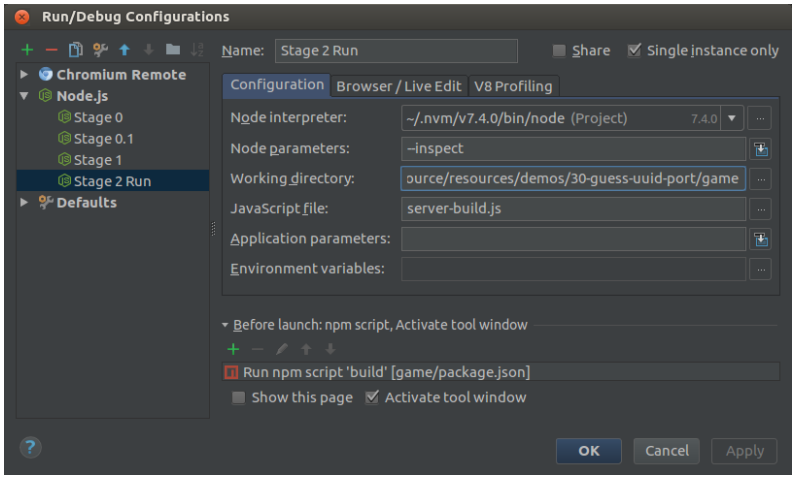
1-5-1

- Allows debugging of remote processes
- Connects via v8-inspector protocol*

*** Versions 7.7 and older use debugger protocol**

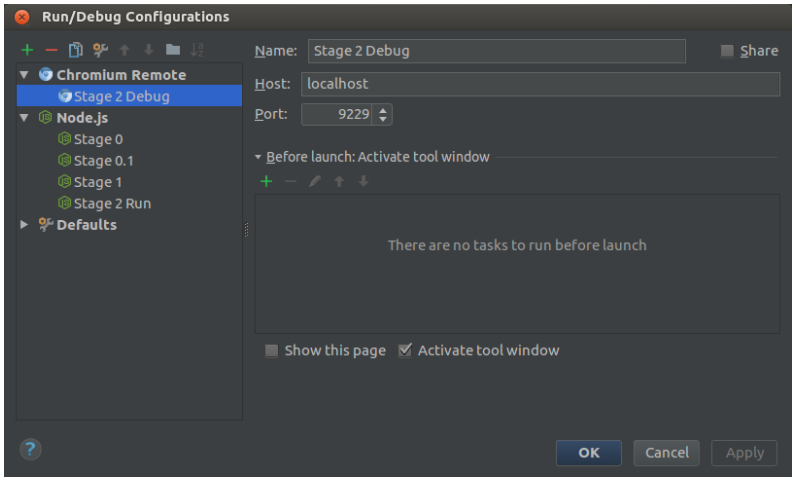
Configuring to run inspection mode

1-5-2



Configuring chromium debugger

1-5-3



Demo

1-5-4

"Remote" debugging locally

Wrapping in docker

1-5-5

```
FROM node:8

WORKDIR /work
COPY ./ /work/
#RUN npm install
#RUN npm run build
EXPOSE 8080
EXPOSE 9229

CMD ["node", "--inspect=0.0.0.0:9229", "server-build.js"]
```

Running our container

1-5-6

```
$ docker run -d -p8080:8080 -p9229:9229 game:s2
11e228cfbf32c68a20735bb4d719b956591b6e7bb77b78b212dff33f619191de
```

Problems

1-5-7

- Port is always open
- Container is in debug mode
- Run command is getting long

Solution

1-5-8

Docker Compose

docker-compose.yml

1-5-9

```
version: '2'
services:
    deployment:
        build: ./
        image: game-deployment
        ports:
            - "8080:8080"

    debug:
        build:
            context: ./
            dockerfile: Dockerfile.debug
        image: game-debug
        ports:
            - "8080:8080"
            - "9229:9229"
```

Using docker compose

1-5-10

```
$ docker-compose build debug
$ docker-compose up -d debug
$ docker-compose down
```

Demo

1-5-11

Running with docker-compose

Pros

1-5-12

- Can connect a debugger into Docker container

Cons

1-5-13

- Security - debug port left open
- Lose context (if not using debug container)
- Have to maintain two different docker files

1-6. Stage 3

Docker networking

Changing node to debug mode

1-6-1

```
$ node server-build.js
Server running on port 8080
```

```
$ ps -C node
  PID TTY          TIME CMD
 7433 pts/15   00:00:00 node
$ kill -s USR1 7433
```

Result:

```
$ node server-build.js
  Server running on port 8080
  Starting debugger agent.
  Debugger listening on 127.0.0.1:5858
```

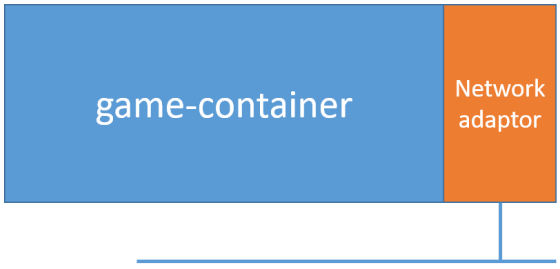
Sending a signal into Docker

1-6-2

```
$ docker run -d game:s0
  c6985986393a2cee97bbb53de8a172176d4662edf9bd2472be61eaf656b2a2f7
$ docker logs c6985
  Server running on port 8080
$ docker kill -s SIGUSR1 c6985
  c6985
$ docker logs c6985
  Server running on port 8080
  Debugger listening on ws://127.0.0.1:9229/619d5499-4c9d...
  For help see https://nodejs.org/en/docs/inspector
```

The situation

1-6-3



Problems

1-6-4

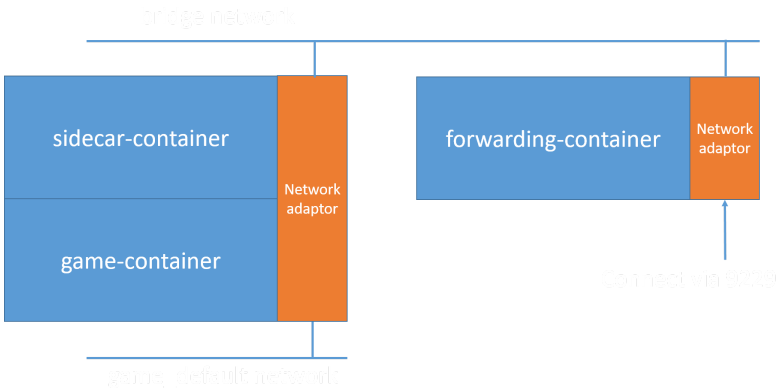
- Container **may** be in an isolated network
- No ports are open
- Debugger is listening on 127.0.0.1

Our Goals

1-6-5

- Connect to the container (even if isolated)
- Don't leave any ports open
- Pretend to be localhost

The plan



1. Send SIGUSR1 to switch to debug mode.
2. Connect a "sidecar" container to the same network interface.
3. Connect a forwarding container to the same network as "sidecar".
4. Connect to the forwarding container via our debugger.

Socat

Socat is a command line based utility that establishes two bidirectional byte streams and transfers data between them.

Connecting the sidecar


```
docker run -d --name socat-attach --network=container:${TARGET_CONTAINER} \
  socat-image socat TCP-LISTEN:${TEMP_PORT},fork TCP:127.0.0.1:9229
```

- `${TARGET_CONTAINER}` is the container ID
- `${TEMP_PORT}` is a port to connect sidecar to forwarding container

Connecting the forwarding container


```
docker run -d -p 9229:9229 --name socat-fw socat-image socat \
  TCP-LISTEN:9229,fork TCP:${TARGET_IP}:${TEMP_PORT}
```

- `${TARGET_IP}` is the targets IP
- `${TEMP_PORT}` is a port to connect sidecar to forwarding container

Finding the IP address

1-6-11

```
$ docker inspect ${TARGET_CONTAINER}
#...
"Networks": {
                "bridge": {
                    #...
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:02",
                    "DriverOpts": null
                }
            }
```

Demo

1-6-12

Putting it all together!

Pros:

1-6-13

- No Ports left open
- No context lost
- One docker file to maintain

Cons:

1-6-14

- Can't audit logs
- App is left in debug mode

Recommendations:

1-6-15

- Use logging to create audit logs
- Debug using Docker networking
- Restart/Kill containers after debugging