Kubernetes for Web Developers

Configuring and Deploying a Microservices Application

Car Rental System

Created by Ivaylo Kenov for the Code It Up Initiative

Requirements

This guidebook requires basic **Docker** knowledge. You need to understand how to build images, run containers, and use **Docker Compose**. You must be comfortable with the commonly used terminal commands before continuing with this **Kubernetes** tutorial. If you do not understand core containerization concepts or if you have never worked with **Docker** before – learn it first and then come back here.

System Requirements – Car Rental Dealers

We have a system in which car dealers can publish their cars for rent. Each car ad must contain manufacturer, model, category, image, and price per day. Categories have a description and must be one of the following - economy, compact, estate, minivan, SUV, and cargo van. Additionally, each vehicle must list the following options: with or without climate control, number of seats, and transmission type.

The system allows users to filter the cars by category, manufacturer, and price range anonymously. Ads can be sorted by manufacturer or by price.

When a user chooses a car, she needs to call the dealer on the provided phone and make the arrangement. The dealer then needs to edit the car ad as “currently unavailable” manually. The system must not show the unavailable cars.

Troubleshooting

If you mess up your local cluster somehow during this tutorial, you can try deleting all **Kubernetes** objects:

```
kubectl delete all --all
kubectl delete pvc --all
```

Then reapply everything you already configured. For the full application, these are the commands:

```
kubectl apply -f .\.environment\
kubectl apply -f .\databases\
kubectl apply -f .\event-bus\
kubectl apply -f .\web-services\
kubectl apply -f .\clients\
```

Make sure you wait a bit between each command to make sure all containers in the pods are up and running.

Additionally, if one of your pods is in "*Pending*" state and does not start, run a "*describe*" command on it. Your machine may not have enough CPU and/or memory resources to run the whole cluster. If this is the case, reduce the pod's limits until it works. However, do not put less than 2 GiB of memory for the databases because of their minimum requirements.

This tutorial assumes you have at least 6 CPU cores and 12 GiB of memory on your development environment. If not, adjust the resource limits in your manifest files as you see fit.

You can also reset the whole cluster if you want to start over.

On **Docker Desktop** - *Settings -> Kubernetes -> Reset Kubernetes*.

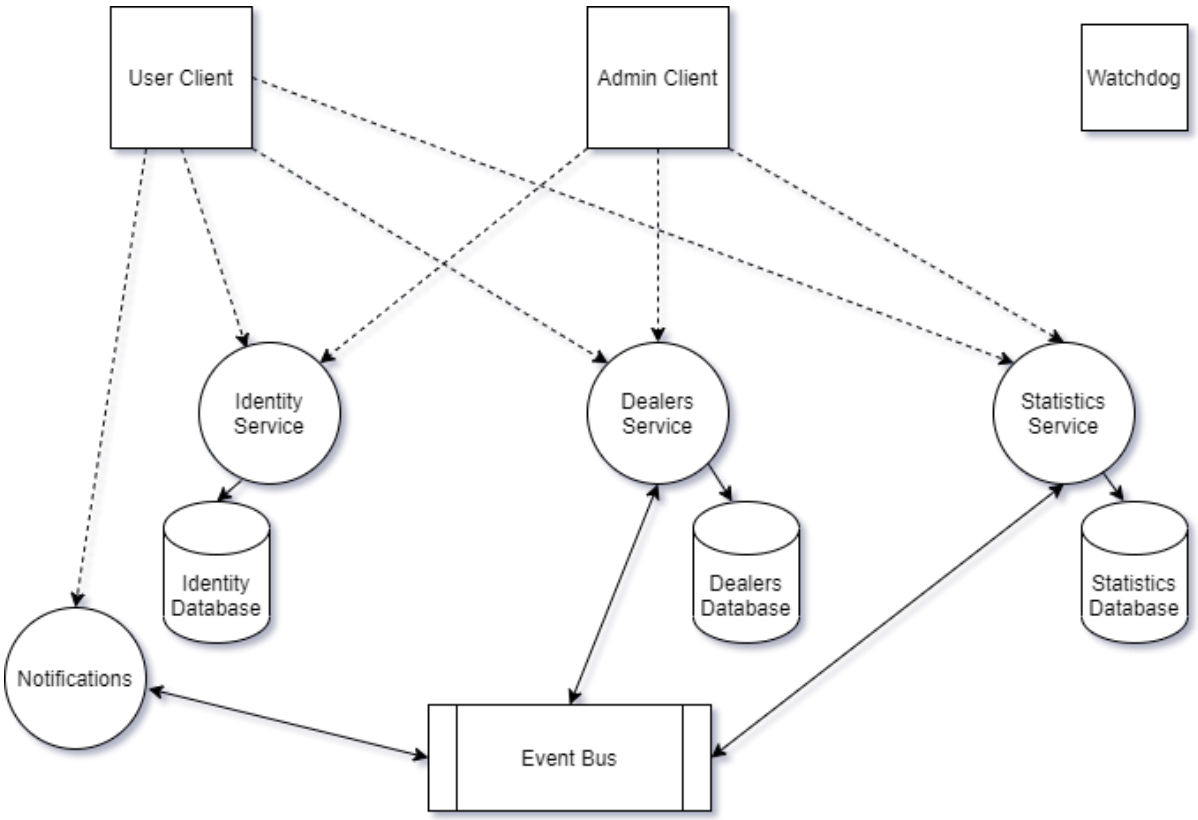
On **Minikube** – run "*minikube delete*" and then "*minikube start*".

Key Kubernetes Concepts

- **Node** – a machine running your pods. Part of the cluster's physical infrastructure.
- **Pod** – an application running containers on your nodes. The smallest unit of deployment. Provides resource limitations. You should never deploy pods directly.
- **ConfigMap** – contains visible key-value pair configuration. Useful for application settings.
- **Secret** – contains secret key-value pair configuration. Useful for passwords, connection strings, and more.
- **Persistent Volume Claim** – used for requesting dynamically provisioned storage. Persistence depends on the infrastructure.
- **StatefulSet** – deployment manifest for pods that have a state. Useful for databases, cache servers, and more.
- **Deployment** – deployment manifest for pods with stateless servers. Useful for API servers. Allows easy pod updating, scaling, and replicating.
- **Service** – provides an internal or external networking interface for the cluster. Supports load balancing.

1. Understanding the Application Architecture

This is the application's architecture:



These are the different parts of the system:

- *Event Bus* – a message queue system allowing communication between the back-end services in an asynchronous manner.
- *Identity Service* – a restful API providing functionality for authentication and authorization. This service has its own database.
- *Dealers Service* – a restful API providing functionalities for dealers to create profiles and publish car ads. Exposes endpoints for searching and viewing car ads. This service has its own database and communicates with the event bus.

- *Statistics Service* – a restful API providing functionalities for storing and retrieving usage statistics – the total car ads in the system, for example. This service has its own database and communicates with the event bus.
- *Notifications Service* – a web service, which sends notifications to the client through web sockets. Communicates with the event bus.
- *User Client* – a front-end service, which serves a single-page application to the end-users. Communicates with the identity, dealers, statistics, and notifications services to receive and visualize data.
- *Admin Client* – a front-end service, which is used by the business administrators. Communicates with the identity, dealers, and statistics services to receive and visualize data.
- *Watchdog Client* – periodically calls all the other services to validate their health state and availability.

The application uses the following technologies:

- **RabbitMQ** as an event bus and message broker.
- **SQL Server 2019** as a database provider.
- **ASP.NET Core 3.1** running on **Kestrel** as a web server for all services except the user client.
- **SignalR** for the real-time notifications service.
- **Angular** running on **nginx** for the front-end user client.

Note: You do not need to understand the above technologies to complete this guidebook and learn how to configure a Kubernetes cluster. All the specific programming details are already implemented. Of course, you may try to recreate all the steps provided here to deploy your own application as well.

Now, let us start!

2. Installing a Local Development Environment

Before you begin the next steps of this practical guidebook, make sure you have **Docker** and **Kubernetes** up and running on your machine. If you already have them – great, you may skip this section. Just make sure you have an active and linked account on **Docker Hub**.

First - **Docker**.

The easiest way to start using it on **Windows** or **Mac** is by installing **Docker Desktop**.

*Note: If your host Operational System is Windows Home, you will need to install WSL 2 (Windows Subsystem for Linux). Follow the steps from **here**.*

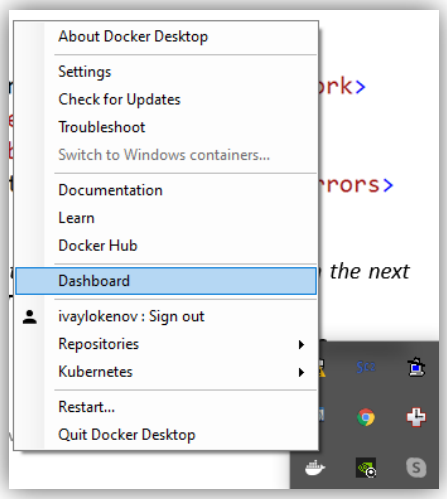
If you work on **Linux** – see the instructions **here**. You may also check **DockStation**.

After the installation is complete, open a terminal and run the following command to make sure **Docker** is ready:

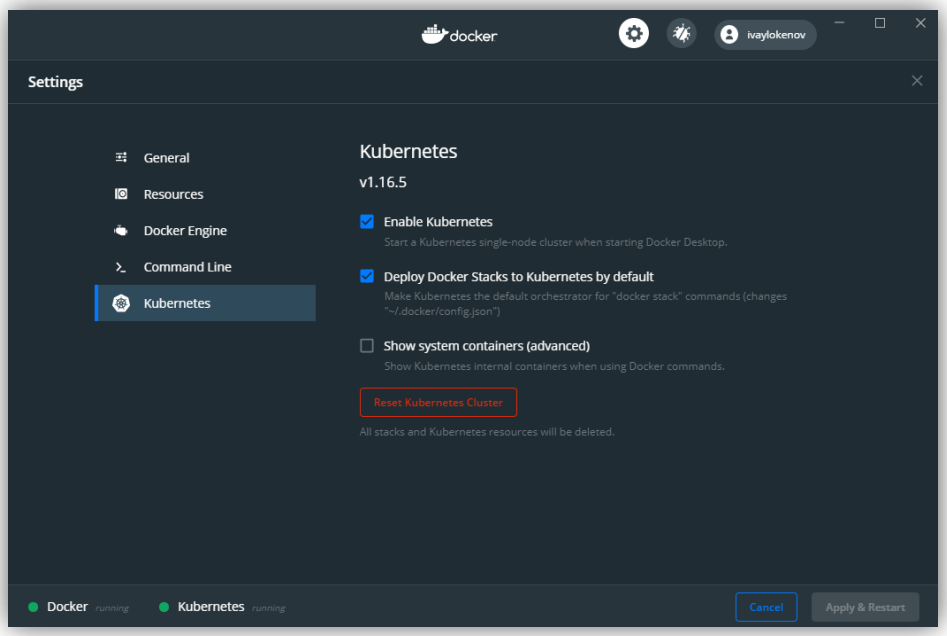
```
docker version
```

Next – **Kubernetes**.

If you are using **Docker Desktop**, you need to turn on **Kubernetes** by opening the dashboard.



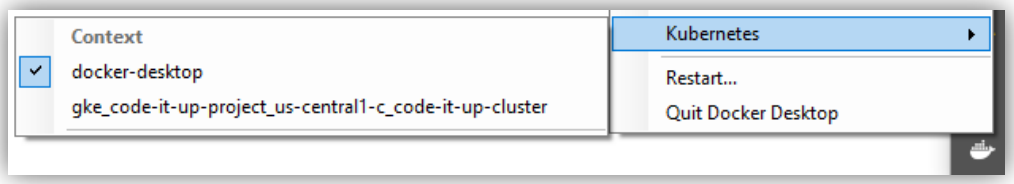
From there choose *Settings -> Kubernetes -> Enable Kubernetes*.



Wait for **Kubernetes** to initialize and make sure it is up and running with:

```
kubectl config current-context
```

The result should be "*docker-desktop*". If not, change it from here:



If you work on **Linux** – install **kubectl** and **Minikube** by following the instructions **here**.

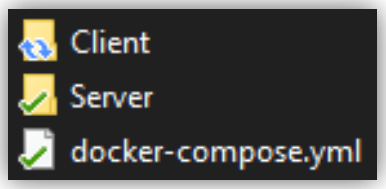
Finally – **Docker Hub**. Go to **https://hub.docker.com** and make an account, if you don't have one. Then go to your terminal and login with:

```
docker login
```

Done. You are good to go!

3. Preparing the Container Images

Navigate your terminal to the **Application** folder or open a new one in it:



The application is already configured to use **Docker Compose**. Open the "*docker-compose.yml*" file in your favorite text editor and edit all the image names to contain your **Docker Hub** username.

Let us say your username is "*codeitup*". Then you will need to replace all "*ivaylokenov*" names with "*codeitup*". For example, this line:

```
image: ivaylokenov/carrentalsystem-user-client:1.0
```

Must become:

```
image: codeitup/carrentalsystem-user-client:1.0
```

You need to edit 7 image names in total – *client*, *identity*, *dealers*, *statistics*, *notifications*, *admin*, and *watchdog*.

Save the file and go back to the terminal. Let us run the application and test it. But first, we need to build it:

```
docker-compose build
```

You will need to wait a bit – a lot of downloading and building is happening! Go grab a beer or something…

Note: Ignore the Angular deprecation warnings. In the JavaScript world, everything moves so fast that even this three-month-old application is now obsolete.

The build process is slow because you do not have any of the required files locally on your machine. When you use **Docker** in real-world application development, you will have a lot of the image layers ready and cached.

Run the following command to analyze the built images:

```
docker images
```

You should see 7 images containing your **Docker Hub** username and some others, which you need during the build process.

Now, run the whole application:

```
docker-compose up -d
```

Again, you will need to wait a bit.

First, for the **SQL Server 2019** and **RabbitMQ** images – they were not part of the build process because we use them "as is" in our configuration. They need to be downloaded.

Second, for our application bootstrapping – all the databases need to be created, all the data needs to be populated, and all the services and communications need to be started.

Note: Even though Docker Compose shows that our containers are started, you still need to wait 5-10 seconds more before the application code completes its initial configurations.

Open your browse and validate all clients:

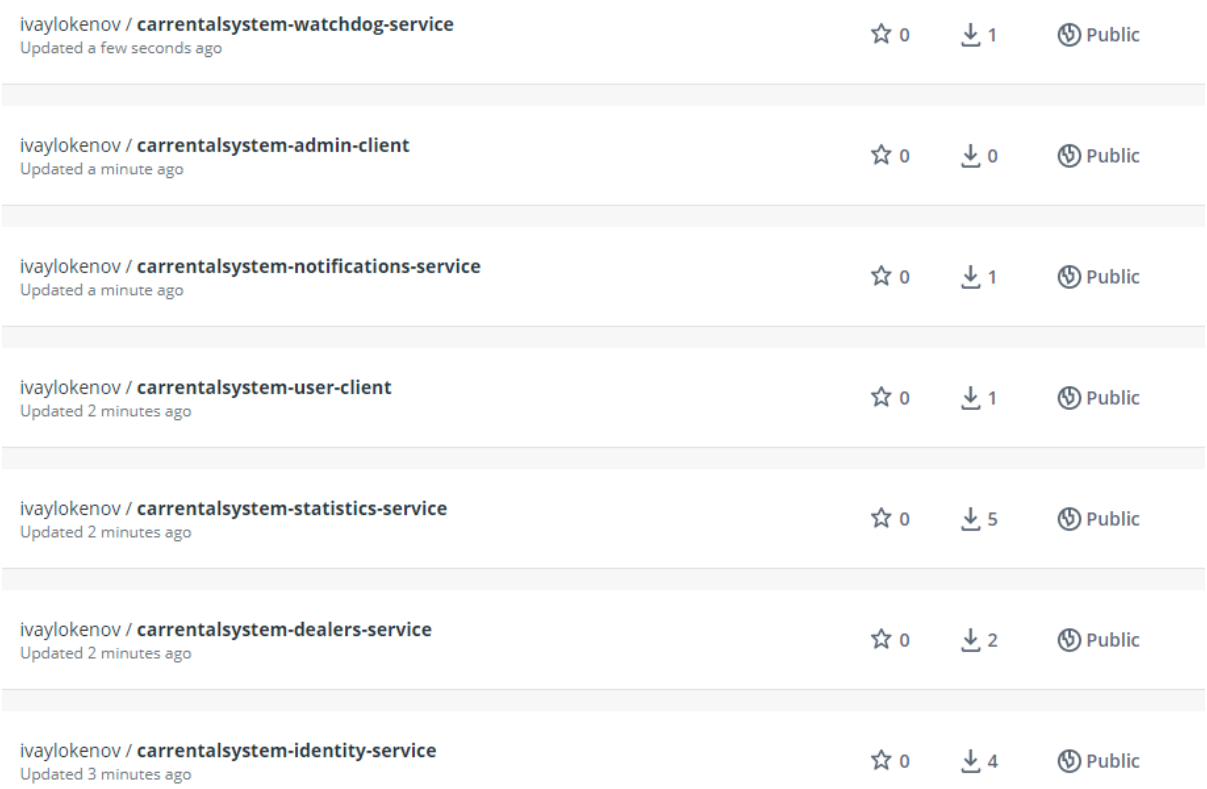
- *User Client* – go to **http://localhost**. The system should have 3 cars already populated. You can log in with a preseeded normal user by using "*coolcars@crs.com*" as an e-mail and "*coolcars12!*" as a password.
- *Admin Client* – go to **http://localhost:5000**. You can log in with a preseeded administrator user by using "*admin@crs.com*" as an e-mail and "*adminpass12!*" as a password.

- *Watchdog* – go to http://localhost:5500/healthchecks. The system should have a healthy status.

If all clients are working as expected, you can now publish the container images to **Docker Hub**:

```
docker-compose push
```

Go to Docker Hub and make sure you have 7 images in your account:



We are done with **Docker**. Before continuing with the next step - stop the application:

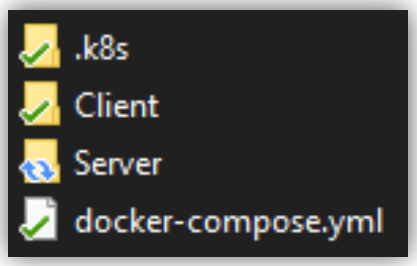
```
docker-compose down
```

Great! Now, let us move to **Kubernetes**!

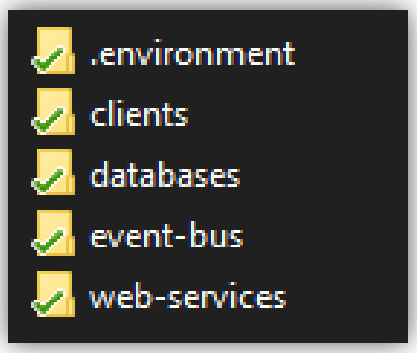
Note: This section's operations use a lot of space on your hard drive. After you finish the whole tutorial, you can delete everything with the "docker system prune --all" command.

4. Creating the Kubernetes Folder Structure

Create a "*.k8s*" folder right next to the "*docker-compose.yml*" file:



In it, add the following folder structure:



Here is how we are going to separate our configuration files:

- *.environment* – a special folder used only locally for our development environment.
- *databases* – all database configuration files will be stored in this folder - *Identity Database*, *Dealers Database,* and *Statistics Database*.
- *event-bus* – the event bus configuration files will be here.
- *web-services* – this folder will contain our web services' configuration files – *Identity Service*, *Dealers Service*, *Statistics Service,* and *Notifications Service*.
- *clients* – all clients' configuration files will be here – *User Client*, *Admin Client,* and *Watchdog Client*.

We are going to store all environment-specific configuration files in the "*.environment*" folder. It will contain locally used connection strings, settings, and secrets.

This folder will not be deployed to our production cluster. For it, we are going to use the imperative way of defining **Kubernetes** objects to make sure our application is secure.

It is time for our first configuration – databases!

5. Configuring the Databases

If we look at the application architecture diagram, we will see that the databases do not have any dependencies. So, we can start with their **Kubernetes** configuration!

Let us analyze the "*docker-compose.yml*" file, particularly the database section:

```
data:
    container_name: sqlserver
    image: mcr.microsoft.com/mssql/server:2019-latest
    ports:
        - "1433:1433"
    environment:
        - ACCEPT_EULA=Y
        - SA_PASSWORD=yourStrongPassword12!@
    volumes:
        - sqldata:/var/opt/mssql
    networks:
        - carrentalsystem-network
```

We can conclude that our databases need the following to run in containers:

- An external image – the official **SQL Server 2019** one.
- A specific port – "*1433".*
- Two environment variables – "*ACCEPT_EULA=Y*" and "*SA_PASSWORD*".
- A persistent volume – mapped to the "*/var/opt/mssql*" container folder.

*Note: The same information can be retrieved from the official image documentation **here**. If you use another database, you will need to read how to set it up. Different database providers have different configurations.*

We are going to create 3 database instances for each web service needing to store data. Each one of them will have an exclusive persistent volume and a different administrator password.

The first database will be the *Identity Database*. We are going to use dynamically provisioned volumes as they are easier to configure and scale better than the static ones. For this reason, we need a persistent volume claim.

Create an "*identity-database.pvc.yml*" file in the "*databases*" folder:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: identity-database-data
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Tip: You can open the whole ".k8s" folder in your favorite IDE. Visual Studio Code with the Kubernetes extension is a good option.

The access mode of our persistent volume claim is "*ReadWriteOnce*" because we will have a single replication of our database. We do not need multiple pods using the volume.

You may notice that we do not provide a storage class. So how are the volumes going to be provisioned? If no storage class is set in the persistent volume claim, the cluster will provide a volume from its default configuration:

- In a local environment - it will be the development machine's hard disk.
- In a production environment – it depends on the cloud or the cluster infrastructure.

You can see the default storage class in your cluster with the following command:

```
kubectl get sc
```

Let us now store the database administrator password in a **Kubernetes** secret.

Our locally used literal will be "*identityDatabasePassword12!@*". We will need to convert it in a *Base64* format. You can use **this online tool** or run a terminal command.

On **Mac/Linux**, you can run:

```
echo -n 'identityDatabasePassword12!@' | base64
```

On **Windows PowerShell**, you can run:

```
[convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes(
"identityDatabasePassword12!@"))
```

All options should produce the same encoded result:

```
aWRlbnRpdHlEYXRhYmFzZVBhc3N3b3JkMTIhQA==
```

Go to the "*.environment*" folder, and create a new file. Name it "*development.secret.yml*". Create a **Kubernetes** secret and put the encoded password in it:

```
apiVersion: v1
kind: Secret
metadata:
  name: environment-secrets
data:
  identity-database-sa-password: {the-encoded-password}
```

Next – we need a manifest file to deploy our pod. Since databases are stateful applications by nature, we are going to use a stateful set with 1 replica.

Create a "*identity-database.statefulset.yml*" file in the "*databases*" folder. First, we need to define our database name and labels:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: identity-database
spec:
  serviceName: identity-database
  selector:
    matchLabels:
      database: identity
  template:
    metadata:
      labels:
        database: identity
        system: database
```

The pod's template labels must contain the deployment selector's match labels property. Otherwise we are going to receive an error. The "*system*" label is there to help us filter our database pods later by entering:

```
kubectl get pods -l system=database
```

The above configuration does not specify a replica count. This is because the default value is 1.

Note: If you want to replicate your database on multiple pods and volumes, you will need to read the documentation on how to do it. Different database providers have different options and configurations.

Next, we need to define our container specifications – image, ports, and resources, to name a few:

```
# Full configuration skipped for brevity…
    metadata:
      labels:
        database: identity
        system: database
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: identity-database
        image: mcr.microsoft.com/msql/server:2019-latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 1433
        resources:
          limits:
            memory: "2Gi"
            cpu: "1000m"
```

We put a termination grace period because databases are usually heavy applications and need some time to shut down successfully. Additionally, the image is quite big, so we want to prevent unnecessary downloads by caching with the "*IfNotPresent*" policy. The resource limits here specify 1 CPU and 2 GiB of memory as an educational example only. These values depend on your business requirements, application usage, and scalability needs. Your infrastructure and available machines too, of course.

Next, we need to configure the environment variables needed by the database image:

- The license acceptance value will be provided as is directly.
- The administrator password will be retrieved from the secret we defined earlier – "*environment-secrets*".

```
# Full configuration skipped for brevity…
        resources:
          limits:
            memory: "2Gi"
            cpu: "1000m"
        env:
        - name: ACCEPT_EULA
          value: "Y"
        - name: SA_PASSWORD
          valueFrom:
            secretKeyRef:
              name: environment-secrets
              key: identity-database-sa-password
```

Finally, the volumes. We need to define a volume for the pod and then mount it in the container:

```
# Full configuration skipped for brevity…
        - name: SA_PASSWORD
          valueFrom:
            secretKeyRef:
              name: environment-secrets
              key: identity-database-sa-password
        volumeMounts:
        - name: identity-database-data
          mountPath: /var/opt/mssql
      volumes:
      - name: identity-database-data
        persistentVolumeClaim:
          claimName: identity-database-data
```

The mount path "*/var/opt/mssql*" comes from the **SQL Server 2019** documentation. Different images need different data paths to be mapped on a volume, so we need to do a little bit of research before we decide to add a ready-to-be-used container in our configuration.

Let us apply our configurations. Navigate your terminal to the "*.k8s*" folder and add our secrets first:

```
kubectl apply -f .\.environment\
```

Then apply the database configurations:

```
kubectl apply -f .\databases\
```

Let us inspect our database pod:

```
kubectl get pods
```

Unfortunately, our pod is not working. It says "*ImagePullBackOff*". We need to debug what is wrong:

```
kubectl describe pods {database-pod-name}
```

We can clearly see our error message – "*Failed to pull image*". Our image cannot be found in the **Docker Hub** registry. If we look at our configuration, we will see that we made a spelling mistake. Our image name is not correct, so we need to change it to:

```
image: mcr.microsoft.com/mssql/server:2019-latest
```

Fix it in your file and apply the new configuration:

```
kubectl apply -f .\databases\
```

Now, delete the previous pod and wait for the stateful set to initialize a new one:

```
kubectl delete pods {database-pod-name}
```

If we analyze the pod again, we will see it is up and successfully running this time. Let us analyze the container logs:

```
kubectl logs {database-pod-name}
```

You should see informational messages only. If that is the case – good job! Our first database configuration is done!

But before we continue, we need to expose the database in our cluster's internal network. Otherwise, the web servers will not be able to connect to it.

We have two options. Both are great for internal cluster connection. A "*ClusterIP*" service or a "*headless*" service. But what is the main difference?

A "*ClusterIP*" service is useful when we have many pod replicas and want to load balance between them.

A "*headless*" service does not allocate a new IP and it is perfect for 1-replica scenarios because it does not add a load-balancing overhead to the network. A "*headless*" service exposes the pod to the internal cluster network via the service name.

Creating one is easy. We just need to provide a "*None*" cluster IP value and **Kubernetes** will do the rest. Go to the "*databases*" folder and add a "*identity-database.service.yml*":

```
apiVersion: v1
kind: Service
metadata:
  name: identity-database
spec:
  clusterIP: None
  ports:
  - port: 1433
    targetPort: 1433
  selector:
    database: identity
```

The important parts here are the port mapping and the service selector. We expose the database port in the cluster network and make sure the service selects our pod by the "*database*" label.

Apply the service configuration:

```
kubectl apply -f .\databases\
```

Let us analyze the service. First, check whether it was created successfully:

```
kubectl get services
```

There should be a service named "*identity-database*". Our web servers will use this name to connect to the database.

Let us check the service endpoints too:

```
kubectl describe services identity-database
```

In the "*Endpoints*" section, there should be a random pod IP with our configured port. For example:

```
Endpoints: 10.1.1.105:1433
```

If this is the case – we successfully exposed our database to the cluster network. Our job here is done!

*Note: If you want to connect to your database externally (to make a backup, to query some data, to validate migrations, etc.), you can apply a load balancer service and expose the connection outside of the cluster temporarily. Just make sure you delete the service after you finish your database tasks. You can also use the "**kubectl expose**" command.*

Now it is your turn! Create the other two databases (*Dealers* and *Statistics*) by using the same steps we did in this section. Here are some hints:

- You add two new passwords in the "*environment-secrets*" (and apply them to the cluster). We want different databases to have different passwords in our system for extra security. Make sure you remember them for later in this tutorial. The passwords must have at least 8 characters including uppercase, lowercase letters, digits and/or non-alphanumeric symbols. Otherwise **SQL Server** will not start in your containers.

- You add new persistent volume claims to separate and isolate the data.
- You add two new services named "*dealers-database*" and "*statistics-database*". This is important because we are going to use the exact same names while setting up the web servers later in the tutorial.

When you are ready, your pods should look like this:

```
dealers-database-0
identity-database-0
statistics-database-0
```

And your services like this:

```
dealers-database
identity-database
kubernetes
statistics-database
```

You should also have three persistent volumes when you run:

```
kubectl get pv
```

If you have any difficulties, you can look at the provided final solution and its manifest files.

You are done? Wow, you are fast! Let us deploy a web server now!

Note: In a real-world application it may be a more suitable option to use a cloud-ready database like Azure SQL Database or Amazon RDS. Such providers have built-in replication, scalability, and automatic back-ups. They are way more expensive though.

6. Deploying a Web Server

We are going to configure the *Identity Service* first because it does not require a connection to the *Event Bus*. It will also allow us to validate the configurations we did for our databases.

Let us analyze the "*docker-compose.yml*" file again, particularly the *Identity Service* section:

```
identity:
    container_name: identity
    image: {image-name}
    build:
        context: ./Server
        dockerfile: {path-to-docker-file}
    ports:
        - "5001:80"
    env_file: Server/CarRentalSystem/Common.env
    environment:
        - {database-connection-string}
        - IdentitySettings__AdminPassword={password}
    restart: on-failure
    volumes:
        - {specific-volume-path}
    networks:
        - carrentalsystem-network
    depends_on:
        - data
```

These are the configuration details:

- The image is locally built, and its name comes from the **Docker Hub** registry. In my case, it is "*ivaylokenov/carrentalsystem-identity-service:1.0*". You will need to change "*ivaylokenov*" with your username.
- Our web server listens on port "*80*" so we need to expose that in the pod template.

- The *Identity Service* application uses a volume to store some data protection keys. The volume must be mapped to a specific folder - "*/root/.aspnet/DataProtection-Keys*".
- There are 4 environment variables. Two of them are in the "*Common.env*" file, and the other two – in the **Docker Compose** file directly.

Let us first configure the environment variables.

The "*Common.env*" contains these entries:

```
ApplicationSettings__Secret={secret}
ApplicationSettings__SeedInitialData=True
```

Additionally, the "*docker-compose.yml*" specifies two more:

```
{database-connection-string}
IdentitySettings__AdminPassword={password}
```

Three of them contain secret data, so we need to add them to our "*development.secret.yml*" configuration.

The "*ApplicationSettings__SeedInitialData*" entry does not contain any security-related information so we can add it to a configuration map.

Our application uses the "*ApplicationSettings__Secret*" variable to issue and validate security tokens for authentication and authorization purposes. Here is the value in plain text:

```
S0M3 M4G1C UN1C0RNS G3N3R4T3D TH1S S3CR3T
```

We need to convert the whole string to *Base64* format and add it to our local secrets:

```
  dealers-database-sa-password: {database-secret}
  statistics-database-sa-password: {database-secret}
  security-token-secret: {security-token-secret}
```

Here is how we can convert plain text to *Base64* one more time to make sure the concept is clear.

On **Mac/Linux**, you can run:

```
echo -n 'S0M3 M4G1C UN1C0RNS G3N3R4T3D TH1S S3CR3T' |
base64
```

On **Windows PowerShell**, you can run:

```
[convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes(
"S0M3 M4G1C UN1C0RNS G3N3R4T3D TH1S S3CR3T"))
```

Both options should produce the same encoded result:

```
UzBNMyBNNEcxQyBVTjFDMFJOUyBHM04zUjRUM0QgVEgxUyBTM0NSM1Q=
```

Do the same for the "*IdentitySettings__AdminPassword*" variable. In my example the admin password is:

```
adminpass12!
```

Here it is in *Base64* format:

```
YWRtaW5wYXNzMTIh
```

Add it to the secret:

```
  statistics-database-sa-password: {database-secret}
  security-token-secret: {security-token-secret}
  admin-password: {admin-password-secret}
```

The database connection string is a bit trickier. We need to provide our service name and our password in plain text. The variable name is "*ConnectionStrings__DefaultConnection*" and this is the desired value format:

```
Server={server-name};Database={database-name};User Id=sa;
Password={server-password};MultipleActiveResultSets=true
```

The server name should be our database service name – "*identity-database*". The database name is irrelevant, and we can provide whatever we like here. The server password is the one we chose before - "*identityDatabasePassword12!@*".

Here is the actual connection string in plain text:

```
Server=identity-
database;Database=CarRentalIdentityDatabase;User Id=sa;
Password=identityDatabasePassword12!@;MultipleActiveResul
tSets=true
```

Let us convert it to *Base64* format. Once again, we need to transform the whole string:

```
U2VydmVyPWlkZW50aXR5LWRhdGFiYXNlO0RhdGFiYXNlPUNhclJlbnRhb
ElkZW50aXR5RGF0YWJhc2U7VXNlciBJZD1zYTsgUGFzc3dvcmQ9aWRlbn
RpdHlEYXRhYmFzZVBhc3N3b3JkMTIhQDtNdWx0aXBsZUFjdGl2ZVJlc3V
sdFNldHM9dHJ1ZQ==
```

Add it to our local secrets:

```
  security-token-secret: {security-token-secret}
  admin-password: {admin-password-secret}
  identity-service-connection-string: {connection-string}
```

Note: In a real-world application, you may want to split your secrets into multiple files to have a better separation of concerns. It is not necessary, though. It depends on how you like to organize your projects.

Let us add a configuration map. We will store non-security related settings in it like the "*ApplicationSettings__SeedInitialData*" variable, for example.

Go to the "*.environment*" folder and create a new file. Name it "*development.configmap.yml*", and add the following:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: environment-settings
data:
  seed-initial-data: "True"
```

The name of our configuration map will be "*environment-settings*", and we do not need to encode the environment variable value in a *Base64* format. We do not want to provide the literal "*True*" directly in our deployment template, because this value usage spans multiple web services.

It is time to create our deployment. Go to the "*web-services*" folder and add a "*identity-service.deployment.yml*" file.

We will start with the usual object definition and labels:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: identity-service
spec:
  selector:
    matchLabels:
      web-service: identity
  template:
    metadata:
      labels:
        web-service: identity
        system: server
```

Since our web servers are stateless in nature, we are using a "*Deployment*" **Kubernetes** object. Once again, the deployment selector must match at least one of the pod's labels.

Next is our container specification. You will need to provide your own image from your **Docker Hub** registry.

```
# Full configuration skipped for brevity…
    metadata:
      labels:
        web-service: identity
        system: server
    spec:
      containers:
      - name: identity-service
        image: {your-identity-service-image}
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        resources:
          limits:
            memory: "512Mi"
            cpu: "250m"
```

Since we are going to replicate this pod in the future, the resource limits should be smaller. This will allow us to scale more granularly when we need to do so. Of course – as we said earlier – the values provided here are pure educational examples.

Let us add the environment variables now. Three of them come from our secrets. Only the *"ApplicationSettings__SeedInitialData"* should be retrieved from the configuration map:

```
# Full configuration skipped for brevity…
        resources:
          limits:
            memory: "512Mi"
            cpu: "250m"
        env:
        - name: ApplicationSettings__Secret
          valueFrom:
            secretKeyRef:
              name: environment-secrets
              key: security-token-secret
        - name: ApplicationSettings__SeedInitialData
          valueFrom:
            configMapKeyRef:
```

```
            name: environment-settings
            key: seed-initial-data
        - name: ConnectionStrings__DefaultConnection
          valueFrom:
            secretKeyRef:
              name: environment-secrets
              key: identity-service-connection-string
        - name: IdentitySettings__AdminPassword
          valueFrom:
            secretKeyRef:
              name: environment-secrets
              key: admin-password
```

It is time to apply the deployment! The moment of truth!

Apply the new environment variables:

```
kubectl apply -f .\.environment\
```

Then the web server:

```
kubectl apply -f .\web-services\
```

Analyze our web server pods:

```
kubectl get pods -l system=server
```

There should be one pod without any errors. We should see its logs to validate the database connection:

```
kubectl logs {your-identity-service-pod-name}
```

Well, there are no database exception messages but there are others! We forgot the persistent volume! We will add it in a minute but let us first check the database logs too:

```
kubectl logs identity-database-0
```

We can clearly see in the last few lines that the database was created successfully. Nicely done!

Let us configure the volume. Create an "*identity-service.pvc.yml*" file in the "*web-services*" folder:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: identity-service-data
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Mi
```

Since the data protection keys have a very small size, we only request 10 MiB of storage.

Now, mount the volume in our deployment manifest:

```
# Full configuration skipped for brevity…
        - name: IdentitySettings__AdminPassword
          valueFrom:
            secretKeyRef:
              name: environment-secrets
              key: admin-password
        volumeMounts:
        - name: identity-service-data
          mountPath: /root/.aspnet/DataProtection-Keys
      volumes:
      - name: identity-service-data
        persistentVolumeClaim:
          claimName: identity-service-data
```

Reapply the deployment:

```
kubectl apply -f .\web-services\
```

If you analyze the cluster pods, you will see the old web server terminating, and a new one starting. If you look at the new logs, you

will see that one of the previous warnings is no longer there. The volume did its job!

The other warning is **ASP.NET Core** specific and requires a certificate installation on the server. More information is available **here** and **here**. Since this operation is outside of the tutorial's scope, we are going to leave it as is.

We need to do one final step – expose the web server via a load balancer service.

Create a "*identity-service.loadbalancer.yml*" file in the "*web-services*" folder:

```
apiVersion: v1
kind: Service
metadata:
  name: identity-service
spec:
  type: LoadBalancer
  ports:
  - port: 5001
    targetPort: 80
  selector:
    service: identity
```

We use the same port which was used in the "*docker-compose.yml*" configuration – "*5001*".

Apply the new configuration:

```
kubectl apply -f .\web-services\
```

Let us validate our endpoints configuration:

```
kubectl describe services identity-service
```

Unfortunately, the endpoints section is empty. This means our configuration is not valid. More specifically, our selectors are wrong. We need to change them to match the pod's labels like so:

```
  selector:
    web-service: identity
```

Reapply the configuration after you fix the issue and check the endpoints again. Our load balancer should be valid now!

Let us test our service. Open an *API* client (like **Postman**), and send a *POST* request to the following *URL*:

```
http://localhost:5001/Identity/Login
```

Use the following *JSON* body to authenticate with the preseeded user:

```
{
    "email": "coolcars@crs.com",
    "password": "coolcars12!"
}
```

You should receive a *200 OK* response containing an authentication token.

Perfect! You rock! Next – our *Event Bus*!

Note: If you receive an error and the cluster is not working as expected, you can look at the provided final solution and its manifest files.

7. Adding the Event Bus

It is time to look at the "*docker-compose.yml*" file once again. This time it is all about the "*messages*" section:

```
messages:
    container_name: rabbitmq
    image: rabbitmq:3-management
    ports:
        - "5672:5672"
    hostname: "rabbitmq"
    environment:
        - RABBITMQ_DEFAULT_USER=rabbitmquser
        - RABBITMQ_DEFAULT_PASS=rabbitmqPassword12!
    volumes:
        - rabbitmq:/var/lib/rabbitmq/mnesia
    networks:
        - carrentalsystem-network
```

You should be quite confident with **Kubernetes** configuration by now. You will try to configure the *Event Bus* by yourself. Here are the necessary steps:

- You need to put the "*RABBITMQ_DEFAULT_USER*" environment variable in the configuration map. Name the key "*event-bus-user*".
- You need to put the "*RABBITMQ_DEFAULT_PASS*" environment variable in the secret's configuration. Its value must be *Base64* encoded. Name the key "*event-bus-password*".
- You need a big persistent volume to store the message broker's data. 5 GiB is a good start. As specified in the configuration above - you should mount the "*/var/lib/rabbitmq/mnesia*" folder.
- You need a stateful set because event buses are stateful applications by nature. Do not make multiple replications, because such configuration is more advanced. For the resource limits – put 0.5 CPU and 1 GiB of memory. Usually, message

queues are asynchronous, so we would want to give them more than one processor, but our values are purely educational.

- You need a headless service to expose the container port internally in the cluster. Name the service "*event-bus*".

*Note: The RabbitMQ image has a preinstalled plugin for managing the message queue via a web portal. If you want to test it, you will need to expose one more port – "15672". Additionally, you will need a load balancer service mapped to that port to access the portal outside of the cluster. You can also use the "**kubectl expose**" command.*

This is pretty much it. Try to implement these steps and if you have any troubles, you can look at the provided final solution and its manifest files. Do not forget to apply the new environment variables before deploying the event bus configuration:

```
kubectl apply -f .\.environment\
kubectl apply -f .\event-bus\
```

*Note: The provided solution is an educational example and it is as minimalistic as possible. While it will serve you well in both development and production scenarios, it is a good idea to check the official **RabbitMQ documentation regarding Kubernetes** clusters, especially if you want multiple replications of your event bus.*

Done? I like you! Such are a fast learner! Let us put this event bus into action now!

8. Connecting the Notifications Service

Take a look at the *Notifications Service* configuration in the "*docker-compose.yml*" file. Deploying it should be a straight-forward process.

Add the two new environment variables to our configuration map:

- "*NotificationSettings__AllowedOrigins*" – the value here is "*http://localhost*". We will need to change it later when we deploy our solution to a production **Kubernetes** cluster.
- "*MessageQueueSettings__Host*" – the value of this variable must be the name of our message queue service – "*event-bus*" in our case. Otherwise, our web servers will not be able to connect.

The event bus's username and password are already present.

This is how our configuration map should look like:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: environment-settings
data:
  seed-initial-data: "True"
  event-bus-host: event-bus
  event-bus-user: rabbitmquser
  notifications-allowed-origins: http://localhost
```

Create a persistent volume claim for the data protection keys. As with the *Identity Service*, it should be a small one – 10 MiB.

Create a load balancer service and expose the "*5004*" port. Make sure you fill in the correct label selectors.

Create a deployment manifest file and configure a single replica pod. Do not forget to mount the persistent volume.

You will need 5 environment variables:

```
        env:
        - name: ApplicationSettings__Secret
          valueFrom:
            secretKeyRef:
              name: environment-secrets
              key: security-token-secret
        - name: NotificationSettings__AllowedOrigins
          valueFrom:
            configMapKeyRef:
              name: environment-settings
              key: notifications-allowed-origins
        - name: MessageQueueSettings__Host
          valueFrom:
            configMapKeyRef:
              name: environment-settings
              key: event-bus-host
        - name: MessageQueueSettings__UserName
          valueFrom:
            configMapKeyRef:
              name: environment-settings
              key: event-bus-user
        - name: MessageQueueSettings__Password
          valueFrom:
            secretKeyRef:
              name: environment-secrets
              key: event-bus-password
```

When you are ready, apply the new environment variables and then the new web service:

```
kubectl apply -f .\.environment\
kubectl apply -f .\web-services\
```

Wait a little bit and analyze your pods:

```
kubectl get pods
```

All of them should be up and running. Let us validate the event bus connection:

```
kubectl logs {your-event-bus-pod-name}
```

At the end of the log you should see that our *Notifications Service* has connected successfully:

```
connection <0.2880.0> (10.1.1.192:42598 ->
10.1.1.189:5672 - CarRentalSystem.Notifications): user
'rabbitmquser' authenticated and granted access to vhost
'/'
```

Of course, your IPs and ports may have other values.

You may notice that the new pod restarted once. Even if it did not on your machine, let us analyze it:

```
kubectl describe pod {your-notifications-service-pod-
name}
```

One of the events may show the following:

```
persistentvolumeclaim "{your-pvc-name}" not found
```

This is because **Kubernetes** tries to create the deployment before its persistent volume claim is available. We should fix that.

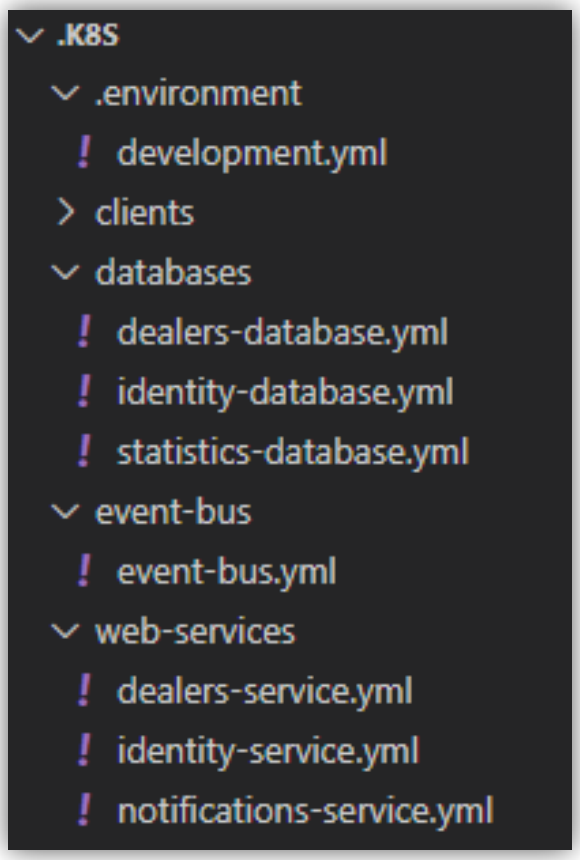
It is a good idea to configure related objects in a single file. You can do it like so:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: environment-settings
data:
  # Data skipped for brevity…

---

apiVersion: v1
kind: Secret
metadata:
  name: environment-secrets
data:
  # Data skipped for brevity…
```

Let us refactor a bit. Merge all related objects until you have the following folder structure:



The configuration map and the secret should be in a single "*development.yml*" file now.

All pod manifests should have the following order – first the persistent volume claim, then the service, and finally the pod itself:

```
# Configuration is shortened for brevity…
kind: PersistentVolumeClaim

---

kind: Service

---

kind: Deployment
```

You can reapply everything to validate the cluster configuration is still valid:

```
kubectl apply -f .\.environment\
kubectl apply -f .\databases\
kubectl apply -f .\event-bus\
kubectl apply -f .\web-services\
```

Your cluster still works? Excellent! One more section with two more configurations and our backend is ready!

9. Finalizing the Backend Microservices

This tutorial section does not have any instructions. You need to configure and deploy the *Dealers Service* and the *Statistics Service* all by yourself.

Of course, if you need assistance, you can look at the provided final solution, but try to write the **YAML** files without any external help.

It should be fairly easy. The configuration is identical to the *Identity Service*'s one. Just be careful with the new connection strings and the passwords in them. All the other environment variables are already configured and are available for you to reuse.

*Note: You may start to realize that configuration files are becoming repetitive. A viable solution is to use template engines like **yq** and **kustomize**.*

You can test the new web servers when you are done deploying their pods. Open an *API* client (like **Postman**), and send a *GET* request to the following *URLs*:

```
http://localhost:5002/CarAds
http://localhost:5003/Statistics
```

Both endpoints should return a *200 OK* status code with data in a *JSON* format.

Enough with the backend! It is time for our clients!

10. Visualizing the User Client

By deploying the user client, we will confirm that our backend is working correctly, and all the network connections are up and running.

The angular application is the easiest to configure. We need a load balancer service, which listens to the default port – *"80"* – and a deployment of our client image.

This is our load balancer:

```
apiVersion: v1
kind: Service
metadata:
  name: user-client
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    web-client: user-client
```

And this is the first part of the deployment:

```
# Full configuration skipped for brevity…
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-client
spec:
  selector:
    matchLabels:
      web-client: user-client
  template:
    metadata:
      labels:
        web-client: user-client
        system: client
```

You need to add the container properties by yourself. Put both configurations in a "*user-client.yml*" file in the "*clients*" folder.

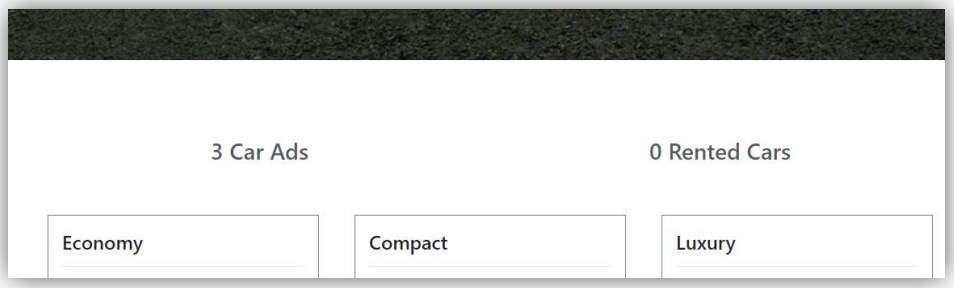
Now, the moment of truth! Apply the new configuration:

```
kubectl apply -f .\clients\
```

Open your browser and navigate it to **http://localhost**.

Note: If you start the client right after you apply the web servers, you will need to wait a bit before the data is shown. Each backend API needs 5-10 seconds to bootstrap, especially if the databases are fresh too.

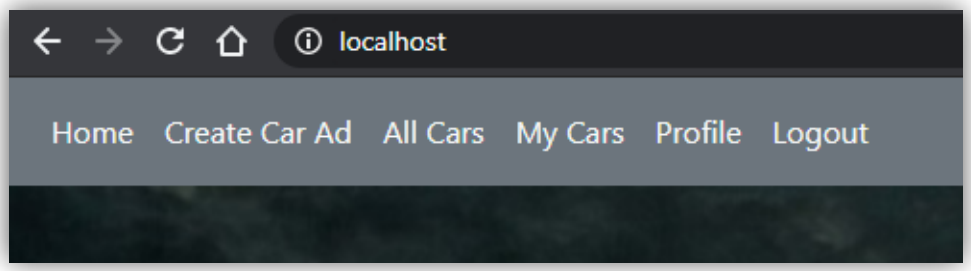
On the home page, you should see 3 car ads in the statistics section below the image slider.



Open the developer tools and check the **JavaScript** console – it should not show any errors, only informational messages.

Go to the login page and authenticate with the preseeded user – "*coolcars@crs.com*" e-mail and "*coolcars12!*" password.

The application should successfully authorize you to add new car ads.



Click on "*Create Car Ad*" and fill the following information:

- Manufacturer – "*BMW*"
- Model – "*X6 M*"
- Category – "*SUV*"
- Image URL – "*https://s.aolcdn.com/dims-global/dims3/GLOB/legacy_thumbnail/640x400/quality/80/https://s.aolcdn.com/commerce/autodata/images/USD00BMS231A01300.jpg*"
- Price Per Day – "*40*"
- Climate Control – "*Yes*"
- Number of Seats – "*5*"
- Transmission Type – "*Automatic*"

After you create the car ad, you should see a green notification in the top right corner of the page:

The above message confirms our *Notifications Service* is working as expected.

Go to the home page and check the statistics again. They should show 4 car ads in the system now. If this is the case – congratulations! The *Event Bus* is also up and running!

Great job! Our cluster is configured properly! Let us now finalize the whole application by deploying the other two clients!

11. Administrating Our Application

You are completely capable of configuring the last two clients on your own.

You just need to be careful with the networking details. The endpoints in the cluster depend on your services' names. If you followed the tutorial by the letter, this configuration map should do the trick for you:

```
notifications-allowed-origins: http://localhost
admin-identity-endpoint: http://identity-service:5001
admin-dealers-endpoint: http://dealers-service:5002
admin-statistics-endpoint: http://statistics-service:5003
watchdog-identity-health: http://identity-service:5001/health
watchdog-dealers-health: http://dealers-service:5002/health
watchdog-statistics-health: http://statistics-service:5003/health
watchdog-notifications-health: http://notifications-service:5004/health
watchdog-admin-health: http://admin-client:5000/health
```

Expose the administration client to port "*5000*", and the watchdog client to "*5500*".

Open your browser and navigate it to **http://localhost:5000**. Try to authenticate with the preseeded administrator user – "*admin@crs.com*" e-mail and "*adminpass12!*" password. If you see the correct statistics – the client works.

Note: You may receive an error when you try to log in. If the logs of the administration client logs show an "antiforgery decryption" error, you will need to clear the browser's cookies for the "localhost" domain. It may still store the ones used during the Docker Compose run, and they are invalid for the new Kubernetes deploy.

The watchdog is available on **http://localhost:5500/healthchecks**. All health checks should be green.

Finally, we are done! As the last step, we should deploy the cluster on a cloud infrastructure and enjoy our application live on the **Internet**!

12. Publishing to a Cloud Cluster

If you made it this far, you are hero! A true legend! You should be proud of yourself! You learned a lot! Congratulations!

Now, let us try an actual cluster deployment and move away from the development environment.

You can publish the application on every public cloud platform. **Kubernetes** is **Kubernetes** and our configuration should run everywhere.

Create a cluster with a minimum of 16 CPUs and 32 GB of memory because we are going to replicate some of our deployments.

Connect to the cloud cluster and validate that you are running on the correct environment:

```
kubectl config current-context
```

Create a "*production.yml*" file in the "*.environment*" folder and copy everything from "*development.yml*".

Important! In a real-world scenario, this new file should not be committed to the source control repository of your project and should be secured properly.

All the other configuration files should be stored in version control before being pushed to the cluster. This allows you to quickly roll back a configuration change if necessary. It also aids cluster re-creation and restoration.

Additionally, we need to change all secrets and passwords in the production environment but for the sake of educational simplicity, we are going to leave them exactly the same.

The "*notifications-allowed-origins*" setting is currently pointing to "*localhost*" and we need to update it with the valid production value.

If we developed an actual web application, we would have a domain name, and it will be easy to specify it here. However, we do not have one so we should create the user client service upfront in order to get its host IP:

```
kubectl create service loadbalancer user-client --
tcp=80:80
```

Run the following to edit the service:

```
kubectl edit service user-client
```

Fix the selector to match our future *User Client* pod:

```
# Full configuration skipped for brevity…
  selector:
    web-client: user-client
  sessionAffinity: None
```

Close and save the file. Wait a bit and analyze the new service:

```
kubectl get services
```

It should have an external IP. If not – wait a bit more and try again. Copy the value and put in the "*production.yml*" file. In my case, the IP is "*35.238.191.128*":

```
notifications-allowed-origins: http://35.238.191.128
```

In an actual production environment, we should change the "*seed-initial-data*" value to "*False*", because we do not want test data on our live application, but here we are playing around, so leave it as is. Preseeded data will help us validate the cloud cluster configuration.

Apply the production configuration:

```
kubectl apply -f .\.environment\production.yml
```

We need to give our databases some file system permissions. Add the following to their configuration files:

```
# Full configuration skipped for brevity…
      terminationGracePeriodSeconds: 60
      securityContext:
        fsGroup: 10001
      containers:
      - name: statistics-database
        image: mcr.microsoft.com/mssql/server:2019-latest
        imagePullPolicy: IfNotPresent
```

Then apply the databases and the event bus:

```
kubectl apply -f .\databases\
kubectl apply -f .\event-bus\
```

Let us boost them a bit. Run the following command:

```
kubectl edit statefulset identity-database
```

Update the container limits:

```
        resources:
          limits:
            cpu: "2"
            memory: 4Gi
```

Close the file and save it. Do the same for the other two databases and the event bus. Of course, these values are just for educational purposes. On a real production server, you may need more resources. We do not update the **YAML** files directly because we do not want to break our development environment.

Next, we want to deploy the web servers and scale them a bit. However, our persistent volume claims will not allow us. Edit the "*identity-service-data*", "*dealers-service-data*", "*statistics-service-data*", and "*notifications-service-data*" configurations:

```
# Full configuration skipped for brevity…
spec:
  accessModes:
  - ReadWriteOnce
  - ReadOnlyMany
  resources:
    requests:
```

This way our first pod will create the data protection keys, and all the replicas will read them.

But this creates another problem. Each persistent volume is scheduled once on a single node. For one pod to access it, it needs to be on the same node. So, we need to make sure our pods are scheduled on the same machine.

How to do that? Enter pod affinity!

Open the "*dealers-service.yml*" file and add the following:

```
# Full configuration skipped for brevity…
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchLabels:
                web-service: dealers
            topologyKey: "kubernetes.io/hostname"
      containers:
```

The above configuration tells **Kubernetes** to schedule all pods with a label "*web-service: dealers*" on the same node.

Note: In a real-life application our pods should be spread across multiple nodes to protect them from machine failure. However, such a scenario requires more complex persistent volume configurations, which we want to avoid in this educational example.

Add the settings in the other web servers too, but make sure you double-check the labels. They should match for every deployment!

Now, apply the folder:

```
kubectl apply -f .\web-services\
```

It is time for scaling! We do not want to edit our **YAML** file once again, so we are going to do it imperatively through the terminal.

Run the following command to update the *Dealers Service*:

```
kubectl scale deployments/dealers-service --replicas=4
```

Repeat the operation for the other three services as well.

Analyze your pods and validate the replication was successful:

```
kubectl get pods -l system=server
```

You should have 4 replicas for each web API. We can even inspect a pod's logs to see it running correctly. The warnings we had on our development environment should not be present in the cloud because the latter has certificates installed.

Before we deploy our clients, we need to configure the endpoints of the **Angular** application. Go to "*Client**src**environments*". You need to get your load balancers' external IPs and put them in the "*environment.prod.ts"* file:

```
kubectl get services
```

In my case, the configuration looks like this:

```
export const environment = {
  production: true,
  identityApiUrl: 'http://34.122.74.94:5001/',
  dealersApiUrl: 'http://35.184.122.109:5002/',
  statisticsApiUrl: 'http://35.223.2.95:5003/',
  notificationsUrl: 'http://35.202.99.212:5004/'
};
```

*Note: In a real-life application, you will have domain addresses in front of these IPs. "dealers.carrentals.com" and "statistics.carrentals.com", for example. Additionally, you may put an **Ingress** service as a reverse proxy to route the client requests to your internal cluster network of load balancers.*

We need to prepare a production *User Client* image with the new configuration. Navigate your terminal to the "*Client*" folder and run:

```
docker build -t {your-docker-hub-username}/carrentalsystem-user-client-optimized:1.0 .
```

Wait a bit, then push the image:

```
docker push {your-docker-hub-username}/carrentalsystem-user-client-optimized:1.0
```

Navigate the terminal back to the "*.k8s*" folder and deploy our clients:

```
kubectl apply -f .\clients\
```

Then replace the *User Client* image with the optimized one:

```
kubectl set image deployments/user-client user-client={your-docker-hub-username}/carrentalsystem-user-client-optimized:1.0
```

The pod should now automatically update its image. You may add more replicas here too:

```
kubectl scale deployments/user-client --replicas=4
```

Go to your load balancer service IP and see your application live! Play with it a bit. Register a user, then add a car ad or two…

The administration and watchdog clients are also up and running!

Magic!

Congratulations! Well done!

YOU. SIMPLY. ROCK!!!

Want More Kubernetes?

Here are the next steps. You can try executing them on the cluster:

- **Pod health** - configure liveness, readiness, and startup probes.
- **Rolling Update** – configure zero-down deployments.
- **Autoscalers** – learn about horizontal and vertical autoscaling.
- **Ingress** – learn about the reverse proxy and its configuration.
- **Annotations** – learn how to add metadata to your pods.
- **Init Containers** – understand the concept of initialization containers.
- **Node Affinity** – understand how node affinity works in detail.
- **Monitoring** – add advanced monitoring tools like **Prometheus** and **Grafana**.
- **Service Accounts** – learn about cluster authorization.
- **Troubleshooting** – a visual guide on **Kubernetes** troubleshooting.
- **Kubectl Commands** – have a better understanding of the command-line management.
- **Configuration Best Practices** **–** learn how to configure applications the right way.
- **Production Best Practices** – learn how to avoid disasters.
- **And Many More** **–** Kubernetes is endless…

Good Luck and Have Fun with your Kubernetes journey!