ISSUE 48 OCT 2020

DNC MAGAZINE

www.dotnetcurry.com

MICROSERVICES ARCHITECTURE PATTERN

Tic Tac Toe in F#

Forms in React.JS

Kubernetes for Developers

Artificial Intelligence: *What, Why and How*

React.JS Application Lifecycle Events

ARCHITECTURE OF WEB APPLICATIONS

EDITOR'S NOTE

@suprotimagarwal

Editor in Chief

Dear Readers,

I want to begin by thanking you for the amazing response for our 8th Anniversary edition! Over 120K developers have downloaded the magazine so far, making it all worthwhile for the team!

ReactJS is one of the most loved UI libraries! In our 48th edition, Mahesh discusses the lifecycle of React class components while Ravi demonstrates how to create React Forms using Functional components and Hooks.

Subodh charts the rise of Microservices, a variant of the service-oriented architecture (SOA) while Daniel demonstrates the why and how of Kuberenetes while spiking your interest to explore it further.

In our Application Architecture series, Damir builds upon his previous tuts and talks about Architecting Web Applications in this edition. Yacoub begins a new series for those interest-ed in F#. If you have enjoyed his Patterns and Practices section so far, you will love this too!

This edition rounds off with the next part of our Machine Learning series by Benjamin and does a deep dive into understanding Artifical Intelligence from ground up.

So how was this edition and what would you like to learn in the upcoming editions? E-mail me at suprotimagarwal@dotnetcurry.com.

Editor In Chief :
Suprotim Agarwal
(suprotimagarwal@dotnetcurry.com)

Art Director :
Minal Agarwal

Contributing Authors :
Yacoub Massad
Subodh Sohoni
Ravi Kiran
Mahesh Sabnis
Daniel Jimenez Garcia
Damir Arh
Benjamin Jakobus

Technical Reviewers :
Benjamin Jakobus
Damir Arh
Dobromir Nikolov
Gouri Sohoni
Ravi Kiran
Subodh Sohoni
Suprotim Agarwal

Next Edition :
Jan 2021

a2z | Knowledge Visuals

et curry.com

CONTENTS

ImageGear PDF

Programmatically process and manipulate PDFs inside your application.

ImageGear PDF has a variety of PDF functionalities that you can integrate into your application. When you integrate ImageGear PDF into your application, you are giving users the ability to:

- Merge Multiple PDFs
- Split a PDF into Multiple PDFs
- Rearrange Pages Within a PDF
- Add Pages or Remove Pages in a PDF
- Convert a File or Compress a File

The SDK adds programmatic annotation capabilities as well.

1. Compress Large Files
2. Add a Digital Signature
3. Compare Different Documents
4. Capture Data Inside a PDF
5. Add-On Optical Character Recognition

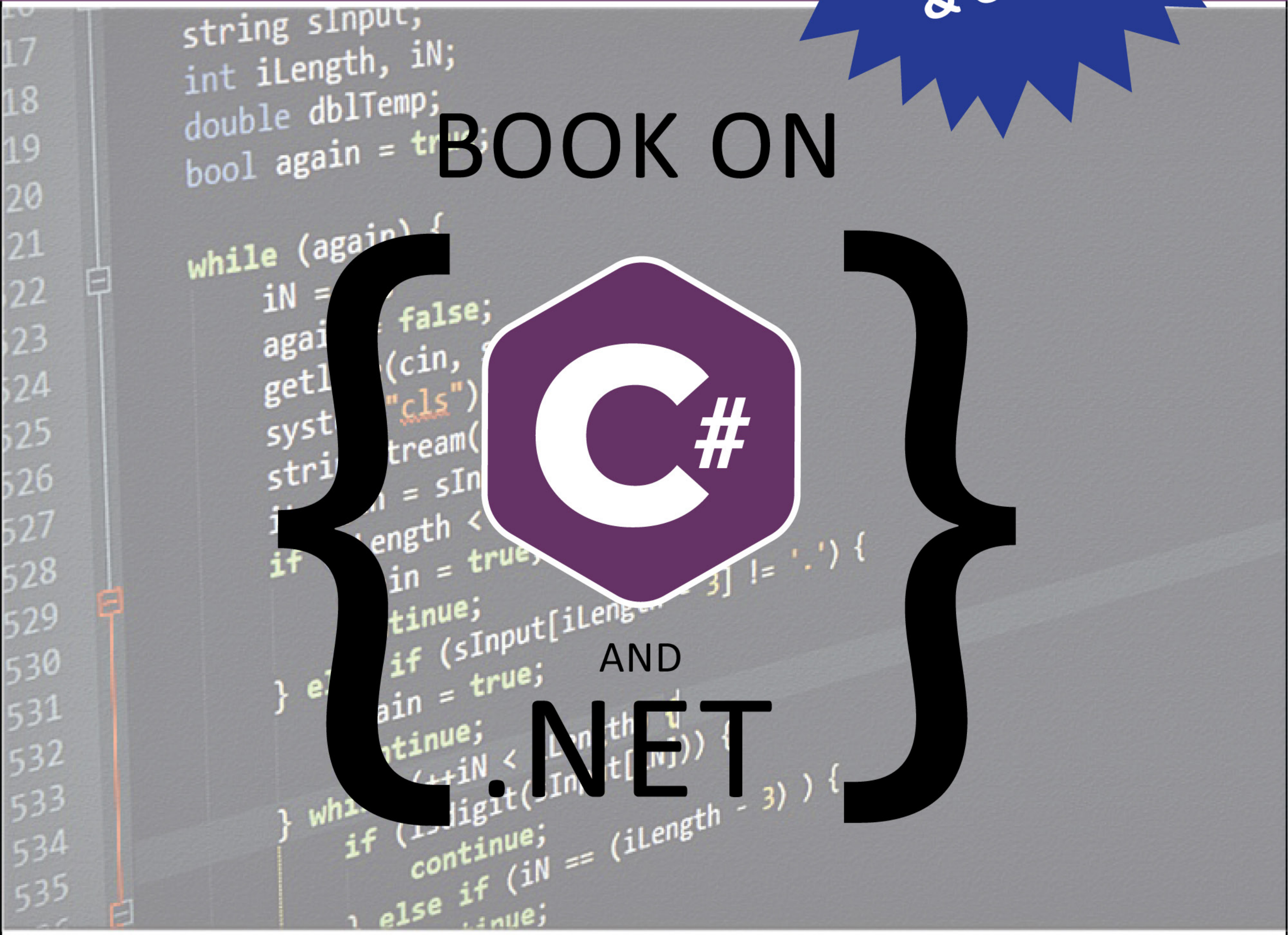
Get your free trial today.

Download Image Gear PDF

COVERS C# v6, v7 AND .NET Core

THE ABSOLUTELY AWESOME BOOK ON C# AND .NET

NOW INCLUDES CHAPTERS ON **.NET Core 3.0 & C# 8.0**



DAMIR ARH

Subodh Sohoni

Microservices Architecture Pattern

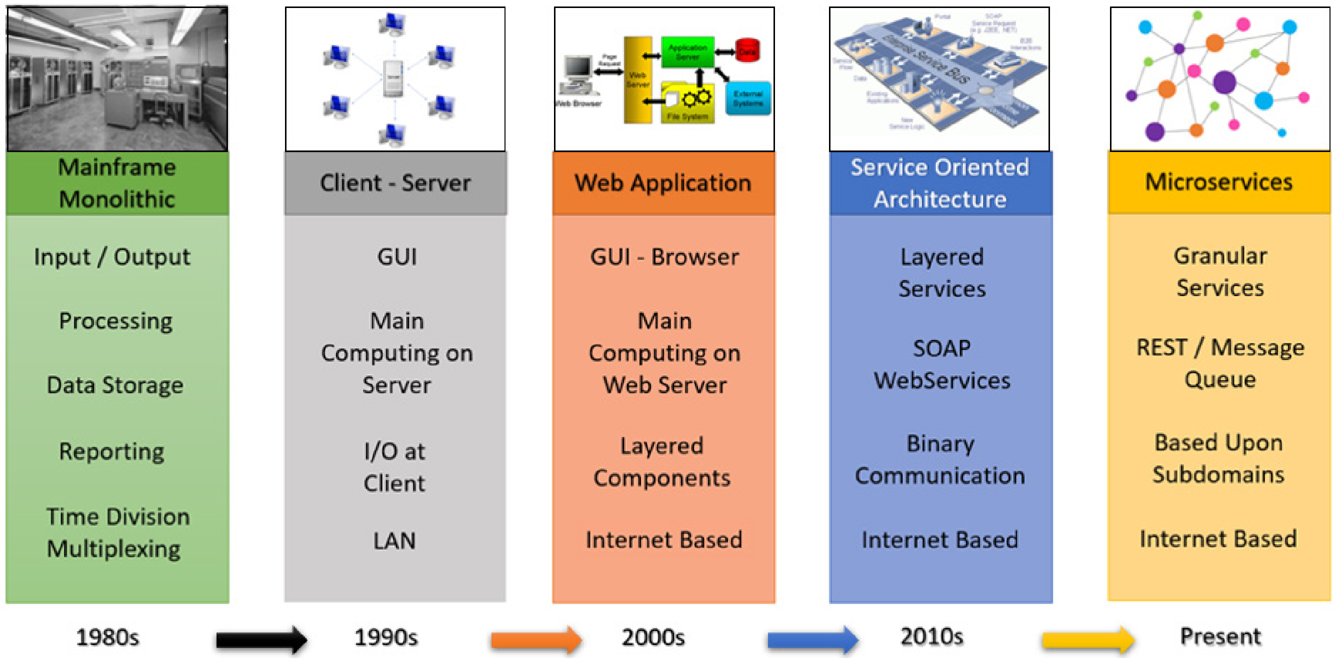
This article is about the **architectural pattern of Microservices**. We will learn **how the microservices architecture pattern evolved**, what are the **benefits** of microservices architecture pattern and an **overview of the evolution process** of microservices architecture.

We will also understand some implementation details of microservices. These details will include the **access methods**, **communication patterns** and an **introduction to containers**.

History of Architecture Patterns in Software

To learn about the microservices architecture, let's take a peek into the history of various patterns of software architecture. We need to understand why did they evolve in progression over time and what were the conditions that were conducive for each of those patterns!

Only then will we be able to understand the reasons of **why Microservices architecture is the most suitable one for modern software**.



Mainframe Monolithic	Client - Server	Web Application	Service Oriented Architecture	Microservices
Input / Output	GUI	GUI - Browser	Layered Services	Granular Services
Processing	Main Computing on Server	Main Computing on Web Server	SOAP WebServices	REST / Message Queue
Data Storage	I/O at Client	Layered Components	Binary Communication	Based Upon Subdomains
Reporting	LAN	Internet Based	Internet Based	Internet Based
Time Division Multiplexing				
1980s	1990s	2000s	2010s	Present

Monolithic Architecture of 1980s

When computing became popular, we used to have monolithic applications on mainframe computers. I remember the days back in the 80s when I used to work in a big corporate organization, where we had one of the most powerful computers of those days. It was a mainframe computer. Despite being a powerful computer, I had to wait for hours to receive the output of a program I was running. If the program had a bug, I would come to know about it only after a couple of hours, make the corrections and wait for a couple of more hours before getting the correct output.

In such applications, everything including the hardware, was integrated. The user interface, business logic, utility programs and even the database was totally integrated into the application itself.

The main issue was the scarcity of resources.

Since the entire hardware and software was shared by many users and background processes, it was very inefficient!

Client-Server Architecture

After spending a few years in the industry, I got introduced to a new architecture at that time. It was called the "client-server" architecture.

In this architecture, we could have a nice GUI to run an application which resided on our own machine; however, the main computing was done on a server somewhere in the main office of our organization.

Although this architecture was much more efficient compared to the mainframe computer and the monolithic applications that we were used to, it still was not that fast when compared to the earlier software. I remember sometimes we used to take a few minutes to get the output from the server, before it could reach the client application.

An irritant about this architecture was that it required us to install the client application on *each and every* user's machine. And if any changes needed to be made in the client application, then we had to start the installation process on each users' machine all over again. Sometimes, the only way to find out the application is no longer working on a client machine would be when the users started complaining about it.

Ever popular browser-based web applications

The Client-Server architecture continued to be popular for a few years but the industry was always on the lookout for something simpler - an architecture where we would not need to install anything on a client machine. This gave rise to the architecture where the client part of the application was available in a browser.

A browser happened to be a generic client. There was no need to install anything on the client because most of the users were using the browser which was already installed along with the operating system. In some cases, users installed their preferred browser and used it for accessing all the applications.

The actual application used to run on servers which was called the **web server**. Web Servers could be in the customer's own network or somewhere on the Internet. For higher scalability, we could form a web farm of multiple web servers which allowed us to serve the same application to hundreds and thousands of users at the same time. This architecture and the applications which were created at that time, survived for many years, and in a way, it still does.

Service Oriented Architecture (SOA)

It was only because of the modern needs of having an application available on different devices including a mobile phone - that the need for a new architecture was felt. This new architecture was called **service-oriented architecture** where server-side application was split into components that were called **services**.

In SOA, Services made calls to other services for consuming them. These calls were based either on certain proprietary protocols or other generic binary protocols, like SOAP. One of the benefits of this architecture was easy scalability of the entire application to hundreds of thousands of users, within a short period of time. An entire application made up of multiple services, could be replicated on multiple machines. In this way, we could support an enormous number of users.

Although this architecture was very popular, it still failed to address some of the critical issues posed by modern applications.

Issues in SOA that gave rise to the evolution of Microservices

First of all, since SOA used binary protocols and sometimes proprietary ones too, it was difficult to communicate and integrate all the components created by different teams.

An even bigger issue was related to the cost of replication of the entire application for dynamic scaling, and the efficiency with which it could be achieved in a small amount of time when the application received a sudden increase in load.

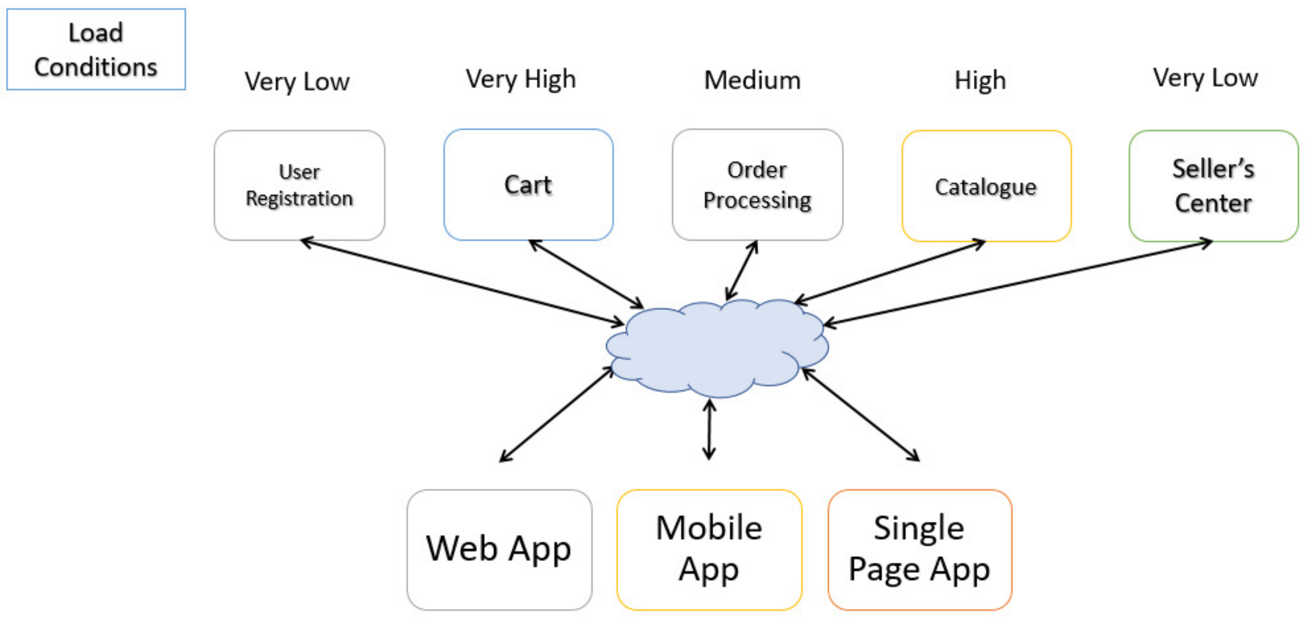
Slowly this architecture evolved into the **Microservices architecture**. In fact, some people call microservices as the **service-oriented architecture, done right.**

Microservices architecture is based upon shared protocols like HTTP, message queuing and the concept of REST API.

Another **difference between the service-oriented architecture (SOA) and the microservices architecture** is the principle of design on which the services are created.

An initial generation of SOA apps was where the services replaced the components of monolithic applications keeping the layers and tiers of the architecture, unchanged.

In the Microservices architecture, **each service is designed to fulfil and implement one of the subdomains** in the domain of the application. For example, in an E-Commerce application, instead of creating services based upon layers and tiers in which they exist, we create the services on the subdomains within the domain space of e-commerce like the Shopping Cart, Catalogue, Shopper Management and the Order processing etc.



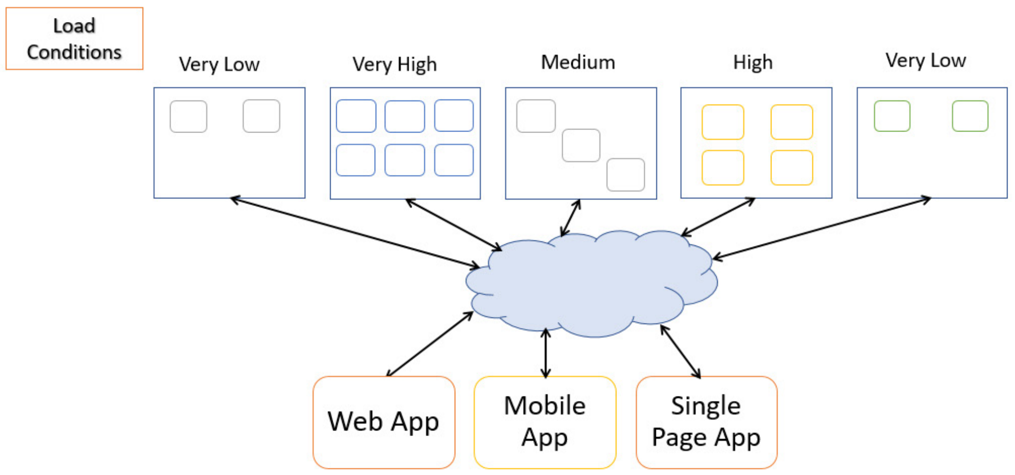
As we can imagine, not each part of the application is equally loaded. The highest demand is for the part of the application which is related to the Shopping Cart and indirectly the catalogue.

Other parts of the application like the Seller's Center or User Registration are not so much in demand and do not put as much high demands on the hardware as much as the Shopping Cart part of the application does.

So, when we need to scale out the application, that is to create higher number of instances of the service to cater to the high demand on the popular parts of the application, we need a greater number of instances of the services which are in high demand as compared to the number of instances of the services which are in less demand.

Since we have independent services for each subdomain, we can decide and define the number of instances of each service which will be running to cater to a specific load condition.

We can even let the load conditions be indeterministic. If the load conditions force us to increase the scale of the application then, we know the proportion in which the number of instances of each service are to be increased because we know the relative demand that each service is going to face in higher load conditions. This allows us to have only a minimum number of instances of each service running, which are just sufficient for the load condition.



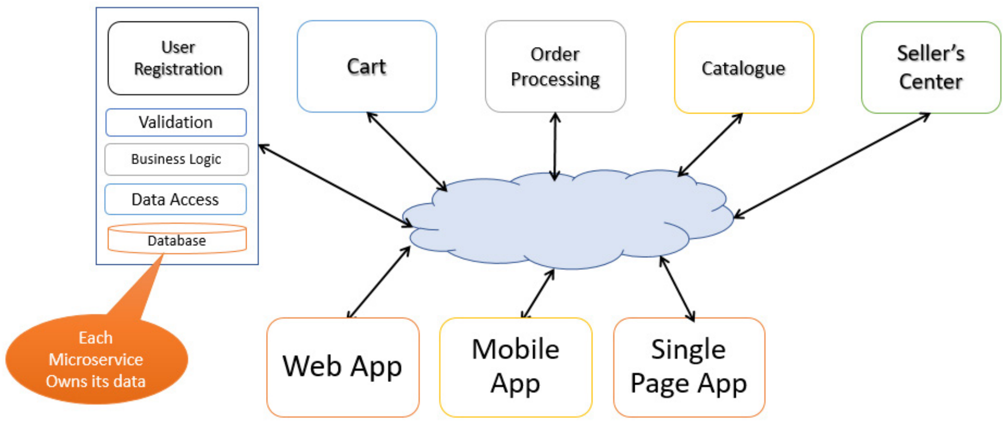
Designing the Logical Structure of Microservices

Let us now take a look at the logical structure of microservices. Each microservice is designed to be logically independent and isolated from the other microservices in the same application.

The entire code of that microservice and the components built on top of that code, are within the microservice itself. So much so, that even the ownership of the data which is required by that microservice, is with the microservice itself.

This type of architecture has an added advantage - the inter process communication, even within the application, is very less.

Logical Structure of a Microservice



It is implicit in the design and the structure of microservices that **one microservice cannot and should not access the data owned by the other microservice**, directly.

We come across the concept of eventual consistency which means that the data will be in a consistent state at a certain point of time in the future, which may not be immediately when a transaction happens. This consistency will be achieved by making the inter microservices calls and allowing the data owner microservice to take care of the database changes which are needed to be done.

Benefits of Microservices architecture:

1. The first and the biggest advantage of this architecture is the **ability to independently scale each service** as per the demand on that service. If the shopping cart subsystem is under high demand, there is no need to scale up the subsystem of the seller's center along with it. This is much more cost-effective compared to scaling up the entire application in response to the increase in the demand for just one subsystem of the application.

2. Another benefit of the microservices architecture is the ability to **update each service independent** of the other services. It is expected that each service is loosely-coupled with the other services so that a change in one service does not necessitate a change in other services. This makes software maintenance much easier compared to earlier architectures.

3. Since the required hardware resources used are minimum for running each service independently, the overall cost of the entire application is much lower even when compared to the service-oriented architecture.

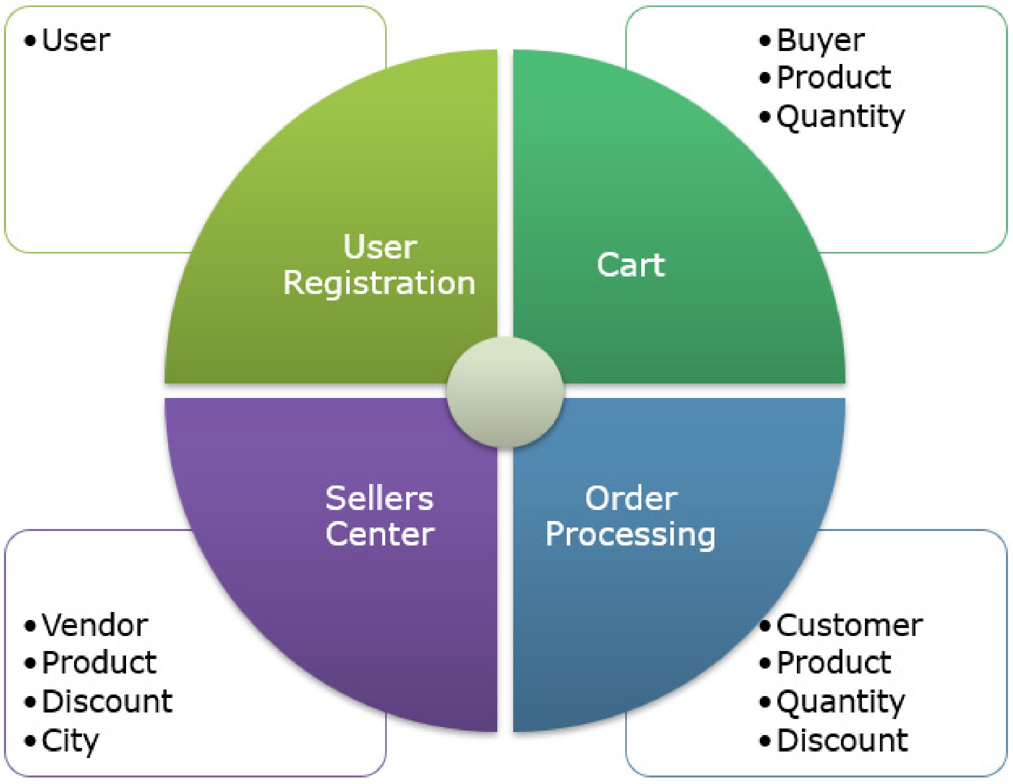
4. The development of each service and then the subsequent maintenance of that service will be done by a separate team which is proficient in that subdomain. This improves the efficiency of that team and the overall productivity of the entire application development.

Bounded Context

It is crucial to design the microservices in line with subdomains of the applications' major domain.

For such a design to be created, the concept of **bounded context** is used. The Subdomain of the application define the entities in its own context which are bound to that subdomain. Even though the entity may be similar or exactly the same, in a different context, it may be called differently and it may have different *attributes* in that subdomain.

For example, in an E-commerce application, the sub domain of the user management may treat an entity called 'user' in its context. May be, the same user entity in the Shopping Cart sub-domain is called as a 'buyer'. The same entity is possibly called as a 'customer' in the order processing subsystem and called 'vendor' in the seller center.



In the bounded context of user registration, the entity named *user* has the attributes of the name, address and email address as the attributes under focus, whereas in the order processing subsystem which has 'customer' entity, the focus is on the payment details and loyalty code of that customer. The bounded context is used to focus on the different attributes of the same entity and to store the data of that entity in different databases in different servers, or at least in different tables that are owned for editing by the respective microservice.

So, we come to the question - **What is the relation between the bounded context and a microservice?**

It is desirable to create one microservice for each bounded context in the domain of the application. While designing microservices, it is first necessary to define the subdomains within the domain of the application. As a next step, create the bounded context for each of the subdomain. Once the bounded contexts are defined, we can design microservices for each of the bounded contexts.

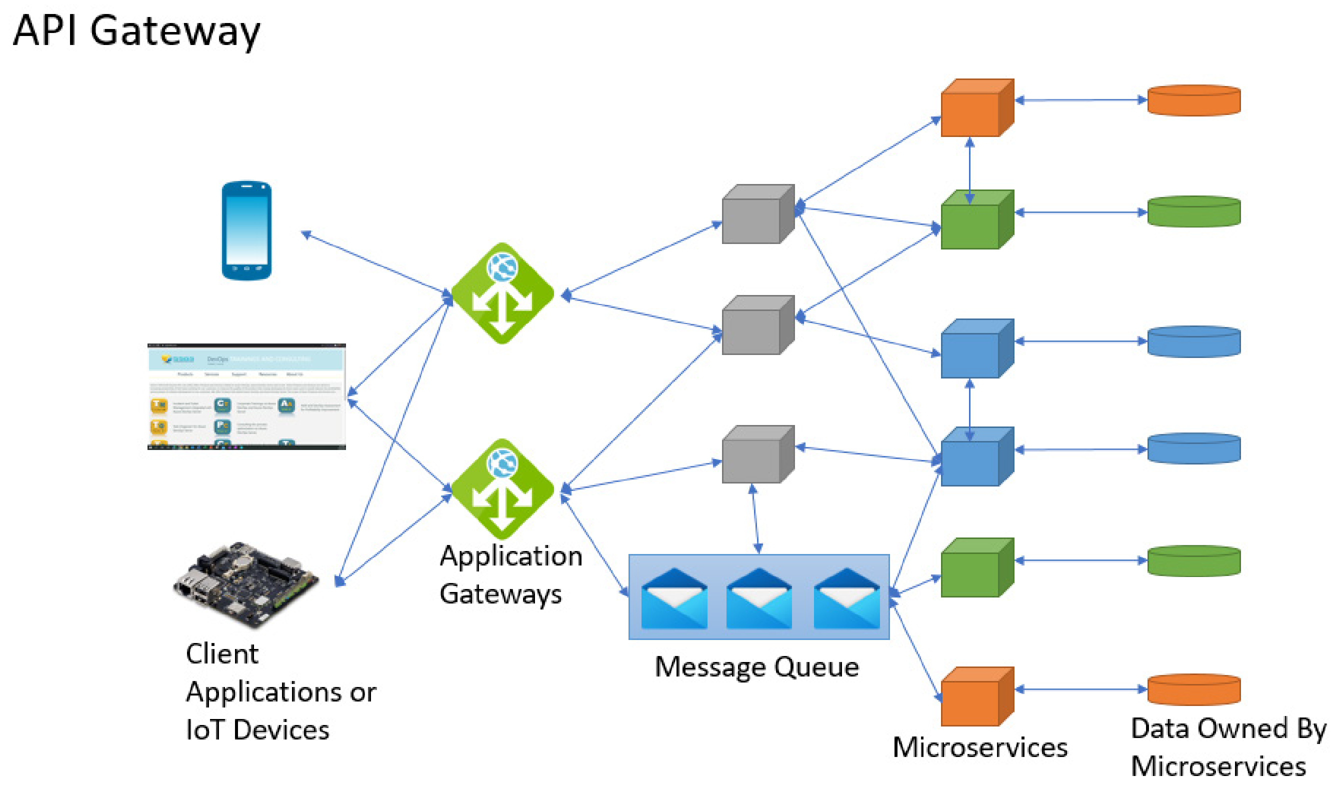
Accessing Microservices from clients

Once microservices are logically defined, we can now talk about how the client application will access these microservices.

The simplest way of accessing a microservice is direct client-to-microservice communication. This may use a simple HTTP request response paradigm. This type of communication has no issue when there are a smaller number of microservices in the application.

However, as the number of microservices increase and the number of clients also increase, the necessity of many to many types of connections to be created will arise and overwhelm the communication channels with this chattiness. As a result, the efficiency of the application will reduce.

This type of communication is not possible in a non-trivial application which has many microservices catering to different sub-domains within the application. In such conditions, it is suggested to use **communication through an API gateway.**


API Gateway
Client Applications or IoT Devices
Application Gateways
Message Queue
Microservices
Data Owned By Microservices

This gateway can be a microservice itself which routes the communication from the clients to the individual microservices intended to be used by the client.

The API gateway may also be a dedicated service offered by the Cloud Service Provider as Platform -as-a-service (PaaS). The API gateway may also actively process the communication from the client, working as Man-in-the-Middle, so that the message received by the business microservice is in the acceptable format, irrespective of the format sent by the client.

For example, the client may send an HTTP request but the business microservice may expect a message in the queue. The API gateway will convert that HTTP request into a message which is put in the queue. While adding the API gateway, care should be taken that the API gateway itself does not become a bottleneck in the application. In that case, it is desirable to create multiple API gateways for each logical subdomain which will give access to different microservices to the client.

Communication between the Microservices

As I mentioned earlier, it is desirable to keep the communication between the microservices to the minimum but it cannot be totally avoided. For example, it may be necessary to communicate between microservices for maintaining eventual consistency.

Let us now view how microservices can communicate with each other.

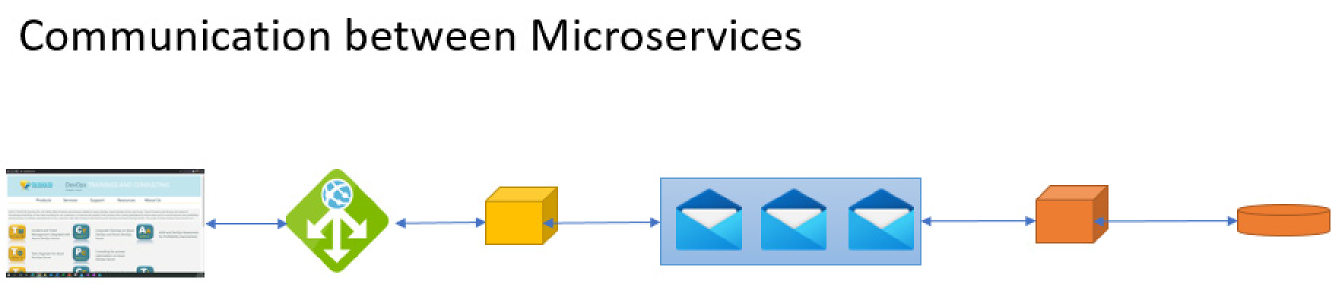
It is a strong suggestion that microservices communicate with each other *asynchronously*. That does not totally exclude simple synchronous communication using a protocol like HTTP which follows the request-response paradigm.

If a client sends HTTP request to the API gateway and expects immediate response in a certain amount of time, then the API gateway which may be a microservice itself, sends a further request to the business microservice using the same HTTP request response paradigm. The business microservice responds synchronously to the API gateway which then forwards the reply back to the client.

This type of communication is used mainly for data queries and reporting.

Another synchronous communication is the push communication from the microservice to the clients, using *messages*. A technology like SignalR is very effectively used for this type of communication. For other type of operations, asynchronous communication is suggested to be used.

For communication between microservices, a more commonly used pattern is to use a **message-based communication**.



In this pattern, a message is sent with the command built-in into it through a broker software like a *queue*. After sending this message, the sending microservice does not wait for any response or reverse message, but continues with its own process. The message is picked up by the target microservice as and when it is free to do so. This is **asynchronous communication**.

There may sometimes be a response from the business microservice which is sent back in another message put in the queue. Most of the times, there may not be any response at all. The mechanism for the queue is sometimes a simple queue mechanism like RabbitMQ. Queue mechanism assures the delivery of the message even if the recipient is not available at the time the message is put in the queue. Such communication is usually between two microservices where there is only one recipient microservice.

In some cases, we intend to have multiple business microservices taking action on a single message which is sent by the sending microservice. This is made possible using asynchronous event driven communication.

Azure Service Bus which uses the publisher subscriber paradigm is one such mechanism to address multiple recipients of the same event. For example, if the price of a product has changed after the product was put in the Shopping Cart, the event of price change will be published by the catalogue microservice and then subsequently consumed by the Shopping Cart as well as the Order Processing microservice. This type of communication between microservices is the most desired type of communication.

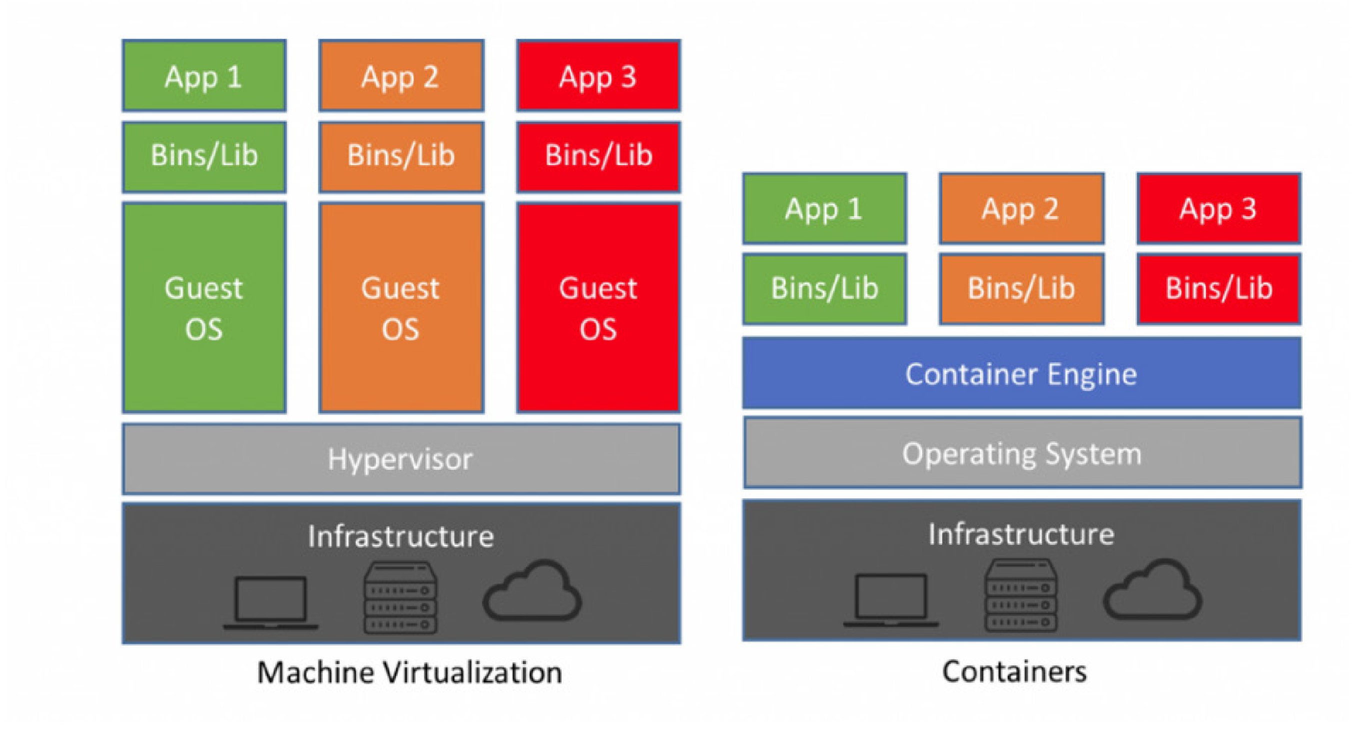
So far, we have discussed the logical structure of the microservices and how the database is part of that logical structure. Let us now see how that logical structure maps to the physical structure.

The most popular target to deploy microservices are **Containers**. Although it is possible to keep all sub services of the microservice in one container only, it is also possible to design a container for each such sub-services which is being provided by the microservice. There can even be a container which stores the database for the microservice.

Storing the microservices in a single container or multiple containers is entirely a decision based on the complexity of the microservice and the desired level of granularity that we require within the services of microservice itself from a scalability point of view.

Container – A Brief overview

Let's now take a brief overview of containers. Containers provide a virtualized platform for the implementation of microservices. Containers are small virtualized processes which utilize the operating system of the host on which they are created. Unlike a virtual machine, it does not have its own operating system.



Since it does not have a full operating system of its own, it usually is very small as compared to the virtual machine. This gives it an advantage over virtual machines, of creation and execution with very small resources.

It is a much more efficient way of virtualization of an application compared to the virtual machine. A container usually runs only a single process. By forcefully creating other processes within the container, we may have multiple processes running in the container but that is a pattern which we should try to avoid as it goes against the intention of independence of microservices.

Docker and Kubernetes

Docker is one of the most popular open source Technologies that provides a platform for the containers to be created and run. Docker provides support for:

- Building the image for the container
- Deploy the image so that a container is created
- Manage the instance of the image (which is the container)
- Tearing down the container which was created based on the image, after its need is over.

Another advantage that containers provide is the **automation of management and orchestration of the instances** of those containers. External services can create the instances off containers as needed and can destroy those when there is no need. These services can also scale the number of containers up or down based upon the need. Kubernetes is one of the most popular external services that manages docker containers and their hosts.

Editorial Note: **NEW TO KUBERNETES? CHECK PAGE 46 OF THIS MAGAZINE.**

Docker has an implementation on Windows also, although basically Docker was created for Linux. The implementation on Windows is called "docker for desktop". The prerequisites to install docker for desktop is an operating system which is either Windows 10 for Windows Server 2016 or Windows Server 2019. On these operating systems, the service of the virtual machine platform should be enabled. We may also enable Windows subsystem for Linux which works as the backend for docker desktop. If we have not enabled Windows subsystem for Linux, then as a process of installation of docker desktop, Windows subsystem for Linux version 2 is installed. Docker for desktop also includes the installation of Kubernetes.

Summary

In this article, we have taken an overview of the following:

- The evolution of architecture of microservices as a response to the needs of the industry.
- Process of how to design the logical microservices and then the design of physical microservices.
- How Microservices are accessed by clients and patterns of communication between Microservices.
- Implementation details of Microservices - Containers.
- Introduction to Docker and Kubernetes.

Subodh Sohoni
Author

Microsoft® MVP Most Valuable Professional

Subodh is a consultant and corporate trainer. He has overall 28+ years of experience. His specialization is Application Lifecycle Management and Team Foundation Server. He is Microsoft MVP – VS ALM, MCSD – ALM and MCT. He has conducted more than 300 corporate trainings and consulting assignments. He is also a Professional SCRUM Master. He guides teams to become Agile and implement SCRUM. Subodh is authorized by Microsoft to do ALM Assessments on behalf of Microsoft. Follow him on twitter @subodhsohoni

Techinical Review
Gouri Sohoni

Editorial Review
Suprotim Agarwal

Modern monitoring & analytics

Aggregate metrics and events from 400 + technologies including .Net, Azure, and AWS

Trace requests across distributed systems and alert on app performance

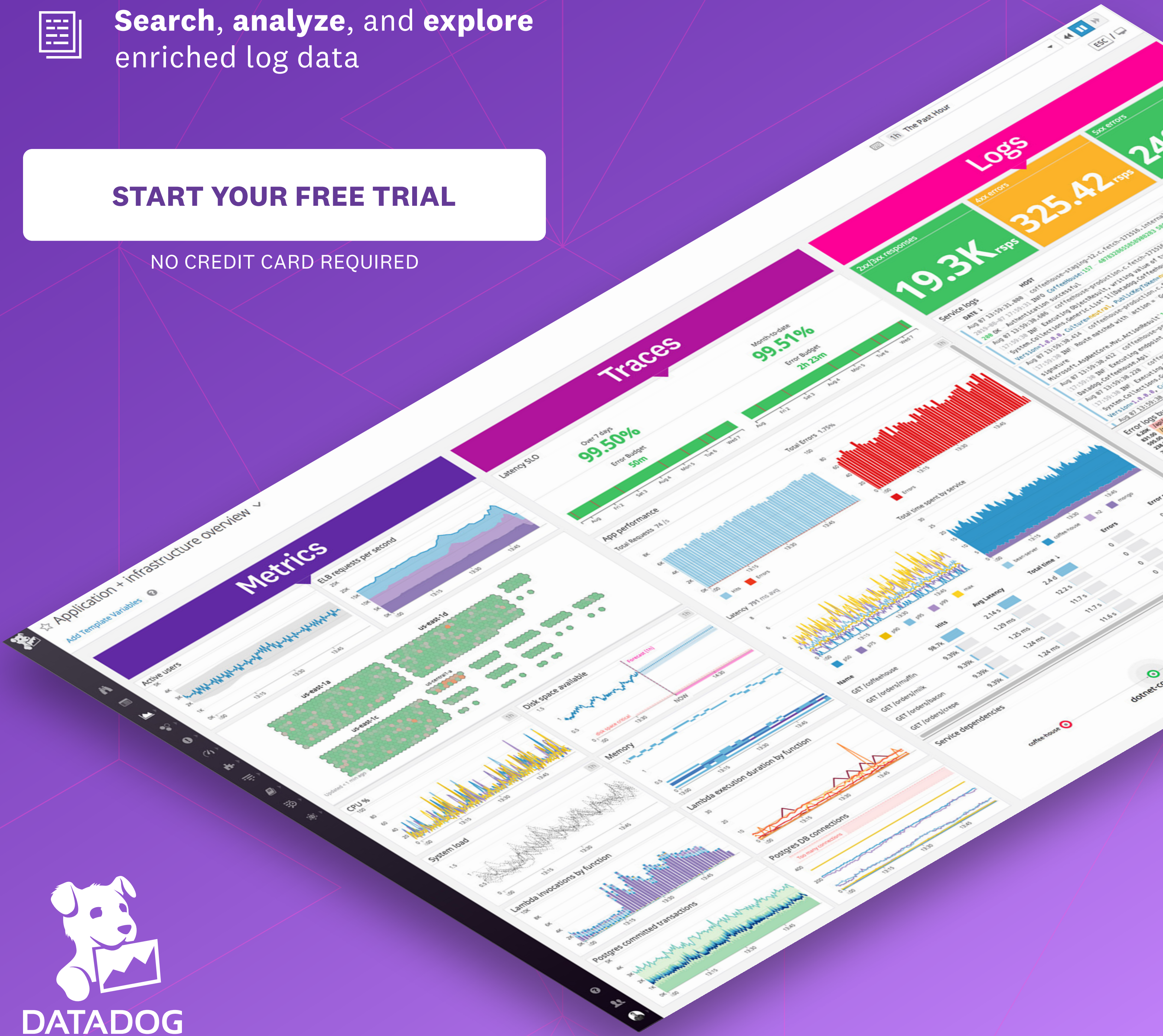
Seamlessly pivot between correlated data for rapid troubleshooting

Monitor your applications and API endpoints via simulated user requests

Search, **analyze**, and **explore** enriched log data

START YOUR FREE TRIAL

NO CREDIT CARD REQUIRED



DATADOG

Damir Arh

ARCHITECTURE OF WEB APPLICATIONS (DESIGN PATTERNS)

This article covers a selection of design patterns that are used in most web applications today.

There are two primary categories of web applications based on where the final HTML markup to be rendered in the browser, is generated:

- In **server-side rendered** applications, this is done on the server. The final HTML is sent to the browser as a network response.
- In **client-side rendered** applications (commonly known as SPAs – single page applications), this is done in the browser. The code to do that and the data required are requested from the server.

In my previous DNC Magazine article Developing Web Applications in .NET, I described the difference between the two in more detail and also listed the technology stacks for both of them that a .NET developer would most likely choose.

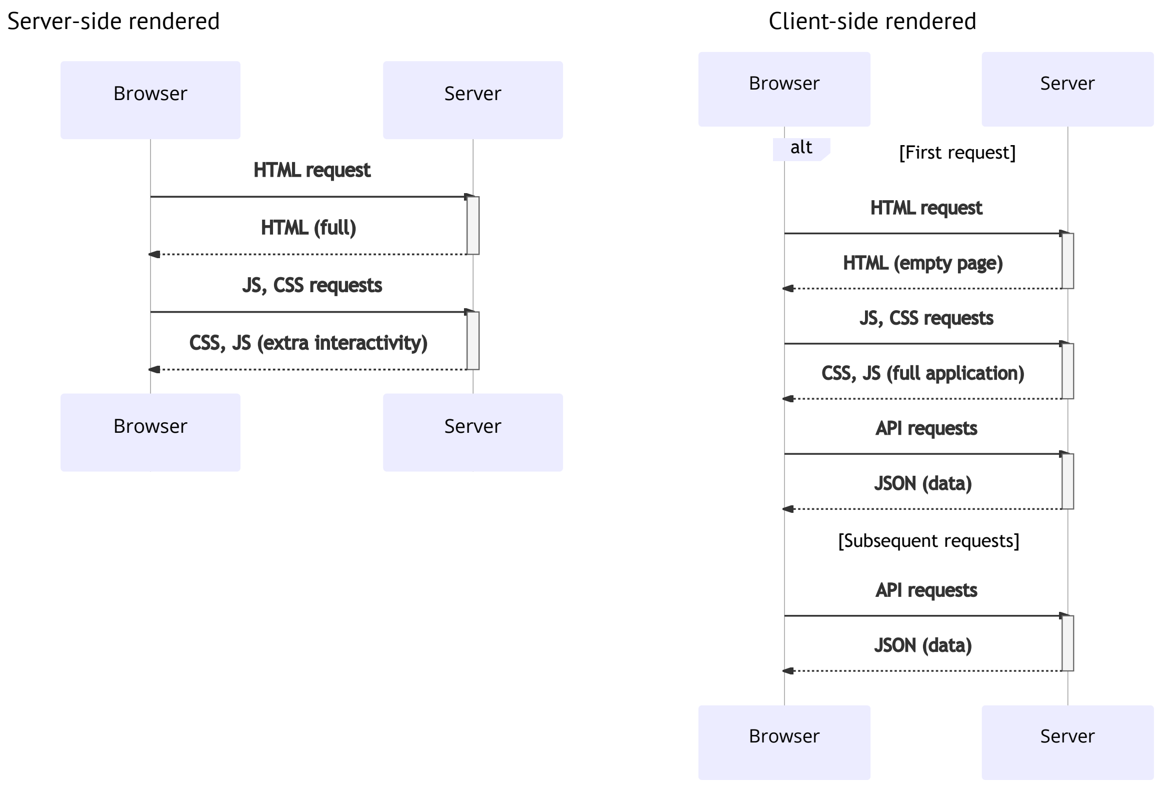


Figure 1: Data exchange between the browser and the server in web applications

This article will cover only the **architecture of the server-side rendered applications.** The architecture of client-side rendered applications has more in common with desktop and mobile applications which I have already covered in my previous article from this series: Architecture of desktop and mobile applications.

The way the application code is decoupled from the user interface in such applications depends on the technology stack used (Blazor or one of the JavaScript SPA frameworks), but it's usually not the MVVM pattern.

The other patterns described in that article (dependency injection, remote proxy and validation using the visitor pattern) are just as applicable to single-page web applications as they are to desktop and mobile applications.

The term server-side rendering is sometimes also used in connection with modern JavaScript SPA frameworks. This is not to be confused with the more traditional meaning of the term I used. The server-side rendering (SSR) feature of JavaScript single-page applications only means that on the first request to the server, the page is rendered there, and sent to the browser in its final form. There, the single-page application is "rehydrated", meaning that any further interaction with the page is handled on the client and only data is retrieved from the server from here on.

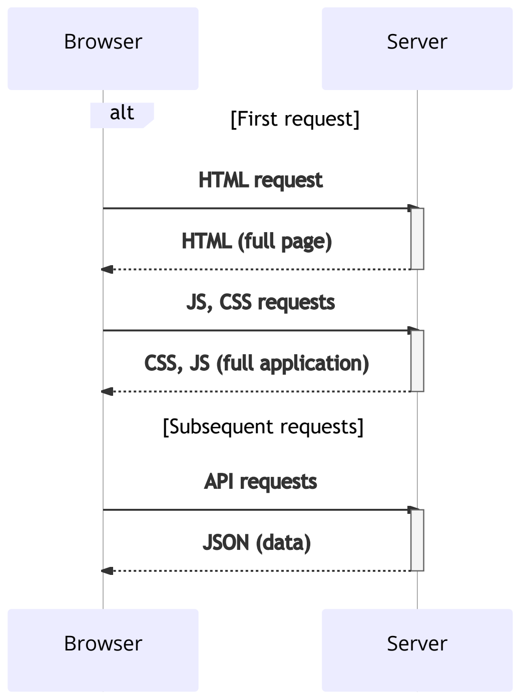


Figure 2: Data exchange between the browser and the server in server-side rendered single page applications

This can be achieved by running the same application code on the server and in the browser. That's why such applications are also called isomorphic (as having the same form on the client and on the server) or universal.

That's for the introduction. I'm not going to discuss such applications any further in this article.

Decoupling application code from UI

Most of the server-side rendered applications today follow the **MVC (model-view-controller) pattern** to decouple the user interface from the rest of the application code. As the name implies, there are three main building blocks in this pattern:

- **Models** describe the current state of the application and serve as a mean of communication between the views and the controllers.
- **Views** are the user interface of the application. They read the data to be rendered from the models. Since they are web pages, interaction with them can trigger new requests to the server.
- **Controllers** handle requests sent to the server when users navigate to a URL or interact with a previously served page. In response, Controllers select a view to be rendered and generate a model that they pass to it.

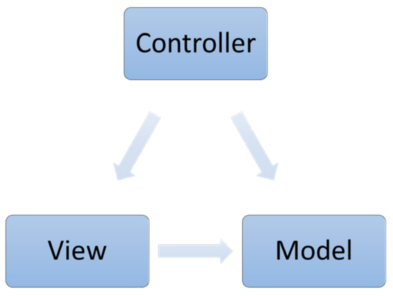


Figure 3: MVC pattern building blocks

In the .NET ecosystem, the currently recommended MVC framework (or application model) is ASP.NET Core MVC. It's a part of .NET Core. Although it has some similarities with ASP.NET MVC for .NET framework, it was rewritten from scratch and is different enough that applications can't easily be migrated between the two.

The controller code in ASP.NET Core MVC is placed in the so-called action methods which are grouped together in controller classes:

```
public class HomeController : Controller
{
  public IActionResult Index()
  {
    return View();
  }
}
```

By default, HTTP requests invoke a corresponding action method based on a routing convention:

```
routes.MapRoute(
  name: "default",
  template: "{controller=Home}/{action=Index}/{id?}");
```

This template specifies that the first part of the URL selects the controller class, the second one selects the action method in it, and the rest is treated as an argument for the action method (arguments will be explained in more detail later in the article). It also specifies the default value for the controller class and the action method if they aren't included in the URL. Of course, the routing convention can be extended and customized.

The sample action method simply triggers the rendering of the view. The selection of the view is again convention-based. By default, the view name will have to match the name of the action method and be placed in a folder matching the name of the controller class (i.e. `Home/Index` in this case). Alternatively, the name of the view to render can be passed as an argument to the `View` method.
The sample action method will always render an identical page. In most cases that's not enough. To make the page content dynamic, the action method will need to pass data to the view using a model:

```
return View(personModel);
```

In the view, the model is then accessed through its `Model` property:

```
<h4>@Model.FirstName @Model.LastName</h4>
```

To respond to user interaction, the action method also needs inputs. In a web application, these will come from the URL (the path and the query string) or from the form fields in the case of a submitted HTML form. The model binding process in ASP.NET Core MVC maps all of them to the action method arguments.

```
public ActionResult Edit(int id, PersonModel person)
{
  // ...
}
```

As in the case of routing, ASP.NET Core MVC includes conventions that are used to automatically bind values from different sources to method arguments based on the names of arguments (`id` in the example above) and the property names of complex types in the role of arguments (`PersonModel` in the example above). These can be further customized using attributes and even custom model binders.

It might be tempting to reuse existing internal entities as models to be passed to the view and to be received as arguments from the model. However, this isn't considered a good practice because it increases the coupling between the internal application business logic and the user interface.

Having specialized models brings additional benefits:

- When they are passed to the view, they can already be structured according to the needs of that view. This can reduce the amount of code that would be needed in the view if more generic models were used.

- When they are received as arguments, they can allow the model binder to do a better job at validating them. If a model doesn't have a property that's not expected in the request, then there's no danger that it would be parsed from a malicious request and used in an unexpected manner. Also, there's no potential for conflicting validation attributes if the model is only used for a single request.

A common argument against using dedicated models is the need for mapping the properties between the entity classes and the models. While writing these manually can be time consuming and error prone, there are libraries available to make this job easier. Among the most popular ones is AutoMapper.

Dependency injection

The described architectural approach makes the view completely unaware of the controller and the rest of the business logic in the application. Each view knows only about the model received from the controller and the application routes it interacts with.

On the other hand, the controller is fully aware of the view that it wants to render and its corresponding model, as well as the model it can optionally receive as input.

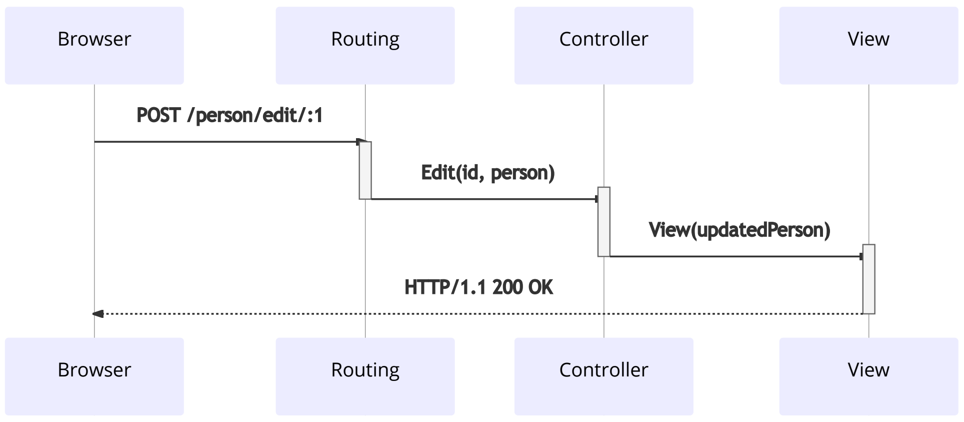


Figure 4: Interaction between MVC building blocks

However, there's another big part to action methods that I haven't talked about yet.

They will need to execute some business logic in response to their input in order to generate the model required for the view. While this business logic could be implemented directly in the action method, it's best to minimize the amount of code inside it and implement the actual business logic in dedicated service classes:

```
public ActionResult Edit(int id, PersonModel person)
{
  var personService = new PersonService();
  var updatedPerson = personService.UpdatePerson(person);
  return View(updatedPerson);
}
```

To avoid instantiating those service classes inside the controller class and thus coupling the controller class to a specific service class implementation, dependency injection can be used. This allows the controller class to depend only on the public interface of the service class and get the instance as a constructor parameter:

```
public class PersonController : Controller
{
  private readonly IPersonService personService;

  public PersonController(IPersonService personService)
  {
    this.personService = personService;
  }

  public ActionResult Edit(int id, PersonModel person)
  {
    var updatedPerson = this.personService.UpdatePerson(person);
    return View(updatedPerson);
  }
}
```

ASP.NET Core has built-in support for dependency injection!

Even its own internals take advantage of it extensively. Hence, it's very easy to use that same dependency injection implementation for your own dependencies. They can be configured in the `ConfigureServices` method of the `Startup` class:

```
services.AddScoped<IPersonService, PersonService>();
```

The configuration consists of the interface, the implementing service and the service lifetime. In a web application, the service lifetime is commonly defined as scoped, meaning that the services are created for each request. This ensures isolation of state between multiple requests which are being processed in parallel.

Data Access

By removing the business logic from the controller, the services become another actor in the MVC architecture. The controllers must be aware of them. Or to be more exact: thanks to dependency injection, they only need to be aware of their public interface, but not of their implementation and internal dependencies.

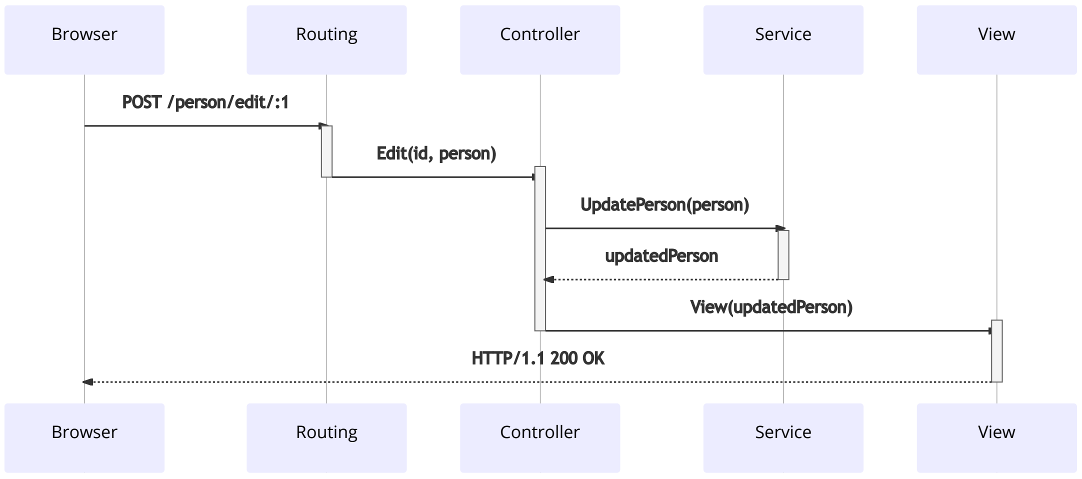


Figure 5: Interaction between MVC building blocks when using a service

Of all the typical service dependencies, data access deserves special attention. It is most likely going to be implemented using an ORM (object-relational mapper) like Entity Framework Core or a micro ORM like Dapper.

But how do data access libraries fit into a web application architecture?

The architecture patterns that are most often mentioned in connection with data access are the repository pattern and the unit of work pattern. So, let's start by exploring these.

The repository pattern isolates the business logic from the implementation details of the data access layer. Its interface exposes the standard data access operations over a specific resource (or entity), often referred to as CRUD (Create, Read, Update, and Delete):

```
public interface IPersonRepository
{
    IEnumerable<Person> List();
    Person Get(int id);
    Person Insert(Person person);
    Person Update(Person person);
    void Delete(Person person);
}
```

Any other type of queries that might need to be performed over the same entity, should be exposed from its repository in a similar manner. This makes the repository the sole container for the remote store queries and any mapping code that might not be done automatically by the ORM used for its implementation.

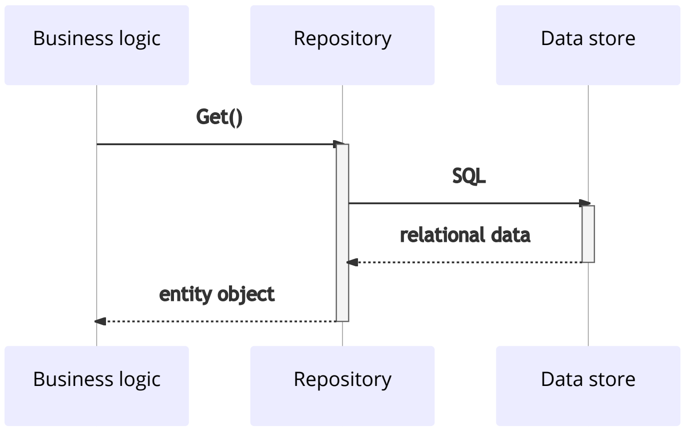


Figure 6: Data access using the repository pattern

In a simple implementation, any operation on the repository (including data modification) will immediately be executed on the underlying data store. However, this will be insufficient in more complex scenarios because it doesn't allow multiple operations to be executed within a single transaction to ensure their atomicity (i.e. that either all operations are performed successfully or none).

One option would be to add a separate Save method to the repository which could be called to apply any pending changes caused by calling other methods in it. But this still doesn't completely solve the problem.

What if changes across multiple repositories need to be applied inside a single transaction?

This is where **the unit of work pattern** comes into play. It exposes a common Save method for multiple repositories. Typically, these repositories will now be accessed through the unit of work instead of directly:

```
public interface IUnitOfWork
{
  IPersonRepository PersonRepository { get;  }
  IOrderRepository OrderRepository { get;  }
  // ...

  void Save();
}
```

With this pattern implemented, data modification method calls to any of the repositories are only applied to the underlying data store when the Save method on the unit of work is finally called.

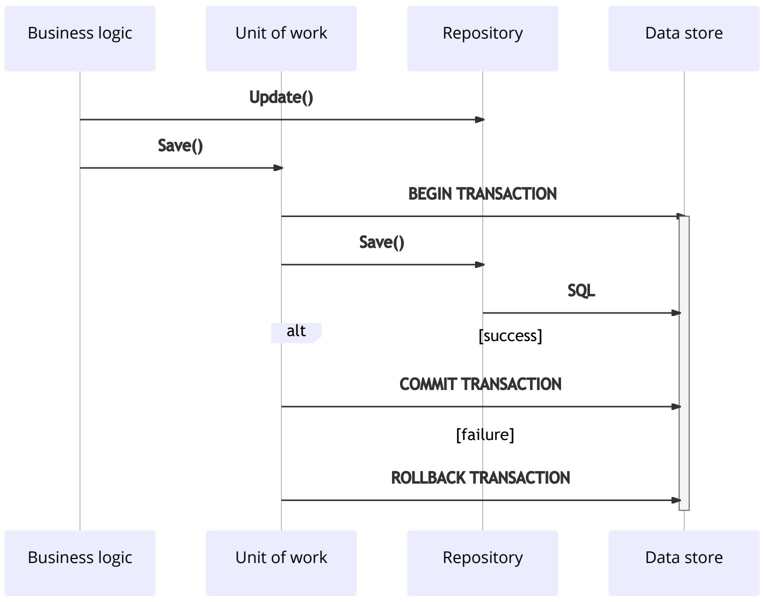


Figure 7: Atomic data modification using the unit of work pattern

If you're familiar with Entity Framework Core, you probably recognized the similarity between the IUnitOfWork interface above and a typical DbContext class in Entity Framework Core:

```
public class SalesContext : DbContext
{
  public DbSet<Person> Persons { get; set; }
  public DbSet<Order> Orders { get; set; }
}
```

In it, the `DbSet<TEntity>` properties have the role of repositories, allowing simple execution of CRUD operation which is evident from the following (subset of) members of the class:

```
public abstract class DbSet<TEntity> : IEnumerable<TEntity>
    where TEntity : class
{
  public virtual EntityEntry<TEntity> Add(TEntity entity);
  public virtual EntityEntry<TEntity> Remove(TEntity entity);
}
```

Reading of entities can be done using the `IEnumerable<TEntity>` interface and editing is done by simply modifying properties of a `TEntity` instance. The `DbContext` class has a `SaveChanges` method which does the job of `IUnitOfWork`'s `Save` method.

This similarity is no coincidence.

Entity Framework Core follows the concepts of the **repository and unit of work patterns**. This raises the question:

Does it make sense to implement these two patterns in a web application when using Entity Framework Core for data access? (Dapper and other micro ORMs don't implement these patterns themselves).

To answer this question, we must look at the benefits that these patterns bring and to what extent they are still applicable when the business logic code uses Entity Framework Core directly:

1. When using Entity Framework Core directly, it's not **abstracted away** from the rest of the application anymore. This will make it much more difficult to replace it as the data access library later.

   But are you ever going to do that?

   Also, if Entity Framework Core is not (going to be) used for all data stores, exposing it directly will make it impossible to always use the same interface and hide this implementation detail from the rest of the code.

2. The concrete unit of work and repository pattern implementations can be **replaced with mocked ones in unit tests**. Again, that's not possible when using Entity Framework Core directly.

   However, there's an in-memory database provider available for Entity Framework Core which can partially replace the mocks by making it possible to run the tests without accessing the underlying data store.

3. The repositories serve as **containers for custom queries** for their entity. This can still be achieved to some extent even when using Entity Framework Core directly. For example, the queries can be implemented as extension methods of the generic `DbSet` class for that specific entity.

To conclude, as often in software architecture, there's no definitive answer to the question. It depends on the value the above points bring in for the application.

It makes more sense in implementing your own repositories and unit of work if your application is large and complex.

Conclusion

In the article, I have described some of the architecture patterns that are often used in web applications.

I started with the overarching MVC pattern that is used for decoupling the application code from the user interface in most of the server-side rendered web applications today. I continued with the application of dependency injection for further decoupling the business logic implementation from the controller.

In the final part, I covered the usage of repository and unit of work patterns in the data access layer.

In comparison to the application models for desktop and mobile applications covered in the previous article from the series, these patterns are already implemented by ASP.NET Core. In most cases, they can be used without installing third party libraries or implementing them yourself.

• • • • • • •

Damir Arh
Author

Microsoft® MVP Most Valuable Professional

Damir Arh has many years of experience with software development and maintenance; from complex enterprise software projects to modern consumer-oriented mobile applications. Although he has worked with a wide spectrum of different languages, his favorite language remains C#. In his drive towards better development processes, he is a proponent of Test-driven development,Continuous Integration, and ContinuousDeployment. He shares his knowledge by speaking at local user groups and conferences,blogging, and writing articles. He is an awarded Microsoft MVP for .NET since 2012.

Techinical Review
Ravi Kiran

Editorial Review
Suprotim Agarwal

Mahesh Sabnis

Understanding Lifecycle Events in a React.js Application and using React Hooks

React.js is one of the most popular JavaScript libraries for building UI interfaces in modern applications.

Unlike other JavaScript libraries like jQuery, React.js defines a lifecycle object model for components.

The compositional principle of React.js uses components for building rich and complex UIs. These components are loaded in the browser so that end-users can work with them.

A **component** is an autonomous object that contains its own UI, data and behavior (methods/events). Components may have events subscribed with them to handle the behavior of the UI and for updating it dynamically.

For components that have an event subscribed with them, what would happen to the event if these components are unloaded. Would it remain subscribed or would it be released? The question here is, how to handle such scenarios in React.js components?

React.js component lifecycle

Every React developer should understand the React.js component lifecycle. The React.js component lifecycle is explained in Figure 1:

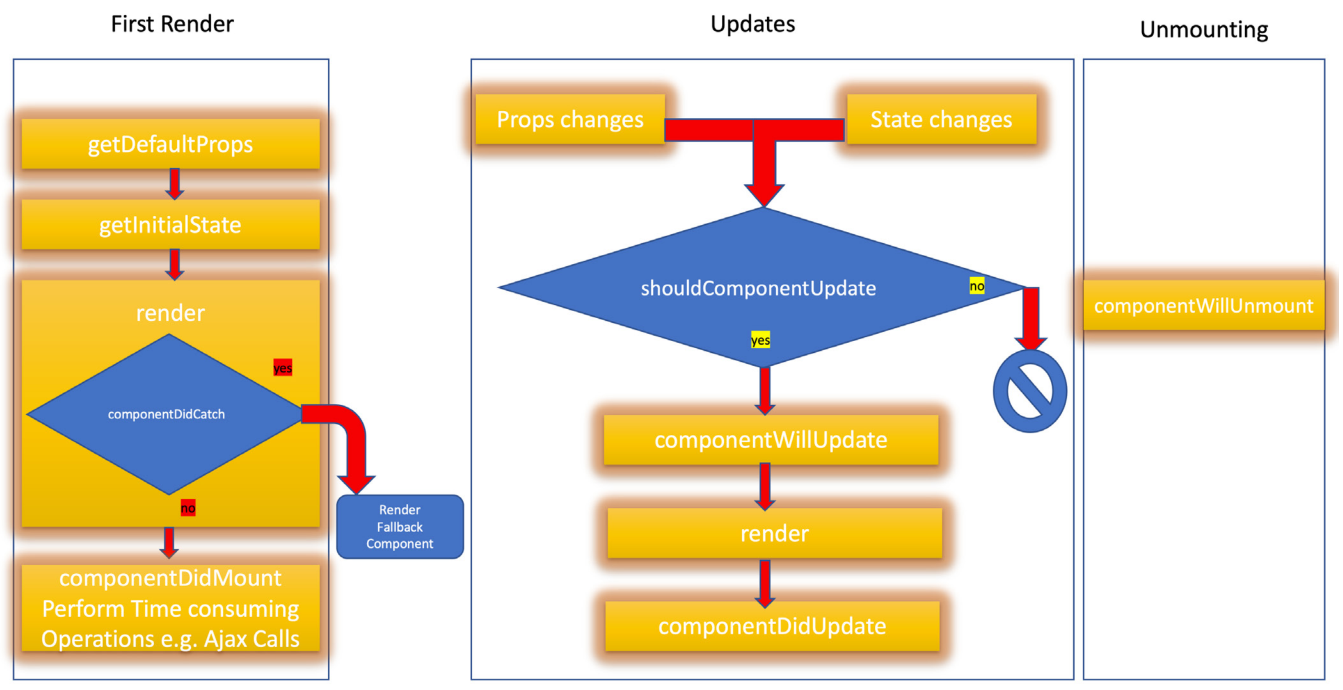


Figure 1: Component Lifecycle

As shown in Figure 1, the component's lifecycle has three steps of execution explained in the following points:

1. The **First Render**: This is the step where the component is loaded for the first time. In the first rendering step, the following operations are executed:

   a. **getDefaultProps**: The default props are passed to the component. These props values may contain some initial data that will be used by the component during rendering.

   b. **getInitialState:** Component's constructor defines state properties. These properties are bound with the UI elements of the component. These state properties may have an initial value to define rendering of the component.

   c. **render:** This is an important step. In this step, the UI rendering process starts. During this rendering process, the state and props values are used to define the rendering of UI elements. If any error occurs during the rendering process, the **componentDidCatch** is invoked which takes care of rendering the fallback UI component. (Note: to do this, you must use the Error Boundary concept of React.js)

d. **componentDidMount**: gets executed once after the first rendering takes place. You can add code here to handle heavy load operations like AJAX calls. All DOM and state updates should be handled. You can also integrate with other JavaScript libraries here.

2. **Component Updates:** This is an important step when the component is already initially rendered. In this step, the end-user's integration will be handled like, changing props and states. The state changes for the local or parent component can be sent to the child component as props changes. The following points explain the update process.

a. **props changes:** occurs when updated props are passed to the child components. Once the child component receives the props changes, it checks if the component needs any updates in UI using **shouldComponentUpdate** method. If this method returns *true*, then the **render** method of the child component will be invoked to re-render the component and **componentDidUpdate** method will be called. If the **shouldComponentUpdate** method returns *false*, then the component will not be changed.

b. **state changes:** if the state properties local to the component are changed, then **shouldComponentUpdate** will be invoked again. The component will be re-rendered if this method returns *true*, else nothing will happen.

3. **Component Unmounting:** When we replace one component with another component, then the **componentWillUnmount** method of the previous component will be invoked.

The reason behind discussing the component lifecycle in React.js is that our code should be appropriately implemented as per the lifecycle. Typically, this is very important when we need to make AJAX calls to fetch data from REST APIs and show it on the UI.

There is one more reason behind discussing this lifecycle. It is very important to manage the event registration and de-registration for components in React, else you may face **memory-leak** issues.

In this tutorial, we will be discussing the issue and its resolve.

The code for this tutorial is implemented using Microsoft's Visual Studio Code (VS Code). The React application is created using the **create-react-app** tool. This can be installed using the following command

```
npm install -g create-react-app
```

Note: *If you are using Linux or MacOS then the command will be as follows:*

```
sudo npm install -g create-react-app
```

To create a new React application, we need to run the following command from the command prompt :

```
create-react-app lifecycle
```

The above command will create a React application. In this application, we will be using Bootstrap. To install bootstrap, navigate to the React project folder that we have just created and run the following command:

```
npm install --save bootstrap
```

Open React application in VS Code. In the **src** folder, add a new folder and name it as components. In this folder, add a new file and name this file as **lifecyclecomponent.jsx**. In this file, add code as shown in the Listing 1.

```
import React, { Component } from 'react';
class ParentComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      toggle:true,
      value:0
    };
  }
  handleClick=()=>{
    if(this.state.toggle){
      this.setState({toggle:false});
    } else {
      this.setState({toggle:true});
    }
  }

  componentDidMount=()=>{
    console.log('Parent Component of ComponentDidMount is invoked');

  }
  componentDidUpdate=()=>{
    console.log('Parent Component ComponentDidUpdate is invoked');
  }

  render() {
    if(this.state.toggle){
      return (
        <div className="container">
          <input type="button" value="Toggle" className="btn btn-success"
            onClick={this.handleClick.bind(this)}/>
            <FirstChildComponent/>
         </div>
        );
     } else {
        return (
          <div className="container">
            <input type="button" value="Toggle" className="btn btn-success"
              onClick={this.handleClick.bind(this)}/>
              <input type="text" value={this.state.value}
              onChange={(evt)=> {this.setState({value: parseInt(evt.target.
              value)})}}/>
              <SecondChildComponent data={this.state.value}/>
             </div>
         );
      }
    }
  }

  class FirstChildComponent extends Component {
    state = {
      xPos:0,
      yPos:0
    };
    captureMousePositions=(evt)=> {
      this.setState({xPos:evt.clientX});
      this.setState({yPos:evt.clientY});
    }
    componentDidMount=()=>{
      console.log('First Child Component of ComponentDidMount is invoked');
      window.addEventListener('mousemove', this.captureMousePositions);
    }
    componentDidUpdate=()=>{
      console.log('First Child Component of  ComponentDidUpdate is invoked');
    }
    componentWillUnmount=()=>{
      console.log('First Child Component of  ComponentWillUnmont is invoked');
    }
    render(){
      return(
        <div className="container">
        <h2>The First Component</h2>

        <span>
          <strong>X Position {this.state.xPos}</strong> : <strong>Y Position {this.
          state.yPos}</strong>
        </span>
      </div>
      );
    }
  }

  class SecondChildComponent extends Component {
    componentDidMount=()=>{
      console.log('Second Child Component of ComponentDidMount is invoked');
    }
    componentDidUpdate=()=>{
      console.log('Second Child Component of  ComponentDidUpdate is invoked');
    }
    componentWillUnmount=()=>{
      console.log('Second Child Component of ComponentWillUnmont is invoked');
    }
    render(){
      return (
      <div className="container">
        <h2>The Second Components</h2>
        <div className="row-cols-1">
          <strong>
            {this.props.data}
          </strong>
         </div>
       </div>
      )
    }
  }

export default ParentComponent;
```

Listing 1: Parent and Child components

The code in Listing 1 shows the `ParentComponent`. This parent component will render `FirstChildComponent` and `SecondChildComponent` based on the value of the `state` property declared in the ParentComponent.

The FirstChildComponent and SecondChildComponent implement lifecycle methods - `componentDidMount`, `componentDidUpdate` and `componentWillUnmount`.

The ParentComponent implements `componentDidMount` and `componentDidUpdate` methods.

The ParentComponent contains `toggle` and `value` as state properties. The default value of the toggle state property is `true`. This property value is changed in the `handleClick` method of the ParentComponent. This method is bound with the `onClick` event of the Toggle button of the ParentComponent. If the value of the toggle state property is `true`, then the FirstChildComponent will be rendered, else the SecondChildComponent will be rendered.

The FirstChildComponent, contains the `captureMousePositions` method. This method captures the mouse movements and returns the current mouse position. The FirstChildComponent subscribes to the `mousemove` event using `window.addEventListener()` method in the `componentDidMoun`t lifecycle method. The SecondChildComponent displays the value of the `value` state property passed to it from the ParentComponent.

Edit the index.js to import the bootstrap CSS, and the ParentComponent to load bootstrap CSS and to run the ParentComponent by mounting in the `ReactDOM.render()` as shown in Listing 2:

```
import './../node_modules/bootstrap/dist/css/bootstrap.min.css';

import ParentComponent from './components/lifecyclecomponent';

ReactDOM.render(
  <React.StrictMode>
    <ParentComponent />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Listing 2: Mounting the Parent Component

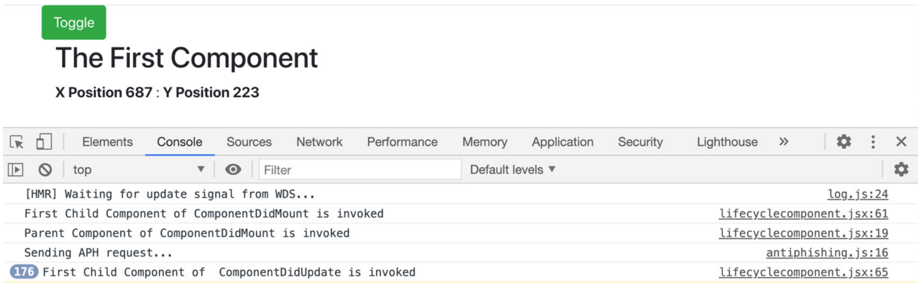
Run the application using the following command from the command prompt:

```
npm run start
```

This command will start the project on port 3000. The browser will launch and will show the following url in the address bar:

```
http://localhost:3000
```

The FirstChildComponent is rendered as shown in Figure 2:

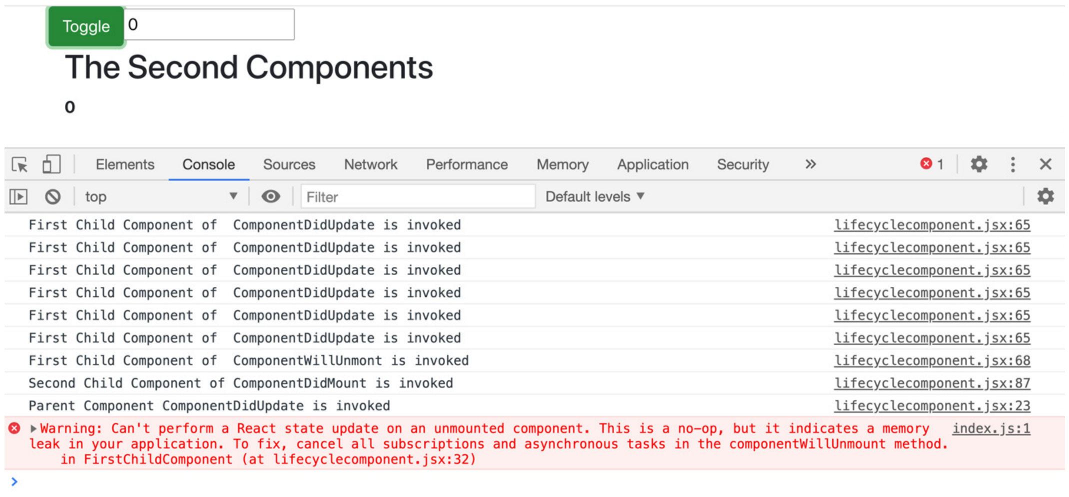


```
[HMR] Waiting for update signal from WDS...                        log.js:24
First Child Component of ComponentDidMount is invoked    lifecyclecomponent.jsx:61
Parent Component of ComponentDidMount is invoked         lifecyclecomponent.jsx:19
Sending APH request...                                        antiphishing.js:16
176 First Child Component of  ComponentDidUpdate is invoked lifecyclecomponent.jsx:65
```

Figure 2: The FirstChildComponent will the log messages in the console window of the browser

As shown in Figure 2, the FirstChildComponent is loaded. Please watch the messages carefully in the console window of the browser (CTRL + SHIFT + J for Chrome and CTRL + SHIFT + K for Firefox), you will find that the FirstChildComponent is mounted first and *then* the Parent component is mounted.

This is exactly like we discussed in the lifecycle as shown in Figure 1 at the beginning of this article. When we start moving our mouse in the browser, the mouse positions will be displayed in the FirstChildComponent. Click on the *Toggle* button, the SecondChildComponent gets mounted as shown in Figure 3:



```
First Child Component of  ComponentDidUpdate is invoked          lifecyclecomponent.jsx:65
First Child Component of  ComponentDidUpdate is invoked          lifecyclecomponent.jsx:65
First Child Component of  ComponentDidUpdate is invoked          lifecyclecomponent.jsx:65
First Child Component of  ComponentDidUpdate is invoked          lifecyclecomponent.jsx:65
First Child Component of  ComponentDidUpdate is invoked          lifecyclecomponent.jsx:65
First Child Component of  ComponentDidUpdate is invoked          lifecyclecomponent.jsx:65
First Child Component of  ComponentWillUnmont is invoked         lifecyclecomponent.jsx:68
Second Child Component of ComponentDidMount is invoked           lifecyclecomponent.jsx:87
Parent Component ComponentDidUpdate is invoked                   lifecyclecomponent.jsx:23
▸Warning: Can't perform a React state update on an unmounted component. This is a no-op, but it indicates a memory   index.js:1
leak in your application. To fix, cancel all subscriptions and asynchronous tasks in the componentWillUnmount method.
    in FirstChildComponent (at lifecyclecomponent.jsx:32)
```

Figure 3: The SecondChildComponent is mounted

Again, pay attention to the messages in the console of the browser. The `componentWillUnMount` method is invoked on the FirstChildComponent and the `componentDidMount` method is invoked on the SecondChildComponent. But the console also shows an error message as:

```
Warning: Can't perform a React state update on an unmounted component. This is
a no-op, but it indicates a memory leak in your application. To fix, cancel all
subscriptions and asynchronous tasks in the componentWillUnmount method. in
FirstChildComponent (at lifecyclecomponent.jsx:32)
```

The reason behind this error is that the `mousemove` event is still attached with the FirstChildComponent and this component is already unloaded. So, when we try to move the mouse on the browser, the `window` object is trying to capture the `mousemove` event and update the state properties *x* and *y* of the FirstChildComponent. But since it's unloaded, it is a **memory leak** situation in the application.

This is the reason why we should spend time understanding the React Component's lifecycle. Since we are subscribing to the mousemove event in the `componentDidMount` method of the `FirstChildComponent`, we have to de-register the event in the `componentWillUnmount` method by adding code as shown in Listing 3:

```
componentWillUnmount=()=>{
  console.log('First Child Component of  ComponentWillUnmont is invoked');
    window.removeEventListener('mousemove', this.captureMousePositions);
}
```

Listing 3: Unmounting the component.

Since we have de-registered the event, when the FirstChildComponent is unmounted, the memory leak error will not occur now. Run the application and verify this.

Here we can conclude that it is important to write code in a React.js component as per the lifecycle so as to avoid errors when dynamically loading these components.

Using React Hooks

So far, we have discussed the lifecycle of React.js applications using class components.

v16.8 of the React.js library introduces a new feature called **Hooks**. The React.js class components' lifecycle can now be handled in new and better way with Hooks. In the next section of this article, we will dive into React Hooks.

What is a Hook?

Using Hooks, we can use all of React features without writing a class. React.js provides State hook named `useState` and Effect hook named `useEffect`. We will explore them shortly.

Hooks are functions that are used to provide state and lifecycle features to a functional component. Hooks do not work in a class component. We need to follow some important rules while working with Hooks.

1. Hooks must not be called in a conditional block, nested functions or loops. We may only call Hooks at the top level of the function component.
2. Hooks may only be called in functional components.

In this article we will use the following Hooks:

1. **useState** - to maintain the state between re-render of the component. The `useState` hook returns a pair consisting of the current state value and a function that is used to update the state. This function can be called in any other event handler to update the state just like the `setState()` method of the class component.

2. **useEffect** - to contain all operations that we write in `componentDidMount()`, `componentWillUnmount()` lifecycle methods of the class component. For example, we write the logic for fetching data from REST API in this hook.

3. **useContext** - to pass data across components. This accepts a context object from the current functional component and returns the current context value for that context. The current context value is determined by the value prop of the nearest component `<DataContext.Provider>` above the calling component in the tree. The called component uses `useContext` to read the data from the value property of the `<DataContext.Provider>`.

There are some additional Hooks provided in React.js v16.8 as listed below (we are not covering these Hooks in this article), but I encourage you to check the official documentation.

- useReducer
- useCallback
- useMemo
- useRef
- useImperativeHandle

- useLayoutEffect
- useDebugValue

In this section, we will use Hooks like **useState**, **useEffect** and **useContext**.

Using Hooks in a React.js Application

Create a React application using the following command:

```
create-react-app hookapp
```

Running this command will create a new React application. Now open this application in VS Code. Open the command prompt in VS Code and navigate to the *hookapp* application folder. Since we will be performing AJAX calls, we need to install the *axios* package. This is a *promise* based object model that has methods to make AJAX calls. We will install this package using the following command:

```
npm install --save axios
```

We will be using Bootstrap CSS classes for the UX. To install bootstrap for the current project, open the command prompt, navigate to the project path and run the following command to install bootstrap:

```
npm install --save bootstrap
```

Step 1: In the src folder, add a new folder and name it as *services*. In this folder, add a new file and name it as httpservice.js. In this file, add code as shown in Listing 4:

```
import axios from 'axios';

export class HttpService {
    constructor(){
        this.url ='https://apiapptrainingnewapp.azurewebsites.net/api/Products';
    }

    getData() {
        let response = axios.get(this.url);
        return response;
    }

    postData(prd) {
        let response = axios.post(this.url, prd, {
            'Content-Type' : 'application/json'
        });
        return response;
    }

    putData(prd) {
        let response = axios.put(`${this.url}/${prd.ProductRowId}`, prd, {
            'Content-Type' : 'application/json'
        });
        return response;
    }
    deleteDaat(id) {
        let response = axios.delete(`${this.url}/${id}`);
        return response;
    }
}
```
Listing 4: The HTTP Service that makes AJAX calls to REST APIs

The code in Listing 4 shows methods of the *axios* object that make AJAX calls to the REST APIs.

Step 2: In the *src* folder, add a new folder and name it as 'components'. In this folder, add a new file and name it as datacontext.js. In this file, add the code as shown in Listing 5

```
import {createContext} from 'react'
export const DataContext = createContext(null);
```

Listing 5: The DataContext definition using createContext

`createContext()` is used to create a context object and pass data across components. When React renders a component that subscribes to this Context object, it will read the current context value from the closest matching Provider above it in the tree, from the parent component. The parent component uses the Context to provide value so that the child component reads values from the parent component.

Step 3: In the components folder, add a new file and name it as TableComponent.jsx - a reusable table component. In this file, add code as shown in Listing 6:

```
import React, { useContext } from 'react';
import {DataContext} from './datacontext';
const TableComponent =() => {
  // subscribe to DataContext and read provider values
  const dataContext = useContext(DataContext);
  // {{products, updateProduct}}
  // if the DataContext is complex object, then propeties
  // from it will be read using the reflection and indexes
  const dataSource = dataContext[Object.keys(dataContext)[0]]; // products array
  const event = dataContext[Object.keys(dataContext)[1]]; // read the method

  const tableColumns = [];
  for(const p in dataSource[0]){
    tableColumns.push(p)
  }
  return (
    <table className="table table-bordered table-striped">
      <thead>
        <tr>
          {
            tableColumns.map((c,idx) => (
              <th key={idx}>{c}</th>
            ))
          }
        </tr>
      </thead>
      <tbody>
      {
        dataSource.map((d,i)=> (
          <tr key={i} onClick={()=>event(d)}>
          {
            tableColumns.map((c,idx) => (
              <td key={idx}>{d[c]}</td>
            ))
          }
          </tr>
        ))
      }
      </tbody>
    </table>
  );
};

export default TableComponent;
```

Listing 6: The TableComponent

The code in the Listing 6, shows the `TableComponent` functional component. This component uses the `useContext` hook to read values from the `DataContext` which we declared in datacontext.js as shown in Listing 5. The `TableComponent` subscribes to the context and reads data sent by the sender component. This component dynamically generates an HTML table based on the data received from the context. The `TableComponent` binds the method from the value of the context, to the click event of the table row. This will emit value from the `TableComponent` to the parent component that uses TableComponent having child component.

Step 4: In the components folder, add a new file and name it as ProductHookComponent.jsx. In this file add the code as shown in Listing 7:

```
import React, {useState, useEffect} from 'react'
import TableComponent from './TableComponent';
import {DataContext} from './datacontext';
import {HttpService} from './../services/httpservice';

const ProductHookComponent=()=>{
  const service = new HttpService();
  const categories =['Electronics', 'Electrical', 'Food'];
  const manufacturers = ['MSIT', 'TSFOODS', 'LS-Electrical'];
  const [product, updateProduct] = useState({ProductRowId:0, ProductId:'',
                                              ProductName: '', CategoryName:'',
                                              Manufacturer:'', Description:'',
BasePrice:0});
  const [products, updateProducts] = useState([]);

  const clear=()=>(
    updateProduct({ProductRowId:0, ProductId:'',
     ProductName: '', CategoryName:'',
     Manufacturer:'', Description:'', BasePrice:0})
  );
  useEffect(()=>{
    service.getData()
      .then(response=>{
        updateProducts(response.data);
      })
      .catch(error=>{
        console.log(`Error Occured ${error}`);
      });

  }, []);

  const save=()=>{
    service.postData(product).then(response=>{
      updateProduct(response.data);
      updateProducts([...products, response.data]);
    })
    .catch(error=>{
      console.log(`Error Occured ${error}`);
    });
  }
  return (
    <div className="container">
      <table className="table table-bordered table-striped">
      <tbody>
        <tr>
          <td>
            <div className="form-group">
              <label>Product Row Id</label>
              <input type="text" className="form-control" value={product.
              ProductRowId} readOnly/>
            </div>
            <div className="form-group">
              <label>Product Id</label>
              <input type="text" className="form-control" value={product.ProductId}
              onChange={(evt)=>{updateProduct({...product, ProductId:evt.target.
              value})}}/>
            </div>
            <div className="form-group">
              <label>Product Name</label>
              <input type="text" className="form-control" value={product.
              ProductName}
              onChange={(evt)=>{updateProduct({...product, ProductName:evt.target.
              value})}}/>
            </div>
            <div className="form-group">
              <label>Category Name</label>

                <select type="text" className="form-control"
                name="CategoryName" value={product.CategoryName}
                onChange={(evt)=>{updateProduct({...product, CategoryName:evt.
                target.value})}}>
                  <option>Select Category Name</option>
                  {
                    categories.map((v,i)=> (
                      <option key={i} value={v}>{v}</option>
                    ))
                  }
                </select>
             </div>
             <div className="form-group">
               <label>Manufacturer</label>
               <select type="text" className="form-control"
               value={product.Manufacturer}
                 onChange={(evt)=>{
                   updateProduct({...product, Manufacturer:evt.target.value});
                 }}
               >
               <option>Select Manufacturer</option>
               {

                 manufacturers.map((v,i)=> (
                   <option key={i} value={v}>{v}</option>
                 ))
               }
             </select>
           </div>
           <div className="form-group">
             <label>Description</label>
               <input type="text" className="form-control" value={product.
               Description}
             onChange={(evt)=>{updateProduct({...product, Description:evt.target.
             value})}}/>
           </div>
           <div className="form-group">
             <label>Base Price</label>
               <input type="text" className="form-control" value={product.
               BasePrice}
               onChange={(evt)=>{updateProduct({...product, BasePrice:parseInt(evt.
               target.value)})}}/>
           </div>
           <div className="form-group">
             <input type="button" value="Clear" className="btn btn-warning"
             onClick={clear}/>
             <input type="button" value="Save" className="btn btn-success"
             onClick={save}/>
           </div>
         </td>
       </tr>
       <tr>
         <td>
           <h2>
             List of Products
           </h2>
           <DataContext.Provider value={{products, updateProduct}}>
             <TableComponent></TableComponent>
           </DataContext.Provider>
         </td>
       </tr>
     </tbody>
   </table>
 </div>
 );
}

export default ProductHookComponent;
```

Listing 7: The ProductHookComponent

The code in Listing 7 has the following specifications:

1. The code defines the functional component of the name `ProductHookComponent`.

2. This component uses the `HttpService` class instance.

3. Constant arrays are declared for Manufacturers and Categories.

4. The state is defined using **useState** Hook. The product and products state objects are defined using `useState`. The `updateProduct()` and `updateProducts()` are the methods used to update product and products state objects respectively.

5. `clear()` is a nested function that is used to clear all properties of product object.

6. The **useEffect** hook is used to invoke the `getData()` method from the service class. Please note that this hook is called at the component level.

   This hook accepts two parameters. The first parameter is a callback function that performs an AJAX call. The second parameter is a dependency parameter. This parameter represents that the effect will be activated only when the dependency is updated. If the dependency parameter is not passed, then useEffect will keep on making AJAX calls. We have to make sure that the dependency parameter must be passed to prevent the repeated execution of the callback.

   You may have a question - **why to pass dependency parameter**?

   The *useEffect()* hook runs after every completed render. So, we need to make sure that after the effect completes its task, we should execute cleanup operation e.g. unsubscribing an event, etc. So the advantage here is if the component renders multiple times, the previous effect is cleaned up before an execution of the next effect. This is where the need of an empty dependency array arises. When we want to run effect and clean it up only once, an empty array is passed as the second argument to the useEffect() hook. This empty array parameter informs React that an effect does not depend on any value of the component (e.g. state and props), so it will never re-run again and again.

7. The `save()` nested method is used to invoke the `postData()` method from the service class. The `save()` method subscribes to the promise object returned from the postData() method. If the promise is successfully executed, then the product and products state properties are updated.

8. The functional component returns HTML. The properties of product state property of component are bound with input elements. To update properties of the product state object, the `onChange` event is defined that updates the properties of the product state object using the object mutation.

9. The `<DataContext.Provider value={{products, updateProduct}}></DataContext.Provider>`, is a context used by the ProductHookComponent to pass the data from the ProductHookComponent to the TableComponent.

   The `value` attribute of the context sends products state object and `updateProduct` method to the TableComponent. The TableComponent will use these values to dynamically generate the table and emit the selected row value from the table component to the ProductHookComponent.

   The TableComponent will re-render for every change in the value send through the context. The question here is, **can we not use props to pass data from ProductHookComponent to TableComponent?** The advantage of using context is that the context provides a way to pass data through the component tree *without* having to pass props values down manually to every component in the tree, which is a cumbersome task.

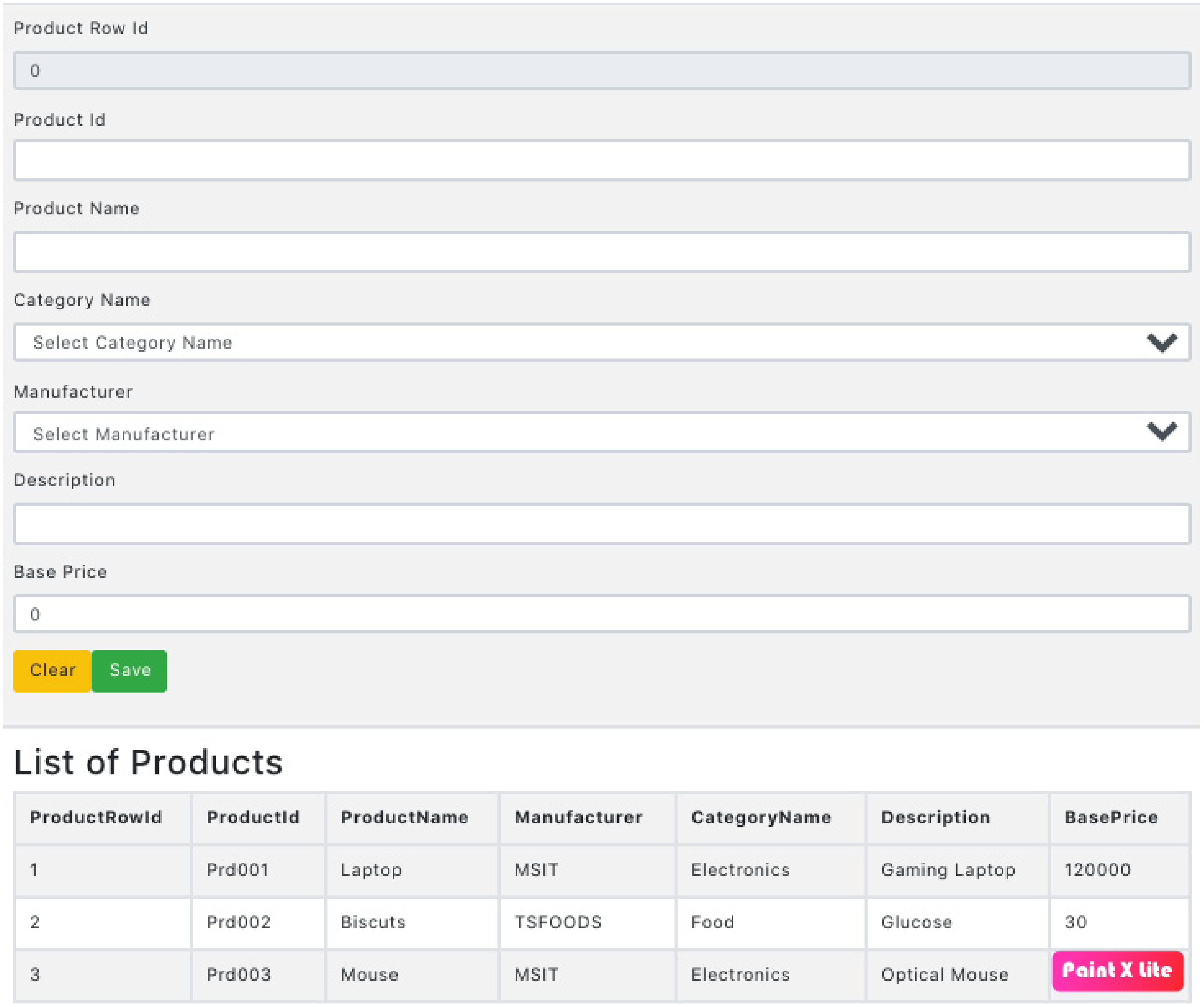
Step 5: Modify index.js and import Bootstrap in it to load CSS classes and also import the ProductHookComponent from the ProductHookComponent.jsx. Mount the ProductHookComponent in `ReactDom.render()` method. The code of the index.js is shown in the Listing 8:

```
import './../node_modules/bootstrap/dist/css/bootstrap.min.css';
import * as serviceWorker from './serviceWorker';
import ProductHookComponent from './components/ProductHookComponent';

ReactDOM.render(
  <React.StrictMode>
    <ProductHookComponent />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Listing 8: The index.js that is importing bootstrap and the ProductHookComponent

Run the application from the command prompt using the command `npm run start`. This will host the application on port 3000. The React application gets loaded in the browser as shown in Figure 4.



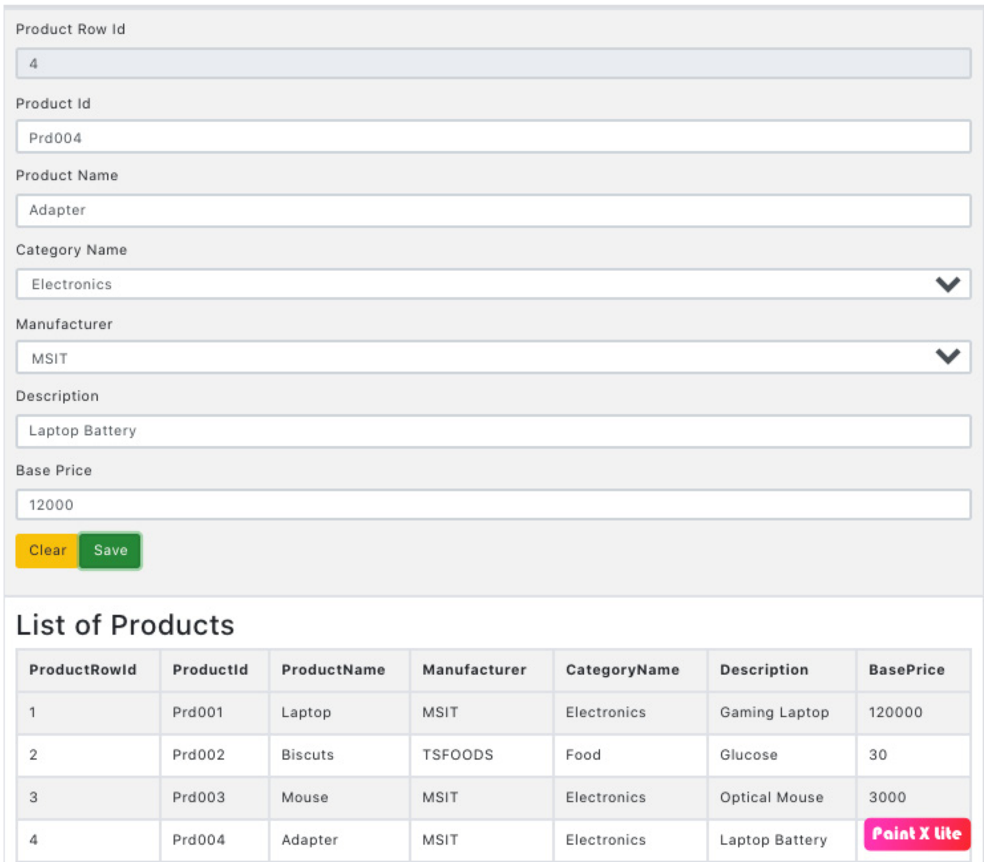
Product Row Id	0
Product Id	
Product Name	
Category Name	Select Category Name
Manufacturer	Select Manufacturer
Description	
Base Price	0

List of Products

ProductRowId	ProductId	ProductName	Manufacturer	CategoryName	Description	BasePrice
1	Prd001	Laptop	MSIT	Electronics	Gaming Laptop	120000
2	Prd002	Biscuts	TSFOODS	Food	Glucose	30
3	Prd003	Mouse	MSIT	Electronics	Optical Mouse	Paint X lite

Figure 4: The React application

Enter Products details in the input elements and select the CategoryName and Manufacturer from the Dropdown. Click on the **Save** button, and a record gets posted to the REST API and saved in the remote database. The newly added record will be updated in the products state array as shown in Figure 5:



Product Row Id	4
Product Id	Prd004
Product Name	Adapter
Category Name	Electronics
Manufacturer	MSIT
Description	Laptop Battery
Base Price	12000

List of Products

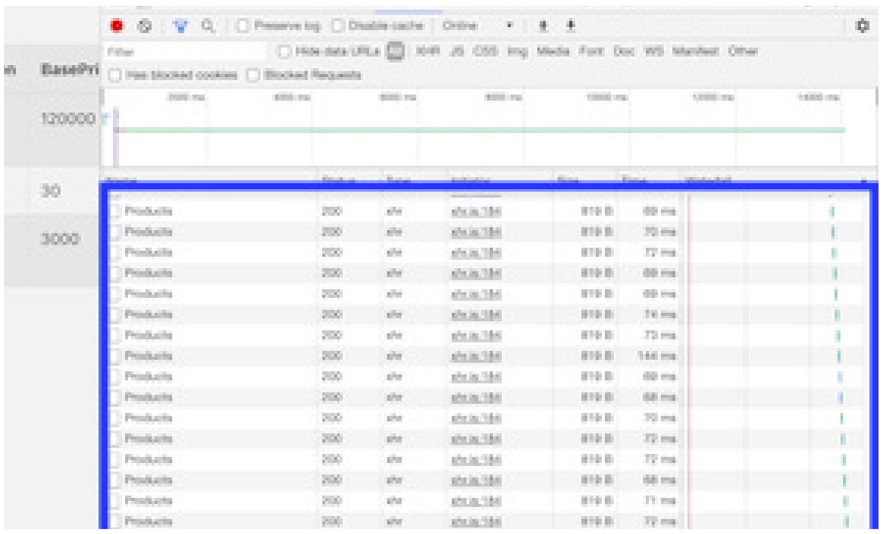
ProductRowId	ProductId	ProductName	Manufacturer	CategoryName	Description	BasePrice
1	Prd001	Laptop	MSIT	Electronics	Gaming Laptop	120000
2	Prd002	Biscuits	TSFOODS	Food	Glucose	30
3	Prd003	Mouse	MSIT	Electronics	Optical Mouse	3000
4	Prd004	Adapter	MSIT	Electronics	Laptop Battery	

Figure 5: Added Record

Likewise, click on the table row to select the product, the selected product will be displayed in input elements.

As explained in Step 4, the *useEffect* hook requires the dependency parameter to activate the hook. If this parameter is not passed, the hook will continuously make AJAX calls.

To test this, modify the ProductHookComponent code for useEffect and remove the array parameter passed as dependency to it. Now run the application in your browser and from the **Network** tab of your browser's tools, you can see these frequent AJAX calls as shown in Figure 6:



Products	200	xhr	[illegible]	819 B	69 ms
Products	200	xhr	[illegible]	819 B	70 ms
Products	200	xhr	[illegible]	819 B	72 ms
Products	200	xhr	[illegible]	819 B	69 ms
Products	200	xhr	[illegible]	819 B	69 ms
Products	200	xhr	[illegible]	819 B	74 ms
Products	200	xhr	[illegible]	819 B	73 ms
Products	200	xhr	[illegible]	819 B	144 ms
Products	200	xhr	[illegible]	819 B	69 ms
Products	200	xhr	[illegible]	819 B	68 ms
Products	200	xhr	[illegible]	819 B	70 ms
Products	200	xhr	[illegible]	819 B	72 ms
Products	200	xhr	[illegible]	819 B	72 ms
Products	200	xhr	[illegible]	819 B	68 ms
Products	200	xhr	[illegible]	819 B	71 ms
Products	200	xhr	[illegible]	819 B	72 ms

Figure 6: useEffect dependencies removed for frequent AJAX calls

Here we can understand **the importance of the dependency parameter of the useEffect hook**.

Conclusion:

React.js, with class components and Hooks, can be used to develop next-gen web front end applications.

In this tutorial on React.js Lifecyle events and Hooks, we saw that the lifecycle events of class components must be used appropriately to make sure that all events attached to the window object must be detached.

The `componentDidMount()` and `componentWillUnmount()` methods of the class component are used to manage long running operations and cleaning operations respectively. The `useEffect()` hook combines `componentDidMount()` and `componentWillUnmount()` methods of the class component. The dependency parameter of the useEffect makes sure that the useEffect hook is executed correctly.

Download the entire source code from GitHub at
http://bit.ly/dncm48-reactjshooks

• • • • • • •

Mahesh Sabnis
Author

Microsoft® MVP Most Valuable Professional

Mahesh Sabnis is a DotNetCurry author and a Microsoft MVP having over two decades of experience in IT education and development. He is a Microsoft Certified Trainer (MCT) since 2005 and has conducted various corporate training programs for .NET technologies (all versions), and Front-end technologies like Angular and React. Follow him on twitter @ maheshdotnet or connect with him on LinkedIn.

Techinical Review

Damir Arh

Editorial Review

Suprotim Agarwal

Daniel Jimenez Garcia

KUBERNETES for Developers – Introduction, Architecture, Hands-On and much more

In a world where most developers and organizations have come to accept the constant churn of new tools, architectures and paradigms, containerized applications are slowly becoming a lingua franca for developing, deploying and operating applications. In no small part, this is thanks to the popularity and adoption of tools like Docker and Kubernetes.

*It was back in 2014 when Google first announced Kubernetes, an open source project aiming to **automate the deployment, scaling and management of containerized applications**. With its version 1.0 released back in July 2015, Kubernetes has since experienced a sharp increase in adoption and interest shared by both organizations and developers.*

Refreshingly, both organizations and developers seem to share the growing interest in Kubernetes. Surveys note how interest and adoption across the enterprise continues on the rise. A the same time, the latest Stack Overflow's developer survey rates Kubernetes as the 3rd most loved and most wanted technology (with Docker rating even better!).

But none of this information will make it any easier for newcomers!

In addition to the learning curve for developing and packaging applications using containers, Kubernetes introduces its own specific concepts for deploying and operating applications.

The aim of this Kubernetes tutorial is to guide you through the basic and most useful Kubernetes concepts that you will need *as a developer*. For those who want to go beyond the basics, I will include pointers for various subjects like persistence or observability. I hope you find it useful and enjoy the article!

Kubernetes architecture

You are not alone if you are still asking yourself ***What is Kubernetes?***

Let's begin the article with a high-level overview of the purpose and architecture of Kubernetes.

If you check the official documentation, you will see the following definition:

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation

In essence, this means Kubernetes is a *container orchestration engine*, a platform designed to host and run containers across a number of nodes.

To do so, Kubernetes abstracts the nodes where you want to host your containers as a pool of cpu/memory resources. When you want to host a container, you just *declare* to the Kubernetes API the details of the container (like its image and tag names and the necessary cpu/memory resources) and Kubernetes will transparently host it *somewhere* across the available nodes.

In order to do so, the architecture of Kubernetes is broken down into several components that track the desired state (ie, the containers that users wanted to deploy) and apply the necessary changes to the nodes in order to achieve that desired state (ie, adds/removes containers and other elements).

The nodes are typically divided in two main sets, each of them hosting different elements of the Kubernetes architecture depending on the *role* these nodes will play:

- **The control plane**. These nodes are the brain of the cluster. They contain the Kubernetes components that track the desired state, the current node state and make container scheduling decisions across the worker nodes.

- **The worker nodes**. These nodes are focused on hosting the desired containers. They contain the Kubernetes components that ensure containers assigned to a given node are created/terminated as decided by the control plane

The following picture shows a typical cluster, composed of several control plane nodes and several worker nodes.

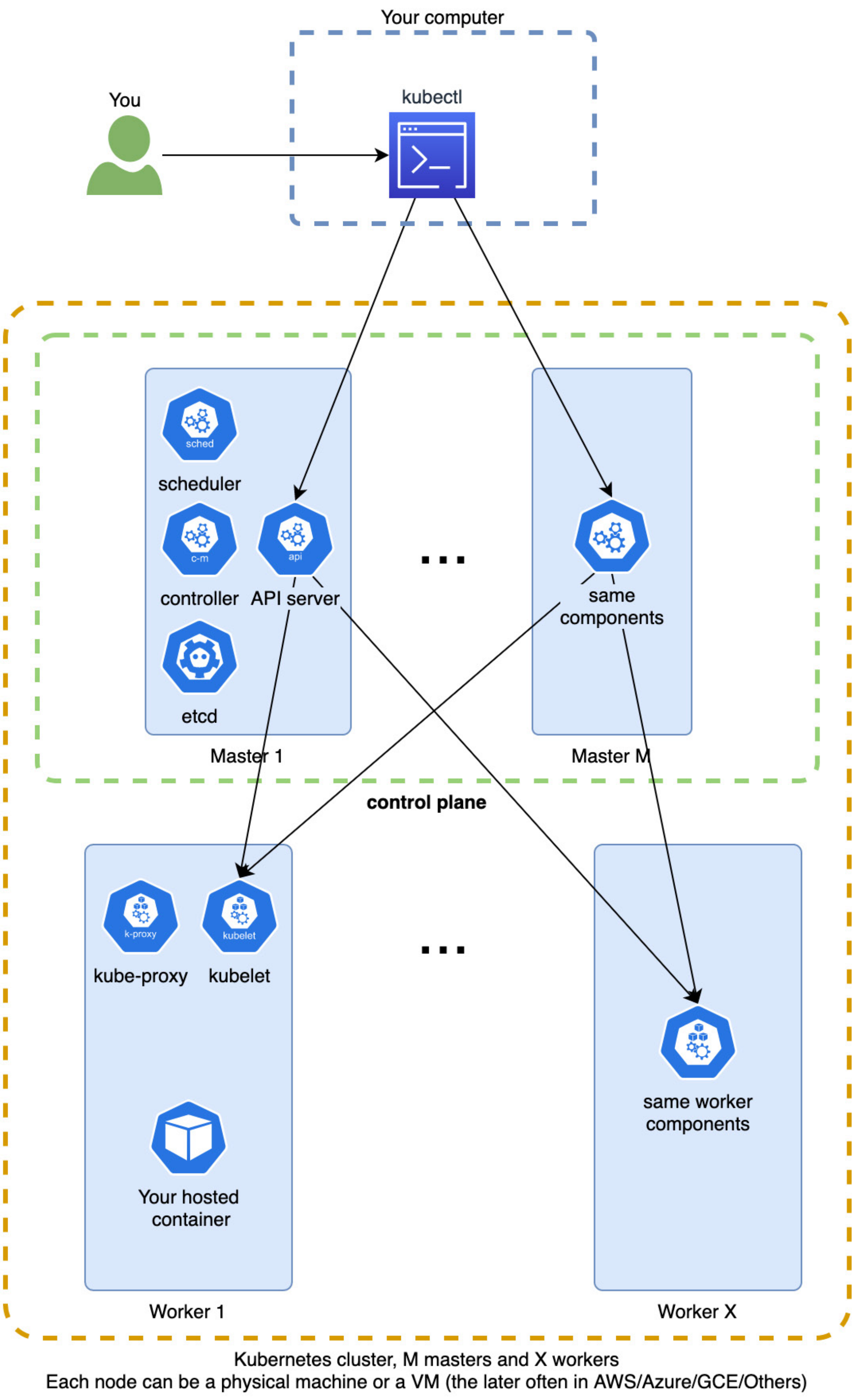


Figure 1, High level architecture of a Kubernetes cluster with multiple nodes

As a developer, you *declare* which containers you want to host by using YAML that follow the API. The simplest way to host the hello-world container from Docker would be declaring a Pod (more on this as soon as we are done with the introduction).

This is something that looks like:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
spec:
  containers:
  - image: hello-world
    name: hello
```

The first step is getting the container you want to host, described in YAML using the Kubernetes objects like Pods.

Then you will need to interact with the cluster via its API and ask it to *apply* the objects you just described. While there are several clients you could use, I would recommend getting used to **kubectl**, the official CLI tool. We will see how to get started in a minute, but an option would be saving the YAML as `hello.yaml` and run the command

```
kubectl apply -f hello.yaml
```

This concludes a brief and simplified overview of Kubernetes. You can read more about the Kubernetes architecture and main components in the official docs.

Kubernetes - Getting started

The best way to learn about something is by trying for yourself.

Luckily with Kubernetes, there are several options you can choose to get your own small cluster! You don't even need to install anything on your machine. If you want to, you could just use an online learning environment like **Katacoda**.

Setup your local environment

The main thing you will need is a way to create a Kubernetes cluster in your local machine. While Docker for Windows/Mac comes with built-in support for Kubernetes, I recommend using **minikube** to setup your local environment. The addons and extra features minikube provides, simplifies many of the scenarios you might want to test locally.

Minikube will create a very simple cluster with a single Virtual Machine where all the Kubernetes components will be deployed. This means you will need some container/virtualization solution installed on your machine, of which minikube supports a long list. Check the prerequisites and installation instructions for your OS in the official minikube docs.

In addition, you will also need **kubectl** installed on your local machine. You have various options:

- Install it locally in your machine, see the instructions in the official docs.

- Use the version provided by minikube. Note that if you choose this option, when you see a command like `kubectl get pod` you will need to replace it with `minikube kubectl -- get pod`.

- If you previously enabled Kubernetes support in Docker, you should already have a version of kubectl installed. Check with `kubectl version`.

Once you have both minikube and kubectl up and running in your machine, you are ready to start. Ask minikube to create your local cluster with:

```
$ minikube start
...
Done! kubectl is now configured to use "minikube" by default
```

Check that everything is configured correctly by ensuring your local kubectl can effectively communicate with the minikube cluster

```
$ kubectl get node
NAME       STATUS   ROLES    AGE     VERSION
minikube   Ready    master   9m58s   v1.19.2
```

Finally, let's enable the *Kubernetes dashboard* addon.

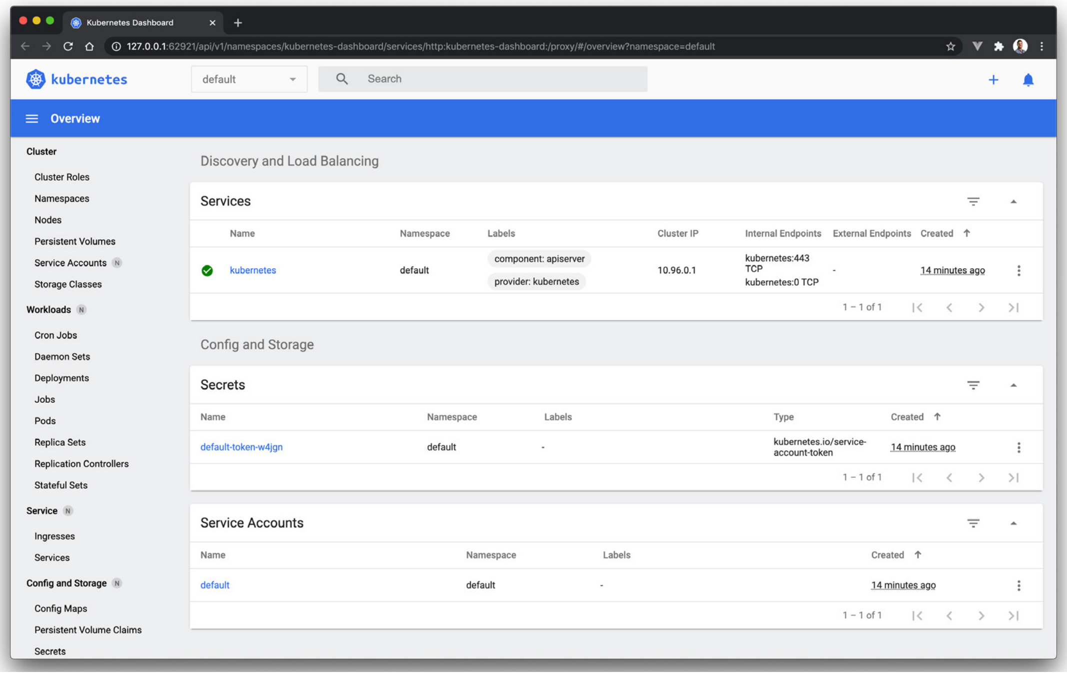
This addon deploys to the minikube cluster the official dashboard that provides a nice UX for interacting with the cluster and visualize the currently hosted apps (even though we will continue to use kubectl through the article).

```
$ minikube addons enable dashboard
...
□  The 'dashboard' addon is enabled
```

Finally open the dashboard by running the following command on a second terminal

```
$ minikube dashboard
```

The browser will automatically launch with the dashboard:



Discovery and Load Balancing

Services

Name	Namespace	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created ↑
kubernetes	default	component: apiserver provider: kubernetes	10.96.0.1	kubernetes:443 TCP kubernetes:0 TCP	-	14 minutes ago

1 – 1 of 1

Config and Storage

Secrets

Name	Namespace	Labels	Type	Created ↑
default-token-w4jgn	default	-	kubernetes.io/service-account-token	14 minutes ago

1 – 1 of 1

Service Accounts

Name	Namespace	Labels	Created ↑
default	default	-	14 minutes ago

1 – 1 of 1

Figure 2, Kubernetes dashboard running in your minikube environment

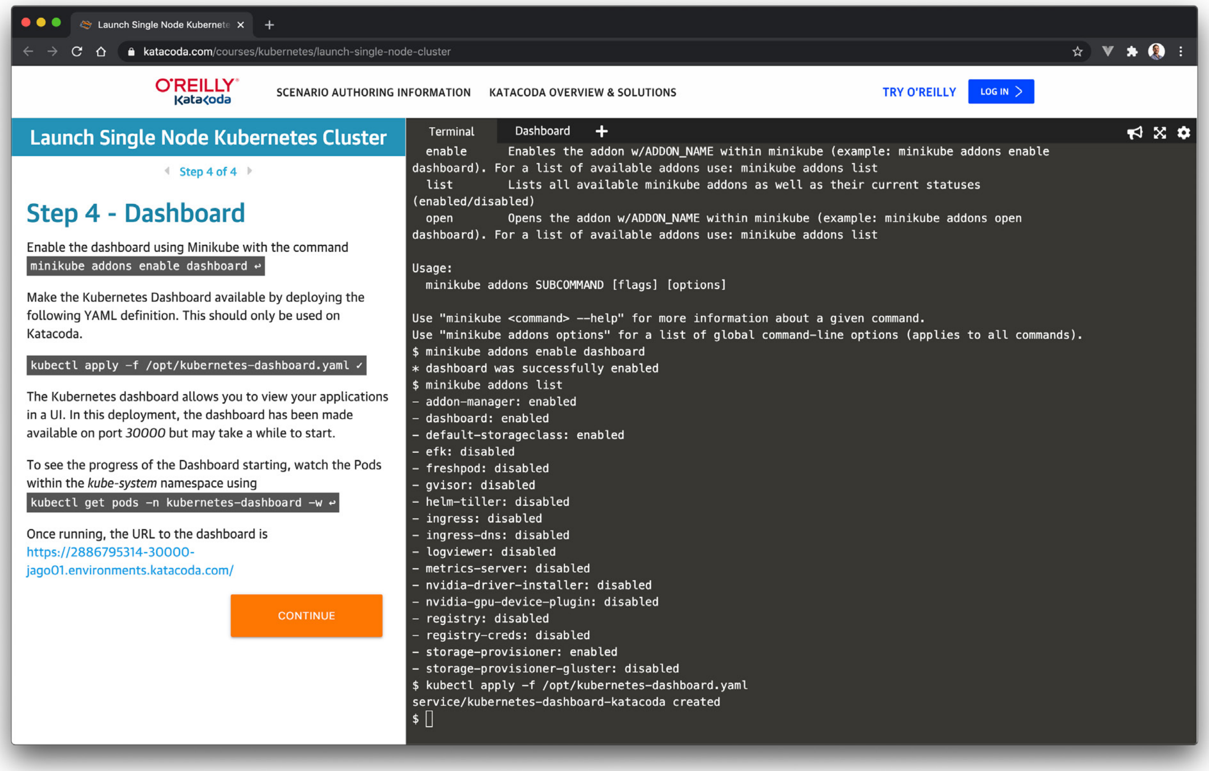
Use the online Katacoda environment

If you don’t want to or you can’t install the tools locally, you can choose a free online environment by using **Katacoda**.

Complete the four steps of the following scenario in order to get your own minikube instance hosted by Katacoda: https://www.katacoda.com/courses/kubernetes/launch-single-node-cluster

At the time of writing, the scenario is free and does not require you to create an account.

By completing all the scenario steps, you will create a Kubernetes cluster using minikube, enable the dashboard addon and launch the dashboard on another tab of your browser. By the end of the four simple steps, you will have a shell where you can run `kubectl` commands, and the dashboard opened in a separated tab.


Launch Single Node Kubernete
katacoda.com/courses/kubernetes/launch-single-node-cluster
O'REILLY Katacoda
SCENARIO AUTHORING INFORMATION
KATACODA OVERVIEW & SOLUTIONS
TRY O'REILLY
LOG IN
Terminal
Dashboard
CONTINUE

Launch Single Node Kubernetes Cluster

Step 4 of 4

Step 4 - Dashboard

Enable the dashboard using Minikube with the command `minikube addons enable dashboard`

Make the Kubernetes Dashboard available by deploying the following YAML definition. This should only be used on Katacoda.

`kubectl apply -f /opt/kubernetes-dashboard.yaml`

The Kubernetes dashboard allows you to view your applications in a UI. In this deployment, the dashboard has been made available on port *30000* but may take a while to start.

To see the progress of the Dashboard starting, watch the Pods within the *kube-system* namespace using `kubectl get pods -n kubernetes-dashboard -w`

Once running, the URL to the dashboard is https://2886795314-30000-jago01.environments.katacoda.com/

```
  enable      Enables the addon w/ADDON_NAME within minikube (example: minikube addons enable
dashboard). For a list of available addons use: minikube addons list
  list        Lists all available minikube addons as well as their current statuses
(enabled/disabled)
  open        Opens the addon w/ADDON_NAME within minikube (example: minikube addons open
dashboard). For a list of available addons use: minikube addons list

Usage:
  minikube addons SUBCOMMAND [flags] [options]

Use "minikube <command> --help" for more information about a given command.
Use "minikube addons options" for a list of global command-line options (applies to all commands).
$ minikube addons enable dashboard
* dashboard was successfully enabled
$ minikube addons list
- addon-manager: enabled
- dashboard: enabled
- default-storageclass: enabled
- efk: disabled
- freshpod: disabled
- gvisor: disabled
- helm-tiller: disabled
- ingress: disabled
- ingress-dns: disabled
- logviewer: disabled
- metrics-server: disabled
- nvidia-driver-installer: disabled
- nvidia-gpu-device-plugin: disabled
- registry: disabled
- registry-creds: disabled
- storage-provisioner: enabled
- storage-provisioner-gluster: disabled
$ kubectl apply -f /opt/kubernetes-dashboard.yaml
service/kubernetes-dashboard-katacoda created
$
```

Figure 3, setting up minikube online with Katacoda

I will try to note through the article when you will experience different behaviour and/or limitations when using the Katacoda online environment. This might be particularly the case when going through the networking and persistence sections.

Hosting containers: Pods, Namespaces and Deployments

Now that we have a working Kubernetes environment, we should put it to good use. Since one of the best ways to demystify Kubernetes is by using it, **the rest of the article is going to be very hands on.**

Pod is the unit of deployment in Kubernetes

In the initial architecture section, we saw how in order to deploy a container, you need to create a Pod. That is because the minimum *unit of deployment* in a Kubernetes cluster is a Pod, rather than a container.

- This indirection might seem arbitrary at first. **Why not use a container as the unit of deployment rather than a Pod?** The reason is that Pods can include more than just a container, where all containers in the same pod share the same network space (they see each other as localhost) and the same *volumes* (so they can share data as well). Some advanced patterns take advantage of several containers like init containers or sidecar containers.

- That's not to say you should add all the containers that make your application inside a single Pod. Remember a Pod is the unit of deployment, so you want to be able to deploy and scale different Pods independently. As a rule of thumb, different application services or components should become independent Pods.

- In addition, while Pods are first type of Kubernetes object we will see through the article, soon we will see others like Namespaces and Deployments.

If this seems confusing, you can assume for the rest of the article that *Pod == 1 container* as we won't use the more advanced features. As you get more comfortable with Kubernetes, the difference between Pod and container will became easier to understand.

In order to deploy a Pod, we will describe it using YAML. Create a file hello.yaml with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
spec:
  containers:
  - image: hello-world
    name: hello
```

These files are called *manifests*. It is just YAML that describes one or more Kubernetes objects. In the previous case, the manifest file we have created contains a single object:

- The `kind: Pod` indicates the type of object.

- The `metadata` allows us to provide additional meta information about the object. In this case, we have just given the Pod a name. This is the minimum metadata piece required so Kubernetes can identify each Pod. We will see more in a minute when we take a look at the idea of **Namespace**.

- The `spec` is a complex property that describes what you want the cluster to do with this particular Pod. As you can see, it contains a list of `containers` where we have included the single container we want to host. The container `image` tells Kubernetes where to download the image from. In this case, this is just the name of a public docker image found in Docker Hub, the `hello-world` container.

- Note you are not restricted to using public Docker Hub images. You can also use images from private registries as in `registry.my-company.com/my-image:my-tag` (Setting up a private registry and configuring the cluster with credentials is outside the article scope. I have however provided some pointers at the end of the article).

Let's ask the cluster to host this container. Simply run the `kubectl` command

```
$ kubectl apply -f hello.yaml
pod/my-first-pod created
```

In many terminals you can use a Heredoc multiline string and directly pass the manifiest without the need to create a file. For example in mac/linux you can use:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
spec:
  containers:
  - image: hello-world
    name: hello
EOF
```

Get comfortable writing YAML and applying it to the cluster. We will be doing that a lot through the article! If you want to keep editing and applying the same file, you just need to separate the different objects with a new line that contains: `---`

Once created, we can then use `kubectl` to inspect the list of existing Pods in the cluster. Run the following command:

```
$ kubectl get pod
NAME           READY   STATUS             RESTARTS   AGE
my-first-pod   0/1     CrashLoopBackOff   4          2m45s
```

Note, it shows as not ready and has status of CrashLoopBackOff. That is because Kubernetes expects Pod containers to keep running. However, the hello-world container we have used simply prints a message to the console and terminates. Kubernetes sees that as a problem with the Pod and tries to restart it.

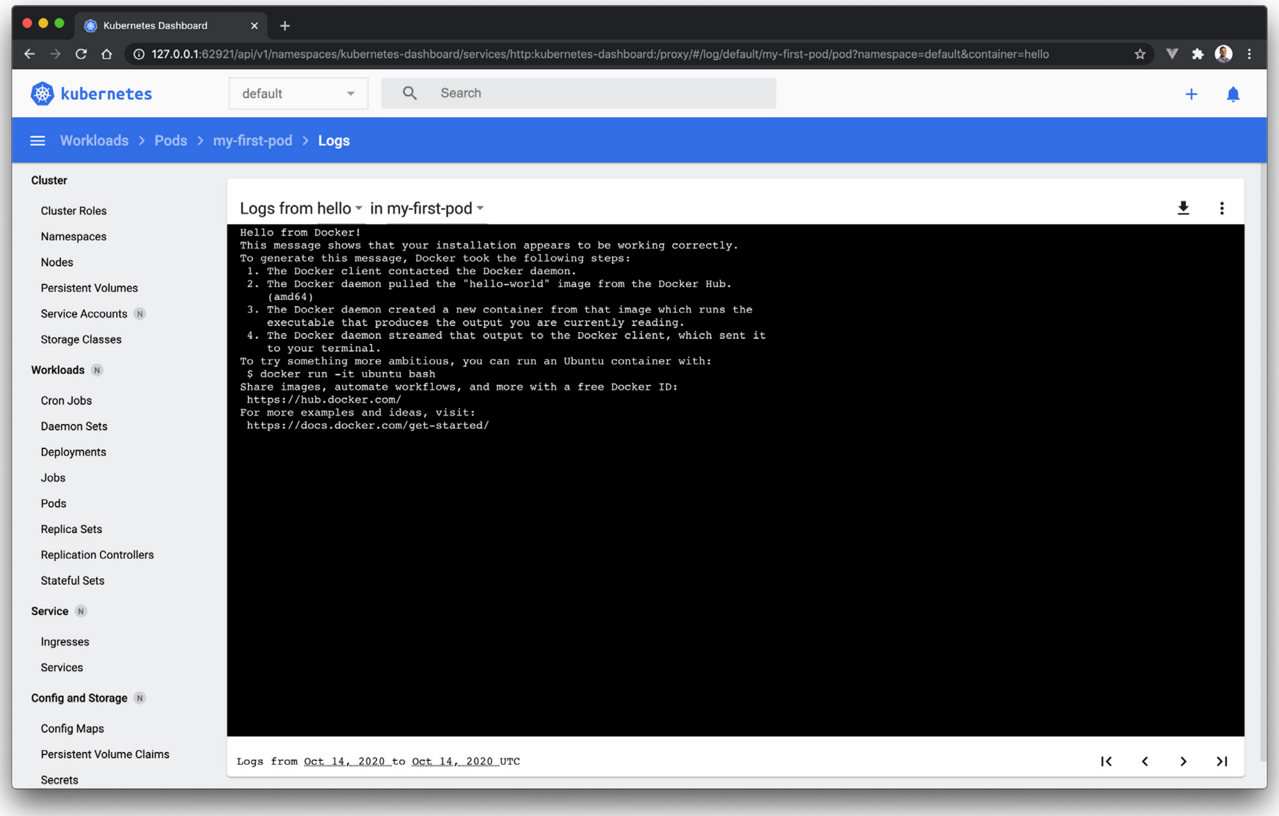
There are specialized Kubernetes objects such as Jobs or Cronjobs that can be used for cases where containers are not expected to run forever.

Let's verify that it indeed runs as expected by inspecting the logs of the pod's container (i.e., what the container wrote to the standard output). Given we have just run the `hello-world` container, we expect to see a greeting message written to the console:

```
$ kubectl logs my-first-pod

Hello from Docker!
This message shows that your installation appears to be working correctly.
... omitted ...
For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

You can also see the Pod in the dashboard, as well as its container logs:



```
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/
For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figure 4, Inspecting the container logs in the dashboard

If you want to cleanup and remove the Pod, you can use either of these commands:

```
kubectl delete -f hello.yaml
kubectl delete pod my-first-pod4
```

Organizing Pods and other objects in Namespaces

Now that we know what a Pod is and how to create them, lets introduce the `Namespace` object. The purpose of the Namespace is simply to organize and group the objects created in a cluster, like the Pod we created before.

If you imagine a real production cluster, there will be many Pods and other objects created. By grouping them in Namespaces, developers and administrators can define policies, permissions and other settings that affect all objects in a given Namespace, without necessarily having to list each individual object. For example, an administrator could create a production Namespace and assign different permissions and network policies to any object that belong to that namespace.

If you list the existing namespaces with the following command, you will see the cluster already contains quite a few Namespaces. That's due to the system components needed by Kubernetes and the addons enabled by minikube:

```
$ kubectl get namespace
NAME                   STATUS   AGE
default                Active   125m
kube-node-lease        Active   125m
kube-public            Active   125m
```

```
kube-system            Active   125m
kubernetes-dashboard   Active   114m
```

You can see the objects in any of these namespaces by adding the `-n {namespace}` parameter to a kubectl command. For example, you can see the dashboard pods with:

```
kubectl get pod -n kubernetes-dashboard
```

This means in Kubernetes, every object you create has at least two metadata values which are used for identification and management:

- The **namespace**. This is **optional**, when omitted the `default` namespace is used.
- The **name** of the object. This is **required** and must be unique within the namespace.

The Pod we created earlier used the default namespace since no specific one was provided. Let's create a couple of namespaces and a Pod inside each namespace:

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace1
---
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace2
---
apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
  namespace: my-namespace1
spec:
  containers:
  - image: hello-world
    name: hello
---
apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
  namespace: my-namespace2
spec:
  containers:
  - image: hello-world
    name: hello
```

There are a few interesting bits in the manifest above:

- You create namespaces by describing them in the manifest. A namespace is just an object with `kind: Manifest`.
- Namespaces have to be created *before* they can be referenced by Pods and other objects. (Otherwise you would get an error)
- Note how both pods have the same name. That is fine since they belong to different namespaces.

During the rest of the tutorial, we will keep things simple and use the default namespace.

Running multiple replicas with Deployments

Kubernetes would hardly achieve its state goals *automate the deployment, scaling and management of containerized applications* if you had to manually manage each single container.

While Pods are the foundation and the basic unit for deploying containers, you will hardly use them directly. Instead you would use a higher level abstraction like the Deployment object. The purpose of the Deployment is to abstract the creation of multiple *replicas* of a given Pod.

With a Deployment, rather than directly creating a Pod, you give Kubernetes a *template* for creating Pods, and specify how many replicas do you want. Let's create one using an example ASP.NET Core container:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aspnet-sample-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: aspnet-sample
  template:
    metadata:
      labels:
        app: aspnet-sample
    spec:
      containers:
      - image: mcr.microsoft.com/dotnet/core/samples:aspnetapp
        name: app
```

This might seem complicated, but it's a template for creating Pods like I mentioned above. The Deployment `spec` contains:

- The desired number of `replicas`.
- A `template` which defines the Pod to be created. This is the same than the `spec` field of a Pod.
- A `selector` which gives Kubernetes a way of identifying the Pods created using the template. In this case, the template creates Pods using a specific label and the selector finds Pods that match that label.

After applying the manifest, run the following two commands. Note how the name of the deployment matches the one in the manifest, while the name of the Pod is pseudo-random:

```
$ kubectl get deployment
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
aspnet-sample-deployment    1/1     1            1           99s
$ kubectl get pod
NAME                                        READY   STATUS    RESTARTS   AGE
aspnet-sample-deployment-868f89659c-cw2np   1/1     Running   0          102s
```

This makes sense since the number of Pods is completely driven by the replicas field of the Deployment, which can be increased/decreased as needed.

Let's verify that Kubernetes will indeed try to keep the number replicas we defined. Delete the current pod with the following command (use the pods name generated in your system from the output of the command above):

```
kubectl delete pod aspnet-sample-deployment-868f89659c-cw2np
```

Then notice how Kubernetes has immediately created a new Pod to replace the one we just removed:

```
$ kubectl get pod
NAME                                        READY   STATUS    RESTARTS   AGE
aspnet-sample-deployment-868f89659c-7qwkz   1/1     Running   0          45s
```

Change the number of replicas to two and apply the manifest again. Notice how if you get the pods, this time there are two created for the Deployment:

```
$ kubectl get pod
NAME                                        READY   STATUS    RESTARTS   AGE
aspnet-sample-deployment-868f89659c-7bfxz   1/1     Running   0          1s
aspnet-sample-deployment-868f89659c-7qwkz   1/1     Running   0          2m8s
```

Now set the replicas to 0 and apply the manifest. This time there will be no pods! (might take a couple of seconds to cleanup)

```
$ kubectl get pod

No resources found in default namespace.
```

As you can see, **Kubernetes is trying to make sure there is always as many Pods as the desired number of replicas**.

While Deployment is one of the most common way of scaling containers, it is not the only way to manage multiple replicas. ***StatefulSets*** *are designed for running stateful applications like databases.* ***DaemonSets*** *are designed to simplify the use case of running one replica in each node.* ***CronJobs*** *are designed to run a container on a given schedule.*

These cover more advanced use cases and although outside of the scope of the article, you should take a look at them before planning any serious Kubernetes usage. Also make sure to check the section ***Beyond the Basics*** *at the end of the article and explore concepts like resource limits, probes or secrets.*

Exposing applications to the network: Services and Ingresses

We have seen how to create Pods and Deployments that ultimately host your containers. Apart from running the containers, it's likely you want to be able to talk to them. This might be either from outside the cluster or from other containers inside the cluster.

In the previous section, we deployed a sample ASP.NET Core application. Let's begin to see if we can send an HTTP request to it. (Make sure to change the number of replicas back to 1).
Using the following command, we can retrieve the internal IP of the Pod:

```
$ kubectl describe pod aspnet-sample-deployment
Name:         aspnet-sample-deployment-868f89659c-k7bvr
Namespace:    default
... omitted ...
IP:           172.17.0.6
... omitted ...
```

In this case, the Pod has an internal IP of `172.17.0.6`. Other Pods can send traffic by using this IP. Let's try this by adding a `busybox` container with *curl* installed, so we can send an HTTP request. The following command adds the `busybox` container and opens a shell inside the container:

```
$ kubectl run -it --rm --restart=Never busybox --image=yauritux/busybox-curl sh
```

If you dont see a command prompt, try pressing enter.

```
/ #
```

The command asks Kubernetes to run the command "`sh`", enabling the interactive mode `-it` (so you can attach your terminal) in a new container named `busybox` using a specific image `yauritux/busybox-curl` that comes with curl preinstalled. This gives you a terminal running inside the cluster, which will have access to internal addresses like that of the Pod. This container is removed as soon as you exit the terminal due to the `--rm --restart=Never parameters`.

Once inside, you can now use curl as in `curl http://172.17.0.6` and you will see the HTML document returned by the sample ASP.NET Core application.

```
/home # curl http://172.17.0.6
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Home page - aspnetapp</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="/css/site.css" />
</head>
<body>
... omitted ...
```

Using Services to allow traffic between Pods

It is great that we can simply send requests using the IP of the Pod. However, depending on the Pod, IP is not a great idea:

- What would happen if you were to use two replicas instead of one? Each replica would get its own IP, which one would you use in your client application?

- What happens if a replica is terminated and recreated by Kubernetes? Its IP will likely change, how would you keep your client application in sync?

To solve this problem, Kubernetes provides the Service object. This type of object provides a new abstraction that lets you:

- Provide a host name that you can use instead of the Pod IPs

- Load balance the requests across multiple Pods

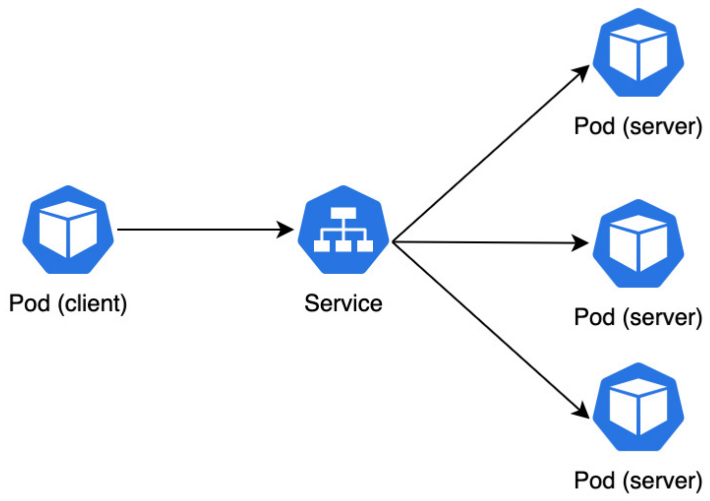


Figure 5, simplified view of a default Kubernetes service

As you might suspect, a service is defined via its own manifest. Create a service for the deployment created before by applying the following YAML manifest:

```
apiVersion: v1
kind: Service
metadata:
  name: aspnet-sample-service
spec:
  selector:
    app: aspnet-sample
  ports:
  - port: 80
    targetPort: 80
```

The service manifest contains:

- a `selector` that matches the Pods to which traffic will be sent
- a mapping of `ports`, declaring which container port (`targetPort`) map as the service port (`port`). Note the test ASP.NET Core application listens on port 80, see https://hub.docker.com/_/microsoft-dotnet-core-samples/

After you have created the service, you should see it when running the command `kubectl get service`.

Let's now verify it is indeed allowing traffic to the `aspnet-sample` deployment. Run another `busybox` container with curl. Note how this time you can test the service with `curl http://aspnet-sample-service` (which matches the service name)

```
$ kubectl run -it --rm --restart=Never busybox --image=yauritux/busybox-curl sh
If you dont see a command prompt, try pressing enter.
/ # curl http://aspnet-sample-service
<!DOCTYPE html>
... omitted ...
```

If the service was located in a different namespace than the Pod sending the request, you can use as host name `serviceName.namespace`*. That would be* `http://aspnet-sample-service.default` *in the previous example.*

Having a way for Pods to talk to other Pods is pretty handy. But I am sure you will eventually need to expose certain applications/containers to the world outside the cluster!

That's why Kubernetes lets you define other types of services, where the default one we used in this section is technically a **ClusterIP service**). In addition to services, you can also use an Ingress to expose applications outside of the cluster.

We will see one of these services (the **NodePort**) and the **Ingress** in the next sections.

Using NodePort services to allow external traffic

The simplest way to expose Pods to traffic coming from outside the cluster is by using a Service of type NodePort. This is a special type of Service object that maps a random port (by default in the range 30000-32767) in every one of the cluster nodes (Hence the name *NodePort*). That means you can then use the IP of any of the nodes and the assigned port in order to access your Pods.

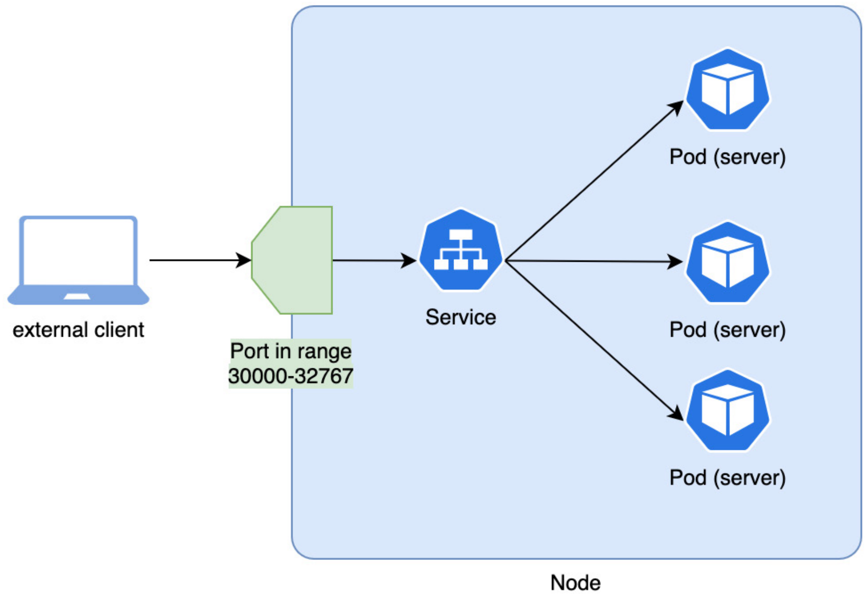


Figure 6, NodePort service in a single node cluster like minikube

It is a very common use case to define these in an ad-hoc way, particularly during development. For this reason, let's see how to create it using a kubectl shortcut rather than the YAML manifest. Given the deployment `aspnet-sample-deployment` we created earlier, you can create a NodePort service using the command:

```
kubectl expose deployment aspnet-sample-deployment --port=80 --type=NodePort
```

Once created, we need to find out to which node port was the service assigned:

```
$ kubectl get service aspnet-sample-deployment
NAME                        TYPE       CLUSTER-IP       EXTERNAL-IP    PORT(S)
AGE
aspnet-sample-deployment    NodePort   10.110.202.168   <none>         80:30738/TCP
91s
```

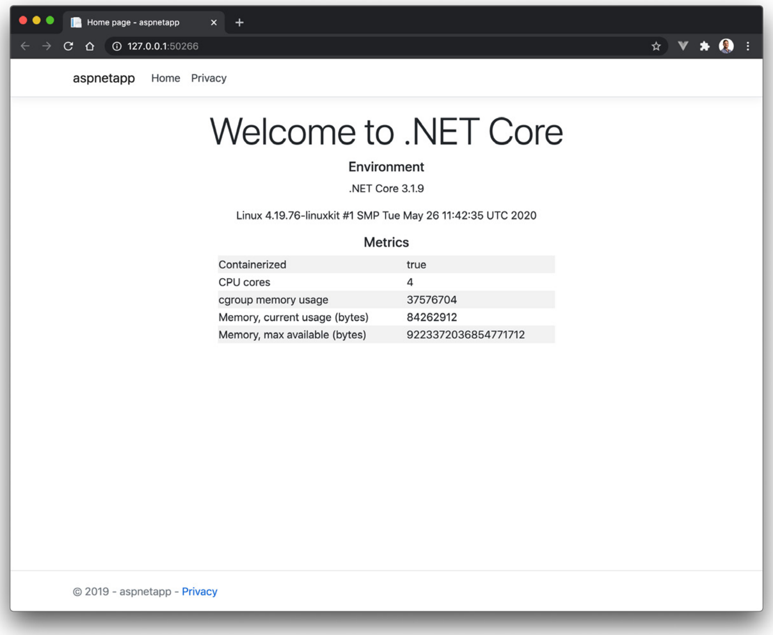
You can see it was assigned port 30738 (this might vary in your machine).

Now in order to open the service in the browser, you would need to find the IP of any of the cluster nodes, for example using `kubectl describe node`. Then you would navigate to port 30738 in any of those node IPs.

However, when using minikube, we need to ask minikube to create a tunnel between the VM of our local cluster and our local machine. This is as simple as running the following command:

```
minikube service aspnet-sample-deployment
```

And as you can see, the sample ASP.NET Core application is up and running as expected!



Home page - aspnetapp
127.0.0.1:50266
aspnetapp Home Privacy

Welcome to .NET Core

Environment

.NET Core 3.1.9

Linux 4.19.76-linuxkit #1 SMP Tue May 26 11:42:35 UTC 2020

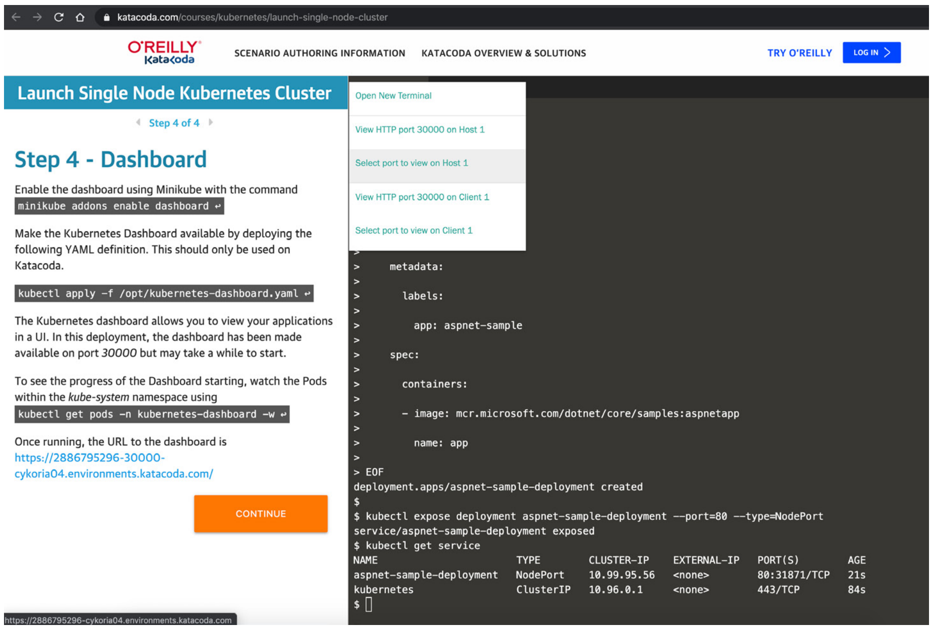
Metrics

Containerized	true
CPU cores	4
cgroup memory usage	37576704
Memory, current usage (bytes)	84262912
Memory, max available (bytes)	9223372036854771712

© 2019 - aspnetapp - Privacy

Figure 7, the ASP.NET Core application exposed as a NodePort service

If you are running in Katacoda, you won't be able to open the service in the browser using the minikube service command. However, once you get the port assigned to the NodePort service, you can open that port by clicking on the + icon at the top of the tabs, then click "Select port to view on Host 1"



katacoda.com/courses/kubernetes/launch-single-node-cluster

O'REILLY Katacoda
SCENARIO AUTHORING INFORMATION
KATACODA OVERVIEW & SOLUTIONS
TRY O'REILLY
LOG IN
Open New Terminal
View HTTP port 30000 on Host 1
Select port to view on Host 1
View HTTP port 30000 on Client 1
Select port to view on Client 1
CONTINUE

Launch Single Node Kubernetes Cluster

Step 4 of 4

Step 4 - Dashboard

Enable the dashboard using Minikube with the command `minikube addons enable dashboard`

Make the Kubernetes Dashboard available by deploying the following YAML definition. This should only be used on Katacoda.

`kubectl apply -f /opt/kubernetes-dashboard.yaml`

The Kubernetes dashboard allows you to view your applications in a UI. In this deployment, the dashboard has been made available on port *30000* but may take a while to start.

To see the progress of the Dashboard starting, watch the Pods within the *kube-system* namespace using `kubectl get pods -n kubernetes-dashboard -w`

Once running, the URL to the dashboard is https://2886795296-30000-cykoria04.environments.katacoda.com/

https://2886795296-cykoria04.environments.katacoda.com

```
>     metadata:
>
>       labels:
>
>         app: aspnet-sample
>
>     spec:
>
>       containers:
>
>       - image: mcr.microsoft.com/dotnet/core/samples:aspnetapp
>
>         name: app
>
> EOF
deployment.apps/aspnet-sample-deployment created
$
$ kubectl expose deployment aspnet-sample-deployment --port=80 --type=NodePort
service/aspnet-sample-deployment exposed
$ kubectl get service
NAME                       TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
aspnet-sample-deployment   NodePort    10.99.95.56   <none>        80:31871/TCP   21s
kubernetes                 ClusterIP   10.96.0.1     <none>        443/TCP        84s
$
```

Figure 8, opening the NodePort service when using Katacoda

Using an Ingress to allow external traffic

NodePort services are great for development and debugging, but not something you really want to depend on for your deployed applications. Every time the service is created, a random port is assigned, which could quickly become a nightmare to keep in sync with your configuration.

That is why Kubernetes provides another abstraction design for exposing primarily HTTP/S services outside the cluster, the Ingress object. The Ingress provides a map between a specific host name and a regular Kubernetes service. For example:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webtool
spec:
  rules:
  - host: aspnet-sample-deployment.io
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: aspnet-sample-service
            port:
              number: 80
```

Note for versions prior to Kubernetes 1.19 (you could check the server version returned by `kubectl version`), the schema of the Ingress object was different. You would create the same Ingress as:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webtool
spec:
  rules:
  - host: aspnet-sample-deployment.io
    http:
      paths:
      - backend:
          serviceName: aspnet-sample-service
          servicePort: 80
```

We are essentially mapping the host `aspnet-sample-deployment.io` to the very same regular service we created earlier in the article, when we first introduced the `Service` object.

How the Ingress works is via an *Ingress controller* deployed on every node of the cluster. This controller listens on port 80/443 and redirects requests to internal services based on the mapping from all the `Ingress` objects defined.

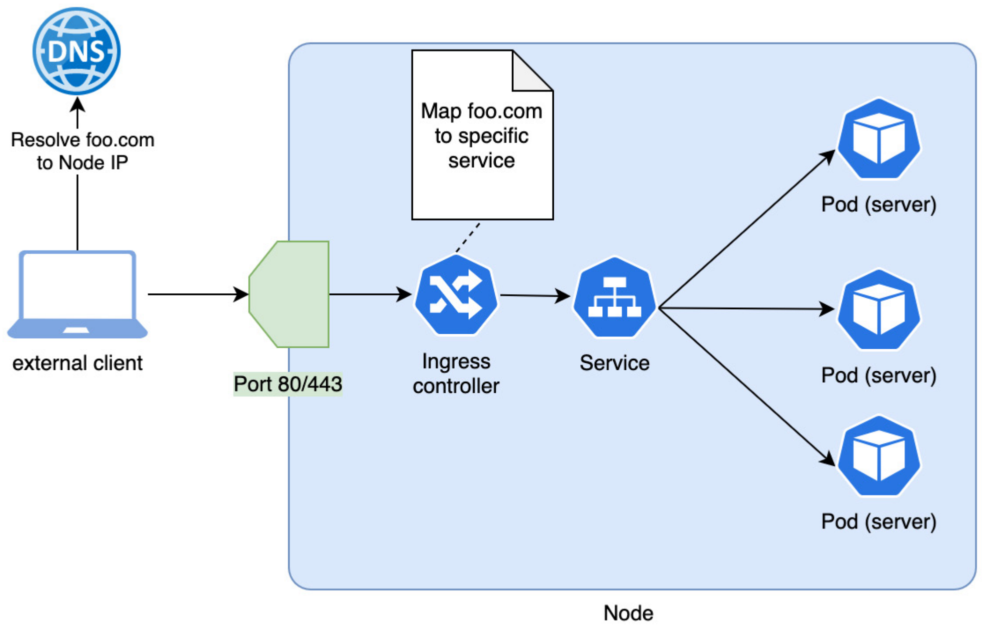


Figure 9, service exposed through an Ingress in a single node cluster

Now all you need is to create a DNS entry that maps the host name to the IP of the node. In clusters with multiple nodes, this is typically combined with some form of load balancer in front of all the cluster nodes, so you create DNS entries that point to the load balancer.

Let's try this locally. Begin by enabling the ingress addon in minikube as in:

```
minikube addons enable ingress
```

Note in mac you might need to recreate your minikube environment using `minikube start --vm=true`

Then apply Ingress manifest defined above. Once the ingress is created, we are ready to send a request to the cluster node in port 80! All we need is to create a DNS entry, for which we will simply update our host file.

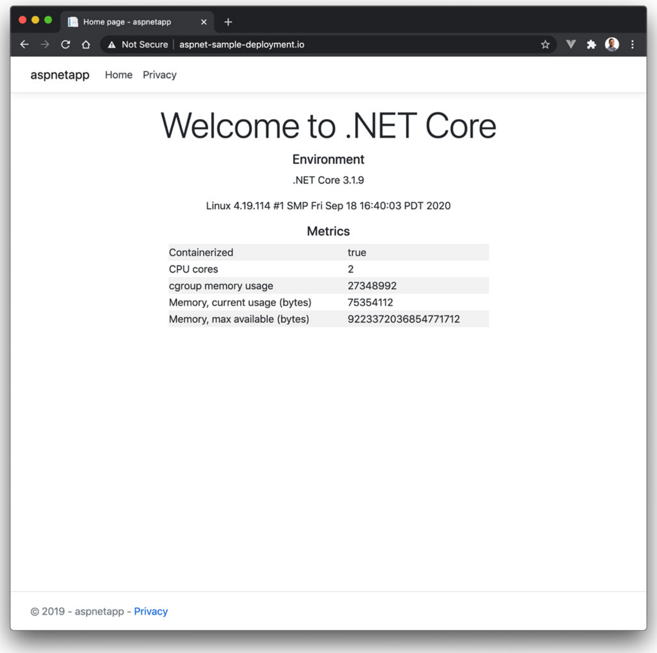
Get the IP of the machine hosting your local minikube environment:

```
$ minikube ip
192.168.64.13
```

Then update your hosts file to manually map the host name `aspnet-sample-deployment.io` to the minikube IP returned in the previous command (The hosts file is located at `/etc/hosts` in Mac/Linux and `C:\Windows\System32\Drivers\etc\hosts` in Windows). Add the following line to the hosts file:

```
192.168.64.13 aspnet-sample-deployment.io
```

After editing the file, open http://aspnet-sample-deployment.io in your browser and you will reach the sample ASP.NET Core application through your Ingress.



Welcome to .NET Core

Environment

.NET Core 3.1.9

Linux 4.19.114 #1 SMP Fri Sep 18 16:40:03 PDT 2020

Metrics

Containerized	true
CPU cores	2
cgroup memory usage	27348992
Memory, current usage (bytes)	75354112
Memory, max available (bytes)	9223372036854771712

© 2019 - aspnetapp - Privacy

Figure 10, the ASP.NET Core application exposed with an Ingress

Kubernetes - Beyond the basics

The objects we have seen so far are the core of Kubernetes from a developer point of view. Once you feel comfortable with these objects, the next steps would feel much more natural.

I would like to use this section as a brief summary with directions for more advanced contents that you might want (or need) to explore next.

More on workloads

You can inject configuration settings as environment variables into your Pod's containers. See the official docs. Additionally, you can read the settings from `ConfigMap` and `Secret` objects, which you can directly inject as environment variables or even mount as files inside the container.

We briefly mentioned at the beginning of the article that Pods can contain more than one container. You can explore patterns like init containers and sidecars to understand how and when you can take advantage of this.

You can also make both your cluster and Pods more robust by taking advantage of:

- Container probes, which provide information to the control plane on whether the container is still alive or whether is ready to receive traffic.

- Resource limits, which declare how much memory, cpu and other finite resources does your container need. This lets the control plane make better scheduling decisions and balance the different containers across the cluster nodes.

Persisting data

Any data stored in a container is ephemeral. As soon as the container is terminated, that data will be gone. Therefore, if you need to permanently keep some data, you need to persist it somewhere outside the container. This is why Kubernetes provides the PersistentVolume and PersistentVolumeClaim abstractions.

While the volumes can be backed by folders in the cluster nodes (using emptyDir volume type), these are typically backed by cloud storage such as **AWS EBS** or **Azure Disks**.

Also remember that we mentioned StatefulSets as the **recommended workload** (rather than Deployments) for stateful applications such as databases.

CI/CD

In the article, we have only used containers that were publicly available in Docker Hub. However, it is likely that you will be building and deploying your own application containers, which you might not want to upload to a public docker registry like Docker Hub. In those situations:

- you will typically configure a private container registry. You could use a cloud service like AWS ECR or Azure ACR or hosting your instance of a service like Nexus or Artifactory.

- You then configure your cluster with the necessary credentials so it can pull the images from your private registry.

Another important aspect will be describing your application as a set of YAML files with the various objects. I would suggest getting familiar with Helm and considering helm charts for that purpose.

Finally, this might be a good time to think about CI/CD.

- Keep using your preferred tool for CI, where you build and push images to your container registry. Then use a gitops tool like argocd for deploying apps to your cluster.

- Clouds like Azure offer CI/CD solutions, like Azure DevOps, which integrate out of the box with clusters hosted in the same cloud. Note Azure DevOps has been recently covered in a previous edition of the magazine.

Cloud providers

Many different public clouds provide Kubernetes services. You can get Kubernetes as a service in Azure, AWS, Google cloud, Digital Ocean and more.

In addition, Kubernetes has drivers which implement features such as persistent volumes or load balancers using specific cloud services.

Interesting tools you might want to consider

External-dns and cert-manager are great ways to automatically generate both DNS entries and SSL certificates directly from your application manifest.

Velero is an awesome tool for backing up and restoring your cluster, including data in persistent volumes.

The Prometheus and Grafana operator provide the basis for monitoring your cluster. With the fluentd operator you can collect all logs are send them to a persistent destination.

Network policies, RBAC and resource quotas are the first stops when sharing a cluster between multiple apps and/or teams.

If you want to secure your cluster, trivy and anchore can scan your containers for vulnerabilities, falco can provide runtime security and kube-bench runs an audit of the cluster configuration.

..and many, many more than I can remember or list here.

Conclusion

Kubernetes can be quite intimidating to get started with. There is plenty of new concepts and tools to get used to, which can make running a single container for the first time a daunting task.

However, it is possible to focus on the most basic functionality and elements that let you, a developer, deploy an application an interact with it.

I have spent the last year helping my company transition to Kubernetes, with dozens of developers having to ramp up in Kubernetes in order to achieve their goals. In my experience, **getting these fundamentals right, is key**. Without them, I have seen individuals and teams getting blocked or going down the wrong path, both ending in frustration and wasted time.

Regardless of whether you are simply curious about Kubernetes or embracing it at work, I hope this article helped you understanding these basic concepts and sparked your interest. In the end, the contents here are but a tiny fraction of everything you could learn about Kubernetes!

Daniel Jimenez Garcia
Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

Techinical Review
Subodh Sohoni

Editorial Review
Suprotim Agarwal

Ravi Kiran

FORMS IN REACT.JS USING React Functional Components and React Hooks

Good user interfaces are amongst the most challenging aspects of modern application development.

One of the most important aspects of getting the UI right is the handling of user input. As React developers, we should know how to handle forms in a React application.

This tutorial will get you started with the forms in React.js by building a simple form and validating the values.

This article will cover how to work with input controls in React. We will then build a simple form in React and show how to perform validations on the form fields.

The examples in the article are built using **React functional components and React hooks**.

Working with Input Controls

In a React component, *state* is the data context for the views. It is natural to assign the values received from form fields into the state. This can be done by listening to the change events on the input controls and assigning the values received to the state.

The following snippet shows how this is done in a React component:

```
import React, { useState } from 'react';

export function SampleInput() {
  const      [name, setName] = useState('Ravi');

  function handleChange(e)  {
    e.preventDefault();

    setName(e.target.value);
  };

  return (
            <div>
        Name:
        <input
          type='text'
          value={name}
          onChange={handleChange}
        />
      </div>
      <div>Name is: {name}</div>

  );
}
```

Listing 1: A React component with input text field

As we can see, the value of the input text field is set using a value in state, and the state is updated on the `onChange` event handler using the value present in the text field. This way, two-way binding is established and the value shown in the `div` element below the text field gets updated on whenever a key is pressed in the textbox. The following screenshot shows how this works:

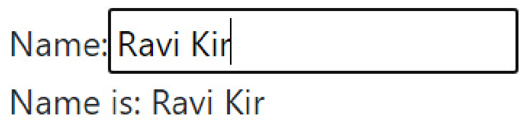

Name: Ravi Kir
Name is: Ravi Kir

Figure 1 – textbox with 2-way binding

The input element controlled this way is called a "controlled component". You can read more about controlled component in the React documentation.

Note: *Notice the way the* `handleChange` *method is defined in the Listing 1. It is created as an arrow function to retain the current class context. Otherwise, the method will get the context of the input field and the instance variables won't be accessible.*

The input fields can also be managed using **refs**. A ref can be created using `useRef` hook. The `ref` attribute assigned on an element receives DOM object of the element. The properties and events of the element are accessible using the DOM object.

The following snippet shows the modified version of the *SampleComponent* using refs:

```
import React, { useState, useRef, useEffect } from 'react';

export function SampleInput() {
  const      [name, setName] = useState('Ravi');
  const      input = useRef(     );

  useEffect(() => {
    input.current.onkeyup = handleChange;
    input.current.value = name;
  });

  function handleChange(e) {
    e.preventDefault();

    setName(e.target.value);
  }

  return (
    <React.Fragment     >
      <div>
        Name:
        <input type='text' ref={input} />
      </div>
      <div>Name is: {name}</div>
    </React.Fragment     >
  );
}
```

Listing 2: Using ref with input element

Here the value and the event handler are assigned to the input element in the `useEffect` hook. This is because the component has to be rendered and the DOM objects have to be ready before we use it to access the value or listen to the events.

The input element handled this way is called "uncontrolled component". You can read more about uncontrolled components in the React documentation.

This component produces the same result as the one shown in Listing 1.

We will be using the controlled component approach in the next section of the article as it integrates better with the state. Also, it is the recommended way to build forms in a React application.

Building a Sign-up form

Let's build a sign-up form and validate the fields to learn more. Let's create a component with a few fields to accept the inputs from a user to sign up. Listing 3 shows the initial state of the component with the fields and the state:

```
import React, { useState } from 'react';
import { SampleInput } from './SampleInput';
import logo from './logo.svg';
import './App.css';
import {
  minMaxLength,
  validEmail,
  passwordStrength,
  userExists,
} from './validations';

function App() {
  const      [user, setUser] = useState({});
  const      [formErrors, setFormErrors] = useState({});



    return (
      <div className='App container col-6'>
        <h3>New User Registration Form</h3>
        <form noValidate>
          <div className='row'>
            <div className='col-md-6'>
              <label htmlFor='firstName'>First Name</label>
              <input
                className='form-control'
                placeholder='First Name'
                type='text'
                name='firstName'
                noValidate
              />
            </div>
            <div className='col-md-6'>
              <label htmlFor='lastName'>Last Name</label>
              <input
                className='form-control'
                placeholder='Last Name'
                type='text'
                name='lastName'
                noValidate
              />
            </div>
          </div>

          <div className='mb-3'>
            <label htmlFor='email'>Email</label>
            <input
              className='form-control'
              placeholder='Email'
              type='email'
              name='email'
              noValidate
            />
          </div>
          <div className='mb-3'>
            <label htmlFor='password'>Password</label>
            <input
              className='form-control'
              placeholder='Password'
              type='password'
              name='password'
              noValidate
            />
          </div>
          <div className='mb-3'>
<label htmlFor='confirmpassword'>Confirm Password</label>
            <input
              className='form-control'
              placeholder='Password'
              type='password'
              name='confirmpassword'
              noValidate
            />
          </div>
          <div className='mb-3'>
            <button type='submit'>Create Account</button>
          </div>
        </form>
      </div>
    );
}
```

Listing 3: Initial component with signup form

The above form consists of fields to enter the first name, last name, e-mail ID, password and to confirm the password. Notice the `noValidate` attribute used on the form and on the input elements; it is done to prevent any default HTML5 validations by the browser. Figure 3 shows how it looks when rendered on the browser:

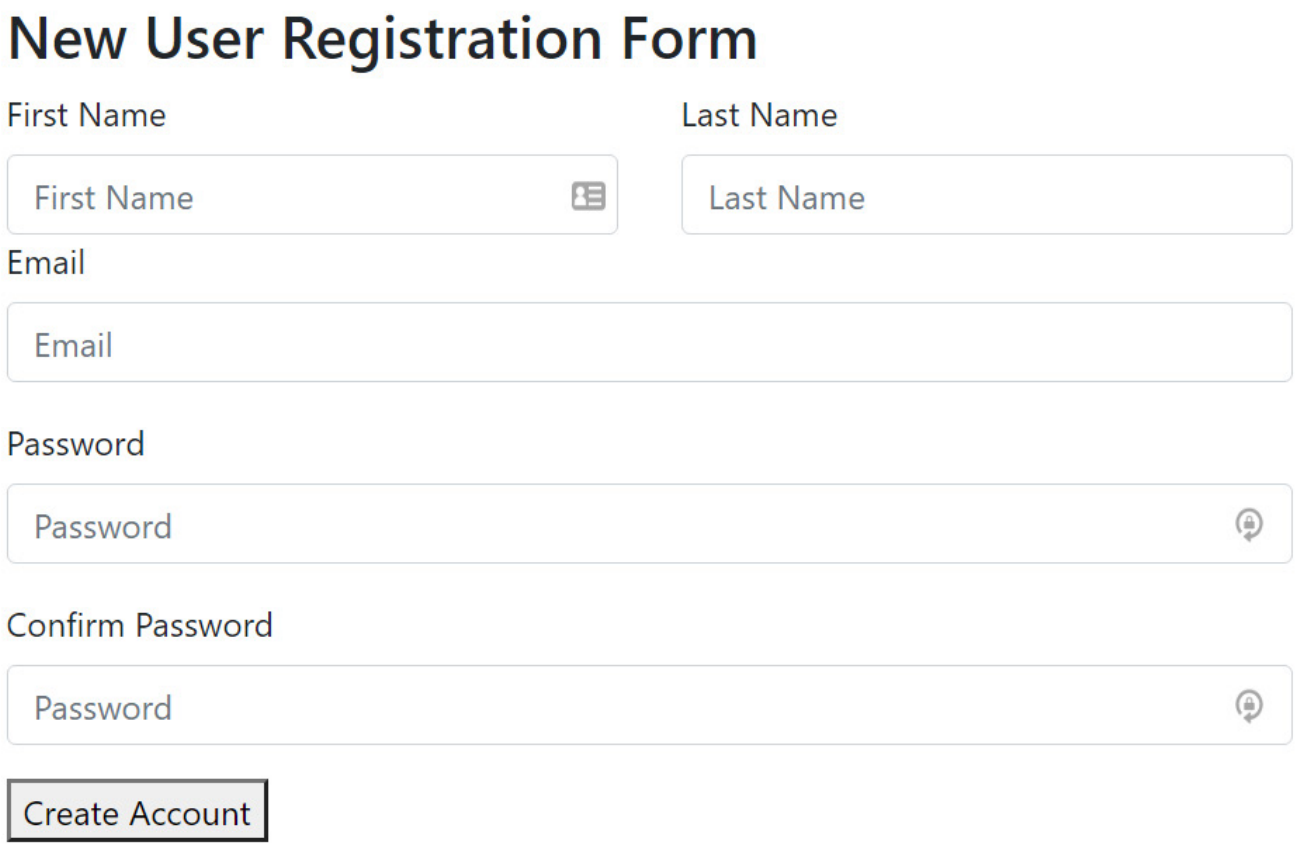


Figure 3: A sign-up form

Let’s add event handlers to all these input controls. It is possible to create a single function and assign it to all the controls. The function has to assign values to the respective state fields based on the source input element.

Following is the function to be used for the event handler:

```
Function handleChange      (e)      {
  const { name, value } = e.target;
  let formErrors = this.state.formErrors;

  switch (name) {
    case 'firstName':
      setUser({ ...user, firstName: value });

      break;
    case 'lastName':

setUser({ ...user, lastName: value });
      break;
    case 'email':
      setUser({ ...user, email: value });

      break;
    case 'password':
      setUser({ ...user, password: value });

      break;
    case 'confirmpassword':
      setUser({ ...user, confirmpassword: value });

      break;
    default:
      break;
  }
}
```

Listing 4: Event handler for input fields

The event handler can be assigned on the input elements to handle the `onBlur` event. Alternatively, the `onKeyUp` event may also be used but it will keep assigning the state on every key stroke. To avoid that, we are going to use `onBlur`. Listing 5 shows how to assign it to one of the fields:

```
<input
  className= 'form-control'
  placeholder='First Name'
  type='text'
  name='firstName'
  noValidate
  onBlur={     handleChange}
/>
```

Listing 5: Input with event handler

Now the value entered in the text box would get updated in the state field when a user moves the focus away from the input element accepting the first name. The same change can be applied on the other input elements.

To see if the values are assigned to the state, the state object can be printed on a console when the form is submitted.

With this, the form is ready for accepting inputs and passing them on to the application.

Let's add now validations to it!

Form Validations

The values received in the input fields can be validated on the change event handler. We will be adding the following validations to the sign-up form:

- First name and last name have to contain at least 3 characters
- Password has to contain at least 6 characters
- E-mail ID has to be valid
- Confirm password has to match password
- Password should be strong
- E-mail ID has to be unique

Note: *The validations added here would be running on the client side. It is important to perform these validations on the server as well, as client-side validations can get compromised by hackers or JavaScript could be turned off in the browsers.*

Let's create generic helper functions to perform these validations. Listing 5 shows the functions to check minimum length, format of the e-mail ID and to check if the e-mail ID is unique:

```
export function minMaxLength(text, minLength, maxLength) {
    let result = !text || text.length < minLength;
    if(maxLength)
        result = result || text.length < minLength;
    return result;
}

export function validEmail(text) {
    const regex = RegExp(
        /^[a-zA-Z0-9.!#$%&’*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$/
      );

    return !regex.test(text);
}

let registeredUsers = [
    'ravi@kiran.com',
    'mail@myblog.in',
    'contact@lucky.com'
];


export function userExists(email) {
    return new Promise(resolve => {
        setTimeout(() => {
            if(registeredUsers.findIndex(u => u === email) !== -1) {
                resolve(true);
            }
            else {
                resolve(false);
            }
        });
    });
}
```

Listing 5: Validation functions

The function `userExists` in Listing 5 returns a promise and it checks for the existence of the e-mail ID in a hard-coded array. In real applications, this has to be replaced with an API call.

Writing logic to check strength of a password is a big task. Let's take the help of a library to do that. Install the package *zxcvbn* using the following command:

```
> npm install zxcvbn
```

This library exposes a function that accepts the password string and returns a result. We can determine whether the password is strong or not, using the score it returns. Listing 6 imports the library and performs the validation:

```
import * as zxcvbn from 'zxcvbn';

export function passwordStrength(text) {
    let result = zxcvbn(text);
    return result.score < 3;
}
```

Listing 6: Password strength validation

Now let's use these validations in the `onBlur` event handler. Listing 7 shows the validation of the field for the first name:

```
case 'firstName':
  if (minMaxLength(value, 3)) {
    currentFormErrors[
      name
    ] = `First Name should have minimum 3 characters`;
  } else {
    delete currentFormErrors[name];
  }
  break;
```

Listing 7: Validation for first name

The Last name can be validated in a similar way. As Listing 7 shows, the error message to be shown is set to the `formErrors` object. This object will be used to show the validation messages and to disable the submit button. When the field is valid, it deletes the error of the field from the object.

The E-mail ID needs to be checked for format and for uniqueness. As the `userExists` validation function returns a promise to simulate the behavior of an API call, the validity has to be set in the success callback of the promise. Listing 8 shows this validation:

```
case 'email':
  if (!value || validEmail(value)) {
    currentFormErrors[name] = `Email address is invalid`;
  } else {
    userExists(value).then((result) => {
      if (result) {
        currentFormErrors[name] =
          'The email is already registered. Please use a different email.';
      } else {
        delete currentFormErrors[name];
      }
    });
  }
```

Listing 8: E-mail validation

The password field has to be checked for length and for strength. A value entered in the *Confirm Password* field has to be matched with password field and if it doesn't match, an error has to be assigned.

The field *Confirm Password* needs to be validated when the password field is changed as well, as the user may decide to change the password entered, before submitting. The following snippet shows the validation of the two password fields:

```
case 'password':
  if (minMaxLength(value, 6)) {
    currentFormErrors[name] = 'Password should have minimum 6 characters';
  } else if (passwordStrength(value)) {
    currentFormErrors[name] =
      'Password is not strong enough. Include an upper case letter, a number or a 
special character to make it strong';
  } else {
    delete currentFormErrors[name];
    setUser({
      ...user,
      password: value,
    });
    if (user.confirmpassword) {
      validateConfirmPassword(
        value,
        user.confirmpassword,
        currentFormErrors
      );
    }
  }
  break;
case 'confirmpassword':
  let valid = validateConfirmPassword(
    user.password,
    value,
    currentFormErrors
  );
  if (valid) {
    setUser({ ...user, confirmpassword: value });
  }
  break;
```

Listing 9: Validation of Password and confirm password fields

As the confirm password validation has to be performed twice, it is better to put it in a separate function. The following snippet shows the function `validateConfirmPassword`:

```
function validateConfirmPassword(
  password,
  confirmpassword,
  formErrors
) {
  formErrors = formErrors || {};
  if (password !== confirmpassword) {
    formErrors.confirmpassword =
      'Confirmed password is not matching with password';
    return false;
  } else {
    delete formErrors.confirmpassword;
    return true;
  }
}
```

Listing 10: validateConfirmPassword function

The last change to be made is to set validations in the state. It is done in the following snippet:

```
setFormErrors(currentFormErrors);
```

Now that the validations are in place, the user needs to be notified about them. Let's apply some styles on the fields to show that the field has an invalid value. The following snippet modifies the style of the field for first name:

```
<input
  className={
    formErrors && formErrors.firstName
      ? 'form-control error'
      : 'form-control'
  }
  placeholder='First Name'
  type='text'
  name='firstName'
  noValidate
  onBlur={handleChange}
/>
```

Listing 11: Error style applied to first name

The attribute `className` is modified to assign the `error` class when the field has a validation error. The same change can be applied to other fields also. The CSS class `error` is set to the following style to show a red border on the input elements:

```
.app      input.error {
  border: 1px solid red;
}
```

Listing 12: Error style

There are different ways to show error messages. They can be either shown below the input elements, listed in an unordered list, displayed as tooltips on the input elements or any other way your UX designer suggests.

In the demo, we will show it as an unordered list. For this, we need to iterate over the fields of `formErrors` object and print the messages in a list. Listing 12 shows how to achieve this:

```
<ul>
  {Object.entries(formErrors || {}).map(([prop, value]) => {
    return (
      <li className='error-message' key={prop}>
        {value}
      </li>
    );
  })}
</ul>
```

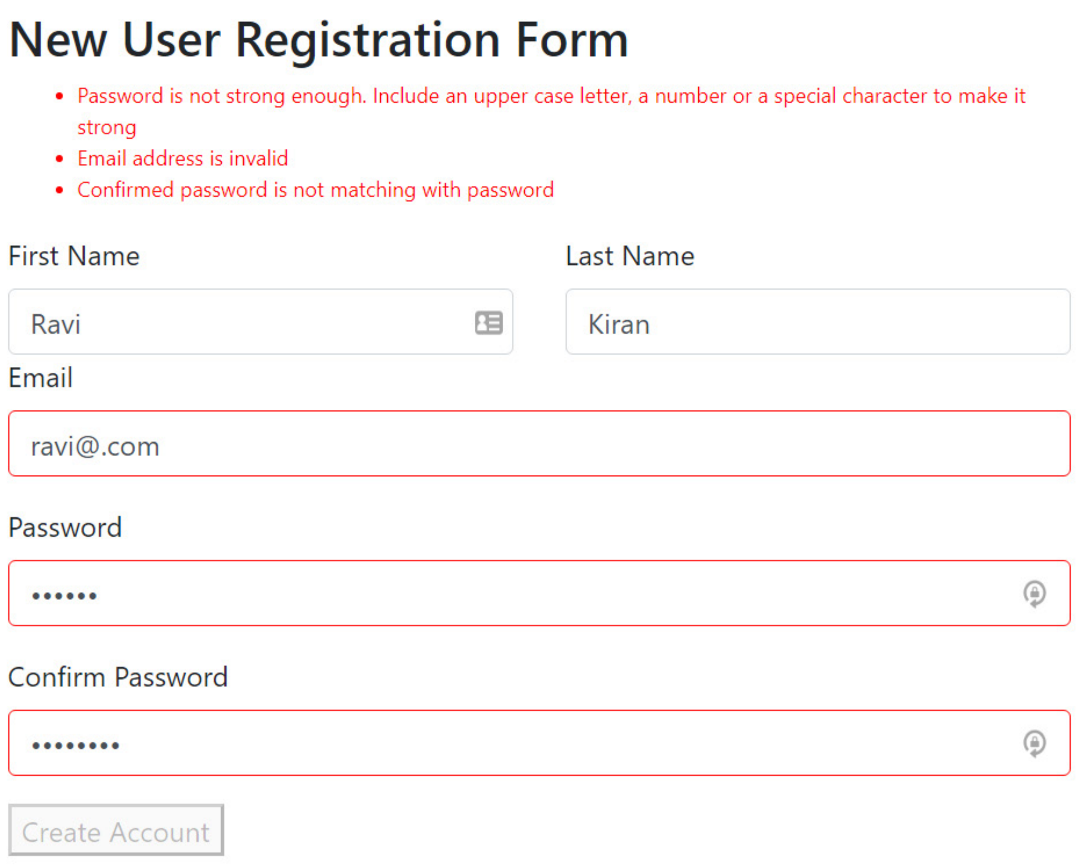
Listing 12: List of errors

The only pending todo now is to disable the submit button when the form has errors. This can be done by checking if formErrors has any entries. See Listing 13.

```
<button
  type='submit'
  disabled={Object.entries(formErrors || {}).length > 0}
>
  Create Account
</button>
```

Listing 13: Disabling the button

Now when there are invalid values, the form will start showing errors and marking the fields with a red border. A state of the form is shown in Figure 4:



New User Registration Form

- Password is not strong enough. Include an upper case letter, a number or a special character to make it strong
- Email address is invalid
- Confirmed password is not matching with password

First Name	Last Name
Ravi	Kiran
Email	
ravi@.com	
Password	
••••••	
Confirm Password	
••••••••	

Figure 4: Form with validation errors

Conclusion

Handling forms is an important part of any business application and it can turn tricky at times. This tutorial shows how to handle simple forms in React and perform validations.

Hope this knowledge will get you started with Forms in React. Reach out to me at my twitter handle @ sravi_kiran with any comments or questions.

Download the entire source code from GitHub at
http://bit.ly/dncm48-reactforms

• • • • • • •

Ravi Kiran
Author

Ravi Kiran is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (Visual Studio and Dev Tools) and DZone MVB awards for his contribution to the community.

Techinical Review
Benjamin Jakobus

Editorial Review
Suprotim Agarwal

Yacoub Massad

TIC x TAC o TOE o in F# Part I

In this article series, I will explore the F# language, a .NET based functional-first programming language, by using the example of the Tic Tac Toe game.

Introduction

Functional programming has been gaining popularity in the last few years.

C#, which is the most used language when targeting .NET, has been getting many functional language features. However, there is another language that targets .NET and is **functional-first**. This language is called F#.

F# is a multi-paradigm programming language. This means that you can, for example, write programs in an object-oriented style if you want. However, the focus in F# is on the functional style of programming.

Editorial Note: *To explore some functional features in C#, check this tutorial Functional Programming (F#) for C# Developers.*

In the Writing Pure Code in C# article, I used the Tic Tac Toe game as an example for writing pure code in C#. I will use the same example in this tutorial. However, please remember that the focus of this article is F#, not pure code.

If you are not familiar with the Tic-Tac-Toe game, please read about it here: https://en.wikipedia.org/wiki/Tic-tac-toe. Or spend a few minutes playing it online. You can find many versions online. If you want, you can play it using the example game I put on Github.

You can find the source code here: https://github.com/ymassad/TicTacToeFSharp. This should also help if you want to try everything yourself.

In this article series, although I will discuss the code in detail, the series is not structured as step-by-step tutorials. Still, I think you can manage to create the game yourself from scratch if you want. To get started with that, you need to create a new Console Application with the language set to F#.

Note that you need to make sure that you got F# installed. If not, you can add it to your Visual Studio installation using the Visual Studio Installer.

Once you create the new project, the program.fs file gets added automatically by Visual Studio. To add a new file, you can right click on the project -> Add -> New Item -> Source File.

The Board module

In F#, it is idiomatic to separate data and behavior.

Therefore, I will define types to model the Board and then define functions to act on the board. I will define these types and functions in a new file called BoardModule.fs. If you create a new file with this name in the project, the file will contain the following by default:

```
module BoardModule
```

This code declares a module called `BoardModule`. Any types and functions we define after the above line will belong to the `BoardModule` module. Modules help us group related types and functions together.

I start by defining the CellStatus type like this:

```
type CellStatus = Empty | HasX | HasO
```

This code declares what is called in F# as a **discriminated union**. This is somewhat like Enums in C#.

However, F# discriminated unions are more advanced!!

I will show you a more advanced usage in another part of the series. For now, what we can understand from the line of code above is that we are declaring a `CellStatus` type to represent the status of a cell in the board. A cell can be empty, contain an X, or contain an O. Unlike Enums in C# where any integer value can be used, only valid values can be used in F# discriminated unions.

I define another discriminated union to model the index of a cell:

```
type Index = One | Two | Three
```

While we can use a normal integer to model the row or column index of a cell, using a discriminated union allows us to make invalid states unrepresentable and therefore make the application more robust.

I then define a *record* to model a single row in the board:

```
type Row = {Cells: Map<Index, CellStatus>}
```

The `Row` record has a single element labeled `Cells`. The type of this element is `Map<Index, CellStatus>`.

A Map is an immutable dictionary. In this case, the dictionary can contain a `CellStatus` for each possible Index (for each column index). Note that a Map does not necessarily have a value for each key. In the way I modeled the `Row`, I assume that if there is no value for a specific Index, then the value is `CellStatus.Empty`. More on this later.

Of course, you can create records with more than one element, I will show you an example in another part of this series.

I am also using a record to model the `Board` like this:

```
type Board = {Rows: Map<Index, Row>}
```

The `Board` record has a single element labeled `Rows`. It has a type of `Map<Index,Row>`. So, for each possible row index, we have a `Row`. Again, in the way I modeled this, when there is no value for a specific Index, I assume that we have an empty row. Next, I define `emptyRow`:

```
let emptyRow = {Cells = Map.empty}
```

The `let` keyword binds the value `{Cells = Map.empty}` to the name `emptyRow`. You can think of `emptyRow` as a variable. However, in F#, values are immutable by default. So, we cannot change the value bound to `emptyRow` after this line.

`{Cells = Map.empty}` is called a *record expression*. It is an expression that creates a record value. In this case, we are creating a `Row` record value. The compiler infers this because we are using the `Cells` label. We don't have to specify that this is a Row record.

I use `Map.empty` here to create an empty `Map<Index, CellStatus>`.

Let us define the `getRow` function:

```
let getRow index board =
    let possibleRow = Map.tryFind index board.Rows
    Option.defaultValue emptyRow possibleRow
```

The `getRow` function takes two parameters: `index` and `board` (functions in F# actually take a single parameter. More on this later). This function returns the row at the specified index.

The `index` parameter is of type `Index`, and the `board` parameter is of type `Board`. I did not have to specify the types manually (although I can). The compiler inferred the types automatically. It was able to do so not because of the names of the parameters, but because of the usage of these parameters in the body.

The first hint to the compiler is the `board.Rows` expression. From this, the compiler infers that the `board` parameter must be of type `Board` because the Board record has the label `Rows`. It will become clear how the compiler inferred the type of the index parameter in about a minute.

Inside the `getRow` function, we define a value: `possibleRow`. You can think of this as a variable, but an immutable one.

Please note the indentation! **In F#, indentation is significant**.

The application might fail to compile if you use the wrong indentation. In the `getRow` function above, there are no curly brackets (like in C#) to tell us where this function begins and where it ends. The indentation has this job! If you remove the four spaces before "let possibleRow =..", you will get a compilation error.

The following is the value bound to the `possibleRow` identifier:

```
Map.tryFind index board.Rows
```

This is a function call. The function name is `tryFind` and it lives in the `Map` module. We pass two arguments to the function: `index` and `board.Rows`. Note that unlike C#, we don't need to use parentheses around the arguments.

This function attempts to find a value for the given index in the given Map. It's return type is `Option<T>` where `T` is the type of the value. In this case, it is `Option<Row>`. In F#, this type is also expressed as "Row option". This Option type is sometimes called Maybe in other programming languages/libraries. It represents a value that may or may not exist. You can read more about it here:
https://www.dotnetcurry.com/patterns-practices/1510/maybe-monad-csharp

In the next line, we have this (also with indentation):

```
Option.defaultValue emptyRow possibleRow
```

This is also a function call. The function name is `defaultValue` and it lives in the `Option` module. This function returns the value inside `possibleRow` if there is a value inside it, or it returns `emptyRow` if there isn't.

Note that there is no `return` keyword used here. In F#, functions don't need to use the return keyword. **F# treats the last expression in the function as the return value**.

So basically, the `getRow` function returns the row specified by the index from the Map stored inside `board.Rows`. If there is no Row for the specified index, the `getRow` function returns an empty row.

If you look at the source code in GitHub, this is not how I define the `getRow` function. Here is how it is defined there:

```
let getRow index board =
    board.Rows |> Map.tryFind index |> Option.defaultValue emptyRow
```

The `|>` operator is called the **forward pipe operator**. It allows us to call the function on the right side of the operator using the value at the left side of the operator as an argument. This means that the following:

```
x |> f
```

Is similar to:

```
f x
```

Q. Both pass `x` as an argument to the function `f`. So why the extra syntax?

The pipe operator allows us to chain function calls as successive operations. This is what is happening in the body of the `getRow` function. We are passing `board.Rows` as an argument to the `Map.tryFind` function (passing also index as the first argument). Then, we are passing the return value of `Map.tryFind` as an argument to the `Option.defaultValue` function (passing also emptyRow as the first argument).

Without the forward pipe operator, we could define `getRow` like we did before, or we could also define it like this:

```
let getRow index board =
    Option.defaultValue emptyRow (Map.tryFind index board.Rows)
```

The parentheses used here are not the argument list parentheses that we are used to in C#. Here, the parentheses control the order of evaluation. This means that the `Map.tryFind` function is called first (using `index` and `board.Rows` as arguments), and then the result of that is passed as the second argument to the `Option.defaultValue` function.

Again, why the forward pipe operator? In the version where the forward pipe operator is used, the order of evaluation is from left to right, unlike the last version here where the evaluation is from right to left. Having function call evaluation be from left to right is more intuitive because functions appear in the same order that they will be evaluated in.

Let's go back to the version that uses the forward pipe operator:

```
let getRow index board =
    board.Rows |> Map.tryFind index |> Option.defaultValue emptyRow
```

Let’s talk in detail about the following expression:

```
board.Rows |> Map.tryFind index
```

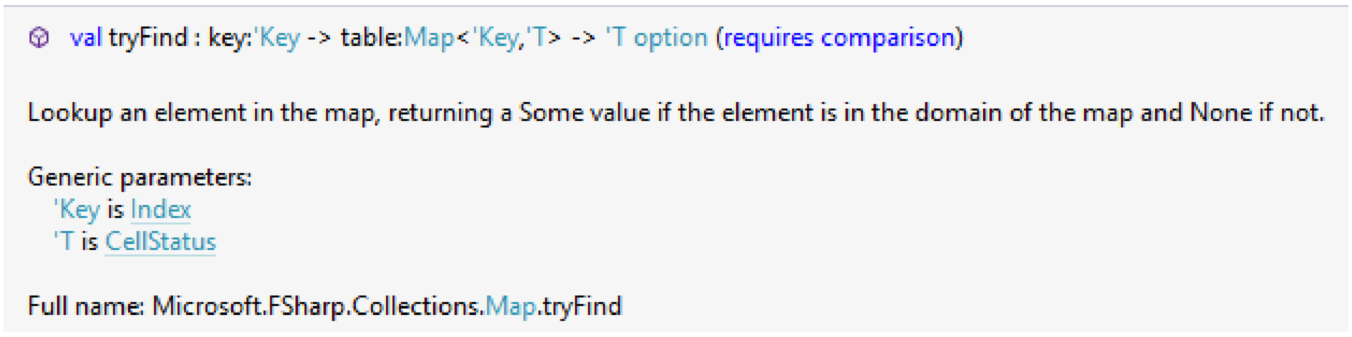
You can think of this expression as passing `board.Rows` as the second argument for the `Map.tryFind` function (and having `index` as the first argument). However, this is not exactly what is going on. Let’s dive deeper.

In this article, I hinted at two things which are:

1. Functions in F# take exactly one parameter.
2. The expression on the right side of the forward pipe operator is function.

Based on point #1, the `Map.tryFind` function should be taking a single parameter, not two. And that is really the case.

Let us look at what IntelliSense tells us about this function (when used in the context of the `getRow` function):



val tryFind : key:'Key -> table:Map<'Key,'T> -> 'T option (requires comparison)

Lookup an element in the map, returning a Some value if the element is in the domain of the map and None if not.

Generic parameters:
'Key is Index
'T is CellStatus

Full name: Microsoft.FSharp.Collections.Map.tryFind

We can think of this function as one that takes both a key and a table (the dictionary or Map) parameter, and returns a possible value. This is one useful way to think about this function.

The other way to think about it is that this function takes only the `key` parameter and returns another function. Let me write a succinct version of the signature for `tryFind` here to explain:

```
tryFind: key -> table -> ‘T option
```

You can think of the `->` symbol to mean *returns*.

So, the `tryFind` function takes a `key` parameter and returns the following:

```
table -> ‘T option
```

..which is another function that takes table (a dictionary) and returns `‘T` option.

By the way, `‘` in F# is used to indicate that this is a generic type parameter. `‘T` option means `Option<’T>`.

We say that the `tryFind` function is a *curried* function (after the mathematician Haskell Curry). This means that it is structured in a way that we call it one parameter at a time.

In C#, a curried `tryFind` function can be written like this:

```
public static class DictionaryFunctions<TKey, TValue>
{
  public static Func<TKey, Func<Dictionary<TKey, TValue>, Option<TValue>>> TryFind =
    key =>
    {
      return dictionary =>
      {
        if (dictionary.TryGetValue(key, out var value))
        {
          return Option<TValue>.Some(value);
        }
        else
        {
          return Option<TValue>.None;
        }
      };
    };
}
```

And can be called one parameter at a time like this:

```
var dictionary = new Dictionary<int, string>();

dictionary.Add(1, "asd");

var result1 = DictionaryFunctions<int, string>.TryFind(1)(dictionary);

var result2 = DictionaryFunctions<int, string>.TryFind(2)(dictionary);
```

Please take a minute to digest this.

Let us go back to the expression I want to discuss:

```
board.Rows |> Map.tryFind index
```

Earlier, I said that the expression on the right of the forward pipe operator should be a function. Now, it should be clear that this is true for the expression above. `Map.tryFind index` is a function. It is a function of type: `(Map<Index, CellStatus>) -> CellStatus` option. That is, it is a function that takes a `Map` (or dictionary) and possibly returns a `CellStatus`.

Calling a "multi-parameter" function with a single argument is called *partial application*. So, in the above expression (`Map.tryFind index`), we partially apply `Map.tryFind` using the argument `index`.

In general, when a function is designed in F#, the order of parameters is important. This is true because if you want to use the forward pipe operator, only the "last" argument of the function can be passed using the operator. We must partially apply the function by passing values for all but the last parameter. Only then we can use the forward pipe operator to pass the value for the last parameter, from the expression on the left of the operator.

This is a lot to digest, so I will stop here.

In the next part of our F# series, I will build on what we have discussed so far. I suggest you open the source code and try to play with the code to understand it.

Use IntelliSense to help you understand the type of values. You can take an expression in the middle of some code and bind it to a name by using the `let` operator and then use IntelliSense to understand the type of that expression. For example, you can bind the `Map.tryFind index` expression to a name like this:

```
let f1 = Map.tryFind index
```

...and then examine the type of `f1` using IntelliSense. Just be careful with the indentation!

Conclusion:

F# is a .NET based functional-first programming language.

In this article series, I explored the language by using the example of the Tic Tac Toe game. In this part, I demonstrated simple discriminated unions, simple records, functions, and the forward pipe operator. I also talked about the concepts of currying and partial application.

• • • • • • •

Yacoub Massad

Author

Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com. *Recently he started a YouTube channel about Roslyn, the .NET compiler.*

Techinical Review

Dobromir Nikolov

Editorial Review

Suprotim Agarwal

Benjamin Jakobus

ARTIFICIAL INTELLIGENCE: What, Why and How

"It is not even clear that intelligence has any long-term survival value. Bacteria, and other single cell organisms, will live on, if all other life on Earth is wiped out by our actions."

- Stephen Hawking

In the previous tutorial of our Machine Learning for Everybody series, we explained the fundamental concept behind computer science, and hopefully transmitted an appreciation of the complexity of the field itself.

Through the sample problem of finding the shortest route between two points on a map, we tried to demonstrate the process of abstracting problems. As such, it should now be clear to the reader, that the first step in the construction of intelligent systems involves the **translation of real world problems into abstract forms**.

What is still a mystery however is what lies beyond this. That is, at what point a system can be considered intelligent, what it means to use AI techniques to solve problems and what the boundaries of these systems and techniques really are.

These questions are the focal point of this tutorial.

What is intelligence?

Before we can explain **artificial intelligence**, we must understand what intelligence actually is. Unfortunately, this is an extremely difficult question, one for which we don't actually have a globally accepted answer. Entire books across multiple disciplines - from psychology to philosophy, biochemistry and neuroscience - have been dedicated to discussing possible answers and definitions.

At its core, one of the main issues with defining intelligence is that we do not understand its purpose. Some objects or terms, such as money, have a clearly defined purpose. For example, the purpose of *money* is the storage and transfer of wealth. This makes defining the term very straightforward: "*A current medium of exchange.*"

On the other hand, not knowing the objective or actual purpose of something, makes it difficult to come up with a universally accepted definition.

One popular argument has always been that **intelligence aids long-term survival of a species**.

But as Stephen Hawking, Ernst Mayr and others eloquently pointed out: there is no clear evidence for this.

Less complex organisms - such as cockroaches or bacteria - tend to be much more robust than the intelligent *homo sapiens*, and have existed for hundreds of millions of years. On the other hand, mammals - a species much more sophisticated and with varying degrees of intelligence - have an average life expectancy of 100,000 to 150,000 years before going extinct.

Indeed, Enrico Fermi - one of the main physicists responsible for the development of the atom bomb at the Manhattan Project - raised the question why, given the massive size of the observable universe, and the probability of many other earth-like planets capable of sustaining intelligent life forms, we have not yet made contact with other intelligent life forms. The question became known as the "Fermi Paradox", and one of the proposed answers is the "Great Filter" theory, which in essence proposes that intelligence acts as a "filter" for any sufficiently advanced civilization. That is, any sufficiently intelligent life form will eventually evolve the means for self-destruction (for example by developing nuclear weapons or developing the ability to manipulate their environment to such a drastic extent that it will no longer be able to sustain them). According to this "Great Filter" theory, few intelligent life forms ever surpass this threshold of self-destruction, and it therefore "***is the nature of intelligent life to destroy itself***".

Not only do we not know the fundamental purpose of intelligence, we also have many different angles from which we can look at the term itself. For example, psychologists tend to look at intelligence from the perspective of behaviour or culture, linguists and philosophers from the perspective of understanding, and computer scientists in terms of the ability to solve problems.

Consider for example the following four definitions of intelligence:

"*The ability to acquire and apply knowledge and skills.*" - Oxford English Dictionary (Lexico.com)

"*Intelligence is not a single, unitary ability, but rather a composite of several functions. The term denotes that combination of abilities required*
for survival and advancement within a particular culture." - A. Anastasi, 1992.

"*[...] the ability to solve hard problems.*" - M. Minsky

"*Achieving complex goals in complex environments*" - B. Goertzel

The first definition - the one given by the Oxford English Dictionary - approaches intelligence from the perspective of *acquisition*. That is being able to gather, retain and apply knowledge.

The second definition on the other hand, coming from the perspective of psychology, ties intelligence to survival and advancement within an established group or species. This makes the definition of intelligence both location and context dependent. That is, placing an astrophysicist amongst an isolated tribe in the Amazonian rainforest would, by definition, make him/her less intelligent as his/her knowledge, skills and brain power would be unlikely to help him/her survive or advance in the jungle. Likewise, a tribesman would not be considered intelligent if placed in a modern metropolis.

Similarly, Minsky's and Goertzel's definitions are problematic if we were to universally accept them, as in order to determine whether someone or something is intelligent, we would need objective standards that define "hard problems", "complex goals" and "complex environments". This poses problems in and of itself, as the difficulty of solving certain problems or achieving certain goals tend to decrease as society becomes… more intelligent.

Furthermore, their definitions are difficult to reconcile with equally sensible definitions such as the one presented in the Oxford English Dictionary, which defines intelligence in terms of the ability to acquire knowledge rather than the ability to solve problems.

So, if we do not understand the purpose of intelligence and cannot agree on a universally accepted definition, then how can we even begin to define what it means to be "*artificially intelligent*"?

The truth is, that we can't. Not completely at least.

The boundaries between what people consider to be *intelligent, artificially intelligent* and *not intelligent*, will (most likely) always be blurry, and subject to disagreement. A lot comes down to perspective, context and intuition.

However, given that as part of this tutorial i) we are neither trying to discuss the purpose (or lack thereof) to our existence; and ii) the context (i.e. that of solving abstract problems) of our discussions are clear, we can view intelligence in terms of ***the ability to learn and use concepts to solve problems***.

We first came across this definition during a lecture by Maja Pantic, a Professor of Affective and Behavioral Computing at Imperial College London, leader of the iBUG group and the Research Director of the Samsung AI Centre in Cambridge. The definition presented by Pantic makes sense as it incorporates both skills, learning and the ability to solve problems, whilst leaving out the more problematic areas such as culture, behaviour and complexity.

Although the definition may not satisfy everybody, it provides a general basis for accepting or rejecting whether an organism or system behaves intelligently. It therefore allows us to move on to defining what it means to be *artificially intelligent*.

The beginnings of artificial intelligence

Having discussed what we mean by intelligence, it is now time to direct our attention back to the silicon and the switch, and look at how the idea of using machines to perform human cognitive tasks, came about.

Following the invention of boolean algebra, the next major epoch in the history of the computer would prove as an eminent turning point in one of the most devastating wars ever fought; World War II.

At the time, Germany commanded the largest submarine fleet in the world, sinking approximately one British vessel every four hours. To coordinate their attacks, the Nazi's developed a complex code known as the "Enigma Code" (Figure 1 depicts the Enigma Machine).

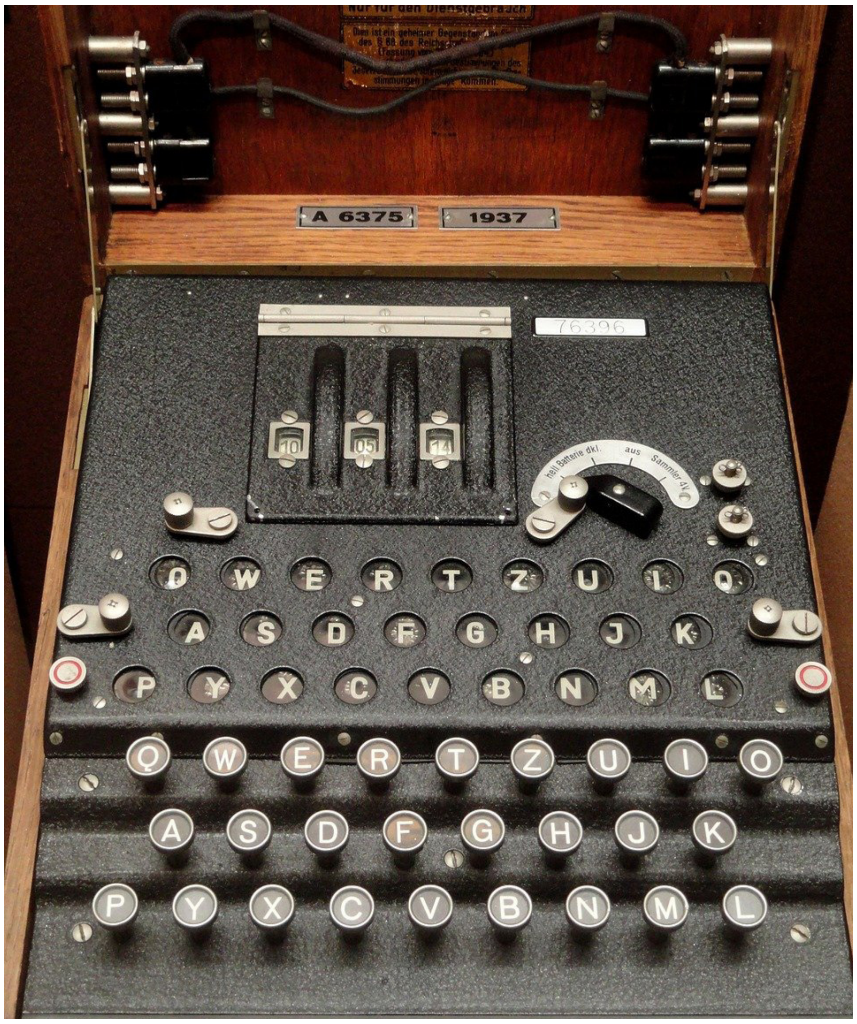


Figure 1: The Enigma Machine

Facing annihilation, the British Government established a top-secret group of intellectuals, dedicated to breaking the Nazi's "unbreakable" code. Mathematicians, chess players, codebreakers, astrologers, secret service officers, technicians, crossword experts and even actresses worked shifts around the clock in an old estate a few kilometers outside of London.

Among them was Alan Turing, a British mathematician who, prior to the war, came up with the concept of the first computer. His machine, called the *Turing Machine (see Figure 2 on the next page)*, was imaginary with the aim of processing equations without human direction. The idea was a conceptual breakthrough and resembled a typewriter that used logic to compute any imaginable mathematical function.

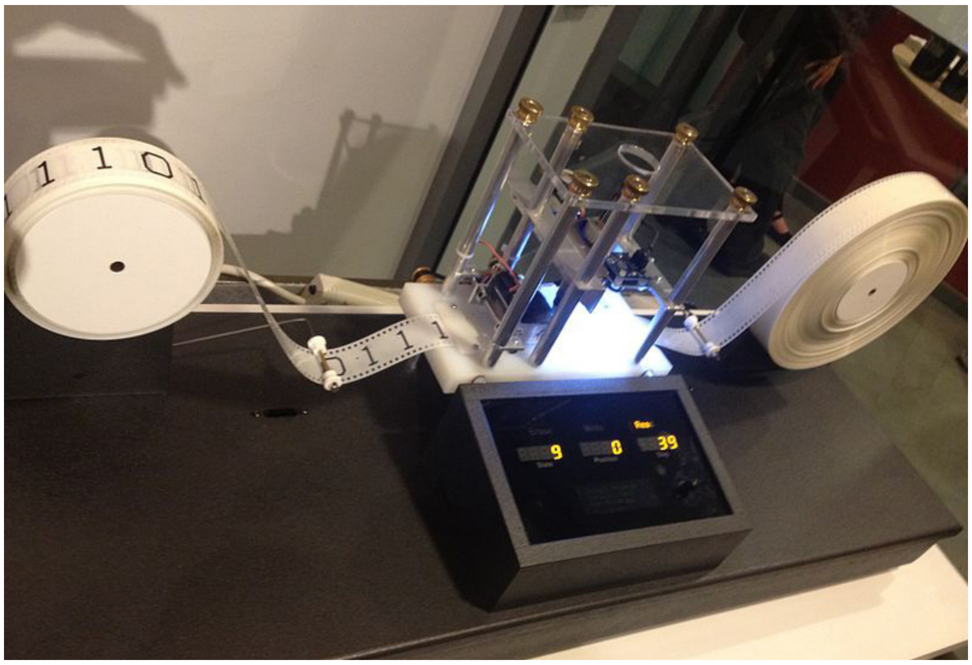


Figure 2: The Turing Machine

To this day computer scientists use the concept of "*Turing Completeness*" to measure computational systems. A machine is said to be Turing-Complete if it can calculate all possible mathematical functions that could ever exist. Based abstractly on his previous work, Turing, with the help of Polish intelligence, developed an electromechanical machine capable of cracking the Nazi cipher. The "Bombe", for this was what they called the ton heavy machine, was to be the predecessor to the first digital computer. By the end of the war, approximately 200 Bombe's were employed by the allied forces, and Alan Turing was to be awarded a medal for his services to the British Government.

However, the Bombe did not prove to be the universal solution the allies had hoped for, as the Germans soon realized that their encryption mechanism had been broken and set about constructing an even more complex code which the allies promptly referred to as "The Fish".

Once again it were the British, together with allied intelligence, that came to aid. At their code- breaking center at Bletchley park they constructed the world's first programmable digital computer, called the "Colossus" (see Figure 3), which they used to decode the German cipher.

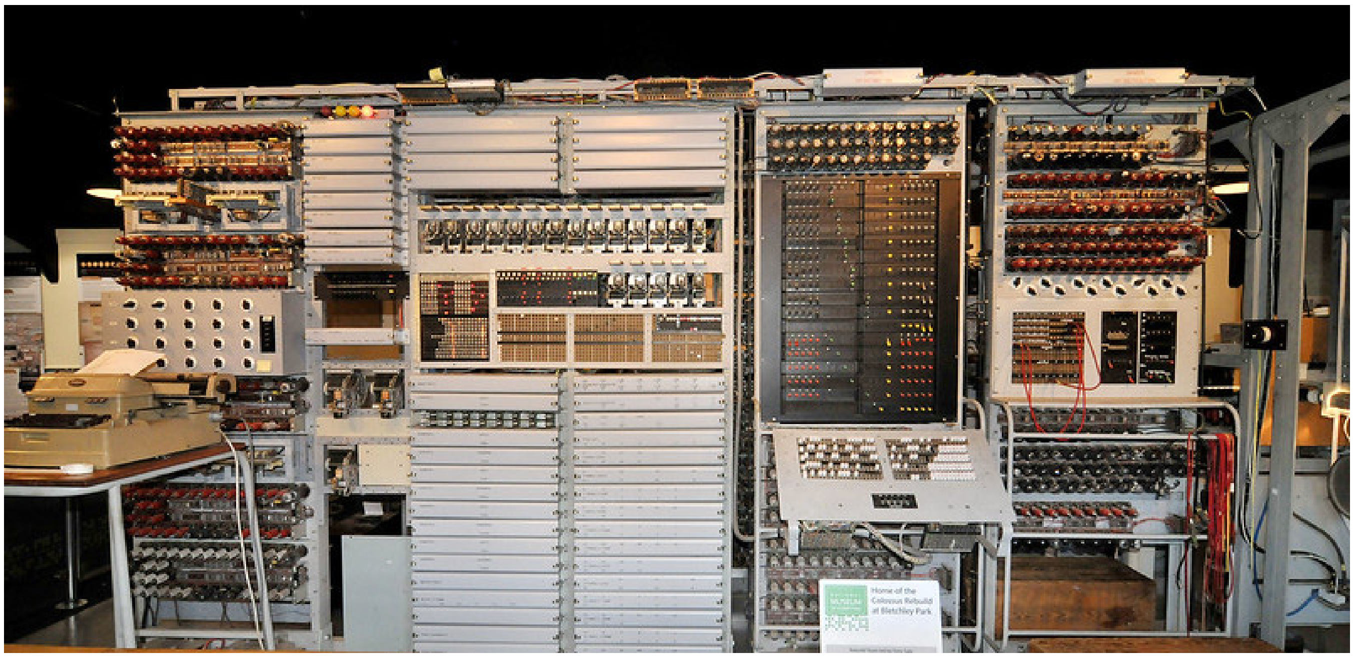


Figure 3: The Colossus Computer

This machine was vastly inferior to today's computers, but what a sight it must have been; the size of a room and consisting of 1,800 vacuum tubes, yet incapable of performing simple decimal multiplication. Its construction was top secret and it wasn't until the 1970s that the British Government acknowledged its existence.

It was due to this that the great minds behind the birth of the computer were never credited. Instead it was the ENIAC (Electrical Numerical Integrator and Calculator, 1946-1955) that laid claim on being the world's first digital, high-speed computer (see Figure 4). Developed by the US Government to calculate ballistic firing tables, the ENIAC was much larger than the Colossus, required 548 square meters of floor space, weighed more than 27 tonnes and consumed approximately 140,000 watts of electricity.

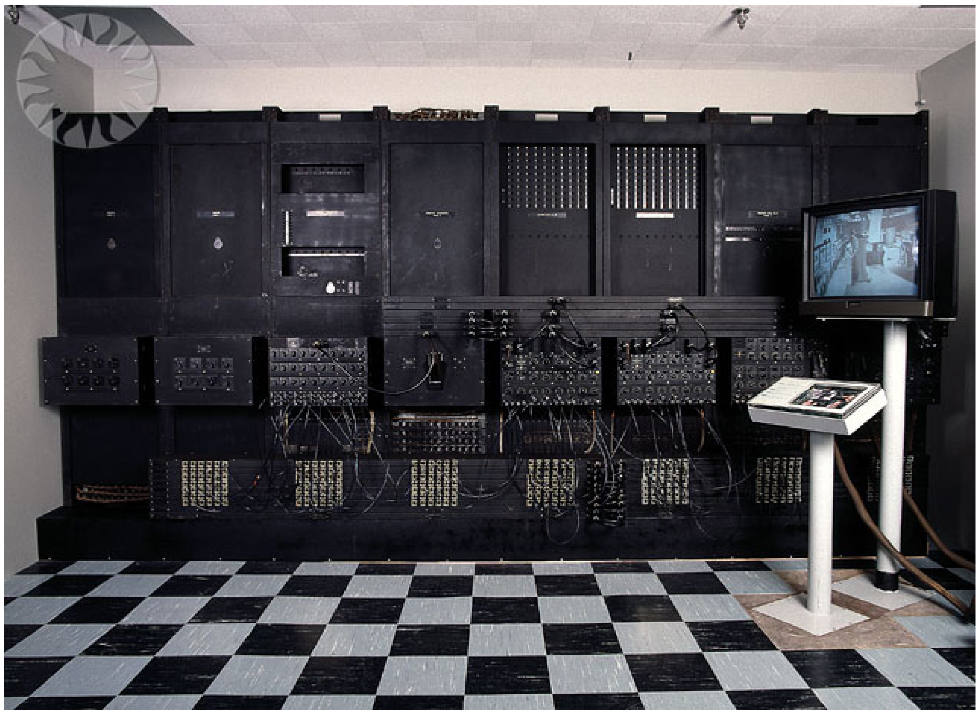


Figure 4: ENIAC, the First Digital General Purpose Computer

For each new program, the ENIAC had to be re-wired, as, unlike today, computer programs could not be stored by the computer. Therefore, a lot of the work was hard manual labour and it took a team of 6 technicians to maintain the machine. The vacuum tubes that made up the actual processor had to be replaced on a daily basis, as the machine consumed approximately 2,000 tubes per month. The machine could predict the effect of weather and gravity on the firing of shells, mathematical equations that could take a human up to one day to solve. Not being able to store data proved to be laborious, and whilst Turing foresaw the development of stored program computers, it was a mathematician by the name of John von Neumann who first authored the concept.

In 1945, von Neumann published a paper that outlined the architecture for such computers, dividing their inner workings into four sections; the central control unit, the central arithmetical unit, memory and input/output devices. Von Neumann was accused of plagiarism by his team of researchers and consequently split up with them. However, the **construed architecture remains at the core of every computer to this date**.

Although the calculations performed by the Colossus, the ENIAC or the Bombe were complex, and solved real, cognitive puzzles and problems that could not be easily computed by a human brain, they were not considered "smart" or "intelligent". The machines were complex, and built by the brightest minds, but at the end of the day, could not acquire knowledge, or learn by themselves.

To use the definition that we set out in the previous section, these machines did not have "*the ability to learn and use concepts*". Instead, they followed fairly simple mathematical rules. What gave them an edge over the human brain, was that they could do so very quickly and reliably.

In 1943, in parallel to all wartime advances in information technology, Warren McCulloch and Walter Pitts started using their knowledge of biology and the human brain to construct a computational model that modelled neurons using the concepts of the switch's "on" and "off".

They showed that using this model, they could mimic the functioning of the human brain whilst computing any computable function. Their work gave rise to the first "neural network" (which we will discuss in detail in an upcoming tutorial in this series about Neural Network) and became the first officially accepted work on artificial intelligence. Their demonstrated way of computing brought information technology one step closer to the functioning of the human brain.

In 1951, Dean Edmonds and Marvin Minsky built the first actual physical computer based on this model whilst at the University of Princeton. And at around the same time, Alan Turing published his famous article "Computing Machinery and Intelligence", which laid the fundamentals of research into artificial intelligence by, amongst other things, introducing a test for determining whether or not a machine is capable of exhibiting characteristics of human thought. The test, known as the "Turing Test", is described concisely by Stuart J. Russel and Peter Norvig, in their book: "Artificial Intelligence - A modern approach":

"Rather than proposing a long and perhaps controversial list of qualifications required for intelligence, he [Turing] suggested a test based on indistinguishability from undeniably intelligent entities - human beings. The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. [...] Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because physical simulation of a person is unnecessary for intelligence."

Although the test is a hallmark in artificial intelligence, and the paper in which it was presented gave birth to many concepts in the field, the test itself is of no major relevance to modern AI. As many observers correctly note: **Imitation is not the same than implementation**.

Russel and Norvig, a mere half a page further down from their description of the test state: "The quest for 'artificial flight' succeeded when the Wright brothers and others stopped imitating birds and learned about aerodynamics. Aeronautical engineering texts do not define the goal of their field as making 'machines that fly exactly like pigeons that they can fool even other pigeons'".

As such, AI researchers concern themselves with studying the human brain and thought processes, and determining how to best build systems that can acquire knowledge and solve very specific problems (such as playing chess or recognizing cancerous growths in medical imagery).

Using our definition of intelligence, AI researchers concern themselves with creating computer programs that have *the ability to learn and use concepts to solve problems*. Given how difficult it already is to merely define intelligence, studying it is an even greater undertaking. Therefore, building systems that behave in an intelligent manner requires not just computer science, but knowledge drawn from a wide variety of fields, such as economics, linguistics, behavioural and cognitive psychology, biology and neuroscience, mathematics and philosophy.

Each of these fields tries to answer unique questions, from whose answers computer scientists working on intelligent systems draw conclusions and solutions, and in many cases try to implement them.

For example, economics tries to answer questions on how to best act under uncertainty, how to make decisions that produce optimal outcomes or how self-interested parties should best interact with each other.

Linguistics explains the structure and meaning of words.

Neuroscience on the other hand studies the physical functioning of the brain, whilst by looking at psychology we can draw general conclusions about how humans and other intelligent animals behave.

Similarly, by studying philosophy we can gain insights into logic, meaning and how we arrive at certain conclusions.

By combining theories and insights from these different fields with mathematics and engineering we can, to some extent, develop software that can use historic data to learn and solve difficult problems within a specific domain.

Advancements in engineering

Implementing and using theories gained through fields outside of engineering and computer science to solve domain-specific problems still required actual advances in engineering.

Scientists and engineers needed the actual physical capability for machines to calculate hundreds of thousands of instructions per second in order to actually realize their implementations. Therefore, advances in artificial intelligence - which as we saw in the previous section is married to a variety of different fields and sciences - relied heavily on actual advances in engineering.

Therefore, as the idea of using machines to help us solve real problems - whether by means of simple mathematics or with the aim of producing *intelligent* systems - began to set foothold among government officials, British, Australian and American Universities began investing in more research projects on the topic of engineering and computer science. Researchers at different locations around the world - Cambridge, Sydney and Manchester - began work on the first stored program computers. Their work soon bore fruit, triggering worldwide academic interest in the development of computers.

The technological revolution whose foundations were laid by warfare, was soon to be embraced by intelligible business men. In 1952 an American company, IBM, who up to now had produced calculators, decided to enter the computer market by releasing the first commercially successful general-purpose computer.

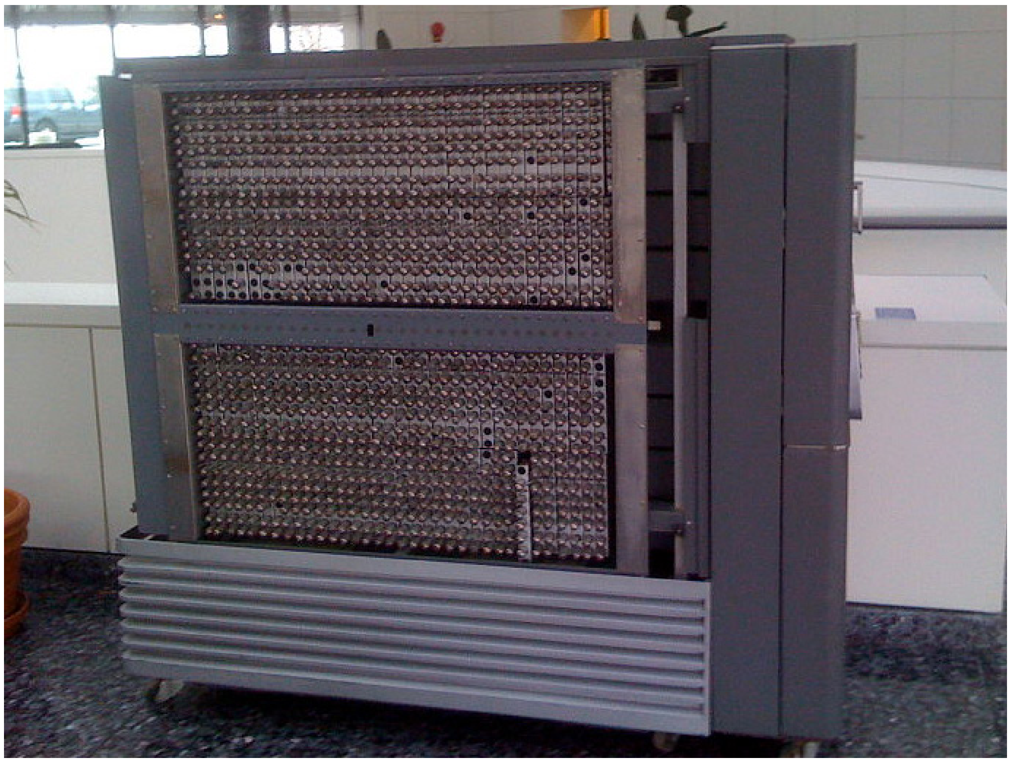


Figure 5: The IBM 701 EDPM Computer

The IBM 701 EDPM Computer could execute 17,000 instructions per second and was available for rent at $15,000 per months. The nineteen models sold, were in use primarily by government bodies such as atomic research laboratories, aircraft companies, weather bureaus and even the United States Department for Defense. With such a demand in super-computers, IBM quickly realized that their future lay in computing technology. In 1954 another model was released, this time for use in Universities and business and by 1958 their computers could execute up to 229,000 instructions per second. IBM's products were in high demand, some costing up to $3 million, with others available for rent at up to $60,000 per month.

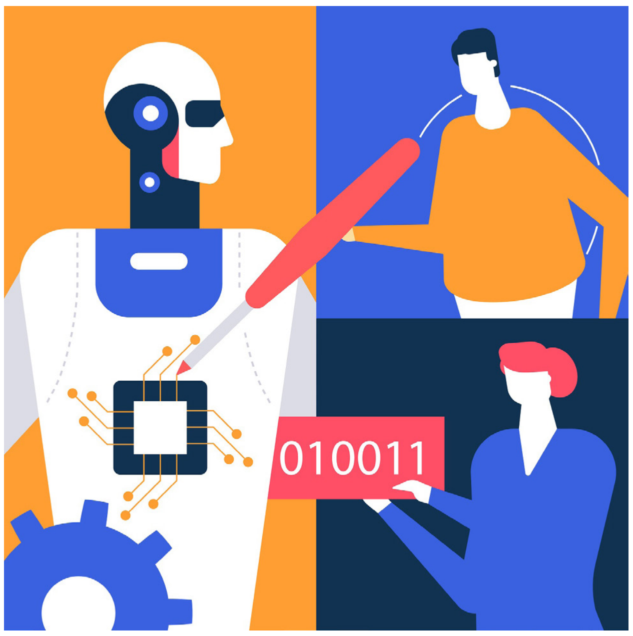
Considering computers and information warfare as a means for the cold war arms race, the US government invested large sums of money to the benefit of corporations and Universities. With such a bright future ahead, interest among academic communities kept growing as more and more Universities began teaching Computer Science degrees. From now on, America became the primary breeding ground for information technology, concentrating many of its resources on technological advancement.

The 1960's were the years for technological revolution. Computers could solve all forms of mathematical equations, and given enough money and time, mathematicians began mirroring more and more aspects of our life in mathematics. Some of this knowledge began manifesting itself amongst large corporations, who saved a lot of money by employing machines that could not only solve problems faster than the human mind, but also store massive amounts of information using a minimal amount of space.

One such example of commercial application that survives to this day is the SABRE system employed by American Airlines. SABRE was developed in the early 60's and was the world's first automated transaction processing system that now connects over 30,000 travel agents with approximately 3 million online users. Others include UNIMATE, the first industrial robot employed by General Motors and the first Computer Aided Design program "DAC-1".

By the mid-seventies, computers had embodied themselves inside most large US corporations and the diminishing cost of hardware together with increasing computer miniaturization created new markets, opportunities and frontiers. Medium to small sized businesses, for the first time in history, could gain access to low-cost computers. Hobbyists and students took advantage of this and began purchasing computer kits. This proved to be the beginning of the personal computer era, and would eventually lead to the wide-range adoption of computers, and hence artificial intelligence.

Solving problems using Artificial Intelligence



So far in this tutorial, we have spoken about how we could go about defining intelligence (as *the ability to learn and use concepts to solve problems*) and how creating systems that behave in an intelligent way requires the combination of a wide range of fields and theories.

We saw that advances in computing are closely linked to wartime efforts, and how complex calculations performed by computers at the time - such as calculating missile trajectories - are difficult for humans to perform, but easy for computers to solve quickly and correctly. However, we also learned that, whilst performing these complex calculations might make the machine *appear* smart, *appearing* intelligent is not the same thing than *behaving* intelligently.

If performing complex mathematical calculations - such as missile trajectories - that are difficult even for the smartest human to solve (and to do so correctly and consistently) is in itself not considered an application of artificial intelligence, then what is?

What type of problems does a computer need to solve in order to be considered *intelligent*?

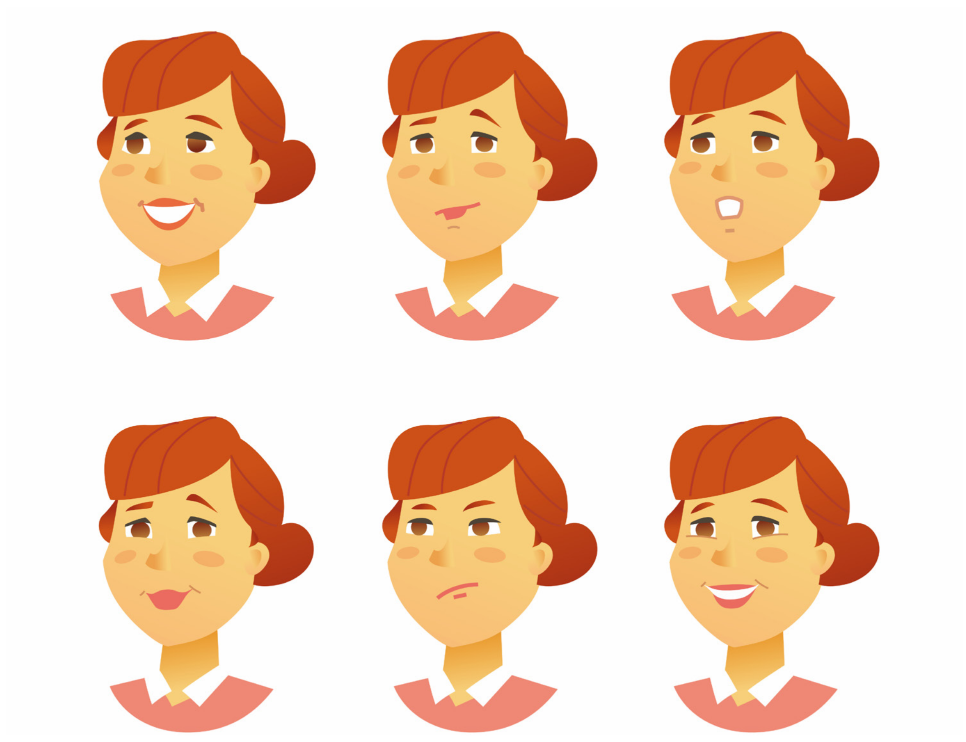
Broadly speaking, artificial intelligence concerns itself with problems are difficult to define precisely. That is, problems that we can't formulate abstractly in a straightforward way so that the computer can just follow a single step of instructions. Mathematical equations, such as those for addition, subtraction or the calculation of a trajectory, might be difficult for our human brain to solve quickly, but their solution can be precisely defined using a simple set of rules.

The problem of multiplying 2 by 2 for example, is easily defined by the rule for multiplication. Once this rule has been defined, a computer then needs to just follow the rule every time that we want to multiply two numbers together. The computer will *precisely follow a precise definition of the problem* and arrive at a consistent solution. There is neither variation in the problem definition itself, nor is there potential scope for ambiguity in the solution itself. Returning to our mathematical example: multiplication will always be multiplication. It will never mean something else if the input data itself changes. If we were to multiply 2 by 2, then the computer will follow the actual rules for multiplication in the same way than if we were to multiply 250 by 12. Likewise, the solution itself is always clear: 2x2 will always produce 4, and 4 will always just be a number. It has a precise meaning that, within the context of the multiplication, won't change.

On the other hand, problems that can't be defined precisely, require a certain amount of knowledge, experience and *feeling* to solve.

Consider for example the problem of identifying emotions in facial expressions.

If you see a friend, or family member, you will most likely be able to read their emotional state - whether they are happy, sad or angry - by looking at their facial expressions. But trying to define a precise set of abstract instructions for defining the emotional state given a facial expression would be very difficult, if not impossible.



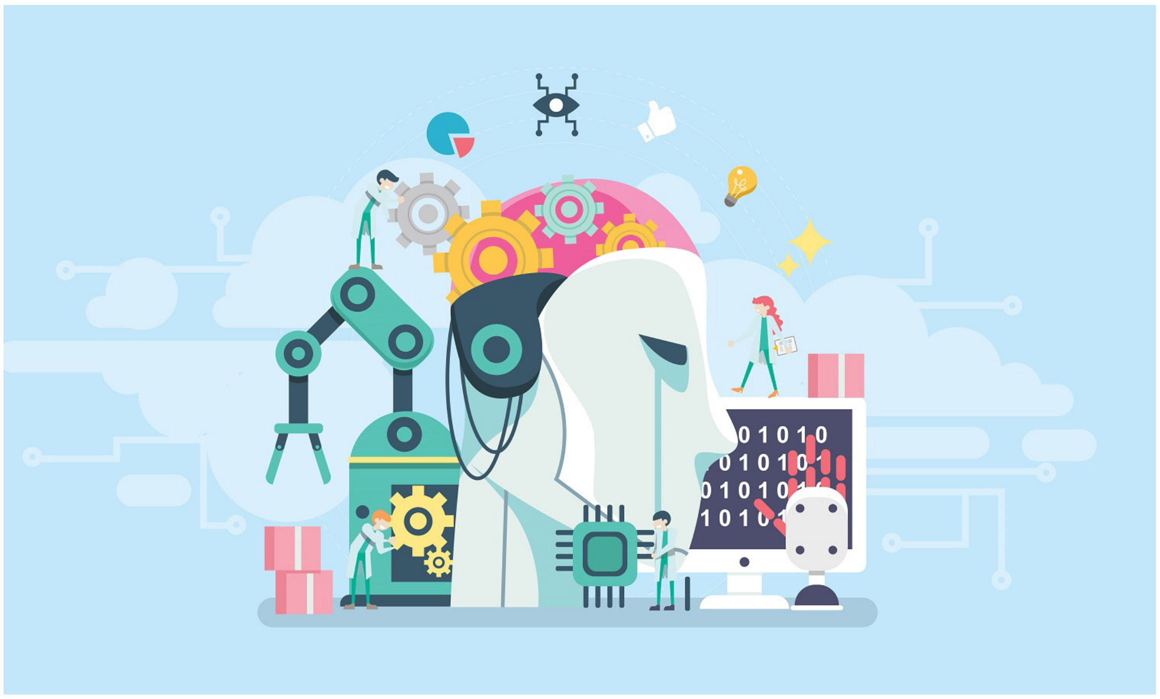
Even if you were to succeed in creating a formula that defines, for example happiness on your friend's face, chances are that this abstract formula will not apply to other faces. This is because each face is unique - the dimensions, colours and shapes vary, and so do the facial expressions given a certain emotional state. For example, when smiling, some people might show their teeth more than others. Others might smile more with their eyes and less with their mouth. And so on, so forth.

Unlike in the case of multiplication, following one precise set of instructions here will not work for detecting emotional states. When you identify an emotional state, you do not follow one or two simple, abstract rules. Instead, you build on a rich history of having encountered different faces and emotional states as a child. You learned through reactions and experience, and subconsciously built up a model in your mind that now allows you to quickly recognize the physical patterns in someone's appearance that convey an emotional state.

Furthermore, unlike with multiplication, there is always a certain level of uncertainty when it comes to problems that can't be clearly defined. When multiplying 2x2, the answer will always be 4. But when trying to detect an emotional state, we may not always make the correct judgement call. We have all been in a situation in which a person's reaction or mood surprised us: maybe we thought they were angry at us, whilst in fact they were merely stressed or worried. This means that every time our mind performs the subconscious series of "calculations" for detecting an emotional state, there is a chance for error. Or in other words, there is a numerical certainty (or *probability*) identified with each emotional state that we classify.

Last but not least, if we were to ask a computer to determine an emotional state based on a picture of a face, then a certain amount of variation is introduced with each photograph that we pass into the computer. When multiplying two numbers, the value of the two numbers that are multiplied might vary, but we are sure to always multiply actual numbers. With the photographs however, there will always be small (or large) difference that we need to account for. In some photographs, the face might be closer to the lens and hence appear larger. In another photograph, the person might appear slightly further away. Noise, contrast and lense quality might introduce additional variations. The problem itself is therefore never quite the same.

Artificial intelligence is exactly about ***solving such complex problems that at times are ambiguous and always difficult to formulate abstractly.***



Instead of following a simple set of abstract instructions, artificial intelligence tries to use a *model* of the world to arrive at a solution.

A model is in essence a simplified or abstract representation of the world - or parts of it - and can be constructed in a variety of shapes and forms. Some are constructed using ways of detecting patterns in knowledge or data that we have collected in the past, and try to represent these patterns in such a way that they can easily be used to make a prediction or classification using new, unseen data (this is the essence behind machine learning). Others are complex mathematical models that express the world using logic and rules that determine what we can or can't do within this model (difficult scheduling or timetabling problems are often solved in this way). Again, other models are generated on the fly as need be.

Coming up with and implementing these models is the main challenge faced by AI researchers. That is, the challenge lies in creating representations of the world that are accurate enough to help us solve complex problems, yet are sufficiently simplified so that they exclude any irrelevant data or assumptions that could negatively impact the result.

Not only do the models need to be an accurate representation of the world or domain in which they are used, they also need to be general enough to account for variations in input data or unexpected changes.

Last but not least, the data used to build the model needs to be organized in such a way that a computer can easily and quickly use it to determine the solution for our problem. Just like a phonebook organizes entries in such a way that we can easily look up a person's number, the model used to solve needs to organize data in such a way that a computer can process it within a reasonable amount of time. As we will see the upcoming tutorials of this series, these are no easy challenges.

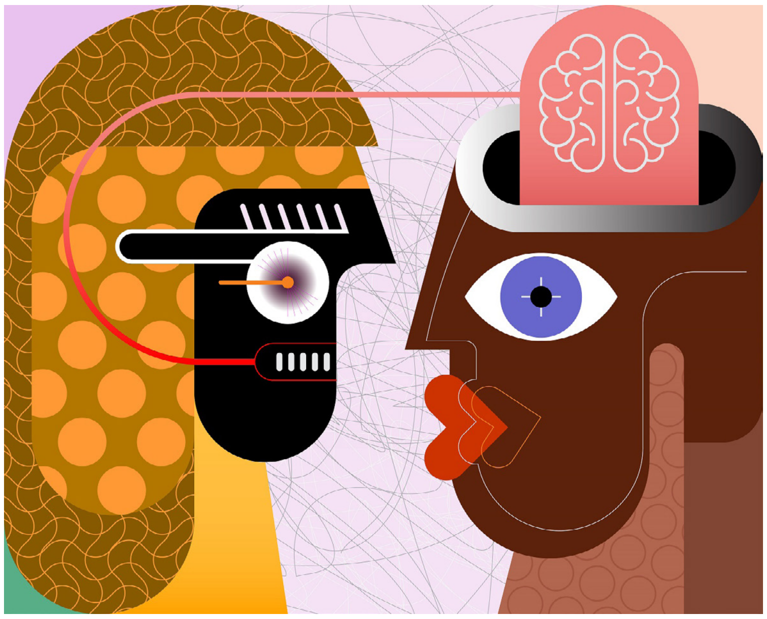
Self-consciousness, subjective experiences and emotions

We often see films, read science fiction novels or listen to pundits on podcasts that depict or predict all-mighty, all-powerful sentient machines. The popular media has sufficiently promoted and intertwined the concept of consciousness with advances in artificial intelligence, that we felt we must briefly speak about the importance (or lack thereof) of s*elf-consciousness, subjective experiences* and *emotions*.

In the previous section, we described how intelligent systems use *models of the world* to learn about, and solve problems. Self-consciousness is the ability to understand one's own position in this model - or, the world. Similar to intelligence, we know very little about consciousness and subjective experiences, how they come about and how they aid long-term survival. But in terms of short-term survival, several arguments exist that aims to explain its immediate existence.

Of these arguments, two are most prominent.

The first suggests that s**elf-consciousness is a mere by-product of our brain**. That is, it argues that consciousness is an emergent property that is the result of a wide range of complex processes - such as emotion and memory - working together. It attributes no actual value to subjective experiences and consciousness itself, and just accepts them as is.



The argument makes sense if we look at the myriad of tasks and actions that we need to accomplish each day in order to stay alive: none of them really requires us to be aware of them. Determining whether a fruit is edible, for example, requires that our sensory abilities - such as taste, smell, touch and sight - are sufficiently adjusted for the task. Understanding our position within this process or "experiencing" the process in its full richness is not really required to achieve this objective. Similarly, finding water, defending ourselves against predators, resting or mating, can be defined and accomplished in terms of rather "mechanical" processes and does not really require our full awareness.

The second line of thought argues that **self-consciousness allows us to understand our position in the world better**, and hence permits us to simulate outcomes of different decisions more effectively. In essence, it argues that self-consciousness provides us with a deeper understanding of the world, and the impact of our actions on it. By being *aware* of the effect that we play on our environment, we can make more complex decisions, plan and think further ahead than if we were merely a collection of cells that react to biochemical signals.

Furthermore, the more we understand ourselves and are aware of ourselves, the easier we can interact and communicate with other beings around us.

Some authors and scientists, such as Noah Yuval Harari, point out that it is really our social ability that separates us from other species. Specifically, our ability to organize ourselves in large numbers. Social organization is something that insects - such as bees or ants - are very good at too. And undoubtedly, their level of self-awareness and consciousness is much lower than ours. However, so is the upper limit on the size of their social structures.

Many colonies vary greatly in their thousands, whilst some reach up to 300 million. The level of human organization however stretches into the billions.

Therefore, is self-consciousness a strict requirement for complex social interactions? And do these interactions indeed aid long-term survival?
The short answer is that we don't know.

There exist opponents and proponents in both camps, many with sensible arguments. However, at least for now, the arguments on both sides are largely irrelevant to AI researchers to date. That is, regardless of whether one believes consciousness to be an emergent property, or whether one looks at it in terms of a complex simulator, to the best of our knowledge, consciousness is in and of itself is irrelevant when it comes to the development of intelligent systems. The domain-specific types of problems solved by computer scientists working in the field of AI don't require the computer to actually be aware of its role in the process of determining a solution.

As we will see in the upcoming tutorials in our Machine Learning for Everybody series, the process of, for example differentiating between images of cats and dogs, does not require the machine to be *aware* of the fact that it is identifying the differences. Similarly, as we saw in the first part, finding the shortest paths between two points on a map, requires a model of the map and a technique for finding the shortest path using that model. At no point were we required to simulate some form of self-awareness on behalf of the computer. The same can be said about emotions.

Whilst AI researchers have an entire field - called "Emotion AI" - dedicated to the study, analysis and implementation of emotions, producing a machine that can actually "*feel*" (whatever that actually means) is not strictly necessary for the development of intelligent systems. Whilst having a machine that can detect, "read" or simulate emotional states certainly has a wide range of benefits (from improving the user experience of interacting with a machine to threat detection) and can help solve domain-specific problems involving emotions, it is not a prerequisite when developing intelligent systems.

In animals, emotions tend to serve as a quick-reaction mechanism, that circumvents the slower thought processes of the brain. For example, pain acts as a protection mechanism by quickly signaling to our body that "something is wrong". If we were to accidentally touch a hot stove, nerves in our skin cells send a signal (pain) to our spinal cord that causes us to quickly withdraw our hand. The withdrawal is automatic and does not actually require us to *think* about the situation or process. We just *do* it. Similarly, emotions such as anger or *the feeling that something is wrong* help us function in certain situations without requiring us to think deeply about the situation itself. Emotions and intuition have evolved over thousands of years to help protect organisms from adverse circumstances or to promote certain behavioural patterns. They help us learn more effectively from others without having to experience certain situations ourselves, and allow us to communicate, organize and maintain beneficial relationships.

Whilst there are clear evolutionary benefits to having emotions, just like self-consciousness, there is no clear advantage in building emotional machines for most of the narrow, domain-specific problems that are solved using AI (unless of course detecting, interpreting or displaying certain emotions is part of the actual problem definition). The fast reactions that emotions allow us to make, or the ability "record" memories using certain emotions (such as intense fright) are implemented using other means when operating within the context of the digital computer.

Problems solved using AI

By now, it should be clear that **when we talk about developing intelligent systems, we don't imply the development of all-knowing, emotional, sentient robot overlords**. Instead, AI researchers are concerned with **developing solutions to specific problems** - problems that are not easily expressed and solved using a set of abstract formulas, but problems that are often "fuzzy" and difficult to define.

Intelligent systems that attempt to solve these problems tend to act within a flexible environment, where input data is subject to a large degree of change and variation and where solutions often are not 100% clear cut but instead involve a certain level of uncertainty. So far, these definitions and explanations all sound very abstract, so let's take a look at some of the problems that are solved using artificial intelligence.

Sample problem #1: Search

Search problems, as their name indicates, revolve around the process of *searching* for a solution given a large amount of data. The route planning problem discussed in the first part is a perfect example of such a problem. As part of the route planning problem, we modelled the map as a graph on which we used a search algorithm to find the shortest path. Our algorithm used the path length (i.e. number of nodes to traverse) as a performance measure, selecting the path which requires us to traverse the minimum number of nodes.

HERE
O
M
N
P
U
R
Q
T
S
DONS

Search problems are well studied, and a wide range of general-purpose algorithms exist to help solve such problems. Route-planning is an intuitive example of a search problem, but is by far not the only one. Many real-world problems outside the space of navigation can be formulated as search problems. For example, robots that perform a certain series of actions such as hoovering different tiles in a room, often "decide" what action to perform next by searching for a certain sequence of decisions or actions that might lead to a goal state (such as having a clean room).

Search problems are formulated by

i) defining a range of possible states and their relations (these form the graph nodes),
ii) defining a goal state and
iii) defining a performance measure (such as minimizing the number of nodes to traverse or assigning weights to the edges between nodes and trying to minimize the overall cost of traversal).

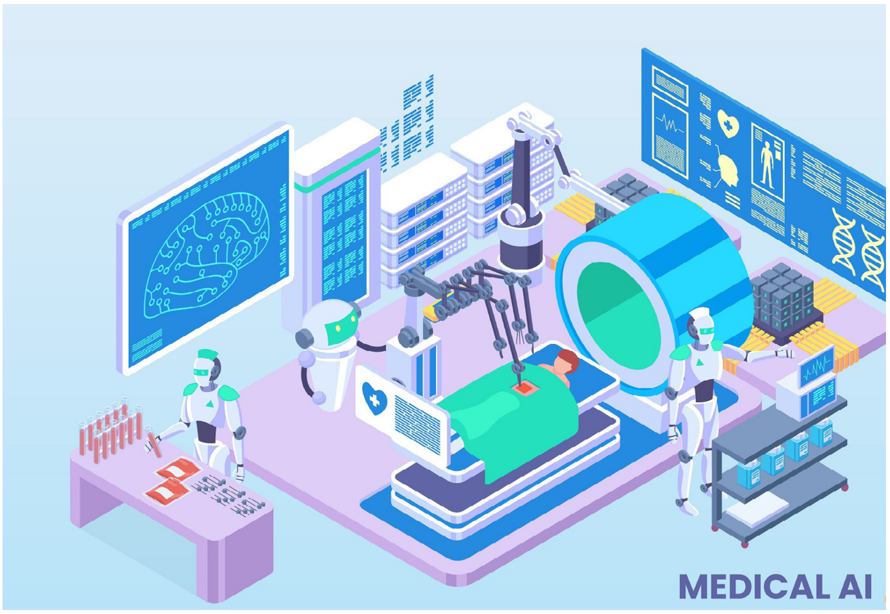
Whilst search problems are a whole field of study in themselves, the models, algorithms and techniques are often shared, used and cross-referenced across other sub-fields of artificial intelligence.

Search problems fit the definition of the types of problems being solved by intelligent systems, as they deal with large amounts of data and a search space that involves a large degree of change and variation (just think about the search space for a route planner might change as the vehicle itself moves, traffic increases or decreases and the destination is changed).

Sample problem #2: Medical diagnosis

Despite decades of research, we do not yet fully understand exactly how the human brain processes images.

Nevertheless, computer vision has been an active topic of research, and machine learning techniques have been applied to detect patterns in images, and recognize and classify objects. One concrete example of the utility of using artificial intelligence to identify certain shapes in images is *medical diagnosis*. Here, computer software is used to diagnose medical conditions based off of MRI scans. For example, the computer can be used to detect tumours, sometimes more accurately than human beings. As part of this, machine learning techniques are used to "train" the software using medical images labelled by experts (how exactly this works we will discover in the upcoming chapter). When feeding in large enough amounts of data, patterns emerge. The AI uses these patterns to then classify the images or detect certain objects.



Given how image quality, organ shapes and sizes, bodies and tumour size, shape and position varies, we see how medical diagnosis, or image recognition in general, fits the criterion of "uncertainty" which makes the problem a tough and suitable one for artificial intelligence.

Furthermore, describing a medical diagnosis using an image and a concise formula or series of steps is difficult. Whilst the problem itself is very domain-specific (again, a characteristic of problem-solving using artificial intelligence), the problem of detecting a cancerous growth itself is difficult to define to begin with, and, from a human perspective, relies a lot on past experience and "intuition".

Sample problem #3: Scheduling, resource allocation and timetables

Constraint programming" is a subfield of artificial intelligence that was pioneered by Eugene Freuder. The technique is used to solve "*Constraint Satisfaction Problems*" (CSPs). These are models that are created by describing the world (or problem) in terms of variables and the possible restrictions (the "*constraints*") on those variables, along with a possible set of values that each variable can assume (the "*domain*").

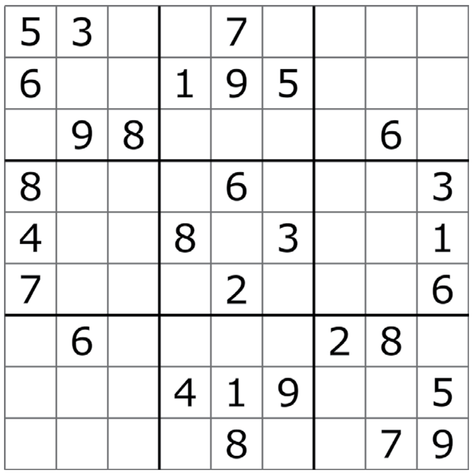
For many types of problems, this is a very natural way of expressing them, which led Eugene Freuder to say that "*Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.*"

A classic example of a constraint satisfaction problem is the Sudoku puzzle, which is often found on the backs of magazines or newspapers. This puzzle - usually a 9x9 square - requires users to fill numbers into the grid in such a way that each box, each row and each column contains the numbers 1 - 9. The numbers cannot repeat themselves (for example, a single row cannot contain the number 5 twice) and a few cells come with preset numbers which cannot be changed. Modelled as a constraint satisfaction problem, the variables in this case refer to the individual cells that need to be completed (see figure 3.1); the "domain" are the numbers 1 to 9; and the constraints are the following rules:

1. Each box must contain only the numbers 1-9. Each row must contain the numbers 1-9. Each column must contain the numbers 1-9.

2. The numbers 1-9 cannot repeat themselves within each box; row and column.

3. Certain cells must have the assignment of certain preset numbers.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Although a Sudoku puzzle is just a simple example of how we can apply the logic behind constraint programming, constraint satisfaction problems themselves can become extremely complex. They are a powerful method for solving real-world problems that include a large number of restrictions on their possible outcomes. Concrete examples of such problems include:

- **Scheduling:** The automatic creation of timetables or schedules, whereby entities (such as rooms and participants) have certain limitations (capacity restrictions, availability etc

- **Procurement:** Finding the cheapest suppliers based on capacity limits and other criteria. For example, given a large (think hundreds or thousands of items) tender for sourcing different types of vaccines, different bidders might place bids at different prices for each vaccine type. The purchaser would want to identify the cheapest bidders, using not only the prices submitted, but maybe additional criteria, such as an upper limit on the total number of different suppliers.

- **Resource allocation:** Factories often produce different types of products. Some of the raw materials used to produce these products are abundant, others are scarce. Factory owners might wish to determine how to allocate these scarce resources most efficiently, so that profit can still be maximized without affecting other constraints.

By formulating problems using variables, constraints and domains, computer scientists can use other techniques from artificial intelligence - specifically, search algorithms - to find solutions to these problems.

It is also important to note that CSPs tend to be very dynamic, in that the constraints are added on the fly and therefore don't require a static problem formulation. This makes them ideal for solving real world situations where restrictions arise, change or disappear quickly. Depending on the problem, the change in even a single, simple restriction can produce thousands, hundreds of thousands or millions of new possible states or solutions.

Sample problem #4: E-mail spam filtering

In 2004, Bill Gates famously predicted that "*Spam will be a thing of the past in two years' time*". His prediction was off by almost 20 years, but at the time of writing it seems to finally come true.
Companies like Google have becoming fairly effective at eliminating spam from their products. Gmail - Google's email service - does an impressive job at classifying spam messages correctly and hiding them from view. This is largely thanks to the massive amounts of data available to the company, that allows their machine learning and natural language processing algorithms to function very effectively (just how important data is to machine learning, we will see in the upcoming chapter).

Classifying emails as spam involves reasoning with uncertainty, and draws upon a range of different fields, such as natural language processing, probability theory and machine learning. As the contents of spam emails and techniques used by the spammers change, the spam filter needs to continuously update and "learn" from the new data.

Sample problem #5: Product recommendations

Recommender systems are becoming ever more ubiquitous. A very profitable example of a recommender system is Amazon's product recommender. Based off of the types of products that you have looked at, purchased or rated in the past, Amazon will suggest products to you that "you might also like". These suggestions appear under titles such as "*Customers who bought this item also bought...*" when browsing their website.

How does Amazon do this?

Using the data available to them, Amazon creates a complex web of product interactions, purchases, ratings and search terms to predict your likes and interests as accurately as possible.

In more general terms, what recommender systems, such as Amazon's product recommender system, do, is build up profiles of thousands, hundreds of thousands or millions of users, based off of their individual behaviors and interactions with a system, application or website. Often times, slightly unethical approaches are also used by companies to factor in third party data that was collected by other websites or services.

As you browse a website or set of websites, these recommender systems continuously adapt and update your profile, factoring in any new behavioural patterns that you might exhibit. Using machine learning techniques, different methods of organising data (such as things like ontologies which are ways of organising concepts and identifying relationships between different concepts) and things like "cluster analysis", these recommender systems can make shockingly accurate predictions as to your likes, dislikes and intentions.

Limitations of AI

We have just seen some concrete examples of the types of problems that can be solved using artificial intelligence. By observing these examples carefully, you will have hopefully seen that these examples are not easily expressed and solved using a set of abstract formulas. The problem definitions themselves can be rather vague and the contexts in which they are framed involve uncertainty and constantly changing data.

Instead of being able to define a set of abstract formulas, these types of problems are solved by building an abstract model of a very specific problem domain, which are used to “infuse” the computer with “knowledge”.

In just four words of the preceding paragraph lies the key to defining the limitations of artificial intelligence: “*very specific problem domain*”. That is, artificial intelligence techniques are used for solving specific problems within very specific contexts.

These techniques are often very effective, and depending on the problem domain, can produce more accurate predictions, classifications or solutions than the human brain could. However, they are not “magic bullets” that can be applied to any problem, regardless of its definition.

Models are developed under specific contexts only. They therefore require a precise understanding of all factors influencing a possible solution. Given our limited understanding of the world, there may be factors that we simply cannot (yet) express abstractly. Consider human intuition for example: it might be possible to formulate intuition abstractly for a very narrow set of circumstances (such as a situation involving a very specific threat), but the broader the circumstance becomes, the more difficult it becomes to abstract and formulate it. In other words: AI fails for problems that are too difficult to formalize and abstract.

In a sense, that contradicts the beginning of this tutorial, in which we discussed how the application of AI is suited to areas where formulating a precise set of rules for a problem is difficult. The difficulty in abstraction is largely due to our lack of understanding of many problems (such as the replication of intuition) itself. We actually still only have a very limited understanding of how the human mind works. Emotions, subconsciousness, human psychology and our decision-making process are still being studied and we are far away from arriving at definitive answers to fundamental questions. Therefore, implementing models that mimic or reproduce these processes is difficult. How can we implement something that we do not fully understand? …And that is precisely why we are very far away from developing an “all-powerful singularity”.

Applying AI to solve problems may also fail when we have a domain-specific problem that we can formulate, but either lack the data to make an implementation work, or have the data but in a form which makes it difficult to detect patterns in the data.

Take for example the case of a procurement department at a large organisation: The act of procuring certain goods at certain rates can be precisely defined, and the actions involved could easily be automated. However, the data required for a procurement AI to learn how to make decisions on how to act, may not exist in a structured form. This data (which arises from decisions and discussions made over the years) could be scattered across emails, Word documents, notes from phone calls, recordings, different tender platforms, and so on so forth.

Collecting, categorizing, and structuring (i.e. organizing the information in such a way that it can be used by the AI) these hundreds of thousands of items so that they can be used by the AI might simply not be possible or financially feasible. Artificial intelligence (and especially machine learning) needs to be backed by data (how much data, depends on the problem and the way the problem is to be solved), and a lack of it can hugely limit the application of AI.

Last but not least, even if we can model the problem, and have sufficient data available to support the model and decision-making process, the model itself might fail. Bruce Bueno de Mesquita describes the 3 ways in which models fail very concisely in his book “Predictioneer's Game”:

"Models fail for 3 main reasons: The logic fails to capture what actually goes on in people's heads when they make choices; the information going into models is wrong - garbage in, garbage out; or something outside the frame of reference of the models occurs to alter the situation, throwing it off course."

Conclusion

In this tutorial, we covered a lot of ground. We began by discussing the difficulties around actually defining what intelligence is, arriving at an acceptable middle ground, defining intelligence as: "the ability to learn and use concepts to solve problems". We then used this definition to explain what exactly we mean by artificial intelligence, dispelling some of the myths around it, and essentially reducing artificial intelligence to a wide range of techniques (which draws from a series of different disciplines) that are used to get computers to solve complex, domain-specific problems that tend to be ambiguous and difficult to formulate abstractly.

We discussed the emergence of artificial intelligence, its history and learned that the problems themselves are being solved by trying to create abstract representations (called "models") of the problem domain.

We then listed some examples of real-world problems that are currently being solved by artificial intelligence, and saw how the problems solved by intelligent systems tend to involve i) large amounts of data, ii) uncertainty and changing environments, as well as iii) significant variations in the input data received. Last but not least, we tried to dispel some of the myths surrounding artificial intelligence by discussing its limitations, illustrating that AI is really a problem-solving technique that tends to be applied to domain-specific problems. The development of intelligent systems really depends on our ability to:

1. Understand, formulate and abstract the problem and domain at hand.
2. Gather and structure the available data in such a way that it can support our model.
3. Incorporate the unexpected into our model.

With an understanding of what artificial intelligence is, we can now move on to the core topic of this tutorial series: Machine Learning. Stay tuned!

• • • • • • •

Benjamin Jakobus
Author

Benjamin Jakobus is a senior software engineer based in Rio de Janeiro. He graduated with a BSc in Computer Science from University College Cork and obtained an MSc in Advanced Computing from Imperial College, London. For over 10 years he has worked on a wide range of products across Europe, the United States and Brazil. You can connect with him on LinkedIn.

Technical and Editorial Review
Suprotim Agarwal

THANK YOU

FOR THE *48th* EDITION

@sravi_kiran

@subodhsohoni

@dani_djg

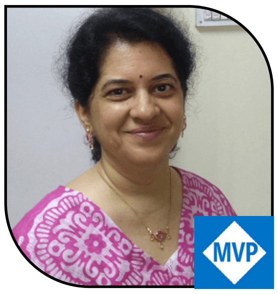
benjamij

@yacoubmassad

dniklovv

@maheshdotnet

@damirarh



@gouri_sohoni

@suprotimagarwal

@saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com