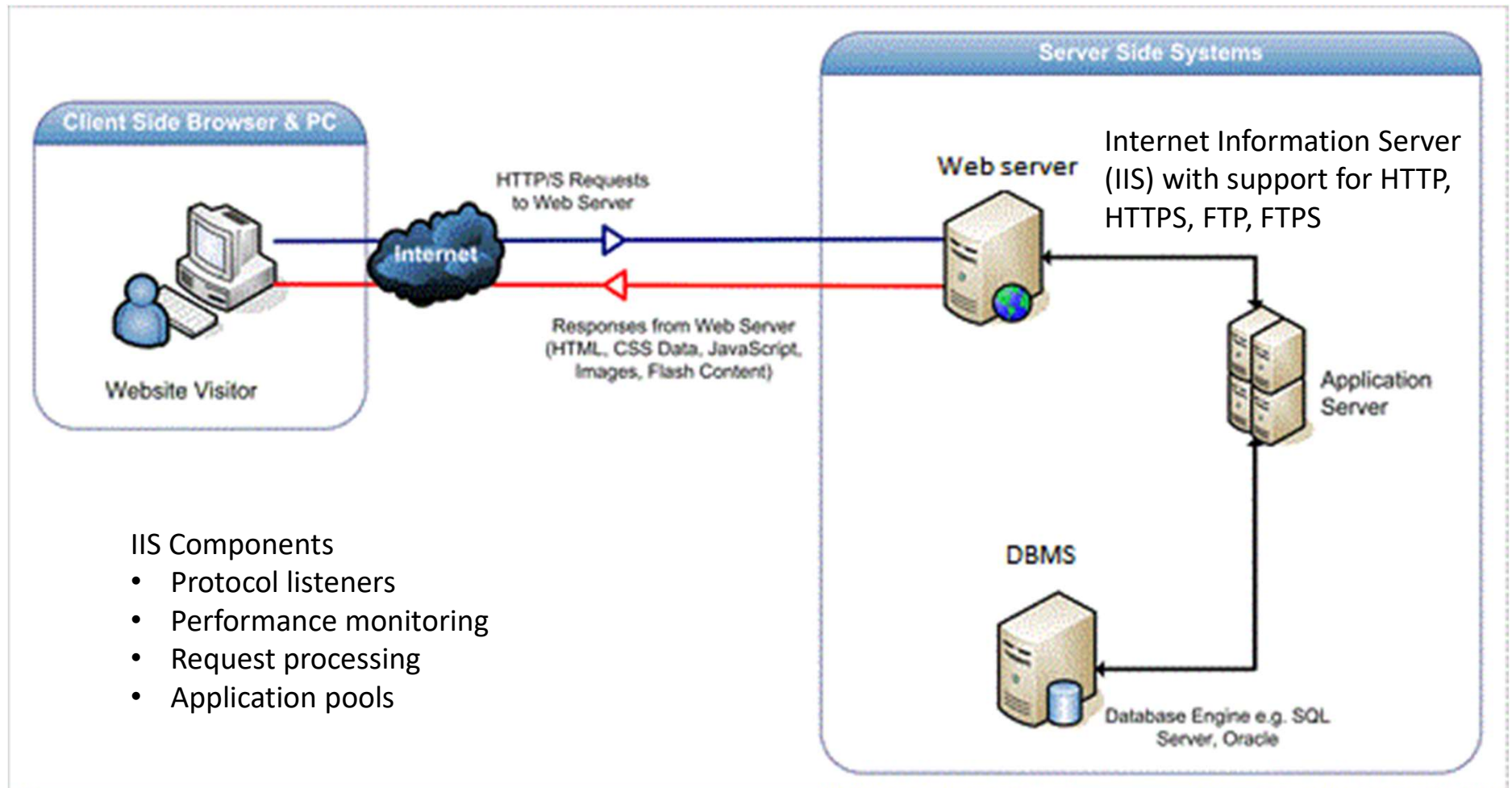


MODULE 3

Server Side API: Part 1



Anatomy of an HTTP request



ASP.NET Framework

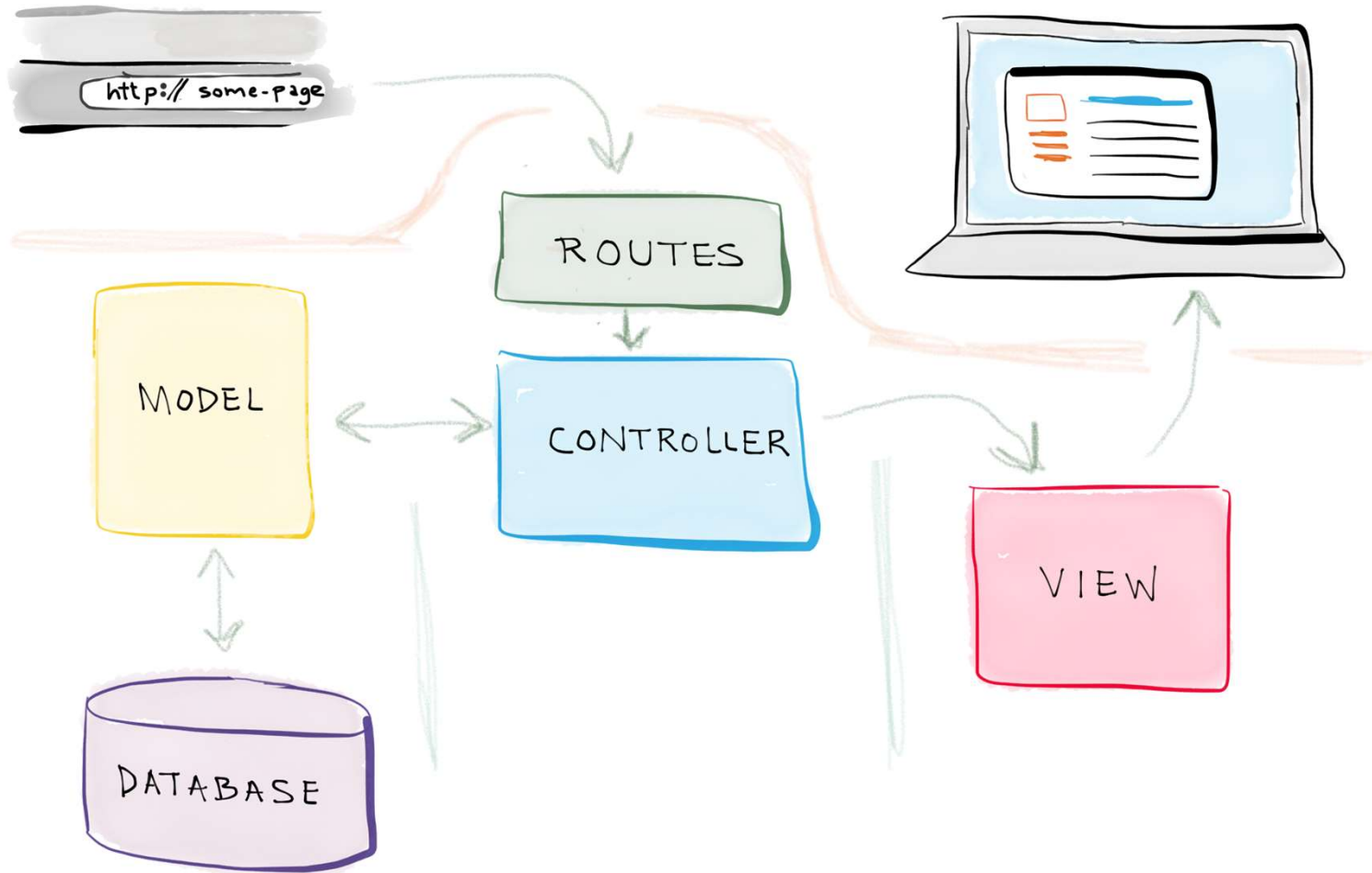
- Offers security, request management, session management, configuration management
- Offers three frameworks for web applications

Web Forms	ASP.NET MVC	ASP.NET Web Pages
Similar to desktop applications	Specific pattern design	Entry level ASP.NET
Very WYSIWYG heavy	Separation of concerns	Similar in structure to PHP apps
Work with web components	Most popular of the three	Best for small projects
Grandfather of others		

MVC Pattern

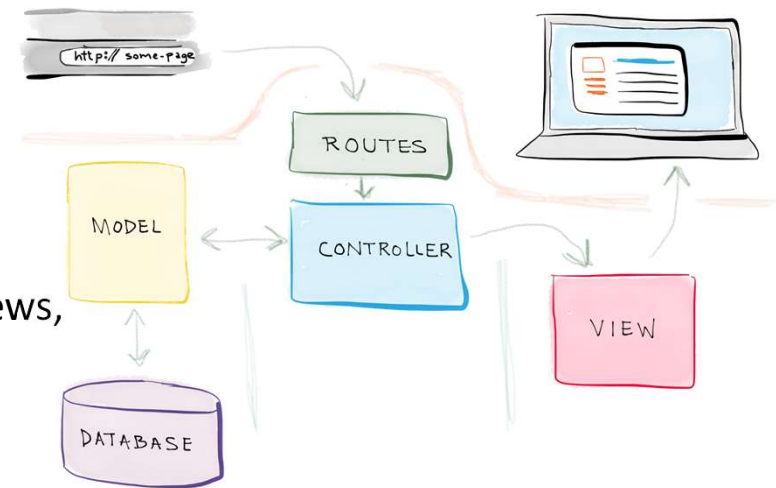
- MVC = Model View Controller
- Model
 - application state and business logic
 - only part of the application that talks to the database
 - Models could be classes
- View
 - presents data to user
 - accepts input from user
 - Views might be a desktop display, a mobile display, a file output based on a model
- Controller
 - Takes input from the view and passes it to the appropriate model objects
 - Grabs all necessary building blocks and organizes them for the output
 - Takes results from the model and passes it to the appropriate view

The Web in Crayons

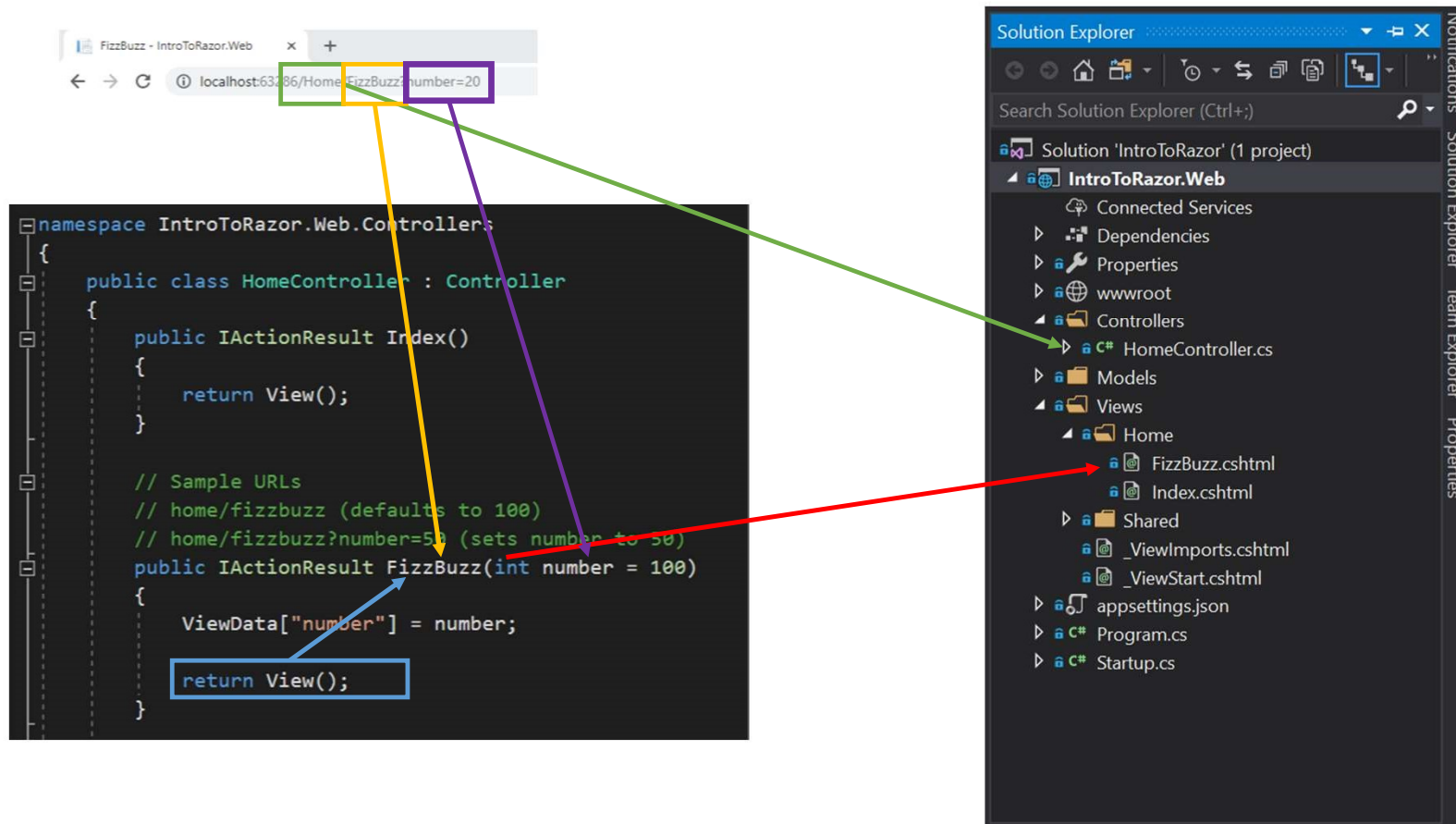


The MVC Application Lifecycle

- Request
 - Creation of Request begins with a URL
 - When the request arrives the route maps to a controller and a specific action
 - The request arrives with additional parameters that are accessible in code
- Response
 - The action is executed to provide a response
 - Response are sent back as an Action Result
 - Action Result is an abstract class. Results can be Views, Files, JSON, etc.



The man behind the curtain



Controller Decoration

```
namespace WebAPI.Controllers
```

```
{
```

```
    [Route("api/[controller]")]
```

```
    [ApiController]
```

```
    public class ValuesController : ControllerBase
```

```
{
```

This defines the route to use.
Normally, starts with api/

This brings in some new features for APIs.

- Auto model validation
- Auto binding of JSON to Model

Provides the needed controller
functions without support for
building Views

Decorating Methods

Attribute	Usage
[HttpGet]	Matches actions on GET requests, to retrieve resources
[HttpPost]	Matches actions on POST requests, to create resources
[HttpPut]	Matches actions on PUT requests, to update resources
[HttpDelete]	Matches actions on DELETE requests, to delete resources

HttpGet

- In APIs, it is common to provide at least two [HttpGet] actions:
 - An action that retrieves all resources.
 - An action that retrieves a single resource given an identifier.

```
[HttpGet]
public ActionResult<List<Article>> GetAll()
{
    return dao.GetArticles().ToList();
}
```

```
[HttpGet("{id}", Name = "GetArticle")]
public ActionResult<Article> GetArticle(int id)
{
    var article = dao.GetArticle(id);

    if (article != null)
    {
        return article;
    }

    return NotFound();
}
```

HttpPost

- For adding new information to the system.
 - If you are updating, use HttpPut

```
[HttpPost]
public ActionResult Create(Article article)
{
    dao.AddArticle(article);

    // Return a created at route to indicate where the resource can be found
    return CreatedAtRoute("GetArticle", new { id = article.Id }, article);
}
```

HttpPut

Used to update data in the system.

1. First look up the resource that the user is updating to ensure it exists (return an HTTP 404 if not).
2. Update the resource with the new content passed in.
3. Return HTTP 204 (No Content), indicating the server has fulfilled the request.

```
[HttpPut("{id}")]
public ActionResult Update(int id, Article updatedArticle)
{
    // Get the existing article
    var existingArticle = dal.GetArticle(id);

    // If that article does not exists, return 404
    if (existingArticle == null)
    {
        return NotFound();
    }

    // Copy over the fields we want to change
    existingArticle.Title = updatedCity.Title;
    // ...

    // Save back to the database
    dao.UpdateArticle(existingArticle);

    // return a 201
    return CreatedAtRoute("GetArticle", new { id = article.Id }, article);
}
```

HttpDelete

- Used for “deleting” data
 1. First look up the resource that the user is deleting to ensure it exists (return an HTTP 404 if not).
 2. Delete the resource.
 3. Return HTTP 204 (No Content)

```
[HttpDelete("{id}")]
public ActionResult Delete(int id)
{
    var article = dao.GetArticle(id);

    if (article == null)
    {
        // return HTTP 404
        return NotFound();
    }

    // delete the resource
    dao.DeleteArticle(id);

    // return HTTP 204
    return NoContent();
}
```

LET'S CODE!



ELEVATE  YOURSELF

WHAT QUESTIONS DO
YOU HAVE?



Reading for tonight: **Server Side APIs Part 2**

