The problem with ASP.NET MVC (or how our application stands now):

With ASP.NET MVC, the view is tightly coupled to the processing logic. This means that the client will have to wait for its request to be processed and also for the view that the server eventually returns. In essence, while we're awaiting for the server to process our request, the view "freezes". A frozen UI is not something that's very user friendly.

Solution:

Decoupling the logic that processes the data (aka your backend logic) from the logic that presents your data (aka your front end logic). Put those into two different servers

Service Oriented Architecture (SOA)

- SOA for short
- Architecture style for building software applications that use services available in a network such as the web
- Services are an implementation of a well defined business functionality
- So what we want to do is to delegate the whole business logic with the data layer associated with it as a service (let's call it the superhero service). This service returns to the client the necessary data it would need to present to the end user, how to present that data is wholly up to the client.
- Note that your services just have to satisfy some general functionality, you can break up your services to be more granular.
    - In fact, another implementation of SOA is MSA (Microservice Architecture). In MSA, each service will have one function and one respective db per service.
    - Thus if applied to the Heros App, there'd be one service per entity, so: SuperHero Service, SuperVillain Service, and a SuperPower Service

REST

- Representational State Transfer
- Representational State Transfer refers to transferring "representations". You are using a "representation" of a resource to transfer resource state which lives on the server into application state on the client.
- Architectural style to design your services
- If you want your service to be RESTful you must follow these constraints:
    - Uniform Interface
    - Client Server
    - Stateless
    - Layered System
    - Cacheable

- Code on demand

Uniform Interface

- Your service would have an interface defined by four interface constraints:
  - Identification of resources
    - You'll be able to identify what resource you're trying to access
    - Ex: Going to the endpoint that ends with /SuperHero means that I'm trying to access/manipulate a superhero resource
  - Manipulation of resources through representations
    - Using the appropriate action verb to access and manipulate your resources
    - Ex: Using the get method to get all superheros, using the post method to add a superhero
  - Self-descriptive messages
    - Including in the response any necessary information to process the data, such as the format of the data
    - If for some reason, you separate the data your sending from information on how to process it, the client would be confused on how to process the data it receives if the messages arrive out of order or if a third response arrives in between the two messages
  - Hypermedia as the engine of application state
    - HATEOAS
    - Not really implemented much
    - Basically what this means is that whenever you access a resource, additional actions you can take based on what you accessed would be returned as part of the response
    - Ex: If you get all the heroes, part of the response would be the endpoints to edit or delete a certain hero based on some identifier

Client Server

- The client app must evolve separately from the server app without any dependency on each other
- Clients should only need to know the resource URIs
- Decoupling the services from each other
- As long as the interface (endpoints) remains the same, they could be developed separately

- Your services should be independent from each other even if they call upon each other

Layered System

- Constraining components to see beyond immediate component
- What this means is that a service should only be access components it is dependent on
- This is honestly just applying the layered architecture but on a service level
- Ex: Service A calls on Service B, Service B calls on Service C, Service A can access Service B, but not Service C, even though Service B is “dependent” on Service C
- Ex: You have an Authorization Service, a subsystem of BL Services, and another subsystem of DL Services. The layers would be UI>Auth & BL>DL, UI components can interact with the authorization service, after being authorized, it should be able to call some of the BL components, but it can never access the DL services

Stateless

- Server isn’t responsible for storing client state
- The server will not store anything about the latest HTTP request the client made.
- It will treat every request as new.
- No session, no history.
- The client should be responsible for storing its own state

Cacheable

- Resources from the server can be cached if applicable, these resources themselves must declare themselves cacheable
- Caching occurs when the client stores previous responses it received from the server, so that when that data is needed again, it can save a round trip over the network by using the cached data.
- The ability to cache is made possible by the interface constraint of “self-descriptive messages”, since the client knows that all the relevant data about a single resource is being sent in one response. It doesn’t have to worry about accidentally only caching part of the information it needs, and missing other parts.
- Some examples of cacheable requests:
    - Get requests
- Non cacheable requests:
    - Delete requests

Code on demand

- Often forgotten since its optional

- Allows client functionality to be extended by downloading and executing code in the form of applets or scripts

Richardson Maturity Model

- Describes how RESTful your service is

Level 0

- Uses HTTP
- Single URI
- One method (usually post)
- The request body will contain the information on what operation the client wants performed as well as the data required in performing that data
- Server has to unpack this request body and parse it to call the appropriate method and pass the appropriate information
- Called the POX swamp because the request body traditionally uses XML

Level 1

- Uses HTTP, Multiple URIs, still one method (POST)
- Unique URI for each unique resource

Level 2

- Uses HTTP, multiple URIs, various http methods
- Operations depend on the action method used

Level 3

- Uses HTTP, multiple URIs, various http methods, and HATEOAS

Resources:

SOA

REST

Extra REST

Richardson Maturity Model 1

Richardson Maturity Model 2

Richardson Maturity Model 3

Richardson Maturity Model 4

MSA for some extra reading

What representational state transfer means