ELTE EÖTVÖS LORÁND UNIVERSITY

MODERN WEB APPLICATION DEVELOPMENT IN .NET

Dr. Máté Cserép

Associate professor

Reviewer: Roxána Provender

ELTE EÖTVÖS LORÁND UNIVERSITY

DATABASE MANAGEMENT WITH ENTITY FRAMEWORK CORE

Modern web application development in .NET

Lecture 1

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

DATABASE MANAGEMENT

Relational database management systems (*RDBMS*) store data in *tables*

- Tables have columns (*fields*) and rows (*records*)
- The definition of the table structure is called a *schema*

For fields, *keys* can be defined to specify constraints (e.g. unique occurrences) and to increase the efficiency of query operations (*indexing*)

- Tables may have a *primary key* that specifies the primary, physical order of the data

Relationships between tables can be defined using *foreign keys* with optionally *reference integrity checks* enforced

DATABASE MANAGEMENT

Microsoft has its own SQL database management solution, *Microsoft SQL Server* (*MSSQL*)

- SQL Server Management Studio is a widely used, free client tool
  - *Visual Studio itself* can also be used (*View/Server Explorer*, *View/SQL Server Object Explorer*, *Tools/Sql Server*), with limited capabilities
- has its own dialect language (*Transact-SQL*), which is compatible with the SQL standard
  - includes some special statements/types, e.g. automated increment feature is supported through the **`IDENTITY`** statement
- user management supports individual accounts as well as Windows-based authentication

DATABASE MANAGEMENT

The *connection string* defines the parameters of the database connection

- usually contains the name or IP address of the server, the name of the database, and further connection data (username/password)
- the exact content varies from one database manager to another
  - e.g. (SQL Server with standard security):
    ```
    "Server=localhost;Database=myDataBase;
     User Id=myUser;Password=myPassword;"
    ```
  - e.g. (PostgreSQL with Windows authentication):
    ```
    "Server=127.0.0.1;Port=5432;Database=myDataBase;
     Integrated Security=true;"
    ```
- Reference: www.connectionstrings.com

DATABASE MANAGEMENT

There are multiple solutions to manage databases in the .NET Core framework

- *native connection*: execute SQL statements directly on the physical database
- *logical relational model*: represents the physical structure and of the relational database in the memory (tables, columns, rows, relations)
- *entity-based object relational model* (*Entity Framework*): maps the database schema and the contained data into an object-oriented structure in a parameterized manner

We will briefly introduce all approaches, but then focus on using the Entity Framework during the course, as the modern approach!

ELTE EÖTVÖS LORÁND UNIVERSITY

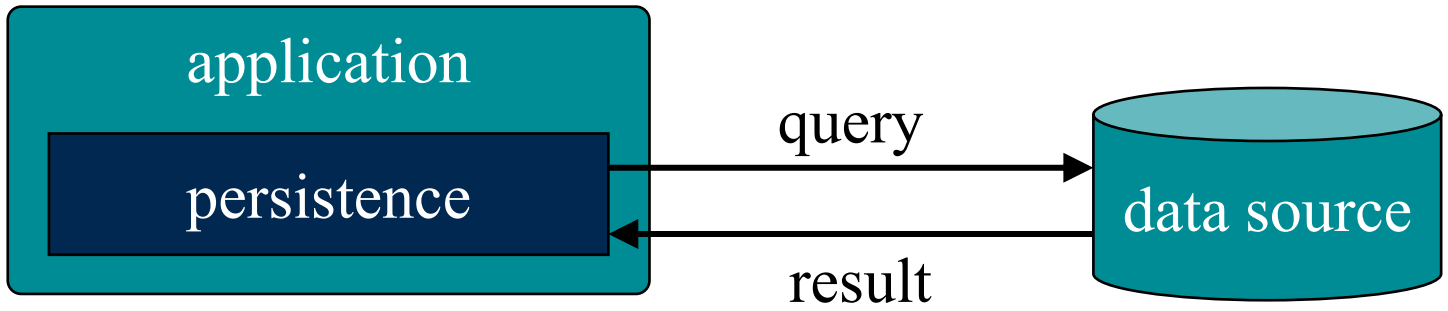
DATABASE MANAGEMENT WITH NATIVE CONNECTION

Modern web application development in .NET

NATIVE CONNECTION

Native (direct) connection allows database queries (SQL) to be executed on the physical database

- *Advantages:* efficient use of resources, direct communication
- *Disadvantages:* SQL knowledge is required, commands run on actual data (thus requiring a constant connection to the database), complex activities could be difficult to describe



NATIVE CONNECTION

The connection to the database is provided by the **`SqlConnection`** class using the appropriate connection string, e.g.:

```
SqlConnection con = new SqlConnection("…");
```

- With a connection, commands can be created using the **`SqlCommand`** class, where the **`CommandText`** property stores the instruction
- The commands can be executed in different ways:
  - **`ExecuteNonQuery()`** runs non-query instructions;
  - **`ExecuteScalar()`** runs instructions that query a single result;
  - **`ExecuteReader()`** runs multi-result queries and places them in a **`SqlDataReader`** reader object, that can be read line by line.

NATIVE CONNECTION

Example:

```
SqlCommand command = con.CreateCommand();
command.CommandText = "select * from MyTable";
SqlDataReader reader = command.ExecuteReader();

while (reader.Read()) {
  // iterate while we can read a new line
  Console.WriteLine(reader.GetInt32(0) + ", " +
     reader.GetString(1));
  // fetch and convert data to appropriate type
};
```

ELTE EÖTVÖS LORÁND UNIVERSITY

DATABASE MANAGEMENT WITH LOGICAL RELATIONAL MODEL

Modern web application development in .NET

LOGICAL RELATIONAL MODEL

The physical organization of the database is reflected in the program code using general types from the **`System.Data`** namespace:

- databases are mapped to **`DataSet`**, tables to **`DataTable`**;
- rows are represented by **`DataRow`**, fields by **`DataColumn`**;
- relational relationships can be described using **`DataRelation`** objects, while other constraints using **`Constraint`** objects.

Data is loaded into the **`DataSet`** via the **`DataAdapter`** and changes are synchronized with the database.

The stored values are not strongly typed and represented as **`Object`**s.

LOGICAL RELATIONAL MODEL

Sample data table (SQL):

```
CREATE TABLE Customer(
  Email VARCHAR(MAX) PRIMARY KEY,
  Name VARCHAR(50)
);
```

Define logical relational model:

```
// Assume that con is valid SqlConnection object
string queryString = "SELECT * FROM Customers";
SqlDataAdapter adapter = new SqlDataAdapter(queryString, con);
DataSet dataSet = new DataSet();
adapter.Fill(dataSet, "Customers");
```

LOGICAL RELATIONAL MODEL

Sample usage:

```
DataTable table = dataSet.Tables["Customers"];
DataRow newRow = table.NewRow();
newRow["Email"] = "mcserep@inf.elte.hu";
newRow["Name"] = "Máté";
table.Rows.Add(newRow); // add new row to the collection

// display all rows from all tables
foreach(DataTable table in dataSet.Tables) {
  foreach(DataRow row in table.Rows) {
    foreach(DataColumn column in table.Columns) {
      Console.WriteLine(row[column]);
    }
  }
}
```

ELTE EÖTVÖS LORÁND UNIVERSITY

DATABASE MANAGEMENT WITH OBJECT-RELATIONAL MAPPING

Modern web application development in .NET

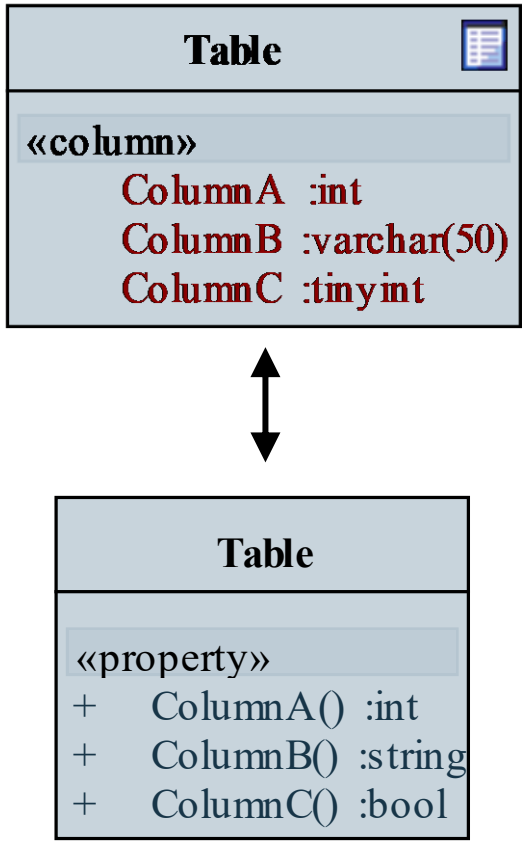
OBJECT-RELATIONAL MAPPING

Data management programs are usually built in an object-oriented way, therefore the data management should also be done this way

In relational databases:

- data is grouped into tables, which defines the schema, the way the data is structured, i.e. the *type*
- a row stores the data for a given element, i.e. the row is an *instance* of the type

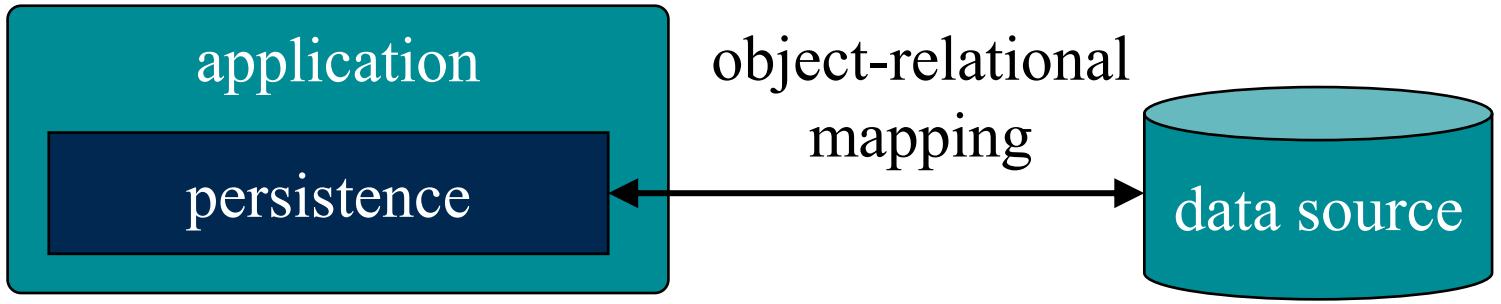
This mapping is easily translatable to an object-oriented environment, rows are objects, tables are classes



OBJECT-RELATIONAL MAPPING

The mapping is called *object-relational mapping* (*ORM*)

- provides a high-level transformation of the database that is easy to use in the program
- also controls the way the data is handled
- the created entity classes are responsible for storing data, not performing operations on them



ENTITY FRAMEWORK

Entity Framework Core implements a platform-independent, complex, object-relational representation of data

- typically, the object-oriented representation of table is an *entity type* (class), and for a row is an *entity object*, but this can be customized
- supports relationships between entities (association, inheritance)
- supports language-integrated queries (LINQ), dynamic data loading, asynchronous data handling
- requires the **`Microsoft.EntityFrameworkCore`** NuGet package and a provider specific package
  - e.g. **`Microsoft.EntityFrameworkCore.SqlServer`** for MSSQL
  - namespace is **`Microsoft.EntityFrameworkCore`**

ENTITY FRAMEWORK

- the advantage of its modular architecture is to load only the necessary components and providers

Database engine	Entity Framework Core NuGet package
MSSQL	Microsoft.EntityFrameworkCore.SqlServer
SQLite	Microsoft.EntityFrameworkCore.Sqlite
MySQL / MariaDB	MySql.Data.EntityFrameworkCore Pomelo.EntityFrameworkCore.MySql
Oracle	Oracle.EntityFrameworkCore
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL

ENTITY FRAMEWORK

There are 3 approaches for defining the model:

- *Code first*: Define the model as code and generate the database schema based on it. Good for new projects.
- *Database first*: Maps the database structure to the entity model, the code is generated based on the database schema. Good for projects with a large pre-existing database.
- *Model first*: Manually build the model with modelling tool and set up the relationships. The database schema and the code is generated from the model.

Changes to the model and database schema can be synchronized.

DEFINING ENTITY MODELS

Example (relation for customers):

```
CREATE TABLE Customer(
  Email VARCHAR(MAX) PRIMARY KEY,
  Name VARCHAR(50),
  AddressId INTEGER,

  CONSTRAINT CustomerToAddress
    FOREIGN KEY (AddressId)
    REFERENCES Address (Id)
);
```

Example (relation for addresses):

```
CREATE TABLE Address(
  Id INTEGER PRIMARY KEY,
  Country VARCHAR(50),
  City VARCHAR(50),
  Address VARCHAR(MAX),
  PostalCode VARCHAR(10)
);
```

DEFINING ENTITY MODELS

Tables are plain C# classes:

- One class per table; one property per column
- No base class or interface required
- Navigation properties express relationships
- By convention the `Id` or `<Class>Id` field becomes the primary key
- Annotations override conventions when needed

Common EF annotations:

- `[Table("...")]` – table name
- `[Column("...")]` – column name, type
- `[Key]` – primary key
- `[ForeignKey("...")]` – relationship
- `[Required]`, `[MaxLength(n)]`
- `[DatabaseGenerated(...)]` identity or computed
- `[NotMapped]` – ignore property

DEFINING ENTITY MODELS

C# class for customers:

```
class Customer {
  [Key] // primary key
  public string Email { get; set; }

  [MaxLength(50)] // DB constraint
  public string Name { get; set; }

  [ForeignKey("AddressId")]
  public Address Address { get; set; }

  public ICollection<Order> Orders
  { get; set; }
}
```

C# class for addresses:

```
class Address {
  [Key] // primary key
  public int Id { get; set; }

  [MaxLength(50)] // DB constraint
  public string Country { get; set; }
  [MaxLength(50)]
  public string City { get; set; }
  public string Address { get; set; }
  [MaxLength(10)]
  public string PostalCode { get; set; }
}
```

DEFINING ENTITY MODELS

Example (relation for orders):

```
CREATE TABLE Order(
  Id INTEGER PRIMARY KEY,
  Content VARCHAR(MAX),
  Price FLOAT,
  CustomerEmail VARCHAR(MAX),

  -- Foreign key
  CONSTRAINT OrderToCustomer
    FOREIGN KEY (CustomerEmail)
    REFERENCES Customer (Email)
);
```

C# class for orders:

```
class Order {
  [Key] // primary key
  public int Id { get; set; }

  public string Content { get; set; }
  public float Price { get; set; }

  [ForeignKey("CustomerEmail")]
  public Customer Customer { get; set; }
}
```

USING ENTITY MODELS

Entities are managed by a database context (**DbContext**) that defines the schema of the included tables (**DbSet**)

- This provides an asynchronous model; changes are only saved to the database upon a separate call (**SaveChanges**)
- Example:

```
public class SalesContext : DbContext
{
  public DbSet<Customer> Customers { get; set; }
  // ...
}
```

USING ENTITY MODELS

The **`DbSet`** ensures the execution of queries and data management

- Supports creation (**`Create`**), addition (**`Add`**, **`Attach`**), search (**`Find`**), modification (**`Update`**), deletion (**`Remove`**)
- Data and queries are managed in a lazy manner
  - Data is only loaded into memory when queried, but it can be loaded in advance (with **`Load`**)
  - LINQ queries are converted to SQL statements and run directly on the database

USING ENTITY MODELS

Example:

```
SalesContext context = new SalesContext();
Customer? customer = context.Customers.FirstOrDefault(
  cust => cust.Email == "mcserep@inf.elte.hu"); // LINQ query

if (customer == null) {
  customer = new Customer {
    Name = "Cserép Máté",
    Email = "mcserep@inf.elte.hu"
  };
  context.Customers.Add(customer); // adding a new entity
  context.SaveChanges(); // persisting changes
}
```

LOADING RELATED ENTITIES

Entities do not store related data (available through *navigational properties*), but they can be loaded in several ways:

- Eager loading can be done by adding the necessary **`Include()`** procedure calls when constructing the LINQ query. Then, the related entities will be loaded together with the original query.

```
SalesContext context = new SalesContext();
var customers = context.Customers
  .Where(cust => cust.Email.EndsWith("elte.hu")
  .Include(cust => cust.Address);
foreach (Customer c in customers) {
  Console.WriteLine(c.Address.City);
}
```

LOADING RELATED ENTITIES

Using `Include()` only descends one level. To eagerly load entities further along a navigation chain, follow it with `ThenInclude()`:

```
var orders = context.Orders
   .Include(o => o.Customer)
   .ThenInclude(c => c.Address);
```

- Each `ThenInclude()` operates on the previously included entity.
- Use another `Include()` to start a new chain from the root.

LOADING RELATED ENTITIES

- Explicit loading can be done to retrieve the related entities of an already loaded entity object, using the **`Load()`** method of the entity.

```
var customer = context.Customers.Single(
  cust => cust.Email = "mcserep@inf.elte.hu");

// Load address
context.Entry(customer).Reference(cust => cust.Address).Load();

// Load orders
context.Entry(customer).Collection(cust => cust.Orders).Load();
```

LOADING RELATED ENTITIES

- Lazy loading loads the related entities when they are needed (when they are first evaluated). For this, we need to install the **`Microsoft.EntityFrameworkCore.Proxies`** NuGet package and enable lazy loading (**`UseLazyLoadingProxies()`**) for out database context.

```
var options = new DbContextOptionsBuilder<SalesContext>()
  .UseLazyLoadingProxies().UseSqlServer("...").Options;
var context = new SalesContext(options);

var customer = context.Customers.Single(
  cust => cust.Email = "mcserep@inf.elte.hu");
var address = customer.Address; // Lazy loading
```

LOADING RELATED ENTITIES

We must mark the navigation properties that we want to load lazily as virtual. They will be replaced with a derived proxy type based on the *proxy design pattern*.

```
class Customer {
  [Key] // primary key
  public string Email { get; set; }
  [MaxLength(50)] // DB constraint
  public string Name { get; set; }
  [ForeignKey("AddressId")]
  public virtual Address Address { get; set; }
  public virtual ICollection<Order> Orders { get; set; }
}
```

QUERYING DATA WITH LINQ

Queries can be prepared and stored as objects using the **`IQueryable<T>`** type. **`IQueryable<T>`** inherits **`IEnumerable<T>`**.

- Queries can be dynamically expanded and executed only upon request (e.g., **`ToList()`** or **`Load()`**) or upon evaluation (e.g., **`foreach`**).
- Database operations can be formulated in a purely object-oriented manner in C#.
- *Example 1:*

```
IQueryable<Customer> query = context.Customers
  .Where(cust => cust.Email.EndsWith("elte.hu"))
  .Where(cust => cust.Orders.Count() > 0);
query = query.OrderBy(cust => cust.Name);
// no database query executed as this point
```

QUERYING DATA WITH LINQ

Example 2:

```
IQueryable<Customer> query1 = context.Customers
  .Include(cust => cust.Address);
  // eager loads the address
  // also works: .Include("Address")
IQueryable<Customer> query2 = query1
  .Where(cust => cust.Address.City == "Budapest")
  .OrderBy(cust => cust.Name);
  // extend query (no execution yet)
List<Customer> r = query2.ToList(); // execute query
foreach(Customer cust in query2) {/* ... */ }
```

QUERYING DATA WITH LINQ

Example 3:

```
bool anyBudapest1 = query1
  .Any(cust => cust.Address.City == "Budapest");
  // executes query on DB

bool anyBudapest2 = query1
  .Any(cust => cust.Address.City == "Budapest");
  // executes query on DB;
  // if data in DB changed, result may be different

query1.Load(); // explicit loading for query
anyBudapest = query1
  .Any(cust => cust.Address.City == "Budapest");
  // executes query in memory
```

QUERYING DATA WITH LINQ

Generating an SQL query:

```
IQueryable<Customer> query = context.Customers
  .Where(cust => cust.Email.EndsWith("elte.hu"))
  .Where(cust => cust.Orders.Count() > 0)
  .OrderBy(cust => cust.Name);
Console.WriteLine(query.ToQueryString());
```

```
SELECT * FROM Customers AS c
WHERE (c.Email LIKE '%elte.hu’)
  AND ((SELECT COUNT(*)
        FROM Orders AS o
        WHERE c.Email = o.CustomerEmail)
       > 0)
ORDER BY c.Name
```

NATIVE SQL COMMANDS WITH EF CORE

Entity Framework can also be used to execute native SQL commands:

```
var customers = await context.Customers
  .FromSql($"SELECT * FROM Customers")
  .ToListAsync();
```

- The **`FromSql()`** method has built-in SQL injection when passing parameters:

```
var email = "%elte.hu";
var customers = await context.Customers
  .FromSql($"SELECT * FROM Customers WHERE Email LIKE {email}")
  .ToListAsync();
```

- Can used to execute stored procedures:

```
var customers = await context.Customers
  .FromSql($"EXECUTE GetMostFrequentCustomers").ToListAsync();
```

THE CHANGETRACKER

- Each **`DbContext`** has a **`ChangeTracker`** that monitors loaded entities and detects modifications.
- On **`SaveChanges()`**, EF generates the necessary `INSERT`, `UPDATE` and `DELETE` statements based on tracked state.
- Every tracked entity has one of five states:
  - `Unchanged` – loaded from the DB, not modified
  - `Added` – new entity, will be INSERTed
  - `Modified` – changed, will be UPDATEd
  - `Deleted` – marked for DELETE
  - `Detached` – not tracked by the context

WORKING WITH THE CHANGETRACKER

- Modifying a tracked entity – EF detects the change automatically:
```
var customer = await context.Customers.FindAsync(id);
customer.Email = "new@elte.hu";
await context.SaveChangesAsync();  // UPDATE
```
- Adding and removing entities:
```
context.Customers.Add(newCustomer);      // Added
context.Customers.Remove(oldCustomer);   // Deleted
```
- Inspecting state and current/original values:
```
var entry = context.Entry(customer);
EntityState state = entry.State;
var original = entry.OriginalValues["Email"];
```
- Use **`AsNoTracking()`** for read-only queries to skip tracking and improve performance.

DETACHING ENTITIES

- Detaching tells the ChangeTracker to stop tracking an entity – its changes will no longer be persisted on **`SaveChanges()`**.
- Typical scenarios:
  - Long-lived contexts – periodically detach to keep the tracker small and fast.
  - Bulk processing – detach already-saved entities to free memory and avoid re-checking them.
  - Discarding pending changes for a single entity without affecting the rest of the unit of work.
  - Re-attaching an entity from another context (detach first, then **`Attach()`** on the new one).
  - E.g.: `context.Entry(customer).State = EntityState.Detached;`

FURTHER TOPICS TO LEARN

Table splitting

- Map two or more entities to a single database table row

Entity splitting

- Map a single entity to multiple rows (in multiple tables)

Inheritance

- Map inherited entity types to database relations
- *Table-per-hierarchy* (TPH) with *discriminator* field
- *Table-per-type* (TPT), *Table-per-concrete-type* (TPC)

Owned entities

- Flatten composite structures on the database level

ELTE EÖTVÖS LORÁND UNIVERSITY

WEB DEVELOPMENT WITH MVC ARCHITECTURE

Modern web application development in .NET

Lecture 2

Dr. Máté Cserép

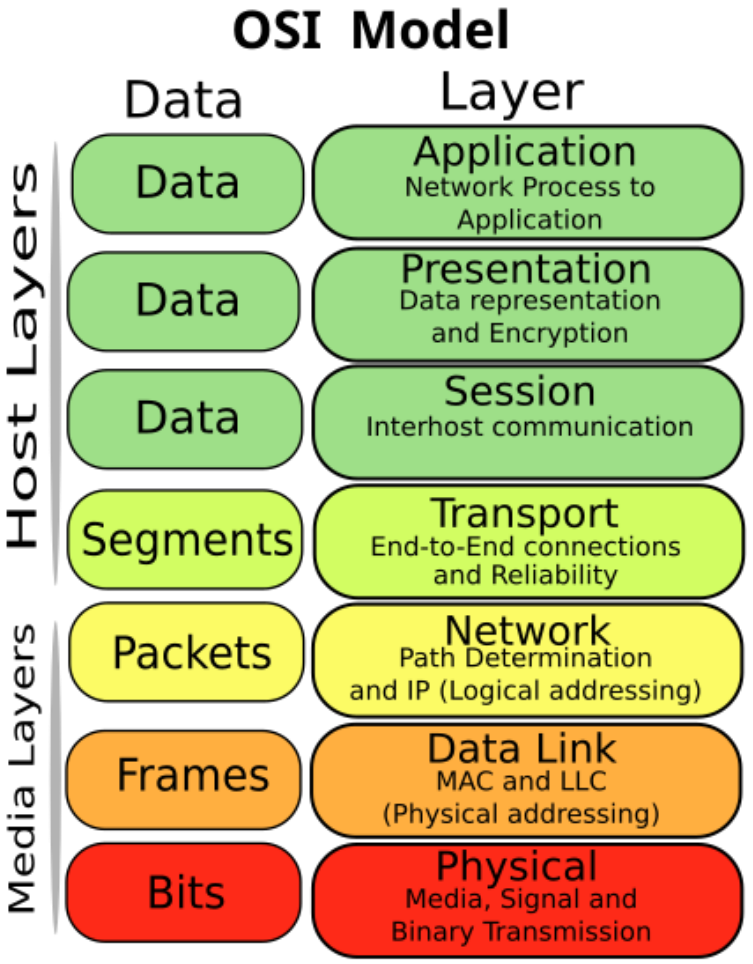
mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

NETWORK COMMUNICATION

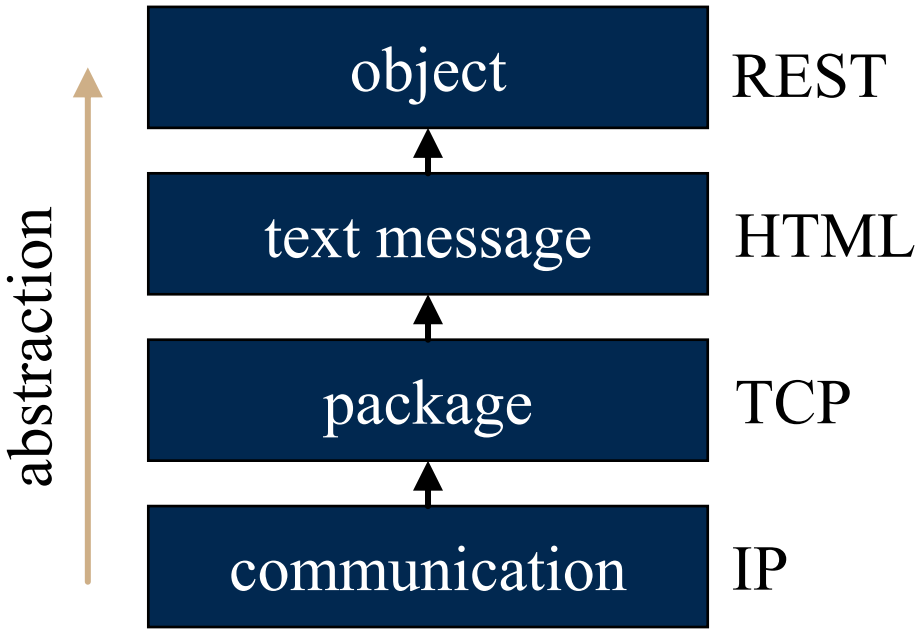
Applications often communicate over a network, usually based on the *OSI model*

- Which provides different communication solutions in a layered architecture
  - *application*: HTTP, SSH, SMTP, etc.
  - *presentation*: HTML, CSS, JSON, TLS, etc.
  - *transport*: UDP, TCP
  - *network*: IP
- Additional communication and service *architectures* (MVC, REST) and *frameworks* (ASP.NET, WCF) are built on top of the model



NETWORK COMMUNICATION

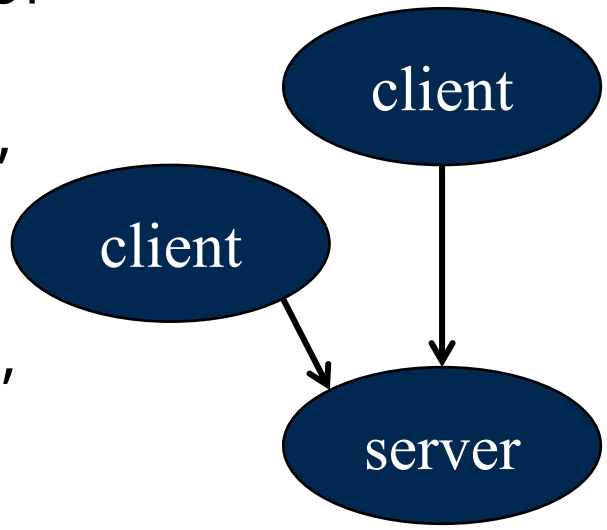
- Upper layers provide abstraction, masking the lower layers
  - Increases *security* and *reliability*
  - Ensures communication with complex content



NETWORK COMMUNICATION

Application systems that communicate over a network can be divided into two main categories:

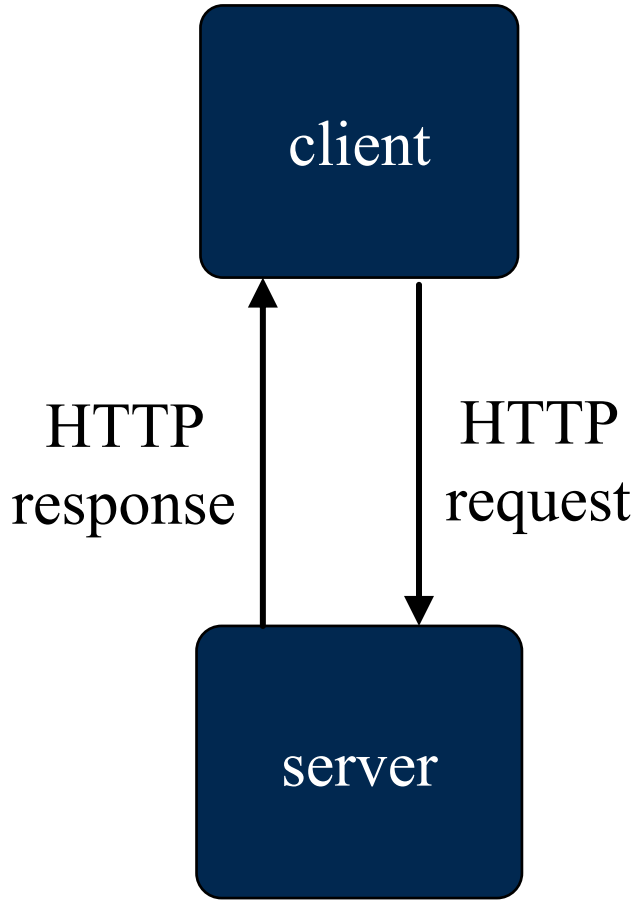
- *decentralized*, directly connected (P2P)
- *client-server*: a central machine serves any number of connected machines that may be:
  - *thick clients*: the execution is done by the client, the server is mainly used for communication and data management
  - *thin clients*: the execution is done by the server, the client is used for interaction



HTTP PROTOCOL

HTTP (Hypertext Transfer Protocol) is the standard protocol for data transfer on the web

- based on the request/response paradigm, i.e. the client sends a request to which the server responds
  - the response is interpreted and displayed by the client
  - the first line of the response is the *status line*, which gives feedback on the execution of the request (e.g. 200 successful, 404 not found, 503 service not available)



HTTP STATUS CODES

The status line of each response begins with a 3-digit code

Grouped by first digit:

- **1xx** informational *(e.g. 100 Continue)*
- **2xx** success – 200 OK, 201 Created, 204 No Content
- **3xx** redirection – 301 Moved Permanently, 302 Found, 304 Not Modified
- **4xx** client error – 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 410 Gone
- **5xx** server error – 500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable, 504 Gateway Timeout

HTTP PROTOCOL

- Request can be of several types (*methods*), e.g.:
  - *GET*: a request for a given resource (associated with a path)
  - *POST*: send content with data (e.g. form)
- The protocol is *stateless*, i.e. requests to the server are independent of each other (no state-preservation between two requests)
  - at the same time, the server can establish a *session* with the client (the identification data is saved on the client side)
- The protocol can be made secure using TLS encryption (HTTPS)

DEVELOPMENT IN ASP.NET

Microsoft ASP.NET is an evolution of *ASP (Active Server Pages)* with .NET programming support

- a (server-side) programming interface for creating dynamic web pages and services over HTTP

ASP.NET Core is an open source, cross-platform framework based on .NET Core, which offers two programming models:

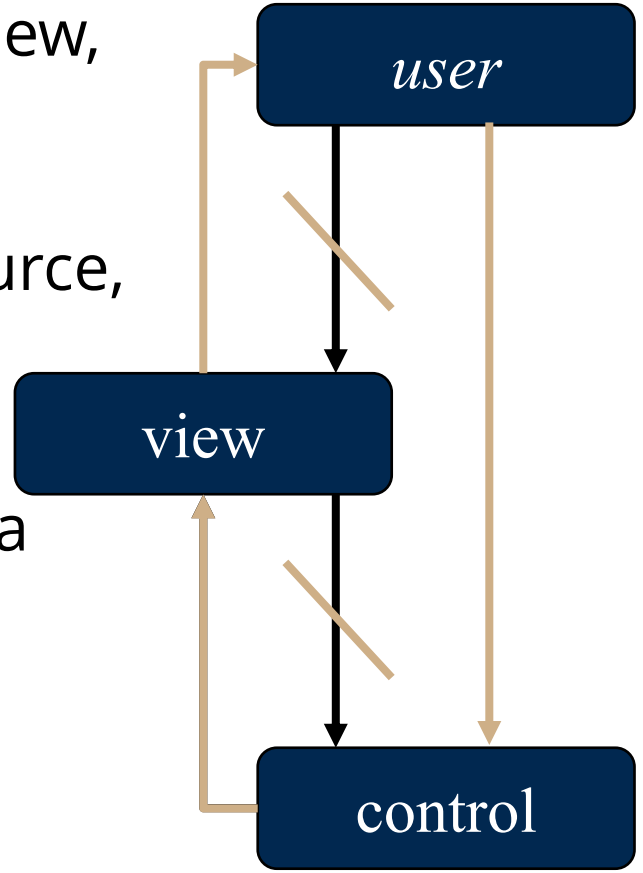
- *Razor Pages*: for simple, largely static applications that support dynamic functionality and data management, based on MVVM
- *ASP.NET MVC*: an architecture for developing complex, dynamic web pages, based on MVC

THE MVC ARCHITECTURE

In a *desktop environment*, the user connects to the view, which ensures the execution of the appropriate command (event handler, command)

In a *web environment*, the user connects to the resource, which is primarily determined by its path

- i.e. the user directly accesses the control
- the application must respond to the control with a (new) view associated with the resource

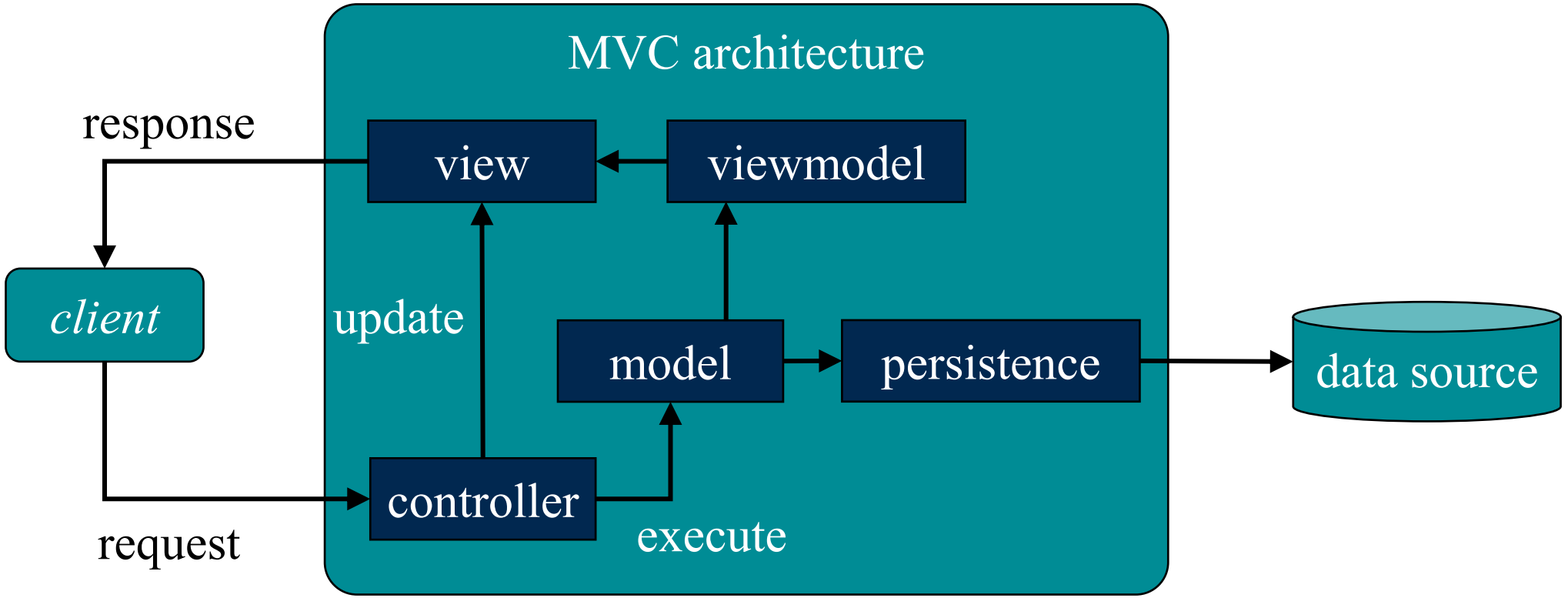


THE MVC ARCHITECTURE

The *Model/View/Controller (MVC)* architecture defines a multi-layered architecture that fits well with the web environment

- *controller* is the request server that provides the view based on the result of the request
- *model* is the implementation of the business logic
- *view* is the mostly declarative definition of the user interface; it does not contain any business logic and presents the data
- *viewmodel* is a gateway to represent data in a way that is appropriate for the view, so model and view representation of data is separated
- *persistence* is responsible for data access

THE MVC ARCHITECTURE

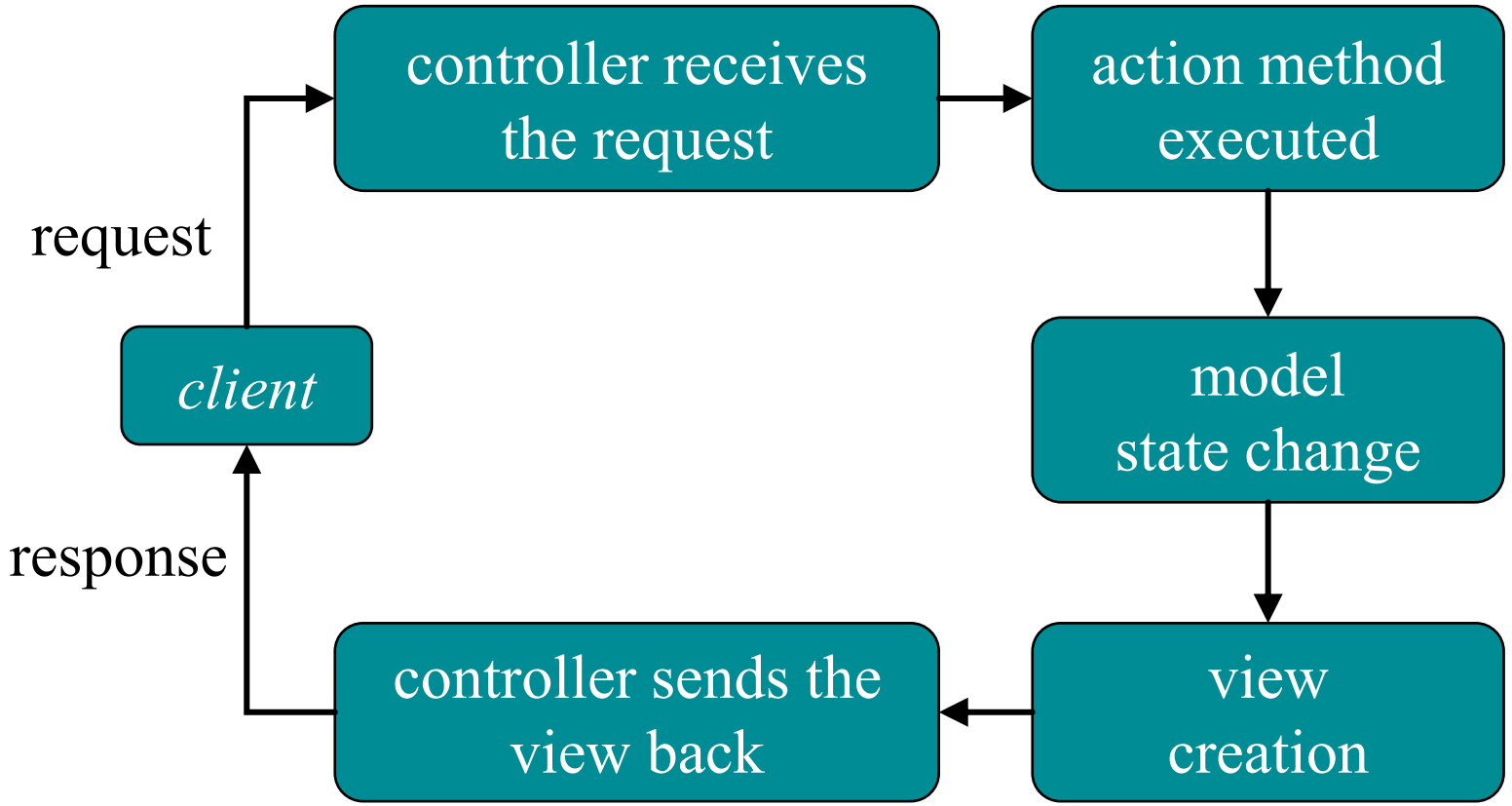


THE MVC ARCHITECTURE

Request process cycle in the MVC architecture:

1. the user makes a request to the server
2. the controller receives the request and executes the corresponding action in the model (*action method*)
3. the action executed in the model (may) causes a state change
4. the controller collects the *action result* and creates the new view (*push-based* approach)
    - another approach is to have the view retrieve the results from the controllers (*pull-based*)
    - data is transferred to the view using a view model
5. the user receives the response (view)

THE MVC ARCHITECTURE



COMPONENTS OF MVC APPLICATIONS

ASP.NET MVC applications implement the MVC architecture using dedicated components

- a view is a class (**`View`**) that is defined using a suitable markup language (e.g. *Razor*)
  - the view can use HTML and can reference the contents of the model (using data binding)
- a controller (subclass of **`Controller`**) is a class of activities in which actions (methods) are defined
  - the action result (**`IActionResult`**) is usually a view
- the model and persistence can be arbitrary

COMPONENTS OF MVC APPLICATIONS

Sample controller with multiple actions:

```
public class MyController : Controller {
  // controller
  public IActionResult Index() { // action
    return View("Index", "Welcome to the website!");
    // the result is a view with a simple string as the viewmodel
  }

  public IActionResult List(){
    ... // fetch some data from the model
    return View(list);
  }

  public IActionResult Details(String id){ … }
```

COMPONENTS OF MVC APPLICATIONS

Sample Razor view:

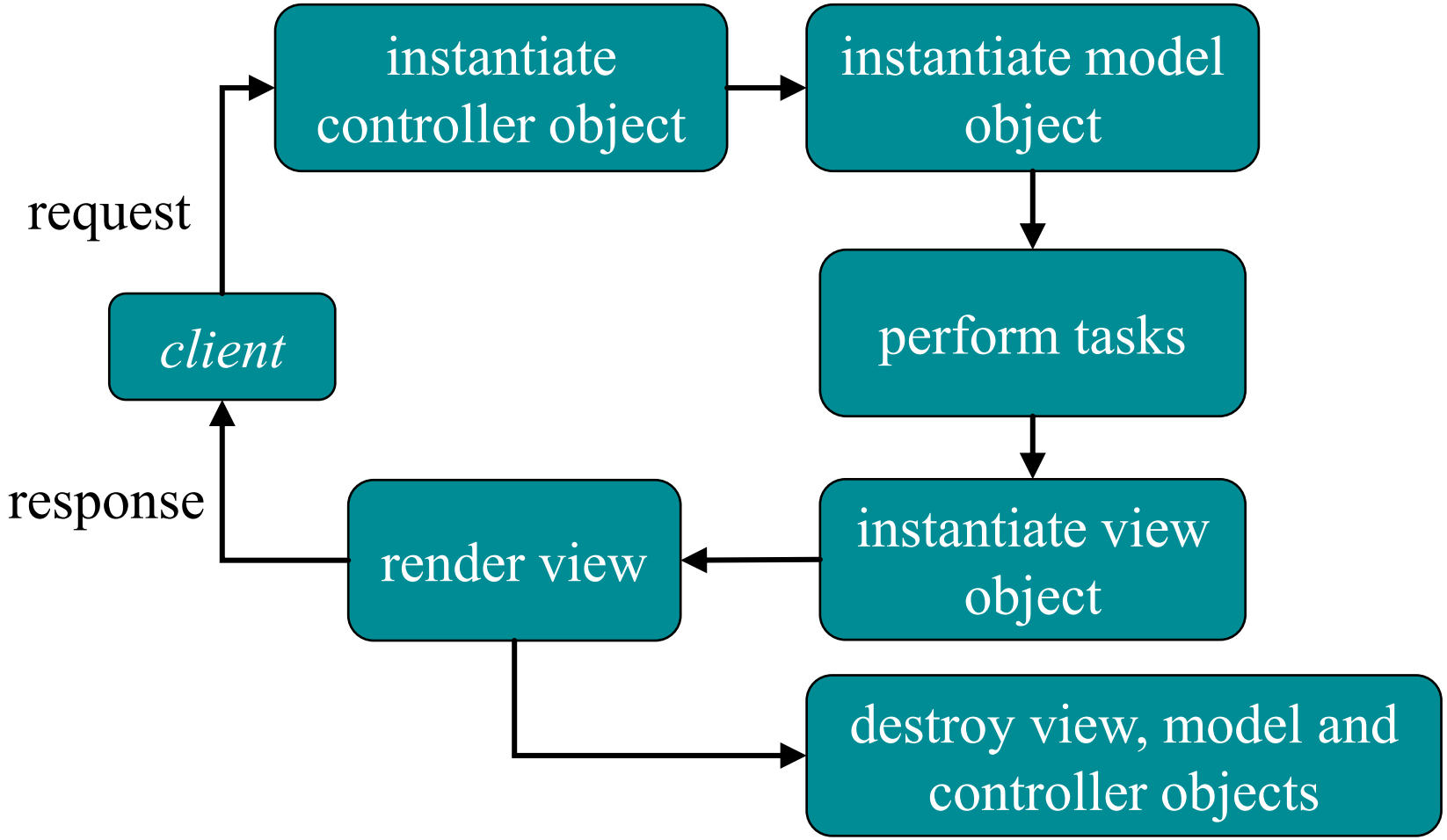
```
@model string @* the viewmodel is a simple string *@
@{ } @* code block *@

<!DOCTYPE html>
<html>
  <head>...</head> @* static content*@
  <body>
    <div>
      @Model @* display the viewmodel *@
    </div>
  </body>
</html>
```

LIFECYCLE OF WEB APPLICATIONS

The lifecycle of web applications is different from desktop and mobile applications

- the application can only respond to requests, requests are independent of each other and can arrive at any time
- *the application therefore does not keep state between two requests*
    - it is triggered by a request and instantiates objects
    - deletes objects when the request is served
    - certain data can remain in memory for a period of time
- *the persistence layer ensures that the data is preserved*

LIFECYCLE OF WEB APPLICATIONS



MVC PROJECT LAYOUT

The folder layout of the MVC application reflects the modular architecture

- the **`View`**, **`Controllers`** and **`Models`** folders contain the appropriate content
- the **`wwwroot`** directory holds the public static files (images, stylesheets, client-side scripts)
- the **`App_Data`** directory stores any data content (e.g. database files)
- the root contains the configuration (**`appsettings.json`**) and the application-level controls (**`Program.cs`**)

Additional software libraries can be installed using the NuGet package manager, as usual

ROUTING ENGINE

In an MVC architecture application, the user connects to the controller and runs its actions (with parameters)

- access and parameterization is provided by routes, which are controlled by a *routing engine*
- access is customizable (**`Program.cs`**), provided by default in the form **`<host>/<controller>/<action>/<parameters>`**
  - loads the default **`HomeController`** controller without specifying a controller
  - runs the **`Index`** action without specifying an action
  - parameters can be resolved in order or by name

ROUTING ENGINE

Examples:

Route	*Activity*
`http://myPage/`	`HomeController` → `Index`
`http://myPage/Hello`	`HelloController` → `Index`
`http://myPage/Hello/List`	`HelloController` → `List`
`http://myPage/Hello/Details/1` `http://myPage/Hello/Details?id=1`	`HelloController` → `Details` (with `id=1` parameter)

CONTROLLERS

Controllers are descendants of the Controller class, which implement actions using public operations

- the action returns a result (**`IActionResult`**), which can be
  - a view (**`ViewResult`**, **`PartialViewResult`**)
  - an error report (**`NotFoundResult`**, **`UnathorizedResult`**, **`StatusCodeResult`**)
  - redirect (**`RedirectResult`**)
  - file (**`FileResult`**), JSON (**`JsonResult`**), object (**`ObjectResult`**), other content (**`ContentResult`**)
  - empty (**`EmptyResult`**)

CONTROLLERS

E.g.:

```
public class MyController : Controller {
  ...
  public IActionResult LoadImage(String id){
    Byte[] image = … // load an image from the disk based on the Id
    if (image == null) // when no image with that Id was found
      return RedirectToAction("Error");
      // redirect to another action
    return File(image, "image/png");
    // return image as file content
}
public IActionResult Error() {
  return NotFound("Content not found.");
} // return error code
```

CONTROLLERS

- These are corresponding helper methods defined int the **`Controller`** class to easily produce the result types, e.g.:

```
return View(...); // the result will be ViewResult
```

- A view result is usually specified with a viewmodel, which is a transformation and/or reduction of the model to the view, e.g.:

```
Object viewModel = ... // create the view model
return View("Index", viewModel);
// specify the name of the view and the viewmodel
```

  - The viewmodel can be of any type, even primitive, and can be completely independent of the original model.

VIEWS

ASP.NET supports multiple description languages for defining views

- Razor has the simplest syntax and is the most popular;
- The view can be strongly typed, in which case the viewmodel type is specified, e.g. **`@model MyProject.Model.ItemModel`**
- Dynamic items are marked with the @ prefix
  - the block **`@{ ... }`** can contain arbitrary view logic in C#;
  - the block **`@* ... *@`** denotes multiline comments;
  - conditional (**`@if`**, **`@else`**) and iterative (**`@for`**, **`@foreach`**, **`@while`**) execution can be used;
  - namespace usage can be resolved with **`@using`** (otherwise canonical paths must be used).

VIEWS

- In the view the viewmodel can be referenced by the **`Model`** element
- The special **`Html`** static class is available as a HTML helper, e.g.:
  - hyperlinks to actions (**`ActionLink`**), in which we specify the action, the controller and the arguments (usually as an anonymous object), e.g.:

```
Html.ActionLink("Load image", "LoadImage",
                "Home", new {id = 1}, null);
                // Home/LoadImage?id=1 link
```

  - forms (**`BeginForm`**, **`EndForm`**)
  - display and input elements (**`LabelFor`**, **`TextBoxFor`**, **`PasswordFor`**), validators (**`ValidationMessageFor`**) for forms
  - unencoded content (**`Raw`**)

VIEWS

- The dynamic user controls can also be defined with *tag helpers*, utilizing special attributes marked with **`asp-`** prefix:
  - Linking an action:
    ```
    <a asp-controller="Home" asp-action="Index">Home</a>
    ```
  - Embed a textbox for the given property of the viewmodel:
    ```
    <input asp-for="Name">
    ```
  - Embed client side script files: `<script src="~/js/site.js" asp-append-version="true"></script>`
    - Appends the **`v=<hash>`** query parameter to the URL, for client side cache invalidation of the loaded file in the browser, in case of content change.
  - Embed a stylesheet: `<link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />`

VIEWS

- More specific paths can be handled by the Url class, e.g.:
  `Url.Content("~/style.css")`
  `Url.Action("Action", "Controller")`
- Views can depend on layouts and customized for different profiles (e.g. desktop/mobile environment)
- In ASP.NET MVC the view is an object that implements the **`IView`** interface and the view's descriptive language (*the engine*) implements the **`IViewEngine`** interface
  - It is supported to implement new engines and views by anyone

VIEWS

Beside the *strongly typed* viewmodel, we can pass data and even functions (as *lambda objects*) to the view through the special **`ViewBag`** property, available both in the controller and the view

- the **`ViewBag`** is a dynamic object, meaning it is dynamically managed and *weakly typed* (runtime type checking);
- it is also an **`ExpandoObject`**, i.e. any property or method can be assigned to it, so it can take any value, e.g.:
  ```
  ViewBag.Message = "Hello";
  ```
- the contents is available in the view as well, e.g.:
  ```
  <div>@ViewBag.Message</div>
  ```
- the **`ViewData`** can be used for the same purpose, but as a dictionary.

VIEWS

E.g. (MyController.cs):

```
public class MyController : Controller {
  // ...
  public IActionResult List() {
    ViewBag.Title = "List of names";
    // set the title of the page
    String[] itemNames = …;
    return View("ListPage", itemNames);
    // pass a string array to the ListPage view
  }

  public IActionResult Details(String name){
    return View("DetailsPage", …);
  }
}
```

VIEWS

E.g. (ListPage.cshtml):

```
@model IEnumerable<String>
@* the viewmodel is strongly typed *@
@{ Layout = null; // no layout used }

<!DOCTYPE html>
<html>
  <head>
    <title>@ViewBag.Title</title>
  </head> @* the title is defined in the controller *@
  <body>
```

VIEWS

E.g. (ListPage.cshtml):

```
    @if (Model != null){ // conditional execution
      <div>
        @foreach (String name in Model){
          <span><b>@name</b>
          <a asp-action="Details" asp-route-name="@name">See details</a>
          </span> @* link to the Details action *@
        }
      </div>
    } else {
      <div>No items found!</div>
    }
  </body>
</html>
```

ELTE EÖTVÖS LORÁND UNIVERSITY

VISUALIZATION AND CONTENT MANAGEMENT WITH ASP.NET MVC

Modern web application development in .NET

LAYOUTS AND PARTIAL VIEWS

In many cases, our view is made up of various parts

- some parts (e.g. headings, menus) appear on more than one page, others are specific to a single or certain pages
- recurring parts give our website a consistent look and experience

Repetitive content can be extracted and reused in multiple views

- these parts are therefore not necessarily associated with a single controller, but are *shared* between controllers (*shared view*) and placed in the **`Views/Shared`** directory

PARTIAL VIEWS

A *partial view* is a view that does not render the whole page, but only a part, a segment of it

- this content can be displayed in another view using the **`Html.RenderPartial()`** statement
  - we can specify the name of the partial view, the passed strongly typed viewmodel and even a customized viewdata
- e.g. (**`Views/Shared/Item.cshtml`**):

```
@model string @* the viewmodel is single string item *@

<span>Item @ViewBag.Index/@ViewBag.Count:
      <b>@Model</b>
</span>
```

PARTIAL VIEWS

- E.g. for using this partial view (**MyView.cshtml**):

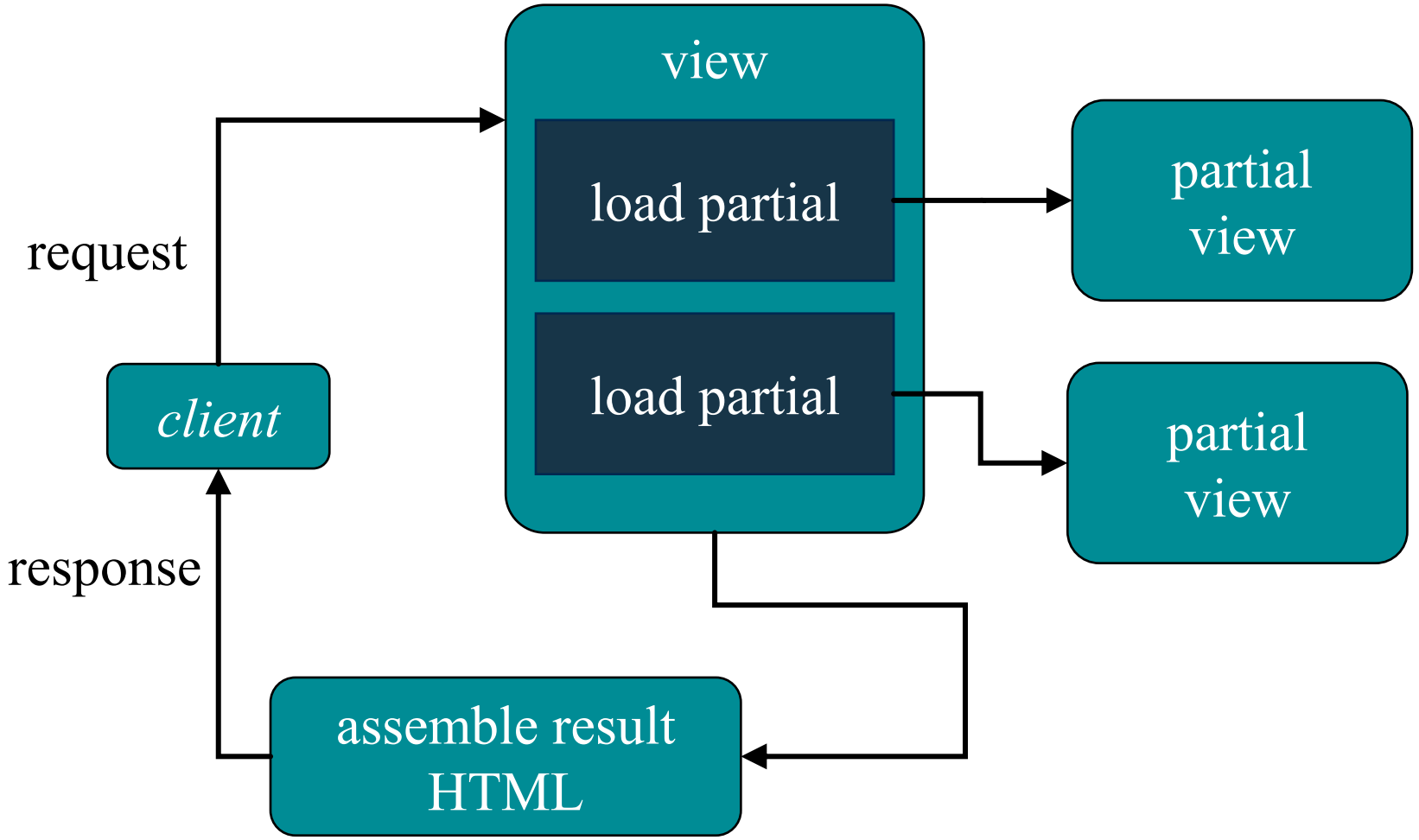
```
@model List<string> @* the viewmodel is a string list *@
@{ int idx = 1; } @* set an index counter *@

<div>
  @foreach(string item in Model) {
    @Html.RenderPartial("Item", @* render Views/Shared/Item.cshtml *@
      item, @* pass the current item *@
      new ViewDataDictionary(ViewData) {
        { "index", idx }, { "count", Model.Count }
      }
    ) @* pass the index and the length of the list *@
  }
</div>
```

PARTIAL VIEWS

The tag helper syntax offers an alternative here as well, e.g.:

```
@model List<string> @* the viewmodel is a string list *@
@{ int idx = 1; } @* set an index counter *@

@{
  var viewData = new ViewDataDictionary(this.ViewData) {
    { "index", idx }, { "count", Model.Count }
  };
} @* extend the viewdata with a new property *@

@foreach (String name in Model) {
  <partial name="Item" model="name" view-data="viewData" />
} @* pass the current item and the extended viewdata *@
```

PARTIAL VIEWS



PARTIAL VIEWS

A *partial view* can be returned by controller action as well, with the inherited **`PartialView`** method, e.g.:

```
public class ProductController : Controller {
  ...
  // PartivalViewResult is an IActionResult
  public PartivalViewResult Details(int id) {
    Item item = …
    return PartialView("Details", item);
  }
}
```

This is still reusable in other views with the **`Html.RenderAction()`** helper method.

LAYOUT VIEWS

Layouts allows us to specify multiple replaceable content areas (placeholders) within a page, for which the actual can be loaded from other views

- the layout view is the frame that defines the fixed content
- the replaceable contents are the *sections* defined in each view
  - sections are specified using the **`@section <name> { ... }`** block
  - the body is a special section, which is not marked with a distinct section name
  - in our views we must specify the layout we want to use with the **`Layout`** property

LAYOUT VIEWS

E.g. (**View.cshtml**):

```
@model ProductsModel
@{ Layout = "~/Views/Shared/_Layout.cshtml;
// we use a layout
}
@section mainMenu { // section defining our main menu
  foreach (String item in Model.ItemTypes)
    <div>@Html.ActionLink(…)</div>
  }
@* content not within a section is the body *@
<div id="title">List of Products</div>
...
@section mainFooter { ... } // another section
```

LAYOUT VIEWS

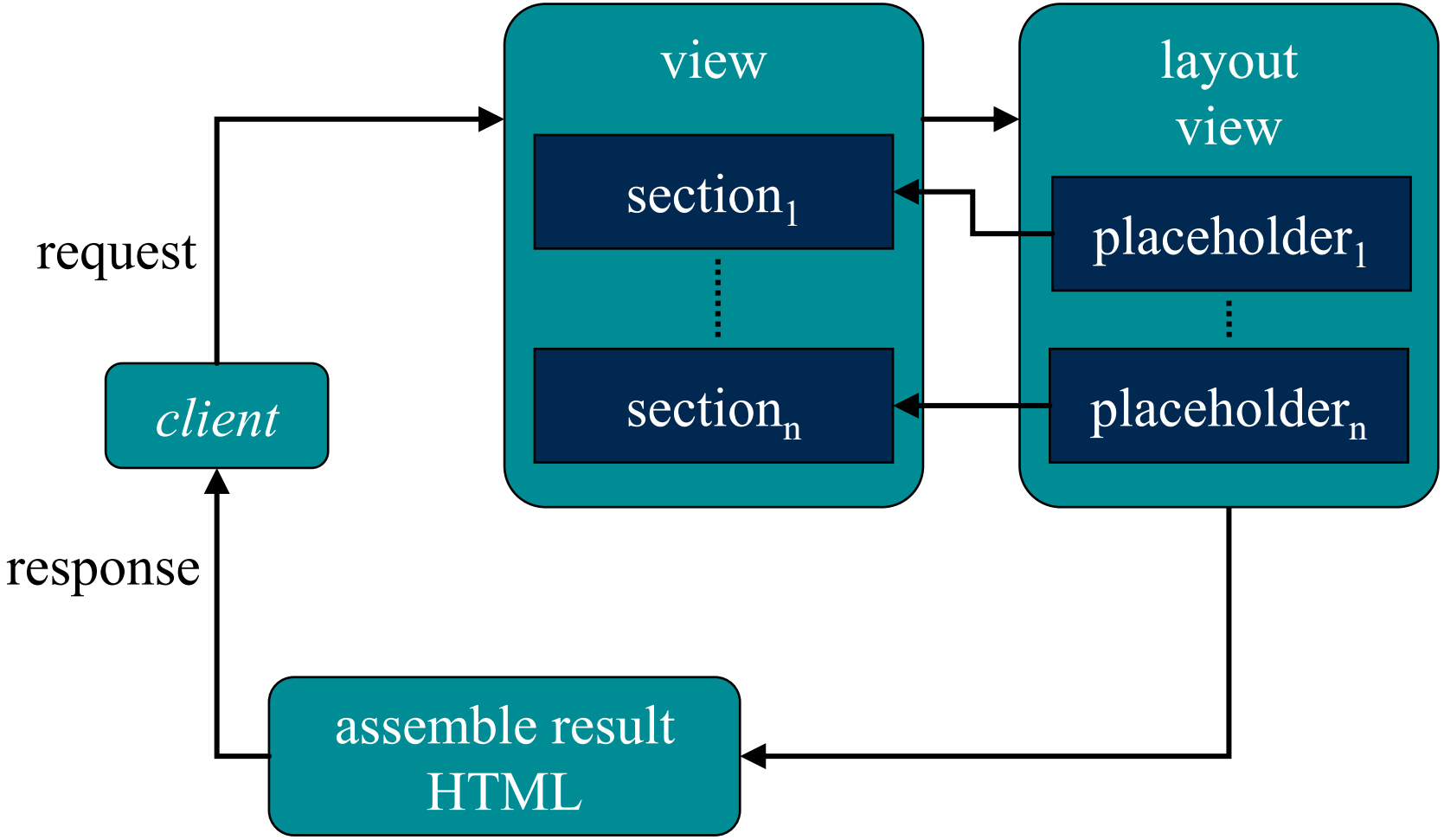
- In the layout view, the body is loaded by the **`RenderBody()`** call, the other sections by the **`RenderSection(<name>)`** call
  - if a section is not required to be defined, the false value of the **`required`** parameter can be used to indicate this;
  - the existence of the section can be checked by **`IsSectionDefined(<name>)`**.
- The file name of the layout view is conventionally started with an underscore, the default layout is **`Views/Shared/_Layout.cshtml`**
  - this does not require the **`Layout`** property;
  - this is controlled by the **`Views/_ViewStart.cshtml`** file.
- Layouts can be nested.

LAYOUT VIEWS

E.g.:

```
<body>
  ... @* other content *@
  <div id="menu">
    @RenderSection("mainMenu")
    @* load section *@
  </div>
  <div id="mainpage">
    @RenderBody() @* load body *@
  </div>
  <div id="footer">
    @RenderSection("mainFooter", false)
    @* load optional section *@
  </div>
</body>
```

LAYOUT VIEWS



LAYOUTS AND PARTIAL VIEWS

Layouts should be used when:

- you want to define the structure of your page (e.g. menu, header)
- you want to display the same elements in the same way on several pages

Partial views should be when:

- you want to use the same elements in various environments (e.g. login box), or you want to make a particular element interchangeable (e.g. table/chart)
- you want to replace or reload only a part of the page

FILE RESULT

We may send not only HTML content, but any file content (e.g. images, documents, compressed files) to the user

- the content has a type (*internet media type*, *MIME type*)
- if the browser can display the received type, it can display it, otherwise it can offer it for download
- file content can be returned using the **`File`** helper method inherited
  - the content can be defined as a binary content, a stream, or path;
  - the MIME type must be specified;
  - we can optionally specify the name of the downloaded file (then the file will be offered for download regardless of the type).

FILE RESULT

E.g.:

```
return File("/Content/Pictures/image.jpg", "image/jpg"); //
load and return a JPEG image from the file system

Byte[] imageFile = … // load an image from the database
return File(imageFile, "image/png", "image.png");
// return PNG image for download with the name image.png
```

- The returned file content can be displayed in a view with the **<img>** HTML tag using the **Url.Action** helper method to create the URL.

EXAMPLE

Task: Create a travel agency website where you can browse apartments.

- The main page (**`Index`**) lists the basic data of the buildings, which can be filtered by the cities.
- The details page (**`Details`**) lists the apartments of a building.
- The webpage is controlled by a controller (**`HomeController`**), with actions to list all buildings (**`Index`**), list the buildings of a city (**`List`**) and retrieve the details of a building (**`Details`**).
    - To list the cities, we use the **`ViewBag`** property
- Data is stored in a database (**`TravelAgencyContext`**)
    - Business logic should be decoupled from the controllers into separate service classes (**`BuildingService`**, etc.)

EXAMPLE

- Use a layout (**`_Layout.cshtml`**) for the header and the list of cities to avoid code redundancy. The **`Index`** and **`Details`** view should only contain the different page content
- The day of the week, shore type and features require special display on multiple pages, so display template partial views are defined (**`DayOfWeek`**, **`ShoreTypeViewModel`**, **`FeatureViewModel`**)
  - display the date according to the current language setting (**`CultureInfo.CurrentCulture`** property)
- Building images are returned as *file result* by two further actions:
  - one to display an image for a building (**`ImageForBuilding`**),
  - one to load a certain image in either small or large size (**`Image`**)
  - there might be no image, return a default one (**`NoImage.png`**)

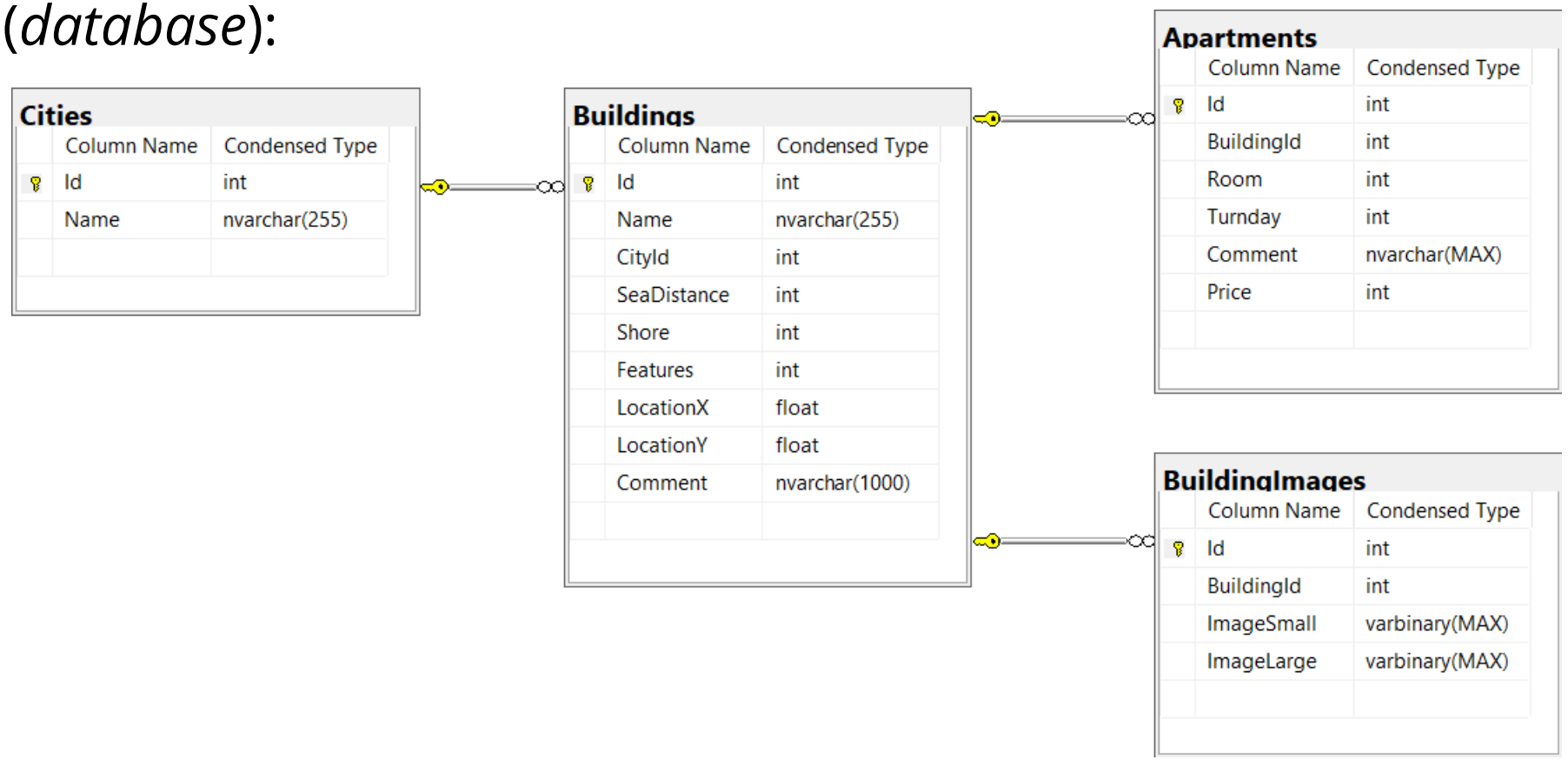
EXAMPLE

Design (*entity model*):

- we use the *Entity Framework Core* for object-relational, entity-based database management;
- four entity types will be used (**`City`**, **`Building`**, **`Apartment`** and **`BuildingImages`**);
  - the **`BuildingImage`** entity stores the images of the buildings, one larger and one smaller version of each image
- the database is created in a *code-first* manner based on the entity model (if it does not exist);
- the primary keys are generated automatically by the database on persistence.

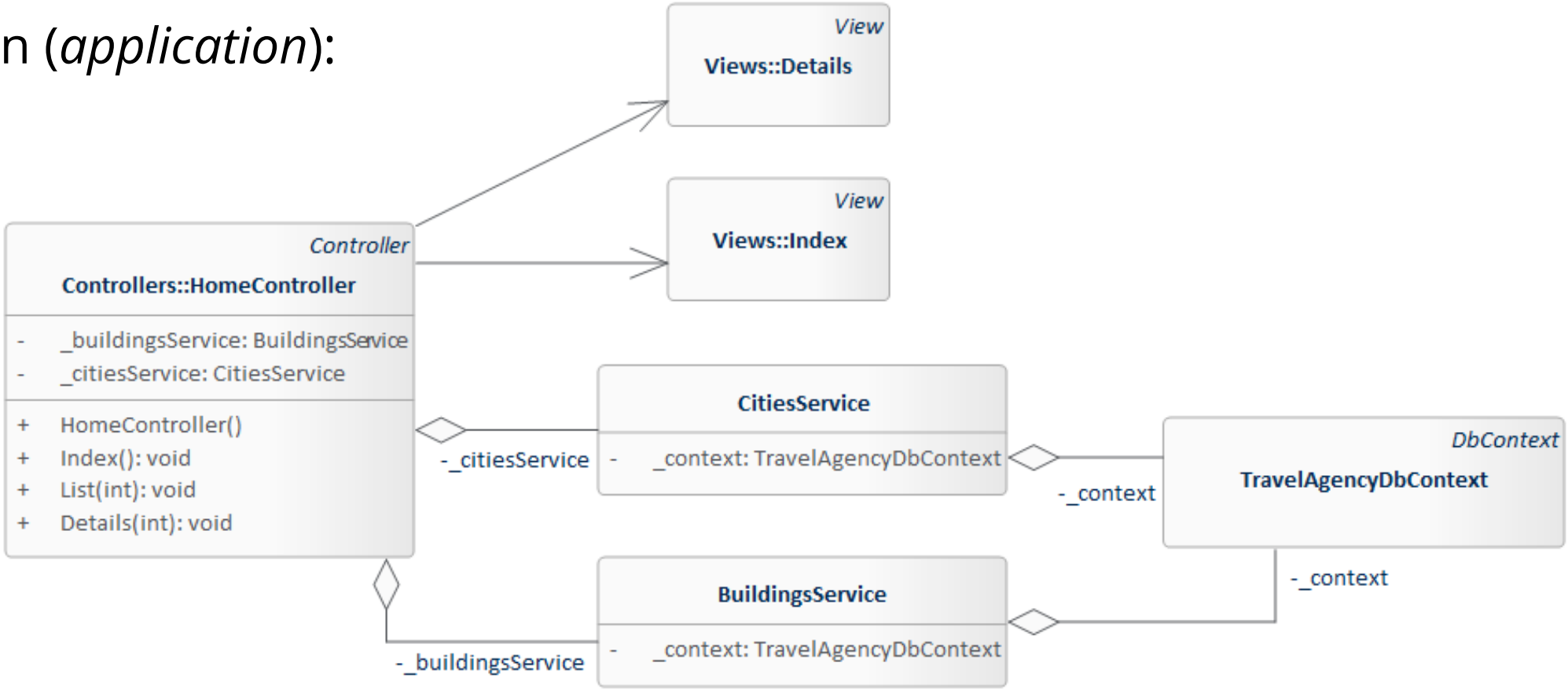
EXAMPLE

Design (*database*):



EXAMPLE

Design (*application*):



EXAMPLE

- Show the location of the buildings on a map
  - the coordinates of the buildings are stored in the buildings table (**`LocationX`**, **`LocationY`**), we just need to pass them to the map in text form (making sure to the decimal separator is a dot)
  - the Google map is displayed on the **`Details`** view, specifying the size of the map and the displayed coordinate
    - The Google Maps JavaScript API is used to display the map.
  - In the configuration (**`appsettings.json`**), we specify API key for Google Maps.
    - One can be obtained from: http://console.developers.google.com

ELTE EÖTVÖS LORÁND UNIVERSITY

DATA INPUT AND VALIDATION

Modern web application development in .NET

Lecture 3

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

FORMS

Users might be requested to enter data on websites (e.g. during a simple login), which can be done with forms

- forms are built up from controls, whose content can be sent to the server in a **`POST`** type request
- forms can be created with the **`Html.BeginForm()`** operation
  - usually placed inside a **`@using`** block, this defines their scope;
  - inside the form we use **`input`** fields can be placed and **`submit`** type **`input`** element to send the request;
  - the **`value`** attribute of the input fields specifies which values (properties) of the model are inserted

FORMS

- e.g. (viewmodel):

```
public class UserData // viewmodel
{
  public int UserId { get; set; }
  public string UserName { get; set; }
  public string UserPass { get; set; }
  public string Birthdate { get; set; }
}
```

FORMS

- e.g. (view):

```
@model UserData
...
@using (Html.BeginForm()){ // begin of form
  <div>Your name:
    <input name="userName" value="@Model.UserName" />
    @* text input field for the UserName of the viewmodel*@
  </div>
  @* … further  input fields … *@
  <input type="submit" value="Login" />
  @* form submit button *@
} // end of form
```

DATA INPUT

Forms also run an action, but pass the data entered as a viewmodel

- They run the same action that created their view by default, but this can be parameterized, e.g.:
  `@using (Html.BeginForm("Index", "Login", …))`
- In the controller, we can specify that an action should be performed only on **`GET`** or **`POST`** requests (by adding the **`HttpGet`** or **`HttpPost`** attribute), thus the two operations can be separated
  - without attributes, the two HTTP methods must be handled in one operation (as overloading is not allowed in this case)

DATA INPUT

- e.g.:

```
public class LoginController : Controller {
  [HttpGet] // executed upon simple page load
  public IActionResult Index() {
    return View(); // return the view without viewmodel
  }

  [HttpPost] // executed upon form submission
  public IActionResult Index(UserData data) {
  // the form data is retrieved as a function parameter
  …
}
```

DATA INPUT

- In a form, input fields can also be generated using operations, e.g.: **`@Html.TextBox("userName", "@Model.UserName")`**
- With a strongly typed viewmodel, input fields can also be generated for a specific property of the viewmodel, e.g.: **`@Html.TextBoxFor(m => m.UserName)`**
- The following input fields can be generated this way:
  - `TextBox`, `TextArea`, `Password`
  - `CheckBox`, `RadioButton`, `DropDownList`, `ListBox`
  - `Hidden`

DATA INPUT

- e.g.:

```
@model UserData
...
@using (Html.BeginForm()){ // begin of form
  <div>Your name:
       @Html.TextBoxFor(m => m.UserName)</div>
  <div>Your password:
       @Html.PasswordFor(m => m.UserPass)</div>
  @* generate input fields for the properties *@
  ...
  <input type="submit" value="Login" />
} // end of form
```

DATA INPUT

When the type of model properties are not known in advance, dynamically generated elements can be used:

- the **`Editor`** operation dynamically generates the input field; while
- the **`Label`** operation generates the label; and
- the **`Display`** operation displays the content in read-only mode for the specified property.

When processing each property one by one is not required, the whole viewmodel can be displayed (**`LabelForModel`**, **`DisplayForModel`**) or the editable form can be generated (**`EditorForModel`**)

- in this case it is often useful to annotate the viewmodel for proper field generation

DATA INPUT

- e.g.:

```
@model UserData // viewmodel of user data
...
@using (Html.BeginForm()) { // begin of form
  <div>@Html.LabelFor(m => m.UserName):
       @Html.EditorFor(m => m.UserName)</div>
  @* dynamically generate label and input field *@
  ...
  <div>@Html.LabelFor(m => m.Birthdate):
       @Html.EditorFor(m => m.Birthdate)</div>
  @* date picker will be displayed here*@
  <input type="submit" value="Login" />
} // end of form
```

DATA INPUT

- e.g.:

```
@model UserData // viewmodel of user data
...
@using (Html.BeginForm()) { // begin of form
  @Html.EditorForModel()
  @* make the whole viewmodel editable *@

  <input type="submit" value="Login" />
  @* we still need a submit button though*@
} // end of form
```

VIEWMODELS FOR FORMS

Our viewmodel and its properties can be *annotated* with a number of attributes that affect the dynamic generation of the form fields

- this way we have better control how data is displayed or requested from the users at generation time, e.g.:
  - content of the label (**`Display`**)
  - format of the content (**`DisplayFormat`**)
  - hide, but include content in the form (**`HiddenInput`**)
  - further specify data type (**`DataType`**),
    or the type of the input field (**`UIHint`**)

VIEWMODELS FOR FORMS

- e.g.:

```
[DisplayName("User login:")] // set label for the VM
public class UserData
{
  // hidden, do not even render as hidden field
  [HiddenInput(DisplayValue=false)]
  public int UserId { get; set; }

  [Display(Name="Your name:")] // set label
  public string UserName { get; set; }
```

VIEWMODELS FOR FORMS

- e.g.:

```
    [Display(Name="Your password:")]
    [UIHint("Password")] // set to password input field
    public string UserPass { get; set; }

    [Display(Name="Your birthday:")]
    [DisplayFormat(DataStringFormat="yy.MM.dd")]
    [DataType(DataType.Date)] // display only the date
    public DateTime Birthdate { get; set; }
}
```

DISPLAY TEMPLATES

Both for display (**`DisplayFor`**) and input (**`EditorFor`**) fields, not only built-in elements, but custom partial views can also be used.

- these are called display templates and are placed in the **`Views/Shared/DisplayTemplate`** directory
- the type of the property will be the viewmodel of the partial view
- e.g.:

```
[Display(Name="Your name:")]
[UIHint("MyNameDisplay")]
// loads the MyNameDisplay.cshtml view
public string UserName { get; set; }
```

TAG HELPERS FOR FORMS

Tag helper syntax can also be used to dynamically generate forms in ASP.NET Core, resulting in more concise codebase that is easier to read:

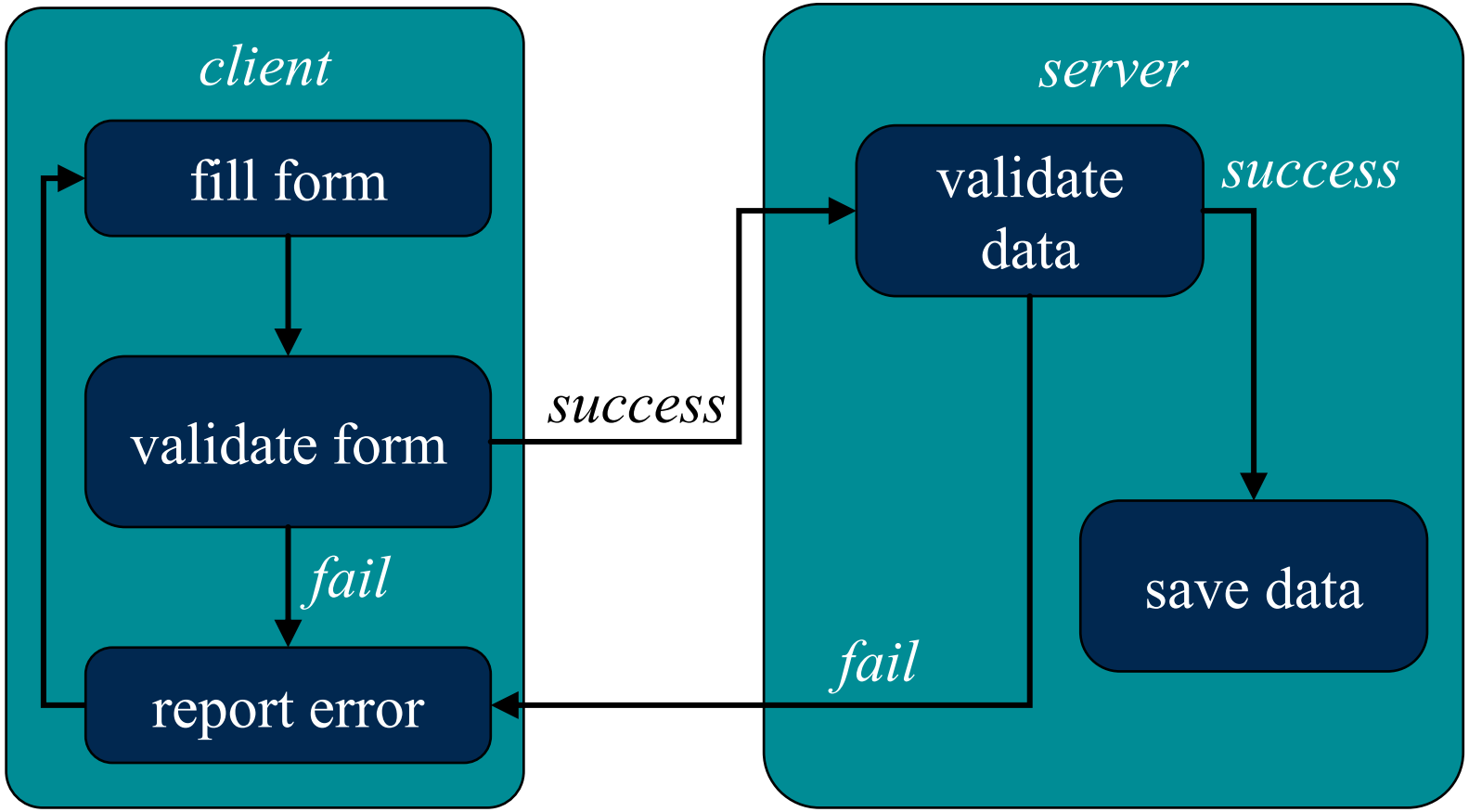
```
@* begin of form *@
<form asp-action="Results"> @* set the action URL *@
  <div><label asp-for="UserName"></label>
       <input asp-for="UserName" /></div>
  @* the label and the input is generated dynamically*@
  <div><label asp-for="UserPass"></label>
       <input asp-for="UserPass" /></div>
  <div><label asp-for="BirthDay"></label>
       <input asp-for="BirthDay" /></div>
</form>
```

VALIDATION

We always need to check the data we receive from the user, this is called the *validation of the (view)model*, which consists of the following steps:

1. On the client side, as soon as the user submits the form:
   - check that the necessary data have been provided;
   - check that the type and format are correct (e.g. e-mail, date);
   - inform the user if there are any problems.
2. When successful, the data will be sent to the server
3. The same validations are performed again on the server side, but this time including *security checks* as well
4. In case of error, report back to the client, otherwise the requested action be executed

VALIDATION


client
fill form
validate form
fail
report error
success
server
validate data
success
save data
fail

VALIDATION

Validation can be done on the server side only, or client and server side

- server-side validation is essential, especially to prevent attacks
- validation can be done completely manually or using built-in tools

On the server side, model state can be managed in the controller through the **`ModelState`** property

- the **`IsValid`** value specifies that all required values are found, has the correct type and passed the validation
- the **`AddModelError`** operation can be used to report an error

VALIDATION

- e.g.:

```
public class LoginController : Controller {
...
[HttpPost]
public IActionResult Index(UserData data) {
  if (string.IsNullOrEmpty(data.UserName))
    // name is empty
    ModelState.AddModelError(
      "UserName", "User name is required!");
    // report the error for the property
  …
  if (ModelState.IsValid)
     // when data passed validation
```

VALIDATION IN THE VIEW

Errors can be specified globally or for each property individually (in the former case, no property is specified)

In the view, you can display error messages

- for a property, the **`Html.ValidationMessageFor()`** method displays the error message indicated
- for the whole model, **`Html.ValidationSummary()`** method displays the error messages
    - it can be specified with a boolean parameter whether all errors (**`false`**) or only those for which no property is specified (**`true`**) should be displayed, e.g.: **`Html.ValidationSummary(true)`**

VALIDATION IN THE VIEW

- e.g.:

```
@using (Html.BeginForm()){
  <div> @Html.ValidationSummary(true, "Errors:")
  @* global errors *@
  </div>
  <div>
    @Html.LabelFor(m => m.UserName):
    @Html.EditorFor(m => m.UserName)
    @Html.ValidationMessageFor(m => m.UserName)
    @* errors reported for the UserName property *@
  </div>
  ...
```

VALIDATION IN THE VIEW

- *tag helpers* can be used for this purpose as well:

```
<form>
  <div asp-validation-summary="ModelOnly">
 @* global errors *@
  </div>
  <div>
    <label asp-for="UserName"></label>
    <input asp-for="UserName" />
    <span asp-validation-for="UserName"></span>
    @* errors reported for the UserName property *@
  </div>
  ...
```

- The **`asp-validation-summary`** can be set to **`All`** or **`ModelOnly`**, in the former case it also prints the errors for each property.

VALIDATION IN THE VIEWMODEL

We can specify the validation criteria directly in the viewmodel, automating the validation

- the validation criteria can be defined per property and the attached error message (**`ErrorMessage`**) can also be specified
- some common validator:
  - **`Required`**
  - **`StringLength`**, **`RegularExpression`**, **`Range`**, **`Compare`**
  - **`Url`**, **`Phone`**, **`CreditCard`**, **`EmailAddress`**
- validation can be performed manually using the **`TryValidateObject`** method of the **`Validator`** class

VALIDATION IN THE VIEWMODEL

- e.g.:

```
public class UserData {
  ...
  [Required(ErrorMessage="User name is required.")]
  [StringLength(15,
    "User name cannot be longer than 15 characters.")]
  [RegularExpression("^[a-z0-9_-]{3,15}$",
    "User name contains invalid characters.")]
  // validation criteria for the UserName
  public string UserName { get; set; }
  …
}
```

CLIENT-SIDE VALIDATION

ASP.NET MVC supports client-side validation with JavaScript

- The easiest solution is to use the *jQuery Validation* library, which can automatically handle validation annotations defined in the server-side model and apply them on the client-side.

```
...
<script src="~/lib/jquery/dist/jquery.min.js">
<script src="…/jquery.validate.js">
<script src="…/jquery.validate.unobtrusive.js">
...
```

- The validation is done before the form is submitted, on the client side (the annotations are properly embedded in the HTML controls).

CLIENT-SIDE VALIDATION

- In *production* environments, it may be useful to load JavaScript libraries from a *CDN (Content Delivery Network)*, but as a secondary solution, you can also provide a local version of the file:

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/
             jquery-2.2.0.min.js"
        asp-fallback-src=
            "~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="{hash}">
</script>
```

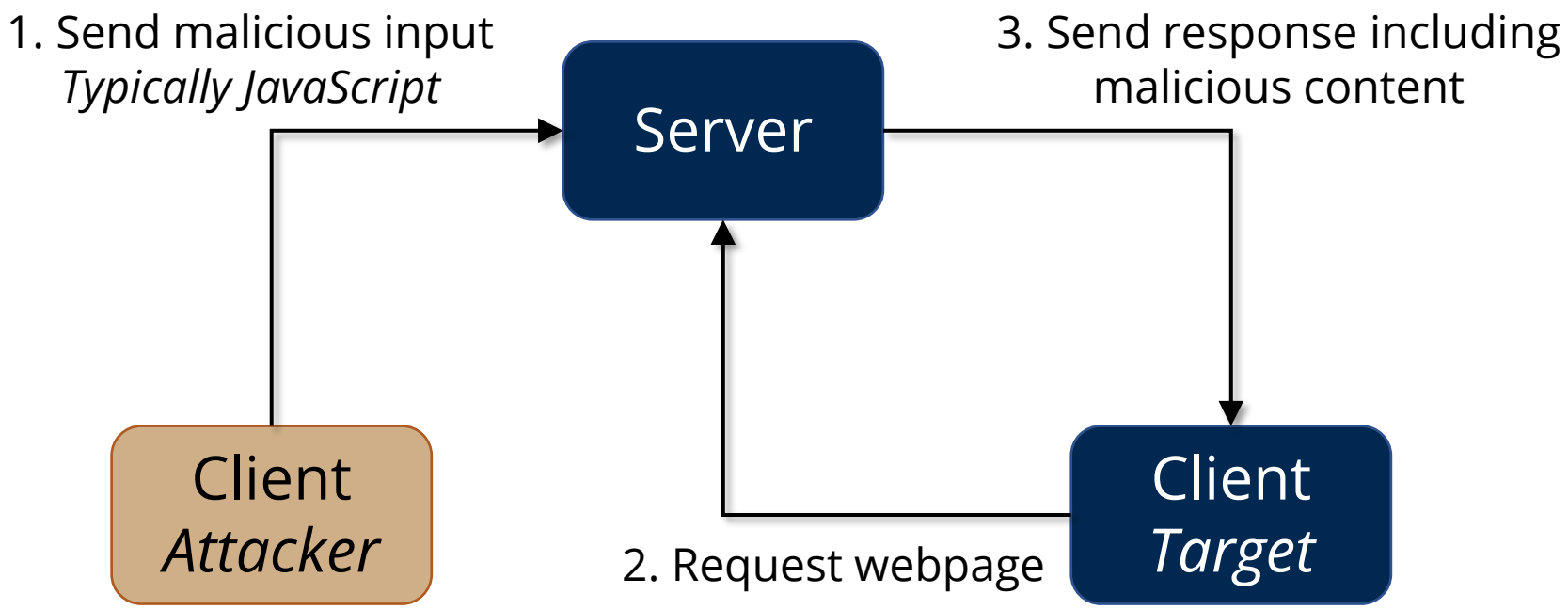
SECURITY VALIDATION

Data sent by users may contain malicious content, so validation is also crucial from a security point of view. Some of the most common attacks:

- *SQL injection:* manipulates SQL statements executed on the server
  - cannot occur when using an ORM approach like EF
- *cross-site scripting (XSS):* a client script is uploaded to the server and executed by the targeted client's browser
  - Razor encodes HTML content by default, but manual input sanitization could be required for other purposes
  - manual override is possible with the **`Html.Raw()`** and the **`Html.Encode()`** helper functions

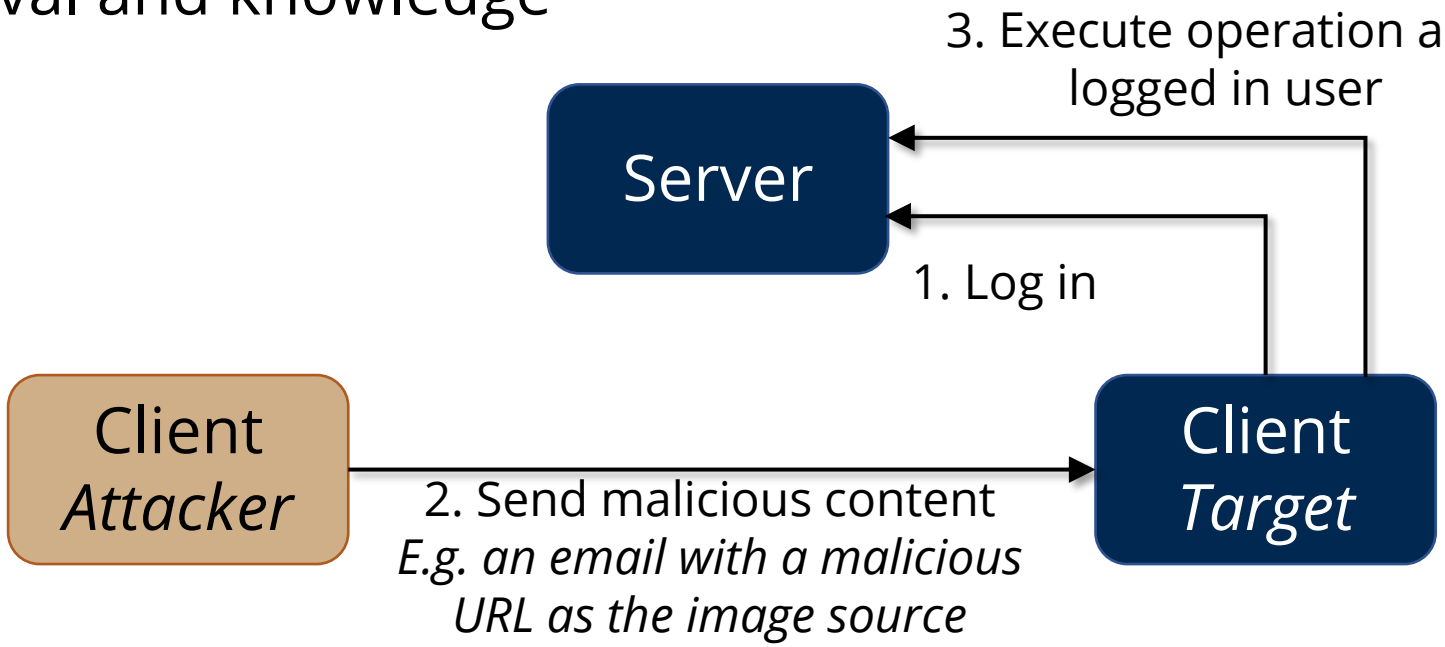
SECURITY VALIDATION

- demonstrating a *cross-site scripting (XSS)* attack:



SECURITY VALIDATION

- *cross-site request forgery (XSRF):* redirecting the user and executing a request without his/her knowledge
  - e.g. the user submits a form with content (**POST**) without his/her approval and knowledge



SECURITY VALIDATION

- How to defend against *cross-site request forgery (XSRF)* attacks on the server side?
  - an attack can be avoided by making sure that the same client filled and then sent the form
  - to achieve this, we can place a token (**`Html.AntiForgeryToken()`**) in the form, which is automatically sent back by the client
  - when the action processing the form submission is executed, we can retrieve and validate the token (manually or with the **`ValidateAntiForgeryToken`** attribute)
    - if the two values match, then there was no attack
    - in modern ASP.NET, the validation for **`POST`** requests is done automatically on default configuration

SECURITY VALIDATION

- Using either the **`Html.BeginForm()`** helper method or *tag helpers*, inclusion of *an anti-forgery token* is the default for the **`POST`** method.
  - Generated HTML content:

```
<form method="post">
...
<input name="__RequestVerificationToken"
       type="hidden" value="{token}" />
</form>
```

  - Can be disabled, if necessary, e.g. for tag helper:

```
<form method="post" asp-antiforgery="false">
```

  - Can be added manually to a form if necessary:
    `@Html.AntiForgeryToken()`

EXAMPLE

Task: Implement the booking function of the travel agency website, i.e. by selecting an apartment and entering the details of the reservation, it should be possible to book it for the given weeks.

- Add a new controller to manage the reservations (**`RentController`**)
  - the Index operation will serve **`GET`** and **`POST`** requests (the parameters are the apartment, and the data provided)
- Two views are added for the controller:
  - one containing a form to make the reservation (**`Index`**)
  - another one for the successful booking (**`Result`**), where the total amount to be paid is provided

EXAMPLE

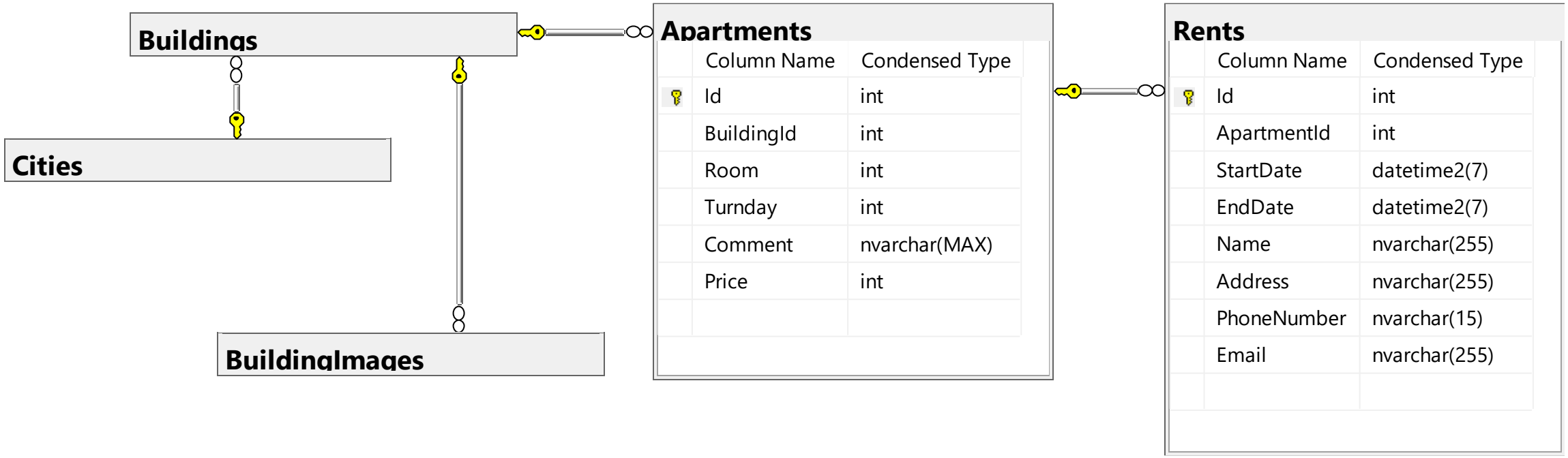
- Reservation data should be stored in a separate database table (**`Rent`**)
  - add convenience functions (e.g. collision detection for **`Rent`**)
- Create a viewmodel class for reservations (**`RentViewModel`**) to gather the booking data from the visitors
  - use annotations in the viewmodel to further specify how to display data and for data validation
- Validate all the data before saving it
  - until the form is filled in correctly, redirect the user back to the form page (and display an appropriate error message)
- Protect the website against *XSRF attacks* in the appropriate places (**`Rent/Index.cshtml`**, **`RentController`**)

EXAMPLE

- Data management business logic should be once again decoupled from the controllers into a separate service class (**`RentService`**)
  - handles and manages the persistence: data retrieval, creating new bookings, updating and deleting existing ones
  - performs mandatory data validation
- Extend the **`ApartmanService`** with a new operation to check availability (**`IsAvailable`**)

EXAMPLE

Design (*database*):



ELTE EÖTVÖS LORÁND UNIVERSITY

IMPLEMENTING WEB SERVICES

Modern web application development in .NET

Lecture 4

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

WEB SERVICES

A *web service* is a set of protocols and rules that allow platform-independent data exchange between applications over a network

- i.e. a *service provider* provides an interface of functions that can be accessed by *service consumers*
  - a consumer can be any application, e.g. a web page, a desktop application, another webservice
- a variety of protocols and solutions can be used for communication, e.g. *SOAP (Simple Object Access Protocol)*, *WSDL (Web Services Description Language)*, or *REST (Representational State Transfer)*
- allows the creation of a *Service Oriented Architecture (SOA)*

REST

REST (Representational State Transfer) is a software architecture that enables the development of scalable, high-performance distributed network applications

- communicates primarily on HTTP basis using basic HTTP methods (**GET**, **POST**, **PUT**, **DELETE**, ...)
- applications following the REST architecture are *RESTful* apps

Defines constraints to the system that must be followed:

1. client-server model,
2. uniform interface,
3. stateless communication,
4. layered architecture,
5. cacheability,
6. code on demand.

ASP.NET WEBAPI

ASP.NET Core MVC enables the development of RESTful applications

- implemented in an MVC architecture, activities are controlled by *controllers* that respond to a given request and resource
- the same **Microsoft.NET.Sdk.Web** SDK can be used for development, same as for the previous MVC web applications
- data in communicated in *JSON (Javascript Object Notation)* format by default, but the web service can automatically modify the format according to a client's request
- can be easily integrated into web pages developed in the ASP.NET Core MVC framework, a single web project can provide a classic web page and a web service interface as well

JSON

- JSON is a simple format for representing objects, e.g.:

```
{ // object
  "id": 1234, // property
  "group": "tool",
  "name": "hammer",
  "responsible": { "name" : "John" },
                  // composite property
  "materials": [ // array property
    { "name": "steel" },
    ...
  ]
}
```

CONTROLLERS

Controllers (still derived from the **`Controller`** class) implement how the HTTP action methods (**`GET`**, **`POST`**, ...) should be served

- on successful execution, the return value is placed in the body of the HTTP response, and an **`OK`** (200) response is generated
  - if there is no return value (void), a **`No Content`** (204) response is sent (without body)
- the operations are resolved according to the path mapping by the *routing engine*, typically in the following form:
  `<domain>/api/<controller>/<parameters>`
  - overloading actions is limited (notice there is no action in route)
  - in addition to the resource address, content may be placed and sent in the body of the request (**`FromBody`**)

CONTROLLERS

- Action methods should still return an **`IActionResult`**
  - since (primitive or complex) objects will be returned, an **`ObjectResult`** should be returned as an implementation
  - the **`Controller`** base class providers helper functions to achieve this, e.g. **`Ok(obj)`** will return an **`ObjectResult`** with the proper status code (200) and body
- For easier syntax action methods can have any return type in a WebAPI project and an object of that type can be returned
  - ASP.NET Core will automatically wrap it into an **`Ok()`** call
- Responses can be still be generated asynchronously (**`Task<IActionResult>`**, **`Task<T>`**)

CONTROLLERS

- e.g.:

```
[Route("api/myproducts")]
public class ProductsController : Controller {
  IList<string> products; // modell
  ...
  // GET /api/myproducts/
  [HttpGet]
  public IEnumerable<string> Get() {
    return products; // return all products
  }
  // GET /api/myproducts/1
  [HttpGet("{id}")]
  public string Get([FromRoute] int id) {
    return products[id]; // return specific product
  }
```

CONTROLLERS

- e.g.:

```
    // POST /api/myproducts/
    [HttpPost]
    public void Post([FromBody] string product) {
      // specify to load the product from the body
      products.Add(product);
    }

    // DELETE /api/myproducts/1
    [HttpDelete("{id}")]
    public void Delete([FromRoute] int id) {
      products.RemoveAt(id);
    }
  }
```

ROUTING

The ASP.NET Core WebAPI application must be properly configured with path resolution rules before usage

- routing is configured usually not globally like for MVC web pages, but at the controller and action level with *routing attributes*
- e.g.:

```
[Route("api/myproducts")]
public class ProductsController : Controller {
  // GET /api/myproducts/1
  [HttpGet("{id}")]
  public string Get(int id) {
    // …
  }
}
```

DATA TRANSFER

Web service operations can communicate through not only primitive types, but also with complex types, *Data Transfer Objects (DTOs)*

- a DTO can be any object that is serializable (in the expected format), i.e. mappable to a structure of primitive values
- its structure must be chosen in such a way that it does not contain internal data to the web service, or unnecessary or circular references (e.g. in the case of an entity object)
- DTO serialization is mostly done automatically by ASP.NET Core
  - we may return a uniquely configured HTTP message (**`ContentResult`**) when needed

DATA TRANSFER

- e.g.:

```
public class ProductDto { // DTO type
  public int Id { get; set; }
  public string Name { get; set; }
}

[Route("api/myproducts")]
public class ProductsController : Controller {
  // GET /api/myproducts/
  [HttpGet]
  public IEnumerable<ProductDto> Get() {
    return products; // return all products
  }
  ...
}
```

DATA TRANSFER

- e.g.:

```
[Route("api/myproducts")]
public class ProductsController : Controller {
  ...
  // GET /api/products/
  public IActionResult Get() {
    return new ContentResult() {
      // custom response, set the status code and the content
      StatusCode = HttpStatusCode.OK,
      Content = JsonConvert.SerializeObject(products);
                // string
    };
  }
}
```

ROUTING

Mapping HTTP methods to controller actions be done

- automatically, based on the prefix of the action's name, e.g.:

```
[Route("api/myproducts")]
public class ProductsController : Controller {
  ...
  // GET /api/myproducts
  public ProductDto GetAllProduct() { ... }
  ...
  // DELETE /api/myproducts
  public void DeleteAllProducts() { ... }
  ...
}
```

ROUTING

- manually, applying attributes to specify the path (**`Route`**) or even the wanted HTTP method (**`HttpGet`**, **`HttpPost`**, **`HttpDelete`**, …), e.g.:

```
public class ProductsController : Controller {
  ...
  // GET /api/myproducts/
  [Route("api/myproducts/{id}")]
  [HttpGet]
  public ProductDto FindProduct(int id) { ... }
  ...
}
```

ROUTING

- routing rules defined on the controller and on the action method can also be combined:

```
[Route("api/myproducts")]
public class ProductsController : Controller {
  ...
  // GET /api/myproducts/1
  [HttpGet("{id}")]
  public ProductDto FindProduct(int id) { ... }
  ...
}
```

- usage of controller classes must be registered in the **`Program`** class: **`builder.Services.AddControllers()`**
  - it was similar for web pages, where views were also registered: **`builder.Services.AddControllersWithViews()`**

ROUTING

When constructing the routing rules, it is supported to:

- add a prefix at the controller level (**`Route`**)
- delimit the parameters in any preferred way (by adding additional route components)
- add constraints to parameters based on their type (bool, **`datetime`**, **`decimal`**, **`double`**, **`float`**, **`guid`**, **`int`**, **`long`**), length (**`length`**, **`minlength`**, **`maxlength`**), value (**`min`**, **`max`**, **`range`**, **`values`**), or format (**`alpha`**, **`regex`**)
- specify the priority (**`Order`**) if the route matches multiple methods

Parameter type specification allows overloading, as the system can run the operation corresponding to the type.

ROUTING

- e.g.:

```
[Route("api/myproducts")] // path prefix
public class ProductsController : Controller {
  ...
  // GET /api/myproducts/1
  [Route("{id:int:min(1)}")]
  public ProductDto GetProduct(int id) { ... }

  // GET /api/myproducts/tools/item/1
  [Route("{group:values(tools|machines)}/
           item/{id:int:min(1)}")]
  public ProductDto GetProduct(string group, int id) { ... }
}
```

ERROR HANDLING

Controllers can respond to requests not only with the default **`OK (200)`**, but also with an arbitrary HTTP code (the generic return **`IActionResult`** must be used as return type)

- inherited helper functions can be used in an action method to easily construct the wanted result: **`Ok(<content>)`**, **`Created(<location>, <content>)`**, **`Redirect(<location>)`**, **`NotFound()`**, **`Forbidden()`**, **`Unauthorized()`**, **`BadRequest(<message>)`**, **`Conflict()`**, **`InternalServerError(<exception>)`**
- proper feedback is also important for error handling
  - uncaught exceptions result in an **`INTERNAL SERVER ERROR (500)`** returned by the service

ERROR HANDLING

- e.g.:

```
public IActionResult GetProduct(int id)
{
  try {
    ...
    return Ok(product);
    // in case the product was found, return code 200
  }
  catch {
    return NotFound(); // otherwise return code 404
  }
}
```

API CONTROLLERS

Controller classes for web services should be annotated with the **`ApiController`** attribute, which adds a range configurations for the controller and its actions. The most important are:

- *routing rules* can only be specified with the **`Routing`** attribute, rules specified with **`app.UseEndpoints()`** in **`Program`** class are not applied
  - same as calling **`app.MapControllers()`** in **`Program`** class
- automatically validate the viewmodel of the actions and return an error message on failure - 400 (*Bad request*)
  - as if all actions include the following check:

```
if (!ModelState.IsValid) {
  return BadRequest(ModelState);
}
```

API CONTROLLERS

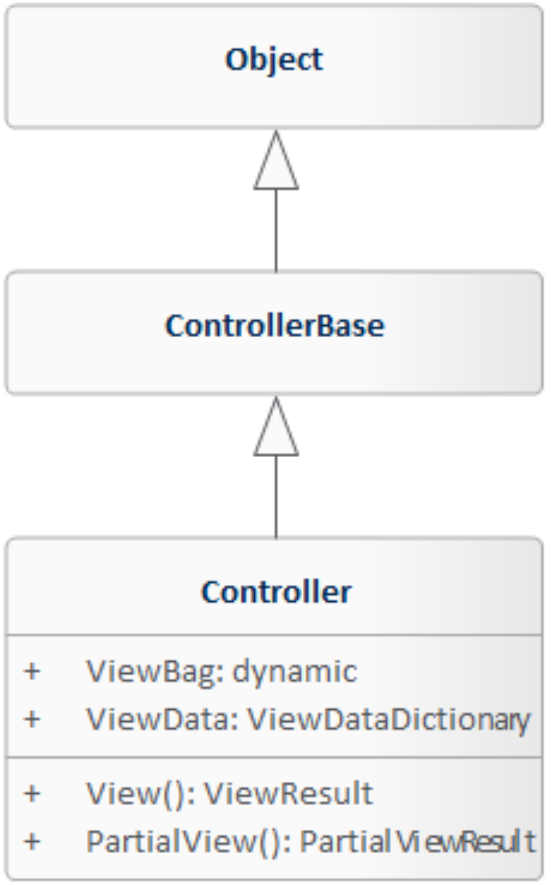
- implicitly define the source of parameters for actions, as follows:
  - **`[FromBody]`** for all complex types, with some exceptions (e.g. **`IFormCollection`**, **`CancellationToken`**)
  - **`[FromForm]`** for all parameters of types **`IFormFile`** and **`IFormFileCollection`**.
  - **`[FromRoute]`** for all parameters defined with the same name as in the routing rule.
  - **`[FromQuery]`** for all additional parameters.
- e.g.:

```
[ApiController]
[Route("api/[controller]")] // [controller] is a placeholder
public class ProductsController : Controller
```

CONTROLLER BASE CLASS

The API controller classes can be derived from the **`ControllerBase`** class instead of the **`Controller`** class.

- The **`ControllerBase`** class is the ancestor of the **`Controller`** class, and has all the necessary functionality except for support for views.
- If a single controller class serves both views and API requests, it is best to derive it from the **`Controller`** class.



CRUD

In many cases the web service provides basic data processing operations, the *CRUD* operations: *Create, Read, Update, Delete*

- these operations have HTTP equivalents (create: **`POST`**, read: **`GET`**, modify: **`PUT`**, delete: **`DELETE`**)
  - upon a successful operation the response code is **`CREATED`** (201) for creation, **`OK`** (200) or **`NO CONTENT`** (204) for other operations
    - with the **`CREATED`** result the created resource (object) and its location (**`GET`** path) must be returned as well
  - when the operation is not executed immediately, the status **`ACCEPTED`** (202) may be indicated
- in a RESTful service, the operations must conform to this scheme

OPENAPI SPECIFICATION

The *OpenAPI specification* (formerly known as *Swagger specification*) is an open format which defines the interface of a web service in JSON or YAML data serialization language, including:

- endpoints (and supported HTTP methods);
- input and output parameters;
- authentication methods used; and
- other meta information (e.g. license).

A formal specification describing a web service in the OpenAPI standard provides a unified interface that can be used by service providers and consumers to implement it.

SWAGGER

Swagger as a company provides implementation tools (some free, some commercial) for OpenAPI. Its main tools are:

- *Swagger Editor*: web editor to simplify editing an OpenAPI specification
- *Swagger UI*: web API testing interface for a developed web service.
- *Swagger Codegen*: code generator to build a skeleton implementation containing e.g. the procedure stubs, based on the OpenAPI specification of the web service. Supports a variety of languages.

Tools from other companies and also available, e.g. *OpenAPI Generator*, *Redoc(ly)*, *NSwag*.

SWAGGER UI

For web application built in ASP.NET Core MVC framework, the OpenAPI specification and a Swagger UI tester interface can be easily generated.

- Multiple NuGet packages are available, most widely used are **`Swashbuckle.AspNetCore`** and **`NSwag`**.
- The default project template in VS uses the **`Swashbuckle.AspNetCore`** package, and automatically registers the necessary generator in the **`Program`** class on the **`WebApplicationBuilder`** object:

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c => {
  c.SwaggerDoc("v1", new OpenApiInfo {
    Title = "Product Manager", Version = "v1"});
});
```

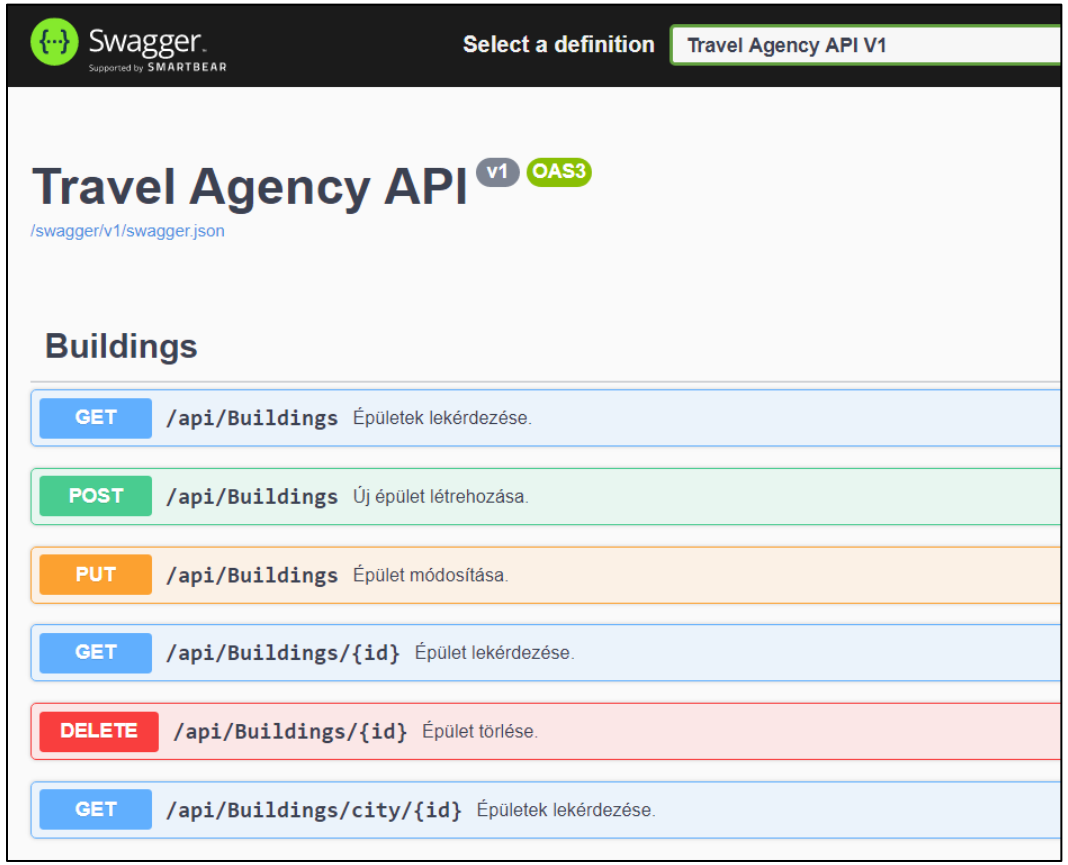
SWAGGER UI

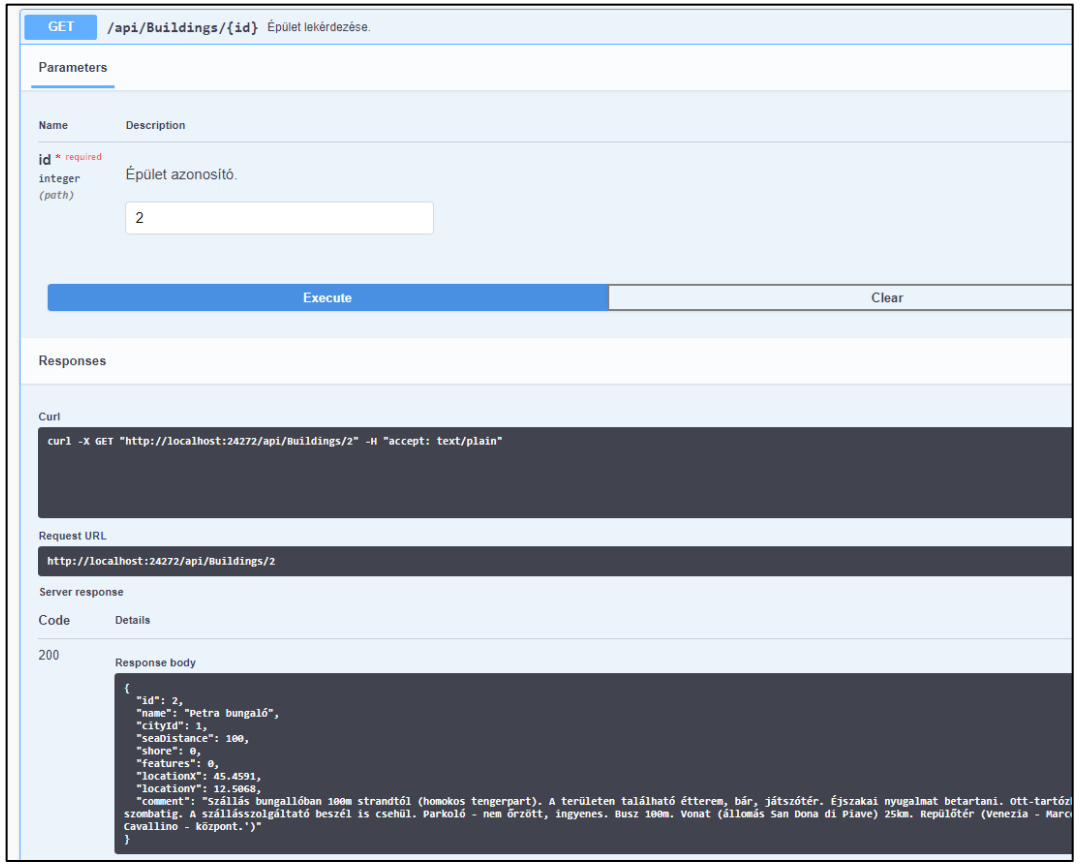
- Then Swagger can be enabled on the **`WebApplication`** object in the Program class:

```
// use Swagger (JSON endpoints)
app.UseSwagger();
// use Swagger UI (HTML endpoints)
app.UseSwaggerUI(c => {
  // configure the JSON endpoint (optional)
  c.SwaggerEndpoint(
    "/swagger/v1/swagger.json",
    "Product Manager API V1");
});
```

- Generated URLs:
  http://<domain>/swagger/index.html
  http://<domain>/swagger/v1/swagger.json

SWAGGER UI


Swagger
Supported by SMARTBEAR
Select a definition
Travel Agency API V1
Travel Agency API
v1
OAS3
/swagger/v1/swagger.json
Buildings
GET /api/Buildings Épületek lekérdezése.
POST /api/Buildings Új épület létrehozása.
PUT /api/Buildings Épület módosítása.
GET /api/Buildings/{id} Épület lekérdezése.
DELETE /api/Buildings/{id} Épület törlése.
GET /api/Buildings/city/{id} Épületek lekérdezése.


GET /api/Buildings/{id} Épület lekérdezése.
Parameters
Name
Description
id * required
integer
(path)
Épület azonosító.
2
Execute
Clear
Responses
Curl
curl -X GET "http://localhost:24272/api/Buildings/2" -H "accept: text/plain"
Request URL
http://localhost:24272/api/Buildings/2
Server response
Code
Details
200
Response body
{
"id": 2,
"name": "Petra bungaló",
"cityId": 1,
"seaDistance": 100,
"shore": 0,
"features": 0,
"locationX": 45.4591,
"locationY": 12.5068,
"comment": "Szállás bungallóban 100m strandtól (homokos tengerpart). A területen található étterem, bár, játszótér. Éjszakai nyugalmat betartani. Ott-tartózk
szombatig. A szállásszolgáltató beszél is csehül. Parkoló - nem őrzött, ingyenes. Busz 100m. Vonat (állomás San Dona di Piave) 25km. Repülőtér (Venezia - Marc
Cavallino - központ.')"
}

EXAMPLE

Task: Implement a web service as a backend for a decoupled frontend for the travel agency website.

- Implement the solution as an ASP.NET Core web API service (**`TravelAgency.WebAPI`**)
  - a decoupled *Angular* frontend will be developer later on
- Create separate types for data transfer (**`BuildingDTO`**, etc.) and place them in a separate class library (**`TravelAgency.Shared`**)
  - they can be easily shared with a C#-based client desktop project later, reducing redundancy and simplifying further development
- Generate a *Swagger UI* web interface for the service to support (manual) testing

EXAMPLE

- Add new WebAPI controllers to fetch cities (**`CitiesController`**), buildings and their images (**`BuildingsContoller`**) and create new reservations (**`RentsController`**)
  - annotate all controllers with the **`ApiController`** attribute for conventional routing management and automatic data validation
  - action methods should return objects, which will be serialized according to the configuration (JSON by default)
    - annotate methods with the **`Route`**, **`HttpGet`**, **`HttpPost`** etc. attributes to set up routing rules and allowed HTTP methods
    - return appropriate error codes in case of errors

ELTE EÖTVÖS LORÁND UNIVERSITY

USING WEB SERVICES

Modern web application development in .NET

Lecture 5

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

WEB SERVICES

Web services enable platform-independent data exchange between applications built on possibly different technologies

- the most common model is *REST (Representational State Transfer)*, which uses the HTTP protocol for communication
- the service can be implemented as an ASP.NET Core MVC application, the client can be implemented in an arbitrary language
- operations can communicate through not only primitive types, but also complex *Data Transfer Objects (DTOs)*
- *JSON (JavaScript Object Notation)* is the most common format for object-oriented data transfer, as it is easy to interpret both by machines and humans

.NET CLIENTS

ASP.NET Core web services can be accessed as a consumer using the classes in the **`System.Net.Http`** namespace

The **`HttpClient`** type provides the connection to HTTP-based services

- since network communication can be time consuming, it provides asynchronous functions to execute HTTP commands (**`GetAsync`**, **`PutAsync`**, etc.)
- the result of these method calls contains the response (**`HttpResponseMessage`**)
- when the operation is successful (**`IsSuccessStatusCode`**), the content of the response can be processed (**`Content`** property)

.NET CLIENTS

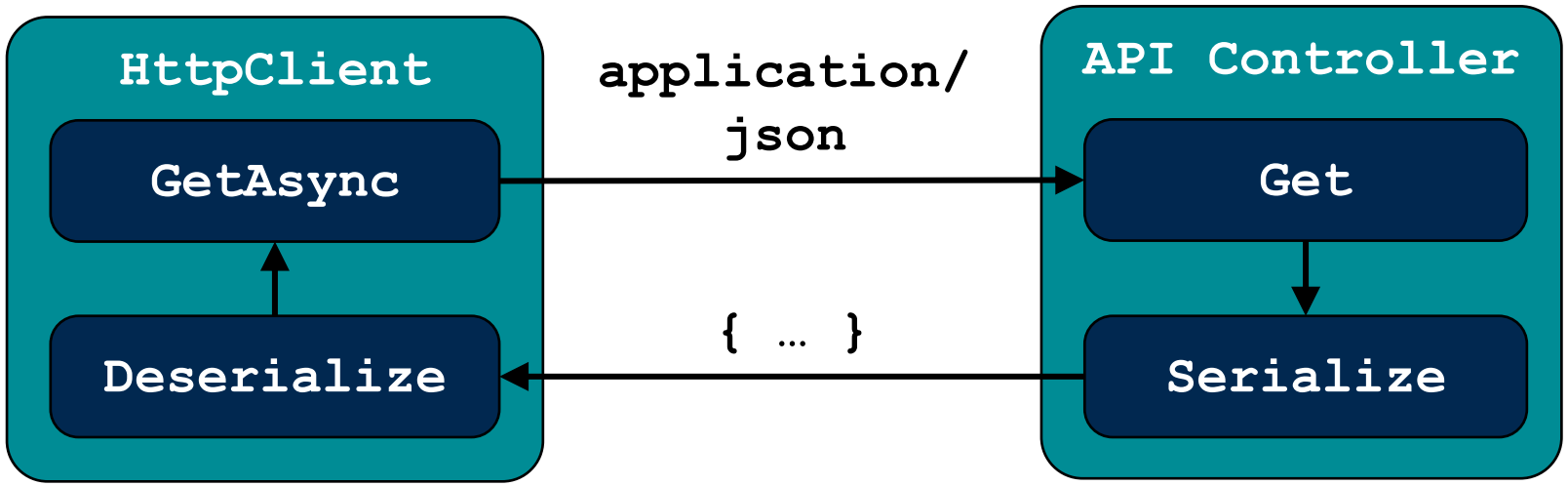
- e.g.:

```
using (HttpClient client = new HttpClient())
// instantiate new client
{
  HttpResponseMessage response =
    await client.GetAsync("http://…");
    // execute asynchronous query
  if (response.IsSuccessStatusCode)
  {
    HttpContent content = response.Content;
    ... // process content
  }
} // destroy client
```

DATA MANAGEMENT IN .NET CLIENTS

Content is managed in an object-oriented way, the format of the data transfer is controlled by the framework

- the way the data transfer object is *serialized* and *deserialized* is done according to the HTTP header information



DATA MANAGEMENT IN .NET CLIENTS

- this requires the data transfer objects to have the same type on both the client and server side – *or at least compatible*
  - best practice is to reference them from the same class library
- reading the response, only the data type needs to be specified, e.g.:

```
if (response.IsSuccessStatusCode) {
  Product myProduct = await
    response.Content.ReadAsAsync<ProductDto>();
  ... // process content
}
```

- we also have the option to handle the received content as text (**ReadAsStringAsync**), binary (**ReadAsByteArrayAsync**) or stream (**ReadAsStreamAsync**) data

DATA MANAGEMENT IN .NET CLIENTS

- serialization can be customized through the specialization of the abstract **`MediaTypeFormatter`** base type, so that specific message formats can be created
- when content is also sent (e.g. **`POST`**, **`PUT`**),
  - its format can be specified with the proper formatter, e.g.:
    ```
    client.PostAsync("http://...", product,
      new JsonMediaTypeFormatter());
      // serialize the data to JSON
    ```
  - alternatively, extension methods (provided by the *Microsoft.AspNet.WebApi.Client* NuGet package), can be used, e.g.:
    ```
    client.PostAsJsonAsync("http://...", product);
    // serialize the data to JSON
    ```

DATA MANAGEMENT IN .NET CLIENTS

Key configuration available on the **`HttpClient`** object:

- the prefix for all API calls (**`BaseAddress`**),
- the timeout for communication (**`Timeout`**),
- the default headers for each message sent (**`DefaultRequestHeaders`**), and within that
  - the content format (**`DefaultRequestHeaders.Accept`**), e.g:

```
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(new
  MediaTypeWithQualityHeaderValue("application/json"));
  // the client will only accept JSON format
```

JAVA CLIENTS

ASP.NET Core web services can be accessed from Java applications using HTTP client libraries. Common options include:

- *HttpClient* (requires only Java 11+ standard library)
- *OkHttp* library
- Spring's *WebClient* or *RestTemplate* implementation

These solutions supports sending various HTTP operations, JSON serialization/deserialization, and asynchronous requests.

JSON is typically mapped to Java objects using libraries such as *Jackson* or *Google Gson*.

JAVA CLIENTS

Example:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://..."))
    .GET()
    .build();

HttpResponse<String> response =
    client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());
```

JAVA CLIENTS

Combining with the Jackson JSON serializer in Java:

```
public class ProductDto {
  public int id;
  public String name;
  public double price;
}
```

```
ObjectMapper mapper = new ObjectMapper();
ProductDto[] products = mapper.readValue(
    response.body(), ProductDto[].class);

for (ProductDto p : products) {
    System.out.println(p.getName());
}
```

JAVASCRIPT CLIENTS

REST APIs can easily be accessed from JavaScript applications.

The most common approach is using the *Fetch API* (a standard browser API), that features:

- asynchronous calls using *Promises*;
- JSON responses can be easily parsed;
- widely supported in browsers without external library.

Some alternatives:

- *axios* library
- AJAX requests (mostly in legacy applications)

JAVASCRIPT CLIENTS – FETCH API

Example for a *GET* request:

```
fetch("https://...")
  .then(response => response.json())
  .then(data => {
    console.log(data);
  })
  .catch(error => console.error(error));
```

JAVASCRIPT CLIENTS – FETCH API

Since 2017, JavaScript supports a similar *async / await* paradigm like C#, that makes working with *Promises* easier and more straightforward:

```
async function loadProducts() {
  const response = await fetch("/api/products");
  const data = await response.json();
  console.log(data);
}
```

JAVASCRIPT CLIENTS – FETCH API

Example for a *POST* request:

```
fetch("https://...", {
  method: "POST",
  headers: {
    "Content-Type": "application/json„
  },
  body: JSON.stringify({
    name: "Keyboard",
    price: 50
  })
});
```

TYPESCRIPT CLIENTS

TypeScript adds optional static typing to JavaScript. This is general provides better maintainability and scalability for large applications. Advantages when consuming REST APIs:

- typed DTO objects
- better IDE support
- compile-time error checking

TypeScript can be used in frameworks such as *React*, *Angular* or *Vue*.

TYPESCRIPT CLIENTS – FETCH API

Example for a GET request:

```
export interface ProductDto {
  id: number;
  name: string;
  price: number;
}

async function getProducts(): Promise<ProductDto[]> {
  const response = await fetch("https://...");
  const data: ProductDto[] = await response.json();
  return data;
}
```

TYPESCRIPT CLIENTS – REACT

React example with `useEffect`:

```
import { useEffect, useState } from "react";
function ProductList() {
  const [products, setProducts] = useState<ProductDto[]>([]);
  useEffect(() => {
    fetch("https://...")
      .then(res => res.json())
      .then(data => setProducts(data)); }, []);
  return (
    <ul>
    {products.map(p => <li key={p.id}>{p.name}</li> )}
    </ul>
  );
}
```

TYPESCRIPT CLIENTS – ANGULAR

Angular provides a built-in or **`HttpClient`** (importable from **`@angular/common/http`**) for network management.

Features:

- typed HTTP requests;
- observable-based asynchronous API;
- automatic JSON parsing.

Sample Angular service:

```
@Injectable()
export class ProductService {
  constructor(private http: HttpClient){}

  getProducts(): Observable<ProductDto[]>
  {
    return this.http.get<Product[]>(
      "https://..."
    );
  }
}
```

TYPESCRIPT CLIENTS – ANGULAR

Angular component example:

```
export class ProductComponent {
  products: Product[] = [];

  constructor(private service: ProductService) {}

  ngOnInit() {
    this.service.getProducts()
      .subscribe(data => this.products = data);
  }
}
```

PLATFORM-INDEPENDENT USAGE OF REST APIS

REST APIs are platform and language independent.

- They rely on widely used web technologies:
  - communication via HTTP;
  - data exchange typically using JSON.
- Because of this, any system capable of sending HTTP requests and processing JSON responses can use the API.
  - REST is often used among service applications, e.g. microservices.
- This makes REST APIs suitable for integrating heterogeneous systems.

DATA SYNCHRONIZATION MODELS

Client-side data management can be implemented in two ways:

- *synchronous:* client and server state are always the same, each change is immediately sent to the server
- *asynchronous:* client and server state are different and can be synchronized manually (save, load, update)

Asynchronous data management is advantageous if you want to save your changes in groups rather than individually

- to achieve this, changes have to be tracked on the client side with *status flags* and indicate what changes have been made to the data (new, modified, deleted)

EXAMPLE

Task: Implement an Angular frontend for the travel agency webservice.

- The backend was already developed in the previous lecture, and is ready to be uses as a *RESTful* service
  - communication is done through *DTO* types (defined in **`TravelAgency.Shared`**), which has to be recreated in TypeScript
- The new frontend should provide the very same functionality, as the previous ASP.NET Core MVC based classic website
- Use the *Material Design* of Angular to design a responsive, modern, user-friendly layout

EXAMPLE

To try out the application, both the backend (**`Travalegency.WebAPI`**) and the frontend (**`TravelAgency.Angular`**) has to be executed.

- You may run both from Visual Studio. Alternatively, to start the Angular project from CLI on an Angular development server:
  ```
  npm install
  npm start
  ```
- During development, *CORS* issues are resolved with a frontend development proxy (see **`proxy.config.json`**), that configures to proxy all requests to *http://localhost:4200/api* (notice the frontend port) to *https://localhost:7143* (this is the backend port).
- In production, the backend and frontend are typically served on the same port, or the backend is configured with proper CORS policies.

ELTE EÖTVÖS LORÁND UNIVERSITY

AUTHENTICATION AND AUTHORIZATION

Modern web application development in .NET

Lecture 6

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

AUTHENTICATION AND AUTHORIZATION

Authentication

Who are you?

Verify the true identity of the user when accessing endpoints

Authorization

What can you do?

Check whether the user is authorized to perform the requested operation

AUTHENTICATION

User management on a website or in a webservice has various elements that need to be implemented securely:

- *registration, data modification*
- *login, logout, automatic login*
- *forgotten password/username management*
  - as the password is always stored in an encrypted form, in case of a forgotten password, the user must be provided with a new password (if we are sure of its identity)
- *extra functionalities*: user groups, data visibility management, email confirmation, etc.

ASP.NET CORE IDENTITY

Due to the complexity of user authentication, ASP.NET Core provides us its own framework for it, named *ASP.NET Core Identity*

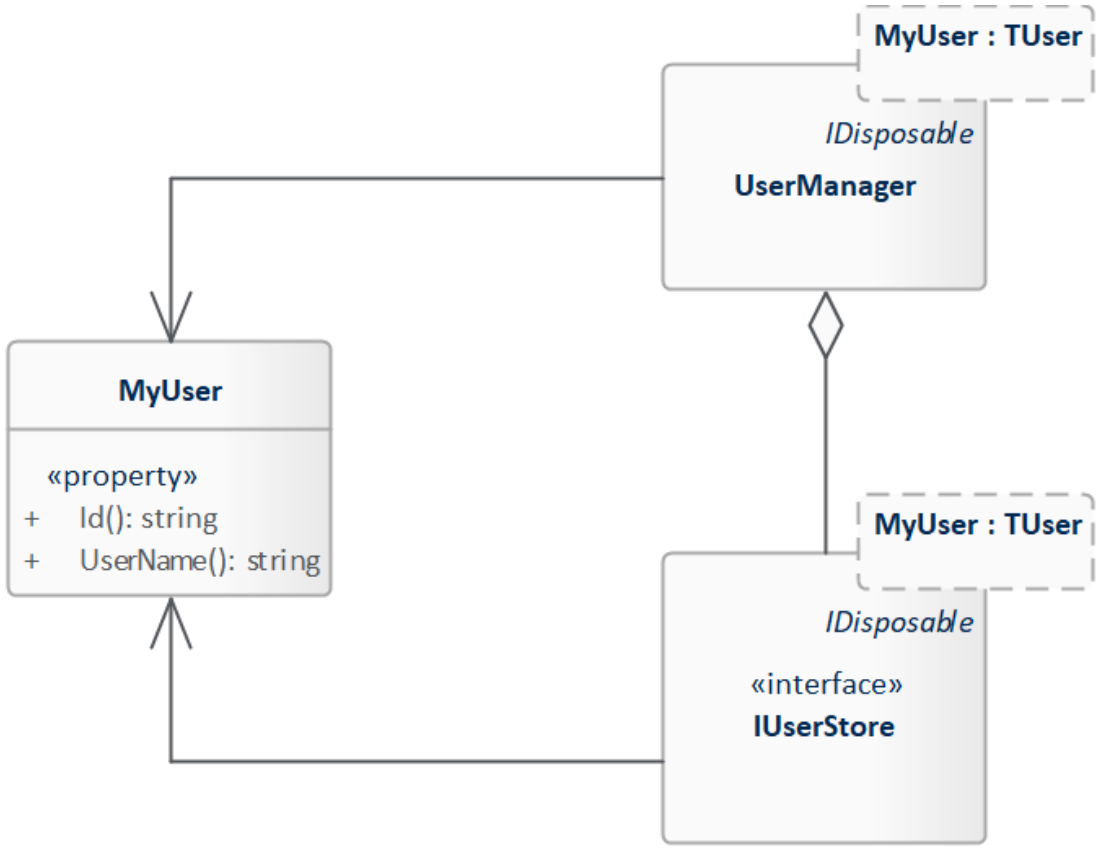
- provides the required functionality for user management, from data management to user interface
- provides multiple options for storing/accessing user data
  - we can use internal solutions (e.g. custom database, *Windows Authentication*), or external services (e.g. *Microsoft*, *Google*, *X (Twitter)*, *Facebook*), *OAuth* and *OpenId* are supported in general.
- The basic functionality is available from the **`Microsoft.AspNetCore.Identity`** library (*NuGet* package), with data storage provided by additional packages.

USER MANAGEMENT

User management is done by the generic **`UserManager<T>`** class, which can handle any specific user type and its operations support asynchronous execution

- the user can be registered (**`CreateAsync`**), identified (**`FindByIdAsync`**, **`FindByEmailAsync`**, etc.), modified (**`ChangePasswordAsync`**), etc.
- the user can be of any type, represented by the required information (password, email address, etc.)
- the manager class decouples how to store users, it stores the users through a repository (**`IUserStore<T>`**)
    - the user store defines the specific data management solution
- each class is specialized for the type of the user

USER MANAGEMENT



USER MANAGEMENT

- e.g.:

```
public class MyUser { … } // type of the user
public class MyStore : IUserStore<MyUser> { … }
// type of the user store, specialized to the user type

UserManager<MyUser> manager =
  new UserManager<MyUser>(new MyStore(), …);
  // user manager, specialized to the user type

MyUser user = await manager.FindByNameAsync(name);
// find user (by name)
```

USER STORAGE IN EF

The easiest way to store user data is to manage it through a local database with an ORM approach, using *Entity Framework* (add the **`Microsoft.AspNetCore.Identity.EntityFrameworkCore`** package)

- the entity model (**`IdentityDbContext`**) can handle users, identified by a username and password (**`IdentityUser`**)
  - the **`IdentityUser`** can be specialized through inheriting from it, any property can be added to the user
  - the database scheme is generated based on the data provided in the code (*code first* approach)
  - provides password encryption (with time-dependent salting)
- the entity model can be used with the provided **`UserStore`** class.

USER STORAGE IN EF

- e.g.:

```
IndentityDbContext<IdentityUser> context = new …;
// database entity model
UserStore<IdentityUser> store =
  new UserStore<IdentityUser>(context);
  // user data store
UserManager<IdentityUser> userManager =
  new UserManager<IdentiyUser>(store);
  // user manager

IdentityUser user = await userManager
                          .FindByNameAsync(name);
                        // find user
```

IDENTIFYING USERS

User login and identity is managed by the **`SignInManager<T>`** class

- is also specialized for the user type as a generic parameter, and provides asynchronous operations;
- direct login based on username-password pairs is handled by **`PasswordSignInAsync()`**, and logout by **`SignOutAsync()`** operations
  - can be specified to persist the login (i.e. remember the data) and lockout the user in case of incorrect data
- supports authentication via external authentication providers (**`ExternalLoginSignInAsync()`**) and two-factor authentication (**`TwoFactorAuthenticatorSignInAsync()`**)

IDENTIFYING USERS

- e.g. (Login):

```
public async Task<IActionResult> Login(…) {
  // ...
  var result = await signInManager.PasswordSignInAsync(
                          userName, password, remember, false);
  // login
  if(result.Succeeded) { … }
}
```

- e.g. (Logout):

```
public async Task<IActionResult> Logout() {
  await signInManager.SignOutAsync();
}
```

AUTHENTICATION MIDDLEWARE

What happens after the `PasswordSignInAsync()` succeeds?

1. The ASP.NET Core Identity framework issues an authentication cookie.
2. This cookie is sent back automatically with the response.
3. The cookie is stored by the browser and is sent automatically with every further request.
4. The ASP.NET backend uses it to identify the user without re-entering credentials and reconstructs the user identity from it.

This is called **cookie-based authentication**.

- We will discuss token-based authentication in the upcoming lecture.

WHAT ARE COOKIES?

A **cookie** is a small piece of data that a server sends to the browser, which the browser stores and sends back automatically with every subsequent request to the same site.

- HTTP is stateless – cookies let the server recognize the same client across requests.
- Set by the server via the `Set-Cookie` response header; sent back by the browser in the `Cookie` request header.
- Stored as a **`name=value`** pair scoped to a domain and path.
- Usage: authentication / session, user preferences, analytics, tracking.
- Lifetime: session (no expiration date) or persistent (expiration date)
- Security attributes: **`Secure`**, **`HttpOnly`**, S**`ameSite`**

COOKIES IN ASP.NET CORE – EXAMPLES

- A cookie is a name/value pair scoped to a web address:

```
Response.Cookies.Append("MyCookie", value, options);
```

- Set an expiration via **`CookieOptions`**:

```
var options = new CookieOptions { Expires = DateTime.Now.AddDays(10) };
```

- Read incoming cookies from **`Request.Cookies`**:

```
string value = Request.Cookies["MyCookie"];
```

- Delete by setting a past expiration, or:

```
Response.Cookies.Delete("MyCookie");
```

AUTHENTICATION IN ACTIONS

The logged in user can also be managed via the **`HttpContext`** class

- the **`User.Identity.IsAuthenticated`** property indicates whether a user has logged in
- the username of the logged in user can be accessed through the **`User.Identity.Name`** property
- e.g.:

```
if (User.Identity.IsAuthenticated) {
  IdentityUser user = await userManager.
    FindByNameAsync(User.Identity.Name);
    // retrieve the logged in user
}
```

ACCESSING REQUEST AND RESPONSE CONTEXT

- The **`HttpContext`** property is available in controllers (descendants of the **`ControllerBase`** class).
- In other classes, e.g. a business logic service, it can be accessed by injecting an **`IHttpContextAccessor`** object.
  - e.g.:

```
public class SomeService {
  private readonly HttpContext _httpContext;
  public SomeService(IHttpContextAccessor
                     accessor) {
    _httpContext = accessor.HttpContext;
  }
}
```

CONFIGURING IDENTITY

ASP.NET Core Identity has to be properly configured for the application

- the *identity type* (user type) can be registered to *IoC container* through the **`builder.Services`** service collection (**`IServiceProvider`**) in the **`Program`** class using **`AddIdentity<MyUser>()`**
  - Identity rules (e.g. required password strength) can also be configured as part of this
- the user store type can be registered with **`AddUserStore<MyStore>()`**
- when using EF, **`AddEntityFrameworkStores<MyDbContext>()`** sets up an Entity Framework Core based data store and specifies the database entity context type to be used

CONFIGURING IDENTITY

- e.g.:

```
builder.Services.AddIdentity<MyUser>(options => {
  // settings
  options.Password.RequireDigit = true;
  options.Password.RequiredLength = 8;
  options.Password.RequireNonAlphanumeric = false;
  options.Password.RequireUppercase = true;
  options.Password.RequireLowercase = false;
  options.Password.RequiredUniqueChars = 3;
  options.User.RequireUniqueEmail = true;
})
  .AddEntityFrameworkStores<MyDbContext>()
  .AddDefaultTokenProviders();
```

CONFIGURING IDENTITY

- The configured user management also must be enabled in the **`Program`** class on the **`WebApplication`** object:
  `app.UseAuthentication();`
  - With cookie-based authentication, this reads the incoming cookie to automatically recreate the user identity
- It is imperative that the authentication and authorization middleware must be ordered correctly in the **`Program`** class as follows:

```
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
```

CONFIGURING IDENTITY

- **`app.UseRouting();`**
  Routing comes first, it determines which endpoint is being accessed, which is required for authorization.
- **`app.UseAuthentication();`**
  This builds the user identity, which is then checked by the authorization.
- **`app.UseAuthorization();`**
  Must be after both routing and authentication therefore.

AUTHENTICATION & AUTHORIZATION

Both web page controller and the application programming interface (API) of web services should be adequately protected

- *authentication*: it may be necessary to verify the identity of the user when accessing endpoints
- *authorization*: when accessing endpoints, it may be necessary to check that the user is authorized to perform the operation

Even when the description of the API is not publicly available (e.g. through a public OpenAPI specification), endpoints can still be known

We cannot expect the user to use only the specified clients, the user can send any HTTP request at any time!

AUTHORIZATION

Access to resources can be restricted to logged in users

- the **`Authorize`** attribute can be applied to controllers or action methods, so that the resource can only be used after proper authorization (automatically enforced by the framework)
- e.g.:

```
[Authorize] // only for logged in users
public IActionResult ManageAccount(){ ... }
```

- access can be restricted to **`Users`** and **`Roles`**
- when authorization is applied on the controller level, certain action methods can be exempted with the **`AllowAnonymous`** attribute

AUTHORIZATION

- e.g.:

```
[Authorize] // applied to all action methods
public class AccountController : Controller
{
  public IActionResult ManageAccount(){ … }

  [Authorize(Roles = "admin")]
  // only users with admin role can access this (next lecture)
  public IActionResult ManageAllAccounts() { … }

  [AllowAnonymous] // available for everyone
  public IActionResult Login(){ … }
  // ...
}
```

SECURE COMMUNICATION

Communicating on a secure channel, encrypting messages using *Transport Layer Security* (TLS, SSL) is essential in most cases

- configuring to use secure communication channels and enforcing them is the task of the webserver
- otherwise interception of communication (including *man in the middle* attacks), *session theft* attacks become trivial
- a webpage can expect a secure channel for a resource (with the **`RequireHttps`** attribute) or check for it with the **`Request.IsHttps`** attribute dynamically

SECURE COMMUNICATION

Possible solutions:

- Do not listen on unsecure channels at all, use only HTTPS
- Redirect all HTTP communication to HTTPS
  `app.UseHttpsRedirection();`
- Additionally, use HSTS as well to cache this setting at the clients
  `app.UseHsts();`
- Require HTTPS only for certain controller or action method
  (not advised in general, especially for WebAPI projects)

```
[Authorize]
[RequireHttps] // available only through secure channel
public class AccountController : Controller { … }
```

EXAMPLE

Task: Implement user management for the travel agency website in the ASP.NET WebAPI backend + Angular frontend architecture.

- Utilize *ASP.NET Core Identity*, the **`TravelAgencyContext`** type specializes the **`IdentityDbContext`** type. **`User`** type is a specialization of **`IdentityUser`**, extending it with full name and address.
- Extend the data access layer with a new **`UserService`**, and the WebAPI with a new **`UserController`** for user management. The framework provided **`UserManager`** and **`SignInManager`** services are used for direct user management.
    - Also extend the **`Program`** class with the proper configuration.
- When logged in, personal data is auto-filled for booking an apartment. Booking is still possible as a visitor, without account.

EXAMPLE

How to observe the authentication cookie?

1. In your browser, open *Developer Tools* → *Application (or Storage)* → *Cookies*, and look for a cookie like (default name): **`.AspNetCore.Identity.Application`**
2. This authentication cookie, created by **`PasswordSignInAsync(...)`**
   - Sent automatically with every request.
   - Used by the server to restore the user identity.
3. If you log out, the cookie is removed / invalidated.

The cookie is encrypted and signed (cannot be modified by the client).
The cookie content is a unique token, not the password or its hash.

ELTE EÖTVÖS LORÁND UNIVERSITY

ACCESS TOKENS AND ROLE-BASED ACCESS CONTROL

Modern web application development in .NET

Lecture 7

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

ACCESS TOKENS

Due to the lifecycle of web applications, the state is not preserved between two requests. Direct authentication of each request by the user (e.g. with username-password pair) is cumbersome

Access token based authentication can provide an automated solution

- after successful username-password authentication, an access token is provided to the client and persisted in the application
- for further requests, the client can send the previously received access token instead of the username-password pair
- secure, encrypted channel (HTTPS) is a critical element of security

ACCESS TOKENS

There are multiple approaches to pass access tokens between the client and the server

- in machine-to-machine communication (between applications)
  - in a GET/POST parameter, or
  - HTTP header (conventionally named **`Authorization`**)
- in machine-to-human communication (e.g. a human user via a browser) typically in a cookie
  - *ASP.NET Core Identity* uses the **`.AspNetCore.Identity.Application`** cookie for this by default
  - in the application, the required information is stored in the **`SecurityStamp`** field of the **`AspNetUsers`** table

ACCESS TOKENS

Typically, an access token is valid only for a specific period of time

- It can have a *sliding expiration*, i.e. it is extended for a limited period of time after each successful request
  - Typically used for cookie-based authentication
- Typically, a fixed lifetime access token is used for inter-application communication
  - When logging in, in addition to the *access token*, a *refresh token* is also provided to client and stored
  - New access tokens can be requested using the refresh token

REFRESH TOKENS

- *Access tokens*, are short lived, typically minutes to hours (e.g. *30 minutes*)
- *Refresh token* are usually long lived, days to month (*e.g. 30 days*)
  - The refresh token can be used to fetch a new access token
  - Usually, the refresh token is rotated on each usage:
    - The old refresh token is revoked
    - A new refresh token is created and returned to the user
- It is the responsibility of the client application to fetch a new access token either close to its expiration or on the next usage after it

ACCESS TOKENS TYPES

There are two types of access tokens:

- *opaque tokens*, where the token refers to a resource stored on the server, but does not carry any content on its own
- *transparent tokens*, where the token carries encrypted content, so that it can be used not only for authentication but also for authorization or to carry any information in general
    - the token can be digitally signed, so that it can be verified that its content has not been modified since it was issued by the server
    - the token can even be digitally encrypted if it is important that its content is not easily readable
    - such a specification is the *JSON Web Tokens* (JWT) standard

JSON WEB TOKENS (JWT)

JSON Web Tokens (JWT) are used to securely transfer information between applications

- After a successful username-password authentication, a server generates a JWT that contains other information (e.g. access privileges) in addition to the user's identity, called *claims*. The JWT is digitally signed by the server and delivered to the client.
- The client can read the JWT (if not encrypted) and return it to the server as an access token for further requests.
- The server checks that the content of the received JWT matches the digital signature and has not been *tampered* with. If it is OK, its content is authentic, therefor e.g. it is not necessary to load the privilege roles directly from the database.

JSON WEB TOKENS (JWT)

The JWT token is in JSON format and consists of three parts:

- *Header*: metadata about the token
- *Payload*: the content carried by the token
- *Signature*: the digital signature of the token

The three parts of the JWT token are encoded in base64 to avoid character encoding problems and are sent separated by dots (so they can be passed even as GET parameters)

- format: `header.payload.signature`, e.g.:
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2VsdGU uaHUvIiwic3ViIjoiMTIzNDU2Nzg5IiwibmFtZSI6IlRlc3p0IEVsZWsiLCJhZG1 pbiI6dHJ1ZX0.U56N4FGI21Dw2GVAACzv7PRxTck7jvsszN23GKHn4ak

JSON WEB TOKENS (JWT)

- The JWT *header* specifies the used algorithm (HMAC SHA256, RSA, ECDSA) for digital signature and the type of the token (which is JWT).

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- The JWT *payload* contains the carried information, e.g.:

```
{
  "iss": "https://elte.hu/",
  "sub": "123456789",
  "name": "Test User",
  "admin": true
}
```

JSON WEB TOKENS (JWT)

The payload contains the JWT claims, which fall into 3 categories:

- There are so-called *registered claims*, predefined in the JWT specification, such as **`iss`** (*issuer*), **`exp`** (*expiration time*) or **`sub`** (*subject*). Registered claims all have a 3-character name for brevity. They are essential to the purpose of JWT.
- There are also conventionally used *public claims* such as **`name`**, **`email`** or **`birthdate`**. They are widely used in many applications, so using the same claim names is beneficial.
- You can also define any private claim you like, such as **`admin`** in the previous example.

JSON WEB TOKENS (JWT)

The JWT *signature* is the digital signature of the *header* and *payload*, encoded in base64.

- When using the HMAC SHA256 algorithm, a *hash value* is generated using a *secret key* with the following algorithm (pseudo-code):

```
signature = HMACSHA256(
  base64(header) + "." + base64(payload),
  secret)
```

- The key is provided by the server (as a configuration value) and should be known only by the server issuing the digital signature.

CONFIGURE ASP.NET CORE FOR JWT

ASP.NET Core provides integrated support for JWT-based authentication and authorization

- at the **`Authorize`** attribute it can be specified to use JWT instead of the default identity token based authorization (**`JwtBearerDefaults.AuthenticationScheme`**)

As a more robust approach, on the **`WebApplicationBuilder`** object in the **`Program`** class, you can also configure this as the new default:

```
builder.Services.AddAuthentication(options => {
  options.DefaultAuthenticateScheme =
    JwtBearerDefaults.AuthenticationScheme;
  options.DefaultChallengeScheme =
    JwtBearerDefaults.AuthenticationScheme;
});
```

CONFIGURE ASP.NET CORE FOR JWT

- Here we can also enable the JWT-based authentication (**AddJwtBearer**) and configure the token validation:

```
builder.Services.AddAuthentication(...)
   .AddJwtBearer(options => {
     options.TokenValidationParameters =
       new TokenValidationParameters
       {
         ValidIssuer = "https://elte.hu", // validate issuer
         IssuerSigningKey =
           new SymmetricSecurityKey(
             Encoding.UTF8.GetBytes("secret key"))
         // validate digital signature
       };
});
```

CREATE JWT TOKENS

- Now, after a successful authentication with username-password pair, a JWT can be generated for the client:

```
var tokenDesc = new SecurityTokenDescriptor {
  Subject = new ClaimsIdentity(new[] {
              new Claim(ClaimTypes.Name, user.UserName)
  }), // claims added to the JWT
  Issuer = "https://elte.hu",
  Expires = DateTime.Now.AddMinutes(30),
  SigningCredentials = new SigningCredentials(
    new SymmetricSecurityKey(// configure digital signature
      Encoding.UTF8.GetBytes("secret key")),
      SecurityAlgorithms.HmacSha256)
};
```

CREATE JWT TOKENS

- The JWT may also contain authorization information, such as the user's roles, which could be added like:

```
var roles = await userManager.GetRolesAsync(user);
foreach (var role in roles)
  tokenDesc.Subject.AddClaim(
    new Claim(ClaimTypes.Role, role));
```

- Create the JWT string:

```
var handler = new JwtSecurityTokenHandler();
var jwtToken = handler.CreateToken(tokenDesc);
string tokenStr = handler.WriteToken(jwtToken);
```

- Finally, the JWT token can be returned to the client in the response body or even in an HTTP header:

```
Response.Headers.Add("Bearer", tokenStr);
```

PASSING JWT FROM SERVER TO CLIENT

- On the client side, the JWT token can be read from the HTTP message header of the response:

```
using (HttpClient client = new HttpClient())
{
  HttpResponseMessage response = await
    client.GetAsync("api/account/login");
  if (response.IsSuccessStatusCode)
  {
    token = response.Headers.GetValues("Bearer")
                            .FirstOrDefault(); // JWT token
  }
}
```

PASSING JWT FROM CLIENT TO SERVER

- For further requests, the client adds the JWT token to the *Authorization* header with the *Bearer* prefix (e.g. **`Bearer eyJhb...n4ak`**), where ASP.NET Core will look for it.

```
using (HttpClient client = new HttpClient())
{
  client.DefaultRequestHeaders.Authorization =
    new AuthenticationHeaderValue("Bearer", token);
  // add JWT token to Authorization HTTP header
  HttpResponseMessage response =
    await client.PostAsJsonAsync("…", <object>);
  // … await response
}
```

CLIENT APPLICATIONS

- Web services decouples backend and frontend, they allow the general usage of the server-side backend application in various environments.
- A client application of any platform can provide a client-side interface to a web service.
  - Can be a desktop application, a mobile application, a website
  - Can be used by other web services
- The backend web service and frontend clients can be developed in parallel on different platforms and programming languages by different development teams.

3RD PARTY CLIENT APPLICATIONS

- With JWT-based authentication and authorization, any client application can connect to our server-side service to perform operations by authorizing users
  - e.g. our Spotify account can post to our Facebook wall
- The authentication and authorization protocol is controlled by a separate specification (e.g. *OAuth2*, *OpenId*), JWT can be the format of the tokens (mandatory for *OpenId*, but not for *OAuth2*)
  - The protocol describes the type of tokens, their content, how they are issued and revoked, etc.
- Alternatives exist, e.g. SAML (*Security Assertion Markup Language*) defines both the protocol and its own XML-based token format.

ELTE EÖTVÖS LORÁND UNIVERSITY

ROLE-BASED ACCESS CONTROL

Modern web application development in .NET

RBAC

RBAC (*Role-Based Access Control*) is an authorization model based on roles assigned to users.

- Users and roles are defined through the Identity framework
- Users are assigned to roles
- Roles define the permissions

Access decisions are based on roles, not on individual users.

WHY USE RBAC?

Advantages of RBAC:

- Scalable (easy to manage many users)
- Maintainable (changing a role affects all users)
- Cleaner than per-user permissions
- Often matches real-world structures (e.g. admin, manager, user)

RBAC IN ASP.NET CORE

ASP.NET Core Identity has built-in support for roles

- Roles are stored in the **`AspNetRoles`** relation (can be renamed)
- User to role assignments are stored in the **`AspNetUserRoles`** relation

As in general with the Identity framework, these relations should not be managed directly (e.g. through *Entity Framework Core*), but through the provided **`UserManager`** and **`RoleManager`** classes.

```
await roleManager.CreateAsync(new IdentityRole("Admin"));
await userManager.AddToRoleAsync(user, "Admin");
```

USING ROLES IN AUTHORIZATION

- In an ASP.NET MVC webpage or web service we may the [Authorize] attribute with roles
  - Only users with required roles can access endpoint. E.g.:

```
[Authorize(Roles = "Admin")]
[Authorize(Roles = "Admin,Manager")]
```

- This works out-of-the-box with JWT tokens as well, if roles are included as claims in the tokens.

```
new Claim(ClaimTypes.Role, role)
```

LIMITATIONS OF RBAC

Possible disadvantages of RBAC include:

- Roles can become too broad (e.g. admin does everything)
- Role explosion (too many roles)
- Hard to model complex rules

As a more fine-grained approach, *PBAC (Policy-based Access Control)* is available in ASP.NET Core and can be integrated with JWT tokens as well.

- E.g.:

```
[Authorize(Policy = "FinancePolicy")

options.AddPolicy("FinancePolicy", policy =>
  policy.RequireClaim("department", "Finance"));
```

ELTE EÖTVÖS LORÁND UNIVERSITY

CRUD DATA MANAGEMENT & TESTING WEB SERVICES

Modern web application development in .NET

Lecture 8

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

CRUD

In many cases the web service provides basic data processing operations, the *CRUD* operations: *Create, Read, Update, Delete*

- these operations have HTTP equivalents (create: **`POST`**, read: **`GET`**, modify: **`PUT`**, delete: **`DELETE`**)
  - upon a successful operation the response code is **`CREATED`** (201) for creation, **`OK`** (200) or **`NO CONTENT`** (204) for other operations
    - with the **`CREATED`** result the created resource (object) and its location (**`GET`** path) must be returned as well
  - when the operation is not executed immediately, the status **`ACCEPTED`** (202) may be indicated
- in a RESTful service, the operations must conform to this scheme

DATA SYNCHRONIZATION MODELS

Client-side data management can be implemented in two ways:

- *synchronous:* client and server state are always the same
- *asynchronous:* client and server state are different and can be synchronized manually (save, load, update)

Asynchronous data management is advantageous if you want to save your changes in groups rather than individually

- to achieve this, changes have to be tracked on the client side with *status flags* and indicate what changes have been made to the data (new, modified, deleted)

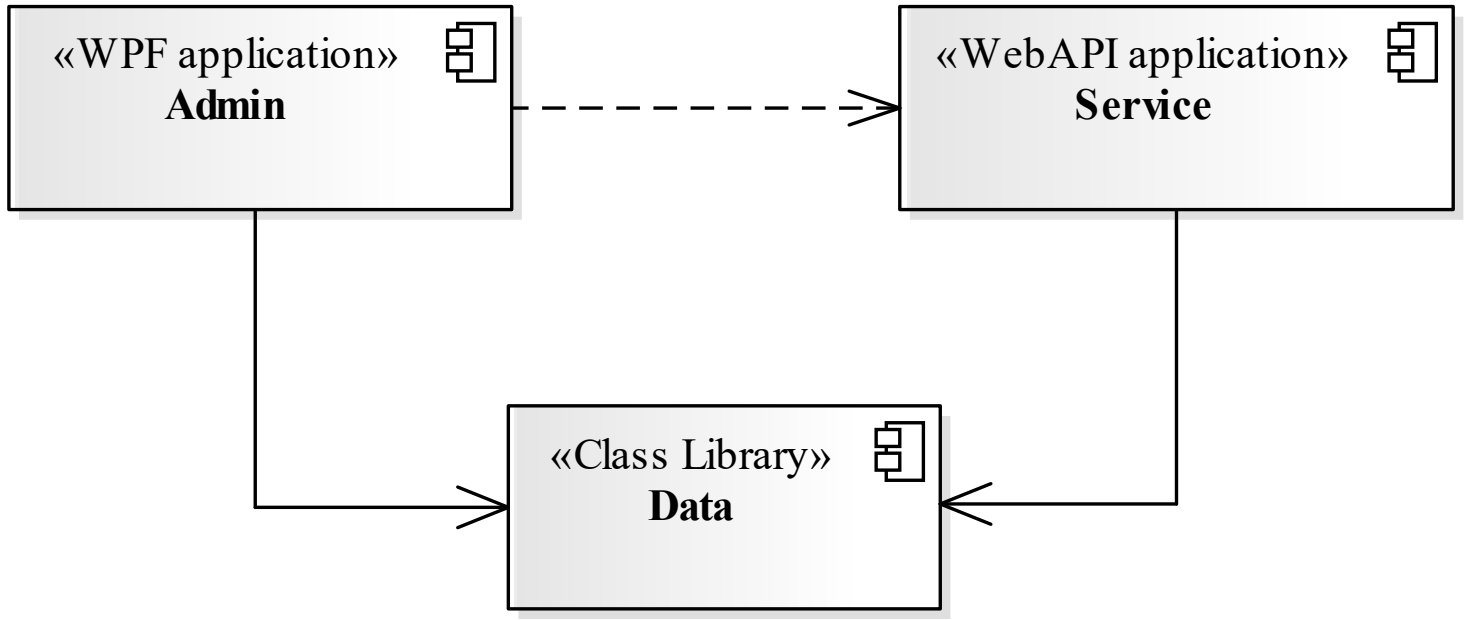
EXAMPLE

Task: Implement a desktop admin application for the travel agency website with the ASP.NET Core based webservice and a *WPF frontend*.

- The client will be a WPF application (**`TravelAgency.Admin`**) that will connect to an ASP.NET Core webservice (**`TravelAgency.WebAPI`**)
- the client application will be built in *MVVM architecture*, where persistence (**`TravelAgencyServicePersistence`**) will provide the network communication
  - the persistence layer is placed in a separate class library project (**`TravelAgency.Admin.Client`**) for easier testing and reusability
- Separate DTO types used for data transfer are placed in a separate class library (**`TravelAgency.Shared`**), and shared between the server and the client project to avoid code redundancy

EXAMPLE

Design (*architecture*):

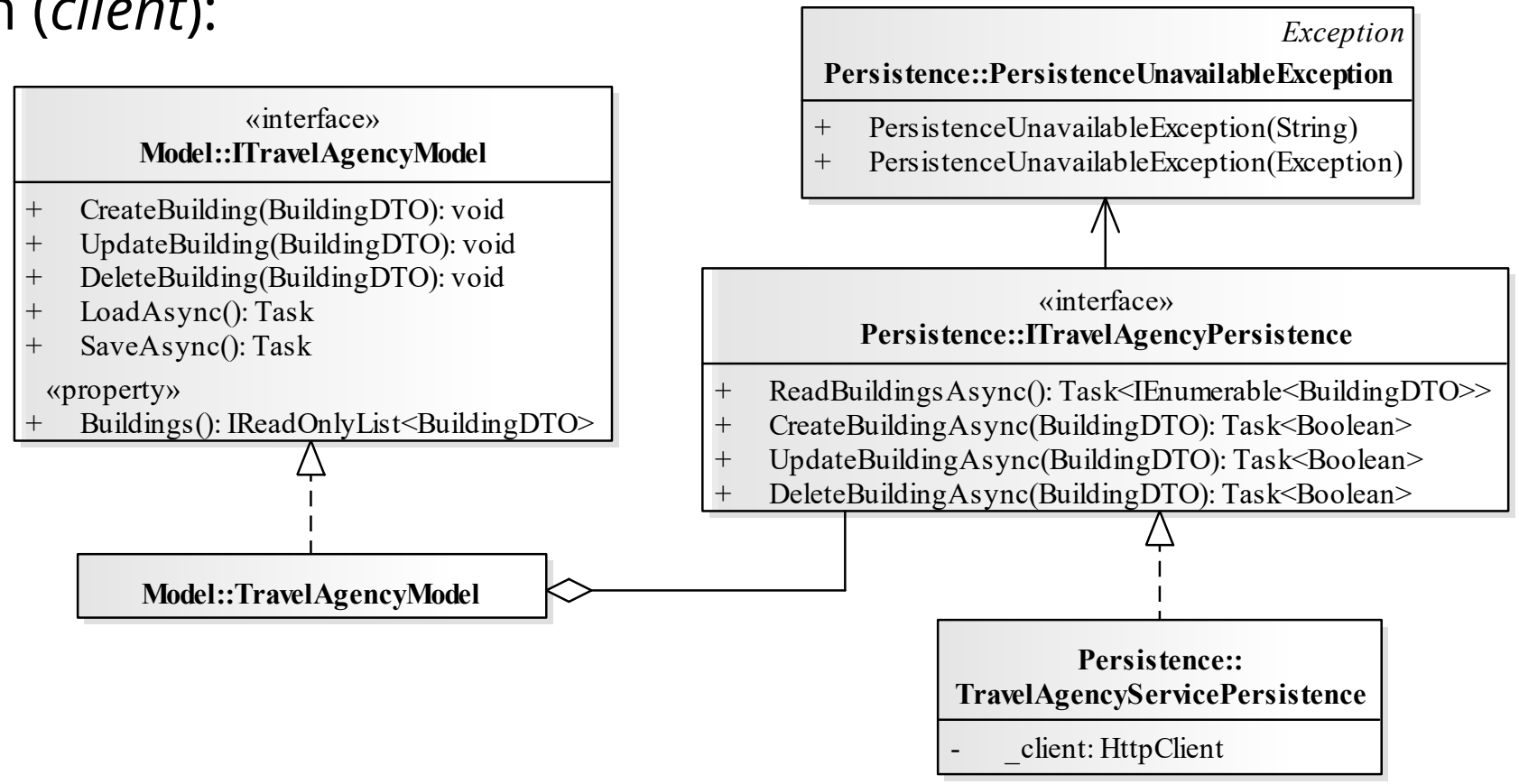


EXAMPLE

- The admin client should be able to add, edit and remove buildings
  - the client provides asynchronous data management, the model (**`ITravelAgencyModel`**) monitors the client state with status flags (**`DataFlag`**), based on which the appropriate action (add / update / delete) can be performed upon saving
  - persistence (**`ITravelAgencyPersistence`**) is responsible for loading, saving data with asynchronous operations
  - a temporary identifier is created for new buildings, which is replaced by a permanent identifier returned by the server when modifications are persisted to the web service
  - the **`BuildingsController`** of the service (**`TravelAgency.WebAPI`**) is extended with the missing CRUD operations

EXAMPLE

Design (*client*):

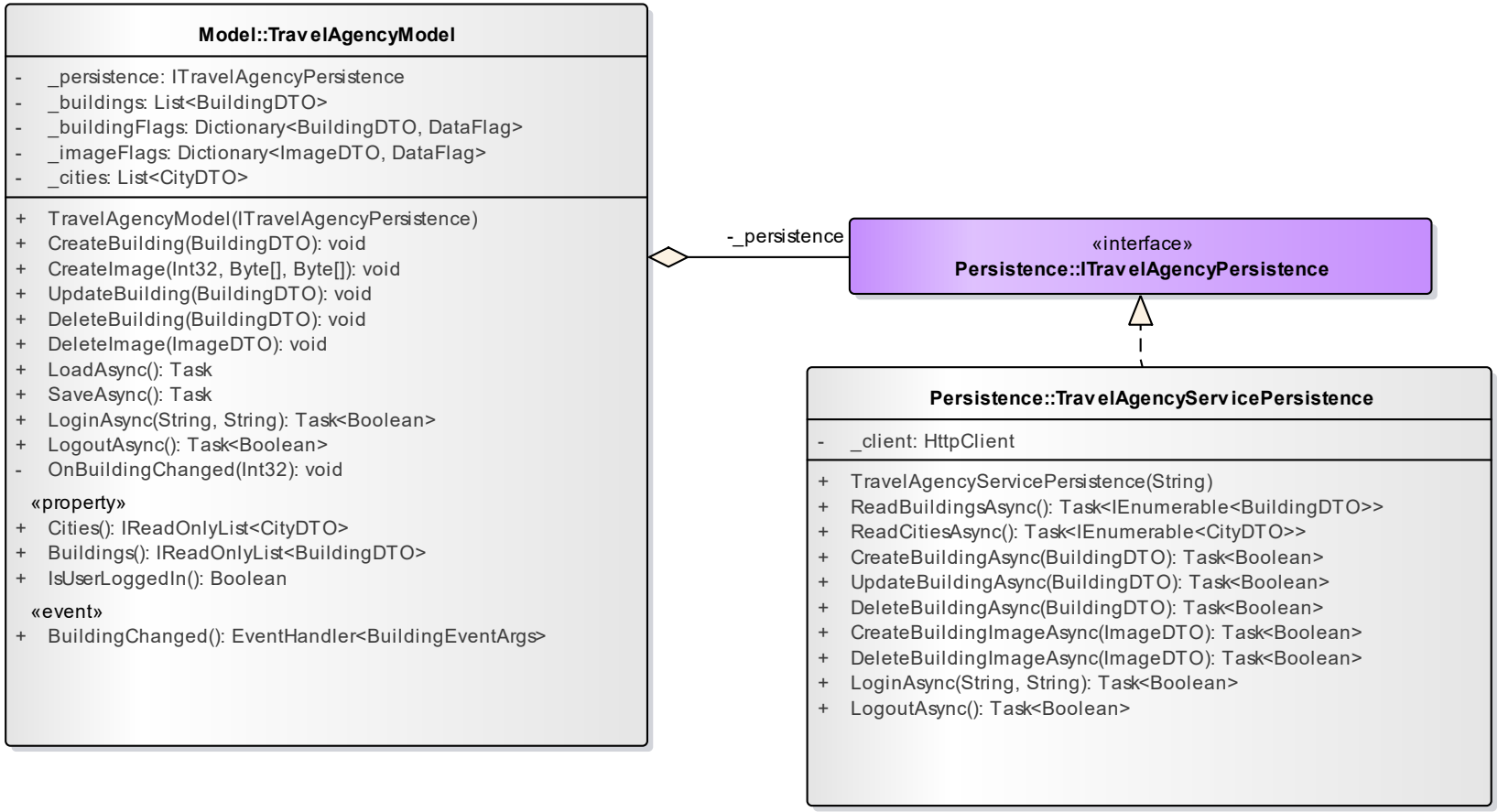


EXAMPLE

- The admin client should also support to view, add and delete pictures for buildings
  - images will be loaded from files, then resized and converted accordingly before upload (small and large size, PNG format)
  - images will be linked to buildings and also have a temporary unique identifier, until their permanent identifier is defined and returned by the webservice
  - the **`BuildingsController`** of the webservice is extended
- To ensure security, data management is protected by authentication and authorization (using *ASP.NET Core Identity*), so that the user must first log in to the application
  - the existing **`UsersController`** is used for login and logout

EXAMPLE

Design (*client*):



ELTE EÖTVÖS LORÁND UNIVERSITY

TESTING WEB SERVICES

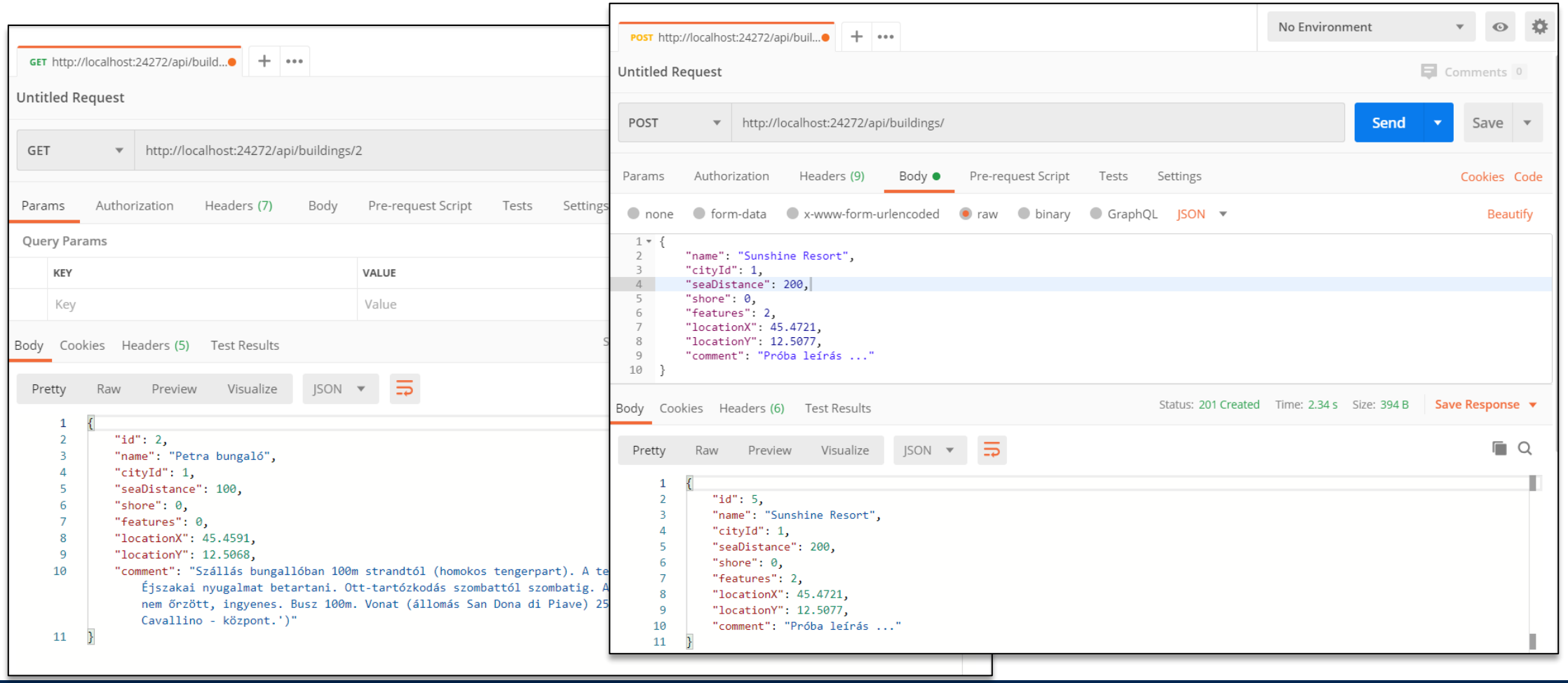
Modern web application development in .NET

TESTING WEB SERVICES

Testing of web services can be done

- manually, on the client side, using a program that sends the requests, such as a browser or target software (e.g. *Postman*, *Insomnia*, *Fiddler*)
- manually, using a tester desktop or web interface generated for the web service API (e.g. *Swagger UI*)
- automatically, on the client side, using a class that sends requests (e.g. using **`HttpClient`** in .NET)
- automatically, on the server side, by directly testing the controllers
  - the testing should be done in a managed environment
  - external factors (e.g. database) could be excluded, *mocked*

USE POSTMAN


GET http://localhost:24272/api/build...
Untitled Request
GET
http://localhost:24272/api/buildings/2
Params Authorization Headers (7) Body Pre-request Script Tests Settings
Query Params
KEY
VALUE
Key
Value
Body Cookies Headers (5) Test Results
Pretty Raw Preview Visualize JSON
{
"id": 2,
"name": "Petra bungaló",
"cityId": 1,
"seaDistance": 100,
"shore": 0,
"features": 0,
"locationX": 45.4591,
"locationY": 12.5068,
"comment": "Szállás bungallóban 100m strandtól (homokos tengerpart). A te
Éjszakai nyugalmat betartani. Ott-tartózkodás szombattól szombatig. A
nem őrzött, ingyenes. Busz 100m. Vonat (állomás San Dona di Piave) 25
Cavallino - központ.')"
}
POST http://localhost:24272/api/buil...
No Environment
Untitled Request
Comments 0
POST
http://localhost:24272/api/buildings/
Send
Save
Params Authorization Headers (9) Body Pre-request Script Tests Settings
Cookies Code
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
{
"name": "Sunshine Resort",
"cityId": 1,
"seaDistance": 200,
"shore": 0,
"features": 2,
"locationX": 45.4721,
"locationY": 12.5077,
"comment": "Próba leírás ..."
}
Body Cookies Headers (6) Test Results
Status: 201 Created Time: 2.34 s Size: 394 B Save Response
Pretty Raw Preview Visualize JSON
{
"id": 5,
"name": "Sunshine Resort",
"cityId": 1,
"seaDistance": 200,
"shore": 0,
"features": 2,
"locationX": 45.4721,
"locationY": 12.5077,
"comment": "Próba leírás ..."
}

TESTING WEB SERVICES

There are various approaches when it comes to testing applications with databases; all have their benefits and drawbacks.

1. Use the in-memory database provider of *Entity Framework Core*, instead of the real database
    - Benefit: simple and very fast
    - Drawback: the in-memory database does not support foreign keys, reference integrity constraints, transactions
2. Use the *SQLite* (or other file-based) database provider of *Entity Framework Core*
    - Benefit: full-fledged relational database and still simple setup
    - Drawback: there could be differences from the production DB

TESTING WEB SERVICES

3. Mock the database management to some in-memory or other fake data provider
    - Benefit: completely mock the database management
    - Drawback: mocking the **`DbSet`** can be quite difficult and complex
4. Use a real test database
    - Benefit: mirror the production environment, therefore tests are more reliable
    - Drawback: more complex setup (especially in *CI*), usually slower test execution

UNIT TEST FRAMEWORKS

For platform-independent unit tests, the most widely used frameworks are *MSTest*, *NUnit* or *xUnit*

- All three frameworks are natively supported by Visual Studio & Rider.
- Most notable differences on a basic level:

MSTest	NUnit	xUnit	
`[TestClass]`	`[TestFixture]`	*n/a*	Test class
`[TestMethod]`	`[Test]`	`[Fact]`	Test case (method)
`[TestInitialize]`	`[Setup]`	*ctor*	Initialize test cases
`[TestCleanup]`	`[TearDown]`	`IDisposable`	Clean up test cases

IN-MEMORY DATABASE

- The database context can be parameterized through dependency injection with the database engine to be used:

```
services.AddDbContext<MyDbContext>(options =>
  options.UseSqlServer(
    Configuration.GetConnectionString(
      "DefaultConnection")));
```

- Temporary in-memory database can also be used for testing:

```
var options = new DbContextOptionsBuilder<MyDbContext>()
  .UseInMemoryDatabase("TestDb").Options;
var context = new MyDbContext(options);
```

MOCK OBJECTS

When testing a program unit with a dependency, we replace the dependency with a simulation of it, called a *mock object*

- the *mock object* implements the interface of the dependency, providing a simple, error-free functionality
- by using *mock objects*, the test actually checks the functionality of the specified program unit, unaffected by any error in the dependency

Mock objects can be created manually or by using a suitable software library

- e.g. *Moq*, *NSubstitute*, *FakeItEasy*, all avaliable as NuGet packages

MOCK OBJECTS

e.g.:

```
interface IDependency
{
  bool Check(double value);
  double Compute();
}
...
class DependencyImplementation : IDependency
{
  public bool Check(double value) { … }
  public double Compute() { … }
}
```

MOCK OBJECTS

e.g.:

```
class Dependant { // class with a dependency
  private IDependency _dependency;
  public Dependant(IDependency d) { _dependency = d; }
  // dependency injection
  ...
}

...

Dependant d = new Dependant(new DependencyImplementation());
// pass the implementation of the dependency
```

MOCK OBJECTS

e.g.:

```
class DependencyMock : IDependency // mock type {
  // simple implementation
  public double Compute() { return 1; }
  public bool Check(double value) {
    return value >= 1 && value <= 10;
  }
}

...

Dependant d = new Dependant(new DependencyMock());
// inject the mock object into the dependant
```

MOCK OBJECTS

Moq makes it easy to create mock objects from interfaces

- the **`Mock`** generic class can be used to instantiate the simulation, which can be accessed with the **`Object`** property and produces default behaviour, e.g:

```
Mock<IDependancy> mock = new Mock<IDependancy>();
// mock object of the dependency
Dependant d = new Dependant(mock.Object);
// can be used immediately
```

- the **`Setup`** method you can set the behaviour of any of its members (**`Returns(...)`**, **`Throws(...)`**, **`Callback(...)`**), parameters can be managed (It)

MOCK OBJECTS

- e.g.:

```
mock.Setup(obj => obj.Compute()).Returns(1);
// define the behaviour, always return 1
mock.Setup(obj =>
  obj.Check(It.IsInRange<Double>(0, 10, Range.Inclusive)))
    .Returns(true);

mock.Setup(obj => obj.Check(It.IsAny<Double>())
    .Returns(false);
// multiple cases based on the parameter
```

- method calls can also be traced whether occurred or not (**`Verify(...)`**), and how many times

MOCKING ENTITY FRAMEWORK

The interface of *Entity Framework* can be mocked as well in a similar way, for each **`DbSet`** property

- e.g.:
  ```
  var productMock = new Mock<DbSet<Product>>();
  ```
- However, to use the *mocked* **`DbSet`** as an **`IEnumerable`** object for querying data, at least the **`ElementType`**, **`Expression`** and **`Provider`** properties, and the **`GetEnumerator()`** method must be configured.
  - To support asynchronous operations, the **`GetAsyncEnumerator()`** method also must be configured.
  - This can be quite complex and difficult, requiring to write multiple helper classes, therefore mocking the **`DbSet`** itself is officially not recommended by Microsoft.

MOCKING ENTITY FRAMEWORK

- Instead, when during testing, the exclusion of the database is done through mocking, it is recommended to mock the *data access layer*
- It is always advisable anyway to restrict direct database management to a *data access layer* for better code maintainability, for enforcing validations and reducing bugs
  - Often implemented through the *Repository Design Pattern*
  - e.g.:

```
var serviceMock = new Mock<IProductService>();
serviceMock.Setup(service => service.GetAllAsync())
           .ReturnsAsync(...);
var controller = new ProductController(serviceMock.Object);
```

INTEGRATION TESTING

Testing the combined behaviour of multiple software components is called an *integration test*. The elimination of external factors, such as errors due to network connectivity, should also be addressed when testing the client and server together.

- Since the service requires a web server, *ASP.NET Core* provides a lightweight web server (**`TestServer`**) that allows the service to run directly in memory without the need for a network connection
- The easiest way to create the test server is to use the **`WebApplicationFactory`** class, which is generically parameterized with the entry point of the web service to be tested, the **`Program`** class
    - We can specify overriding modifications of the configuration (e.g. *in-memory* database), in the **`ConfigureTestServices()`** method

INTEGRATION TESTING

- E.g. (*test server*):

```
var server = new WebApplicationFactory<Program>()
  .WithWebHostBuilder(builder => {
    // Remove previous ProductContext registration
    services.RemoveAll(
      typeof(DbContextOptions<ProductContext>));
    // Override ProductContext registration
    builder.ConfigureTestServices(services => {
      // Add in-memory database context
      services.AddDbContext<ProductContext>(options =>
        options.UseInMemoryDatabase("TestConnection"));
    });
  });
```

INTEGRATION TESTING

- The client (a **`HttpClient`** object) can be instantiated with the **`CreateClient()`** method
  - client objects created this way are automatically connected to the test server instance,
  - all client requests are executed in memory.
- E.g.:

```
HttpClient client = server.CreateClient();
// create and connect client to the server
var response = await client.GetAsync("api/products");
// in-memory communication
```

EXAMPLE

Task: Test the web service of the travel agency website.

- Create a separate *xUnit* test project (**`TravelAgency.WebAPI.Test`**)
  - implement some test cases for all 3 showcased methods in separate classes (**`BuildingsControllerTest`**, **`BuildingsMockServiceTest`**, **`BuildingsIntegrationTest`**)
  - the database is initialized with a sample dataset before each test case in the constructor.
  - the database is destroyed after each test case execution in the **`Dispose()`** method.
  - add **`Microsoft.EntityFrameworkCore.InMemory`**, **`Moq`**, **`Microsoft.AspNetCore.Mvc.Testing`** and **`Microsoft.AspNet.WebApi.Client`** NuGet packages

EXAMPLE

- Extend testing with the CRUD operations in all 3 test classes (**BuildingsControllerTest**, **BuildingsMockServiceTest**, **BuildingsIntegrationTest**)
  - In integration testing test the persistence layer of the client, e.g.:

```
var client = server.CreateClient();
// instantiate client
var persistence =
  new TravelAgencyServicePersistence(_client);
// instantiate client-side persistence layer
IEnumerable<BuildingDto> result =
  await persistence.ReadBuildingsAsync();
```

  - During integration testing, verify the successful and unsuccessful authorization as well!

ELTE EÖTVÖS LORÁND UNIVERSITY

INTRODUCTION TO BLAZOR

Modern web application development in .NET

Lecture 9

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

WHAT IS BLAZOR?

Blazor is a framework for building multiplatform interactive web applications based on .NET Core and C#.

- Enables developers to write client-side web UI using C# instead of JavaScript or TypeScript.
- Supports various hosting models for rendering
  - the *Blazor WebAssembly* uses *WebAssembly*
  - the *Blazor Server* uses *SignalR*
- Provides a *component-based architecture*, making it easier to build and manage UI elements efficiently
- Part of *ASP.NET Core*, benefiting from the security, performance, and scalability of the .NET ecosystem

WHAT IS WEBASSEMBLY?

WebAssembly (*WASM*) is a binary instruction format designed for running high-performance applications in web browsers and other environments.

- WebAssembly is designed as a portable, efficient, and secure execution environment that runs across different platforms.
- Supported by all modern browsers, including *Chrome*, *Firefox*, *Edge*, and *Safari*.
- *Blazor WebAssembly* (*Blazor WASM*) allows C# code to run in the browser by compiling it into WebAssembly.
- Uses .NET runtime compiled to WebAssembly, enabling full .NET applications in the browser without requiring JavaScript.

WHY USE BLAZOR?

- Allows *full-stack* development using C#
  - Desktop and hybrid application development is also supported (e.g. MAUI, WPF, Avalonia)
- Eliminates the need for JavaScript in many scenarios
  - Can interoperate with JavaScript, allowing integration with existing JavaScript libraries if needed
- Provides rich UI components and two-way data binding
  - Supports real-time updates and two-way data binding, making the UI more interactive with less effort
- Integrates well with existing .NET libraries

BLAZOR HOSTING MODELS

Blazor applications can run in 3 major ways:

1. **Blazor WebAssembly (WASM)**
   - Runs in the browser (or webview) using *WebAssembly*
   - Executes .NET code client-side
2. **Blazor Server**
   - Runs on the server, sending UI updates via *SignalR*
   - Requires constant connection to the server
3. **Blazor Hybrid (via .NET MAUI or WPF)**
   - Runs inside native desktop and mobile applications
   - Enables building cross-platform apps with Blazor

BLAZOR APPLICATION ARCHITECTURE

Blazor app typically follows a three-layered architecture that includes:

1. **Presentation Layer (UI)**
   - contains Razor components that define UI elements
   - handles routing, event handling, and data binding
2. **Business Logic Layer (Services)**
   - contains application logic and business rules
   - provide services like: API calls, state management, validation
3. **Data Access Layer (DAL)**
   - Blazor Server: connects to a database via e.g. EF Core
   - Blazor WebAssembly: calls APIs for data retrieval

COMPONENT-BASED ARCHITECTURE

Blazor applications are built using *Razor Components*, which allow developers to create interactive web UIs using C# and .NET.

- Blazor provides a modern alternative to JavaScript-based frameworks by leveraging .NET technologies.
- Blazor applications are structured using *components*, which are reusable UI elements written in .razor files.
- Each component can include *HTML*, *C#*, and *Razor* syntax, allowing for dynamic content and event handling.
- Components can be nested inside other components, making UI development modular and maintainable.

COMPONENT-BASED ARCHITECTURE

- e.g. (*Blazor component*):

```
<h3>Hello, @Name!</h3>
<ul>
  @foreach (var item in Items)
  {
    <li>@item</li>
  }
</ul>

@code {
  string Name = "Blazor";
  string[] Items = { "...", "...", "..." }
}
```

EVENT HANDLING

Razor components have built-in event handling

- uses **@on[event]** syntax, e.g.:

```
<h3>Counter Example</h3>
<p>Current count: @count</p>
<button @onclick="IncrementCount">Increment</button>

@code {
  int count = 0;
  void IncrementCount() {
    count++;
  }
}
```

EVENT HANDLING

Razor components can handle various user interactions and browser events. Some common event types in Blazor:

- mouse events: **`@onclick`**, **`@onmouseenter`**, **`@onmouseover`**, etc.
- keyboard events: **`@onkeydown`**, **`@onkeyup`**, **`@onkeypress`**
- form events: **`@onchange`**, **`@onsubmit`**, etc.
- focus events: **`@onfocus`**, **`@onblur`**
- touch events: **`@ontouchstart`**, **`@ontouchend`**, **`@ontouchmove`**, etc.
- clipboard events: **`@oncop`**, **`@onpaste`**, **`@oncut`**
- etc.

RENDERING & UI UPDATES

- Blazor automatically updates the UI, based on changes in data
- When the state of a component changes, Blazor re-renders only the affected parts of the page, optimizing performance
  - State change can also be manually invoked with the **`StateHasChanged()`** method
- Blazor does not re-render the entire page when a change occurs
  - Instead, it uses a Virtual DOM diffing algorithm (similar to *React*), to efficiently update the UI

RENDERING & UI UPDATES

- e.g. when the **count** variable is modified, Blazor detects the change and updates only the **<p>** tag instead of reloading the entire page:

```
<h3>Counter Example</h3>
<p>Current count: @count</p>
<button @onclick="IncrementCount">Increment</button>

@code {
  private int count = 0;
  void IncrementCount() {
    count++; // Blazor automatically updates the UI
  }
}
```

RENDERING & UI UPDATES

- when working with lists of data, use the **@key** directive to optimize rendering, preventing unnecessary renders:

```
<ul>
  @foreach (var item in Items)
  {
    <li @key="item.Id">@item.Name</li>
  }
</ul>

@code {
  Product[] Items = …
}
```

DATA BINDING

Blazor supports one-way and two-way data binding, allowing components to interact with user input

- *One-way binding* updates the UI based on a C# property
- *Two-way binding* keeps the UI and the underlying C# property in sync
- e.g.:

```
<input @bind="name" />
<p>Hello, @name!</p>

@code {
    string name = "Blazor";
}
```

COMPONENT PARAMETERS

Component parameters allow data to be passed from a parent component to a child component

- This enables reusable and dynamic components that can behave differently based on the provided data
- Parameters are defined using the **`[Parameter]`** attribute inside the child component
- The **`[EditorRequired]`** attribute can mark a parameter as required.
- e.g. (**`ParentComponent.razor`**):

```
<h3>Parent Component</h3>
<ChildComponent Title="Welcome to Blazor!" Number="5" />
```

COMPONENT PARAMETERS

- e.g. (**ChildComponent.razor**):

```
<h3>Child Component</h3>
<p>Title: @Title</p>
<p>Number: @Number</p>

@code {
    [Parameter]
    public string Title { get; set; }
    [Parameter]
    public int Number { get; set; }
}
```

COMPONENT PARAMETERS

- sometimes, child components need to send data back to the parent
  - this is achieved using an **`EventCallback`** parameter, which allows a child component to notify its parent when an event occurs.
  - e.g. (**`ChildComponent.razor`**):

```
<button @onclick="SendMessage">Send Message to Parent</button>
@code {
  [Parameter]
  public EventCallback<string> OnMessageReceived { get; set; }

  async Task SendMessage() {
    await OnMessageReceived.InvokeAsync("Hello from Child!");
  }
}
```

COMPONENT PARAMETERS

- e.g. (**ParentComponent.razor**):

```
<h3>Parent Component</h3>
<p>Message from Child: @message</p>

<ChildComponent OnMessageReceived="HandleMessage" />

@code {
  private string message = "No message yet";
  void HandleMessage(string msg) {
    message = msg;
  }
}
```

ROUTING

Routing in Blazor allows the application to navigate between different components or views based on the URL.

- It is a key feature for building *single-page applications* (SPAs), where the page does not reload when navigating between different views.
- This guarantees fundamental browser and user expectations, like bookmarking, back/forward navigation, refresh behaviour, SEO concerns, link sharing, access control, etc.
- Blazor uses a *router* to map URLs to components, and each component can represent a different page or view in the app.

ROUTING

- The router is defined in the **App.razor** file and listens for changes in the browser's address bar to determine which component to render.

```
<App>
  <Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
      <RouteView RouteData="@routeData"
                 DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
      <p>The page you're looking for could not be found.</p>
    </NotFound>
  </Router>
</App>
```

ROUTING

- Each Blazor component that should be a routable page must include the **@page** directive at the top.
- This directive binds the component to a specific URL, which is used by the router to determine which component to load.
  - e.g.:

```
@page "/home"
<h3>Hello, @Name!</h3>

@code {
  string Name = "Blazor";
}
```

ROUTING

- Blazor allows components to accept *dynamic route parameters*, similar to traditional URL query strings or route segments.
- These parameters can be accessed in the component using the **`[Parameter]`** attribute.
  - e.g.:

```
@page "/product/{Id}"
<h3>Product Details</h3>
<p>Product ID: @Id</p>

@code {
  [Parameter] public int Id { get; set; }
}
```

ROUTING

- Constraints to restrict the route parameters can also be defined.
  - e.g.:

```
@page "/product/{Id:int:min(1)}"
<h3>Product Details</h3>
<p>Product ID: @Id</p>

@code {
  [Parameter] public int Id { get; set; }
}
```

- Blazor supports nested routing. Components can be nested within each other and routed based on the parent component's URL.

IOC CONTAINER AND DI

Blazor supports built-in **dependency injection** using the .NET Core IoC container.

- Services like data access, logging, configuration, or state management can be **registered once** and **injected where needed**.
- This concept promotes **loose coupling**, **reusability**, and **testability** of components.
- ASP.NET Core uses the same DI and IoC container system in MVC / WebAPI and in Blazor as well.

IOC CONTAINER AND DI

Services must be registered with the DI container in the **`Program`** class (*Blazor WASM*) or in the **`Startup`** class (*Blazor Server*)

The usual 3 lifetimes are available:

- **Transient:** New instance every time it's requested
- **Scoped:**
  - *Blazor Server:* one instance per request session
  - *Blazor WebAssembly:* there is no concept of a per-request or per-user session, therefore it is the same as *Singleton*
- **Singleton**: one instance for the entire app

IOC CONTAINER AND DI

Beside the already familiar *constructor injection* (DI through constructor parameters), DI is supported in *Razor Components* as well:

- with Razor syntax, through the **`@inject`** directive:

```
@inject IMyService MyService
```

- in the C# code part of a component, using the **`[Inject]`** attribute on a property:

```
[Inject]
private IMyService MyService { get; set; }
```

NAVIGATION

Navigation among pages can be achieved via the **`NavLink`**

- **`NavLink`** is similar to an anchor **`<a>`** tag but provides additional functionality, such as *active link styling*
- e.g.:

```
<NavLink href="/home" class="nav-link">Home</NavLink>
<NavLink href="/about" class="nav-link">About</NavLink>
```

- when the user clicks on these links, the app will navigate to the corresponding routes without causing a full page reload

NAVIGATION

- alternatively, **NavigationManager** allows navigation from the C# code
- e.g.:

```
@inject NavigationManager Navigation
@* dependency injection *@

<button @onclick="GoToHome">Go to Home</button>

@code {
    private void GoToHome()
    {
        Navigation.NavigateTo("/home");
    }
}
```

BLAZOR COMPONENT LIFECYCLE

Blazor components have a well-defined lifecycle that determines how they are initialized, rendered, and disposed.

- Lifecycle methods are invoked automatically at specific stages
- Developers can override these methods to perform custom logic
- The component lifecycle supports both synchronous and asynchronous operations

BLAZOR COMPONENT LIFECYCLE

A Blazor component passes through several key stages during its existence. Each stage corresponds to specific lifecycle methods that can be overridden.

- Initialization (**`OnInitialized`** / **`OnInitializedAsync`**)
- Parameter setting (**`OnParametersSet`**, **`OnParametersSetAsync`**)
- Rendering (**`OnAfterRender`** / **`OnAfterRenderAsync`**)
- Disposal (**`IDisposable.Dispose`** / **`IAsyncDisposable.DisposeAsync`**)

ELTE EÖTVÖS LORÁND UNIVERSITY

INPUT MANAGEMENT IN BLAZOR

Modern web application development in .NET

DATA BINDING

Blazor supports *two-way data binding*, allowing components to interact with user input and keep the UI and the underlying C# property in sync.

- e.g.:

```
<label for="fullNameInput">Full Name:</label>
<input id="fullNameInput" @bind="fullName" />
<label for="birthYearInput">Birth Year:</label>
<input id="birthYearInput" @bind="birthYear" />

@code {
    string fullName = "Loránd Eötvös";
    int birthYear = 1848;
}
```

FORMS IN BLAZOR

Input fields can be grouped into forms, for which the **`@onsubmit`** event can be binded and handled.

- e.g. (*HTML segment of component*):

```
<form method="post" @onsubmit="HandleSubmit">
  <label for="fullNameInput">Full Name:</label>
  <input id="fullNameInput" @bind="fullName" />
  <label for="birthYearInput">Birth Year:</label>
  <input id="birthYearInput" @bind="birthYear" />
  <button type="submit">Submit</button>
</form>
```

FORMS IN BLAZOR

- e.g. (*C# code segment of component*):

```
@code {
    string fullName = "Loránd Eötvös";
    int birthYear = 1848;
    private void HandleSubmit() {
        // … do form processing …
    }
}
```

- when required, the event handler method can be asynchronous
  - should return a **`Task`** in that case

THE EDITFORM COMPONENT

Blazor provides the **`EditForm`** component for managing entire forms, including validation and submission

- **`EditForm`** seamlessly integrates with data models, validation logic, and UI rendering
- It binds an object – a (view)model – to a form, while manages submission and validation
  - The **`Model`** parameter represents the data-bound object
- Handles submit events with **`OnValidSubmit`**, **`OnInvalidSubmit`**, or **`OnSubmit`**. (The latter is executed regardless of the validation status.)
- Supports both client-side and server-side models

INPUT COMPONENTS

Blazor provides several built-in input components for form fields

- These components automatically bind to model properties and support validation
  - **`InputText`**, **`InputNumber`**, **`InputSelect`**, **`InputCheckbox`**, etc.
  - File management is also supported with **`InputFile`**
- Each input uses the **`@bind-Value`** directive to link to a model property
- E.g. (custom complex type):

```
public class Person
{
    public string FullName { get; set; }
    public int BirthYear { get; set; }
}
```

FORM SUBMISSION

- E.g. (Razor component):

```
<EditForm Model="@person" OnValidSubmit="HandleSubmit">
    <InputText @bind-Value="person.FullName" />
    <InputNumber @bind-Value="person.BirthYear" />
    <button type="submit">Submit</button>
</EditForm>

@code {
    private Person person = new Person();
    private void HandleSubmit() {
        // process the form …
    }
}
```

FORM VALIDATION

Blazor supports form validation using standard *.NET Data Annotations*, which we already used for viewmodel validation in MVC applications.

- The properties in the (view)model can be decorated with attributes like **`[Required]`**, **`[StringLength]`**, **`[Range]`**, **`[Compare]`**, **`[Url]`**, **`[EmailAddress]`**, [**`Phone`**], **`[RegularExpression]`**, etc.
- Blazor automatically validates the form before submission, when **`DataAnnotationsValidator`** is added to the form
- Validation messages can be displayed using **`ValidationSummary`** (displays all validation messages) or **`ValidationMessage`** (shows validation for a specific field)

FORM VALIDATION

- E.g. (custom complex type):

```
public class Person
{
    [Required]
    public string Name { get; set; }

    [Range(1000, 3000)]
    public int BirthYear { get; set; }
}
```

FORM VALIDATION

- E.g. (Razor component):

```
<EditForm Model="@person" OnValidSubmit="HandleSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText @bind-Value="person.FullName" />
    <ValidationMessage For="@(() => person.FullName)" />

    <InputNumber @bind-Value="person.BirthYear" />
    <ValidationMessage For="@(() => person.BirthYear)" />

    <button type="submit">Submit</button>
</EditForm>
```

CUSTOM FORM VALIDATION

Further *custom validator* attribute classes can be inherited from **`ValidationAttribute`**, implementing the **`IsValid()`** method.

Alternatively, Blazor also allows custom validation logic beyond data annotations.

- The **`IValidatableObject`** interface can be implemented for more advanced rules.
- This allows to write logic that depends on multiple fields or implements complex rules.

CUSTOM FORM VALIDATION

- e.g.:

```
public class Person : IValidatableObject
{
  public string FullName { get; set; }
  public int BirthYear { get; set; }

  public IEnumerable<ValidationResult> Validate(
    ValidationContext context)
  {
    if (!Name.Contains(" ")) {
      yield return new ValidationResult(
        "One-part names now allowed.", new[] { nameof(Name) });
    }
  }
}
```

JAVASCRIPT INTEROP IN BLAZOR

Blazor allows C# code to run in the browser via WebAssembly

- However, some functionality is only available through JavaScript (JS)
- To bridge this gap, Blazor provides JavaScript interoperability (JS Interop)
    - Enables calling JavaScript functions from C#
    - Also allows calling C# methods from JavaScript
    - Essential for integrating existing JS libraries or browser APIs

WHEN TO USE JAVASCRIPT INTEROP

While Blazor covers many common scenarios, some require JS.
Examples include:

- Accessing browser-specific APIs
  - e.g., cookies, local storage, geolocation
  - NuGet packages could provide a better alternative (e.g. *Blazored.LocalStorage*, *Blazor.GeoLocation*), hiding the JS Interop
- Using third-party JavaScript libraries
- Manipulating the DOM directly (which is not idiomatic in Blazor)

Use JS Interop only, when necessary, to improve maintainability!

CALLING JAVASCRIPT FROM C#

Blazor provides **`IJSRuntime`** service to invoke JavaScript functions

- Use **`InvokeAsync<T>("functionName", args...)`**
  - Alternatively, **`InvokeVoidAsync()`** should be used when no return value is expected
- The function must be globally accessible in JS (e.g., in window)
- Supports async calls and returns a **`ValueTask<T>`**
- E.g.:

```
@inject IJSRuntime JS
...
await JS.InvokeVoidAsync("alert", "Hello from C#");
```

CALLING C# FROM JAVASCRIPT

C# methods can also be invoked from JavaScript

- Requires marking the C# method with `[JSInvokable]`
- The method must be static, or the instance must be passed explicitly
- Useful for event callbacks, timer completions, etc.
- E.g. (in C# code):

```
[JSInvokable]
public static void ShowMessage(string message) { … }
```

- E.g. (in JS code):

```
DotNet.invokeMethodAsync('AssemblyName’,
  'ShowMessage', 'Hello Blazor');
```

JAVASCRIPT INTEROP IN BLAZOR

Best practices and considerations:

- Minimize interop use to keep Blazor code clean and maintainable
- Always validate and sanitize inputs to/from JavaScript
- Prefer **`invokeVoidAsync`** for functions that don't return values
- Keep JS files organized and clearly separate interop functions
- Use **`try-catch`** around interop calls to handle JS errors gracefully

ELTE EÖTVÖS LORÁND UNIVERSITY

AUTHENTICATION IN BLAZOR

Modern web application development in .NET

AUTHENTICATION IN BLAZOR

- Authentication strategy depends on the hosting model
  - *Blazor Server:* stateful SignalR circuit on the server. By default it uses cookie-based auth (like classic ASP.NET Core).
  - *Blazor WebAssembly:* runs entirely in the browser, no server session. Usually implements token-based auth (*OpenID/OAuth2*).
- Common abstractions Blazor provides across all models:
  - **`AuthenticationStateProvider`**: service exposing the current user
  - **`<AuthorizeView>`**: declarative show/hide based on roles / policies
  - **`[Authorize]`**: attribute on routable components
  - **`AuthorizeRouteView`**: route-level guards

BLAZOR SERVER – COOKIE AUTHENTICATION

- The component tree lives on the server, communicating via a SignalR
  - We will discuss SignalR as the next topic.
- Because the initial request is a normal HTTP request, the server can issue an auth cookie just like in MVC / Razor Pages
- ASP.NET Core Identity and the cookie middleware work unchanged
- The cookie is sent on the initial page load; the SignalR circuit is then bound to that authenticated user
- No tokens needed for calling the app's own server logic, since the code runs server-side already

BLAZOR WEBASSEMBLY – TOKEN AUTHENTICATION

- The app runs in the browser as WebAssembly — there is no server session to attach a cookie to
- Access token (JWT) stored in browser memory
  - Attached to outgoing API calls
  - Best to implement OpenID / OAuth2 with an Identity Provider
- Use the following NuGet package for simplicity: **`Microsoft.AspNetCore.Components.WebAssembly.Authentication`**
- Why not cookies?
  - No server-rendered HTML response to set a Set-Cookie on
  - CORS / cross-site cookie rules make it impractical

EXAMPLE

Task: Implement a new Blazor WebAssembly frontend for the travel agency webservice (**`TravelAgency.Admin.Blazor`**)

- The frontend should provide administrative functionality: adding, editing and deleting buildings and managing their images
- Create separate, reusable components for the page contents
- The backend was already developed throughout the previous lectures, and is ready to be uses as a *RESTful* service
  - communication is done through *DTO* types (defined in **`TravelAgency.Shared`**), reference it in the Blazor project
- Use the Bootstrap CSS library for styling for the frontend

EXAMPLE

Authentication will be implemented with simple token exchange, as WebAssemblys does not support cookies

- Extend the **`UsersController`** in the **`TravelAgency.WebAPI`** project with a new **`LoginToken()`** action, returning the access token in the *X-Access-Token* header
- Configure a simple opaque token handler (**`SimpleTokenHandler`**) in the application (**`Program`**) as an alternative login source, processing the *Authorization* header of client requests
- The Blazor webapp will be served at *https://localhost:7087*, enable this origin for CORS in the backend configuration (see **`Program.cs`**)

WHERE TO GO NEXT? ECOSYSTEM & LIBRARIES

You don't have to build every UI component from scratch.

A rich ecosystem of Blazor component libraries is available:

- MudBlazor: Material Design, free & open source
- Microsoft Fluent UI Blazor: Official Microsoft library
- Radzen Blazor: Free UI components + low-code IDE
- Blazor Bootstrap: Bootstrap 5 components
- Telerik / DevExpress / Syncfusion: Commercial, enterprise-grade Blazor component libraries

ELTE EÖTVÖS LORÁND UNIVERSITY

UI & E2E TESTING WITH BLAZOR

Modern web application development in .NET

Lecture 10

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

RECAP ON TESTING

So far, we have focused on testing the backend of our web application

- We have covered *unit testing* with various frameworks like **`MSTest`**, **`xUnit`**, and **`NUnit`**.
  - Each test isolates one method or class; dependencies are mocked.
  - Fast, deterministic, and run on every commit.
- We have also written *integration tests* for the backend APIs.
  - They verify controllers, middleware, and EF Core queries against a real (or in-memory) database.
  - ASP.NET Core supported this with a special **`TestServer`**
- These are essential — but they never renders or interacts with the UI.

THE TESTING PYRAMID

Tests form a pyramid: many cheap tests at the base, a handful of expensive tests at the top.

- **Unit tests** — narrow, fast, isolated. Verify one class.
- **Integration tests** — broader, slower. Verify modules working together.
- **UI / component tests** — render a single component into a test DOM and assert on its output. We will use *bUnit*.
- **End-to-end (E2E) tests** — drive the full app through a real browser, like a real user. Slow and more fragile, but the only tests that can catch bugs with complex setup. We will use *Playwright*.

Higher in the pyramid means more confidence per test, but also more cost and flakiness.

WHAT IS UI TESTING?

UI testing verifies that a single component renders correctly and reacts to user input as expected.

- The component is rendered into an **in-memory test DOM** — there is no real browser, no real network.
- A typical UI test:
  - renders the component with specific parameters,
  - queries the resulting DOM with CSS selectors (**`Find`**, **`FindAll`**),
  - triggers DOM events (**`Click`**, **`Input`**, **`Change`**),
  - asserts on the resulting markup, text content, or attributes.
- UI tests are almost as fast as unit tests and can run without browser.

WHAT IS END-TO-END TESTING?

E2E testing drives the entire application through a real browser, exactly the way a real user would.

- Everything runs together: Blazor frontend, ASP.NET Core backend, database, authentication, JavaScript, CSS — nothing is mocked.
- A typical E2E test simulates a user journey, for example:
  - open the home page, click "Sign in", fill the form, submit,
  - navigate to the product list, add an item to the cart,
  - proceed to checkout, verify the order confirmation page.
- E2E tests give the highest confidence — but they are also the slowest and most fragile, because they depend on every layer of the stack.
- E2E tests does not substitute UI component tests.

UI TESTS VS E2E TESTS

	UI tests	E2E tests
Scope	One component in isolation	The whole app, all layers
Runtime	In-memory test DOM, no browser	Real browsers (Chromium, Firefox, WebKit)
Speed	Milliseconds per test	Seconds per test
Dependencies	Services and JS interop are mocked	Real backend, database, and browser
What they catch	Component bugs: markup, bindings, event handlers	Integration bugs: routing, auth, network, JS interop

They serve a different purpose — a healthy test suite uses both.

WHY TEST THE UI OF A BLAZOR APP?

- A Blazor component is not just markup — it contains real C# logic: parameters, state, lifecycle hooks, event callbacks, async data loading.
- Razor mixes **HTML**, **C#**, and **data bindings** in the same file. Bugs in any of them only surface when the component is rendered.
- Common UI bugs to catch:
  - a refactor breaks an **`@bind`** or a **`[Parameter]`** wiring,
  - a conditional **`@if`** block silently disappears for one role,
  - an **`EventCallback`** stops firing because of a signature change,
  - an **`@onclick`** handler throws when the model is null.
- Manual click-testing every PR does not scale. The goal is to catch UI regressions in CI, in seconds, with no browser involved.

BUNIT — BLAZOR COMPONENT TESTING

`bUnit` is the de-facto standard library for testing Blazor components

- It renders components into an **in-memory test renderer** — no browser, no SignalR connection, no JS runtime.
- It is not a test framework itself — it plugs into the one you already use: **`xUnit`**, **`MSTest`**, or **`NUnit`**.
  - We need to add the **`bunit`** NuGet package additionally.
  - It will provide the **`TestContext`** base class, capable of in-memory component rendering.
- Tests run as fast as unit tests; we can set breakpoints in component code and step through, etc.

WHAT BUNIT CAN DO

- **Render** components with parameters, cascading values, **`ChildContent`**.
  - **Assert on re-renders**: markup snapshots (**`MarkupMatches`**), render counts, **`EventCallback`** invocations
- **Query** the rendered DOM with CSS selectors via **`Find`** and **`FindAll`**, or assert on the raw markup string.
- **Trigger DOM events**: **`Click`**, **`Input`**, **`Change`**, **`Submit`**, **`KeyDown`**, etc.
- **Inject services** into the test DI container (real or mocked) so the component sees its dependencies.
- **Stub JavaScript interop** — no real **`IJSRuntime`**, no browser. Set up **`JSInterop`** handlers and verify calls.
- **Wait for async work** to finish with **`WaitForState`** / **`WaitForAssertion`** — bUnit polls until the condition holds or times out.

SETTING UP A BUNIT TEST PROJECT

- Create a normal xUnit test project, then add the **`bunit`** NuGet package and reference your component project.

```
dotnet new xunit -n MyApp.Tests
cd MyApp.Tests
dotnet add package bunit
dotnet add reference ../MyApp/MyApp.csproj
```

- Test classes inherit from **`TestContext`** — bUnit's base class that gives access to the renderer, services, and JS interop stubs.

```
public class CounterTests : TestContext
{
    [Fact]
    public void CounterStartsAtZero() { /* ... */ }
}
```

RENDERING A COMPONENT

- **`RenderComponent<T>()`** instantiates a component, runs its lifecycle, and returns an **`IRenderedComponent<T>`**.

```
var cut = RenderComponent<Counter>();

// Markup is the rendered HTML as a string
Assert.Contains("Current count: 0", cut.Markup);
```

- **`cut`** stands for *"component under test"*. From it you can read the markup, find elements, fire events, and re-render.
- Pass parameters with the **`ComponentParameter`** builder or — more readable — the **`ParameterCollectionBuilder`** lambda overload:

```
var cut = RenderComponent<Greeting>(p => p
    .Add(g => g.Name, "Anna"));
```

FINDING ELEMENTS AND ASSERTING

- **`Find(selector)`** returns the first matching element; **`FindAll(selector)`** returns all of them. CSS selectors only.

```
var button   = cut.Find("button.btn-primary");
var rows     = cut.FindAll("table tbody tr");
var heading  = cut.Find("h1");

Assert.Equal("Welcome", heading.TextContent);
Assert.Equal(3, rows.Count);
```

- For larger DOM checks, prefer **`MarkupMatches`** — it ignores insignificant whitespace and attribute order:

```
cut.MarkupMatches(@"<h1>Welcome</h1>
                    <p>Hello, Anna!</p>");
```

- *Tip:* a failing **`Find`** throws **`ElementNotFoundException`** with the rendered markup printed — usually enough to debug without a debugger.

TRIGGERING DOM EVENTS

- Once we have an element, we can call extension methods like **`Click`**, **`Input`**, **`Change`**, or **`Submit`** on it. *bUnit* re-renders synchronously afterwards.

```
var cut = RenderComponent<Counter>();
var btn = cut.Find("button");

btn.Click();
btn.Click();
btn.Click();
Assert.Contains("Current count: 3", cut.Markup);
```

- For form fields, **`Change`** / **`Input`** to inject the new value:

```
cut.Find("input[type=text]").Change("Anna");
cut.Find("form").Submit();
```

- All event helpers also have async overloads (**`ClickAsync`**, **`ChangeAsync`**) for handlers that return **`Task`**.

PARAMETERS, CHILDCONTENT, CASCADING VALUES

- The **ParameterCollectionBuilder** overload of **RenderComponent** lets you set parameters, **ChildContent**, and cascading values strongly typed:

```
var cut = RenderComponent<Card>(p => p
    .Add(c => c.Title, "Hello")
    .AddChildContent("<p>Inner markup</p>")
    .AddCascadingValue(new Theme { Dark = true })
    .Add(c => c.OnSelected, () => clicked = true));
```

- **EventCallback** parameters are passed as plain delegates — bUnit wraps them in an **EventCallback** automatically.
- After the initial render we can change parameters with **SetParametersAndRender** — bUnit re-runs the lifecycle so we can verify **OnParametersSet** behavior:

```
cut.SetParametersAndRender(p => p.Add(c => c.Title, "Bye"));
```

MOCKING SERVICES AND JS INTEROP

- Register service mocks (we use *Moq*) directly on the **`Services`** property, exactly like ASP.NET Core DI:

```
var repo = new Mock<IProductRepository>();
repo.Setup(r => r.GetAllAsync())
    .ReturnsAsync(new[] { new Product("Tea", 5) });
Services.AddSingleton(repo.Object);

var cut = RenderComponent<ProductList>();
Assert.Contains("Tea", cut.Markup);
```

- For JavaScript interopability, **`TestContext`** exposes **`JSInterop`**. Set up the calls your component makes — anything else throws.

```
JSInterop.Setup<string>("prompt", "Your name?")
         .SetResult("Anna");
var cut = RenderComponent<NameAsker>();
JSInterop.VerifyInvoke("prompt");
```

WAITING FOR ASYNC RENDERS

- Components often load data in **`OnInitializedAsync`**. E.g. the first render shows a spinner; the markup we want to assert appears later.
- Use **`WaitForState`** or **`WaitForAssertion`** — bUnit polls until the condition holds (default timeout is 1 second):

```
var cut = RenderComponent<ProductList>();
// Wait until the list is populated
cut.WaitForAssertion(() =>
    Assert.Contains("Tea", cut.Markup));
// Or wait for a specific component state
cut.WaitForState(() => cut.Instance.IsLoaded);
```

- Avoid **`Task.Delay`** or **`Thread.Sleep`** in tests — they slow the suite and hide real timing bugs. Always wait on a condition.
- Increase the timeout for slower flows:: **`WaitForAssertion(..., TimeSpan.FromSeconds(3))`**.

WHAT BUNIT CANNOT DO

- bUnit renders into a **simulated DOM** — it is not a real browser, so a number of things are out of scope:
  - CSS layout, computed styles, and visual rendering — **`:hover`**, **`@media`** queries, animations are not evaluated.
  - Real JavaScript — **`IJSRuntime`** calls must be stubbed.
  - Routing — **`NavigationManager`** is faked. You can verify navigations were requested, but no page actually loads.
  - Browser APIs — **`localStorage`**, **`fetch`**, **`WebSocket`** are not present.
- If a test needs any of those, it is no longer a UI test — it has become an E2E test, which is what Playwright is for.
- The right mental model: bUnit verifies the **Razor + C#** half of a component; Playwright verifies the **browser-and-stack** half.

ELTE EÖTVÖS LORÁND UNIVERSITY

END-TO-END (E2E) TESTING

Modern web application development in .NET

MOVING UP THE PYRAMID: E2E TESTING

- bUnit answers whether a component renders and behave correctly in isolation? **E2E tests answers whether the whole application works for a real user, with all the moving parts together?**
- Simple examples only E2E can verify:
  - the user can actually log in — auth cookies, redirect, real database, real password hashing all work;
  - client-side JavaScript libraries (charts, maps, editors) actually load data from the backend and render properly;
  - responsive breakpoints, and accessibility attributes look right in real browsers.
- We use ***Microsoft Playwright*** — a modern, fast, multi-browser automation framework with first-class .NET support.

MICROSOFT PLAYWRIGHT

- **`Playwright`** is an open-source browser automation library from Microsoft.
  - Beside .NET, it supports Java, Python and TypeScript as well.
- **One API — three browsers**: Chromium, Firefox, and WebKit (Safari engine). The same test runs against all three.
- **Auto-waiting** by default: every action waits for the element to be visible, enabled, and stable before clicking or typing. Most flakiness simply disappears.
- **Network interception**, multiple browser contexts, file uploads, downloads, geolocation, mobile emulation — all built in.
- **Tooling**:
  - **`playwright codegen`** — record interactions and produce C# test code;
  - **`playwright open`** — interactive REPL for trying selectors;
  - **`Trace Viewer`** — full timeline replay with screenshots, network, console.

WHAT PLAYWRIGHT CAN DO

- **Drive any modern browser**: navigate, click, type, hover, drag, upload, download, screenshot, record video.
- **Locator-first API**: `Page.GetByRole`, `.GetByLabel`, `.GetByText` — selectors that match what users see, not brittle CSS paths.
- **Auto-waiting assertions**: `Expect(locator).ToBeVisibleAsync()`, `ToHaveTextAsync()` — retry until the condition passes or a timeout elapses.
- **Multiple contexts** in one test — simulate two users in the same browser process, each with its own cookies and session.
- **Mock the network**: intercept any request, return canned responses, or block third-party tracking scripts to keep tests fast and deterministic.
- **Headless or headed**, switch with a single launch option.
- **CI-ready**: single Docker image with all browsers preinstalled.

INSTALL PLAYWRIGHT AND WRITE A FIRST TEST

- Add the NuGet package to a test project, build, then run the install script to download the browser binaries:

```
dotnet add package Microsoft.Playwright.NUnit
dotnet build
pwsh bin/Debug/net8.0/playwright.ps1 install
```

- Inherit from **`PageTest`** to get a **`Page`** property already wired to a fresh browser context per test.

```
public class HomePageTests : PageTest {
    [Test]
    public async Task HomePageShowsWelcome() {
        await Page.GotoAsync("https://localhost:7001/");
        await Expect(Page.GetByRole(AriaRole.Heading,
            new() { Name = "Welcome" })).ToBeVisibleAsync();
    }
}
```

LOCATORS: FIND ELEMENTS LIKE A USER

- Prefer **`user-facing`** locators over CSS or XPath — they are stable across refactors and self-document the test.

```
// By accessible role and name
Page.GetByRole(AriaRole.Button, new() { Name = "Sign in" })

// By visible label or text
Page.GetByLabel("Email address")
Page.GetByText("Forgot password?")

// By placeholder, alt text, or test id
Page.GetByPlaceholder("Search...")
Page.GetByTestId("product-row")
```

- Locators **`auto-wait`** — the action retries until the element is visible, enabled, and stable, or fails with a clear timeout error.

LOCATORS: SCOPING AND FILTERING

- Chain locators to scope a search **inside** another — the inner query only looks within the matched element.

```
var row = Page.GetByRole(AriaRole.Row,
    new() { Name = "Tea" });

await row
    .GetByRole(AriaRole.Button, new() { Name = "Delete" })
    .ClickAsync();
```

- Use **`Filter`** to narrow a set of matches by visible text or by a nested locator:

```
Page.GetByRole(AriaRole.Listitem)
    .Filter(new() { HasText = "out of stock" })
    .First();
```

- *Tip:* prefer **`data-testid`** attributes on Razor components for stable selectors that survive CSS and class renames.

NETWORK MOCKING AND AUTH REUSE

- Intercept any backend call with **`Page.RouteAsync()`** — return canned responses so tests are fast and deterministic.

```
await Page.RouteAsync("**/api/products", async route =>
    await route.FulfillAsync(new()
    {
        ContentType = "application/json",
        Body = """[{ "name": "Tea" }]"""
    }));
```

- Don't log in once per test — sign in once, save the storage state, then reuse it on every other context:

```
await Context.StorageStateAsync(new() { Path = "auth.json" });
// Later:  new() { StorageStatePath = "auth.json" }
```

DEBUGGING E2E TESTS

- **Run with a real browser visible**: **`Headless = false`** + **`SlowMo`** — watch the test happen.

```
PLAYWRIGHT_DEBUG=1 dotnet test
```

- **Trace Viewer** is a nice feature. Configure it to record on every failed test:
  - a full timeline of actions, network calls, console logs,
  - DOM snapshots before and after every step,
  - open the trace with **`playwright show-trace trace.zip`**
- **Codegen** speeds up writing new tests dramatically — interact with the app, get C# code to copy:

```
pwsh playwright.ps1 codegen https://localhost:7001
```

- **In CI**, attach traces and screenshots to failed test runs — almost always enough to diagnose without running locally.

COMPARISON TO OTHER TOOLS

- **Selenium** is the historical baseline — works, but its API is verbose, waiting is manual, and parallelism in .NET is painful.
- **Cypress** has a great developer experience, but it is JavaScript-only and runs tests **inside** the browser tab. That model has trade-offs:
  - limited multi-tab / multi-origin testing,
  - no native iframe support across origins,
  - no .NET API — we need to write tests in JavaScript / TypeScript.
- **Playwright** was designed after both:
  - first-class .NET API — tests sit next to bUnit tests in the same solution,
  - auto-waiting, retries, multiple contexts, multiple browsers, no flakiness boilerplate,
  - Trace Viewer is uniquely good for diagnosing intermittent failures.
- For a Blazor/.NET project, Playwright is the obvious default.

BEST PRACTICES

- **Stay low in the pyramid by default**. A well-tested component with bUnit needs only a thin smoke-test in Playwright.
- **Test behavior, not implementation**. Assert on what the user sees (text, roles, visible markup), not on internal class names or component state.
- **Keep tests independent**. Each test sets up its own data and cleans up after itself. Order of execution must not matter.
- **No sleeps**. Use **`WaitForAssertion`** / Playwright auto-waits — never **`Thread.Sleep`** or **`Task.Delay`**.
- **Run the full suite in CI on every PR**. Fast feedback is the whole point.
- **On failure, attach evidence**: rendered markup for bUnit, traces and screenshots for Playwright. Diagnose without re-running locally.
- **Treat flaky tests as bugs**, not noise. Never *"retry until green"*.

EXAMPLE

Task: Test the administrative fronted of the travel agency website.

- Create a new *xUnit* test project (**`TravelAgency.Admin.Blazor.Test`**) and add *bUnit* as a NuGet dependency
  - implement some component tests (**`BuildingManagerTest`**)
  - set up mock data and inject a **`FakeNavigationManager`** in the constructor
- Create another *xUnit* test project, add **`Microsoft.Playwright.Xunit`** as a dependency (**`TravelAgency.Admin.Blazor.E2ETest`**)
  - implement some E2E test workflows (**`AuthTest`**, **`BuildingTest`**)
  - extract the common settings into a **`TestBase`** class

ELTE EÖTVÖS LORÁND UNIVERSITY

INTRODUCTION TO SIGNALR

Modern web application development in .NET

Lecture 11

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

REAL-TIME COMMUNICATION

Modern applications often need to update the UI without user interaction.

- Examples include live chat, multiplayer games, live dashboards, collaborative tools, etc.
- Traditional HTTP is insufficient for low-latency, bi-directional communication.
- This is where technologies like WebSocket and *ASP.NET Core SignalR* become essential.

WEBSOCKET

WebSocket is a protocol that provides a persistent, full-duplex communication channel over a single TCP connection.

- Unlike HTTP, which follows a request-response model, WebSockets allow **continuous, bidirectional communication between client and server**, including the server to push data to clients at any time.
- This significantly reduces overhead and improves responsiveness.
- It is ideal for scenarios requiring low latency and real-time updates.
- WebSocket supports text and binary data transfer as well.
- Uses the **`ws://`** and **`wss://`** protocol names in URLs.

WEBSOCKET HANDSHAKE

The WebSocket protocol begins with a standard HTTP request, upgraded to WebSocket via headers, e.g.:

Client Request:

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
```

Server Response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

Once this handshake is done successfully, both sides can exchange data directly over TCP.

WEBSOCKET DATAFRAME

WebSocket communicates with *data frames*. Structure for each frame:

- **FIN** (1 bit): indicates whether this is the final fragment (WebSocket supports frame fragmentation)
- **Opcode** (4 bits): type of the frame (*text*, *binary*, *ping*, *pong*, *close*, etc.):
- **Mask** (1 bit): security feature to mask the payload with a random key
- **Payload Length** (7+ bits): length of the message
- **Payload Data**: actual content of the message

WEBSOCKET MESSAGE FLOW

The WebSocket Message Flow is the following:

1. Client sends handshake to upgrade HTTP → WebSocket
2. Server responds with 101 Switching Protocols
3. Connection established over TCP
    - Either side can send messages at any time (no need to wait)
    - Heartbeat via *ping/pong* ensures connection is alive
4. Sending a *close frame* gracefully terminates the connection

HTTP AND WEBSOCKET COMPARISON

Feature	HTTP	WebSocket
Communication Model	Request → Response	Bidirectional, event-based
Connection Type	Stateless (short-lived)	Stateful (long-lived)
Latency	High (new connections, headers per request)	Low (constant open connection after initial handshake)
Server Push Support	No native support (requires polling or SSE)	Yes (server can send data anytime)
Transport Protocol	HTTP/1.1 or HTTP/2 over TCP	TCP (after HTTP handshake upgrade)
Scalability	Well-supported by CDNs, caches, load balancers	Requires WebSocket-aware infrastructure
Security	HTTPS (HTTP over TLS)	WSS (WS over TLS)

WHAT IS ASP.NET SIGNALR?

ASP.NET Core SignalR is a library to easily add features based on real-time functionality (like live chat, dashboards, or notifications) to ASP.NET Core based web application.

Key features:

- Automatic transport negotiation
- Automatic connection management
- Group messaging
- Scalable backplane support (e.g., *Redis*, *Azure SignalR Service*)
- Strong client-side support across platforms

AUTOMATIC TRANSPORT NEGOTIATION

When WebSockets are unavailable (e.g., due to firewalls or proxies), SignalR falls back to:

- **Server-Sent Events (SSE):** Server pushes text-based updates to client (unidirectional). This is included in the HTML5 standard and supported by modern browsers.
- **Long Polling:** Client polls the server repeatedly for updates, simulating real-time.

SignalR hides these details and always chooses the best transport automatically.

SIGNALR CORE COMPONENTS

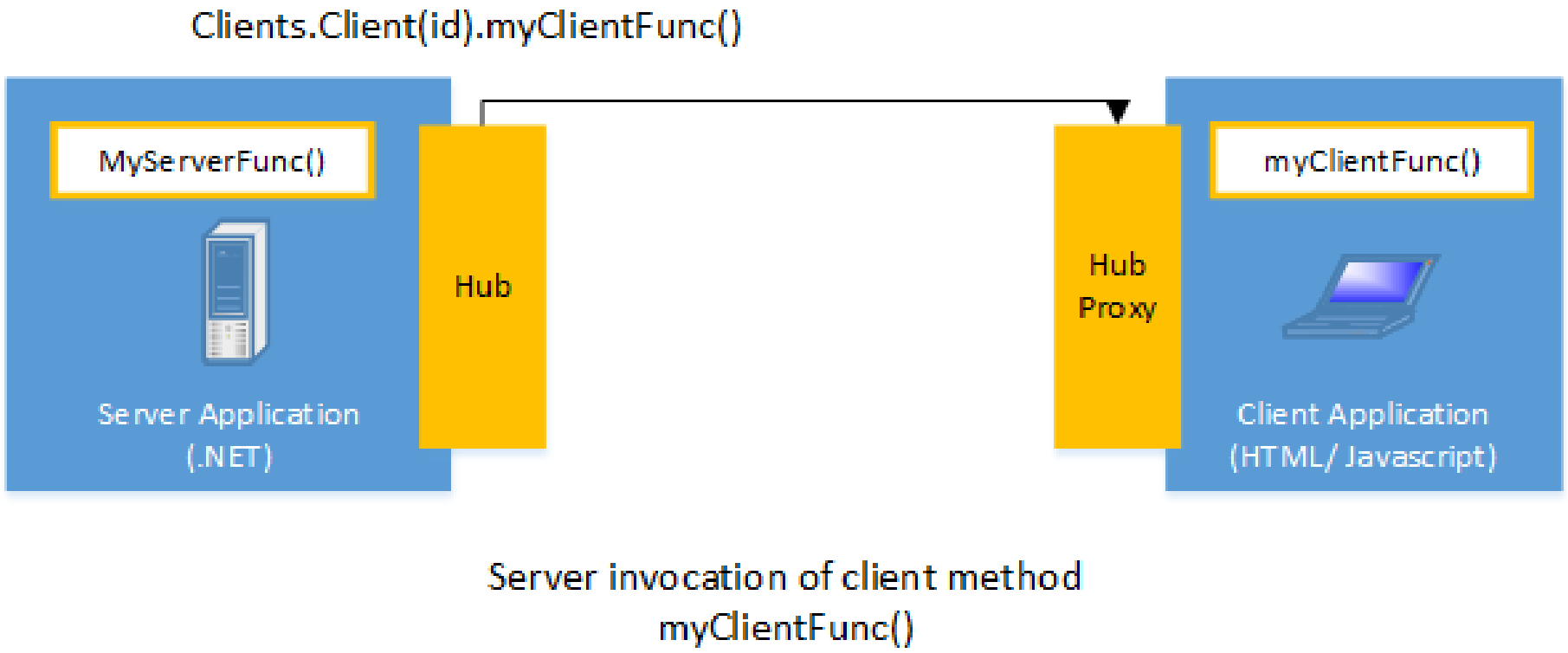
SignalR consists of three main elements:

1. **Hub:** The central point of communication on the server
2. **Clients:** Applications connected via WebSocket/HTTP
3. **Connection Protocol:** Handles message format and transport negotiation

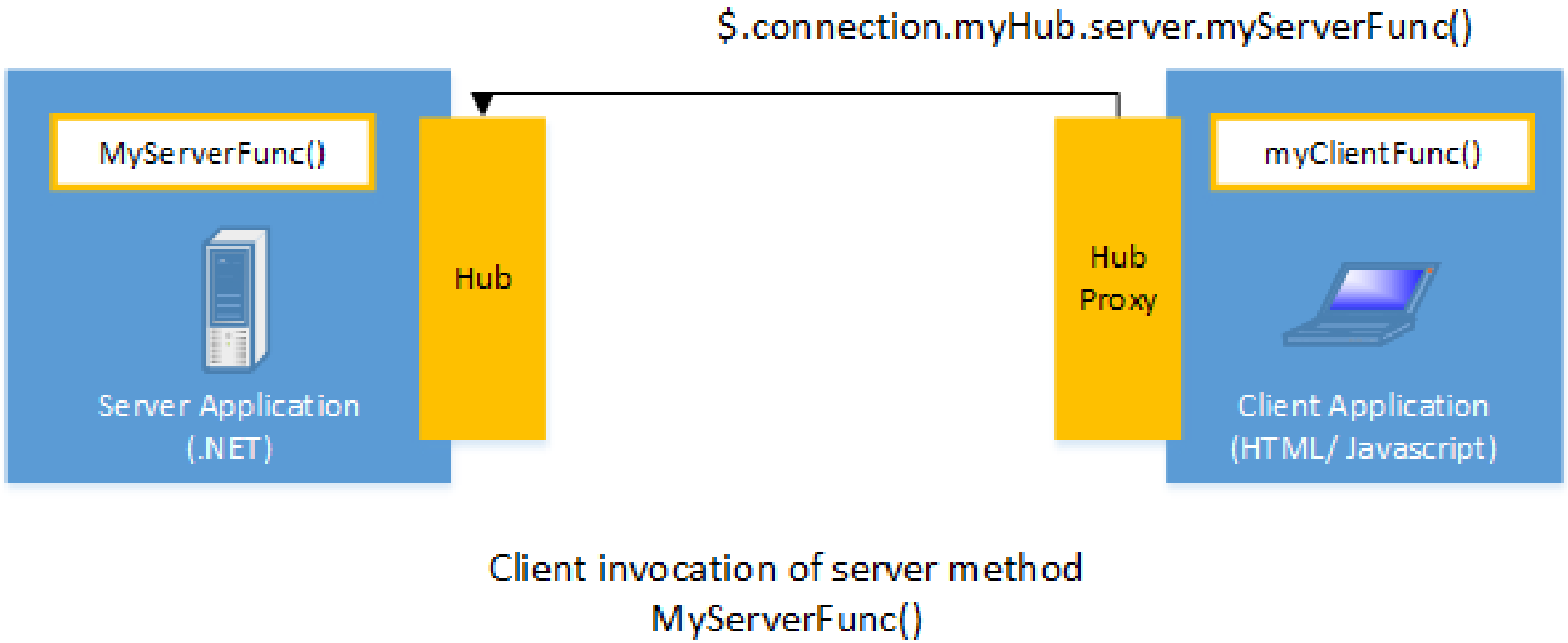
SignalR provides a high abstraction level API to communicate between hubs and clients through *remote procedure calls (RPC)*.

- Allows the server to call methods on connected clients and vice versa.

SERVER → CLIENT COMMUNICATION EXAMPLE



CLIENT → SERVER COMMUNICATION EXAMPLE



DEFINING A SIGNALR HUB

Creating a SignalR Hub in ASP.NET Core is done easily, inheriting from the **Hub** base class:

```
public class ChatHub : Hub
{
  public async Task SendMessage(string user, string message)
  {
    await Clients.All.SendAsync(
      "ReceiveMessage", user, message);
  }
}
```

STRONGLY TYPED HUBS

Beside *dynamic dispatching*, defining a contract between the hub and the client is also supported, using strongly typed hubs:

```
public interface IChatClient
{
  Task ReceiveMessage(string user, string message);
}

public class ChatHub : Hub<IChatClient>
{
  public async Task SendMessage(string user, string message) {
    await Clients.All.ReceiveMessage(user, message);
  }
}
```

CONTROLLING MESSAGE TARGETS

The Clients property hub has key properties and methods to manage message targets:

- **`Clients.All`**: calls a method on all connected clients
- **`Clients.Caller`**: calls a method on the client invoking the hub method
- **`Clients.Others`**: calls a method on all connected clients except the client that invoked the method

- **`Clients.Group(…)`**: calls a method on all connections in the group
- **`Clients.User(…)`**: calls a method on all connections associated with a specific user (e.g. to notify the user's other devices)

CONFIGURING SIGNALR ON THE SERVER

Adding SignalR support in a ASP.NET Core web app can be done through installing the **`Microsoft.AspNetCore.SignalR`** NuGet package.

- Then in the **`Program`** (or in the **`ConfigureServices()`** of **`Startup`**) enable it, by registering it to the service collection collection: **`builder.Services.AddSignalR();`**
- Define the new endpoints (typically one endpoint / hub), in the **`Program`** (or int **`Configure()`** of **`Startup`**):

```
var app = builder.Build();
…
app.MapHub<ChatHub>("/chatHub");
```

SIGNALR CLIENT SUPPORT

SignalR officially supports the following client platforms. Each client library is designed to work seamlessly with the server-side SignalR hubs.

- .NET
- JavaScript
- TypeScript
- Java
- Swift

Further programming environments have 3rd party support (e.g. Python).

.NET CLIENT

Install the **`Microsoft.AspNetCore.SignalR.Client`** NuGet package for your console, desktop or mobile client application.

- Then, it is ready to use, e.g. calling a method on a server hub:

```
var connection = new HubConnectionBuilder()
  .WithUrl("https://localhost:5001/chatHub")
  .WithAutomaticReconnect()
  .Build();

await connection.StartAsync();

await connection.InvokeAsync(
  "SendMessage", ".NET Client", "Hello from .NET!");
```

.NET CLIENT

- Or receiving incoming messages from the server:

```
var connection = new HubConnectionBuilder()
  .WithUrl("https://localhost:5001/chatHub")
  .WithAutomaticReconnect()
  .Build();

connection.On<string, string>(
  "ReceiveMessage", (user, message) => {
    Console.WriteLine($"{user}: {message}");
  }
);
await connection.StartAsync();
```

JAVASCRIPT CLIENT

Install the **`@microsoft/signalr`** NPM package, or load through a CDN.

- Then, it is ready to use, e.g. calling a method on a server hub:

```
<script src="~/lib/signalr/signalr.js"></script>
<script>
  const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .withAutomaticReconnect()
    .build();

  connection.start().then(() => {
    connection.invoke("SendMessage",
                      "JS Client", "Hi from JS!");
  });
</script>
```

JAVASCRIPT CLIENT

- Or receiving incoming messages from the server:

```
<script src="~/lib/signalr/signalr.js"></script>
<script>
  const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .withAutomaticReconnect()
    .build();

  connection.on("ReceiveMessage", (user, message) => {
    console.log(`${user}: ${message}`);
  });

  connection.start();
</script>
```

TYPESCRIPT CLIENT

Install the **`@microsoft/signalr`** NPM package, or load through a CDN.

- Then, it is ready to use, e.g. calling a method on a server hub:

```
import * as signalR from "@microsoft/signalr";

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .withAutomaticReconnect()
    .build();

await connection.start();
await connection.invoke(
  "SendMessage", "TS Client", "Message from TypeScript");
```

TYPESCRIPT CLIENT

- Or receiving incoming messages from the server:

```
import * as signalR from "@microsoft/signalr";

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .withAutomaticReconnect()
    .build();

connection.on(
  "ReeiveMessage", (user: string, message: string) => {
    console.log(`${user}: ${message}`);
});

await connection.start();
```

JAVA CLIENT

Integrating SignalR differs on the build system used in the Java project.

- Using Gradle, add the following line to the dependencies section of your *build.gradle* file:
  `implementation 'com.microsoft.signalr:signalr:8.0.15'`
- Using Maven, add the following lines inside the <dependencies> element of your *pom.xml* file:

```
<dependency>
    <groupId>com.microsoft.signalr</groupId>
    <artifactId>signalr</artifactId>
    <version>8.0.15</version>
</dependency>
```

JAVA CLIENT

- Then, it is ready to use, e.g. calling a method on a server hub and listening for server message:

```
HubConnection hubConnection = HubConnectionBuilder
  .create("https://localhost:5001/chatHub")
  .build();

hubConnection.on("ReceiveMessage", (user, message) -> {
  System.out.println(user + ": " + message);
}, String.class, String.class);

hubConnection.start().blockingAwait()
hubConnection.send("SendMessage",
                   "Java User", "Hello from Java!");
```

SWIFT CLIENT

Requires the `signalr-client-swift` package (installable from the Swift Package Manager into the Package.swift file)

- Build the connection, register a handler, then start:

```
let connection = HubConnectionBuilder()
    .withUrl(url: "https://localhost:5001/chatHub")
    .withAutomaticReconnect()
    .build()

await connection.on("ReceiveMessage") {
    (user: String, message: String) in
        print("\(user) says: \(message)")
}
try await connection?.start()
```

SWIFT CLIENT

- Invoke a hub method to send a message:

```
let connection = HubConnectionBuilder()
    .withUrl(url: "https://localhost:5001/chatHub")
    .withAutomaticReconnect()
    .build()

try await connection?.start()
try await connection?.invoke(method: "SendMessage",
            arguments: ["Swift User", "Hello from Swift!"])
```

HUB METHOD RETURN VALUES

Hub methods can return values directly to the calling client, not just broadcast to other clients.

- This completes the RPC picture: the client invokes a hub method and awaits a response, just like a regular async function call.
- E.g.:

```
public class ChatHub : Hub {
  public async Task<List<string>> GetMessageHistory() {
    List<string> messageHistory = // fetch data ...
    return messageHistory;
  }
}
```

ELTE EÖTVÖS LORÁND UNIVERSITY

USERS & GROUPS IN SIGNALR

Modern web application development in .NET

Lecture 12

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

GROUP MANAGEMENT IN HUBS

Groups in SignalR allows to target a subset of connected clients.

- Each connection can belong to zero or more groups.
- Groups are dynamic: you can add or remove connections at runtime.
- Typical use cases include:
  - Private chat rooms
  - Team collaboration workspaces
  - Broadcasting updates to selected users, e.g. viewing the same user interface of the application

GROUP MANAGEMENT IN HUBS

Groups are just labels in the Hub, they don't have to be created explicitly

- Users sharing the same group label are in the same group
- Add a client to a group:

```
await Groups.AddToGroupAsync(
  Context.ConnectionId, "mygroup");
```

- Remove a client to a group:

```
await Groups.RemoveFromGroupAsync(
  Context.ConnectionId, "mygroup");
```

- Send message to a group of clients:

```
await Clients.Group("mygroup").SendAsync(
  "ReceiveMessage", "Server", "Hello Group!");
```

GROUP MANAGEMENT IN HUBS

- Send messages to a group of clients in a weakly-typed hub:

```
public class ChatHub : Hub
{
  public async Task SendMessage(
    string group, string user, string message)
  {
    await Clients.Group(group).SendAsync(
      "ReceiveMessage", user, message);
  }
}
```

GROUP MANAGEMENT IN HUBS

- Send messages to a group of clients in a strongly-typed hub:

```
public interface IChatClient {
  Task ReceiveMessage(string user, string message);
}
public class ChatHub : Hub<IChatClient>
{
  public async Task SendMessage(
    string group, string user, string message)
  {
    await Clients.Group(group)
                 .ReceiveMessage(user, message);
  }
}
```

GROUP MANAGEMENT IN HUBS

Further methods for finer control over group management:

- **`Clients.Group()`**: calls method on all connections in the group
- **`Clients.Groups()`**: calls method on multiple groups of connections
- **`Clients.GroupExcept()`**: calls a method on all connections in the specified group, except the specified clients
- **`Clients.OthersInGroup()`**: calls a method on a group of connections, excluding the client that invoked the hub method

AUTHENTICATION IN SIGNALR

SignalR integrates seamlessly with ASP.NET Core authentication middleware, which we have already used for WebAPI development.

- Clients must send a valid access token or cookie when connecting
- Token can be transparent (e.g. a JWT) or opaque as well
- The token can be transported as a *cookie*, in the *query string* or in a *HTTP header* (*Authorization* conventionally)
    - The use of Authorization header is only supported if the SignalR client is not running in a browser.
    - In case a WebSocket or SSE connection is initiated from a browser, it does not allow the use of an *Authorization* header.
    - The preferred solution is to include it in the *query string*.

AUTHENTICATION IN SIGNALR

- Cookie-based authentication works out of the box
  - Same origin is a requirement, otherwise CORS issues will arise
- JWT based authentication from query string requires configuration:

```
builder.Services.AddJwtBearer(options => {
  // ...
  options.Events = new JwtBearerEvents
  {
    OnMessageReceived = context => {
      context.Token = context.Request.Query["access_token"];
      return Task.CompletedTask;
    }
  };
});
```

AUTHENTICATION IN SIGNALR

- Custom (non-JWT) token authentication can be implemented by defining a custom **AuthenticationHandler**:

```
public class SimpleTokenHandler :
  AuthenticationHandler<AuthenticationSchemeOptions>
{
  // ... ctor ...
  protected override async
    Task<AuthenticateResult> HandleAuthenticateAsync()
  {
      var token = Request.Query["access_token"].FirstOrDefault();
      if (/* token validation fails */)
        return AuthenticateResult.Fail("Invalid Token");
      var ticket = new AuthenticationTicket(/* ... */);
      return AuthenticateResult.Success(ticket);
  }
}
```

AUTHENTICATION IN SIGNALR

- Then configuring the web application to use it:

```
builder.Services.AddAuthentication(
  options => {
    options.DefaultAuthenticateScheme = "Token";
    options.DefaultChallengeScheme = "Token";
  })
  .AddScheme<AuthenticationSchemeOptions,
             SimpleTokenHandler>("Token", null);
```

- Compare to Lecture 9: this is almost identical to the custom opaque token authentication for *TravelAgency*, but using query string instead of the *Authorization* header.

AUTHENTICATION ON THE CLIENT

SignalR clients must be configured for each hub to pass the access token as a *query string* parameter.

- For a TypeScript / JavaScript client:

```
const connection = new signalR.HubConnectionBuilder()
  .withUrl("/chatHub", {
    accessTokenFactory: () => authToken
  })
  .withAutomaticReconnect()
  .build();
```

- Where **`accessTokenFactory`** can be a function returning a **`string`** – or even a **`Promise<string>`** to support asynchronous execution.

AUTHENTICATION ON THE CLIENT

- For a .NET client:

```
var connection = new HubConnectionBuilder()
  .WithUrl("https://localhost:5001/chatHub", options => {
    options.AccessTokenProvider = () =>
      Task.FromResult("access token value");
  })
  .WithAutomaticReconnect()
  .Build();
```

- Where the **`AccessTokenProvider`** can be a custom **`Task<string>`**, loading the access token from a storage.

AUTHENTICATION ON THE CLIENT

- For a Java client:

```
HubConnection hubConnection = HubConnectionBuilder
  .create("https://localhost:5001/chatHub")
  .withAccessTokenProvider(Single.defer(() -> {
    return Single.just("access token value");
  }))
  .build();
```

- Uses RxJava's **`Single`** to provide async tokens. **`Single`** is reactive pattern, an observable emitting exactly one item.

HUB CONTEXT

For each hub, the base class provides the **`Context`** property with various information related to the connection:

- **`ConnectionId`**: gets the unique connection identifier (assigned by SignalR). There's one connection ID for each connection.
- **`UserIdentifier`**: gets the user identifier (case sensitive).
- **`User`**: gets the **`ClaimsPrincipal`** object associated with the user.
- **`Items`**: key/value collection that can be used to share data within the scope of the connection. Data will be persisted for the connection across different hub method invocations.

AUTHORIZATION IN SIGNALR

The **`[Authorize]`** attribute can be added to SignalR hubs or just individual methods for authorization with the ASP.NET Core middleware

- This ensures that only authenticated and authorized users can send or receive messages or perform sensitive actions.
- The user identifier for a connection can be accessed by the **`Context.UserIdentifier`** property in the hub.
  - By default, SignalR uses the **`ClaimTypes.NameIdentifier`** from the *ClaimsPrincipal* associated with the connection
- You can also check **`Context.User.Identity`** or use **`Context.User.IsInRole(…)`** manually inside hub methods

USER MANAGEMENT IN SIGNALR

A single user in SignalR can have multiple connections to an application

- E.g. a user could be connected on their desktop and their phone.
- Each device has a separate SignalR connection, but they're all associated with the same user.
- If a message is sent to the user, all connections associated with that user receive the message, e.g.:

```
await Clients.User("someuser")
              .SendAsync("ReceiveMessage", message);
```

- E.g. send a message to all clients of the current user:

```
if (Context.UserIdentifier != null)
  await Clients.User(Context.UserIdentifier)
               .SendAsync("ReceiveMessage", message);
```

CONNECTION MANAGEMENT IN SIGNALR

Connection and disconnection can be handled by overriding the **`OnConnectedAsync()`** and **`OnDisconnectedAsync()`** methods in a hub.

- Useful for managing groups, logging or cleaning up resources
- Examples:
  - Online/offline indicators for users
  - Real-time user list
  - Reassigning resources when someone leaves
- Both methods are guaranteed to be called, however the disconnection is not necessarily detected immediately on the server

CONNECTION MANAGEMENT IN SIGNALR

E.g. maintain active user list in a chat hub:

```
public class ChatHub : Hub {
  private static readonly HashSet<string> _users = [];
  public override async Task OnConnectedAsync() {
    var userId = Context.UserIdentifier;
    if (userId != null) {
      _users.Add(userId);
      // Notify all other clients that this user came online
      await Clients.Others.SendAsync("UserOnline", userId);
      // Send the user list to the newly connected client
      await Clients.Caller.SendAsync("OnlineUsers", _users);
    }
    await base.OnConnectedAsync();
} }
```

CONNECTION MANAGEMENT IN SIGNALR

E.g. maintain active user list on disconnection, logging unexpected disconnection events through the passed exception:

```
public override async Task OnDisconnectedAsync(
  Exception? exception) {
  var userId = Context.UserIdentifier;
  if (userId != null) {
    _onlineUsers.Remove(userId);
    await Clients.Others.SendAsync("UserOffline", userId);
  }
  if (exception != null) {
    ... // Handle unexpected disconnection
  }
  await base.OnDisconnectedAsync(exception);
}
```

CONNECTION MANAGEMENT IN SIGNALR

Client disconnection can occur for various reasons

- The client can gracefully close the SignalR connection, by calling the **`stop()`** / **`StopAsync()`** method on the hub connection object
  - The JS/TS SignalR client also detects the disconnection when the browser unloads the page and initiates graceful disconnection
- If the client closes abruptly (or the client cannot notify the server in time), the server detects the dropped connection after a timeout
  - When the client just lost connection, it will try to recover the connection, in which case the same connection will be kept
  - The server detects lost connections through transport-level timeouts (*keep-alive* + *ping-pong*)

SEND MESSAGES FROM OUTSIDE A HUB

The SignalR hub is the core abstraction for sending messages to clients connected to the SignalR server.

- It's also possible to send messages from other places in your app using the **`IHubContext`** service.
- An instance of **`IHubContext`** can be fetched via dependency injection and injected into an API controller, middleware, service class, etc.

```
public class HomeController : Controller {
  private readonly IHubContext<NotificationHub> _hubContext;
  public HomeController(IHubContext<NotificationHub> hubContext) {
  _hubContext = hubContext;
  }
}
```

EXAMPLE

Task: Add a real-time support chat functionality to the travel agency website, using SignalR

- Logged in users should be able to open a support chat to contact and message the administrators
- Logged in administrators should be able to see all chat threads and respond to them
- Authentication and authorization should be performed
- Messages should be groupped, regular users should only see (and receive) their own chat thread

EXAMPLE

Add a new class library project (**`TravelAgency.SignalR`**) to the solution

- Implement the **`SupportChatHub`**, receiving / sending chat messages
  - Messages should be grouped (chat threads)
  - Admins should receive all messages
  - Listen on **`OnConnectedAsync`** to monitor connecting clients
  - Accessing the hub should be an authorized action
- For simplicity, store chat messages in an in-memory dictionary
- Use the **`Microsoft.AspNetCore.SignalR`** NuGet package
- Reference the *SignalR project* in the *WebAPI project* and host it

EXAMPLE

Extend the **`TravelAgency.Angular`** project with a chat functionality

- **`ChatService`** will be responsible for the SignalR network communication management
- **`ChatComponent`** will be the new Chat component (UI + view logic)
- Refactor the authentication workflow (**`UserService`**), to use the simple token-based approach (added for the Blazor client), since SignalR cannot depend on the cookie-based method
- Update the development proxy configuration (**`proxy.config.json`**) to avoid CORS policy issues during development
- Use the **`@microsoft/signalr`** NPM package

EXAMPLE

Extend the **`TravelAgency.Admin.Blazor`** project with an administrative chat functionality

- Add a new **`SupportChat`** page, that lists all chat threads and displays the messages in the selected one
- DTO types go to the **`TravelAgency.Shared`** project, so that they are not redundant between the WebAPI and the .NET client project
- Use the **`Microsoft.AspNetCore.SignalR.Client`** NuGet package

ELTE EÖTVÖS LORÁND UNIVERSITY

DEPLOYMENT, SCALING, CACHING

Modern web application development in .NET

Lecture 13

Dr. Máté Cserép

mcserep@inf.elte.hu

https://mcserep.web.elte.hu/

DEPLOYING ASP.NET APPLICATIONS

After development, the application must be deployed to a web server

- The configuration (**`appsettings.json`**) can be specified for a *development*, *staging* (testing) and *production* version (**`appsetting.<environment>.json`**)
- The active environment is controlled by the **`ASPNETCORE_ENVIRONMENT`** environment variable
  - If unspecified, the default at runtime is *Production*
  - It can also be set per profile in **`launchSettings.json`** (Visual Studio defaults this to *Development*)
- Choose the environment based on what features and diagnostics you want enabled

DEPLOYING ASP.NET APPLICATIONS

- Usual configuration setups:
  - **Development:** Enables detailed error pages, Webpack hot reloading, runtime Razor compilation, and more.
  - **Staging:** Used for final testing before production. Similar to *production* but may include extra logging or test data.
  - **Production:** Optimized for performance and security. Disables development features such as detailed exceptions.
- *Razor views* (**`.cshtml`** files) are also compiled by default when the project is compiled or deployed. Runtime recompilation can also be enabled, and is automatically enabled when using Visual Studio.

```
builder.Services.AddControllersWithViews()
                .AddRazorRuntimeCompilation();
```

DEPLOYING ASP.NET APPLICATIONS

The .NET Core framework is platform-independent, so the compilation and execution steps are also available as terminal commands:

- Use the **`dotnet run`** command to compile and run your .NET Core project.
- Use the **`dotnet build`** command to compile only.
- The **`dotnet publish -o out_dir`** command places the .NET Core project in the specified directory, copying all compiled binary files, configuration files, and dependencies to a single location.
  - The web application can then be run without the SDK, only the .NET Core Runtime is required: **`dotnet MyWebApp.dll`**
  - Use the **`--self-contained`** flag to bundle the .NET Runtime too

KESTREL AND REVERSE PROXIES

- **Kestrel** is the cross-platform web server built into ASP.NET Core
  - It is fast, lightweight, but rarely exposed directly in production
- Typically a **reverse proxy** sits in front of Kestrel and handles:
  - TLS termination, HTTP/2 and HTTP/3, request buffering
  - Compression, URL rewriting, static files, load balancing
- Common choices: Apache, Nginx, IIS (Windows), YARP, or Azure Front Door as a cloude-native solution
- Behind a proxy, enable forwarded headers so the app sees the real client IP and scheme: **`app.UseForwardedHeaders(...)`**

PRODUCTION HARDENING: HTTPS & SECURITY HEADERS

- **HTTPS everywhere:** `UseHttpsRedirection()` + `UseHsts()` in order to force secure transport
- **Certificates:** terminate TLS at the proxy in production; locally use `dotnet dev-certs https --trust`
- **Errors:** `UseExceptionHandler("/Error")` replaces detailed Development pages; never expose stack traces in Production
- **Cookies:** set `Secure`, `HttpOnly`, `SameSite`
- **Security headers:** use CSP, X-Content-Type-Options, X-Frame-Options headers via middleware or proxy
- **Patch dependencies:** review regularly, preferably in the CI `dotnet list package --vulnerable`

CONTAINERIZING ASP.NET CORE

- A container packages the app + its runtime + dependencies into a single, immutable image
  - Same image runs locally, in CI, in staging, and in production
- Microsoft publishes official base images on **`mcr.microsoft.com`**
  - **`dotnet/sdk`** for building, **`dotnet/aspnet`** for running
- Use a **multi-stage Dockerfile** to keep the final image small:

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
```

SCALING OUT: ORCHESTRATION PLATFORMS

- Running *many* containers — across machines, with health checks, rolling updates, autoscaling — needs an **orchestrator**
- **Kubernetes (k8s):** the de-facto standard for Docker orchestration
  - Managed offerings: AKS (Azure), EKS (Amazon), GKE (Google)
- Lighter-weight options on Azure
  - **Azure Container Apps** — serverless, scale-to-zero
  - **Azure App Service for Containers** — fully managed PaaS
- What the orchestrator expects from your app:
  - Liveness / readiness **health endpoints**
  - Graceful shutdown on **`SIGTERM`** (handled by ASP.NET Core)
  - Stateless workers — push state to a database, cache, or queue

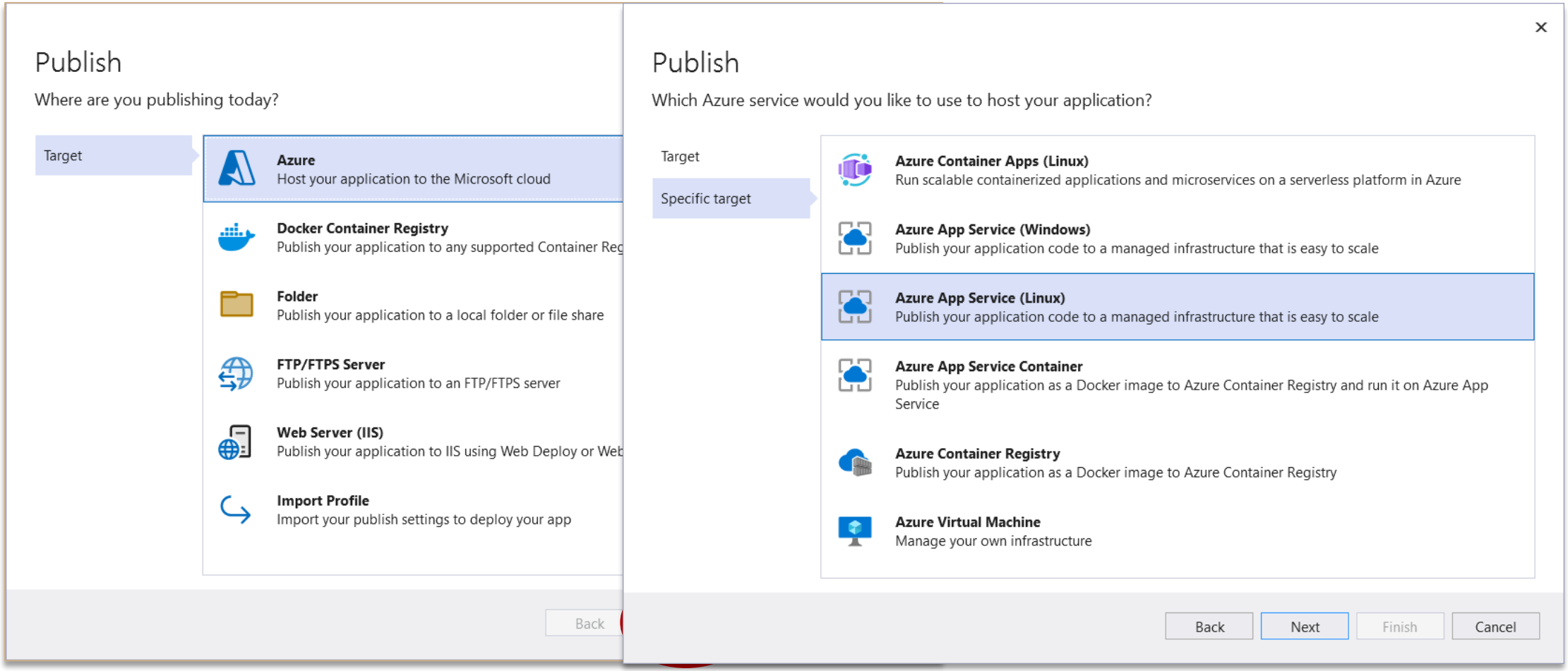
DEPLOYING TO MICROSOFT AZURE

- Pay-as-you-go pricing: you only pay for the resources you actually consume
- Free $200 credit for 30 days on registration
- Many services have a *Free tier* (F1) suitable for small demos
- Authentication: Microsoft account, GitHub, or work / school account

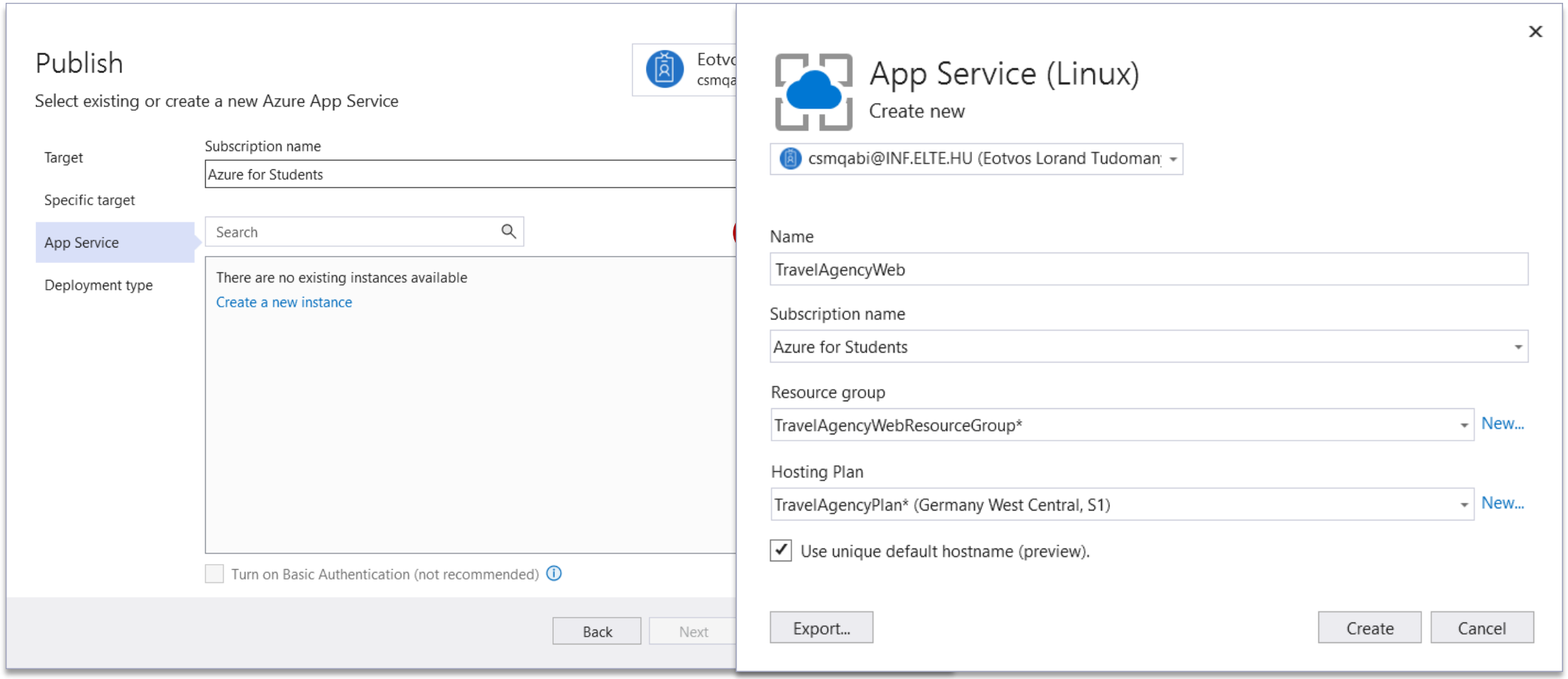
Basic tier	Cores / RAM	Linux	Windows
B1	1 vCPU / 1.75 GB	~$13/mo	~$55/mo
B2	2 vCPU / 3.5 GB	~$25/mo	~$110/mo
B3	4 vCPU / 7 GB	~$51/mo	~$220/mo

Basic tier: 10 GB storage, manual scale up to 3 instances.
Source: azure.microsoft.com/pricing/details/app-service (2026.05.)

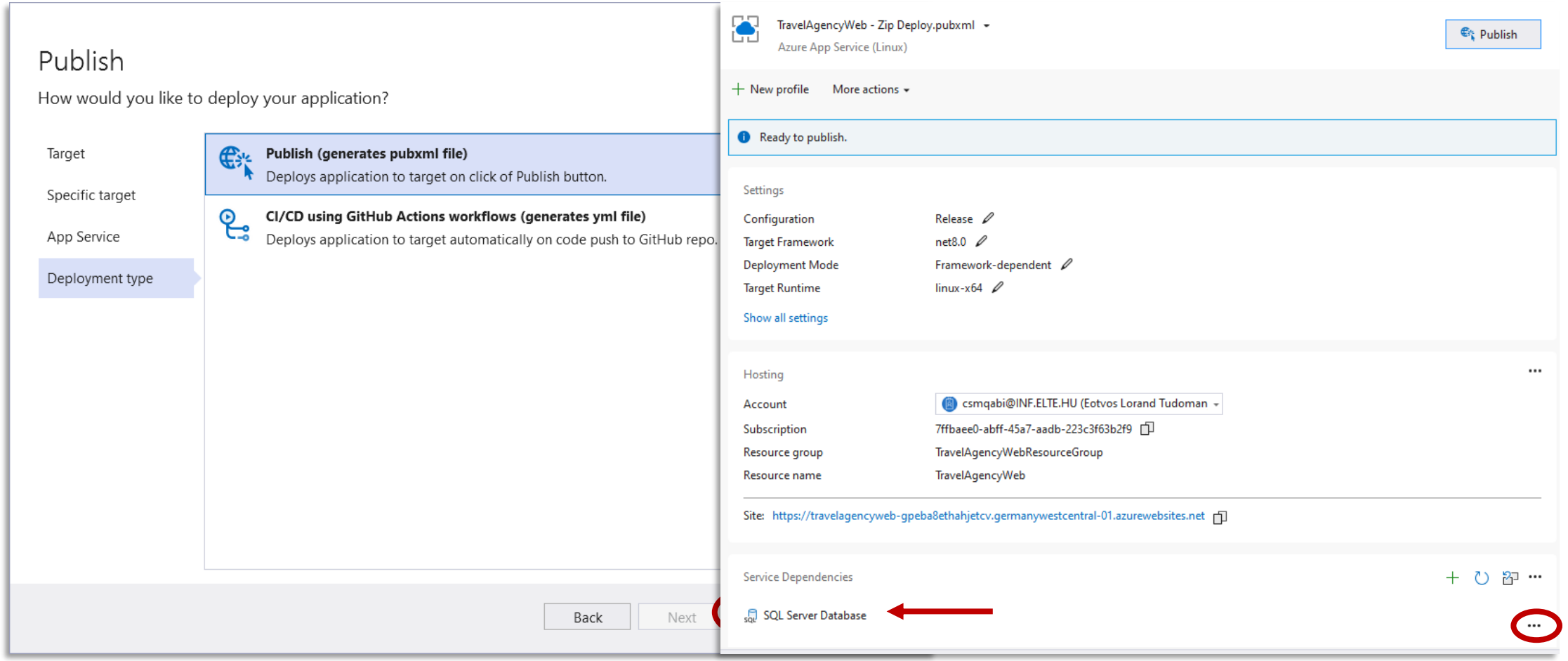
DEPLOYING AN APP SERVICE FROM VISUAL STUDIO



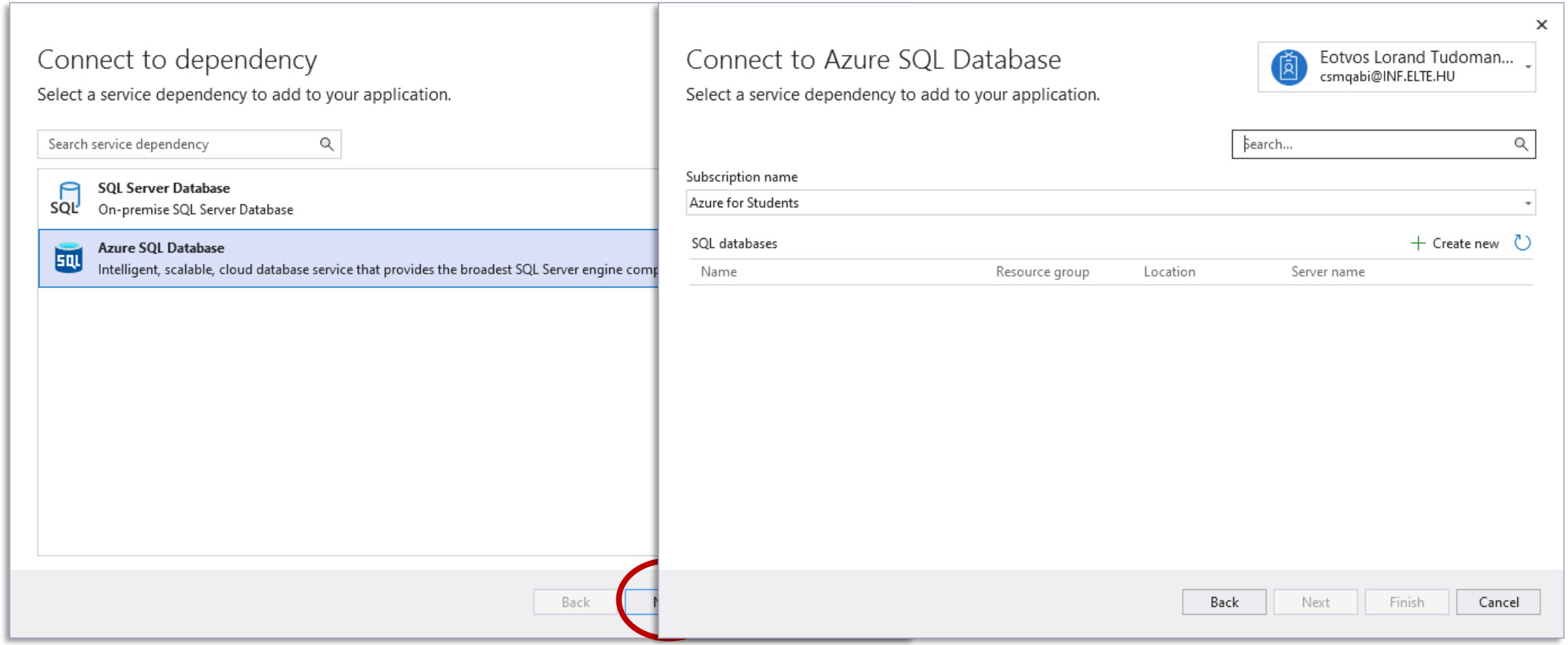
CREATING AN APP SERVICE: CONFIGURATION



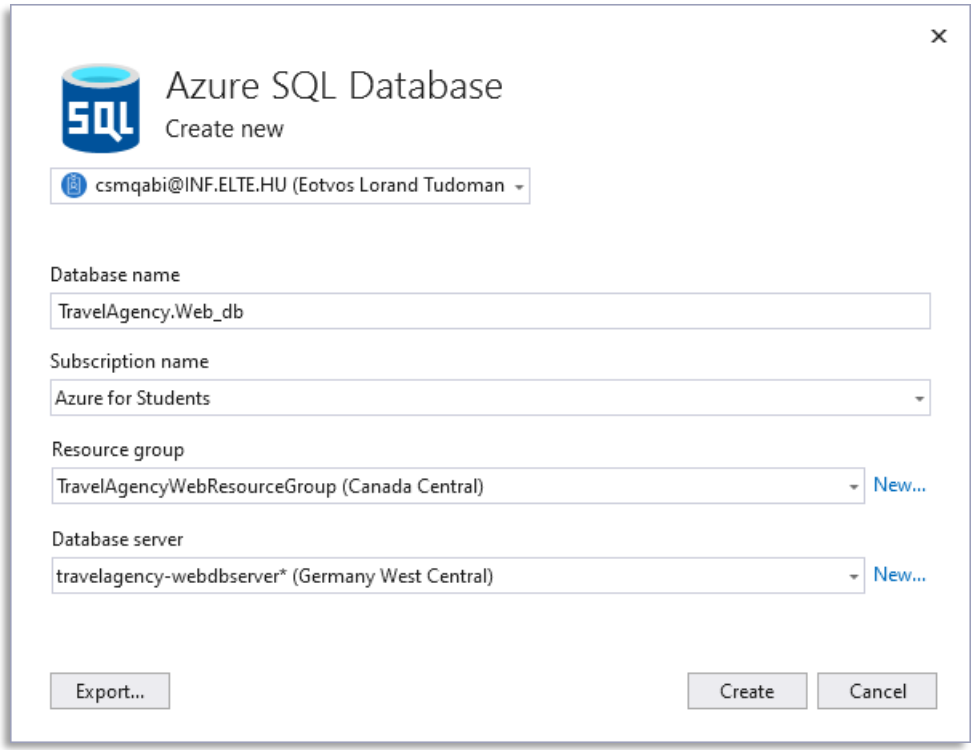
CREATING AN APP SERVICE: REVIEW AND CREATE

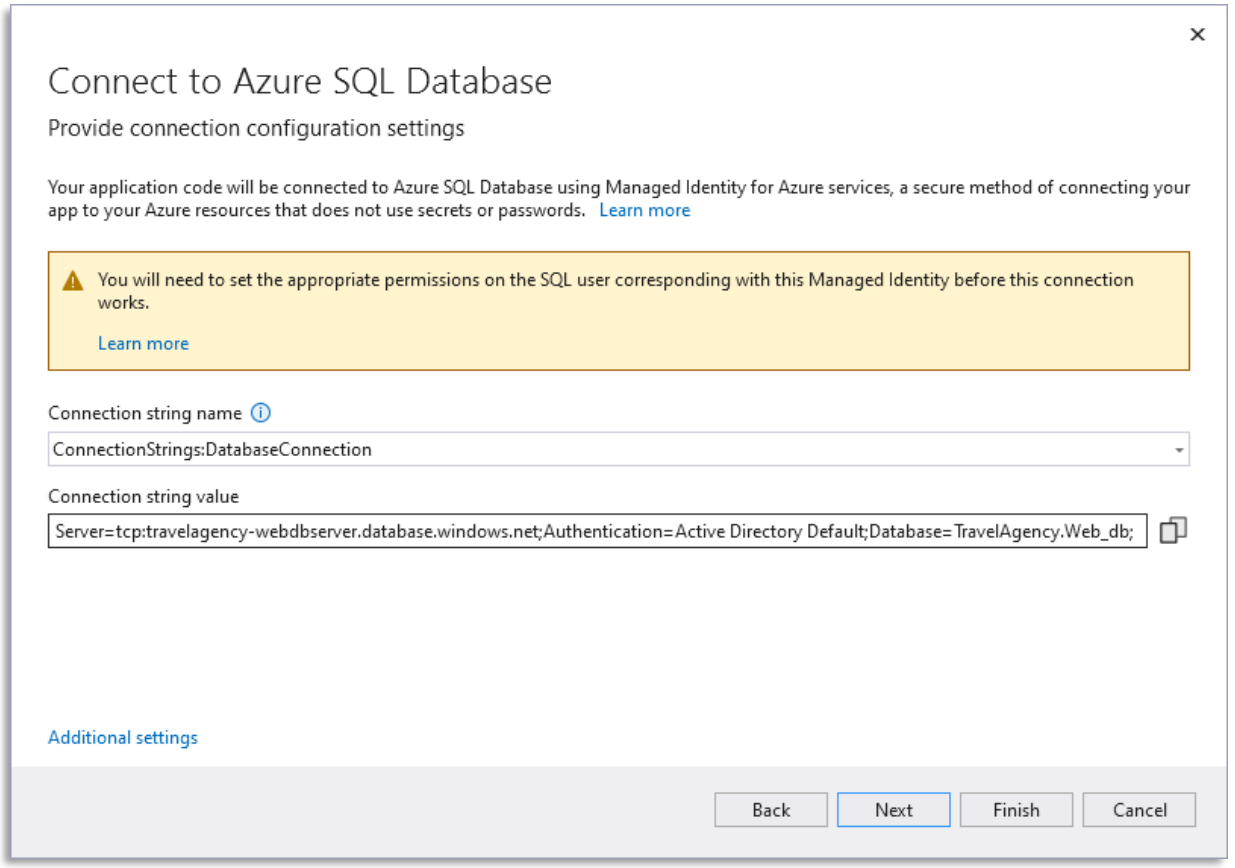


CREATING AN APP SERVICE: DATABASE MANAGEMENT



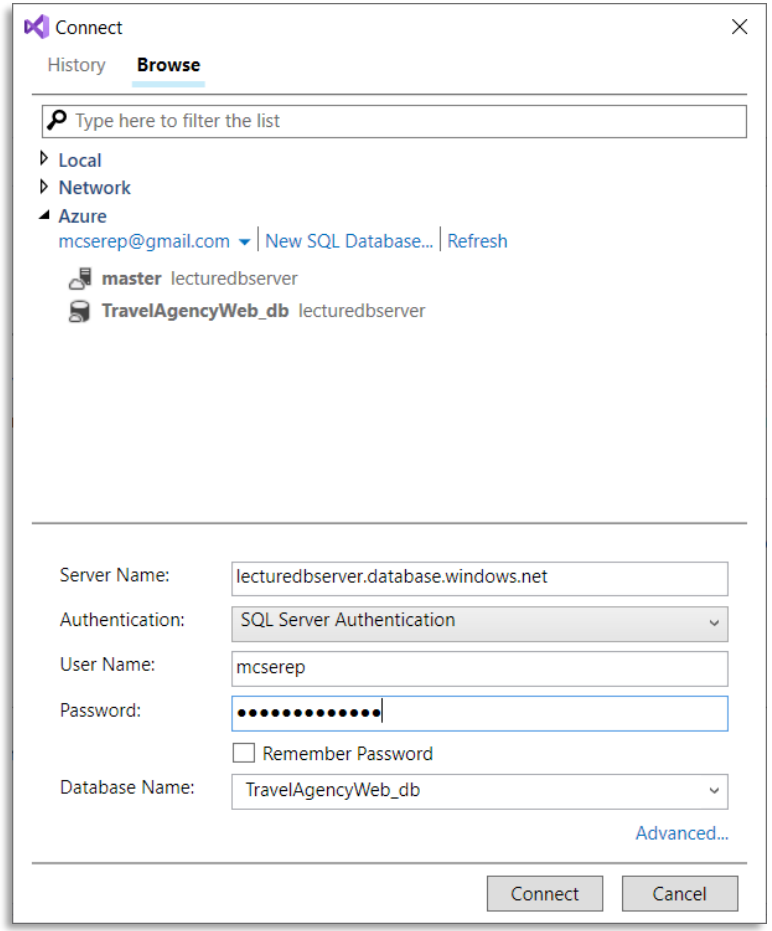
CREATING AN APP SERVICE: DATABASE CONFIGURATION





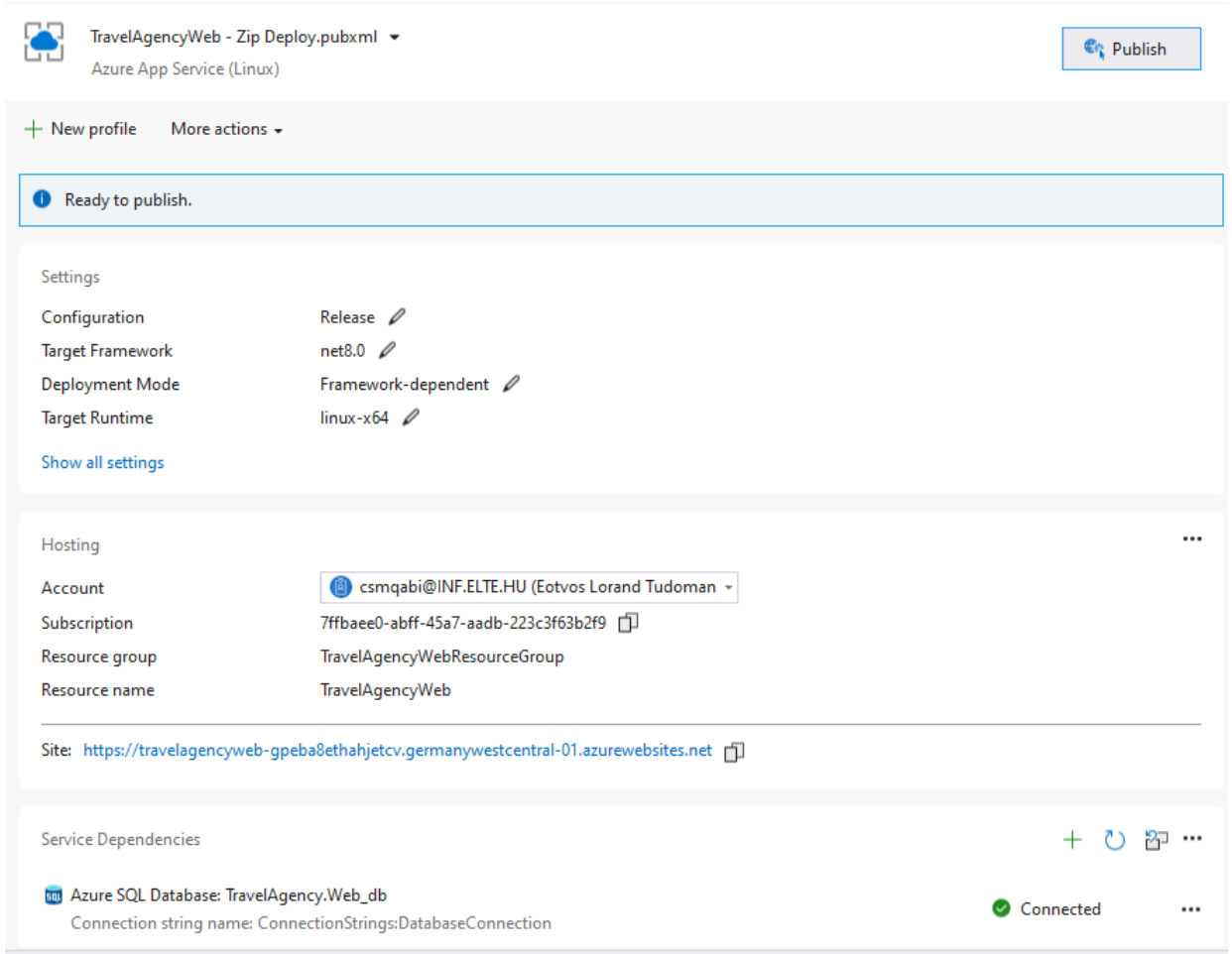
CONNECTING TO AZURE SQL FROM VISUAL STUDIO

- In Visual Studio:
  SQL Server Object Explorer → *Add SQL Server* → *Azure*
  - Given signed in with our Azure account, the available remote databases can be browsed directly
- The same is configurable in *SQL Server Management Studio*



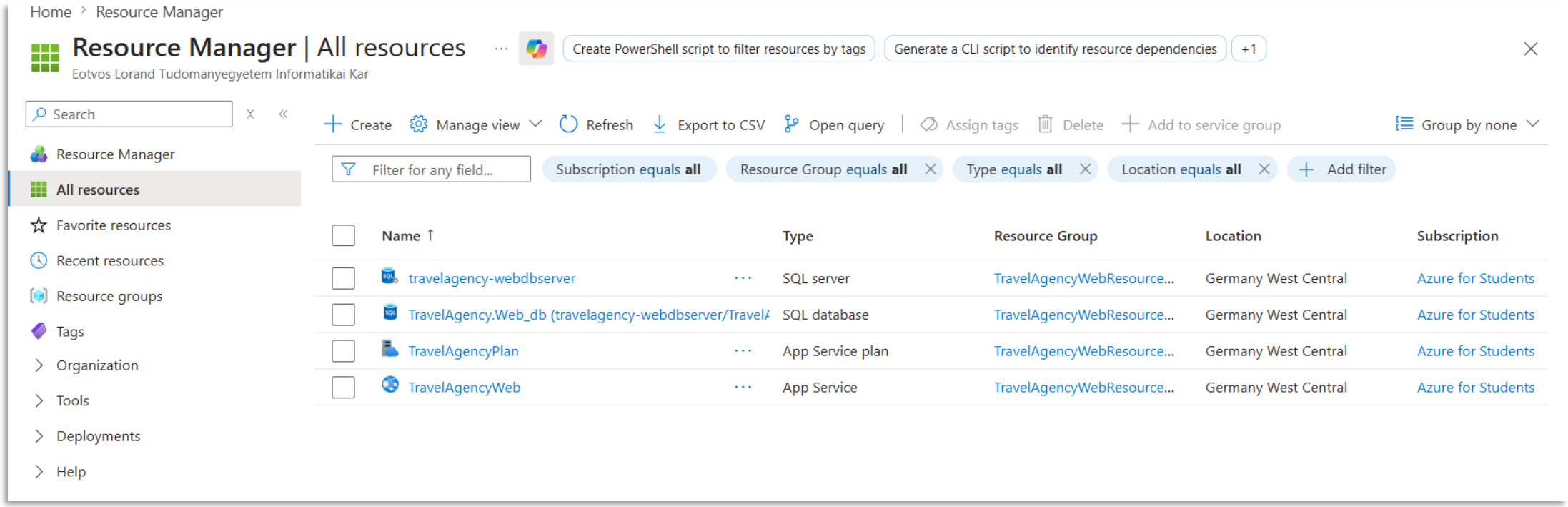
PUBLISHING THE APPLICATION FROM VISUAL STUDIO

- Visual Studio builds in *Release* configuration and uploads the artifacts via *Web Deploy*
- The publish profile is stored in the project – subsequent deploys are one click
- **Framework-dependent** deployment ships only the app – relies on the .NET runtime installed on the App Service (smaller, faster)
- **Self-contained** deployment bundles the runtime with the app – larger payload, but immune to host runtime upgrades



THE MICROSOFT AZURE PORTAL

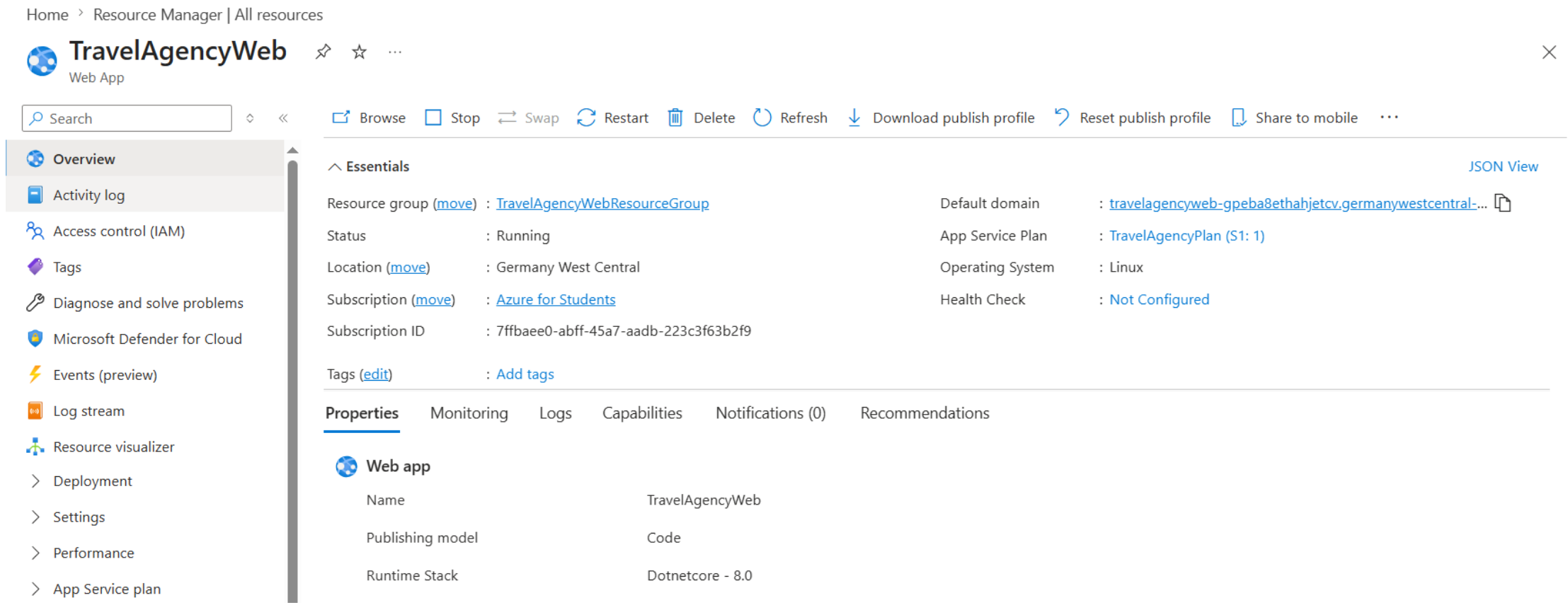
- Web-based control panel for all Azure services: https://portal.azure.com



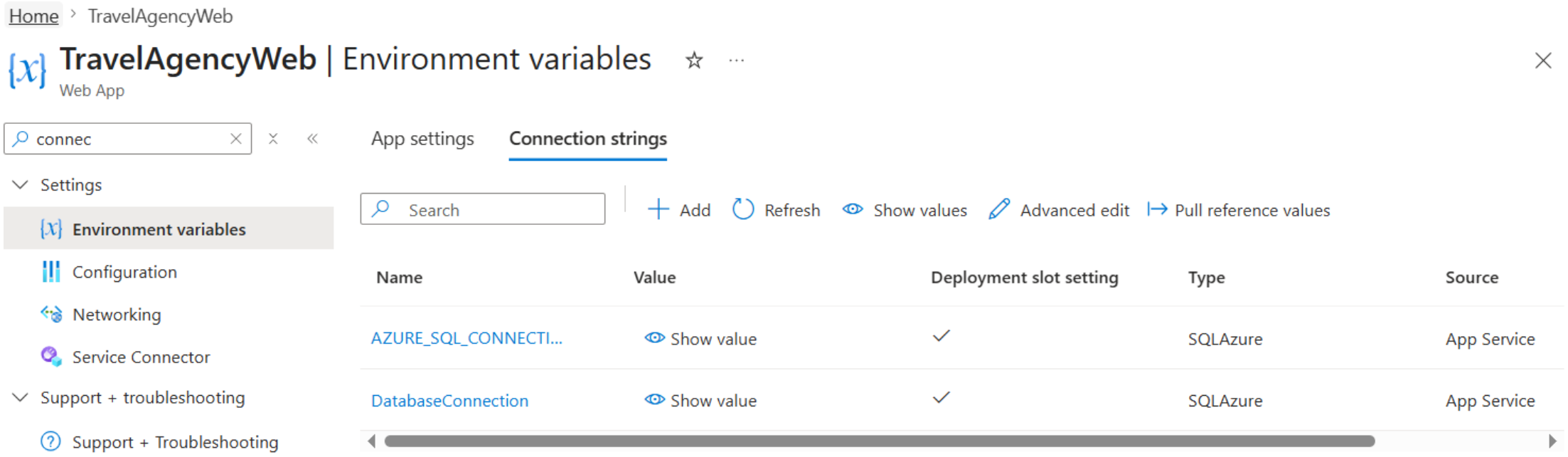
THE MICROSOFT AZURE PORTAL

- From the portal you can create, configure, monitor and delete resources
  - Resources are organized into *resource groups*, with access governed by role-based access control (RBAC)
  - Cost is tracked per subscription
- Equivalent operations are also available via the Azure CLI (**az**) and Azure PowerShell: easier to script and put under version control
- The portal also provides direct access to running resources – for example, SSH into a container or VM, or FTP/FTPS deployment credentials for an App Service

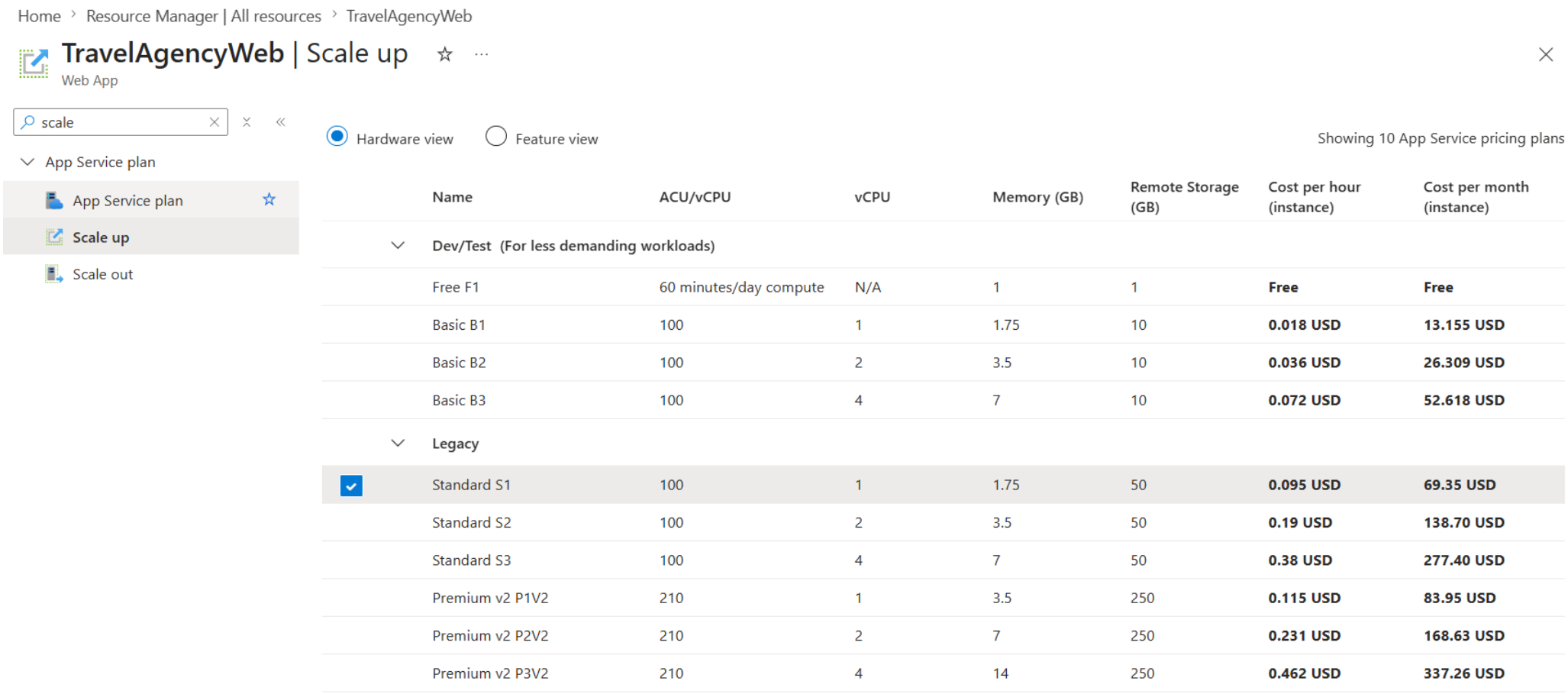
APP SERVICE OVERVIEW IN THE AZURE PORTAL



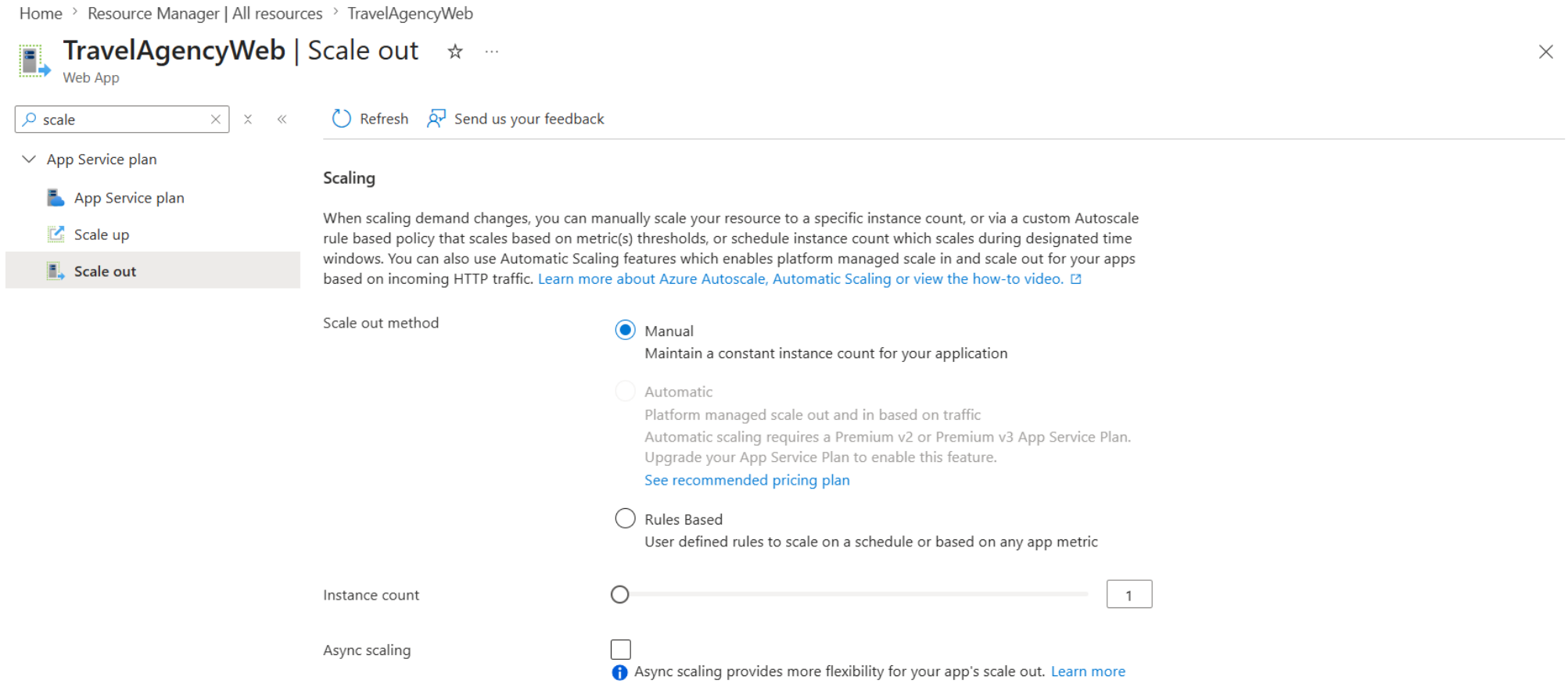
APP SERVICE CONFIGURATION IN THE AZURE PORTAL



SCALING UP THE SERVICE



SCALING OUT THE SERVICE



CACHING IN ASP.NET CORE

- Caching trades memory for time — store the result of an expensive operation so the next request can serve it instantly
- ASP.NET Core offers three layers of caching:
  - **In-memory: `IMemoryCache`**, scoped to one process
  - **Distributed: `IDistributedCache`**, shared across instances (Redis, NCache, SQL Server)
  - **HTTP-level** — Response & Output Caching middleware
- Cache invalidation is the hard part — always think about *when* entries become stale

IN-MEMORY AND DISTRIBUTED CACHING

- **In-memory cache:** register **`AddMemoryCache()`**, inject **`IMemoryCache`**:

```
var products = await _cache.GetOrCreateAsync("products",
  async e => {
    e.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5);
    return await _db.Products.ToListAsync();
  });
```

- **Distributed cache:** same API via **`IDistributedCache`**, backed by e.g. Redis: **`AddStackExchangeRedisCache(...)`**
- *Tip:* Use **absolute** expiration for time-bound data, **sliding** for session-style data; always have a fallback when the cache is cold

CACHING WITH REDIS

Using Redis as a backplane

- Redis can synchronize messages between servers
- Acts as a message bus for SignalR across a web farm
- Install Redis, then can be easily used with the **`StackExchange.Redis`** NuGet package, which implements a Redis client
- Configuration:

```
builder.Services.AddStackExchangeRedisCache(options => {
  options.Configuration = "localhost:6379",
  options.InstanceName = "MyApp";
});
```

- Inject an **`IDistributedCache`** interface where caching is required

AZURE SIGNALR SERVICE

Azure SignalR for provides auto-scaling for SignalR applications

- Fully managed real-time messaging service.
- Handles connection management, scaling, and message delivery.
- Ideal for applications with unpredictable or high traffic.
- Requires hosting the application in the Azure cloud:
  - Create an Azure SignalR resource on the Azure portal or with the Azure CLI (see the **`az signalr create`** command).
  - Save the generated URI to *Azure:SignalR:ConnectionString* environment variable.
  - Enable Azure SignalR Services in your app: **`builder.Services.AddSignalR().AddAzureSignalR();`**

HTTP RESPONSE & OUTPUT CACHING

- **Output Caching** middleware (available since .NET 7) caches the rendered response on the server:

```
builder.Services.AddOutputCache();
app.UseOutputCache();
app.MapGet("/products", ...)
   .CacheOutput(p =>
      p.Expire(TimeSpan.FromSeconds(30)).Tag("products"));
```

- **Response Caching** middleware respects **`Cache-Control`** so clients and proxies can cache too
- **ETags / Last-Modified** enable conditional GETs — **`304 Not Modified`** — and tag-based eviction invalidates entries when data changes

WHY SCALE SIGNALR?

- **Why scale a SignalR application?**
  - SignalR uses **long-lived** client connections (WebSocket, SSE, or long polling)
  - Each client keeps a persistent connection open for the entire user session
- **A single server has hard limits**
  - Practically a few thousand concurrent connections per node
  - Memory, CPU, threads and sockets are quickly exhausted
  - No redundancy — if the node fails, every client disconnects
- Horizontal scaling — multiple servers behind a load balancer — is required!

HORIZONTAL SCALING AND SIGNALR

Horizontally scaling an application

- Deploy multiple application server instances
- Use a load balancer to distribute incoming client connections
- Enables high availability and load distribution

Horizontally scaling a SignalR app

- SignalR hubs are in-memory by default
- Scaling out requires all nodes to share connection information
- Sticky sessions (client always goes to same server) are not reliable or scalable
  - Goes against the very purpose of load balancing
  - Doesn’t handle failover

THE BACKPLANE PATTERN

To share state across servers, a custom backplane (e.g. Redis) or *Azure SignalR Service* should be used.

Tasks of the backplane:

- Broadcast messages across all instances
- Share group membership and connection data

Benefits of Redis backplane:

- High performance, low latency
- In-memory and distributed
- Supports publish / subscribe for message broadcasting
- Simplifies state sharing and group membership tracking across nodes

CONFIGURING A REDIS BACKPLANE FOR SIGNALR

- Install **`Microsoft.AspNetCore.SignalR.StackExchangeRedis`** NuGet package and register Redis as the SignalR backplane:

```
builder.Services.AddSignalR()
                .AddStackExchangeRedis("localhost:6379");
```

- Uses **`StackExchange.Redis`** internally and implements a **backplane** for SignalR scale-out
- All instances share connection state via *Redis Pub/Sub* — clients on **any** node still get group/user messages
- In production: point at managed Redis (*Azure Cache for Redis*, *ElastiCache*) and load the connection string from configuration or from a secret store

OBSERVABILITY: LOGS, HEALTH, TELEMETRY

- **Structured logging:** built-in **`ILogger<T>`** integrates with *Serilog*, *NLog*, *App Insights* – pick what your prefer!
  - https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/
- **Health checks:** let load balancers detect a sick instance:
  - Register service: **`builder.Services.AddHealthChecks()`**
  - Create endpoint: **`app.MapHealthChecks("/health");`**
  - https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks
- **OpenTelemetry:** distributed tracing + metrics; export to *App Insights*, *Prometheus*, *Jaeger*, *Zipkin*, etc.
- **Graceful shutdown:** ASP.NET Core stops accepting new requests on **`SIGTERM`** and drains in-flight ones