Kashyap Sayani

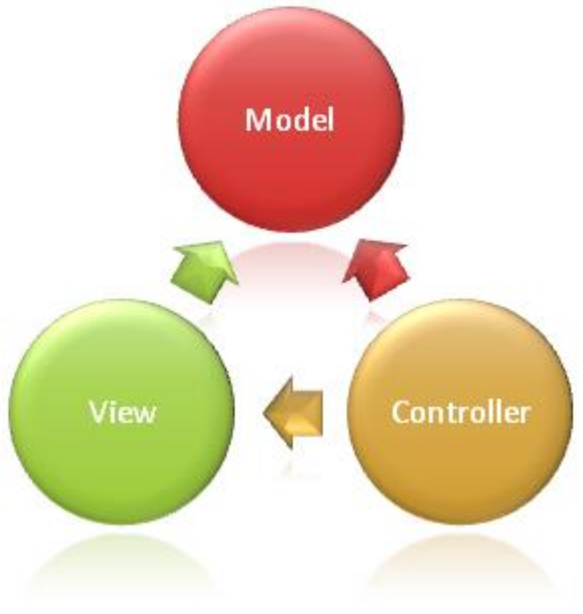
Module – 5

- **MVC**

MVC pattern

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. 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.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something (model, view, or controller) that has a single job. It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, an object containing business logic must be modified every time the user interface is changed. This often introduces errors and requires the retesting of business logic after every minimal user interface change.

Note

Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

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 typically use ViewModel types designed to contain the data to display on that view. The controller creates and populates these ViewModel instances from the model.

View Responsibilities

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. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a View Component, ViewModel, or view template to simplify the view.

Controller Responsibilities

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, 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).

• Routing

ASP.NET Core MVC is built on top of ASP.NET Core's routing, a powerful URL-mapping component that lets you build applications that have comprehensible and searchable URLs. This enables you to define your application's URL naming patterns that work well for search engine optimization (SEO) and for link generation, without regard for how the files on your web server are organized. You can define your routes using a convenient route template syntax that supports route value constraints, defaults and optional values.

Convention-based routing enables you to globally define the URL formats that your application accepts and how each of those formats maps to a specific action method on a given controller. When an incoming request is received, the routing engine parses the URL and matches it to one of the defined URL formats, and then calls the associated controller's action method.

C#Copy

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

Attribute routing enables you to specify routing information by decorating your controllers and actions with attributes that define your application's routes. This means that your route definitions are placed next to the controller and action with which they're associated.

C#Copy

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
   [HttpGet("{id}")]
   public IActionResult GetProduct(int id)
   {
     ...
   }
}
```

• Rest Web API

In addition to being a great platform for building web sites, ASP.NET Core MVC has great support for building Web APIs. You can build services that reach a broad range of clients including browsers and mobile devices.

The framework includes support for HTTP content-negotiation with built-in support to format data as JSON or XML. Write custom formatters to add support for your own formats.

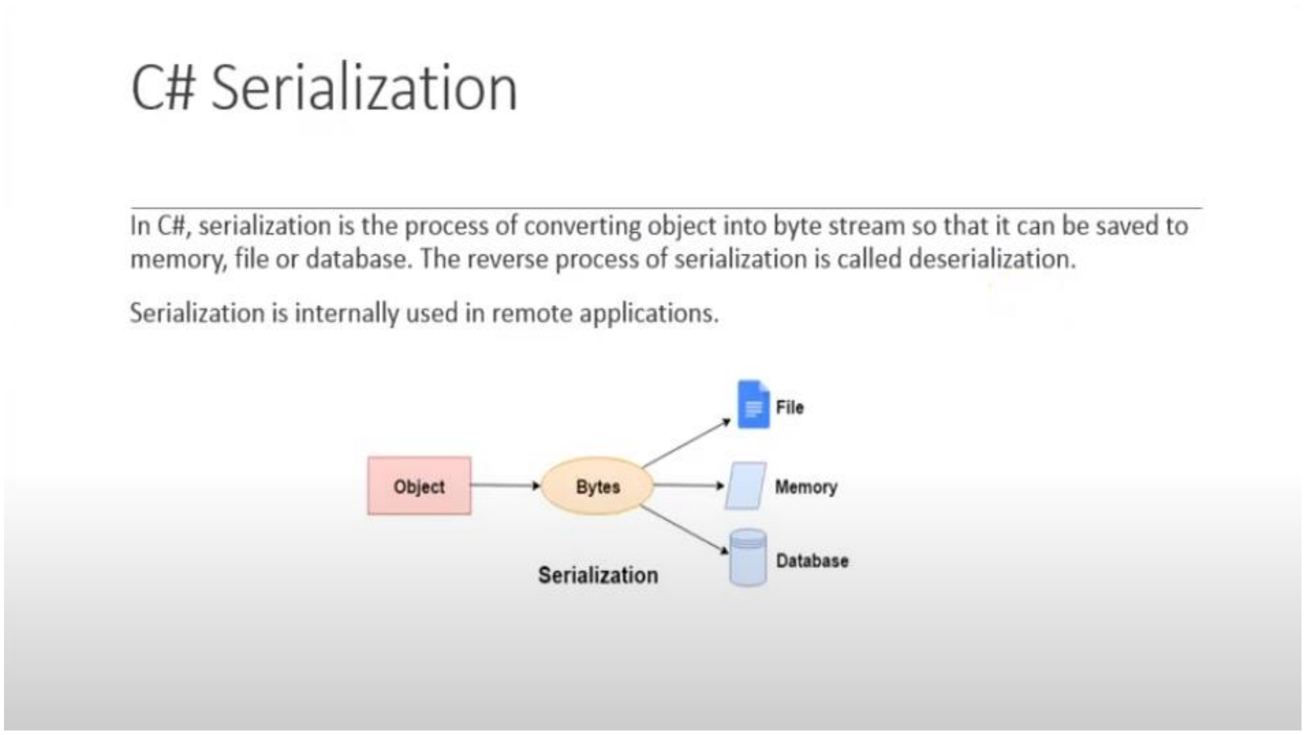
Use link generation to enable support for hypermedia. Easily enable support for cross-origin resource sharing (CORS) so that your Web APIs can be shared across multiple Web applications.

Methods

The four main HTTP methods (GET, PUT, POST, and DELETE) can be mapped to CRUD operations as follows:

- GET retrieves the representation of the resource at a specified URI. GET should have no side effects on the server.
- PUT updates a resource at a specified URI. PUT can also be used to create a new resource at a specified URI, if the server allows clients to specify new URIs. For this tutorial, the API will not support creation through PUT.
- POST creates a new resource. The server assigns the URI for the new object and returns this URI as part of the response message.
- DELETE deletes a resource at a specified URI.

- **Serialization**



C# Serialization

In C#, serialization is the process of converting object into byte stream so that it can be saved to memory, file or database. The reverse process of serialization is called deserialization.

Serialization is internally used in remote applications.

- **Parameters**

Using [FromUri]

To force Web API to read a complex type from the URI, add the **[FromUri]** attribute to the parameter. The following example defines a GeoPoint type, along with a controller method that gets the GeoPoint from the URI.

C#Copy

```
public class GeoPoint
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}

public ValuesController : ApiController
{
    public HttpResponseMessage Get([FromUri] GeoPoint location) { ... }
}
```

The client can put the Latitude and Longitude values in the query string and Web API will use them to construct a GeoPoint. For example:

```
http://localhost/api/values/?Latitude=47.678558&Longitude=-122.130989
```

Using [FromBody]

To force Web API to read a simple type from the request body, add the **[FromBody]** attribute to the parameter:

C#Copy

```
public HttpResponseMessage Post([FromBody] string name) { ... }
```

In this example, Web API will use a media-type formatter to read the value of *name* from the request body. Here is an example client request.

ConsoleCopy

```
POST http://localhost:5076/api/values HTTP/1.1
User-Agent: Fiddler
Host: localhost:5076
Content-Type: application/json
Content-Length: 7

"Alice"
```

When a parameter has [FromBody], Web API uses the Content-Type header to select a formatter. In this example, the content type is "application/json" and the request body is a raw JSON string (not a JSON object).

At most one parameter is allowed to read from the message body. So this will not work:

C#Copy

```csharp
// Caution: Will not work!
public HttpResponseMessage Post([FromBody] int id, [FromBody] string name) { ... }
```

The reason for this rule is that the request body might be stored in a non-buffered stream that can only be read once.