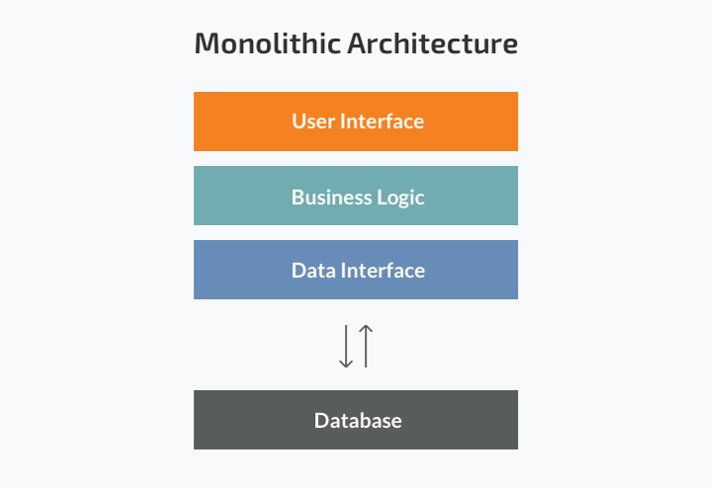
# What have we been doing so far with ASP.NET MVC?

* A monolithic architecture
* Meaning the view(UI) is tightly coupled to the logic and describes application that have everything built into it.
* The backend and frontend are all tightly coupled.

## Cons

* Client will have to wait for its request to be process before showing a proper view.
  + Essentially too many codes dependency of the UI and BL
  + There is a research paper showing how just 3 seconds of blank screen the user would close the website immediately.
* They send html pages across the network is not efficient and will slow down the process heavily



# SOA introduction

* It stands for Service Oriented Architecture
* A styles of software design where services are provided to other components by application components through a communication protocol over the network.
* Meaning is we will decouple between the frontend and the backend of our project.

## Services

* They are responsible for sending and receiving data between your frontend and backend.
* Usually, services are deployed as a Web service, so they are independent of platforms and programming language.

## SOA principles

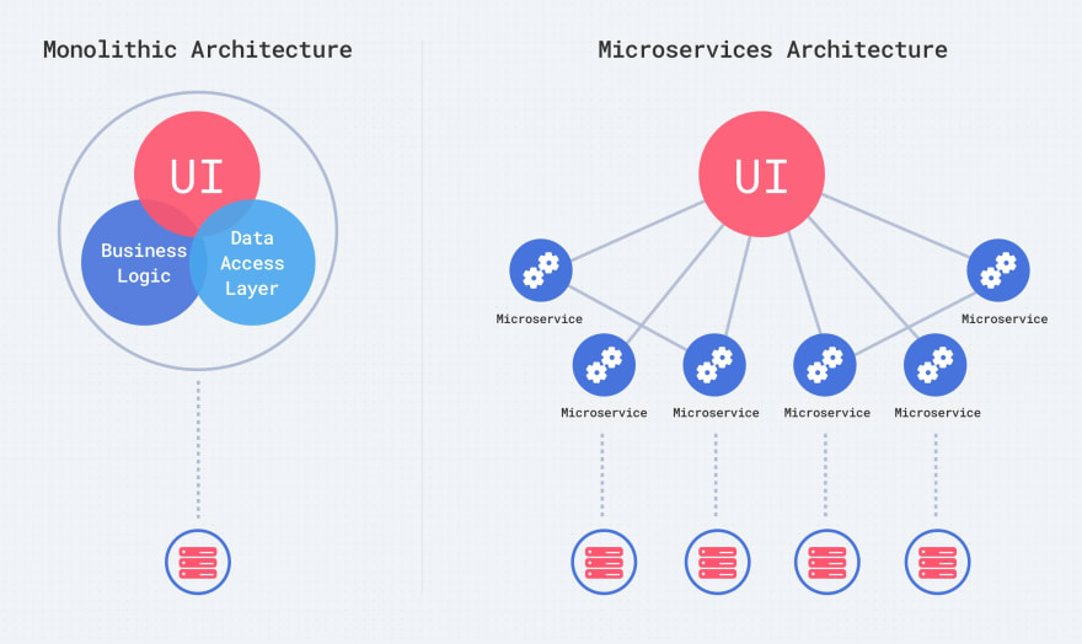
* Standard Service Contract
  + You must have a description on what the service is about. This makes it easier for the client to understand what the service will do for them.
* Loose Coupling
  + Less dependency between the frontend and backend.
  + So, if the service functionality changes at any point in time, it should not break the client application.
* Service Abstraction
  + Services hide the logic they encapsulate from the outside world.
  + It should just tell the client application on what it does and now how it does it
* Service Reusability
  + Logic is divided into services with intent of maximizing the reuse.
  + Just so you don’t have to keep making the same code again and so on.
* Service Autonomy
  + Should have control over the logic they encapsulate.
  + The client application should not have control over the service.
* Service Statelessness
  + Services should be stateless in that the service should not withhold information from one state to the other.

## Pros

* Front end and backend are separate so you can share your API to other people and use the backend portion of your application to their own frontend.
  + Such as google map api to have your own map interface in your front-end application
* It is easier to scale by just adding more servers (horizontal scaling)

## Con

* Is that it will costs a lot.



# Web services

* They are services that are deployed on the web and are available for use by clients or everyone.
* They allow communication between the client’s web and your server applications.
* This allows us to decouple the front end from our backend (unlike our MVC projects so far).
* They are XML markup language that any programming language can understand and allows our web service to be used by different application running on different languages.

# SOAP

* Stands for Simple Object Access Protocol
* It is a messaging protocol specification for exchange structure information in the implantation of web services in computer networks.
* So, another standardize way to communicate between computers.
* It is protocol independent meaning it can sent over HTTP, HTTPS, SMTP, etc.
* Well documented
  + You can easily figure out what endpoints you can sue and what they do and they require and so on.

## WSDL

* It stands for Web Service Definition Language
* XML based file and it tells the client what exactly the web service will do.
* This is essentially what makes SOAP a well-documented SOA.
* Think of it as an instruction manual that you get whenever you buy a product.

### 3 Components of WSDL (ABCs of WSDL)

1. Address
   1. URL address used to connect to the SOAP service
2. Binding
   1. Describes how the service is bound to the SOAP messaging protocol
   2. Essentially, it gives the details require for the client and service to communicate with each other.
3. Contracts
   1. It gives what the input the method takes and what output you should expect.

## Soap Messaging

* Another XML document as a SOAP message

### 4 Components of SOAP Messaging

1. Envelope – Identifies the XML document as a SOAP message.
2. Header – Contains more information on what exactly the service is receiving from the client or what the client is receiving from the server.
3. Body – Contains the response information
4. Fault – Gives error or some status info

# Contract first

* The contract needs to be created first so that means the WSDL needs to be created without writing actual implementation details of the service

# Contract last

* The web service will have actual logic and implantation details and then a contract will be made.

# XML namespace

* Used to avoid naming convention and give a structure to your code (separate of concern)
* Useful for SOAP since it heavily uses XML.
* Also, a way for you to organize your XML file.

# Rest

* Stands for Representational State Transfer.
* Another architectural style to design your services.

## Guiding principles of Rest

* Uniform Interface
  + Your service would have an interface defined by four constraints
    - Identification of resources
      * You should be able to identify the resource you are trying to access.
      * Ex: An endpoint that ends with /SARest/AllStore will give all the store resource stored in the DB.
    - Manipulation of resources through representation
      * Using appropriate action verbs to access/manipulate your resource.
      * Ex: Get method to get all stores or post method to add a store.
    - Self-description message
      * Any other information needed to process the data
      * Ex: the format the data will come in (XML, HTML, JSON, etc.)
    - Hypermedia as the engine of application state (HATEOAS)
      * Not implement as much now a days
      * It will give other potential actions the user can take after doing one action.
      * Ex: The user gets all the store then the http response will include all potential action(s) the user can take.
* Client-Server
  + The client app will evolve separately from the server app without any dependencies on each other.
  + Same concept of what we did with our DL and BL projects.
  + Decoupling the services from each other.
  + Ex: Service A should not affect Service B if Service A gets updated.
* Layered System
  + Constraining the interactions of your components to the ones in the next layer
  + The service should only access the component it is dependent on
  + Ex: Our BL can access our DL while our UI can access our BL but the UI cannot access the DL.
* Stateless
  + Server isn’t responsible for storing the client state.
  + It will just treat every http request as a new or completely different even if it came from the same client or even doing the same http request (refreshing the page)
  + The client is the one responsible for storing its own state (using cookies/cache as an example)
* Cacheable
  + Resource from the serve can be cached if applicable.
  + Meaning the client can store information that was sent by the server.
  + Ex: One you get the list of stores, refreshing the page you instantly get that list of stores since it was stored in your computer.
* Code on demand (optional)
  + Allows the client functionality to be extended by downloading and executing code in a form of applets.

# Richardson Maturity Model

* Describes how Restful your service is
* Level 0
  + Uses HTTP
  + Single URL
  + One method (usually post)
  + The request body will contain the information you need or what client wants
  + Server has to unpack this request
    - Do its inner process
* Level 1
  + Uses HTTP
  + Multiple URL
    - Unique URL for each unique resource
  + Still uses one method (post)
* Level 2
  + Uses HTTP
  + Multiple URL
  + Multiple methods
  + Operations depend on the action method used
* Level 3
  + Uses HTTP
  + Multiple URL
  + Multiple methods
  + HATEOAS