WEB PROGRAMMING ASP.NET MVC CORE

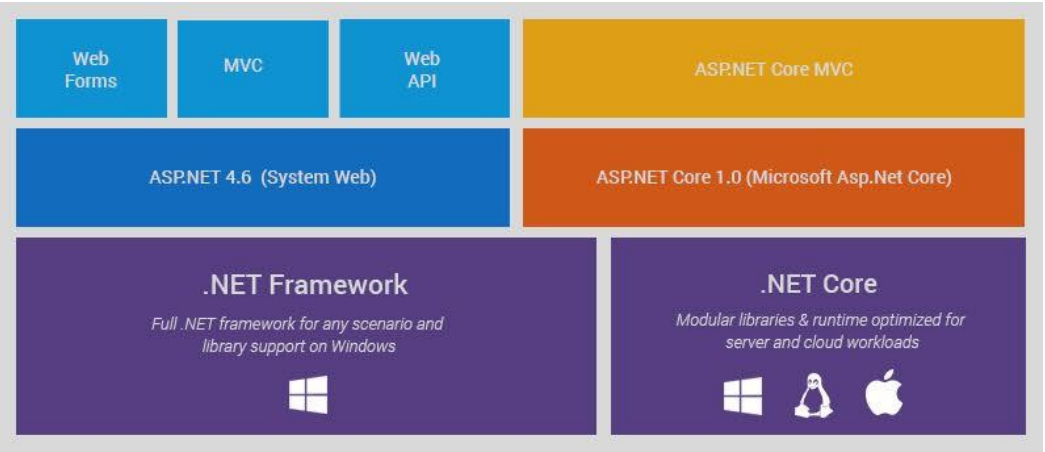
© 2017-2020, NOEL LOPES

ASP.NET Core MVC is a web application development framework from Microsoft that combines the effectiveness and tidiness of model-view-controller (MVC) architecture, ideas and techniques from agile development, and the best parts of the .NET platform.

It emphasizes clean architecture, design patterns, and testability, and it doesn’t try to conceal how the Web works.

ASP.NET CORE MVC

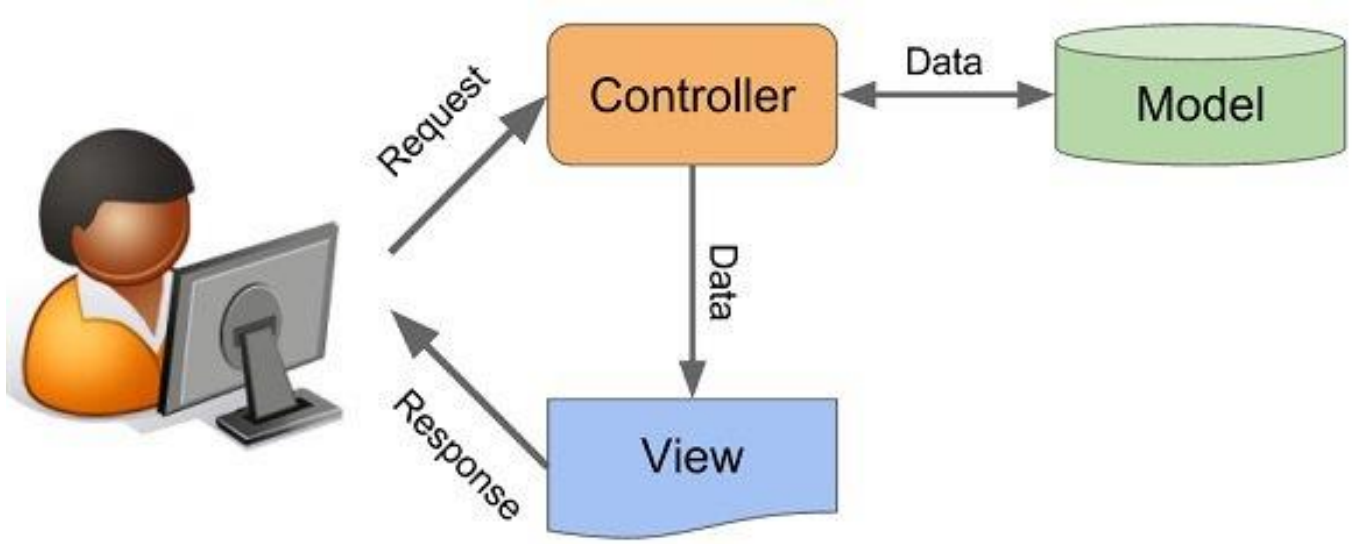
ASP.NET CORE MVC



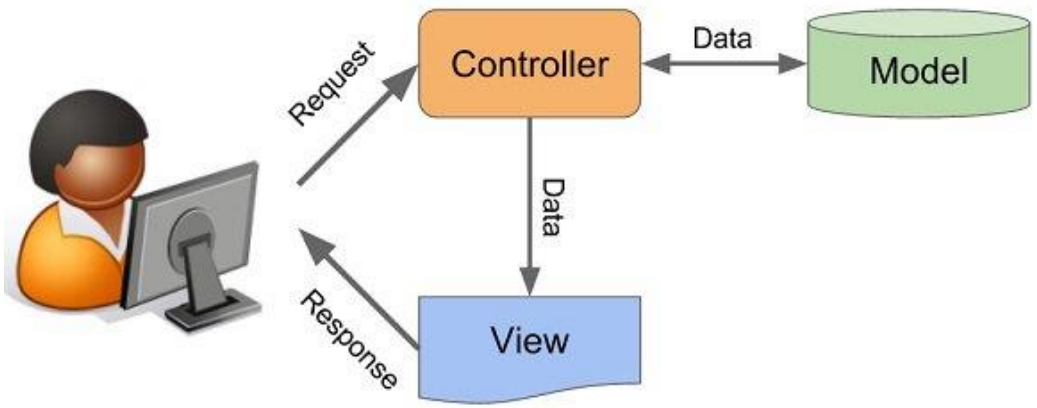
- ASP.NET Core is built on .NET Core, which is a cross-platform version of the .NET Framework without Windows-specific APIs.
- Web applications are increasingly hosted in small and simple containers in cloud platforms, and by embracing a cross-platform approach Microsoft extended the reach of .NET, making possible the deployment of ASP.NET Core applications to a broader set of hosting environments, and, as a bonus, made it possible for developers to create ASP.NET Core web applications on Linux and OS X.

MVC PATTERN

- ASP.NET Core MVC follows a pattern called model-view-controller (MVC), which guides the shape of an ASP.NET web application and the interactions between the components it contains.

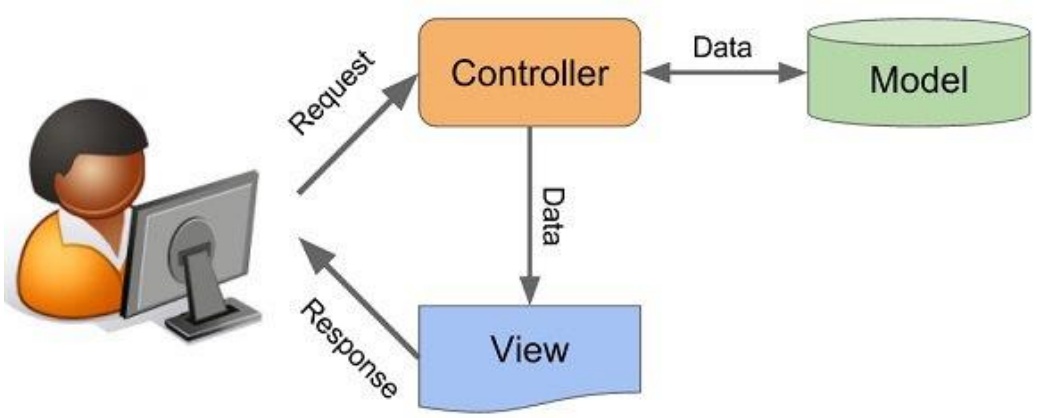


MVC PATTERN



- The MVC pattern dates back to 1978 and the Smalltalk project at Xerox PARC. But it has gained popularity recently as a pattern for web applications, for the following reasons:
  - User interaction with an application that adheres to the MVC pattern follows a natural cycle: the user takes an action, and in response the application changes its data model and delivers an updated view to the user. And then the cycle repeats.
  - This is a convenient fit for web applications delivered as a series of HTTP requests and responses.
  - Web applications necessitate combining several technologies (databases, HTML, and executable code, for example), usually split into a set of tiers or layers. The patterns that arise from these combinations map naturally onto the concepts in the MVC pattern.

MVC PATTERN



- The MVC architectural pattern helps to achieve separation of concerns, by separating an application into three main groups of components: Models, Views, and Controllers.
- Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries.
- The Controller chooses the View to display to the user, and provides it with any Model data it requires.

MVC PATTERN

01

The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

02

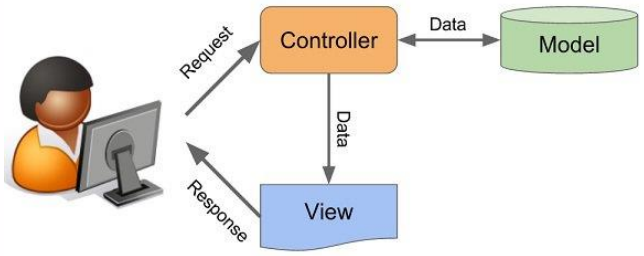
The pattern specifies where each kind of logic should be located in the app:

- The UI logic belongs in the view.
- Input logic belongs in the controller.
- Business logic belongs in the model.

03

This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another.

MODEL RESPONSIBILITIES



The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it.

Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application.

Strongly-typed views will typically use ViewModel types specifically designed to contain the data to display on that view; the controller will create and populate these ViewModel instances from the model.

VIEWMODEL

- In an MVC web application, a ViewModel is a type that includes just the data a View requires for display (and perhaps for sending back to the server).
- ViewModel types can also simplify model binding in ASP.NET MVC. ViewModel types are generally just data containers; any logic they may have should be specific to helping the View render data.

Views are responsible for presenting content through the user interface.

They use the Razor view engine to embed .NET code in HTML markup.

There should be minimal logic within views, and any logic in them should relate to presenting content.

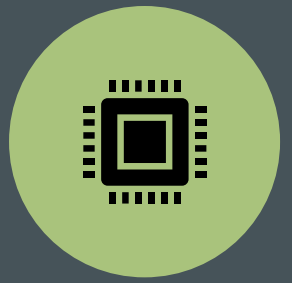
A view template should never perform business logic or interact with a database directly. Instead, it should work with the data provided by the controller.

VIEW RESPONSIBILITIES

Controllers handle user interaction, work with the model, and ultimately select a view to render.

Controllers are responsible for providing whatever data or objects are required for a view template to render a response to the browser.

In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values and passes these values to the model. The model might use these values to query the database.



The controller is the initial entry point and is responsible for selecting which model types to work with and which view to render (hence its name – it controls how the app responds to a given request).

CONTROLLER RESPONSIBILITIES

CONTROLLERS

- Controllers should not be overly complicated by too many responsibilities. To keep controller logic from becoming overly complex, use the Single Responsibility Principle to push business logic out of the controller and into the domain model.
- If you find that your controller actions frequently perform the same kinds of actions, you can follow the Don't Repeat Yourself principle by moving these common actions into filters.

MVC WEB FRAMEWORK

Model

Data Access Layer (e.g. using a tool like Entity Framework or Nhibernate)

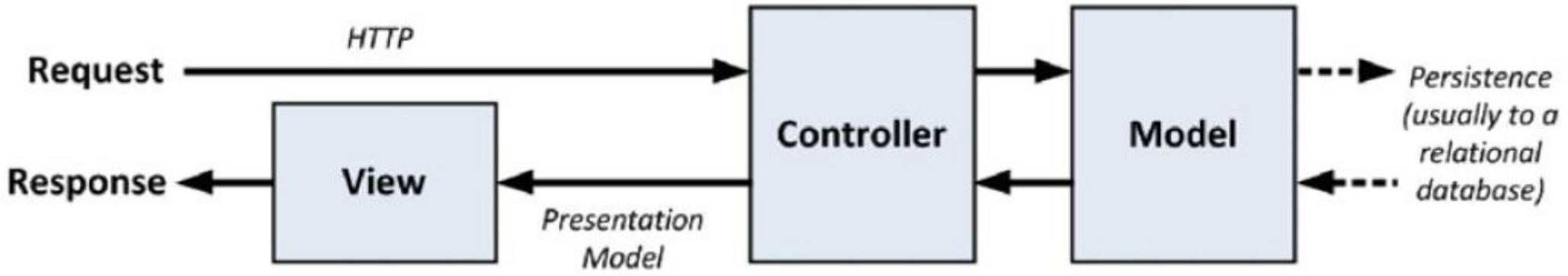
View

This is a template to dynamically generate HTML.

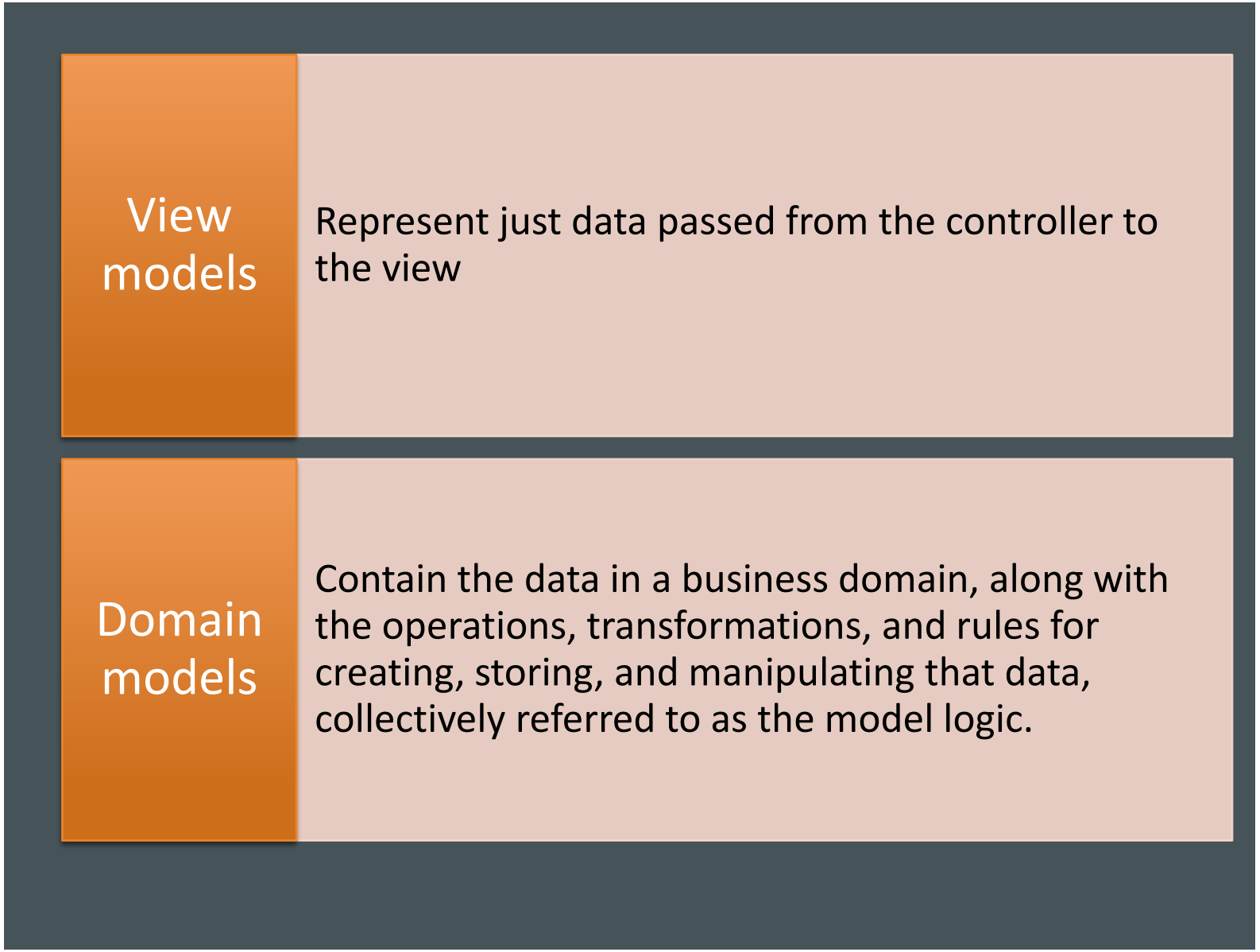
Controller

Manages the relationship between the View and the Model. It responds to user input, talks to the Model, and decides which view to render

THE ASP.NET IMPLEMENTATION OF MVC



MODELS


View models
Represent just data passed from the controller to the view
Domain models
Contain the data in a business domain, along with the operations, transformations, and rules for creating, storing, and manipulating that data, collectively referred to as the model logic.

DOMAIN MODELS

A model should	A model should not
• Contain the domain data • Contain the logic for creating, managing, and modifying the domain data • Provide a clean API that exposes the model data and operations on it	• Expose details of how the model data is obtained or managed (in other words, details of the data storage mechanism should not be exposed to controllers and views) • Contain logic that transforms the model based on user interaction (because that is the controller's job) • Contain logic for displaying data to the user (that is the view's job)

CONTROLLERS

A controller should

- Contain the actions required to update the model based on user interaction

The controller should not

- Contain logic that manages the appearance of data (that is the job of the view)
- Contain logic that manages the persistence of data (that is the job of the model)

VIEWS

Views should

- Contain the logic and markup required to present data to the user

Views should not

- Contain complex logic (this is better placed in a controller)
- Contain logic that creates, stores, or manipulates the domain model

1

Razor is the view engine responsible for incorporating data into HTML documents.

2

Razor provides features that make it easy to work with the rest of the ASP.NET Core MVC using C# statements.

3

Razor expressions are added to static HTML in view files. The expressions are evaluated to generate responses to client requests.

RAZOR