Introduction to the Middleware Pattern

In Typescript

Sean Cooper

July 27, 2020

1 Introduction

When starting out with any new pattern the first thing you ought to try and do is to design the interfaces, all this really means is defining what's responsible for storing and manipulating each bit of data and the way in which parts of the system can communicate with one another.

An interface is a statement of the contract that a thing that implements it will adhere too. Meaning, if anything instance is known to be of any given type you can be sure it will contain the data defined in the interface and it will have methods for accessing and updating that data.

To give you an idea about how a pattern works you'll often be provided with a short synopsis along with some UML diagrams, before we get started writing a middleware let's briefly have a look over a couple of different types of these diagrams.

2 Diagrams

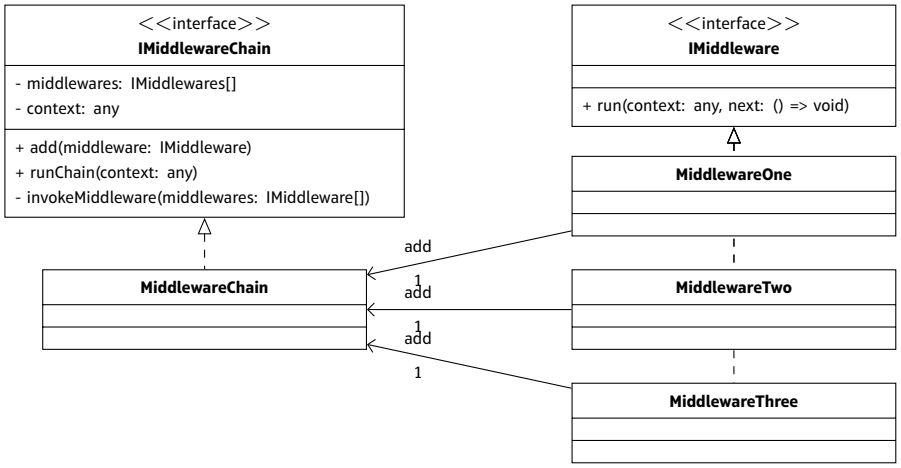
2.1 UML

You'll often see design patterns described with UML diagrams like the one in *Fig. 1*. There are conventions and meanings to the structure and symbols, it's worth checking some guides to get a little more familiar with them. If you ever try to make any don't get too hung up on whether or not you're doing it right, they're meant to help you describe systems, not get in the way. As long you understand what you've created and can use it to explain it to someone else, it's served its purpose.

In the *Fig. 1* diagram you'll see there are two interfaces defined and four classes. The interfaces describe the make up of the classes that implement them. You can see *IMiddleware* interface will have a single method called *run* which has two parameters. It's prefixed with a "+", this means the method is publicly accessible. In the *IMiddlewareChain* there are a mixture of attributes and methods. The two attributes are both prefixed with "-", as you may have guessed, this means they're private, so they won't be accessible outside of the class. The *invokeMiddleware* method is also private, so can't be called outside of the class.

To show how each of the entities relate to one another there are a whole host of different line, symbol and arrow combinations in UML diagrams. In this diagram the dashed arrows point to the interface each class implements. While the solid lines show that *MiddlewareChain* will receive the three instances of the *IMid-*

Figure 1: Middleware UML



dleware via the add method.

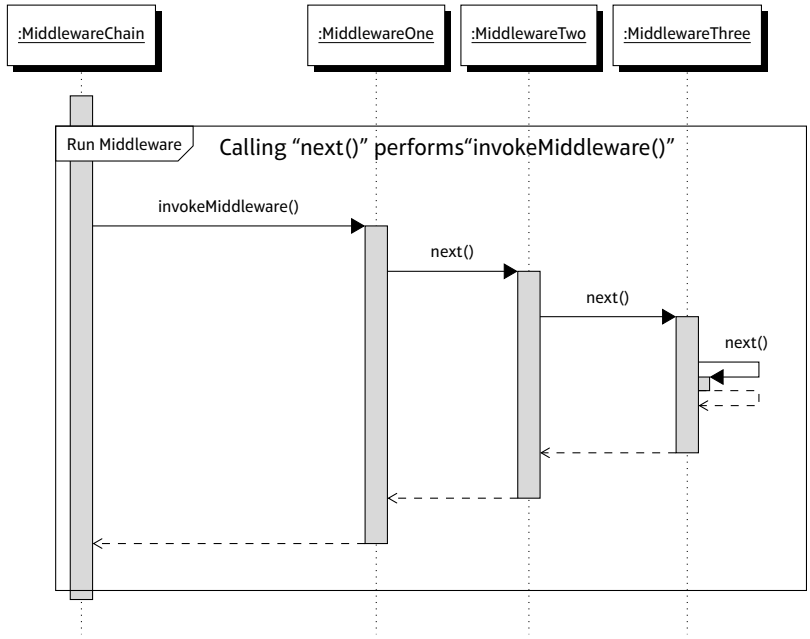
This is by no means a conventional UML diagram, it's my interpretation of the rules and standards, but it's good enough to demonstrate the system, which is all that really matters when it comes down to it.

2.2 Sequence Diagram

A sequence diagram *Fig. 2* can be used to show the flow of an applications threads or the communication between clients and servers. It's useful for describing both synchronous and asynchronous event streams.

To begin, you'll call the 'runChain()" method of an instance of the "MiddlewareChain". "runChain" then calls the private method "invokeMiddleware", this is passed an array of "Middleware" instances and will be responsible for calling the middleware's "run" methods. "invokeMiddleware" also creates a function that will call another "invokeMiddleware" function which is passed to the middleware "run" method as the "next" parameter. The "run" method can then call "next()" to create a recursive chain of method invocations.

Figure 2: Middleware Sequence Diagram



3 Implementation

Now we can take the information from the diagrams to define the system's interfaces. Once the interfaces have been defined we'll walk through each of the methods before finally writing a simple demo. Though if you'd rather see the full implementation feel free to skip ahead to the final script *Code Ref. 11*.

3.1 Interfaces

First we'll define the "IMiddlewareChain" interface *Code Ref. 1*. When defining an interface in code you can omit the private methods and attributes, only the public facing api is of concern. It's a merely a contract telling other parts of the system what they can interact with on this class type, nothing else needs to know what's going on internally.

The only public methods are "add" and "runChain" so they are defined below. "add" will accept an instance of "IMiddleware" as an argument. "runChain" accepts anything as the "context", this will be the data that you want to be manipulated throughout the middleware chain.

Code Ref. 1: IMiddlewareChain

```
interface IMiddlewareChain {
    add(middleware: IMiddleware): void
    runChain(context: any): any
}
```

Secondly lets define the "IMiddleware" interface. This only has one public method "run" which accepts the current context data and a "next" callback function.

Code Ref. 2: IMiddleware

```
interface IMiddleware {
    run(context: any, next: () => void): void
}
```

3.2 Classes

Next up we need to create classes that implement the interfaces. The "MiddlewareChain" class implements *Code Ref. 3* by providing the two public methods, "add" and "runChain" defined in the interface. The functionality for this method stubs will be added in the next section.

Code Ref. 3: MiddlewareChain Class

```
class MiddlewareChain implements IMiddlewareChain {
    private middlewares: IMiddleware[] = [];
    private context: any = {};

    add(middleware: IMiddleware) {
       // Todo: implement method
    }

    runChain(context?: any) {
       // Todo: implement method
    }

    private invokeMiddleware(middlewares: IMiddleware[]) {
        // Todo: implement method
    }
 }
```

Any number of "IMiddleware" classes *Code Ref. 4* can be created to perform any task you can dream up. We'll declare two for now, with the imaginative names of "MiddlewareOne" and "MiddlewareTwo". Again the method stubs are currently empty ready to be filled in later on.

Code Ref. 4: Middleware Classes

```
class MiddlewareOne implements IMiddleware {
    run(context: any, next: () => void) {
          // ... Todo: Implement method
    }
}

class MiddlewareTwo implements IMiddleware {
    run(context: any, next: () => void) {
          // ... Todo: Implement method
    }
}
```

3.3 Methods

Now comes the crucial part, writing the actual functionality, first the "MiddlewareChain" methods that are responsible for handling the middleware system, then we'll write some "run" middleware methods that will demonstrate the principles.

3.3.1 Add Method

This method is responsible for adding "Middleware" instances to the private "middlewares" array attribute.

Code Ref. 5: add Method

```
class MiddlewareChain implements IMiddlewareChain {
    private middlewares: IMiddleware[] = [];

    // ...

    add(middleware: IMiddleware) {
        this.middlewares.push(middleware);
```

```
    }

    // ...
}
```

3.3.2 Run Chain Method

This method is responsible for initialising the "context" data attribute, invoking the middleware chain and returning the resulting "context"

Code Ref. 6: runChain Method

```
class MiddlewareChain implements IMiddlewareChain {
    private middlewares: IMiddleware[] = [];
    private context: any = {};

    // ...

    runChain(context?: any) {
        if (context) this.context = context;
        this.invokeMiddleware([...this.middlewares])
        return this.context;
    }

    private invokeMiddleware(middlewares: IMiddleware[]) {
        // Todo: implement method
    }
}
```

3.3.3 Invoke Middleware Method

The "invokeMiddleware" method is responsible for calling the "run" method of each "Middleware" instance. The "invokeMiddleware" method is passed an array of the "Middleware" instances which have been added to the "MiddlewareChain".

When it is invoked it takes the first middleware off of the array and calls it, passing in the "context" parameter which is a store for data and the "next" parameter which is an anonymous function that will call another "invokeMiddleware" which is passed the array of remaining "middlewares" array as an argument.

Each middleware will need to make call the "next()" function that it is passed in order to continue the chain. A middleware can perform actions before and after

that function call, this is the crux of the pattern. Once all of the middlewares have been called "runChain" returns the updated context.

Code Ref. 7: invokeMiddleware Method

```
class MiddlewareChain implements IMiddlewareChain {
    private middlewares: IMiddleware[] = [];
    private context: any = {};

    runChain(context?: any) {
        if (context) this.context = context;
        this.invokeMiddleware([...this.middlewares])
        return this.context;
    }

    private invokeMiddleware(middlewares: IMiddleware[]) {
        if(!middlewares.length) return;
        const middleware = middlewares.shift();
        if(middleware){
            middleware.run(
                this.context,
                () => this.invokeMiddleware(middlewares)
            )
        }

    }
 }
```

3.3.4 Run Method

Just to prove that the middleware is working as intended, all these middlewares will do is "console.log" data and make minor changes to a "context" object.

Code Ref. 8: Full Script

```
class MiddlewareOne implements IMiddleware {
    run(context: any, next: () => void) {
        console.log('I Ran');
        context.paramOne = "one";
        console.log(1);
        next();
```

```
        console.log(4);
        console.log(context.paramTwo);
        context.end = 'here';
    }
}
```

Code Ref. 9: Full Script

```
class MiddlewareTwo implements IMiddleware {
    run(context: any, next: () => void) {
        console.log('I Ran Too');
        context.paramTwo = "two";
        console.log(2);
        next();
        console.log(3);
        console.log(context.paramOne);
    }
}
```

4 Testing it Out

First instantiate the "MiddlewareChain" class, then instantiate the two "IMiddleware" implemations. Next "add" the two middlewares to the "MiddlewareChain". Lastly set up some dummy "input" data. Now the system is all set up and ready to run. Calling "runChain" will execute the middleware system. You should see the logs count in order 1, 2, 3, 4 and that some data has been added to the context in the process.

Code Ref. 10: Full Script

```
const middlewareChain = new MiddlewareChain();

const mwOne = new MiddlewareOne();
const mwTwo = new MiddlewareTwo();

middlewareChain.add(mwOne);
middlewareChain.add(mwTwo);
```

```
const input = {start: 'here'};
const result = middlewareChain.runChain(input);
console.log(result);
```

5 Middleware Full Example

Code Ref. 11: Full Script

```
interface IMiddlewareChain {
    add(middleware: IMiddleware): void
    runChain(context: any): any
}

interface IMiddleware {
    run(context: any, next: () => void): void
}

class MiddlewareChain implements IMiddlewareChain {
    private middlewares: IMiddleware[] = [];
    private context: any = {};

    add(middleware: IMiddleware) {
       this.middlewares.push(middleware);
    }

    runChain(context?: any) {
        if (context) this.context = context;
        this.invokeMiddleware([...this.middlewares])
        return this.context;
    }

    private invokeMiddleware(middlewares: IMiddleware[]) {
        if(!middlewares.length) return;
        const middleware = middlewares.shift();
        if(middleware){
            middleware.run(
                this.context,
                () => this.invokeMiddleware(middlewares)
            )
```

```
        }
    }
}

class MiddlewareOne implements IMiddleware {
    run(context: any, next: () => void) {
        console.log('I Ran');
        context.paramOne = "one";
        console.log(1);
        next();
        console.log(4);
        console.log(context.paramTwo);
        context.end = 'here';
    }
}

class MiddlewareTwo implements IMiddleware {
    run(context: any, next: () => void) {
        console.log('I Ran Too');
        context.paramTwo = "two";
        console.log(2);
        next();
        console.log(3);
        console.log(context.paramOne);
    }
}

const middlewareChain = new MiddlewareChain();

const mwOne = new MiddlewareOne();
const mwTwo = new MiddlewareTwo();

middlewareChain.add(mwOne);
middlewareChain.add(mwTwo);

const input = {start: 'here'};
const result = middlewareChain.runChain(input);
console.log(result);
```