***O QUE É MVC ?***

MVC é um padrão de projeto criado pelo Xerox na década de 70/80, que é dividido em três camadas interconectadas;

***CARACTERÍSTICAS DO MVC***

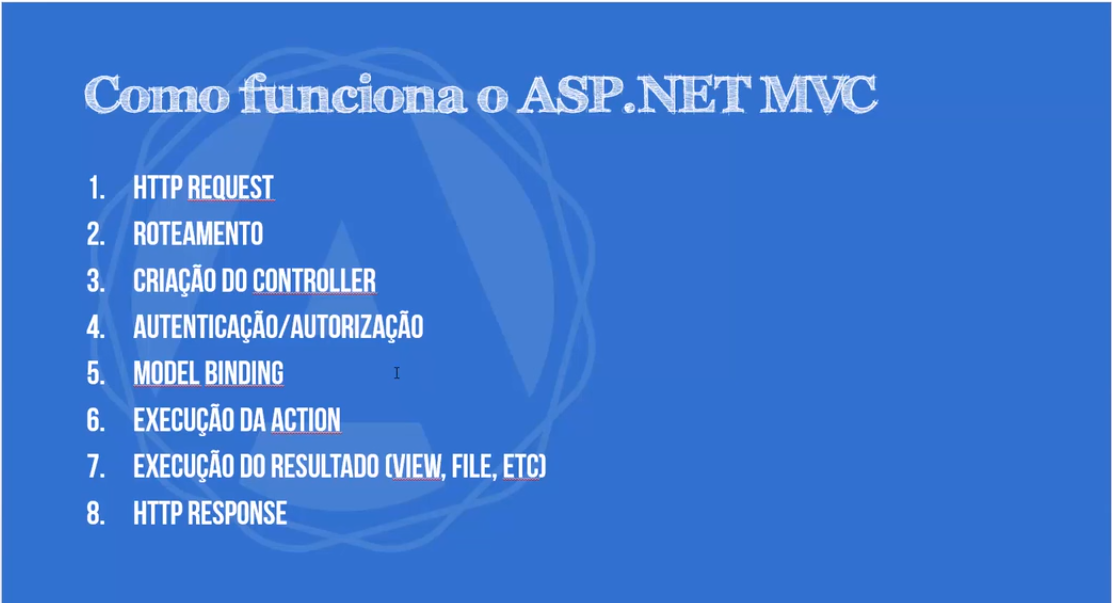
* Auto desacoplamento => as camadas funcionam independentes, podemos criar a Model, sem conhecer a View e a Controller, ou vice versa, facilitando os testes unitários;
* Maior organização do projeto;
* Altamente testável, no padrão de projeto do Web Forms os testes eram quase impossíveis de serem realizados, pois os métodos eram amarrados entre código e HTML;
* Divisão de responsabilidades é muito boa, cada camada tem a sua responsabilidade no projeto;

***ASP.NET MVC – IMPLEMENTAÇÃO***

Esse tipo de projeto implementa o padrão MVC sobre o ASP.NET, usando toda a base do ASP.NET.

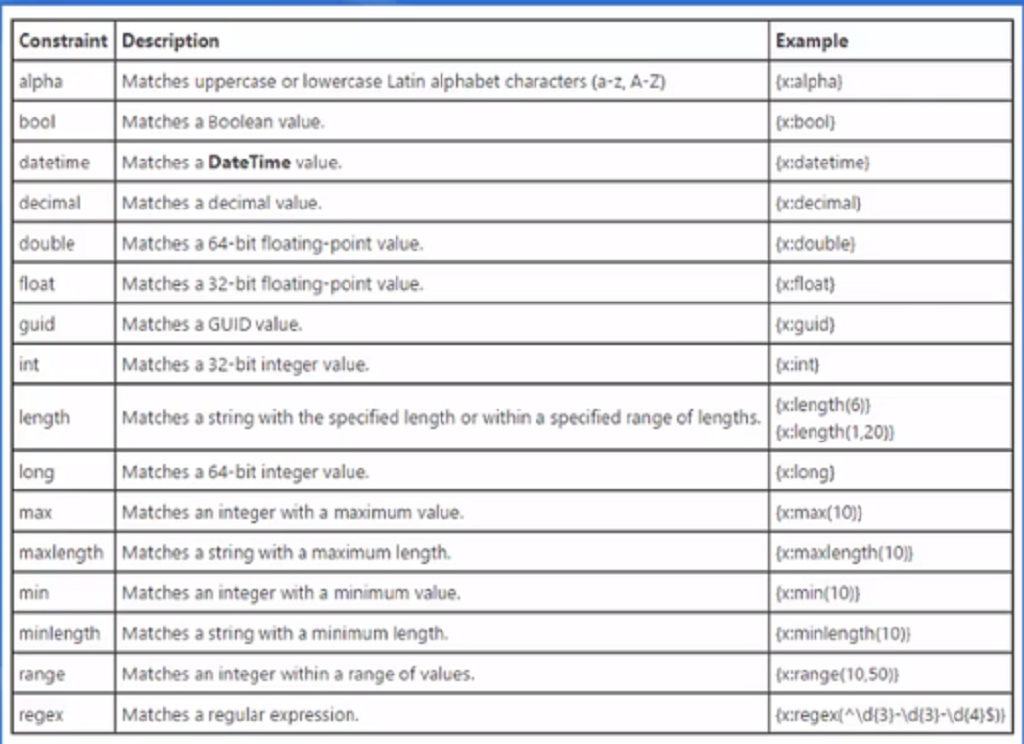
* Model => Classes do C# que refletem a nossa base de dados. EX: Produto.cs, Cliente.cs, etc.
* View => São os arquivos CSHTML(Padrão RAZOR);
* Controller => Classes que herdam da classe Controller onde contém:
  + Action => Um método que executa algum código e retorna uma view;
  + Route => rotas para alcançar um controller/action via URL;

***Como Funciona o ASP.NET MVC***



***Detalhes Importantes:*** Esse passos determinam como o pipeline do **MVC** funciona no **ASP.NET**, a primeira coisa que o **MVC** vai fazer é segmentar a URL, vai ver qual a action que está sendo chamado, qual o controller que ele precisa criar para executar esta **action**, depois ele passa para **autenticação/autorização**, se o usuário não tiver autenticado ou autorizado ele interrompe o processo, seguindo a ordem ele faz o **model** **binding (de/para)**, ou seja, pegar os valores que vieram na requisição e atribuir aos seus objetos e ou parâmetros do seu método, daí então ele faz a execução da **ACTION**, note que se quisermos que alguma validação seja feita antes de concluir a execução da **ACTION**, temos que criar algo(validação) entre os passos 5 e 6. Após a execução no passo 7 ele executa o resultado da **ACTION**, que pode retornar uma **VIEW**, um arquivo ou só realizar um processo e não retornar nada, e retorna isso via o **HTTP RESPONSE**;

***Mapeamento de Rotas, tipos e valores aceitos para rotas personalizadas***



***EF – Entity Framework – Noções básicas***

* DATA CONTEXT
  + É a representação do banco de dados em memória, podemos adicionar, atualizar, e excluir objetos em memória, e persistir somente quando chamarmos o método SaveChanges(). Em uma aplicação podem existir vários Data Contexts.
* CRUD
  + Sigla para **C**reate, **R**ead, **U**pdate, **D**elete;
  + Estas são as quatro operações básicas de um banco de dados e que na maioria das vezes são as operações da nossa aplicação. Ex.: **Criar** um Cliente, **Ler** os dados de um Cliente, **Atualizar** as informações de um cliente, e **Deletar** um cliente;
* MIGRATION
  + É a forma como abstraímos o banco de dados, o migration se encarrega de atualizá-lo sempre que o nosso modelo (Domínio) mudar.

***DESIGN PATTERNS – Repository Pattern***

São padrões de desenvolvimento de software, prega por boas práticas na escrita e arquitetura de softwares, é importante saber quando usar, do que como utilizar;

* ***Repository Pattern*** 
  + ***Particularmente eu acho ele bem interessante, por separar bem as responsabilidades;***
  + Padrão para desacoplamento do acesso à dados da aplicação
  + Cria uma unidade de acesso à dados para cada entidade modelo
  + Cada repositor é responsável pela leitura e persistência das informações referentes a entidade
  + Caso precise alterar alguma fonte de dados um dia, sua aplicação continua a mesma;

***INTERFACE – CONTRATO***

É uma espécie de contrato que a classe que o implementa precisa seguir, define quais métodos e propriedades que um objeto deve expor. Uma interface pode ter várias implementações e o ideal é depender da interface e não da implementação.

Outro detalhe interessante é que como interfaces são contratos e contratos podem ter várias implementações, então podemos criar uma implementação “falsa” para realizar algum teste, ou uma implementação diferenciada para cada situação. Ex.: Pessoa Física/Jurídica.

***INJEÇÃO DE DEPENDÊNCIA***

Analisamos o cenário hipotético, temos uma classe X que depende de uma classe Y para funcionar, essa classe Y depende de uma classe W, e assim mais algumas dependências, imagine agora quantas classes teríamos de instanciar para ter uma instância de X ? Como resolver estas dependências? É aí que entra a injeção de dependência, ou Dependency Injection (DI).

Primeiro, sua classe deve sempre depender de uma interface (contrato), da qual a sua dependência possui. Exemplificando, se temos um repositório de Autor, e a controller depende dele, o ideal é a controller depender da interface da qual o repositório implementa, porque toda vez que chamarmos o controller Autor, falamos pra ele instanciar a implementação X da interface injetada, com isso não precisamos saber do que o repositório depende, e resolve as dependências em cascata.

* ***UNITY***
  + Container nativo da Microsoft que uma vez inicializado, resolve as dependências da sua aplicação;

***Repositórios nos Controllers:*** a regra é a mesma, sempre depender da interface e não da implementação, com isso temos os controllers desacoplados da fonte de dados;

***Views Engines***

As View Engines são mecanismos para interagir com funções c# no HTML, elas fornecem recursos extras ao html. A engine mais famosa no ASP.NET é o Razor, esses recursos adicionais tem caraterísticas próprias e são identificados por partes diferenciadas no código html.



***RAZOR VIEW ENGINE***

* É o mecanismo mais usado no desenvolvimento ASP.NET MVC
* Incorpora muitas funções C#, tipo: IF, While, For, Foreach
* Podemos criar funções próprias
* Podemos criar nossos próprios componentes(HELPERS)

***HELPERS - Utilitários***

* São utilitários para construção de componentes do html
* Depois de compilado, vira um código html mesmo
* Podem ser criados sem referência de propriedade.
  + Ex.: @HTML.DISPLAY(“NAME”)
* Ou pode ser referenciando uma propriedade.
  + Ex.: @HTML.DISPLAYFOR(x => x.Name)
* Principais Helpers
  + Formulários:
    - @HTML.BEGINFORM: inicia um formulário na tela;
  + Componentes
    - EDITOR: input type text;
    - TEXTAREA: input com entrada com quebra de linha;
    - LABEL: para definir títulos de editor, etc.;
    - DISPLAY: para jogar qualquer informação na tela;
    - CHECKBOX: caixa para marcar e desmarcar opções;
    - DROPDOWLIST: o “famoso” combobox na web, um campo seletor de opções;

***Partial Views – Pedaços***

São pedaços de uma view que podem ser retornadas do controller ou podem ser chamadas a partir de uma outra view ou até mesmo partial view.