WEB PROGRAMMING ASP.NET MVC CORE

MODELS

- Models represent the domain the application focuses on. They describe the data as well as the business rules for how the data can be changed and manipulated.
- *The model* is the most important part of the application. It is a representation of the real-world objects, processes, and rules that define the subject, known as the *domain*, of the application. The model, often referred to as a *domain model,* contains the C# objects (known as *domain objects*) that make up the universe of the application and the methods that manipulate them.
- The views and controllers expose the domain to the clients in a consistent manner, and a well-designed MVC application starts with a well-designed model, which is then the focal point as controllers and views are added.
- The MVC convention is that the classes that make up a model are placed inside a folder called the Models.

```
namespace PartyInvites.Models {
    public class GuestResponse {
        public string Name { get; set; }
        public string Phone { get; set; }
        public string Email { get; set; }
        public bool? WillAttend { get; set; }
    }
}
```

MODELS

```
namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        // ...

        public ViewResult Register() {
            return View();
        }
    }
}
```

REGISTER ACTION

STRONGLY TYPED VIEW

A strongly typed view is intended to render a specific model type

```
@model PartyInvites.Models.GuestResponse

@{
    ViewData["Title"] = "Register";
}

<h2>Are you comming to the party?</h2>
```

```
@{
    ViewData["Title"] = "Home";
}

<h2>Welcome to the party</h2>

<p>We are going to throw an exciting party</p>

<p>You are here because we sent you an invite</p>

<a asp-action="Register">
Are you planning to come to the party?</a>
```

LINKING ACTION METHODS

LINKING ACTION METHODS

```
@{
    ViewData["Title"] = "Home";
}

<h2>Welcome to the party</h2>

<p>We going to throw an exciting party</p>

<p>You are here because we sent you an invite</p>

<a asp-action="Resgister">
Are you planning to come to the party?</a>
```

- The attribute `asp-action` is a *tag helper* attribute, which is an instruction for Razor that will be performed when the view is rendered.
- An `href` attribute to the `<a>` element that contains a URL for an action method will be rendered.
- There is an important principle at work here, which is that you should use the features provided by MVC to generate URLs, rather than hard-code them into your views. When the tag helper created the `href` attribute for the `<a>` element, it inspected the configuration of the application to figure out what the URL should be. This allows the configuration of the application to be changed to support different URL formats without needing to update any views.

CREATING A FORM VIEW

The `asp-action` attribute uses the application's URL routing configuration to set the action attribute to a URL that will target a specific action method.

```
<form asp-action="Register" method="post">
    <!-- ... -->
</form>
```

CREATING A FORM VIEW

Each element is associated with the model property using the `asp-for` attribute, which is another tag helper attribute.

The `asp-for` attribute on the label element sets the value of the for attribute.

The `asp-for` attribute on the input element sets the id and name elements.

```
<form asp-action="Register" method="post">
    <div class="form-group">
        <label asp-for="Name">Name:</label>
        <input asp-for="Name" class="form-control“
           placeholder="Enter your name" />
    </div>

    <!-- ... -->
</form>
```

CREATING A FORM VIEW

```
<form asp-action="Register" method="post">
    <!-- ... -->

    <div class="form-group">
        <label asp-for="WillAttend">
            Are you comming to the party?</label>

        <select asp-for="WillAttend" class="form-control">
            <option value="">
                I don't know yet
            </option>
            <option value="true">
                Yes, I will go to the party
            </option>
            <option value="false">
                No, sorry can't make it
            </option>
        </select>
    </div>

    <!-- ... -->
</form>
```

CREATING A FORM VIEW

```
<form asp-action="Register" method="post">
    <!-- ... -->

    <div class="mt-4">
       <button type="submit"
           class="btn btn-primary">Submit</button>

       <a asp-action="Index"
          class="btn btn-secondary">Cancel</a>
    </div>
</form>
```

GET AND POST REQUESTS

Web applications generally use GET requests for reads and POST requests for writes (which typically include updates, creates, and deletes).

A GET request represents an independent read-only operation. You can send a GET request to a server repeatedly with no ill effects, because a GET should not change state on the server. Moreover, you can bookmark the GET request because all the parameters are in the URL (thus the form input values are preserved). A GET request is what a browser issues normally each time someone clicks a link.

Performing a create, delete or edit operation in response to a GET request (or for that matter, any other operation that changes data) opens up a security hole.

A POST request generally modifies state on the server and repeating the request might produce undesirable effects (*e.g.* double billing).

```
public class HomeController : Controller {

    // ...

    [HttpGet]
    public ViewResult Register() {
        return View();
    }

    [HttpPost]
    public ViewResult Registe(GuestResponse response) {
        //TODO: Store guest response
        return View("ThankYou");
    }

    // ...
}
```

GET VS POST

MODEL BINDING

- *Model binding* is a useful MVC feature whereby incoming data is parsed and the key/value pairs in the HTTP request are used to populate properties of domain model types. It eliminates the grind and toil of dealing with HTTP requests directly and lets you work with C# objects rather than dealing with individual data values sent by the browser.
- Model binding free us from the tedious and error-prone task of having to inspect an HTTP request and extract all the data values that are required.

ACADEMIC EXAMPLE

For now we will store data in an in-memory collection of objects. This isn't useful in a real application because the response data will be lost when the application is stopped or restarted

```
// NEVER DO THIS !!!
// This is only for demonstration/academic purposes

public class Repository {
    private static List<GuestResponse> responses =
        new List<GuestResponse>();

    public static IEnumerable<GuestResponse> Responses =>
        responses;

    public static void AddResponse(GuestResponse response) =>
        responses.Add(response);
}
```

```
namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        // ...

        [HttpPost]
        public ViewResult Rsvp(GuestResponse response) {
            Repository.AddResponse(response);

            return View("Thanks", response);
        }

        // ...
    }
}
```

REGISTER POST ACTION

The Thanks.cshtml view uses Razor to display content based on the value of the GuestResponse properties that I passed to the View method in the Register action method.

The Razor @model expression specifies the domain model type with which the view is strongly typed.

To access the value of a property in the domain object, use Model.PropertyName. For example, to get the value of the Name property, call Model.Name.

```
@model PartyInvites.Models.GuestResponse

@{
    ViewData["Title"] = "Thanks";
}

<h2>Thank you, @Model.Name !!!</h2>

@if (Model.WillAttend == null) {
    @: Thank you for your answer. When you decide if you
can come to the party, give me a call.
} else if (Model.WillAttend == true) {
    @: Thank you for your answer. I will prepare you a
special drink.
} else { // Model.WillAttend == false
    @: Sorry to hear that you can't make it, but tanks
for letting us know.
}
```

```
namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        // ...

        public ViewResult GuestList() {
            return View(Repository.Responses);
        }

        // ...
    }
}
```

GUESTLIST ACTION

```
@model IEnumerable<PartyInvites.Models.GuestResponse>

@{
    ViewData["Title"] = "GuestList";
}

<h2>Guest List</h2>

<table class="table">
    <thead>
        <tr>
            <th>Name</th>
            <th>Will Attend</th>
        </tr>
    </thead>
    <tbody>
        <!-- ... -->
    </tbody>
</table>
```

GUESTLIST VIEW

```
foreach (var g in Model) {
    <tr>
        <td>@g.Name</td>
        <td>
            @if (g.WillAttend == true) {
                @: Yes
            } else if (g.WillAttend == false) {
                @: No
            } else {
                @: Don’t know
            }
        </td>
    </tr>
}
```

GUESTLIST VIEW

GUEST LIST ACTION

```
public IActionResult GuestList() {
    var guestList = Repository.Responses;

    if (guestList.Count() == 0) {
        return View("RegisterFirst");
    } else {
        return View(guestList);
    }
}
```

REGISTERFIRST VIEW

```
@{
 ViewData["Title"] = "Be the first to register";
}

<h1>@ViewBag.Title</h1>

<p>Be the first to register and win a special prize.</p>

<div>
    <a asp-action="Register" class="btn btn-primary">Register</a>
    <a asp-action="Index" class="btn btn-secondary">Home</a>
</div>
```

GUESTLIST CONTROLLER

```
namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        // ...

        public IActionResult PeopleCommingToParty() {
            return View(Repository.Responses.Where(r => r.WillAttend == true);
        }

        // ...
    }
}
```