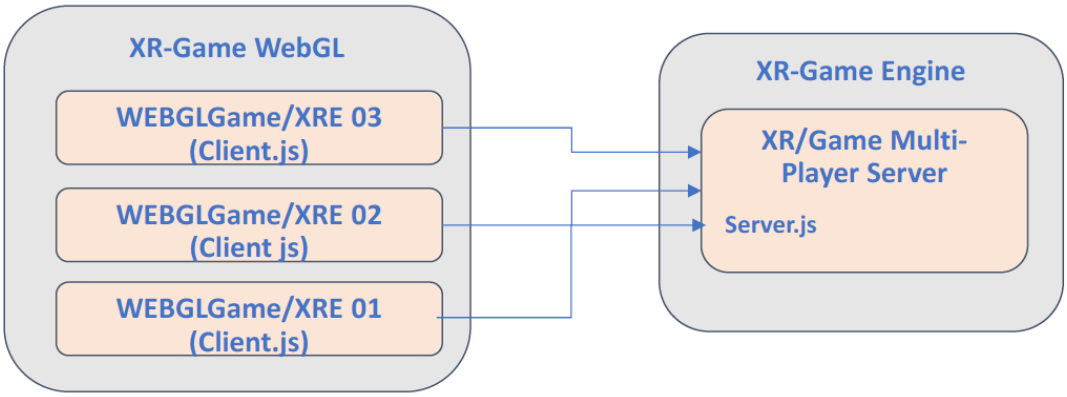
Multi-Player Server WEBGL/WEBXR R1.0 – Documentation

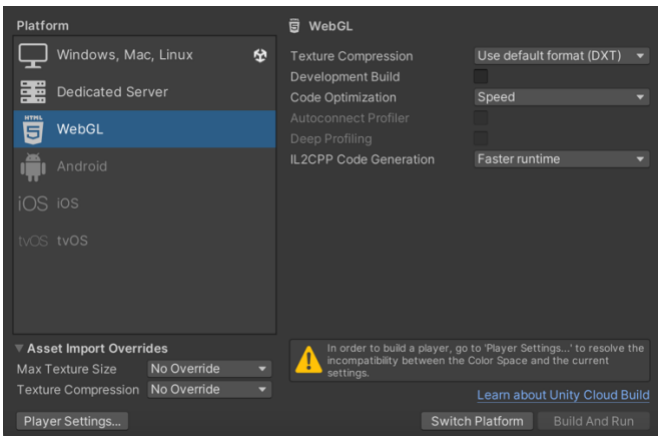
Daniel Godfrey

Intro

The purpose of this build was to convert a monolith Unity multi-player game build to a microservice architecture, namely separating the client and server elements into their separate nodes, then virtualizing them (serverless function, cloud function, whatever term you may like to use) and having them communicate successfully living on different ports.



Unity Build



Compiling a WebGL Unity build will output a vanilla JavaScript build to run out of the box, using an Express server and Socket.IO to communicate between the client.js file and server.js file.

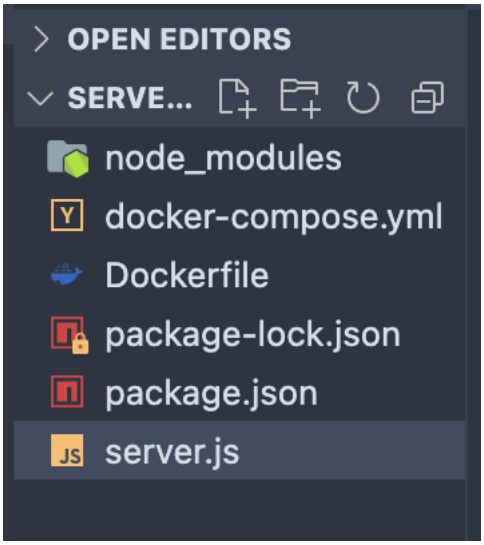
A few things to note:

- HTML skeleton and Server.js file will always be the same
- Client.js file will vary from game to game
- Game will run fine (if the build works correctly) on a single port on a local machine

The first two points highlight why the architecture can and will be retooled to separating the client and server according to the new design. This will allow for multiple games to be developed independently of a consistent server.

Server Separation

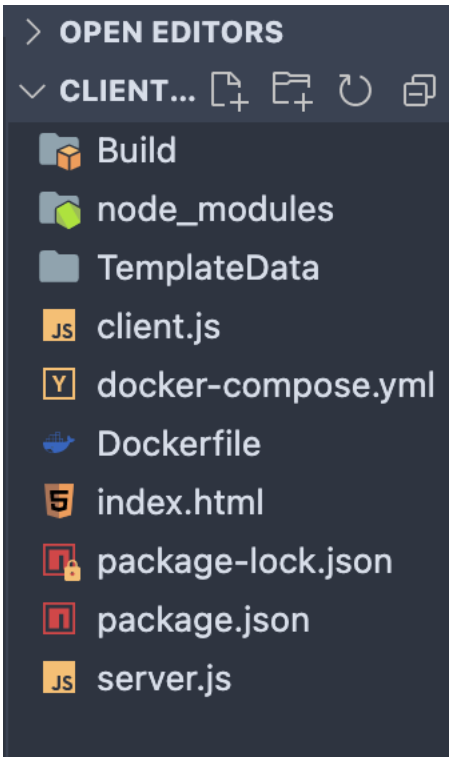
The files can be separated according to the original design into separate nodes quite easily. The files client.js and related tools (build [unity files], template [css, graphics, etc.], and anything that is not the server.js file can be moved to a separate node for the time being.



…the Docker additions will be addressed later. This is the engine for virtualization of both the server and client.

Client Separation

The client is slightly trickier but not by much. Running on a separate node from our established server, we will put together a new, very simple, express server of its own to point to a new location where the client.js file and related tools will run in a browser. Some code will be cut from the original server.js file because it is no longer needed there, but on the new client node side:



…a package.json file is still necessary and can be copied straight from the original server side (minor tweaks like the name, etc.). Now we have the index.html, client.js, and Build + TemplateData functioning on its own server as can be seen here with a very simple Express build (this is a new file):

This is simply used to point to a port where the client can run in a browser, along with its related tools. The server game logic exists on the separate build we left behind on the original node.

Running both independently of each other, all that is left to do is retool the pointers for communication between ports.

Communication Logic

Ports for both the ServerNode and ClientNode that can now run independently from one another need to be redefined in a total of four places. First is the ServerNode:

```
http.listen(3010, '127.0.0.1', function(){
  console.log('listening on *:3010');
});

console.log('------- NodeJS server is running -------');
```

The IP address is our localhost, but in the future can be changed to wherever our ServerNode may live (AWS, etc.). For now it is a local IP, with a new port defined as 3010 in this case.

…the ClientNode requires three changes. First is a new port (3020 in this case) that can be seen above in our new "server" file for the client.

Next is changing the socket.io definition in the client.js file as shown here:

```
client.js > window.addEventListener('load') callback
1   var socket = io('127.0.0.1:3010') || {};
2   socket.isReady = false;
3
4   window.addEventListener('load', function() {
```

The original code is simply 'io()'in the highlighted area. This would direct communication to the game server running on the same port. Now we need to point it to the new area, with IP and port defined as above. Now it will point at our separated ServerNode and communication will be successful. However, one more change is necessary in the HTML skeleton:

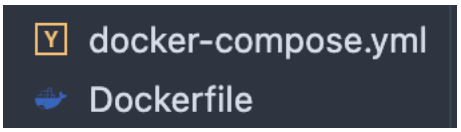
```
<!-- must modify this part of the HTML skeleton to accomodate target separate server port-->
<script src="http://127.0.0.1:3010/socket.io/socket.io.js"></script>
<!-- Daniel G.-->

<script src="client.js"></script>
```

The highlighted portion is the needed addition to the original code.

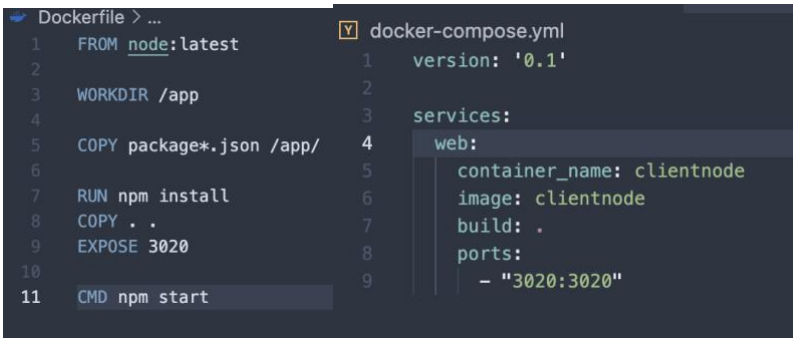
Virtualization

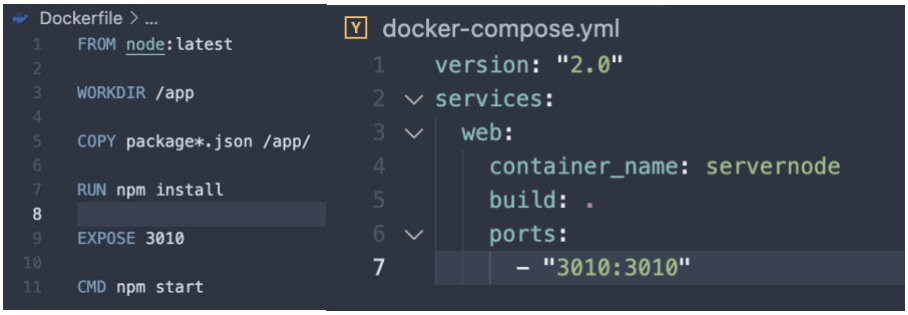
Now we are ready to "Docker-ize" both the ClientNode and ServerNode. Both should work as is running on a local machine already…but the final step is to make them both "serverless functions" that exist in a container which can be placed anywhere on any cloud, substituting our localhost IP for any new locations.

I chose Docker because it is familiar and simple to use with Node.js. Two inclusions are necessary on both the ServerNode and Client nodes: a Dockerfile and docker-compose.yml file:



…the contents of these files will allow Docker to created virtualized containers with open ports to access, with our socket.io communication method redefined and successful as above.





The contents are nearly identical, with some slight retooling necessary for the ClientNode. Without going to deep into how Docker works, these simple files will allow for a command line "docker compose up -d" command to "spin up" a container running the full contents of the newly established ClientNode and ServerNode.

Summary

In completing this task and proving communication between the ClientNode and ServerNode can work with this build, I chose to keep the layers of complexity very thin. There are tools that can establish a separated node and dockerfile build with their own command line tools and interfaces (i.e. OpenFAAS), but I chose to do this manually to eliminate a layer and keep things as simple as possible, given that this is a 1.0 build.

This may change in the future. There are other systems (Kubernetes, Minikube, and I'm sure others) that can produce the same results. Maybe there is value in comparing them, but for now proving a virtualized microservice architecture can work with the WebGL Unity builds will suffice.

For supplementary video documentation, refer to this link:
https://drive.google.com/file/d/1VMOczdP2avGMLOTwpGkvtvNupNsRl5_E/view?usp=sharing