Haaga-Helia
University of Applied Sciences

Productivity tool for microservices implementation

Dipesh Singh Yadav

Master's Thesis
Degree Programme in
Information Systems Management
2018

Haaga-Helia
University of Applied Sciences

Abstract

10 Dec 2018

Author(s)

Dipesh Singh Yadav

Degree programme

Information Systems Management, Master's Degree

Thesis title	**Number of pages + appendix pages**
Productivity tool for microservices implementation	**48 + 13**

Microservices is a fast growing architecture for modern applications. Microservices separates and decouples large monolithic application to small independent components. This benefits businesses to implement and deploy components independently and fast. Rapidly changing business functions or experiments can be implemented as separate microservice. This is gives power to test new features in real world as fast. Each individual component can be easily improved and changed fast. This helps businesses to keep up with the pace of market and lays foundation to innovate. This gives businesses major competitive advantage.

While microservices provides elegant solution for fast implementation and deployment problems, this architecture suffers drawbacks in other areas. The architecture is very complex as now single monolith application is transformed into distributed and interconnected tiny applications (microservices). Developing on this distributed architecture becomes complicated. Instead of managing one development server, developer has to now additionally manage many development servers hosting respective microservices. This tools helps developers to overcome this development challenges by providing GUI tool. This tool will make developing features in microservices easy, fun and productive.

Keywords

Microservices, Productivity, Development, Integration, Debugging, Docker, Nodejs, Electron app

Table of contents

1 Introduction

Progress in digital technologies has changed how business operates. New emerging businesses and innovations has shaped and evolved market trends in unpredictable way. New kind of demands and trends while rapidly decaying the existing ones has made businesses difficult to survive. Market is changing fast forcing business to be agile and innovative.

Most businesses services are powered by single or set of huge and complex monolithic applications. These application are small and simple in beginning. As times goes by, new business functions are added making application large and complex. Implementing and deploying even tiniest change in such application takes tremendous amount of time. In order to be agile and innovative, business needs to keep developing and testing new concepts. Given an architecture which takes such long time to deploy new concepts makes it challenging for businesses to keep up with the market pace.

Luckily there is a widely used modern architecture which supports faster deployment strategy. Microservices is particularly adopted to facilitate rapid implementation and deployment. Microservices breaks down application into smaller applications (microservices). These applications are now separate and independent which makes it possible to implement and deploy each microservice independently. Each microservice application works and communicates with other microservices to provide business services. In user point of view, it is a single application providing services but in reality it is many tiny applications are working together as single unit.

One challenging and overlooked aspect is how team develops and implements these microservices. To function properly microservice needs communicates with other microservices or other resources such as databases. This add a bit of effort for developer to start up the other dependent resources. Also, those dependent microservice may need other microservices. For example, as shown in figure (Figure 1) a developer wants to implement a feature in microservice A. Some data for this feature is coming from Microservice B and C. So for implementing this feature Microservice A requires Microservice B and Microservice C to be running. Developer can only then start implementing and testing the feature in target microservice A. This thesis provides a productivity tool named Nova designed to solve this microservices development challenges. The tool Nova will help developers to easily and smoothly implement features in microservices. This makes development fun and productive.

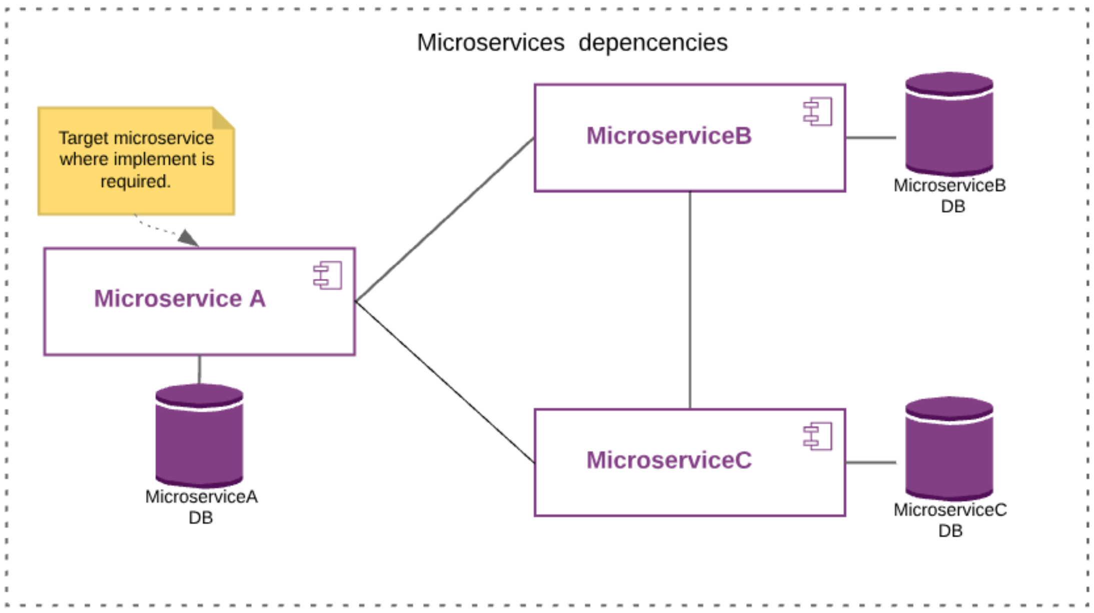


Figure 1. Microservices implementation depencencies

We start by looking at related solutions and strategies in market at the moment used for microservices development. Next chapter describes demo application that will be used as an example to understand monolith and microservices. Next chapter describe monolith architecture and its concepts. After it follows in depth view of microservices architecture and its concepts. Next chapter describes comparison of monolith and microservices explaining the benefits and reasoning for adopting complex microservices architecture. Chapter after that take a deeper look in the development problems existing while developing in microservices. Next chapter demonstrates demo application built in microservices architecture reflecting the problems mentioned in previous chapter. Finally, we take a look at the solution tool this thesis provides to resolve the development challenges.

1.1 Objectives

This tool Nova helps developers to implement microservices with greater speed and focus. It reduces time and frustration for configuring and starting up development environment. It makes the work productive, easy and fun. Without such tool developer needs to get burdened with several trivial tasks required before beginning the actual implementation. With this tool developer free themselves with this burden and hassle. It enables developers to focus on actual feature to implement creative and bug free solutions.

More detailed objectives of this project are as follows.

- Easily start-up dependent resources locally such as services or required external resources such as databases.

- Developers can easily navigate to project workspace to start coding immediately. Instead of navigating and finding the project in directory, once set up developers can load the workspace with ease.

- Centralize logging place to view the logs of dependent services. This help to debug and trace application data flows easily.

1.2 Research questions

- How does this tool increase productivity?
- How does this tool help implementation of features with greater focus?
- How does this tool help to easily debug microservices?
- Why is this tool easy to use?

1.3 Scope

This project is currently targeted for nodejs (Nodejs, 2018) microservice development. Nodejs is a javascript runtime where applications can be written in javascript. Even though nodejs is not as powerful as java or C# for enterprise application, most common microservices web application does not require much power. Nodejs does not need extra overhead to get started with the application. It is lightweight and easy to use which makes nodejs excellent candidate for microservices.

This tool is designed to work with popular source code editor Visual Studio Code (Visual Studio Code, 2018). Visual Studio Code is lightweight and is as powerful as full IDE. It has great support for javascript and web programming languages. Additional community driven extensions can be added to make this editor even more powerful.

The targeted tool is a desktop application created using Electron framework (Electron, 2018). With Electron framework we can use javascript, html and css to easily crate cross-platform desktop GUI applications. For this project the application will only support OSX.

For simulating and demonstrating the tool, a simple microservice driven web application is created which will be used to describe the features of this tool.

1.4 Abbreviation and Terminology

Repository

Version controlled code sharing tool such as svn and git.

IDE

Integrated development environment is a tool which includes source code editor along with other development tools which helps in debugging, building application, running development server and even starting database servers. Example: Visual studio, Eclipse, Intellij IDEA.

Source Code Editor

Lightweight editor for source codes. Examples: notepad++, sublime test and visual studio code.

OS

Operating systems such as windows, linux and osx.

GUI

Graphical user interface. Applications provide graphical interface which can be interacted with mouse. They are very usable as most operations can be perform by single click.

CLI

Command line interface. Interface provided by application where user interacts with help of commands given by keyboard. Lacks usability as it does not have much visual aids. user has to learn and type long commands. Although lacking usability it is more powerful than GUI. It provides users with much wider range of operations and capability than GUI.

1.5 Concepts

Deployment

Deployment involves packaging the target application and testing the full application thoroughly both manually and using automated systems. When the package passes all required and planned tests then it is released to production. Again after release the application must be tested to check if new and existing features are working properly. Successful deployment requires proper planning and coordination. Doing this for each new feature even for tiny change is time consuming and expensive. So, team has to plans and groups features properly for deployment as shown in figure (Figure 2). Usually team plans to implement and deploy bunch of features depending on their capability. They deployment cycle is usually very long and missed feature need to wait another deployment schedule. This deployment strategy tiny change in one business functions requires to wait for features of other business functions to be deployed in next schedule. This is not agile and very slow to test new concepts fast.

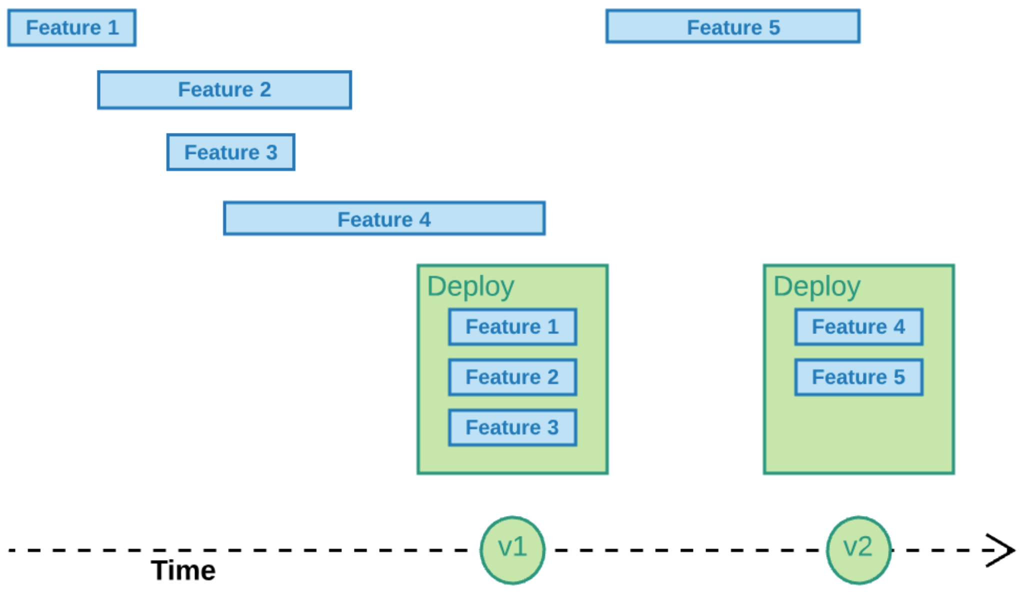


Figure 2 Traditional deployment strategy

Containerization and Virtualization

Containerization is similar to virtualization. As shown in figure (Figure 3) In virtualization, target application run on top of desired guest operating system. This guest OS is running on top of virtual hardware provided by virtual machine. Containerization does not require virtual hardware and it runs by sandboxing the host operating system. Each container has light weight guest OS where application can run. Since virtual machine needs to create virtual hardware stack, starting up virtual machine is slow and consumes lots of resource from host operating system and machine. Containers are very lightweight and does consumes only what is needed from host OS. It can start up in few seconds making containerization much more favourable. Containerization gives ability to pack our target application together with its operating system and required runtime libraries. So, the containerized application always starts and runs in same consistent way in any machine. No more “it works in my machine” phrase.

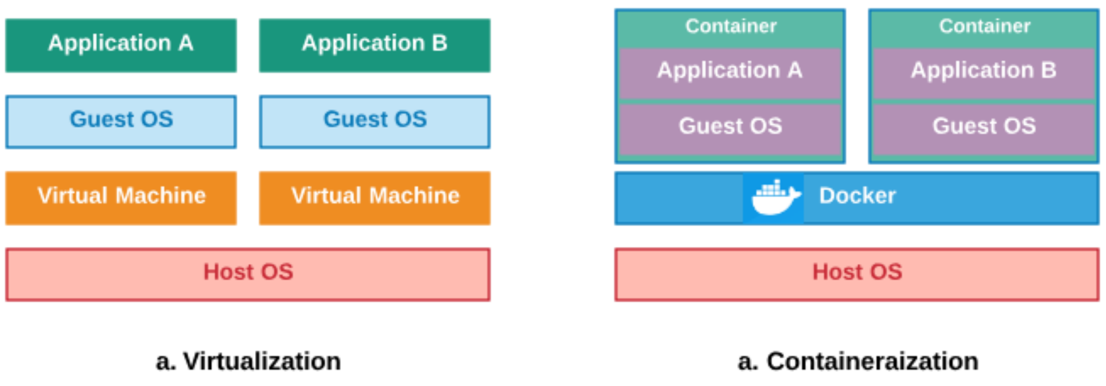


Figure 3 Virtualization and containerization

Container orchestration

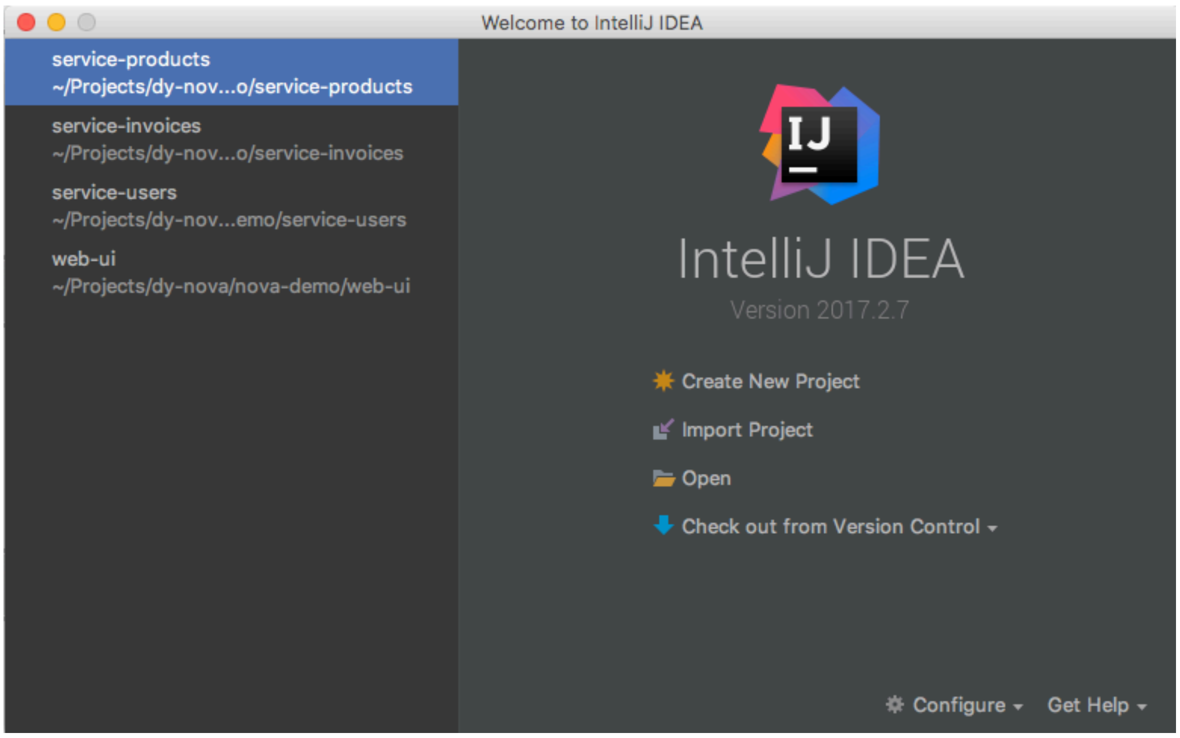
Microservices run in container is efficient and best practice. However, it gets difficult and challenging to manage and operate large number of containers manually. Microservice containers need to be deployed and scaled on demand. They need to be managed and correctly coordinated so they can communicate each other. There exist several tools to help automate such orchestration tasks. Most popular is the Kubernetes (Kubernetes, 2018). It is an open-source container orchestrating tool designed by Google. Kubernetes helps to automate deployments, scaling and much more. Kubernetes connects several nodes as single cluster unit. Microservices containers is then automatically deployed and scaled in this cluster.

2 Related works

2.1 Development using IDE

IDE is productive approach for developing monolith applications. Developers navigate the project code, makes and tests the implementations by starting the application with the help of IDE. Multiple monolith application projects are implemented by adding the project to the IDE workspace. Microservices are basically multiple projects. So, in naive way developers just adds and configures all microservices projects in the IDE similar to multi monolith projects.

If we take a look at microservices development process in popular IDE (Intellij Idea, 2018) as shown in figure (Figure 4) we have four independent microservices application. Developer has to choose on application to work with. Developers can open another application in separate window to start stop the selected application. In figure (Figure 5) we have web-ui application which requires products service application to start-up which is loaded in separate window. As one can imagine development like this is not productive.


Welcome to IntelliJ IDEA

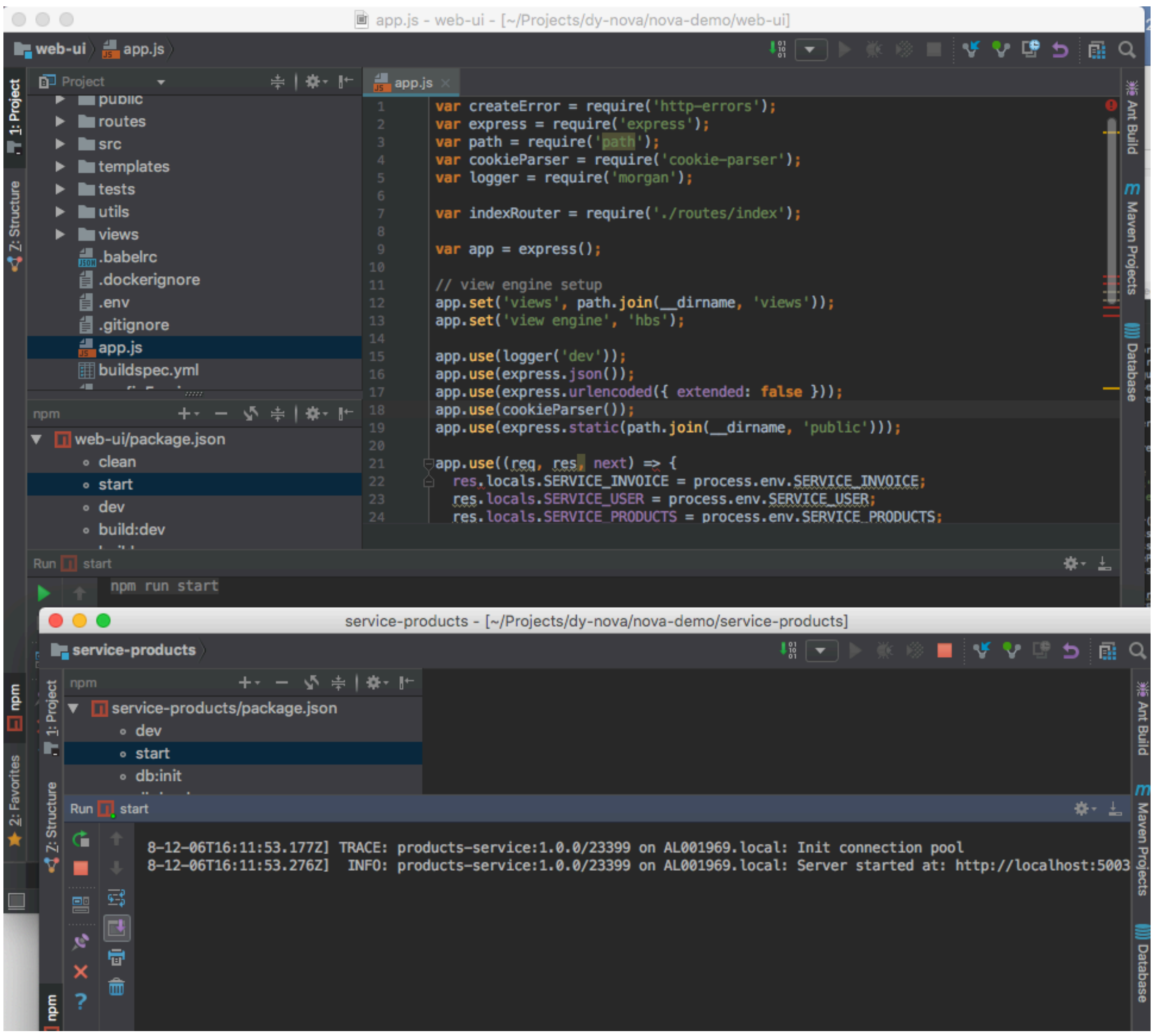
service-products
~/Projects/dy-nov...o/service-products

service-invoices
~/Projects/dy-nov...o/service-invoices

service-users
~/Projects/dy-nov...emo/service-users

web-ui
~/Projects/dy-nova/nova-demo/web-ui

Figure 4 Microservices in IntelliJ IDEA



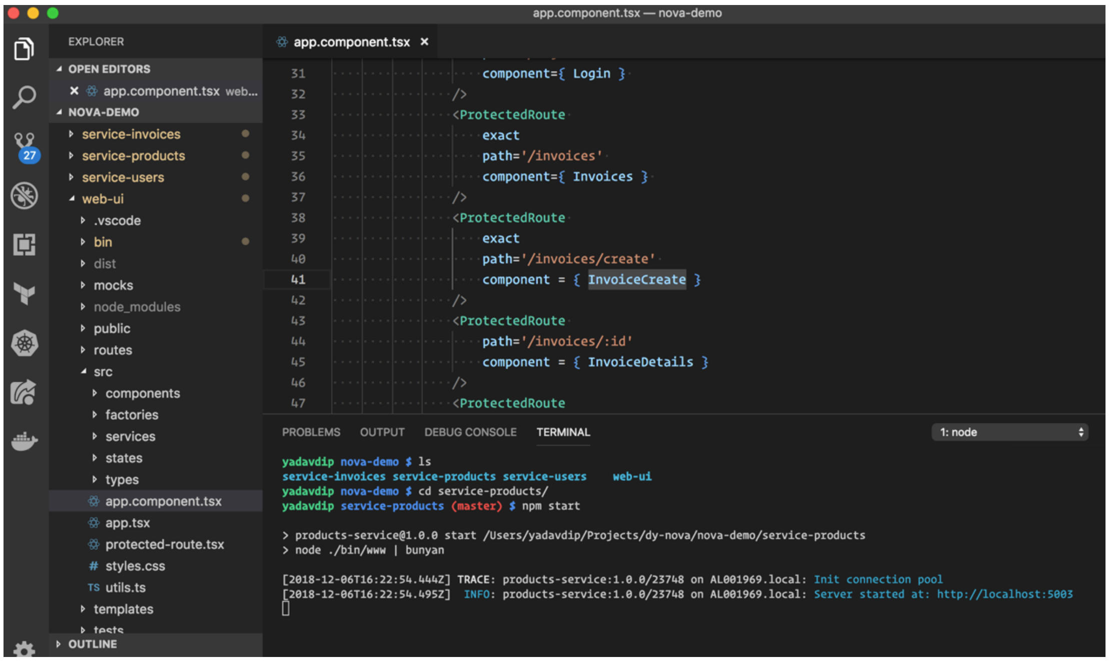
```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');

var indexRouter = require('./routes/index');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use((req, res, next) => {
  res.locals.SERVICE_INVOICE = process.env.SERVICE_INVOICE;
  res.locals.SERVICE_USER = process.env.SERVICE_USER;
  res.locals.SERVICE_PRODUCTS = process.env.SERVICE_PRODUCTS;
```

```
npm run start
```

```
8-12-06T16:11:53.177Z] TRACE: products-service:1.0.0/23399 on AL001969.local: Init connection pool
8-12-06T16:11:53.276Z]  INFO: products-service:1.0.0/23399 on AL001969.local: Server started at: http://localhost:5003
```

Figure 5 Web-ui and service-products microservice projects in IntelliJ IDEA

Similarly figure (Figure 6) shows loading web-ui and service-projects in Visual Studio Code. Here we have manually entered the desired directory using integrated terminal and started the desired application. This is not an easy solution as developers have to navigate to root of each project and start each dependent project and resources. Also since, all projects are running in same IDE it creates noise and makes it harder to navigate to correct file if there exists same filename in multiple projects.



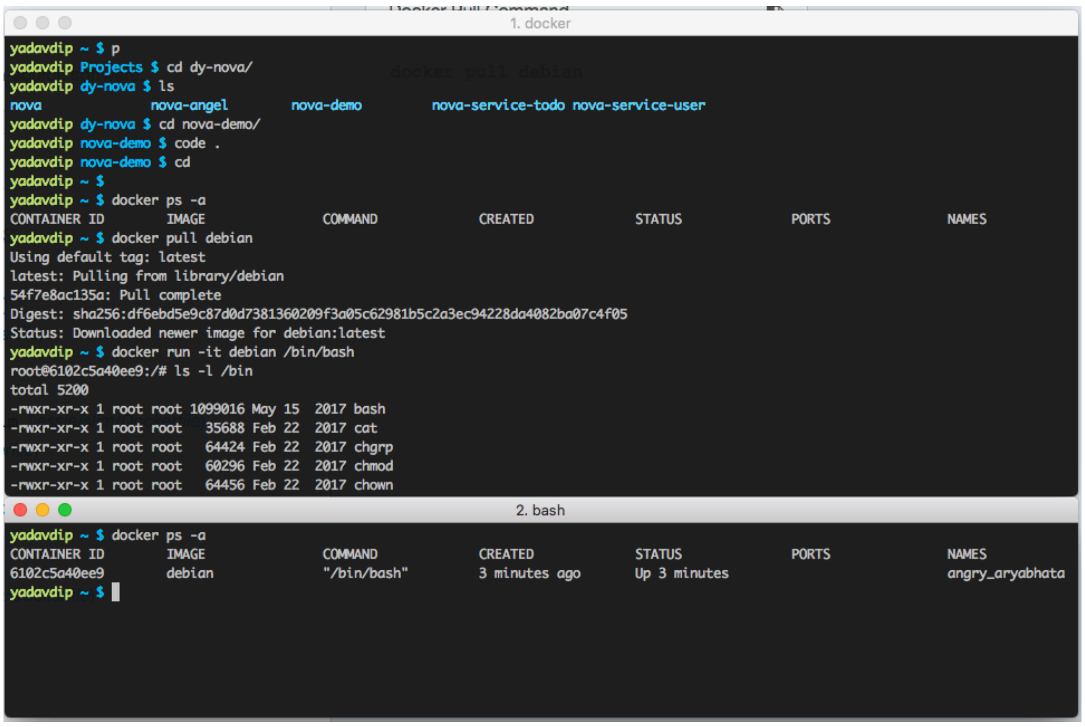
```
                component={ Login }
            />
            <ProtectedRoute
                exact
                path='/invoices'
                component={ Invoices }
            />
            <ProtectedRoute
                exact
                path='/invoices/create'
                component = { InvoiceCreate }
            />
            <ProtectedRoute
                path='/invoices/:id'
                component = { InvoiceDetails }
            />
            <ProtectedRoute
```

```
yadavdip nova-demo $ ls
service-invoices service-products service-users    web-ui
yadavdip nova-demo $ cd service-products/
yadavdip service-products (master) $ npm start

> products-service@1.0.0 start /Users/yadavdip/Projects/dy-nova/nova-demo/service-products
> node ./bin/www | bunyan

[2018-12-06T16:22:54.444Z] TRACE: products-service:1.0.0/23748 on AL001969.local: Init connection pool
[2018-12-06T16:22:54.495Z]  INFO: products-service:1.0.0/23748 on AL001969.local: Server started at: http://localhost:5003
```

Figure 6 Web-ui and service-products microservice projects in Visual Studio Code

2.2 Containerization with Docker

Docker (Docker, 2018) is the most popular tool for application containerization. As shown figure below (Figure 7) we can easily start-up Docker container by pulling its image from Docker repository ([https://hub.docker.com/)](https://hub.docker.com/) and starting it with Docker run command. Lower terminal in figure (Figure 7) shows Debian linux is up and running in container.

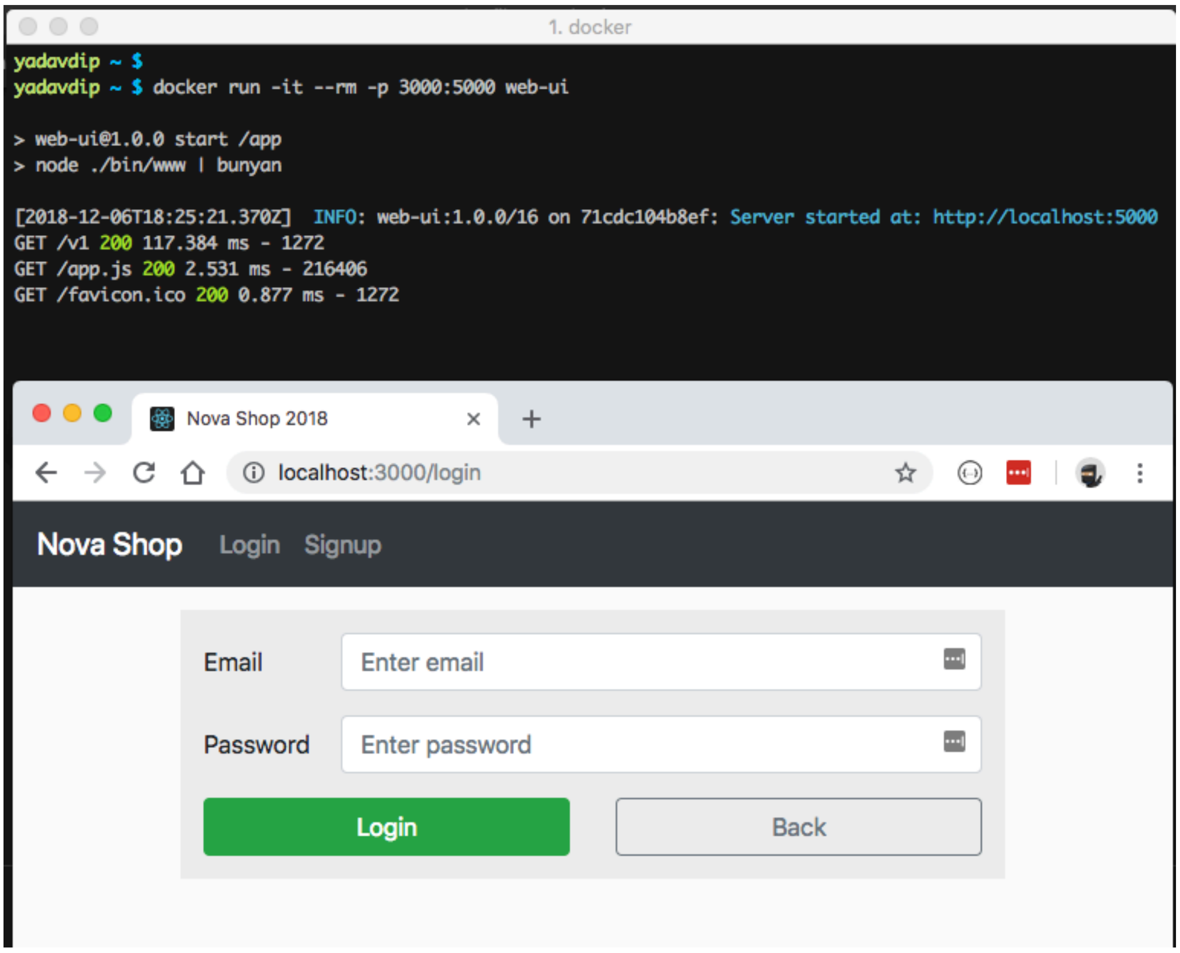


```
yadavdip ~ $ p
yadavdip Projects $ cd dy-nova/
yadavdip dy-nova $ ls
nova                nova-angel          nova-demo           nova-service-todo nova-service-user
yadavdip dy-nova $ cd nova-demo/
yadavdip nova-demo $ code .
yadavdip nova-demo $ cd
yadavdip ~ $
yadavdip ~ $ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
yadavdip ~ $ docker pull debian
Using default tag: latest
latest: Pulling from library/debian
54f7e8ac135a: Pull complete
Digest: sha256:df6ebd5e9c87d0d7381360209f3a05c62981b5c2a3ec94228da4082ba07c4f05
Status: Downloaded newer image for debian:latest
yadavdip ~ $ docker run -it debian /bin/bash
root@6102c5a40ee9:/# ls -l /bin
total 5200
-rwxr-xr-x 1 root root 1099016 May 15  2017 bash
-rwxr-xr-x 1 root root   35688 Feb 22  2017 cat
-rwxr-xr-x 1 root root   64424 Feb 22  2017 chgrp
-rwxr-xr-x 1 root root   60296 Feb 22  2017 chmod
-rwxr-xr-x 1 root root   64456 Feb 22  2017 chown
```

```
yadavdip ~ $ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
6102c5a40ee9        debian              "/bin/bash"         3 minutes ago       Up 3 minutes                            angry_aryabhata
yadavdip ~ $
```

Figure 7 Containerized Debian OS running in docker

In another example (Figure 8) we started web-ui containerized application with Docker. Here inside Docker application starts at port 5000 but while running we told Docker to expose this application in port 3000. So, we can now view the application in http://localhost:3000.

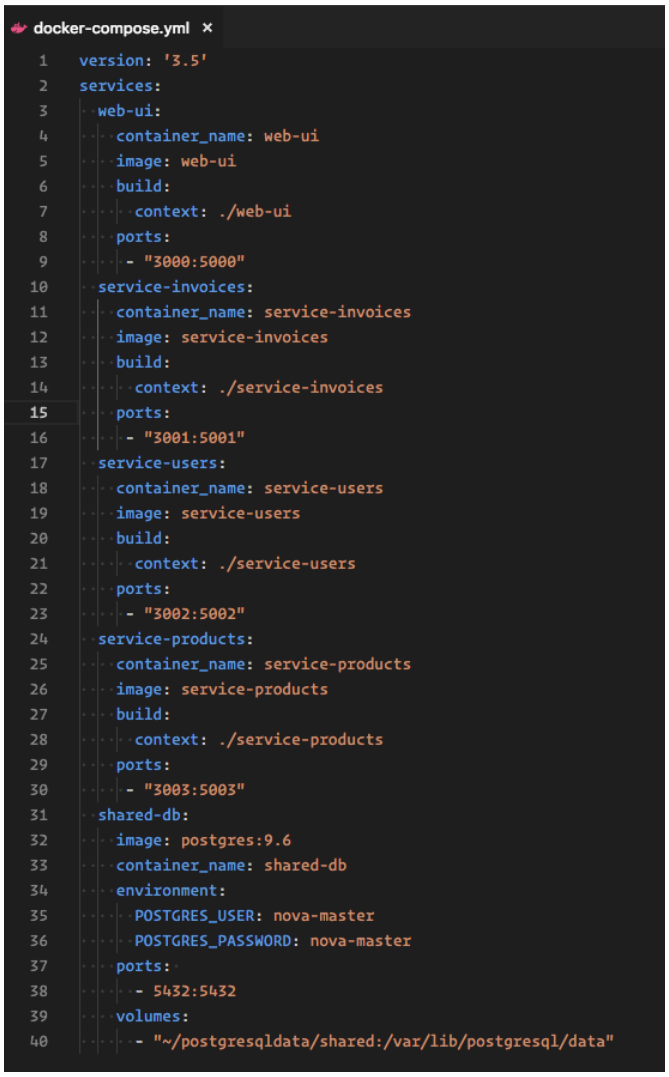


```
yadavdip ~ $
yadavdip ~ $ docker run -it --rm -p 3000:5000 web-ui

> web-ui@1.0.0 start /app
> node ./bin/www | bunyan

[2018-12-06T18:25:21.370Z]  INFO: web-ui:1.0.0/16 on 71cdc104b8ef: Server started at: http://localhost:5000
GET /v1 200 117.384 ms - 1272
GET /app.js 200 2.531 ms - 216406
GET /favicon.ico 200 0.877 ms - 1272
```

Figure 8 Containerized application running in docker

2.3 Docker compose

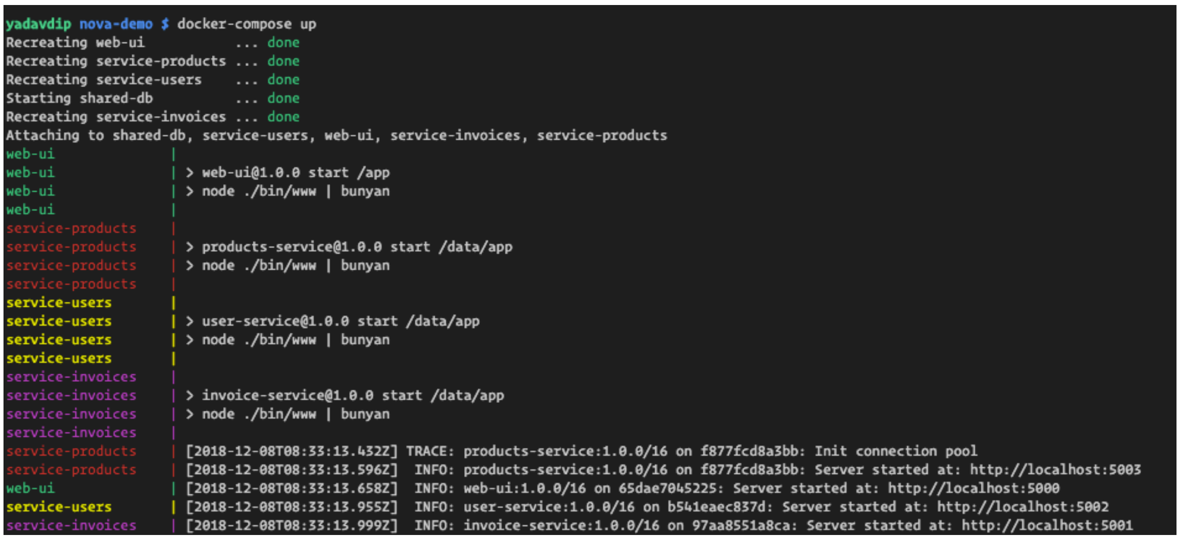
Docker compose is command line tool from Docker which enables us to start-up multiple containers with one command. Docker compose uses manually created compose file (Figure 9) to start up services using command line. As shown in figure (Figure 10), developer can start up all services with a single command. Also, all logs from running services can be seen in on location which makes debugging easier.



docker-compose.yml

```yaml
version: '3.5'
services:
  web-ui:
    container_name: web-ui
    image: web-ui
    build:
      context: ./web-ui
    ports:
     - "3000:5000"
  service-invoices:
    container_name: service-invoices
    image: service-invoices
    build:
      context: ./service-invoices
    ports:
     - "3001:5001"
  service-users:
    container_name: service-users
    image: service-users
    build:
      context: ./service-users
    ports:
     - "3002:5002"
  service-products:
    container_name: service-products
    image: service-products
    build:
      context: ./service-products
    ports:
     - "3003:5003"
  shared-db:
    image: postgres:9.6
    container_name: shared-db
    environment:
      POSTGRES_USER: nova-master
      POSTGRES_PASSWORD: nova-master
    ports:
      - 5432:5432
    volumes:
      - "~/postgresqldata/shared:/var/lib/postgresql/data"
```

Figure 9 Docker compose file

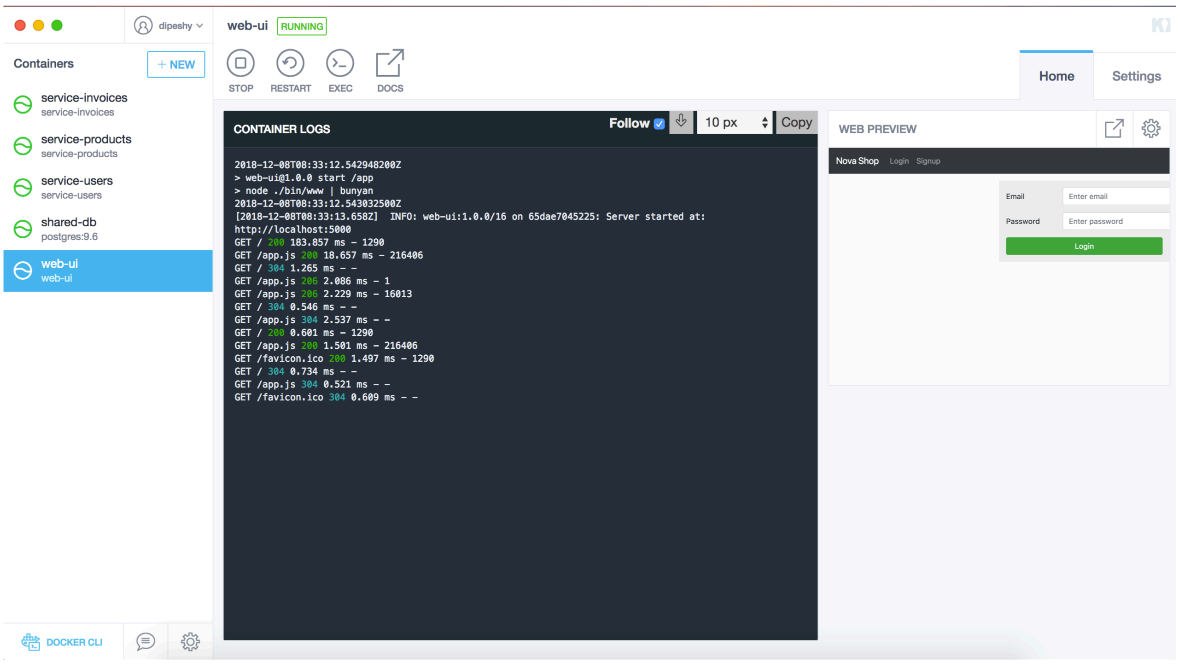


```
yadavdip nova-demo $ docker-compose up
Recreating web-ui           ... done
Recreating service-products ... done
Recreating service-users    ... done
Starting shared-db          ... done
Recreating service-invoices ... done
Attaching to shared-db, service-users, web-ui, service-invoices, service-products
web-ui              |
web-ui              | > web-ui@1.0.0 start /app
web-ui              | > node ./bin/www | bunyan
web-ui              |
service-products    |
service-products    | > products-service@1.0.0 start /data/app
service-products    | > node ./bin/www | bunyan
service-products    |
service-users       |
service-users       | > user-service@1.0.0 start /data/app
service-users       | > node ./bin/www | bunyan
service-users       |
service-invoices    |
service-invoices    | > invoice-service@1.0.0 start /data/app
service-invoices    | > node ./bin/www | bunyan
service-invoices    |
service-products    | [2018-12-08T08:33:13.432Z] TRACE: products-service:1.0.0/16 on f877fcd8a3bb: Init connection pool
service-products    | [2018-12-08T08:33:13.596Z]  INFO: products-service:1.0.0/16 on f877fcd8a3bb: Server started at: http://localhost:5003
web-ui              | [2018-12-08T08:33:13.658Z]  INFO: web-ui:1.0.0/16 on 65dae7045225: Server started at: http://localhost:5000
service-users       | [2018-12-08T08:33:13.955Z]  INFO: user-service:1.0.0/16 on b541eaec837d: Server started at: http://localhost:5002
service-invoices    | [2018-12-08T08:33:13.999Z]  INFO: invoice-service:1.0.0/16 on 97aa8551a8ca: Server started at: http://localhost:5001
```

Figure 10 Starting services with docker compose

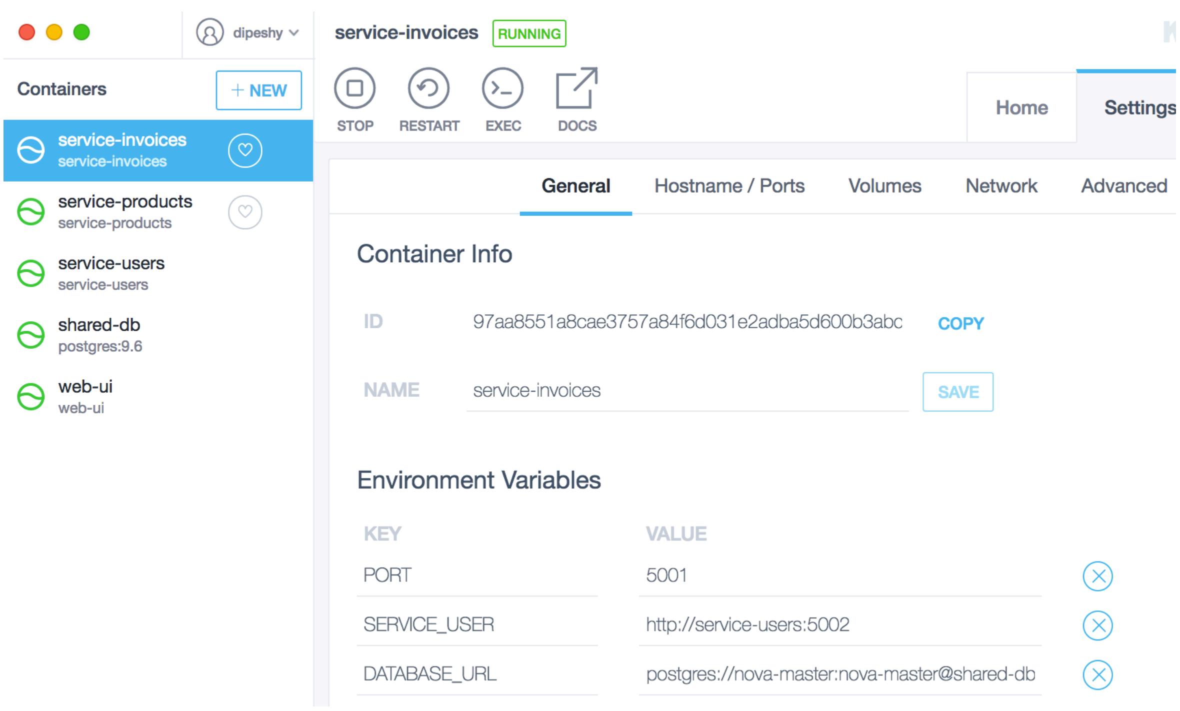
2.4 Kitematic

Another helpful tool from Docker is Kitematic (Kitematic, 2018). It provides GUI to start and stop containers. Shown in figure (Figure 11) we can navigate all containerized services on left sidebar. From here developers can see the logs. This tool also helps developers to customize the application to be run with different configuration as shown in figure (Figure 12).



```
2018-12-08T08:33:12.542948200Z
> web-ui@1.0.0 start /app
> node ./bin/www | bunyan
2018-12-08T08:33:12.543032500Z
[2018-12-08T08:33:13.658Z]  INFO: web-ui:1.0.0/16 on 65dae7045225: Server started at:
http://localhost:5000
GET / 200 183.857 ms - 1290
GET /app.js 200 18.657 ms - 216406
GET / 304 1.265 ms - -
GET /app.js 206 2.086 ms - 1
GET /app.js 206 2.229 ms - 16013
GET / 304 0.546 ms - -
GET /app.js 304 2.537 ms - -
GET / 200 0.601 ms - 1290
GET /app.js 200 1.501 ms - 216406
GET /favicon.ico 200 1.497 ms - 1290
GET / 304 0.734 ms - -
GET /app.js 304 0.521 ms - -
GET /favicon.ico 304 0.609 ms - -
```

Figure 11 Running containers with Kitematic



Container Info

ID	97aa8551a8cae3757a84f6d031e2adba5d600b3abc	COPY
NAME	service-invoices	SAVE

Environment Variables

KEY	VALUE
PORT	5001
SERVICE_USER	http://service-users:5002
DATABASE_URL	postgres://nova-master:nova-master@shared-db

Figure 12 Container configuration in Kitematic

2.5 Telepresence

The Telepresence (Telepresence, 2018) is a tool for running single service locally while being connected to other services is remote cluster. Developer can pick target service from remote cluster and use respective telepresence command to replace the remote communication with local ones. Figure (Figure 13a) shows a remote cluster running three microservices. When developer makes connection with telepresence it replaces the target microservice in this case Invoices with container which forwards the communication to the local machine. As show in figure (Figure 13b) developer call the communication line to cluster invoices is forwarded to local laptop.

Telepresence is a very helpful tool and similar in approach to the target thesis tool. With Telepresence developer can develop and debug microservices using production like environment. However, it runs only cluster managed by Kubernetes and requires a bit of setup to make it work. In most cases for security reasons it might not be possible to tunnel a connection to local machine in cloud clusters. Also it uses cli which developers require to learn and memorize.

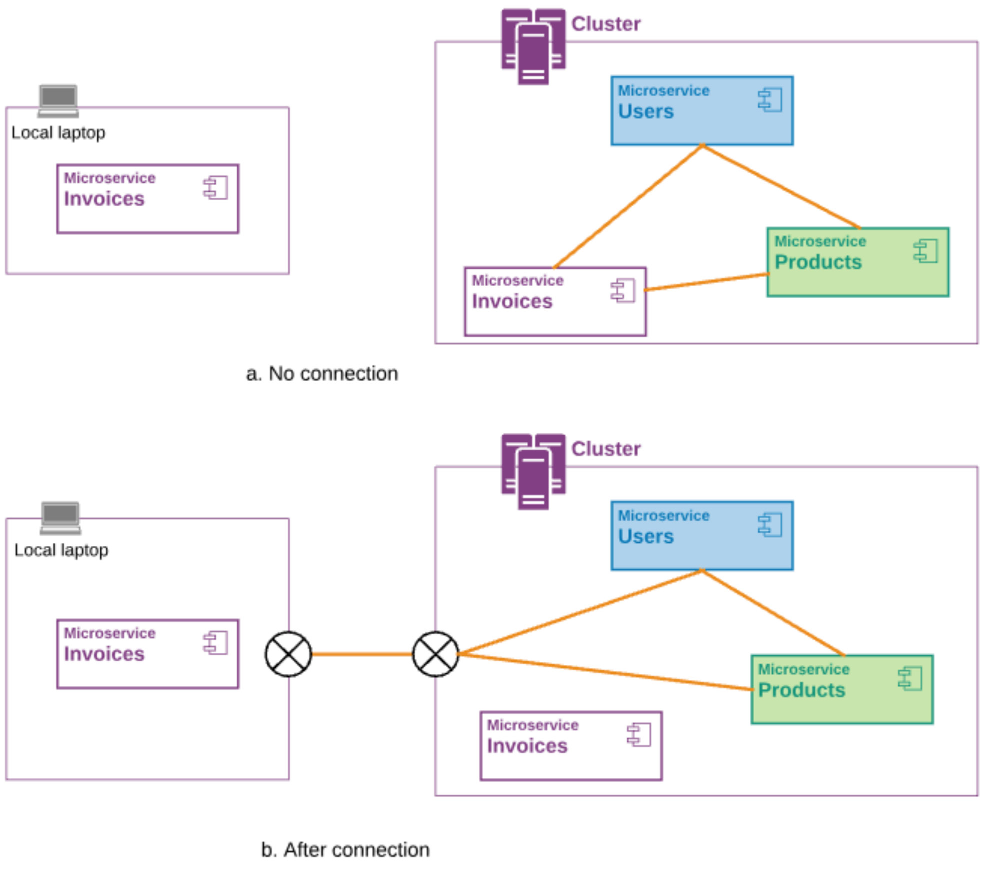


Figure 13 Development with Telepresence

3 Overview of demo application

For the purpose of demonstrating the use case of this tool, a demo e-commerce web application is created. In this application user can purchase digital movies. Each movie purchase will generate invoice for the user which has to be paid within seven days.

Here are the basic features of the application:

1. Member user can login in using credentials.
2. Member user can view available movies
3. Member user can purchase movies.
4. Member user can view purchased movies' invoices

Operational requirements of this application are:

1. Application should be able to deploy efficiently. The goal is to deploy completed features independently as fast as possible in a reliable way. This gives possibility for rapid testing new business concepts without having to wait months or years.

2. Application should be highly available. Application should be constantly serving users 24/7 with zero maintenance downtime.

4 Monolith architecture

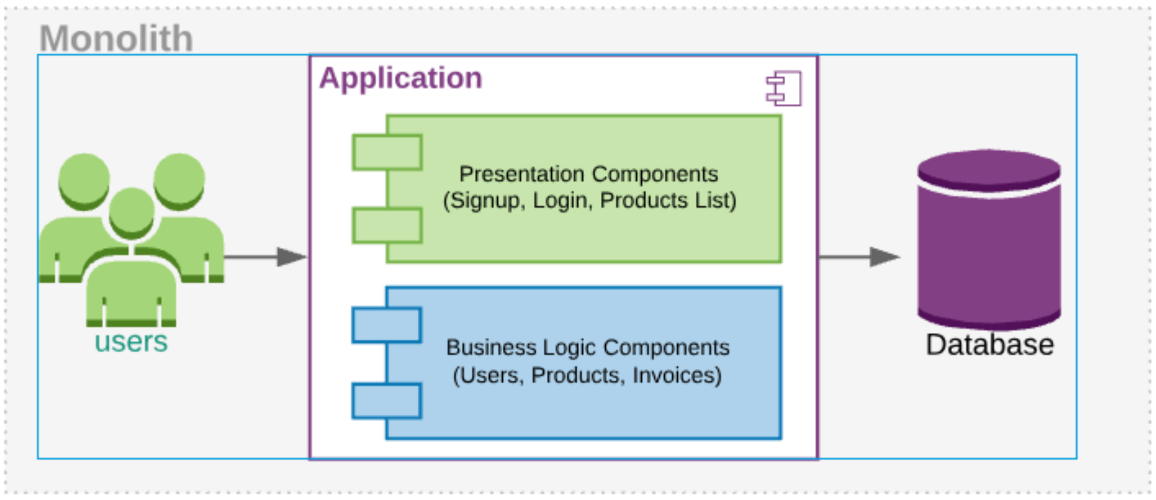

Monolith
Application
Presentation Components
(Signup, Login, Products List)
Business Logic Components
(Users, Products, Invoices)
users
Database

Figure 14 Monolith architecture of demo application

If the demo application were to be designed in monolith architecture, it would be as presented in figure above (Figure 14). This single application consists of a presentation components and business logic components.

As shown in figure (Figure 14) the upper component in application is presentation components responsible for encapsulating UI components. This UI component is responsible for returning html, css, and javascript to client's browser. The html page contains and presents data provided by the business logic components. For example, products listing UI component would present products data from products business component (Fowler, Lewis, 2014).

The lower component represents business components in figure (Figure 14). It provides UI components with respective data with applied business rules. For the demo application these components have features for authenticating users, creating invoices, purchasing products etc. These business components persist required data in the database. In this application all features are implemented, built and deployed as one single application. Hence, the term monolith application.

4.1 Development in a monolith

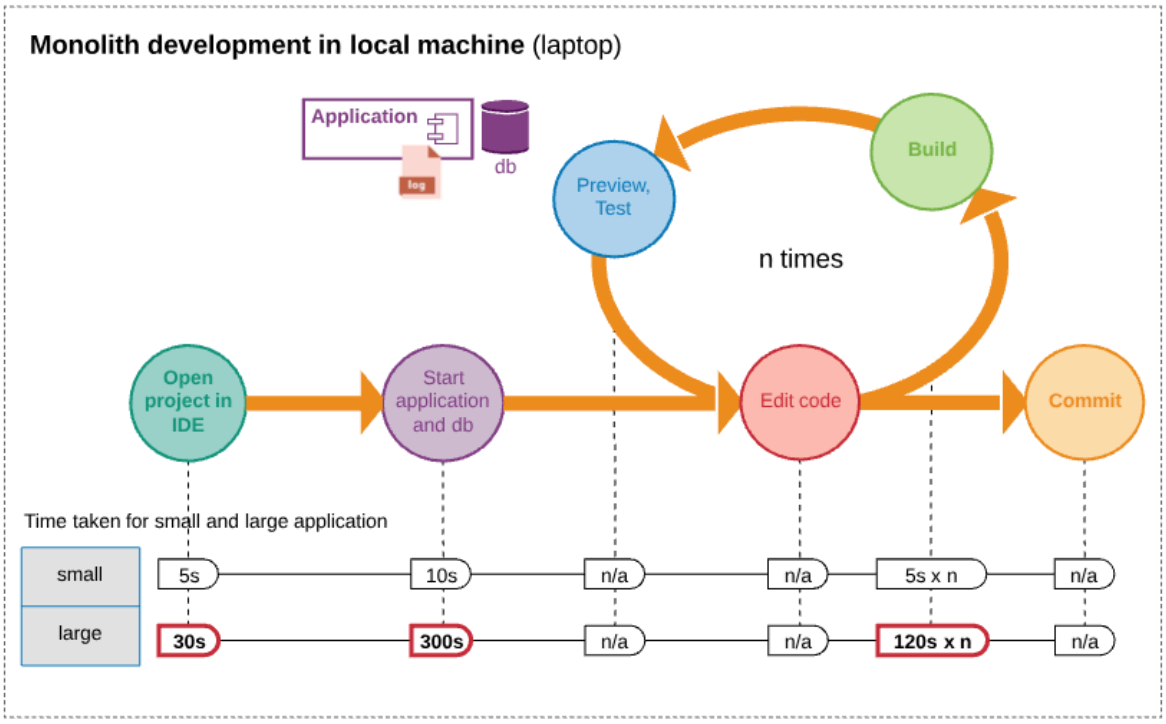


Figure 15 Developing monolith application

Monolith application are quite easy to start. The application does not depend on any other external dependencies except for the database. All the dependent components are implemented in same repository. When new requirements come, the developer can setup the environment and start coding right away. As show in figure above (Figure 15) developer opens the target project in favourite IDE. Then developer then starts the application in the local machine together and also start-up dependent database server. Once the application is running, developer can start editing the code, build the changed code and preview the changes. Once satisfied with code, developer can commit the code and push to the repository.

Referring to time taken for processes in figure above (Figure 15) there is huge difference in time required by some process when the application is large compared to small. These numbers are coming from my own experience in monolith application development. Problematic times are highlighted with thick red border. Large application has huge code base so starting the IDE takes some time to load compared to small application. Also starting the large application requires huge time due to large amount of code needing to be initially built and loaded in development server. In my experience once a large monolith application project took shockingly about 30mins to start-up. Editing is more or less the same in small and large application however, the large code size and folder makes it difficult to navigate and find desired source code easily. After editing, the code requires rebuilding again which is about 1-2 mins again which get quite annoying if one has to do it 100 times. So, while developing in monolith system setup time becomes very slow when application size is large. Monolith application architecture half of the time is consumed waiting for application to be ready to be develop

One important facility that monolith provides is the central logging. There is only one place to check for logs when eases in debugging and monitoring application while developing.

4.2 Development team

In Monolith all team members work on the singe application. Team can be composed of up to 10 – 20 developers. Some, project do divide them by business context. However, they require tremendous planning and to coordination to avoid conflicts. Sometime, merging and resolving code conflict can even take up to 1 day. It becomes quite hard to manage this amount of people for single project and often the project is confusing and chaotic.

4.3 Deploying a monolith

When the application is well tested and ready application is deployed to production servers. Simplest process of deploying is replacing existing code with new codes. Traditional approach of deploying is to use git repository. Where new targeted deployment commit/tag is pulled in production server. Then the repository is built and started up. The process of deployment can take up to 5 hours for large application. During the deployment time in most common scenario the application is unavailable for the end users. Have a service unavailable for long time is not profitable. Also when application fails due to some error, whole application needs to be rolled back to previous version which prolongs the process. So, deployment tends to be expensive process for the business which why deployment is done seldom.

On top of that in monolith even the tiniest update or bug fix results in deploying the whole application. And deployment can be done only after rigorously testing all critical parts of application. This makes mistakes very time consuming and costly. Deploying in monolith is hard and painful process and usually there is system administration which is specialised in deploying of new versions without hiccups.

4.4 Scaling a monolith application

Vertical scaling and horizontal scaling are two ways to scale an application. Common approach scaling monolith application is vertical scaling. Here the running production server is made more powerful by upgrading its hardware. After that application gets more resource to accommodate large amount of users as shown in figure (Figure 16a) below. Horizontal scaling can also be done where more servers are added and application is replicated to the newly added servers. Here, users are split and directed to different applications to balance the load as shown in figure (Figure 16b).

Deploying monolith can be very expensive. In case of scaling vertically it is very expensive to upgrade hardware and costs more than buying a new server of same low capacity. Horizontal scaling is more appropriate as it is also fault tolerant. In case one of the server goes down another server is able to handle the traffic to make it available.

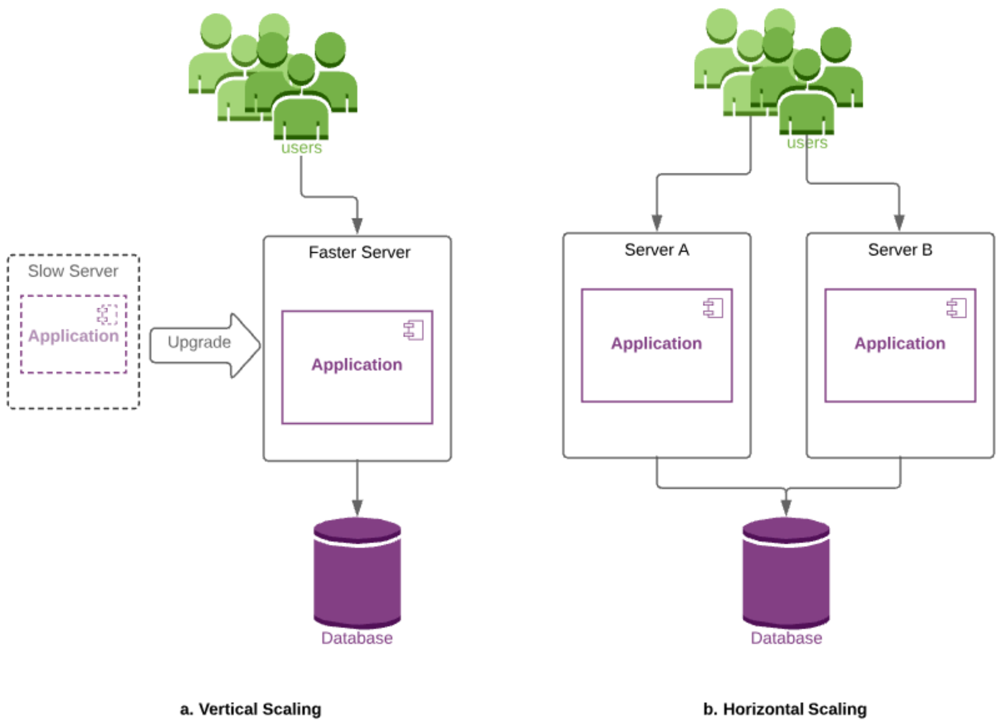


Figure 16 Scaling monolith application

Monolith is status quo architecture for developing and deploying application. Development and deploying is easier in beginning. However, as time goes on and new features are added application gets harder and harder to implement. Event tiny bit of change may end up taking days. As in a large codebase it is difficult to integrate new feature. Adding one line of code change may introduce bugs in our application which makes testing long time as whole application may need to be tested. Development ends of taking long time due to this issue and deployment becomes very seldom. Most large application is able to deploy only 4 – 6 times a month as many features need to be bundles up together as single deployment package. Due to this reason mistakes and experimentation in monolith is extremely expensive operation.

5 Microservice architecture

Fowler and Lewis (2014), describes the microservice as an architecture of developing a single application as small services built around business capabilities. Microservices is breakdown of a monolithic application to small loosely coupled independent services. The primary design goal of microservice architecture is the freedom to develop and deploy portion of application independently. Besides deploying independently microservices enables to easily scale only requires services. Since they are distributed, failure of one microservice will not bring down the whole application which makes our application highly available. Other benefits include that each microservice team is usually small so there is more decision making power in the team. Since, microservices are completely isolated from one another they can be easily migrated to use cutting edge technologies.

So, basically microservices is breaking of an application to separate tiny applications or services. Hence, the term microservices. These applications or services are wired up and works together in providing the required business service to end users. In simple setup microservices communicates with each other with the help of RESTful endpoints in json.

5.1 Splitting demo monolith to microservices

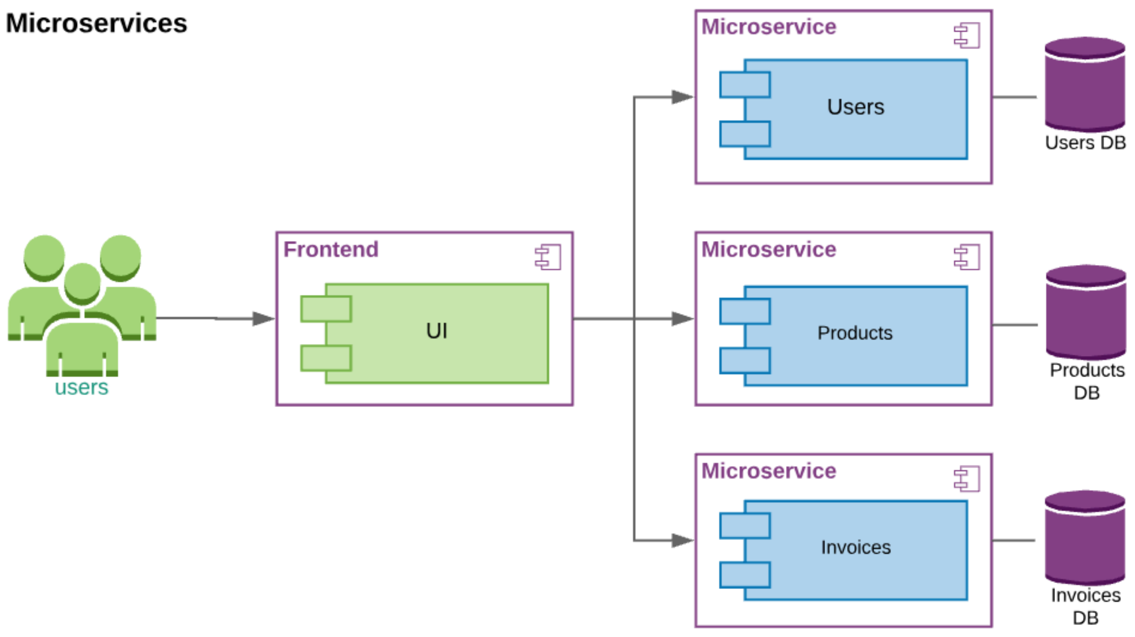


Figure 17 Microservices architecture of demo application

Splitting application to smaller services is completely business dependent. When developing monolith application, is it is advised to organize the code by similar business functions also called bounded business context. This creates all related components to one business function in same place. Usually microservices are broken down by this business functions. In our sample application the monolith presentation layer is separated as Frontend UI microservice application (Figure 17). The remaining business layer functions such as user management, product management and invoices management are separated as user service, products service and invoices service as show in figure (Figure 17). Each microservice provides services and features relating to single business function. Here the microservice is separate tiny application providing small set of related services. Microservices are not limited to breaking by business function. Microservice is about continuously identifying parts of our application that can be separated and deployed independently. Sometimes it also makes sense to merge two microservices to single microservice.

5.2 Development in microservices

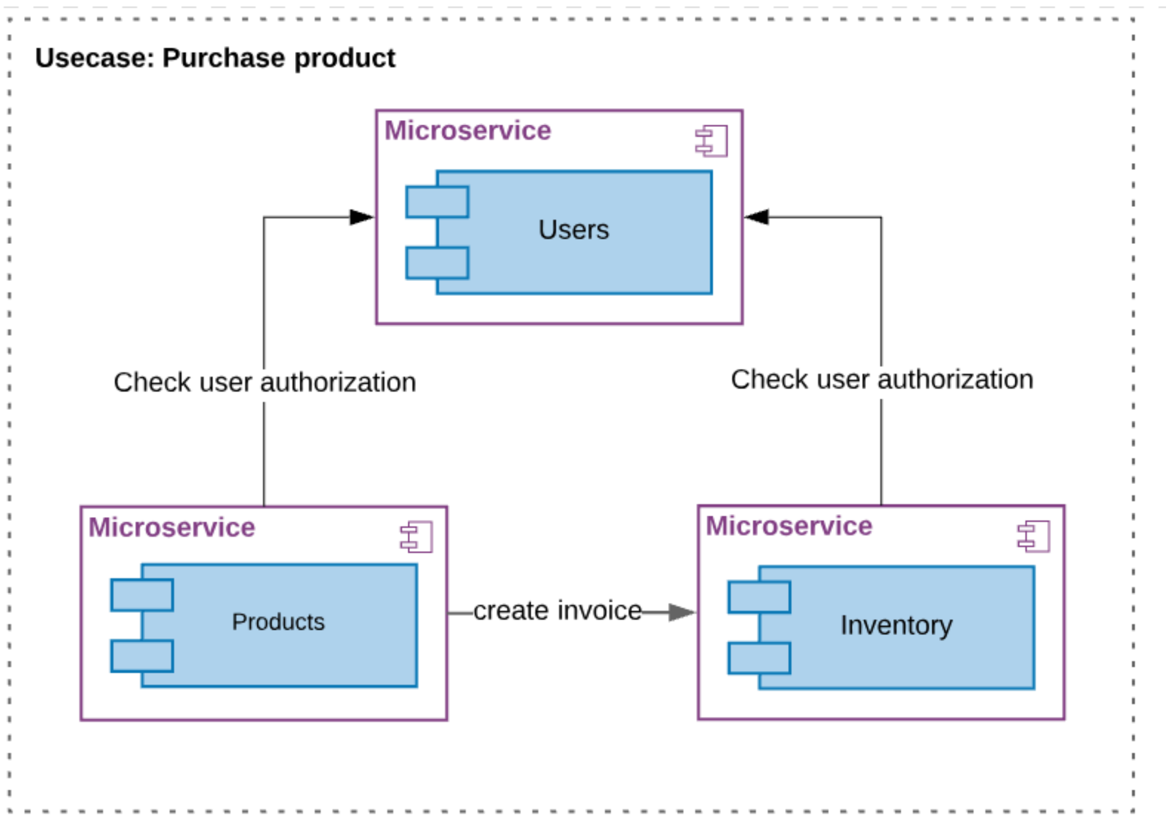


Figure 18 Feature purchasing product

Since, microservice is only part of the bigger application it requires to communicate with other microservices for provide some solutions. For example, as shown in figure above

(Figure 18) in our demo application purchasing a movie product requires products microservice to check if the user is authenticated and authorized to do so. This information is provided by users microservice. Also, our business requirement is such that when user purchases a movie invoice is generated in invoice microservice. This is initiated by products microservice when purchase is successful. Invoices microservice also requires to validate the authenticity and authorization of the user. If one were to implement such feature it is required by the developer to spin up users and invoices microservice so that products microservice can be tested while developing.

Since, each service has its own database which may differ in technology it is required to start up respective databases as well. So, as shown in figure (Figure 18) to develop purchasing movie feature in products microservice, developer needs to spin up 2 more microservices and 3 databases for each microservice.

In contrast to just 1 additional resource in monolith which was the database. This makes setting up development of the target service quite challenging especially if it has more than 1 dependent services. Also, makes it hard to keep track on which service depends on which resources. However, once all dependencies are started running on local machine, the application does not take time to perform rebuilding and reloading while development since the application size and codebase is small. So, this create smooth edit, build and preview cycle.

5.3 Deploying and scaling microservice

Microservices is a distributed architecture. Best practice of deploying microservices is in a cluster of servers namely referred as nodes. Microservice deployed to cluster can be started up in random nodes or nodes with less load. All services are distributed in the cluster to work as a single unit. In figure (Figure 19) we have a cluster running four nodes. This a horizontally scalable system where when need more nodes can be added or removed. Services are distributed throughout the nodes to share the best possible load. One best reason to have such an architecture is to have high availability and fault tolerant. Computer hardware are unreliable and fails now and then. In figure below when one node crashes it can be easily replaced by another node.

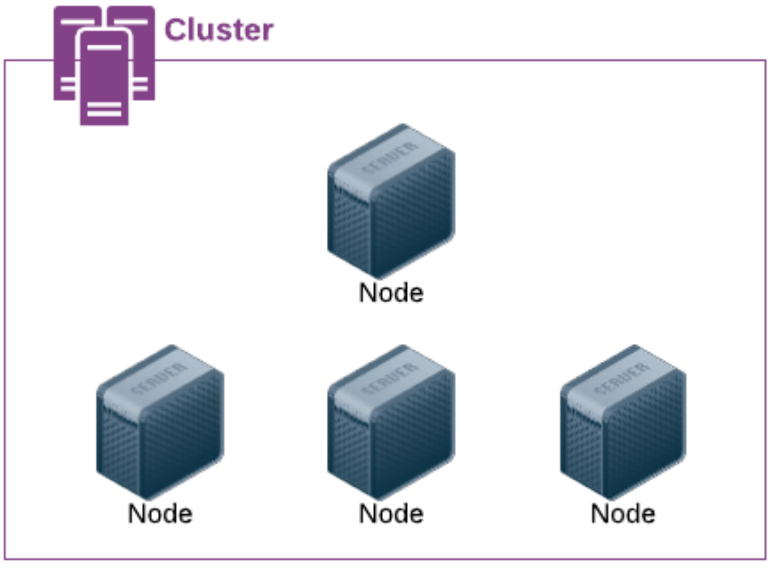


Figure 19 Cluster of nodes

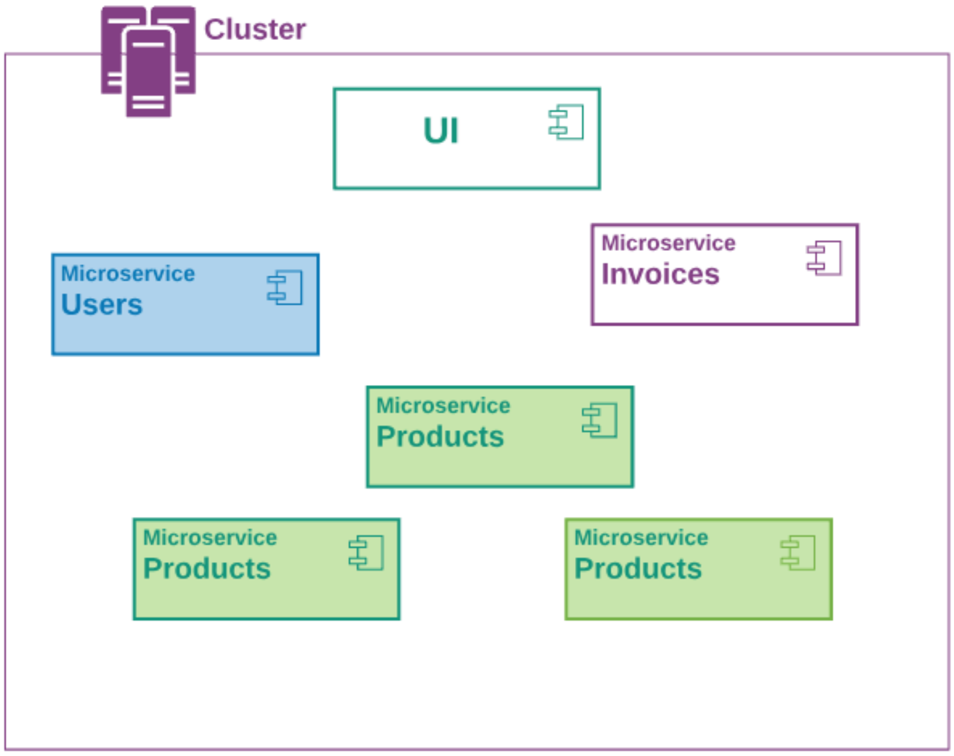


Figure 20 Deployment and scaling microservices in cluster

As show in figure (Figure 20) each microservice is running independently in the cluster. This gives freedom to develop and deploy any service independently.

Comparing to how application was scaled in monolith architecture, figure (Figure 20) shows clearly that not all application is required to be scaled. In microservice architecture service that requires more power can be replicated easy and deployed in cluster. In figure (Figure 20) it was identified that products microservice was unable to handle load. So, it was scaled 3 times to balance the load. When the load is small the scaled nodes can easily be removed. This a really good feature as now our application can easily scale up or down when required to support any amount of traffic. Also the application now consumes only needed resources unlike monolith where resources that did not need to be scaled were forced due to tight coupling. This save resource consumption keeping the operational cost low.

6 Comparison on feature development and deployment

Demand is changing rapidly in current market. Competition is high and businesses are struggling to find competitive edge. Agility of business is on high requirement to keep up with the speed of market change. Businesses need to be capable of experimenting innovative solutions and test market to get competitive edge. Here is a comparison of lead time shown as value process mapping (Figure 21) for a medium size feature between monolith and microservices. This comparison is from my own experience.

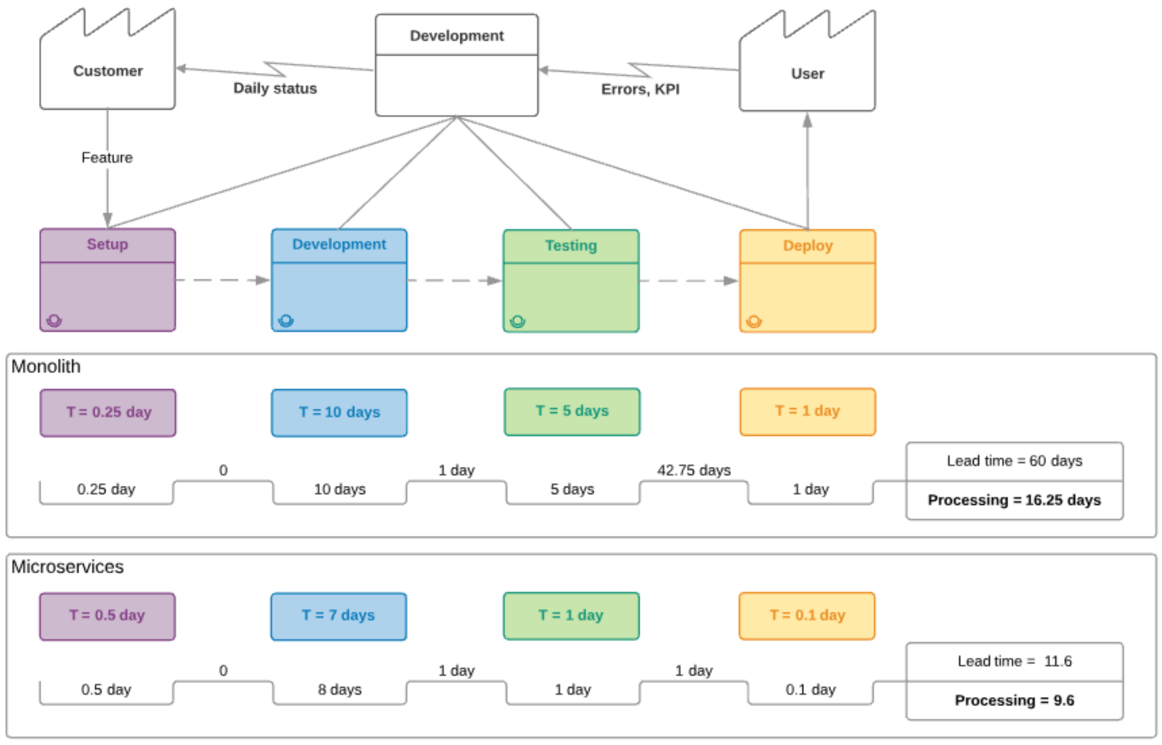


Figure 21 Value stream mapping

Setup

In monolith setup is easy as the developer only needs to start up one service. In microservices setup requires a bit of effort as the developers need to spin of dependent resources. So, microservices requires double the effort than monolith.

Development

In monolith usually code bases are huge and may have technical debt. It takes time to make implementation on large and complex code bases. Developer has to carefully plan and implement the solution to support future needs as well as so not breaking existing features. However, in microservices development is relatively fast as codes are small, clean and well maintained.

Testing

In monolith testing is huge process as it has to go through testing of full application several times to ensure that new feature did not break the application. In microservices only part of the system or microservice that was changed gets tested thoroughly. This saves lot of testing time. In figure (Figure 21) we can see that in monolith waiting time between test to deploy is huge as tested feature needs to wait for the next scheduled deployment. In microservices tested feature can be deployed immediately or within the same day.

Deploy

In monolith it is expensive to deploy single feature at a time. Usually 4 -5 features are bundled together and deployed once every 2 months. So, any feature small or big ends up having lead time of 2 months from concept to launch. This is the biggest advantage microservices has over monoliths. Only desired and changes services are deployed. Also the deployment is small so it takes very short time. Another big advantage here is if the last deployment was failure it is easy to roll back to previous working state without difficulty.

Summary

2 months is very long time to test a concept. And if it the feature does perform and yields negative value instead than removing that feature is another expensive task. This makes the whole project quite expensive. Monolith architecture limits businesses from trying out any new concept as it is expensive and slow. By the time a valuable concept reaches the users, it might be already useless. Microservice architecture on the other hand works on principle of deploying any time. Microservices approach is more based on continuous development of an idea while monolith approach is to make it work the first trail. As imagine continuously development is agile and attractive choice. Also as described in microservices on AWS (2018) agility, innovation, quality, scalability and availability are good reason for choosing microservices.

7 Development complexity in microservice

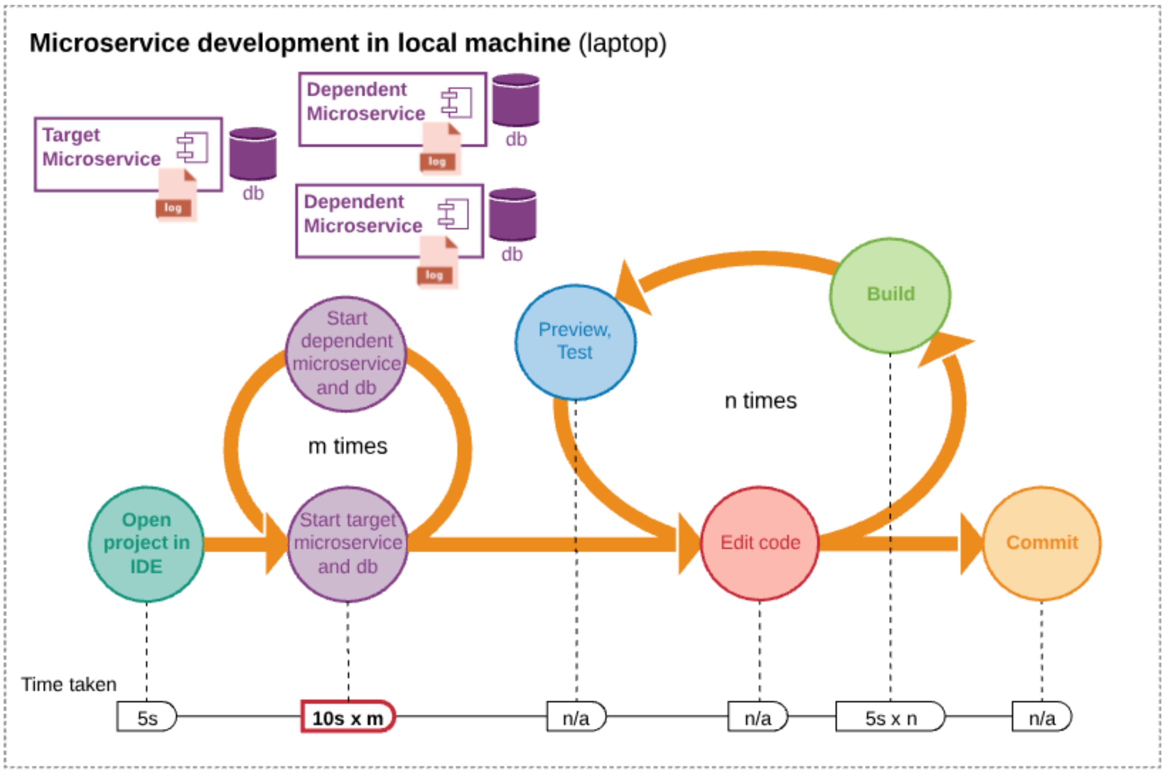


Figure 22 Developing microservices application

Microservices architecture is decoupled and distributed which makes the whole architecture quite complex. Developer works on implementing one microservice at a time, however, most cases the target microservice requires to communicate with other microservices to fulfil its request. Such, implementation can get quite challenging. Also maintaining shared code becomes challenging as they need to be implemented in separate project. (Sharma, 2017) describes some areas that can be shared such as utilities, database apis, logging and common algorithms etc.

As shown in figure above (Figure 22). The target project is opened in IDE quite fast as microservice code size is as large as small monolith application. This makes it easier for developer to navigate the project codes. However, the most cumbersome process is managing and spinning up other microservices that our target microservices communicates with. Although the start time of each microservice is quite small. It takes time to navigate to required project and start up the service manually. Depending on the amount of dependencies it becomes quite annoying and frustrating to start up required microservices each time development needs to be done.

Once all the target and dependent microservices with its respective databases are started the rest is quite easy as reloading and building after code change is fast. However, since each microservices are running independently each microservices will end up having its logs in separate location. This makes debugging through logs quite challenging as now the developer has to cycle through many log windows to trace the data flow. This in not convenient and frustrating when comes to tracing logs while developing.

There are many other strategies as described by Turner-Trauring (2017) which developer choose to use for developing microservices. First strategy is starting up all the services locally as described above using IDE. Others include running microservices in remote cloud and connecting to the remote services. This make things easier as developer don’t need to run everything in the local machine. Turner-Trauring (2017) also mentions a strategy where all services including the one being developed is running remotely. Developer makes changes to source code and pushes the code to remote machine for getting feedback. These approach are very good but most companies cannot give access to remote cloud for security reasons. And also using cloud service as development environment is expensive. So, most cases developer end up running majority or all services in local machine.

The hardest part on getting the work done is the effort it requires to setup the environment to start the work. This is also true for other activities for example learning to play guitar. Each time if one had to take the guitar in and out of the box, it becomes tedious and brain starts to loose motivation and procrastinates. In contrast if the guitar learning environment was already setup and one just had to pick up the guitar and start learning. One and imagine that this became much easier and fun activity.

One important principle in software development is to be lazy and find ways to automate repetitive task. This thesis is such application with facilitate developers to eliminate some of the difficulties while developing in microservices architecture. The goal of this project is to eliminate annoying and frustrating procedures and make development fun and easy. This helps developers do more focused implementation saving time and energy.

8 Running demo microservices in naive way

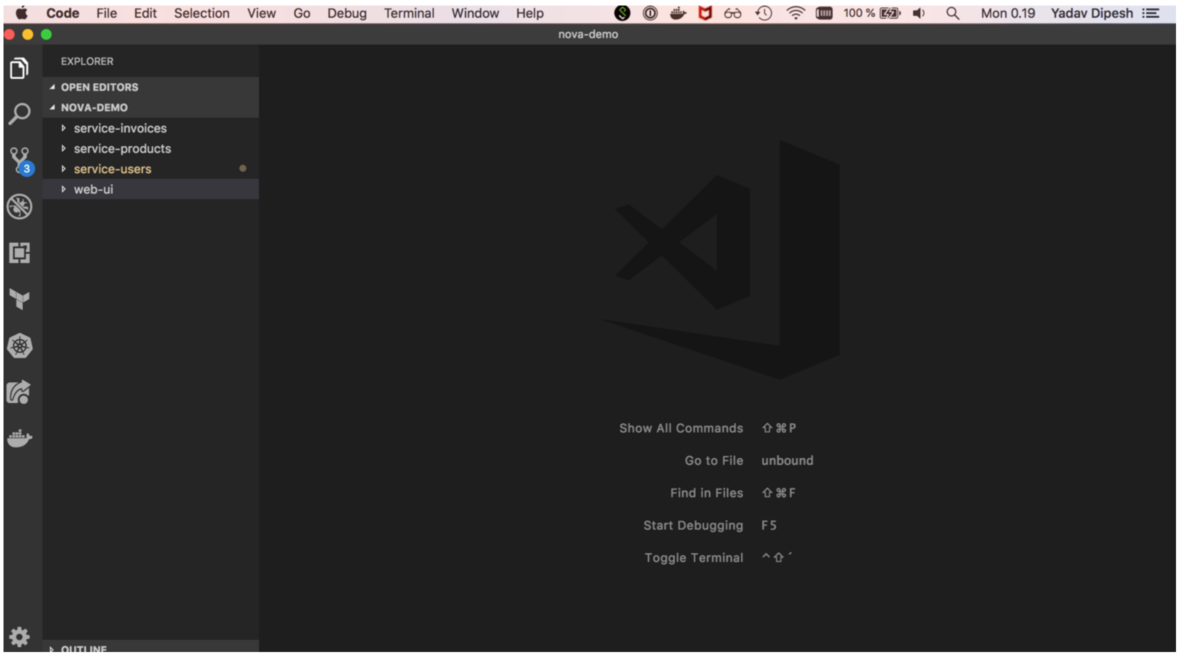


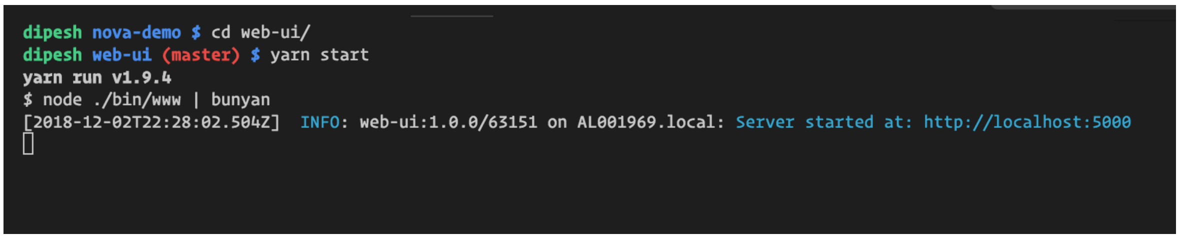
Figure 23 Demo application projects in IDE

As shown in figure (Figure 23) demo application is divided into four separate git repositories. We have on web-ui service which provides our frontend ui application. The frontend web-ui communicates with other three microservices service-invoices, service-products, service-users. All four git project are put under nova-demo folder which is our root project workspace folder. In order to start each application, we need to navigate to microservice working directory and run the appropriate command.

8.1 Required application features

8.1.1 Logging in

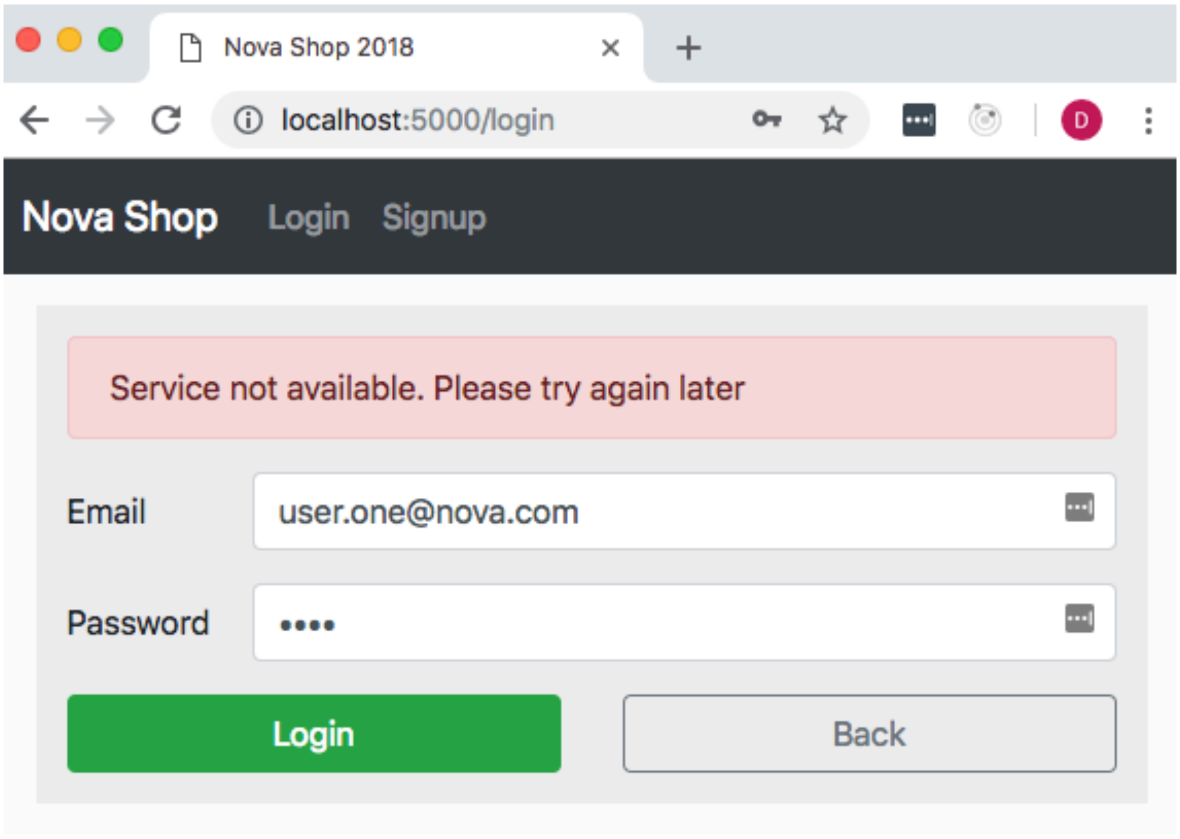
To start web-ui developer need to go the web-ui's root working directory and run the starting command as shown in figure below.



```
dipesh nova-demo $ cd web-ui/
dipesh web-ui (master) $ yarn start
yarn run v1.9.4
$ node ./bin/www | bunyan
[2018-12-02T22:28:02.504Z]  INFO: web-ui:1.0.0/63151 on AL001969.local: Server started at: http://localhost:5000
```

Figure 24 Starting web-ui development server

Now if we navigate to the URL shown in figure (Figure 24) we will see login page. Since the user is not authenticated as shown in figure (Figure 25) and we can sign in with previously created user. This did not work indicated by “Service not available” error message because it could not communicate with users microservice which was not started up.

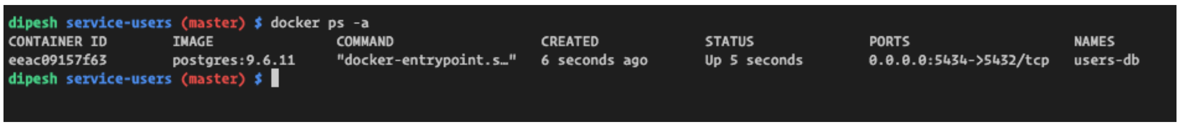


Service not available. Please try again later

user.one@nova.com

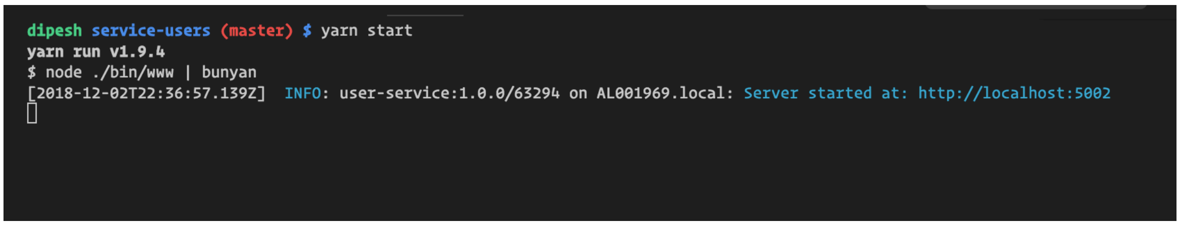
Figure 25 Login page

In order to start our users microservice along with its database. Users-db is started as postgres service in Docker container which is listening on port 5434 as shown in figure (Figure 26). In next figure (Figure 27) we can see that our users microservice is now running on port 5002.



```
dipesh service-users (master) $ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
eeac09157f63        postgres:9.6.11     "docker-entrypoint.s…"   6 seconds ago       Up 5 seconds        0.0.0.0:5434->5432/tcp   users-db
dipesh service-users (master) $ 
```

Figure 26 Docker users-db server



```
dipesh service-users (master) $ yarn start
yarn run v1.9.4
$ node ./bin/www | bunyan
[2018-12-02T22:36:57.139Z]  INFO: user-service:1.0.0/63294 on AL001969.local: Server started at: http://localhost:5002
```

Figure 27 Starting users microservice

Now when we try to login in again after starting users microservice we should be able to get in as show in figure below (Figure 28)

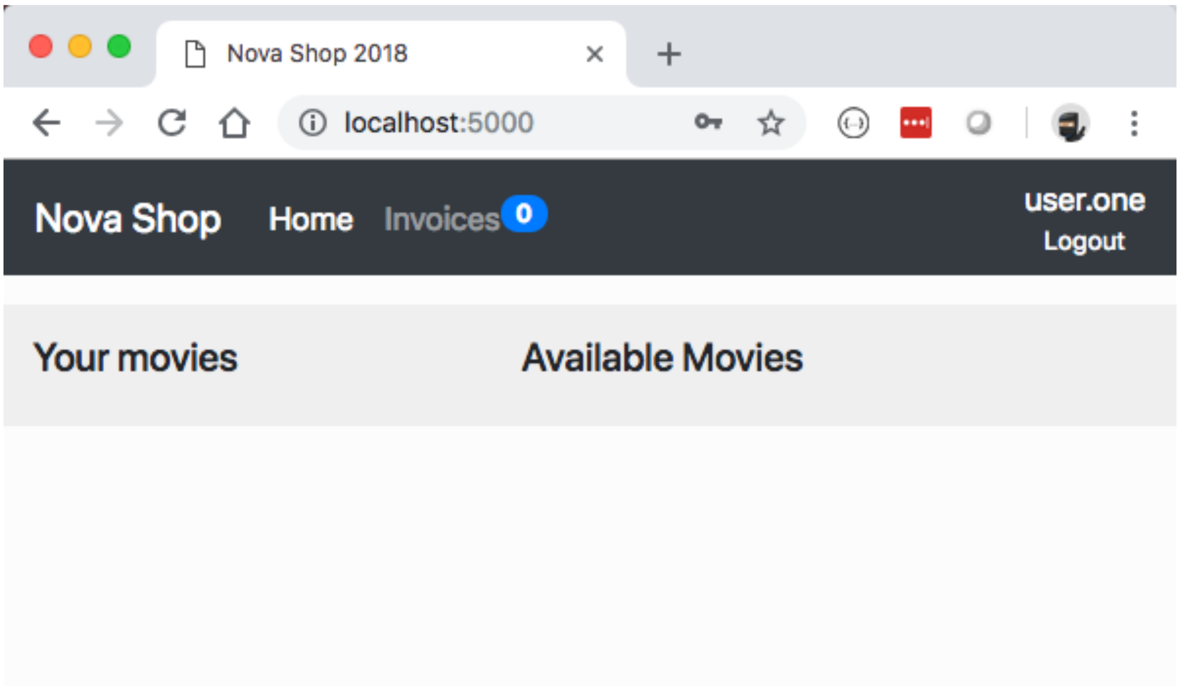
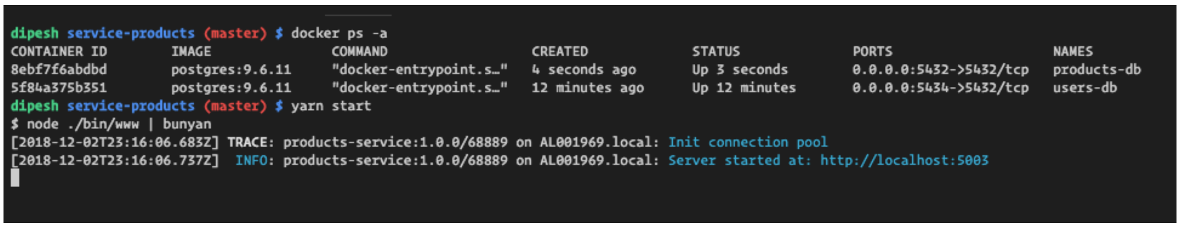


Figure 28 Login successful

8.1.2 View available movie products

In order to get available products, we need to start up products microservice and its database similar to users microservice. In figure (Figure 29), products-db database is started on port 5432 and products microservice started on port 5003.



```
dipesh service-products (master) $ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
8ebf7f6abdbd        postgres:9.6.11     "docker-entrypoint.s…"   4 seconds ago       Up 3 seconds        0.0.0.0:5432->5432/tcp   products-db
5f84a375b351        postgres:9.6.11     "docker-entrypoint.s…"   12 minutes ago      Up 12 minutes       0.0.0.0:5434->5432/tcp   users-db
dipesh service-products (master) $ yarn start
$ node ./bin/www | bunyan
[2018-12-02T23:16:06.683Z] TRACE: products-service:1.0.0/68889 on AL001969.local: Init connection pool
[2018-12-02T23:16:06.737Z]  INFO: products-service:1.0.0/68889 on AL001969.local: Server started at: http://localhost:5003
```

Figure 29 Starting products microservice with database

Now if we reload the page we can see the available products user one can purchase in figure (Figure 30).

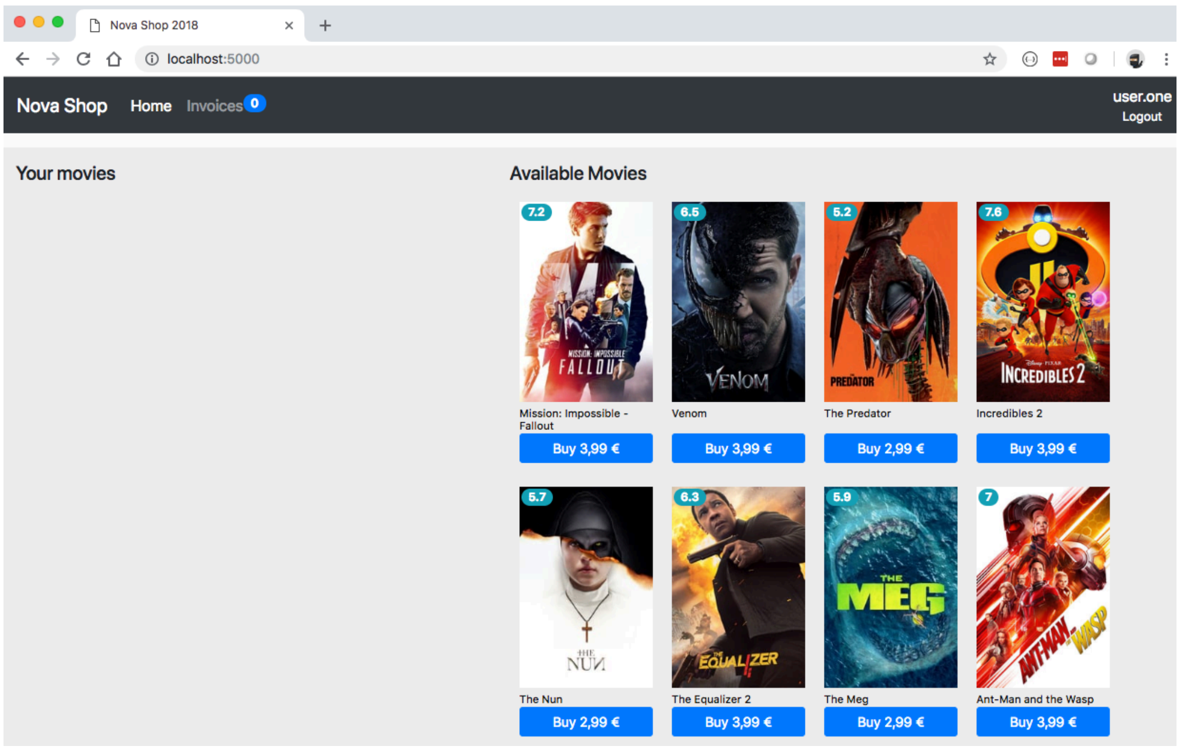
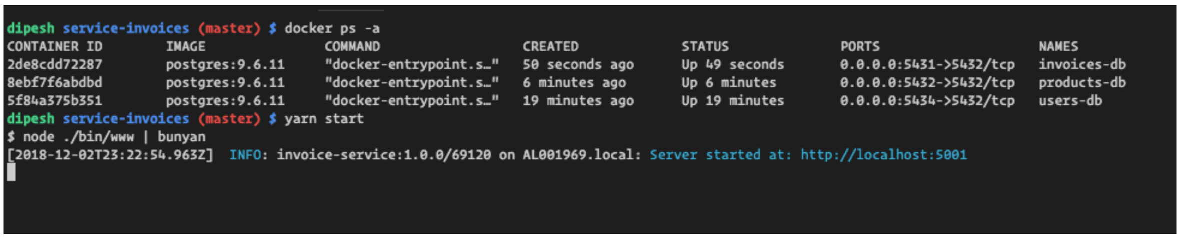


Figure 30 Available products

8.1.3 Purchase movie

In order to purchase movie, we need to enable our invoices microservice with its database. As shown in figure below (Figure 31), invoices-db database is started on port 5431 and invoices microservice started on port 5001.



```
dipesh service-invoices (master) $ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
2de8cdd72287        postgres:9.6.11     "docker-entrypoint.s…"   50 seconds ago      Up 49 seconds       0.0.0.0:5431->5432/tcp   invoices-db
8ebf7f6abdbd        postgres:9.6.11     "docker-entrypoint.s…"   6 minutes ago       Up 6 minutes        0.0.0.0:5432->5432/tcp   products-db
5f84a375b351        postgres:9.6.11     "docker-entrypoint.s…"   19 minutes ago      Up 19 minutes       0.0.0.0:5434->5432/tcp   users-db
dipesh service-invoices (master) $ yarn start
$ node ./bin/www | bunyan
[2018-12-02T23:22:54.963Z]  INFO: invoice-service:1.0.0/69120 on AL001969.local: Server started at: http://localhost:5001
```

Figure 31 Starting invoices microservice with database.

Now when we try to purchase a movie, selected movie should be shown as purchased and one invoices should be created for this purchase as shown in figure below (Figure 32).

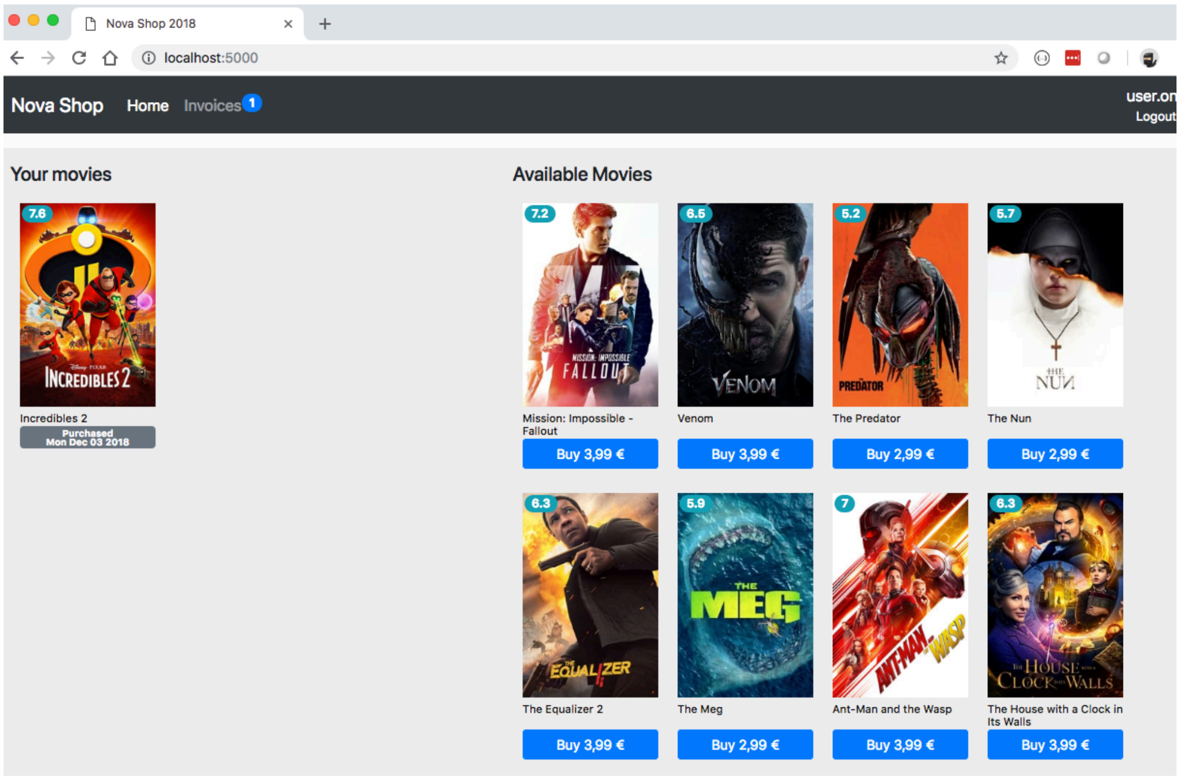
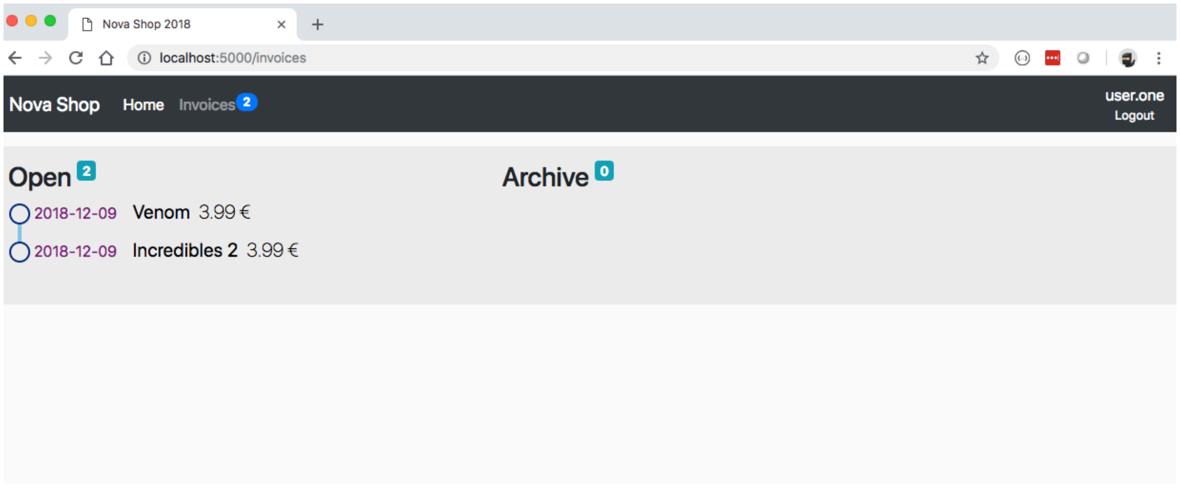


Figure 32 Purchase movie

8.1.4 View invoices

Now that we have our all microservices and database resources required by application to fully function running. After purchasing one more movie we can we that our purchases created respective invoices (Figure 33).



2018-12-09 Venom 3.99 €
2018-12-09 Incredibles 2 3.99 €

Figure 33 View invoices of purchased movies

8.2 Development problems with naive implementation strategy

8.2.1 Difficulty in starting the whole application

As seen in steps performed to start up microservices (8.1) it took quite an effort to start up the whole application since each microservices needed to be started independently.

8.2.2 Difficult to navigate source code

As shown in figure below (Figure 34) when searching app, we are presented with app from all projects. This project workspace already as four applications. Adding more applications will create noisy and distracting application development environment.

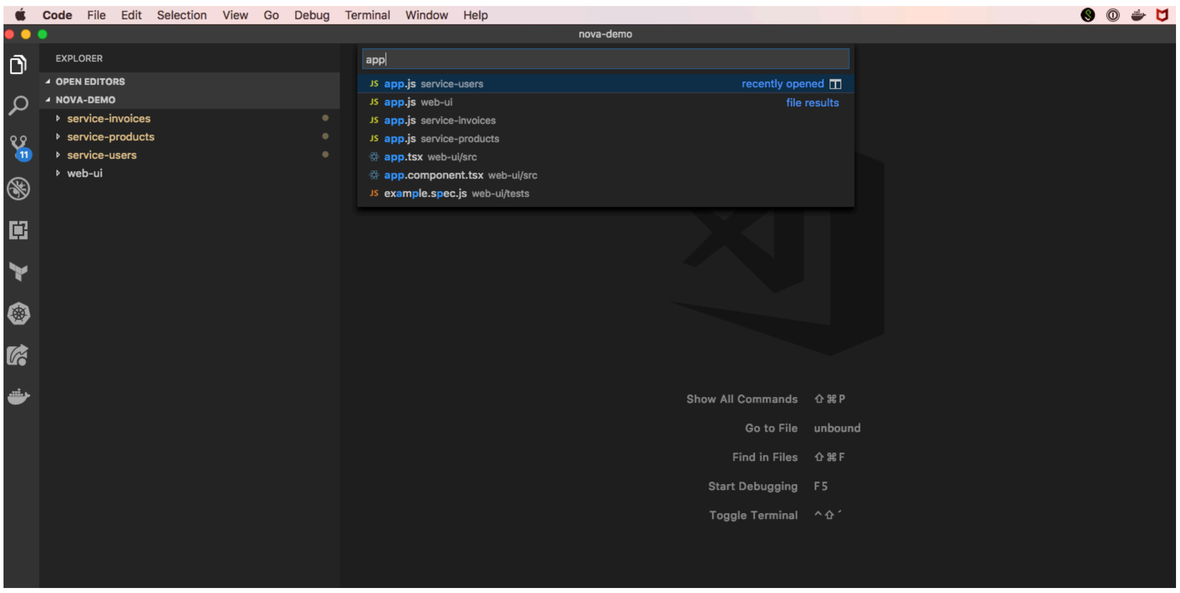
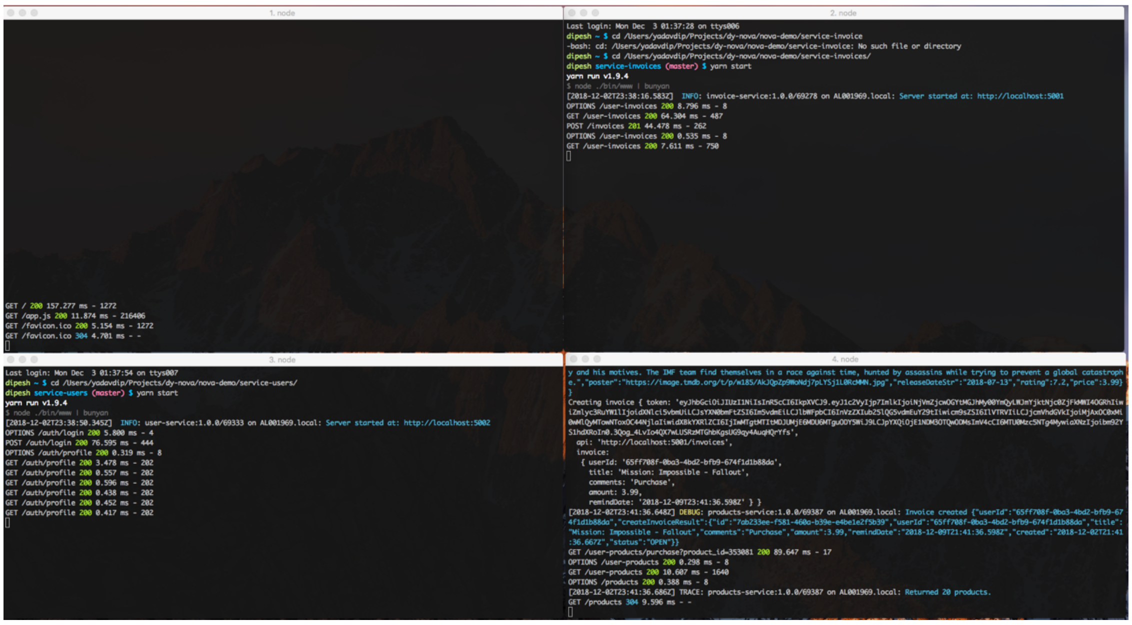


Figure 34 Code navigation

8.2.3 Logging in multiple places

Since, our microservices has to run separately each application had its own logging. So, as shown in figure below (Figure 35) each application has its own logging terminal. This makes tracing requests and errors difficult.



```
1. node

GET / 200 157.277 ms - 1272
GET /app.js 200 11.874 ms - 216406
GET /favicon.ico 200 5.154 ms - 1272
GET /favicon.ico 304 4.701 ms - -
```

```
2. node

Last login: Mon Dec  3 01:37:28 on ttys006
dipesh ~ $ cd /Users/yadavdip/Projects/dy-nova/nova-demo/service-invoice
-bash: cd: /Users/yadavdip/Projects/dy-nova/nova-demo/service-invoice: No such file or directory
dipesh ~ $ cd /Users/yadavdip/Projects/dy-nova/nova-demo/service-invoices/
dipesh service-invoices (master) $ yarn start
yarn run v1.9.4
$ node ./bin/www | bunyan
[2018-12-02T23:38:16.583Z]  INFO: invoice-service:1.0.0/69278 on AL001969.local: Server started at: http://localhost:5001
OPTIONS /user-invoices 200 8.796 ms - 8
GET /user-invoices 200 64.304 ms - 487
POST /invoices 201 44.478 ms - 262
OPTIONS /user-invoices 200 0.535 ms - 8
GET /user-invoices 200 7.611 ms - 750
```

```
3. node

Last login: Mon Dec  3 01:37:54 on ttys007
dipesh ~ $ cd /Users/yadavdip/Projects/dy-nova/nova-demo/service-users/
dipesh service-users (master) $ yarn start
yarn run v1.9.4
$ node ./bin/www | bunyan
[2018-12-02T23:38:50.345Z]  INFO: user-service:1.0.0/69333 on AL001969.local: Server started at: http://localhost:5002
OPTIONS /auth/login 200 5.800 ms - 4
POST /auth/login 200 76.595 ms - 444
OPTIONS /auth/profile 200 0.319 ms - 8
GET /auth/profile 200 3.478 ms - 202
GET /auth/profile 200 0.557 ms - 202
GET /auth/profile 200 0.596 ms - 202
GET /auth/profile 200 0.438 ms - 202
GET /auth/profile 200 0.452 ms - 202
GET /auth/profile 200 0.417 ms - 202
```

```
4. node

y and his motives. The IMF team find themselves in a race against time, hunted by assassins while trying to prevent a global catastroph
e.","poster":"https://image.tmdb.org/t/p/w185/AkJQpZp9WoNdj7pLYSj1L0RcMMN.jpg","releaseDateStr":"2018-07-13","rating":7.2,"price":3.99}
}
Creating invoice { token: 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjp7ImlkIjoiNjVmZjcwOGYtMGJhMy00YmQyLWJmYjktNjc0ZjFkMWI4OGRhIiw
iZmlyc3RuYW1lIjoidXNlci5vbmUiLCJsYXN0bmFtZSI6Im5vdmEiLCJlbWFpbCI6InVzZXIub25lQG5vdmEuY29tIiwicm9sZSI6IlVTRVIiLCJjcmVhdGVkIjoiMjAxOC0xMi
0wMlQyMTowNToxOC44NjlaIiwidXBkYXRlZCI6IjIwMTgtMTItMDJUMjE6MDU6MTguODY5WiJ9LCJpYXQiOjE1NDM3OTQwODMsImV4cCI6MTU0Mzc5NTg4MywiaXNzIjoibm92Y
S1hdXRoIn0.3Qog_4LvIo4QX7wLUSRzMTGhbKgsUG9qy4AuqHQrYfs',
  api: 'http://localhost:5001/invoices',
  invoice:
   { userId: '65ff708f-0ba3-4bd2-bfb9-674f1d1b88da',
     title: 'Mission: Impossible - Fallout',
     comments: 'Purchase',
     amount: 3.99,
     remindDate: '2018-12-09T23:41:36.598Z' } }
[2018-12-02T23:41:36.648Z] DEBUG: products-service:1.0.0/69387 on AL001969.local: Invoice created {"userId":"65ff708f-0ba3-4bd2-bfb9-67
4f1d1b88da","createInvoiceResult":{"id":"7ab233ee-f581-460a-b39e-e4be1e2f5b39","userId":"65ff708f-0ba3-4bd2-bfb9-674f1d1b88da","title":
"Mission: Impossible - Fallout","comments":"Purchase","amount":3.99,"remindDate":"2018-12-09T21:41:36.598Z","created":"2018-12-02T21:41
:36.667Z","status":"OPEN"}}
GET /user-products/purchase?product_id=353081 200 89.647 ms - 17
OPTIONS /user-products 200 0.298 ms - 8
GET /user-products 200 10.607 ms - 1640
OPTIONS /products 200 0.388 ms - 8
[2018-12-02T23:41:36.686Z] TRACE: products-service:1.0.0/69387 on AL001969.local: Returned 20 products.
GET /products 304 9.596 ms - -
```

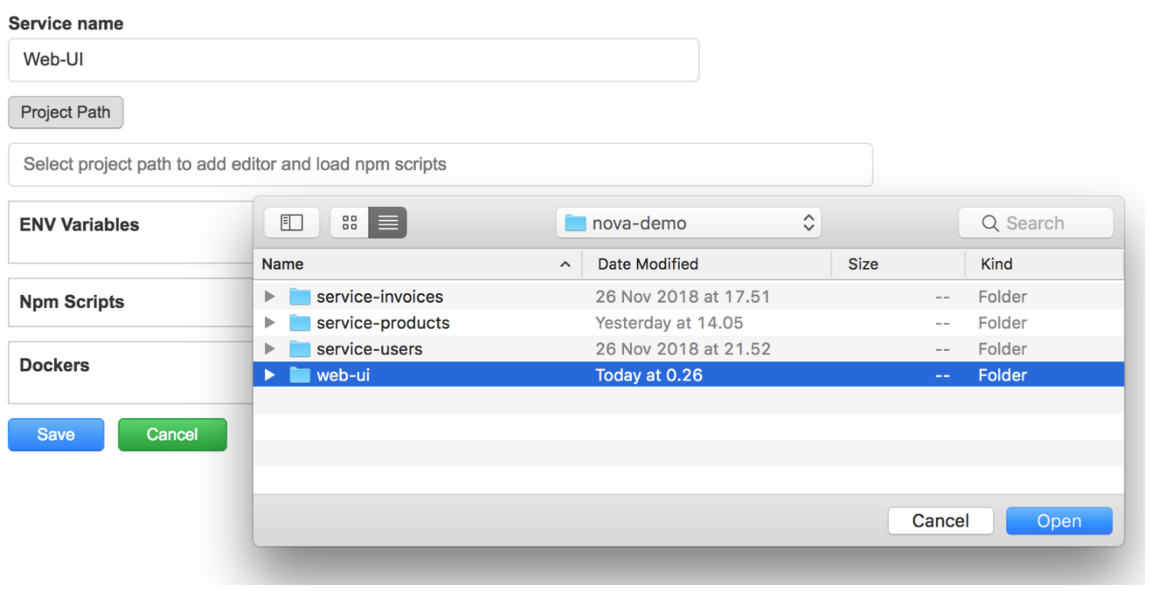
Figure 35 Scattered application logs

9 Solution for increasing microservices productivity

Nova is a GUI tool designed and developed to increase microservices productivity. This is an MIT licensed open source project. The core source code of this tool is available as appendices. Full source code is available at public github repository (https://github.com/dipeshfort/nova-tool). This tool was created with Electron framework which is powered by nodejs.

9.1 Setup frontend application

In order to start using nova for microservice applications we need to configure projects. We can start by setting up Web-UI frontend application. In figure below (Figure 36). We have added the project path by pressing the “Project Path” and selecting target directory.


Service name: Web-UI

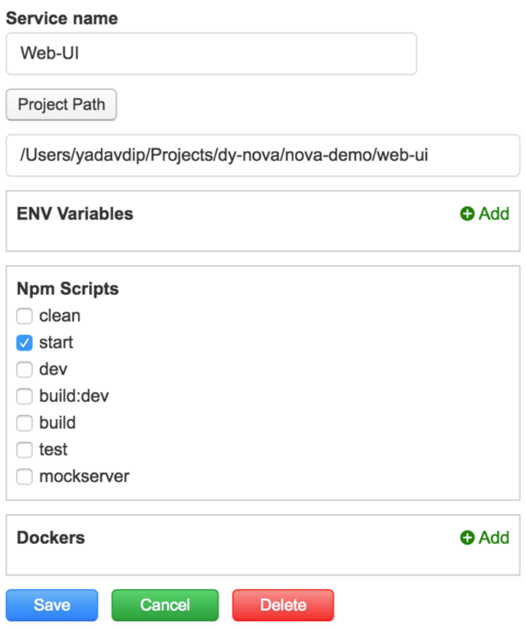
Project Path

Select project path to add editor and load npm scripts

Name	Date Modified	Size	Kind
service-invoices	26 Nov 2018 at 17.51	--	Folder
service-products	Yesterday at 14.05	--	Folder
service-users	26 Nov 2018 at 21.52	--	Folder
web-ui	Today at 0.26	--	Folder

Figure 36 Selecting project

After selecting project, the view will automatically load the npm script from project. Here in figure (Figure 37) we have selected “start” script command which starts the development server.


Service name

Web-UI

Project Path

/Users/yadavdip/Projects/dy-nova/nova-demo/web-ui

ENV Variables

Npm Scripts

- [ ] clean
- [x] start
- [ ] dev
- [ ] build:dev
- [ ] build
- [ ] test
- [ ] mockserver

Dockers

Figure 37 Setup Web-UI

On save we can see that our microservice appears on the main dashboard. On clicking the vscode icon we can open the project in editor for editing as shown in right window of figure (Figure 39). Developer can start the application without navigating to the application project simply by clicking on the play button beside “start” npm script as show in figure (Figure 38). When the application is started application log can be visible in lower section of the Nova tool (Figure 39). Here, it shows that our application is running on port 5000.

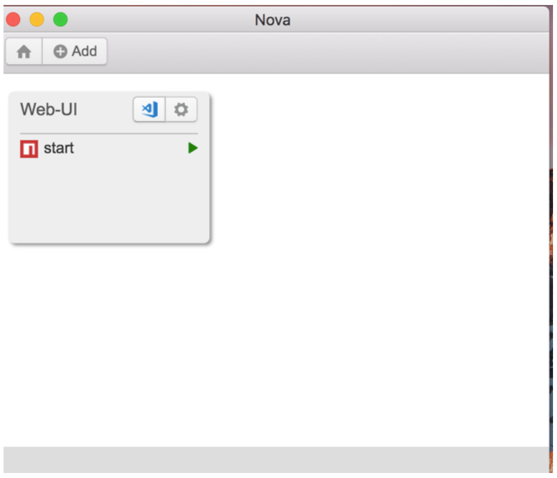
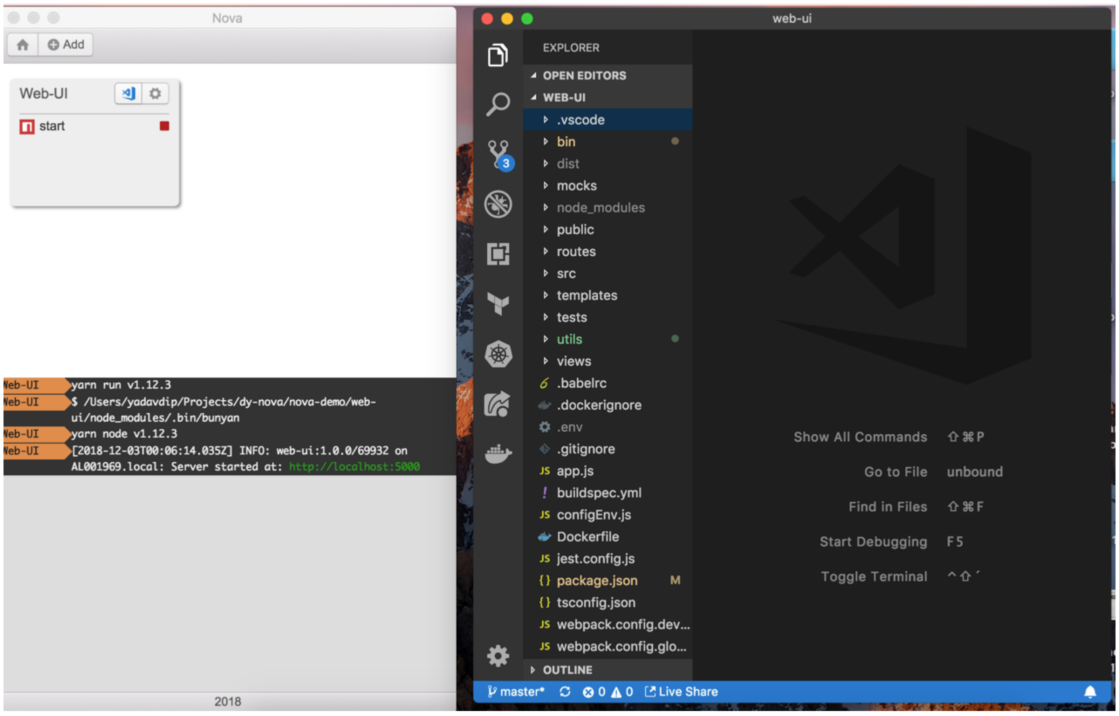


Figure 38 Configured Web-UI

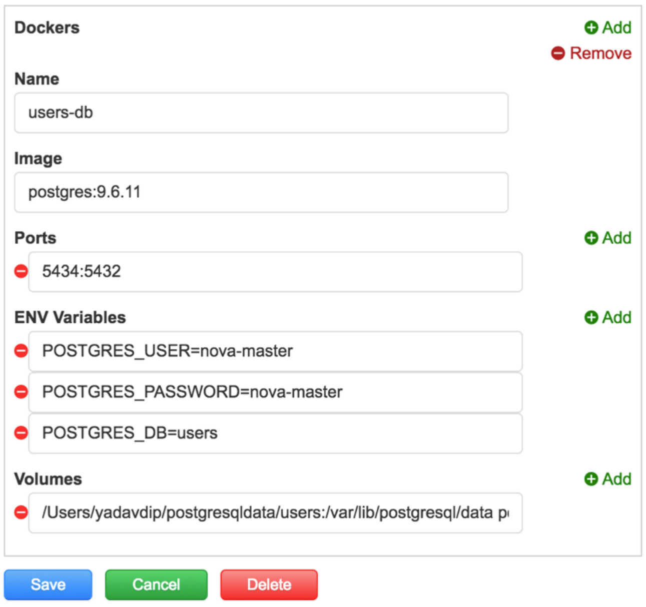


```
Web-UI  yarn run v1.12.3
Web-UI  $ /Users/yadavdip/Projects/dy-nova/nova-demo/web-
        ui/node_modules/.bin/bunyan
Web-UI  yarn node v1.12.3
Web-UI  [2018-12-03T00:06:14.035Z] INFO: web-ui:1.0.0/69932 on
        AL001969.local: Server started at: http://localhost:5000
```

Figure 39 Start application and edit code.

9.2 Setup microservices

Microservices application setup is two folds. One is similar to setting up frontend Web-UI. Second is to configure nova so database can be easily managed. For running database, we are going to use postgres Docker container. As shown in figure (Figure 40) we have entered desired properties for the users postgres database.



Dockers

Name
users-db

Image
postgres:9.6.11

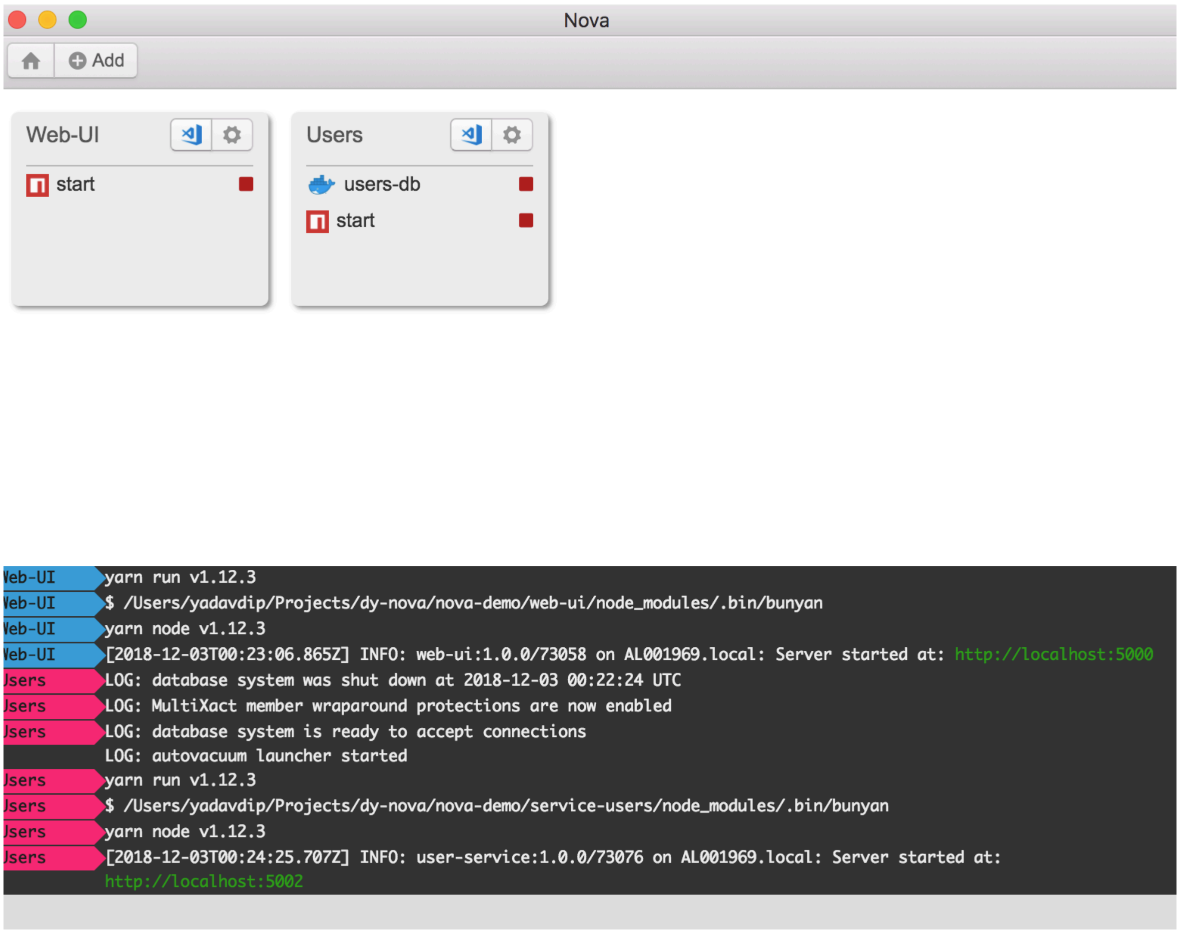
Ports
5434:5432

ENV Variables
POSTGRES_USER=nova-master
POSTGRES_PASSWORD=nova-master
POSTGRES_DB=users

Volumes
/Users/yadavdip/postgresqldata/users:/var/lib/postgresql/data p

Figure 40 Setup postgres docker

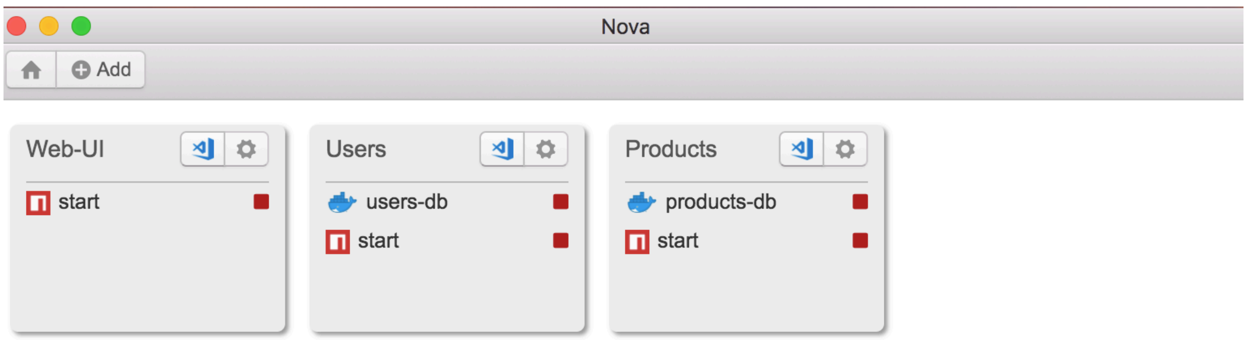
Now that our users microservice application and database is configured we can start both services. In the nova logs in figure (Figure 41) we can see that the users microservices started on port 5002. Also the stop icon on the service card indicates that the server is running.

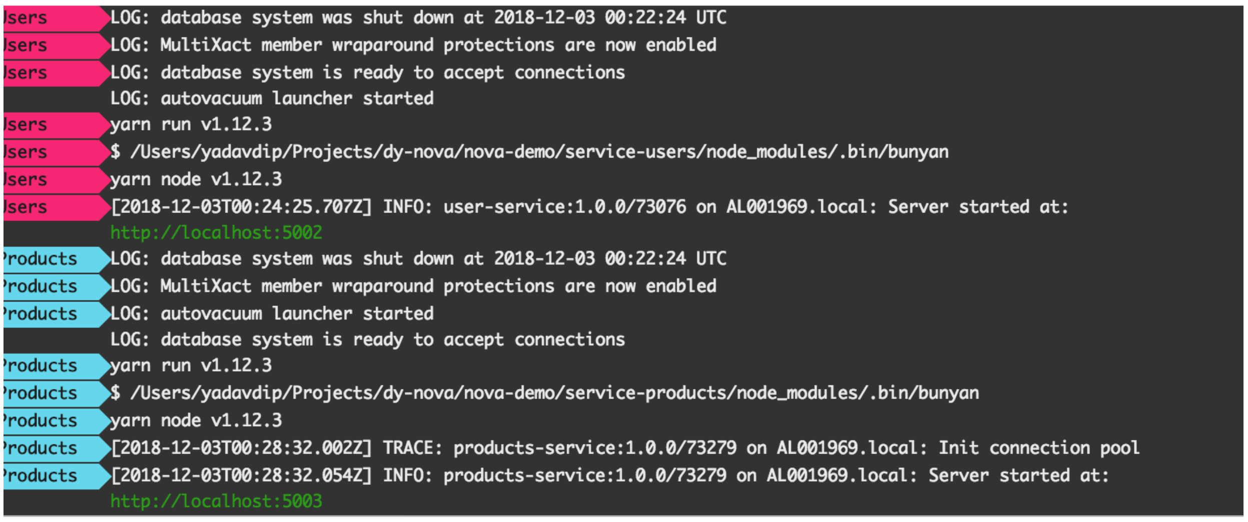


```
Veb-UI  yarn run v1.12.3
Veb-UI  $ /Users/yadavdip/Projects/dy-nova/nova-demo/web-ui/node_modules/.bin/bunyan
Veb-UI  yarn node v1.12.3
Veb-UI  [2018-12-03T00:23:06.865Z] INFO: web-ui:1.0.0/73058 on AL001969.local: Server started at: http://localhost:5000
Jsers   LOG: database system was shut down at 2018-12-03 00:22:24 UTC
Jsers   LOG: MultiXact member wraparound protections are now enabled
Jsers   LOG: database system is ready to accept connections
        LOG: autovacuum launcher started
Jsers   yarn run v1.12.3
Jsers   $ /Users/yadavdip/Projects/dy-nova/nova-demo/service-users/node_modules/.bin/bunyan
Jsers   yarn node v1.12.3
Jsers   [2018-12-03T00:24:25.707Z] INFO: user-service:1.0.0/73076 on AL001969.local: Server started at:
        http://localhost:5002
```

Figure 41 Start Users microservice

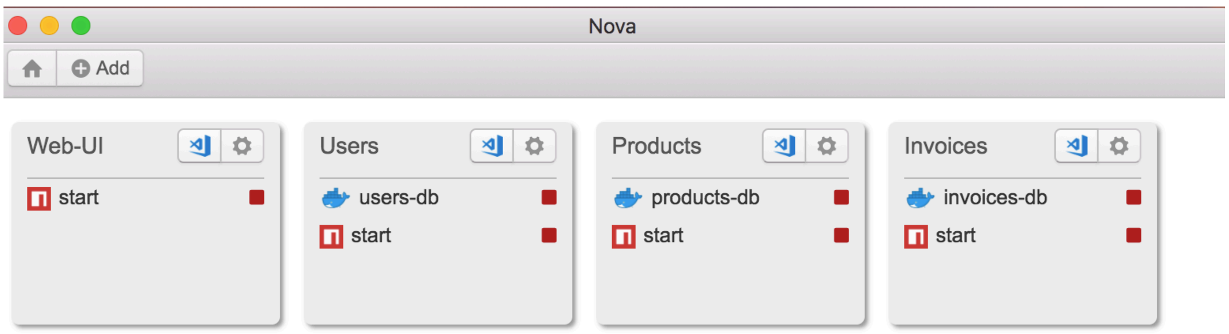
Similarly, Products and Invoices microservices can be configured and started up. Figure 42 and Figure 43 shows that Products and Invoices microservices are running along with its respective databases.

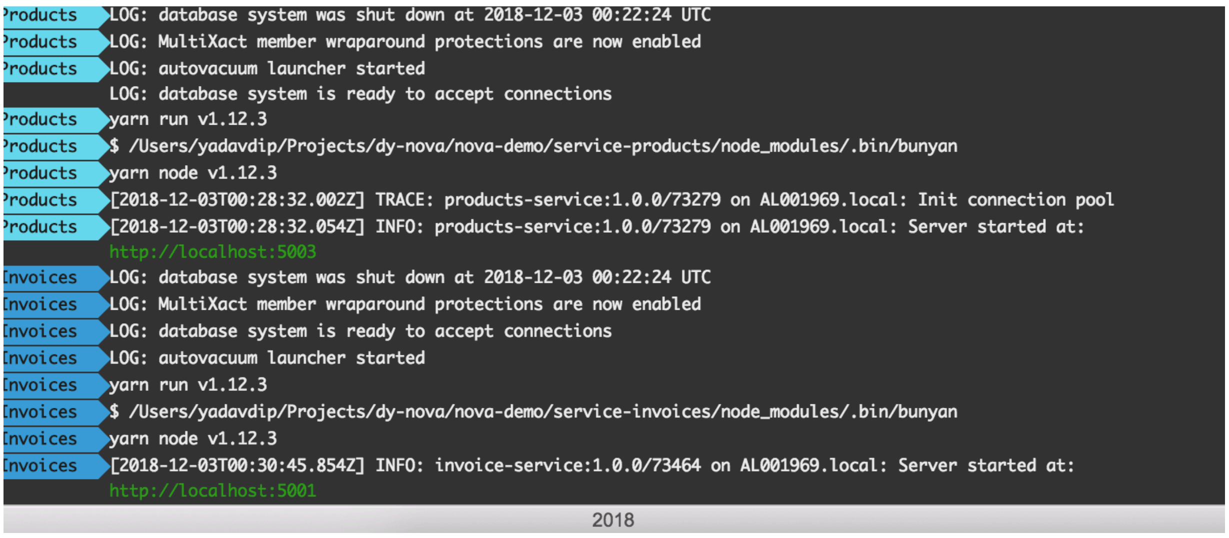




```
sers     LOG: database system was shut down at 2018-12-03 00:22:24 UTC
sers     LOG: MultiXact member wraparound protections are now enabled
sers     LOG: database system is ready to accept connections
         LOG: autovacuum launcher started
sers     yarn run v1.12.3
sers     $ /Users/yadavdip/Projects/dy-nova/nova-demo/service-users/node_modules/.bin/bunyan
sers     yarn node v1.12.3
sers     [2018-12-03T00:24:25.707Z] INFO: user-service:1.0.0/73076 on AL001969.local: Server started at:
         http://localhost:5002
roducts  LOG: database system was shut down at 2018-12-03 00:22:24 UTC
roducts  LOG: MultiXact member wraparound protections are now enabled
roducts  LOG: autovacuum launcher started
         LOG: database system is ready to accept connections
roducts  yarn run v1.12.3
roducts  $ /Users/yadavdip/Projects/dy-nova/nova-demo/service-products/node_modules/.bin/bunyan
roducts  yarn node v1.12.3
roducts  [2018-12-03T00:28:32.002Z] TRACE: products-service:1.0.0/73279 on AL001969.local: Init connection pool
roducts  [2018-12-03T00:28:32.054Z] INFO: products-service:1.0.0/73279 on AL001969.local: Server started at:
         http://localhost:5003
```

Figure 42 Start Products microservice





```
Products  LOG: database system was shut down at 2018-12-03 00:22:24 UTC
Products  LOG: MultiXact member wraparound protections are now enabled
Products  LOG: autovacuum launcher started
          LOG: database system is ready to accept connections
Products  yarn run v1.12.3
Products  $ /Users/yadavdip/Projects/dy-nova/nova-demo/service-products/node_modules/.bin/bunyan
Products  yarn node v1.12.3
Products  [2018-12-03T00:28:32.002Z] TRACE: products-service:1.0.0/73279 on AL001969.local: Init connection pool
Products  [2018-12-03T00:28:32.054Z] INFO: products-service:1.0.0/73279 on AL001969.local: Server started at:
          http://localhost:5003
Invoices  LOG: database system was shut down at 2018-12-03 00:22:24 UTC
Invoices  LOG: MultiXact member wraparound protections are now enabled
Invoices  LOG: database system is ready to accept connections
Invoices  LOG: autovacuum launcher started
Invoices  yarn run v1.12.3
Invoices  $ /Users/yadavdip/Projects/dy-nova/nova-demo/service-invoices/node_modules/.bin/bunyan
Invoices  yarn node v1.12.3
Invoices  [2018-12-03T00:30:45.854Z] INFO: invoice-service:1.0.0/73464 on AL001969.local: Server started at:
          http://localhost:5001
                                        2018
```

Figure 43 Start Invoices microservice

9.3 Developing microservices using Nova

Now that all our services are up and running as seen in Nova panel (Figure 44) indicated by red stop icon beside the tasks. We can visit url at port 5000 in our browser (Figure 45) to see that our application is up and running. Also note that all logs can be visible in Nova panel’s log window (Figure 46).

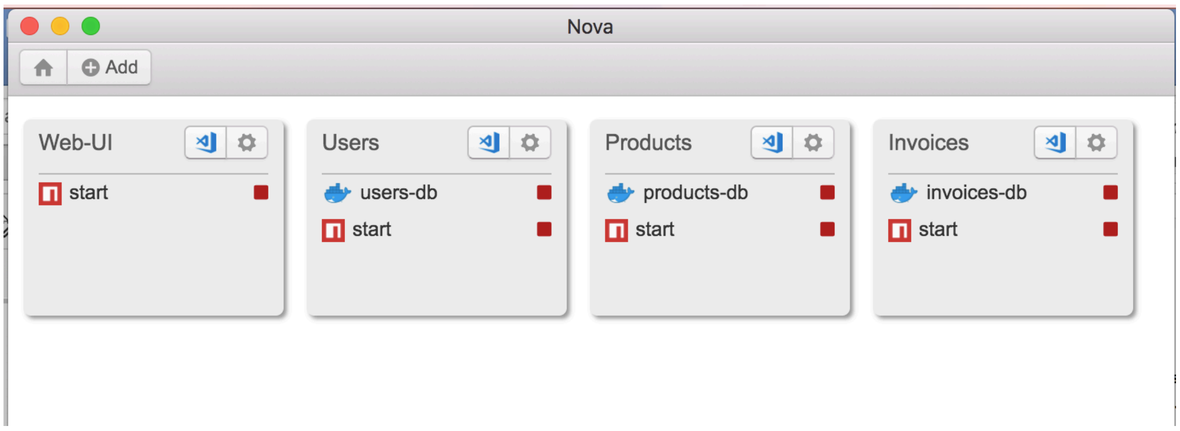


Figure 44 Nova panel

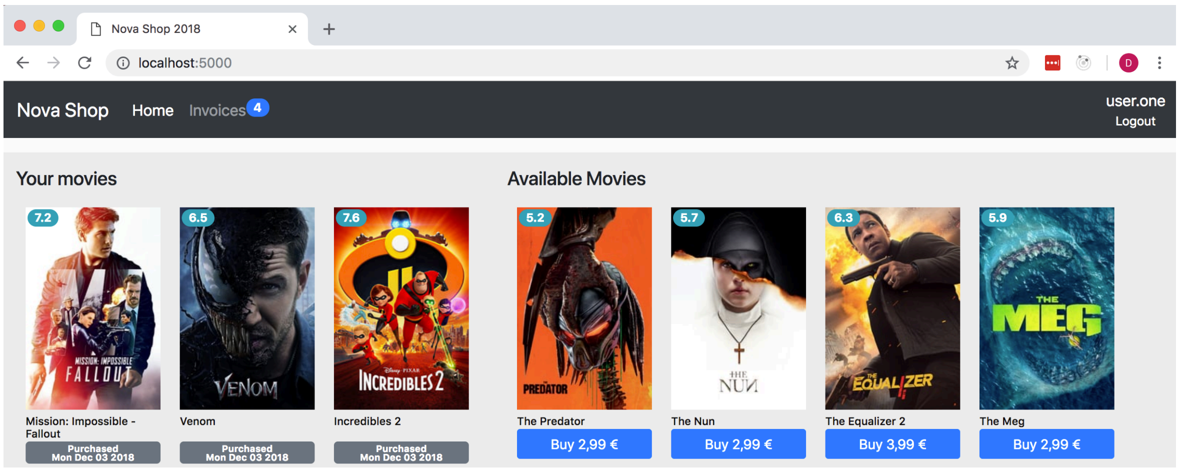
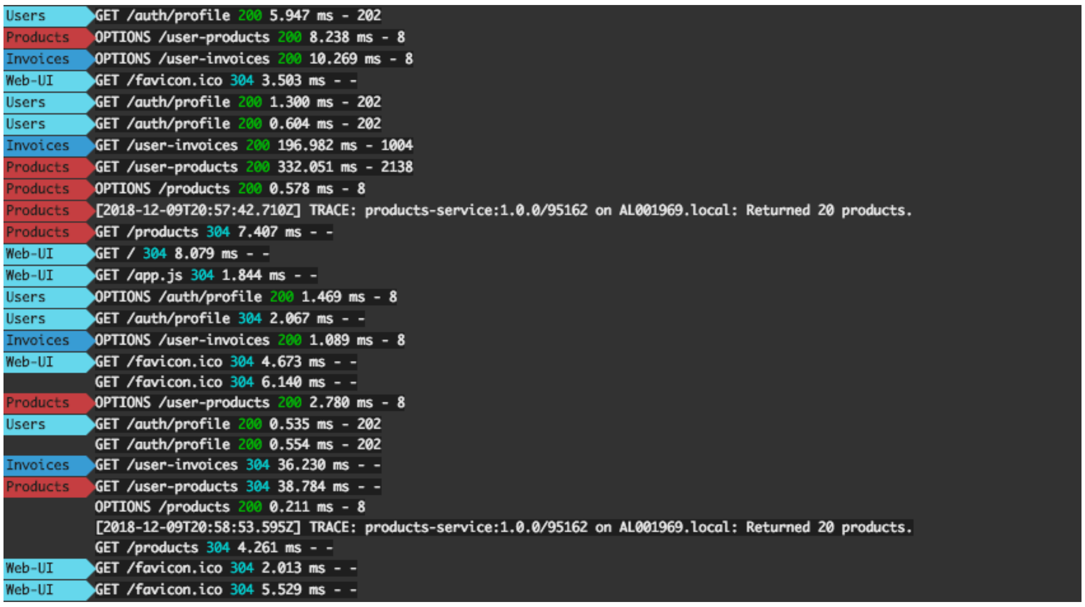


Figure 45 Application started from nova



```
Users    GET /auth/profile 200 5.947 ms - 202
Products OPTIONS /user-products 200 8.238 ms - 8
Invoices OPTIONS /user-invoices 200 10.269 ms - 8
Web-UI   GET /favicon.ico 304 3.503 ms - -
Users    GET /auth/profile 200 1.300 ms - 202
Users    GET /auth/profile 200 0.604 ms - 202
Invoices GET /user-invoices 200 196.982 ms - 1004
Products GET /user-products 200 332.051 ms - 2138
Products OPTIONS /products 200 0.578 ms - 8
Products [2018-12-09T20:57:42.710Z] TRACE: products-service:1.0.0/95162 on AL001969.local: Returned 20 products.
Products GET /products 304 7.407 ms - -
Web-UI   GET / 304 8.079 ms - -
Web-UI   GET /app.js 304 1.844 ms - -
Users    OPTIONS /auth/profile 200 1.469 ms - 8
Users    GET /auth/profile 304 2.067 ms - -
Invoices OPTIONS /user-invoices 200 1.089 ms - 8
Web-UI   GET /favicon.ico 304 4.673 ms - -
         GET /favicon.ico 304 6.140 ms - -
Products OPTIONS /user-products 200 2.780 ms - 8
Users    GET /auth/profile 200 0.535 ms - 202
         GET /auth/profile 200 0.554 ms - 202
Invoices GET /user-invoices 304 36.230 ms - -
Products GET /user-products 304 38.784 ms - -
         OPTIONS /products 200 0.211 ms - 8
         [2018-12-09T20:58:53.595Z] TRACE: products-service:1.0.0/95162 on AL001969.local: Returned 20 products.
         GET /products 304 4.261 ms - -
Web-UI   GET /favicon.ico 304 2.013 ms - -
Web-UI   GET /favicon.ico 304 5.529 ms - -
```

Figure 46 Centralized logging

In web-ui in order to change “Available movies” to “Recommended movies” developer can open the web-ui application from Nova. Also while searching for app.js in editor it shows only one app.js file compared to figure (Figure 34).

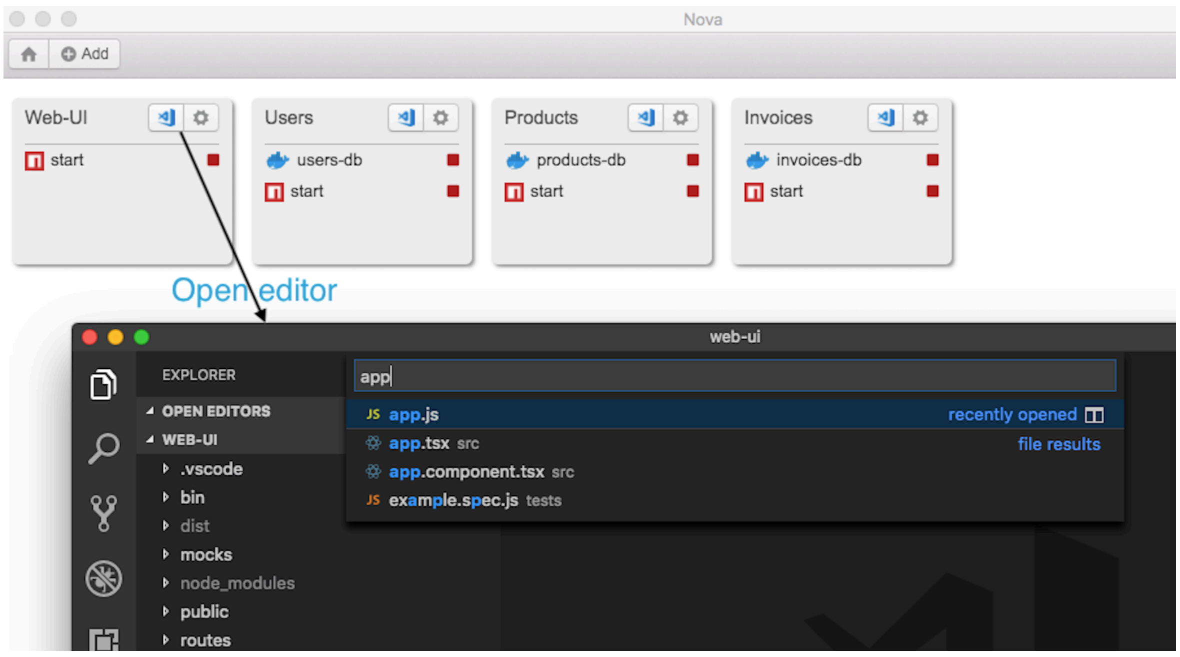
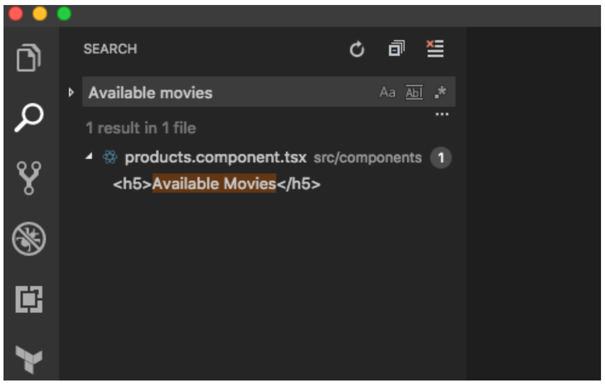


Figure 47 Open web-ui in editor

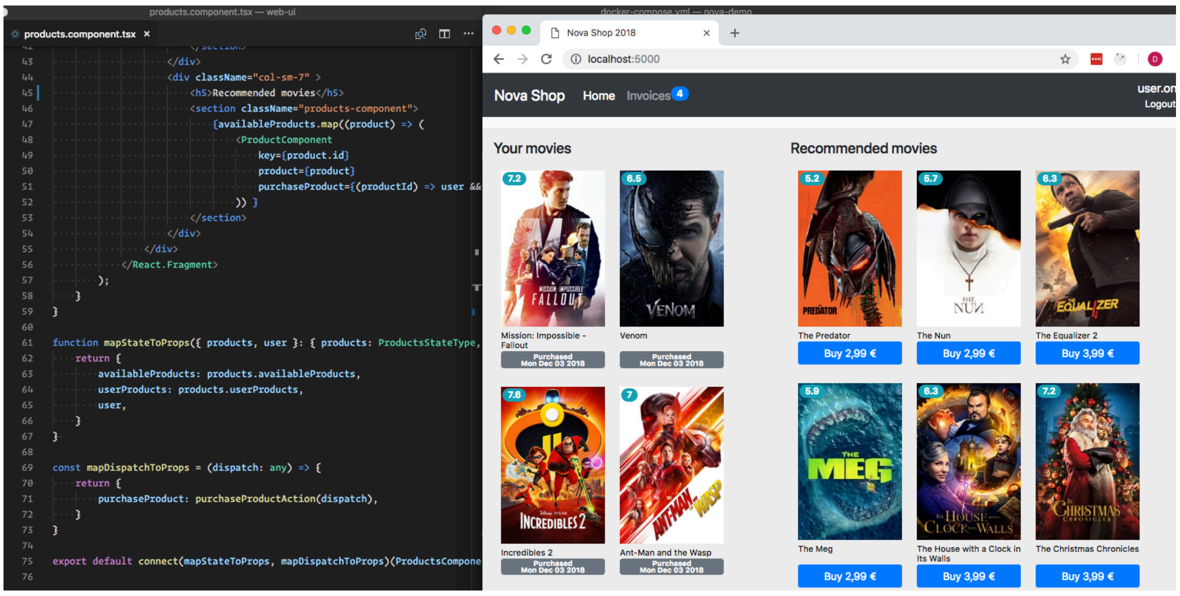
Now we can search for “Available movies” to find the implementation code. Here figure (Figure 48) finds one file matching the search. Since, our code base is small and our editor has only one project running, finding source code becomes super easy.



```
<h5>Available Movies</h5>
```

Figure 48 Search code location

Now we can navigate to the file and update the code. The code change is reflected in figure (Figure 49). Similarly, any microservice can be opened for editing and make the changes immediately.



```
                    </div>
                    <div className="col-sm-7" >
                        <h5>Recommended movies</h5>
                        <section className="products-component">
                            {availableProducts.map((product) => (
                                <ProductComponent
                                    key={product.id}
                                    product={product}
                                    purchaseProduct={(productId) => user &&
                                )) }
                        </section>
                    </div>
                </div>
            </React.Fragment>
        );
    }
}

function mapStateToProps({ products, user }: { products: ProductsStateType,
    return {
        availableProducts: products.availableProducts,
        userProducts: products.userProducts,
        user,
    }
}

const mapDispatchToProps = (dispatch: any) => {
    return {
        purchaseProduct: purchaseProductAction(dispatch),
    }
}

export default connect(mapStateToProps, mapDispatchToProps)(ProductsCompone
```

Figure 49 Edit code

As one can see from this demonstration microservices implementation this was becomes very smooth and clean. Getting into the project and finding the correct code become a easy process no matter how many microservices the project has.

10 Conclusion

Microservice is powerful architecture for developing modern applications. This helps on creating high quality complex applications. This architecture is constantly growing and improving. New tools and technologies has removed and minimized many challenges which previously made microservices architecture difficult to adopt. Nova is one of the tool created in this thesis to overcome development challenges in microservices architecture.

Developer find development easy and fun when they do not need to put much effort on trivial day to day tasks. Microservices requires developers to start up multiple resources every time before starting implementation. With this tool developer can start and stop resources with click of a button.

When the system is running in desired state developer can choose any microservice and open it in editor for implementation. The editor loads only the targeted microservice making the workspace clean and lightweight. Clean environment makes navigating and finding codes easier thus increasing focus and concentration.

Microservices is separate application thus having debug logs in separate places. With this tool all microservices logs can be viewed from one location. Developer now don't need to jump between different log windows. This makes it easier to trace the application flow.

GUI provides best usability to facilitate developer with many trivial and automation tasks. Nova was created with usability goals to increase productivity while implementing microservices. Complex operations can be hidden inside GUI so that developer don't need to type or memorize repetitive commands. Nova hides repetitive operation in form of easy to use UI which makes developers life easy, fun and productive.

References

Amazon Web Services 2018, AWS Whitepaper - Microservices on AWS. [online] Amazon.com, Inc. Available at: <https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/microservices-on-aws.pdf?icmpid=link_from_whitepapers_page> [Accessed: 06 Dec 2018].

Amazon Web Services 2018. What are Microservice? [online] Amazon.com, Inc. Available at: <https://aws.amazon.com/microservices> [Accessed: 06 Dec 2018].

Docker 18.09.0, 2018. What is a container. [online] Docker. Available at: < https://www.docker.com/resources/what-container> [Accessed: 28 Sep 2018].

Electron 2.0.6, 2018. Build cross platform desktop app with Javascript, HTML, and CSS. [online] Electron. Available at: <https://electronjs.org/> [Accessed: 06 Dec 2018].

Fowler, M., Lewis, J., 2014. Microservices. [online] Martin Fowler. Available at: < https://martinfowler.com/articles/microservices.html> [Accessed 12 Dec 2018].

IntelliJ Idea, 2018. Java integrated development environment <https://www.jetbrains.com/idea> [Accessed 14 Dec 2018].

Kitematic, 2018. Run container through graphical user interface. [online] Docker. Available at: <https://kitematic.com/> [Accessed: 29 Sep 2018].

Kubernetes 1.13, 2018. Production-Grade Container Orchestration. [online] Kubernetes. Available at: https://kubernetes.io/ [Accessed: 28 Sep 2018].

Nodejs 8.x, 2018. Javascript runtime. [online] Joyent, Inc. Available at: < https://nodejs.org/en/> [Accessed: 06 Dec 2018].

Sharma, A. 2017. Development of Microservices – Problems and Solutions. [online] Hackernoon. Available at: <https://hackernoon.com/development-of-microservices-problems-and-solutions-b3ce8f1f7ff1> [Accessed: 01 Oct 2018].

Spring Tool Suite 3. IDE for developing Spring Cloud microservices. [online] Pivotal. Available at: <https://spring.io/tools3> [Accessed: 29 Sep 2018].

Telepresence, 2018. Introduction to Telepresence. [online] Datawire, Inc. Available at: <https://www.telepresence.io/discussion/overview> [Accessed: 28 Sep 2018].

Turner-Trauring, I. 2017. Development Environments for Kubernetes. [online] Datawire, Inc. Available at: <https://www.datawire.io/guide/development/development-environments-microservices> [Accessed: 01 Oct 2018].

Visual Studio Code 1.29, 2018. Popular lightweight and powerful source code editor. [online] Microsoft. Available at: <https://code.visualstudio.com> [Accessed: 29 Sep 2018].

Appendix 1. Code for running child process

```
const childProcess = require('child_process');

export default function runCommand(command, args, options) {
    const child = childProcess.spawn(command, args, options || {});
    console.log(
        `runCommand: pid: ${child.pid}, cmd: ${command} ${args.join(' ')}`
    );
    child.stdout.on('data', data => {
        const response = data.toString().trim();
        if (response) {
            console.log(`STDOUT:${response}`);
        }
    });
    child.stderr.on('data', data => {
        const response = data.toString().trim();
        if (response) {
            console.log(`STDERR:${response}`);
        }
    });
    // stdio is closed
    child.on('close', () => console.log(child.pid, 'process closed'));
    // Parent closes
    child.on('disconnect', () => console.log(child.pid, 'process discon-
nect'));
    child.on('exit', (code, signal) =>
        console.log(
            'runCommand: Process exiting',
            JSON.stringify({
                pid: child.pid,
                code,
                signal
            })
        )
    );
    child.on('error', err => {
        console.log('runCommand: Error opening process', err);
    });
    return child;
}
```

Appendix 2. Core code for running npm and docker tasks

```
import { ChildProcess } from 'child_process';
import { resolve as pathResolve } from 'path';
import { createWriteStream } from 'fs';
import runCommand from './commands';
import {
    EDITOR_OPEN,
    NPMSCRIPT_START,
    NPMSCRIPT_STOP,
    NPMSCRIPT_VIEWLOG,
    DOCKER_START,
    DOCKER_STOP,
    DOCKER_VIEWLOG
} from '../constants/tasks-constants';
import {
    TaskType,
    NpmTaskType,
    EditorTaskType,
    DockerTaskType
} from '../types/task';
import { ServiceType } from '../types/service';
import { deleteDir, cleanAndCreateDir, parseEnvvarList } from './utils';

let LOGS_PATH: string;
let sendEvent;

const debug = (...args) => {
    sendEvent('message', ...args);
};
const NPMSCRIPT_BIN = 'yarn';
const CODE_BIN = 'code';
const DOCKER_BIN = 'docker';

export default function createTaskRunner(cwd, _sendEvent) {
    LOGS_PATH = pathResolve(cwd, 'logs');
    sendEvent = _sendEvent;
    cleanAndCreateDir(LOGS_PATH);

    return {
        taskRunner,
        stopAllTasks
    };
    function taskRunner(serviceContext: ServiceType, taskName, task: Task-
Type) {
        debug(
            `Running Task: ${taskName} (${serviceContext.name}: ${
                serviceContext.id
            })`,
            JSON.stringify({ task, NPMSCRIPT_BIN, CODE_BIN })
        );
        switch (taskName) {
            case EDITOR_OPEN:
                openEditor(serviceContext, task);
                break;
            case NPMSCRIPT_START:
                taskStart(
                    serviceContext,
                    task,
                    parseNpmCommand(serviceContext, task)
                );
                break;
            case NPMSCRIPT_STOP:
                taskStop(serviceContext, task);
                break;
            case NPMSCRIPT_VIEWLOG:
                break;
            case DOCKER_START:
                taskStart(
                    serviceContext,
                    task,
                    parseDockerCommand(serviceContext, task)
                );
                break;
            case DOCKER_STOP:
                taskStop(serviceContext, task);
                break;
            case DOCKER_VIEWLOG:
                break;
            default:
                debug('Unknown task', { taskName, task });
        }
    }

    function stopAllTasks(cb) {
        console.log('Stopping all tasks');
        Object.keys(global.runningTasks).forEach(taskId => {
            const { taskProcess } = global.runningTasks[taskId];
            if (taskProcess) {
                console.log(`KILL SIGTERM process group ${-process.pid}`);
                process.kill(-taskProcess.pid, 'SIGTERM');
            }
        });
        if (cb) cb();
    }
}

function openEditor(serviceContext: ServiceType, task: EditorTaskType) {
    const taskProcess = runCommand(CODE_BIN, [task.projectDir], {
        cwd: serviceContext.projectDir,
        env: {
            ...process.env,
            CWD: serviceContext.projectDir
        }
    });
    taskProcess.on('close', () =>
        debug(`openEditor closed: ${serviceContext.name}`)
    );
    taskProcess.on('exit', () =>
        debug(`openEditor exit: ${serviceContext.name}`)
    );
}

function parseNpmCommand(serviceContext: ServiceType, task: NpmTaskType) {
    return parse(
        task.cmd.replace(/(np(m|x)\s+(run)?|yarn)/g, ''),
        NPMSCRIPT_BIN
    );

    function parse(cmdString, cmd) {
        const chained = cmdString.split('&&').map(x => parsePipes(x, cmd));
        return ['AND', ...chained];
    }

    function parsePipes(cmdString, cmd) {
        const piped = cmdString.split('|').map(x => {
            const args = x
                .trim()
                .replace(/\s+/g, '::')
                .split('::');

            const [maybeCmd, ...restArgs] = args;
            if (maybeCmd === 'docker') {
                return {
                    cmd: maybeCmd,
                    args: restArgs
                };
            }
            return {
                cmd,
                args
            };
        });
        return piped.length > 1 ? ['PIPE', ...piped] : piped[0];
    }
}

function parseDockerCommand(serviceContext: ServiceType, task: DockerTask-
Type) {
    const cmd = DOCKER_BIN;
    const args = [];

    args.push('run');
    args.push('--rm');
    args.push('--name');
    args.push(`${task.container_name}-${serviceContext.id}`);
    task.ports.forEach(port => {
        args.push('-p');
        args.push(port);
    });

    task.env.forEach(singleEnv => {
        args.push('-e');
        args.push(singleEnv);
    });

    task.volumes.forEach(volume => {
        args.push('-v');
        args.push(volume);
    });

    args.push(task.image);

    return [
        'AND',
        {
            cmd,
            args
        }
    ];
```

```
}

/**
 * Mutates taskData
 *
 * @param {*} serviceContext
 * @param {*} task
 * @param {*} cmdDescription
 */
function taskStart(
    serviceContext: ServiceType,
    task: TaskType,
    cmdDescription
) {
    const taskData = (global.runningTasks[task.id] =
        global.runningTasks[task.id] || {});

    if (taskData.taskProcess) {
        sendEvent('taskstates', 'start', task);
        return;
    }

    debug(`Task running ${JSON.stringify(cmdDescription)}`);

    runChain(serviceContext, cmdDescription);
    sendEvent('taskstates', 'start', task);

    taskData.consoleAppNS = serviceContext.name;
    // Enable console for app by default
    taskData.consoleAppEnabled = true;

    /* eslint-disable */
    async function runChain(context: ServiceType, cmds) {
        const [operator, ..._cmds] = cmds;
        for (let i = 0; i < _cmds.length; i += 1) {
            const cmd = _cmds[i];
            try {
                await runAndWait(context, cmd);
            } catch (err) {
                console.log('Failed to run command', {
                    cmd,
                    err
                });
            }
        }
        sendEvent('taskstates', 'stop', task);
    }
    /* eslint-enable */
    async function runAndWait(context: ServiceType, cmd) {
        // PIPED commands
        if (Array.isArray(cmd)) {
            const [operator, ...cmds] = cmd;
            if (operator !== 'PIPE') {
                throw new Error(`Unknown operator ${operator}`);
            }
            const childs = [];
            cmds.forEach((_cmd, idx) => {
                const curPS: ChildProcess = runTaskProcess(context, _cmd);
                const prevPS: ChildProcess | null =
                    idx >= 1 ? childs[idx - 1] : null;
                if (prevPS) {
                    prevPS.stdout.pipe(curPS.stdin);
                    prevPS.stderr.pipe(curPS.stdin);
                }
                childs.push(curPS);
            });
            const [main, ...pipeProcesses] = childs;
            taskData.taskProcess = main;
            attachConsoleLogView(
                pipeProcesses[pipeProcesses.length - 1],
                task.id
            );
            await waitTillclosed(main);
            // End all pipe processes
            pipeProcesses.forEach((_ps: ChildProcess) => {
                process.kill(_ps.pid, 'SIGTERM');
            });
        } else {
            taskData.taskProcess = runTaskProcess(context, cmd);
            attachConsoleLogView(taskData.taskProcess, task.id);
            await waitTillclosed(taskData.taskProcess);
        }

        debug(
            'Task terminating',
            JSON.stringify({
                pid: taskData.taskProcess.pid
            })
        );
        taskData.taskProcess = null;
        handleTaskExit(task);
    }
}

function waitTillclosed(taskProcess) {
    return new Promise((resolve, reject) => {
        taskProcess.on('close', () => {
            resolve('close');
        });
        taskProcess.on('error', () => {
            reject(new Error('error'));
        });
    });
}
function runTaskProcess(serviceContext: ServiceType, cmdDesc) {
    const { cmd, args } = cmdDesc;
    let env = {};
    if (serviceContext.envvars) {
        env = {
            ...env,
            ...parseEnvvarList(serviceContext.envvars)
        };
    }
    // Add path manually so executables in /bin are available
    env.PATH = process.env.PATH;

    const taskProcess = runCommand(cmd, args, {
        cwd: serviceContext.projectDir,
        detached: true,
        env
    });
    taskProcess.unref();
    return taskProcess;
}

function taskStop(serviceContext: ServiceType, task: TaskType) {
    const { taskProcess } = global.runningTasks[task.id];
    if (taskProcess) {
        console.log(`Sending SIGTERM to process group ${-taskProcess.pid}`);
        process.kill(-taskProcess.pid, 'SIGTERM');
    }
}

function handleTaskExit(task) {
    const taskData = (global.runningTasks[task.id] =
        global.runningTasks[task.id] || {});
    if (taskData.logstream) {
        taskData.logstream.end();
        taskData.logstream = null;
        deleteDir(taskData.logFile);
    }

    sendEvent(
        'consolewindow:log',
        task.id,
        taskData.consoleAppNS,
        'Closing...'
    );
}

function attachConsoleLogView(taskProcess, taskId: string) {
    const taskData = (global.runningTasks[taskId] =
        global.runningTasks[taskId] || {});

    taskData.logFile = pathResolve(
        LOGS_PATH,
        `${taskId.replace(':', '-')}-${taskProcess.pid}`
    );
    taskData.logstream = createWriteStream(taskData.logFile);
    taskData.logstream.on('close', () => {
        debug(`Closing log stream: ${taskData.logFile}`);
    });

    const outputWriter = createOutputWriter(taskId);
    taskProcess.stdout.on('data', outputWriter);
    taskProcess.stderr.on('data', outputWriter);
}

function createOutputWriter(taskId) {
    return data => {
        const taskData = (global.runningTasks[taskId] =
            global.runningTasks[taskId] || {});

        // Write to log file for preserving log
        if (taskData.logstream) {
            taskData.logstream.write(data);
        }
        // Send output to app via event if enabled
        if (taskData.consoleAppEnabled) {
            sendEvent(
                'consolewindow:log',
                taskId,
                taskData.consoleAppNS,
                data.toString().trim()
            );
        }
    };
}
```

Appendix 3. Code for parsing data from processes

```
module.exports = {
    decorateAnsiColours,
    parseAnsiCursorCommands,
    decorateLinks
};

/**
 * http://tldp.org/HOWTO/Bash-Prompt-HOWTO/x361.html
 * <N>A: Move N line up
 * <N>B: Move N line down
 * K: Erase to end of line
 * @param {string} data
 */
function parseAnsiCursorCommands(data) {
    return (
        data
            .replace(/\u001B\[(\d+)([A-BK])/g, (_, line, command) => {
                switch (command) {
                    case 'A':
                        return '::UP::';
                    case 'B':
                        return '::DOWN::';
                    case 'K':
                        return '::CLEAR_LINE::';
                    default:
                        console.log(`Unknown Command: ${command}`);
                }
            })
            .split('::')
    );
}
```

```
function decorateAnsiColours(message) {
    // eslint-disable-next-line no-control-regex
    const match = message.match(/\u001B\[((?:\d+;)*\d+)m/);
    let classNames = [];
    if (match) {
        const capture = match[1];
        // 0 is reset all
        if (capture === '0') {
            classNames.push('ansi-reset');
        } else {
            classNames = capture.split(';').map(code => {
                switch (+code) {
                    case 1:
                        return 'ansi-bold';
                    case 31:
                        return 'ansi-red';
                    case 32:
                        return 'ansi-green';
                    case 33:
                        return 'ansi-yellow';
                    case 34:
                        return 'ansi-blue';
                    case 36:
                        return 'ansi-cyan';
                    case 39:
                        return 'ansi-white';
                    default:
                        console.log(`Unknown code: ${code}`);
                        return 'ansi-default';
                }
            });
        }

        const preTag = `<span class="${classNames.join(' ')}" >`;
        const postTag = '</span>';
        return decorateAnsiColours(message.replace(match[0], preTag) +
postTag);
    }
    return message;
}

function decorateLinks(data) {
    return data.replace(/(http[^\s\n<]+)/, '<a href="$1">$1</a>');
}
```

Appendix 4. Main code for running GUI

```javascript
import { ipcRenderer } from 'electron';
import React from 'react';
import { render } from 'react-dom';
import { AppContainer } from 'react-hot-loader';
import Root from './containers/Root';
import { configureStore, history } from './store/configureStore';
import './app.global.css';
import persistentStore from './utils/store';
import { ADD_LOG } from './actions/pconsole';
import { updateTaskstateAction } from './actions/task-state.actions';

const store = configureStore({
    counter: 0,
    consoles: [
        {
            id: 'main-console',
            name: 'Main Console',
            logs: [],
            nsColours: {}
        }
    ],
    services: persistentStore.get('services')
});

ipcRenderer.on('message', (event, ...messages) => {
    console.log('main:', ...messages);
});
ipcRenderer.on('error', (event, ...messages) => {
    console.error('main:', ...messages);
});
ipcRenderer.on('tasks-snapshot', (_, runningtasks) => {
    console.log('tasks-snapshot:', runningtasks);
});
ipcRenderer.on('consolewindow:log', (_, taskId, ns, log) => {
    store.dispatch({
        type: ADD_LOG,
        id: 'main-console', // Add log to main console.
        ns,
        log
    });
});

ipcRenderer.on('taskstates', (_, state, task: TaskType) => {
    updateTaskstateAction({
        id: task.id,
        state
    })(store.dispatch);
});

render(
    <AppContainer>
        <Root store={store} history={history} />
    </AppContainer>,
    document.getElementById('root')
);

if (module.hot) {
    module.hot.accept('./containers/Root', () => {
        const NextRoot = require('./containers/Root'); // eslint-disable-
line global-require
        render(
            <AppContainer>
                <NextRoot store={store} history={history} />
            </AppContainer>,
            document.getElementById('root')
        );
    });
}
```