Interfaces, Generics and IO

Table Of Contents

Processes and Servers

We are able to build real-time networking applications. Previously you have built applications that have interacted with an express application.

However, we will be exploring building applications and servers that can support real-time communications. This means that the application establishes a connection to the server and both can communicate without needing to re-establish a connection.

Processes

We have gone over what a program and process is, however, interprocess communication is an important concept for build complex application suites.

What kind of servers have you we interact with?

What are the different ways to connect to a process?

Processes

The main focus is to understand that interacting with servers are processes and we can interact with them through:

- IO/STDIO
- Files
- Pipes and Unix Sockets
- Sockets (TCP/UDP)
- Shared Memory

  ...

Anything that will allow processes to communicate, can make a process a kind of server.

Standard Input/Output

With C# you have used `Console.ReadLine` and `Console.WriteLine` to interact with input and output from a program. In JS/TS we need to use event system and retrieve data from buffers attached to the process.

We need to use the `async` and `Promise` objects in this situation.

Standard Input/Output

We have seen and heard the term Input and Output and to apply **Standard** before them is to imply that there is one way that all processes interact with the IO system.

Whenever we launch a program and it turns into a process, each process has access to 3 buffers.

- Input
- Output
- Error

Whenever you call `ReadLine` in C#, it is your process going to sleep until the IO system wakes it up to process the data present there.

How do we do this in node?

Standard Input/Output

A typical pattern we can use is using the `await` keyword within a for loop along with an `of` operator that will use the `SymbolIterator` type associated with the object.

```
import readline from 'node:readline';
import { stdin } from 'node:process';

async function RunProgram() {
    const rl = readline.createInterface({input: stdin });
    for await(const line of rl) {
        console.log(line + '!');
    }
    return;
}
RunProgram();
```

Input/Output - Files

Files are a familiar IO construct and one we use to create these very slides and also write code in. Programs will interact with files by using them to save data or read them.

A file is a logical map to a range (or many ranges) of memory. We can have files mapped to the hard disk, mapped to ram or mapped over a network.

What are some good examples of files which aren't on a hard drive?

How do we read a file with javascript?

Input/Output - Files

Remember the code from before? Lets now adapt it.

```
import readline from 'node:readline';
import fs from 'node:fs';

async function RunProgram() {
    const fstream = fs.createReadStream('some_file.txt');
    const rl = readline.createInterface({input: fstream });

    for await(const line of rl) {
        console.log(line + '!');
    }
    return;
}
RunProgram();
```

Input/Output - Pipes and Unix Sockets

We also have other files, more special in this case.

Within Unix and Unix-Derivative operating systems exist pipes. These are memory mapped files and act similarly as standard input and output for a program but that we can read and write to them from anywhere.

Unix Sockets operate similarly but provide the ability to have two-way communication.

Do you know any applications that somehow know if another application has done something?

- What about steam?
- What about Task manager?
- What about Web browsers with multiple tabs and windows?

REST and Websockets and Real-Time Communications

We will be visiting concept that is similar to pipes but is used in real-time communications for internet enabled applications.

However, we will first revise over REST apis and how application development typically requires us to communicate wtih endpoints.

REST

Within the javascript development environment we are able to easily develop backend REST apis using **express**.

When developing a REST api (typically just called a Web API), we usually have endpoints that our mobile applications send messages to.

- Browser will send `GET /user/1` which when our server receives the message it routes to a function associated with `get('/user/:id', ...);`

So how do websockets differ?

REST apis typically treat every action/method as its own connection where the websockets establish a single connection for a client and all messages are sent and received with that connection.

This resembles TCP connections that want to maintain state and a persistant connection.

Websockets

This is where we venture into developing an application that will be a server and a client using websockets.

Websockets - Server

Lets create a server. We will be using a module called `ws` that will allow us to construct a server websocket (You can't create these in the browser).

```
import { WebSocketServer } from 'ws';
const wss = new WebSocketServer({ port: 8080 });

wss.on('connection', function connection(ws: WebSocket) {
  ws.on('error', console.error);

  ws.on('message', function message(data: string) {
    console.log('received: %s', data);
  });

  ws.send('Hello!');
});
```

The following is annotated:

```
import { WebSocketServer } from 'ws';

//Creates a server socket on port 8080
const wss = new WebSocketServer({ port: 8080 });

//The event when a connection is created
wss.on('connection', function connection(ws: WebSocket) {

  //If an error occurs, it will send it to console.erro
  ws.on('error', console.error);

  //When a message is received by the server, it
  // will run this function
  ws.on('message', function message(data: string) {
    console.log(`Received: ${data});
  });

  ws.send('Hello!');
});
```

Websockets - Client

Lets create a client.

```
const wsclient = new WebSocket('ws://localhost:8080');

wsclient.addEventListener('open', () => {
    console.log("Connection Open");
    wsclient.send("Hello Server!");
});

wsclient.addEventListener('message', (msgdata: any) => {
    console.log(`Got: ${msgdata.data}`);
});

wsclient.addEventListener('close', () => {
    console.log("Connection Closed");
});
```

Debrief of Websockets

Similar to what we have developed before with express where we have a client and a server. This particular case is more reflective of a realtime system. The actions are immediate and also in order.

With HTTP requests being formed for any requests we make from the client, the order of the operations is not guaranteed to be in the order that it was sent.

Protocols

Great! We can do some form of communication and maintain a connection, however we have to establish a protocol of communication between the server and client.

With a protocol, we have to consider the states that it could be in and how we package the data and deliver it. A common container type is to use JSON (similar with the WebAPI) to package data and send it.

However, we want to outline the expected forms so it is easy to determine.

How could we design a protocol for an instant messaging system?

Consider:

- How we could label sending and receiving of messages
- Data to outline how much data there is
- Agreeing to both serialisation formats and expected fields
- Definitely consider state information and errors as these are typically overlooked.

Interfaces

You have been introduced to interfaces prior but as a refresher and extension, we need to look into interfaces within Typescript.

```
interface Operation {
    apply(): void

    opCode: string
}
```

Why would we use interfaces?

Interfaces in typescript can have fields/require classes have methods and fields specified in the interface.

The compiler will verify if the type adheres to the interface and implements what is required.

Without this information, we have to rely on the run-time component to check for us, throw an exception or accept arbitrarily executed code.

Why would we use interfaces?

- Over inheritance on classes? We typically want to implement and generalise behaviour rather than things. Animals can all make noise, move and eat but they all do it differently and have different properties each.
- If we adhere to only having methods or properties, it is easier to reason with and we do not run into common problems with inheritance.
- It works elegantly with composition and generics.

Generics

While, JS is a dynamically typed language and we can lean into it, we want to be able to build types that have generic qualities.

It would be quite constraining if the only things we could do is have a `number[]` or `string[]`.

Hang on, do they need to duplicate the array type for different types?

Generics

No, but we have a feature in typescript (and you've seen it in C#) called Generics. Generics allow you to have a **type parameter** as part of your construct.

How do we use generics. Lets start with a simple case.

I want to create a `class` called `Thing` that will contain `T`, where `T` can be some `type`.

Generics

```
class Thing<T> {
    // snipped the rest
}
```

The above class specifies a type parameter `T`, which can be used within the definition of the class `Thing`.

Let's expand on this.

```
class Thing<T> {

    thing: T;

    getThing(): T {
        return this.thing;
    }

    setThing(obj: T) {
        this.thing = obj;
    }
}
```

Iterators

The thing to note here is that all these components relate to an interesting pattern called an `iterator`.

An iterator is object that allows reading through a collection. It maintains state within the collection and where to go next. A collection itself could be something where it generates or retrieves input on demand.

Iterators

Generally, iterators have their place in any domain where a series or sequence can be an expression.

This can include:

- Database Cursors
- Line Retrieval from a file
- Data Streams
- Data Structure Traversals (Trees, Graphs, Grids...)
- ... everywhere

Iterators

What is the advantage here?

We get to maintain state during the iteration, linking back to our time-complexity of our `get(...)` call on a linkedlist, this can be **O(n)**. This will result in the following code being O(n^2).

```
let list = new LinkedList();
// adds

for(let i = 0; i < list.size(); i++) {
    const e = list.get(i);
    // other things
}
```

Maintaining the state of the cursor means we aren't always restarting from the beginning and to find the **next** element.

Making our own iterator in Typescript

The current iterators that have been outlined isn't the same as what Javascript expects but it at least breaks down the different components of it.

If we want our iterators, we need to leverage `Symbol.iterator` and implement a method that corresponds with this builtin.

Lets go!

```
class RandomIntegerIter {
  toGen = 0;
  constructor(toGen) { this.toGen = toGen; }

  [Symbol.iterator]() {
    const endpoint = this.toGen;
    let step = 0;
    return {
      next: () => {
        let res = {
            value: Math.floor(Math.random()* 100),
            done: step >= endpoint
        };
        step += 1;
        return res;
      }
    }
  }
}
```

What about typescript?

Well, we need to implement Iterable.

```
class RandomIntegerIter implements Iterable<number> {
  toGen: number = 0;
  constructor(toGen: number) { this.toGen = toGen; }

  [Symbol.iterator]() {
    const endpoint = this.toGen;
    let step = 0;
    return {
      next: () => {
        let res = { value: Math.floor(Math.random()* 100), done: step >= endpoint };
        step += 1;
        return res;
      }
    }
  }
}
```

Now, we can simply start using the `of` keyword where appropriate.

```
let ngen = new RandomIntegerIter(5);

for(let n of ngen) {
  console.log(n);
}
```

Resources

1. Readline, Nodejs, https://nodejs.org/docs/latest/api/readline.html
2. Process, Nodejs, https://nodejs.org/docs/latest/api/process.html
3. Filesystem, Nodejs, https://nodejs.org/docs/latest/api/fs.html
4. Net/IPC, Nodejs, https://nodejs.org/docs/latest/api/net.html

Resources

5. ws, Luigi Pinca, https://github.com/websockets/ws
6. Websocket, MDN, https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
7. Writing WebSocket client applications, MDN, https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_client_applications

Resources

8. Iterators, Typescript, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_generators

9. Iteration protocols, MDN, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols

Enjoy coding!