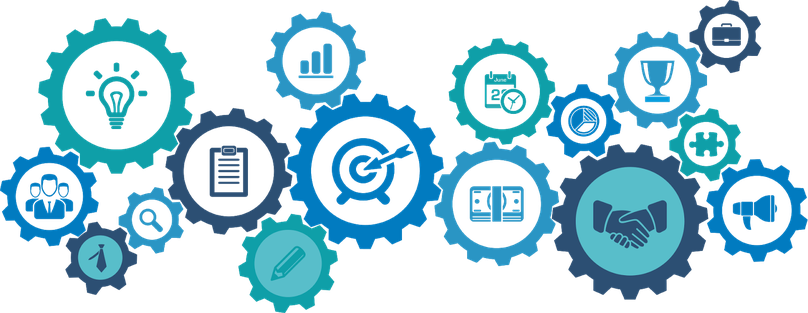
**Université Saint Joseph**

École Supérieure d’ingénieurs de Beyrouth (ESIB)

5th Year - Computer and Communications Engineering

2019-2020

**Microservices in Node.js**



**By:**

**Gayel (Imad) Abou Imad**

**Cindy (Fady) Chedid Al Rahbani**

**22/09/2019**

Table of Contents

[1. Introduction 5](#_Toc23088186)

[1.1. Definition 5](#_Toc23088187)

[1.2. Advantages of using microservices 5](#_Toc23088188)

[1.3. Drawbacks of microservices 6](#_Toc23088189)

[1.4. Mongo 6](#_Toc23088190)

[1.5. Docker 6](#_Toc23088191)

[1.5.1. What is docker? 7](#_Toc23088192)

[1.5.2. Containerization 7](#_Toc23088193)

[1.5.3. Example 7](#_Toc23088194)

[2. Creating Microservices with Node.js 7](#_Toc23088195)

[2.1. Requirements 7](#_Toc23088196)

[2.2. Visualization of the problem 7](#_Toc23088197)

[2.3. Implementing the services 8](#_Toc23088198)

[2.3.1. DataAccess.js 8](#_Toc23088199)

[2.3.2. EntityModel.js 9](#_Toc23088200)

[2.4. Using Mongo DB 9](#_Toc23088201)

[2.4.1. Connection to the database 9](#_Toc23088202)

[2.4.2. MongoDB Commands 10](#_Toc23088203)

[3. Implementing the Front-End – Angular 8 10](#_Toc23088204)

[3.1. Creating the services in the front-end 10](#_Toc23088205)

[3.2. Fetching data in the container 11](#_Toc23088206)

[3.3. Showing data in the web application (Front – end) 12](#_Toc23088207)

[3.4. Using MVC architecture 12](#_Toc23088208)

[3.4.1. Using MVC in the front-end 13](#_Toc23088209)

[3.4.2. Using MVC in the back end 13](#_Toc23088210)

[4. Using Docker 13](#_Toc23088211)

[4.1. Some commands to work with docker 13](#_Toc23088212)

[4.2. Docker File 14](#_Toc23088213)

[4.3. Docker Compose 14](#_Toc23088214)

[5. Connecting services using the MQTT protocol 16](#_Toc23088215)

[5.1. Why MQTT 16](#_Toc23088216)

[5.2. Implementation 16](#_Toc23088217)

[6. HeartBeat 19](#_Toc23088218)

Table of figures

[Figure 1: Difference between a monolithic application and a microservices application 4](file:///D:\Gayel%20Abou%20Imad\USJ\Semestre%209\Integration%20Des%20Applications\ProjetIntegrationDesApplications\Rapport%20Integration%20Des%20Applications.docx#_Toc23088219)

[Figure 2 - Diagram showing a basic conception of our application 7](#_Toc23088220)

[Figure 3 - Schema explaining the role of each file 10](#_Toc23088221)

[Figure 4 - Schema explaining the concept of the MVC architecture 12](#_Toc23088222)

[Figure 5: Schema explaining the functioning of MQTT protocol 15](file:///D:\Gayel%20Abou%20Imad\USJ\Semestre%209\Integration%20Des%20Applications\ProjetIntegrationDesApplications\Rapport%20Integration%20Des%20Applications.docx#_Toc23088223)

# Introduction

## Definition

Microservices are an architectural approach based on building an application as a collection of small services.  Applications are traditionally “monolithic” which means they are built as a single, autonomous unit. Everything is contained inside the unit. So, every change (even minor changes) can be slow or tedious as it affects the entire system. So, scaling specific functions or components of the application, also means you must scale the entire system.

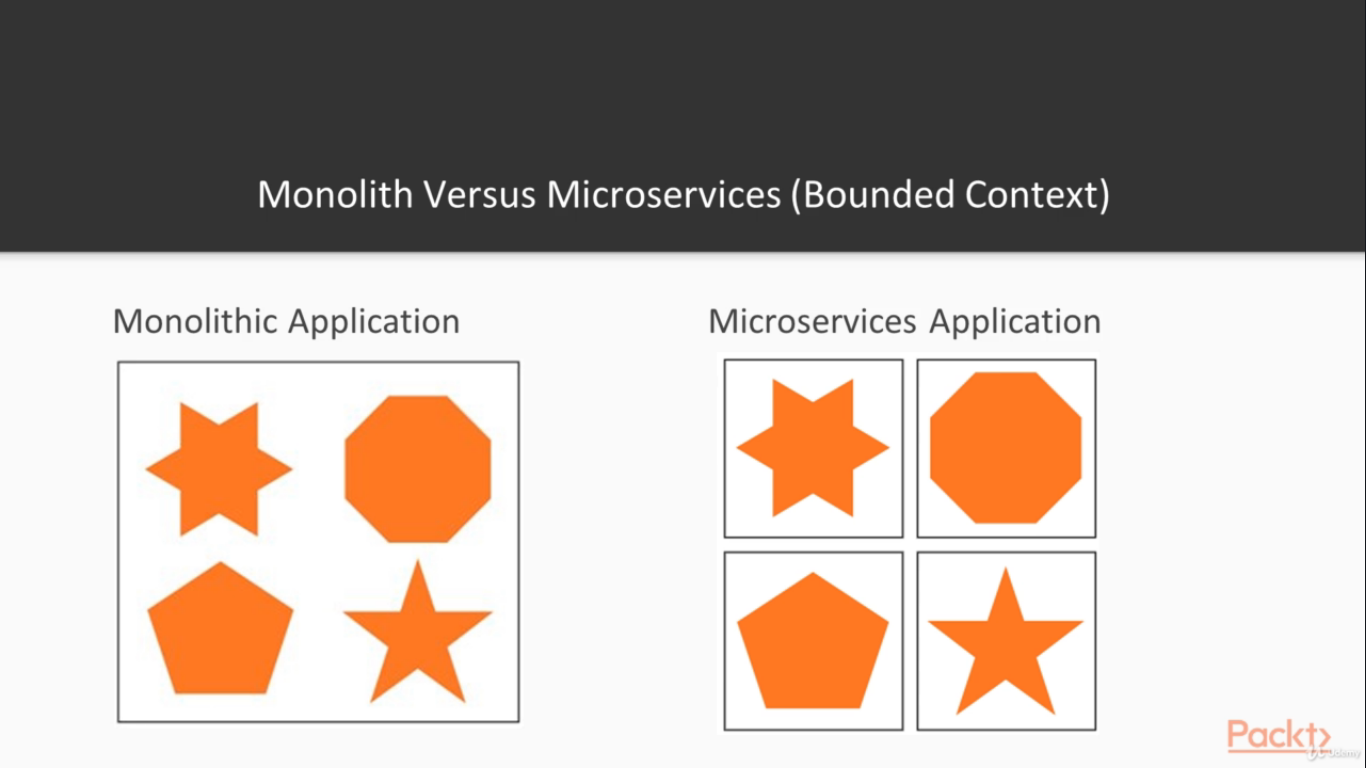


Figure 1: Difference between a monolithic application and a microservices application

Other problems with a monolithic approach in an online application are:

* **Inflexibility:** it cannot be built using different technologies
* **Potentially unreliable:** if even one feature of the system does not work, then the entire system does not work
* **Unscalable:** applications cannot be scaled easily, since each time the application needs to be updated, the complete system has to be rebuilt
* **Not suitable for continuous development:** many features of an application cannot be built and deployed at the same time
* **Slow development:** As you can likely guess from the preceding points, development in monolithic applications takes a lot of time, since each feature has to be built individually, one after the other, rather than allowing multiple features to be worked on concurrently

This is where microservices come to the rescue!

Instead of containing everything in a single unit, the microservices-based application is broken down into smaller, lightweight pieces based on a logical construct. The application consists of independent small (micro-) services, and when we deploy or scale the app, individual services get distributed within a set of machines which we call “a cluster” in the service fabric world.

Each service has its own unique and well-defined role, runs in its own process, and communicates via HTTP APIs or messaging. Each microservice can be deployed, upgraded, scaled, and restarted independently of all the sibling services in the application. They are typically managed by an automated system, making it possible to deploy frequent updates to live applications without affecting the end-users.

## Advantages of using microservices

* Allows us to build, operate and manage services independently, and we can easily scale them out based on the resources they need.
* Microservices take a lot of infrastructure risk out of the project straight away. With the infrastructure made almost invisible, microservice teams can iterate quickly.
* Each developer on a team can avoid getting tangled up in the underlying infrastructure and focus on their piece of the project. Then, in production, if individual project modules don’t work exactly right together, it’s easy enough to isolate, disassemble and reconfigure them until they do. If shoppers aren’t big fans of the mall’s specialty ketchup store, a shoe store can be built in its place. It offers better resource utilization and cost optimization.
* Microservices have their own load balancer and execution environment to execute their functionalities, and at the same time, capture data in their own databases.
* Finally, microservices offer language and platform freedom, so teams can choose the best language for the job at hand (even if that’s .NET for one team and Node.js for another team).

## Drawbacks of microservices

* Microservices are not automatically the right solution for every project. When you are running multiple instances of the same service or worker, you don’t necessarily need microservices. A well-built monolithic system can scale just as well for some classes of problems.
* One of the big problems with microservices is “orchestration”, which means how to integrate the services with a guide to drive the process, much like a conductor in an orchestra. Integrating microservices can be quite complex.
* Another complex process is “discovery” which is how applications and (micro)services locate each other on a network.
* Moving away from a monolithic app architecture means the loss of an opinionated workflow that previously glued all the pieces together.
* There is a risk in getting a very fragmented system where developers need to spend a lot of time and effort on gluing together services and tools, and where there’s a lack of common patterns and platforms that makes it difficult to work across different projects.
* Microservices can also require increased testing complexity and possibly increased memory/computing resources.
* It’s possible to create un-scalable microservices. It all comes down to how well you apply the fundamental principles. It’s all too easy to jump into shopping for all the microservices you want to apply without first truly considering the problem set you’re applying them to.

## Mongo

MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use. MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time. The document model maps to the objects in the application code, making data easy to work with.

## Docker

Plainly put, Docker is an open-source technology used mostly for developing, shipping and running applications. With it, you can isolate applications from their underlying infrastructure so that software delivery is faster than ever. Docker’s main benefit is to package applications in “containers,” so they’re portable for any system running the Linux operating system (OS) or Window OS. Though container technology has been around for a while, the hype around Docker’s approach to containers has moved this approach to the mainstream as one of the most popular forms of container technology.   
The brilliance of [Docker](https://www.simplilearn.com/docker-tutorial-article) is that, once you package an application and all its dependencies into a Docker run container, you ensure it will run in any environment.

### What is docker?

* Enterprise-ready container platform
* Provides security and governance
* Automation by design
* Support and certification

### Containerization

Docker takes advantage of a concept called containerization.

* It solves the problem of how to reliably move software between environments
* Encapsulates the entire runtime environment
* Includes the application and its dependencies, libraries and other binaries

Docker and Virtual box cannot be installed on the same machine and work simultaneously because of Hyper-V. So, in order to use docker we install docker Toolbox from:

<https://github.com/docker/toolbox/releases>

After installing it, click on docker QuickStart terminal. When it’s done, we can now use docker.

### Example

As an example, we can test the following command

$ docker run hello-world

Or

$ docker run -it ubuntu bash

To create a container:

$ docker run --name <container\_Name> -it <image\_name>

# Creating Microservices with Node.js

## Requirements

* Have Node.js installed
* Express package

Express package can be installed using

$ npm install express --save

## Visualization of the problem

Let’s consider we want to create an application where customers are invited to order books.  
Using microservices, we will need to create for each of the three entities: **book’s entity**, **customer’s entity** and **ordering’s entity** a service in a backend folder. And then for each service we will have to create its own database. Finally, each service will be deployed in a container (Docker’s container), and the three services will be able to interact with each other’s with the help of the API Gateway.  
The following schema explains the whole situation.

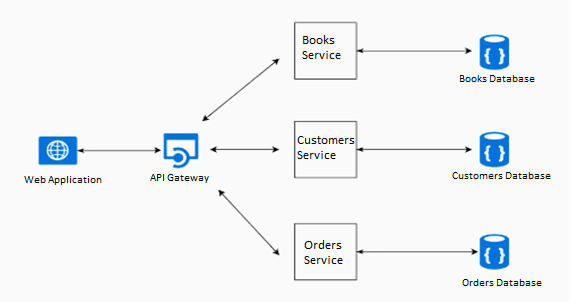


Figure 2 - Diagram showing a basic conception of our application

In our case, we will develop a web application only in order to focus more on the services, the databases and the API Gateway.

## Implementing the services

The three services that we need to create in our backend folder of the application are:

* Book service
* Customer service
* Ordering service

These services will communicate with the API Gateway through API calls and MQTT publish/subscribe protocol. The structure of each service should look like the following:

│   app.js

│   Dockerfile

│   package-lock.json

│   package.json

|

└───models

        booksModel.js

        DataAccess.js

### DataAccess.js

In DataAccess.js, an asynchronous function getEntities is implemented in order to make the connection to the corresponding collection in the database and applying the query, noting that the collection name, the database name is given as arguments to this function.

DataAccess.prototype.GetEntities = async *function* (*dbName*, *CollectionName*)

### EntityModel.js

In EntityModel.js, an asynchronous function getInventory is implemented calling the function getEntities giving it the right arguments: database name and collection name corresponding to the entity service.

Example (Book service):

Model.prototype.GetInventory = async *function*(){

    try{

*var* response = await DataAccess.GetEntities('books\_microservice', 'books');

        return response;

    }catch(err){

        return err;

    }

};

## Using Mongo DB

For each of our services, a different database is needed, database for the books, database for the customers and a database for the orderings.

### Connection to the database

As we said the connection to the database from the application is easier when we use mongo dB package. So, to grant the access we need to set the following:

*var* DataAccess = *function*(){

    this.MongoClient = require('mongodb').MongoClient, assert = require('assert');

    this.Mongo = require('mongodb');

    this.DBConnectionString = 'mongodb://mongo:27017';

};

**NB:** In the connection string we have set mongo as IP variable because we will connect the service to the container (who’s name is mongo) that has the database.

Then in the function that needs the connection we can make the connection to the client, then database then to the collection as follows:

try {

*var* response = await that.MongoClient.connect(that.DBConnectionString, {

        useNewUrlParser: true,

        useUnifiedTopology: true

      }

    );

*var* database = await response.db(dbName);

*var* collection = await database.collection(CollectionName);

*const* items = await collection.find();

*const* documents = await items.toArray();

    return documents;

}catch(err){

    if(err.name == 'MongoNetworkError'){

        console.log("Connection to DB Failed");

        return 'Connection to DB Failed';

    }

    return err;

}

### MongoDB Commands

1. How to run Mongo DB **Locally**:

We should open the shell and write down this command

$ mongod --bind\_ip <YOUR\_IP\_ADDR>

**or if you saved the data base data in a specific folder:**

$ mongod --bind\_ip <YOUR\_IP\_ADDR>

--dbpath "C: //YOURDIRECTORY//ProjetIntegrationDesApplications//database"

Then in another shell

$ mongo.exe <YOUR\_IP\_ADDR>

1. How to work in Mongo DB:
2. To create a database:

$ use <Database\_Name>

1. To create a collection:

$ db.createCollection(“<Coll\_Name>”)

1. To add a row to the collection

$ db. <Coll\_Name>.insert({<all the key-value pairs>})

1. To get all the collection

$ db. <Coll\_Name>.find ()

# Implementing the Front-End – Angular 8

## Creating the services in the front-end

In order to fetch data, a service needs to be created. For example, in the terminal in the services folder, the following command needs to be written:

$ ng generate service <Service\_Name>

In this service you will need to do an Http request, so you need to import the Http module in the app module.

import { HttpClientModule } from '@angular/common/http';

and place HttpClientModule under the imports array in NgModules.

Now after importing it in the app module, you need to configure the service. You should import the http client and the http headers packages in the service as follows:

import { HttpClient } from '@angular/common/http';

import { HttpHeaders } from '@angular/common/http';

and then add HttpClient to the constructor in order to use them in the functions.

*constructor* (private *http*: HttpClient) { }

And finally, to make the call to the backend, a function should be implemented in each service. This function will be called by the container’s component. Example :

  getBooks() {

*const* httpOptions = {

      headers: new HttpHeaders({

        'Content-Type': 'application/json',

        'Access-Control-Allow-Origin': '\*',

        'Access-Control-Allow-Credentials': 'true'

      })

    };

    return this.http.get('http://192.168.99.100:8080/getBooks', httpOptions);

  }

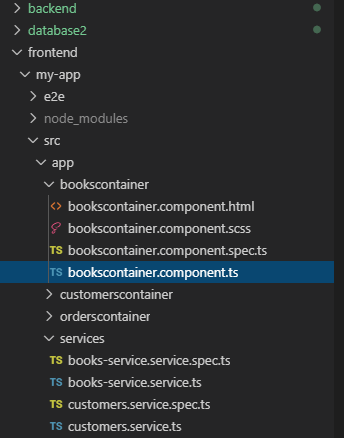
As we can notice, the call is made to “192.168.99.100” which is the docker toolbox’s IP. The port is set to 8080 which is the **API Gateway**’s port.

## Fetching data in the container

We have created a component for each service. The following figure show the different parts of a container.

Showing data in the web app (front – end)

Logic of the component:   
Call of the function from the service.



File with same name as component

File inside the component

Fetching data from the back – end

Figure 3 - Schema explaining the role of each file

In the Books Container, we should fetch the data in the component.ts file this function will be used as follows:

  public books;

*constructor*(private *booksService*: BooksServiceService) { }

  ngOnInit() {

    this.showBooks();

  }

  showBooks() {

    this.booksService.getBooks()

    .subscribe((*data*) *=>* {

      this.books = data;

    });

  }

In order to use this service, it should be imported as follows:

import { BooksServiceService } from '../services/books-service.service';

## Showing data in the web application (Front – end)

After fetching the data in the component ts file. We should show it in the html.

Since the data in ts is binded to the data in the ts we can simply use the books object that's in the ts by using **{{ books }}** in the html file as follows:

<div \*ngFor="let book of books">

  {{book.name}}

</div>

As the code above shows, we are looping over the array **books** of objects using \*ngFor, then for each book, displaying its name. In our case the code is:

<mat-list>

  <mat-list-item \*ngFor="let book of books">

    <h4 mat-line>{{ book.name }}</h4>

    <p \*ngIf="book.status==0; else notShow" style='color: blue'>Available</p>

    <ng-template #notShow>

      <p style='color: red'>Taken</p>

    </ng-template>

    <mat-divider></mat-divider>

  </mat-list-item>

</mat-list>

In few words, this code will show the name of each book followed by its status (taken of available).

## Using MVC architecture

As we can see, in both the front-end and the back-end the architecture used is the: Model – View – Controller architecture.

* **Model:** The Model part of an MVC architecture encapsulates business logic as well as data access. It can be a set of functions (procedural model) or classes (object-oriented model).
* **Controller:** The Controller part manages the dynamics of the application. It is the link between the user and the rest of the application.
* **View:** The view part deals with interactions with the user: presentation, data entry and validation.

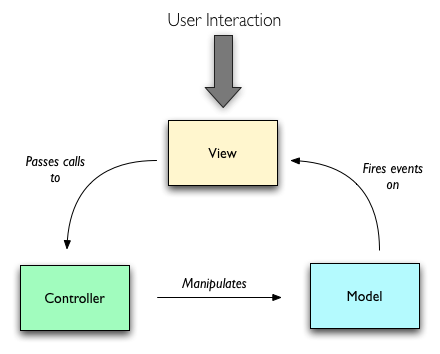


Figure 4 - Schema explaining the concept of the MVC architecture

### Using MVC in the front-end

In the front-end, the MVC architecture is followed as following:

* **Model**: **Entity-service.ts** – file where the data is being fetched
* **View:** **EntityContainer.component.html** – file where the design and the representation of the app is written.
* **Controller:** **EntityContainer.component.ts** – file where the logic is built, the function from the service is called and given to the html file.

### Using MVC in the back end

In the back-end, the MVC architecture is followed as following:

* **Model**: DataAcess.js – where the data is being transferred from and to mongo DB
* **View:** app.js
* **Controller:** EntityModel – where the function of the DataAccess is called and given to the app.js

# Using Docker

## Some commands to work with docker

In order to see all the docker processes that are running, open another command line and write the following command:

$ docker ps

In order to see all the docker images, write the following command:

$ docker images

In order to stop a container, write the following command:

$ docker stop <Container\_Name>

In order to create a container

$ docker run –name <Container\_Name> -it <Image\_Name>

In order to remove a container

$ docker rm <Container\_Name> or <Container\_ID>

## Docker File

A docker file is the main configuration of the docker container. We can set all we need in a single file as follows. The followings is an example of a Docker file corresponding to a microservice:

1. FROM node:latest

2. COPY . /src

3. WORKDIR /src

4. RUN npm install

5. EXPOSE 8081

6. CMD node app.js

1. Here we are setting the environnment (we need the node package in this container so we pull it)
2. now we need to copy all the booksservice folder to the container. This means the current directory and /src is the destination folder in the container
3. now we should set the working directory in the container
4. and we need to specify the command that should be ran before building the service which is npm install so we install all the dependencies
5. in order to make the calls to the backend we call on port 8081. now that the service is in the container. the container looks like a black box so we should call the container's port. this port should be binded to the microservices port which is 8081 so we expose port 8081
6. after setting everything we should start the app (microservice) so we set the command

The followings is an example of a Docker file corresponding to the front end:

1. FROM node:latest

2. COPY /my-app /src

3. WORKDIR /src

4. RUN npm install

5. RUN npm install -g @angular/cli

6. EXPOSE 4200

7. CMD ng serve --host 0.0.0.0

1. Here we are setting the environnment (we need the node package in this container so we pull it)
2. now we need to copy the my-app folder to src folder in the container
3. now we should set the working directory in the container
4. npm install that will install all the dependencies
5. npm install the angular cli so we can run later the CMD ng serve
6. expose the port 4200
7. set the first command

## Docker Compose

If we have multiple docker file, it is better to create a docker compose which will run them all in once.

version: '2'

services:

# Front End

  frontend:

    container\_name: FrontEnd

    build: ./frontend

    ports:

      - "4200:4200"

# Service #0 API Gateway

  apigateway:

    container\_name: APIGateway

    build: ./backend/APIGateway

    ports:

      - "8080:8080"

# Service #1 Books Service

  books:

    container\_name: BooksService

    build: ./backend/booksservice

    ports:

      - "8084:8084"

# Service #2 Customers Service

  customers:

    container\_name: CustomersService

    build: ./backend/customersservice

    ports:

      - "8082:8082"

# Service #3 Orders Service

  orders:

    container\_name: OrdersService

    build: ./backend/ordersservice

    ports:

      - "8083:8083"

# MongoDB

  mongo:

    container\_name: mongodb

    restart: always

    image: mongo

    volumes:

      - ../data/db:/data/db

# MongoDB GUI

  mongo-express:

    image: mongo-express

    restart: always

    ports:

      - "8081:8081"

Above is the docker compose file. We specify the build file which is the docker file corresponding to each microservice and the local port that will be binded to the container’s port.

# Connecting services using the MQTT protocol

## Why MQTT

As mentioned before, the front – end fetches data from the back – end by making http request (post) requests on port 8080.

**Now how can we make the three services in the back – end share information and talk to each other? MQTT is the answer.**

MQTT is a simple messaging protocol, designed for constrained devices with low bandwidth.  
MQTT allows to send commands to control outputs, read and publish data from and to multiple interfaces. Therefore, it makes it easy to establish a communication between multiple devices.  
In our case, each of our three services: orders, customers and books must be updated whenever any action is made.

For example:

Whenever a customer orders a book, a message:

* To the **books service** must be sent using the MQTT protocol to update the books database about the availability of the book
* to the **orders service** must be sent to add the order to the database
* to the **customers service** in order to update the customer’s book list

So, the API Gateway is the publisher that will publish messages to the MQTT broker. Each of the 3 other services (books, orders and orders) that are subscribed to the MQTT broker will receive the messages published by the customers.

Figure 5: Schema explaining the functioning of MQTT protocol

API Gateway

Orders Service

Books Service

MQTT Broker

Message

Message

Message

Message

Customers Service

## Implementation

As we mentioned in the paragraph above. The API gateway will receive the order from the front end as a JSON which format will be :

*let* data = {

      "bookID": this.selectedBook,

      "ClientID": this.selectedCustomer

    }

These fields are filled using 2-way data binding feature. Angular provides this feature as follows:

<mat-form-field style="margin-left: 2em">

  <mat-label>Choose a Customer</mat-label>

  <select matNativeControl  [(ngModel)]="selectedCustomer">

    <option value={{customer.firstName}} \*ngFor="let customer of customers">{{customer.firstName}} {{customer.lastName}}</option>

  </select>

</mat-form-field>

<mat-form-field style="margin-left: 2em">

  <mat-label>Choose a Book</mat-label>

  <select matNativeControl [(ngModel)]="selectedBook">

    <option value={{book.name}} \*ngFor="let book of books">{{book.name}}</option>

  </select>

</mat-form-field>

<button mat-raised-button color="primary" style="margin-left: 2em" (click)="makeOrder()">Order!</button>

We should focus on the ngModel field and the button which will trigger the makeOrder() function.

Using MQTT as follows in the API Gateway, the JSON will be published to the 3 services in order to update each of the corresponding databases.

app.post("/receiveOrder", async *function* (*req*, *res*) {

  console.log("fet 3al received", req.body.bookName)

*var* client = mqtt.connect('mqtt://192.168.16.7:1883');

  client.on('connect', *function* () {

    console.log("Connected")

    client.subscribe('GayelTest', *function* (*err*) {

      console.log("Subscribed")

*var* message = JSON.stringify(req.body);

      if (!err) {

        client.publish('GayelTest', message)

      } else console.log("erroor")

    })

  })

  client.on('message', *function* (*topic*, *message*) {

    // message is Buffer

    console.log(message.toString())

    res.send(message.toString())

    client.end()

  })

});

In a few words, this script is connecting to the mqtt local server, then subscribing to the topic and finally publishing the message.

Whenever a message is present on the topic, it also gets it and sends it as a response to the call.

On the other side, as an example in the books service, the message are received as follows:

// Client Connection

*var* client = mqtt.connect('mqtt://192.168.16.7:1883')

client.on('connect', *function* () {

    console.log("Connected")

    // Client Subscription

    client.subscribe('GayelTest', *function* (*err*) {

        console.log("Subscribed");

        if(err){

            console.log("error");

        }

    })

})

// When someone else publishes data

client.on('message', *function* (*topic*, *message*) {

    // message is Buffer

*let* message2 = JSON.parse(message.toString());

    model.orderBook(message2.bookID);

  })

1. Connection to the local mqtt server
2. Subscrtiption to the topic
3. Listen to any message that will arrive
4. Trigger the function in the model

The function in the model will call the data access function which will connect to the database and make the necessary changes if possible.

NB : a book (or customer) that is already taken, will not figure in the list of available book sot order.

# HeartBeat

So the first thing we do once we start creating our Micro's service that serves as an API gateway is to add a heartbeat endpoint to each of the APIs on each of the microservices so that in app.js we can find a route called Heartbeat that returns the status of the API along with its address and port.

This is added to each microservice and will be used by the API Gateway to monitor the status of each microservice.

In the API Gateway a route called get Status contains all the routes and the corresponding ports that should be tested. A promise is created and will call the model which will call the function that will make a call to each of microservice’s heartbeat routes.

app.get("/getStatus", async *function* (*req*, *res*) {

*let* services = [

    {

      name: "Books Microservice",

      address: "192.168.99.100",

      port: 8084

    },

    {

      name: "Customers Microservice",

      address: "192.168.99.100",

      port: 8082

    },

    {

      name: "Orders Microservice",

      address: "192.168.99.100",

      port: 8083

    },

  ];

*var* promises = await model.GetStatus(services);

  Promise.all(promises)

    .then(*function* (*values*) {

      for (*var* i = 0; i < values.length; i++) {

*var* value = values[i];

        if (value.address && value.port) {

          for (*var* j = 0; j < services.length; j++) {

            if (services[j].address == value.address && services[j].port == value.port) {

              value.config = services[j];

            }

          }

        }

      }

      res.send(values);

    })

    .catch(*function* (*err*) {

      res.send(err);

    });

});

Then after finishing all the calls, a response to the call will be formulated as follows:

[

{

"success": true,

"name": "Customers Microservice",

"address": "192.168.99.100",

"port": 8084

},

{

"name": "Customers Microservice",

"success": true,

"address": "192.168.99.100",

"port": 8082

},

{

"success": true,

"name": "Orders Microservice",

"address": "192.168.99.100",

"port": 8083

}

]

As we can see, we can notice that the 3 microservices are on because success is equal to true.