Object-oriented programming - lab in .NET environment

Lecture 03

Web Service Architecture (API)

- User **sends a request** to the server via:
  - Graphical interface (desktop, web or mobile application)
  - Command-line interface

- Server **sends a response** on request using some data source (database or background service)
  - The answer becomes visible to the user through the used interface

Types of web services

- **REST** (Representational State Transfer)
  - Uses standard **HTTP** protocol
  - Allows different data formats (preferred **JSON**)
- **SOAP** (Simple Object Access Protocol)
  - Uses **XML** as a data format
  - Standard messaging protocol (worse performance and greater complexity than REST, but greater security)

REST

- Ensures interoperability on the Internet (**RESTful API**)
  - Different types of applications can communicate with each other
- Use HTTP verbs to create queries:
  - **POST** – creating a new resource (**C**reate)
  - **GET** – retrieving resources (**R**ead)
  - **PUT** or **PATCH** – update resource (**U**pdate)
  - **DELETE** – delete resource (**D**elete)
- Uses the predefined **stateless** operations
  - Each HTTP request is isolated (does not remember state)

Example of RESTful API request and response

- **Request**:
  `GET /api/users`

- **Response**:
  `200 OK`

```
{ "data": [
    {
      "id": 1,
      "email": "john@mail.com",
      "first_name": "John",
      "last_name": "Doe"
    }
]}
```

Using the RESTful API in C#

- There are several different C# libraries in charge of consuming a RESTful API, and some of the most famous are:
  - HttpWebRequest
  - WebClient
  - HttpClient
  - **RestSharp**
  - ServiceStack
  - Flurl

Mapping JSON objects to C# models (1/2)

JSON object

```
{
   "id": 1,
   "email": "john@mail.com",
   "first_name": "John",
   "last_name": "Doe"
}
```

C# model

```
class User
{
   public int Id { get; set; }
   public string Email { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
}
```

Mapping JSON objects to C# models (2/2)

- There are several different C# libraries responsible for mapping JSON objects to C# models, and some of the most famous are:
  - **Newtonsoft.Json**
  - AutoMapper

Asynchronous data retrieval

Synchronous operation

- Each task must be completed before the next one can begin
- The task is performed on one thread

Asynchronous operation

- The next task can start during the execution of the current task
- The task can be executed on several threads simultaneously

Advantages of asynchronous operation

- **Multiple tasks** can be executed **simultaneously** which generally results **better performance**
  - Performance is definitely better if the tasks are executed on a computer that has more processor threads and/or cores, or more processors
  - If we have one processor thread, it is necessary to do context switching which slows down the overall performance
- In the case of applications with a graphical interface (eg Windows Forms), asynchronous operations allow **responsiveness of the application**
  - Possible to handle an event (eg Click) during data retrieval

Asynchronous work in C# (in general)

- The concept is based on using a class **Task**
  - It enables the abstraction of writing asynchronous code
  - A task can be a wrapper around any data type

- Using keywords **async** and **await**
  - **async** defines an asynchronous method (executed in a separate thread)
  - **await** defines an operator that waits for the asynchronous method to be executed, and then retrieves the data that we "wrapped" in a Task
    - `await` runs an asynchronous method so that it does not block the thread from which it is called
    - If we don't use `await`, the execution of the program continues after the asynchronous method is called (no waiting for the result)

Asynchronous work in C# (specifically for applications with a graphical interface)

- Use of control **BackgroundWorker**
  - Events defined on the control:
    - **DoWork** (background thread)
      - Defines a task that runs on a background thread
      - It starts with a method call **RunWorkerAsync** in an instance of BackgroundWorker
    - **ProgressChanged** (main thread)
      - Defines a change in a task running on a background thread
      - It starts with a method call **ReportProgress** in an instance of BackgroundWorker
    - **RunWorkerCompleted** (main thread)
      - Defines a completed task that was running on a background thread
      - The task can be successful, unsuccessful, and can also be canceled by setting the property **Cancel** (defined in **DoWorkEventArgs**) to the value **true**

Comparison of asynchronous operations in C#

async / await

- Easier work if you only need to do a task on a background thread
- Better performance

BackgroundWorker

- Built-in mechanism for publishing changes in an executing task
- Built-in mechanism for canceling a started task