**Session Plan: State Management**

**​Objective**

In this lecture, the students should be introduced to State management, how to use and control the HTTP Cookies and HTTP Sessions.

**Motivation**

It's important to know how does the browser work, sessions and cookies. HTTP, by itself, is stateless and that's why we use Cookies.

**Content**

### HTTP Cookies (~25 min)

* **What are Cookies?**
  + A small file of plain text with no **executable code**
    - Sent by the server to the client's browser
    - **Stored** by the browser on the **client's device** (computer, tablet, etc.)
    - Hold small piece of data for a **particular client** and a web site
* **What Are Cookies Used for?**
  + Session management
    - Logins, shopping carts, game scores or anything else the server **should remember**
  + Personalization
    - User preferences, themes and other custom settings
  + Tracking
    - Recording and analyzing user behavior
* **Session Management**
  + The HTTP object is **stateless** 
    - It **doesn't store** information about the requests
* **Stateless HTTP – the Problem**
  + The server **doesn't know** if two requests come from the same **client**
  + State management problems
    - **Navigation** through pages requires **authentication** each time
    - **Information** about the pages is lost between the **requests**
    - Harder **personalization** of functionality of pages
* **Stateless HTTP – the Cookie Solution**
  + A reliable mechanism for websites to remember **stateful information**
    - To know whether the **user** is **logged** in or **not**
    - To know which account the **user** **is logged** in with
    - To record the user's **browsing activity**
    - To **remember** pieces of information **previously** entered into form fields (usernames, passwords, etc.)
* **How are Cookies Used?**
  + The response holds the cookies to be saved within the **Set-Cookie** header
  + The request holds the specific web site cookie within the **Cookie** header
* **Client-Server Cookies Exchange**
* **Cookie Structure**
  + The cookie consists of **Name**, **Value** and **Attributes** (optional)
  + The attributes are **key-value pairs** with additional information
  + Attributes are **not included** in the **requests**
  + Attributes are used by **the client** to control the **cookies**
* **Scope**
  + Defined by the attributes **Domain** and **Path**
  + **Domain** – defines the website that the cookie belongs to
  + **Path** – Indicates a **URL** path that must exist in the requested resource before sending the **Cookie** header
* **Lifetime**
  + Defined by the attributes **Expires** and **Max-Age**
  + **Expires –** defines the date the browser should delete the cookie
    - By default the cookies are deleted after the end of the session
  + **Max-Age** – interval of seconds before the cookie is deleted
* **Security**
  + Security flags do not have associated values
  + **Secure –** tells the browser to use cookies only via **secure/encrypted** connections
  + **HttpOnly** – defines that the cookie cannot be accessed via client-side scripting languages
* **What is in the Cookie?**
  + The cookie file contains a table with **key-value** pairs
* **Third Party Cookies**
  + Cookies stored by an **external party** (different **domain**)
  + Mainly used for advertising and tracking across the web

### Q&A [Sli.do] (~5 min)

### HTTP Sessions (~25 min)

* **What** **are Sessions?**
  + A way to store information about a user to be used across **multiple pages**
* **Session management**
  + The exchange mechanism be used between the user and the web application
* **Relation with Cookies**
* **Session structure**

### Q&A [Sli.do] (~5 min)

### BREAK: 15 min

### Session vs Cookies (~10 min)

* **Session vs Cookies**
  + **Session**
    - Stored on the server
    - Expires when the user closes the browser
    - It can store an unlimited amount of data
    - Depends on the cookie
    - Secure –saves data in encrypted form and cannot be accessed by anyone easy
  + Cookies
    - Stored on the user's computer as a text file
    - Expires on its expiration date
    - It can store only limited data
    - Does not depend on the session
    - Have security issues, as data is stored in a text file and it can be accessed by anyone easily

### Practice (~35 min)

1. Examine Your Cookies
2. Control Your Cookies (Mozilla)
3. Control Your Cookies (Chrome)
4. Change Cookies
5. Authentication with Cookies
6. Session

### Q&A [Sli.do] (~5 min)

### BREAK: 15 min

### Synchronous Programming (~20 min)

* **Synchronous Programming**
  + Executing program components **sequentially**
  + "Sequential programming"
  + Actions happen one after another
  + Components **wait** for previous components to finish
  + Program resources are accessible at all points
* **Synchronous Code**
  + Synchronous code is executed **step by step**
* **Synchronous Code – Long Running Operation**
* **Synchronous Programming Drawbacks**
  + If one **component is blocked**, the **entire program is blocked**
  + UI may become **unresponsive**
  + No utilization of multi-core systems
  + CPU-demanding tasks **delay execution** of all other tasks
  + **Accessing resources** blocks entire program
  + Especially problematic with web resources
* **Asynchronous Programming**
  + Program components can execute in **parallel**
    - Some actions run alongside other actions
    - Each action can happen in a **separate** thread
  + **Independent** components don't wait for each other
  + Program resources shared between threads
    - If one thread uses a resources, others shouldn't use it
* **Asynchronous Programming – Benefits**
  + If a component is blocked, other **components still run**
    - UI runs separately and always remains responsive
  + Utilization of multi-core systems
    - Each core executes **one or more** threads
  + CPU-demanding tasks run on "**background**" threads
  + Resource access runs on "**background**" threads
* **Asynchronous Programming – Drawbacks**
  + Hard to know which code parts are running at a specific time
  + Harder than usual to **debug**
  + Have to **protect resources**
    - One thread uses a resource
    - Other threads must wait for the resource
  + **Hard to synchronize** resource access
    - **Deadlocks** can occur
* **Asynchronous Code**
  + Asynchronous programming allows the execution of code **simultaneously**

### Q&A [Sli.do] (~5 min)

### Threads (~20 min)

* **Instruction Execution**
  + Each program's code is translated to CPU instructions
* **Multi-Tasking**
  + A computer can run **many processes** (applications) at once
  + But each CPU core can only execute one instruction at a time
  + **Parellelism** is achieved by the operating system's **scheduler**
    - Grants each **thread** a small interval of time to run
* **Threads**
  + A **thread** is a fundamental unit of code execution
  + Commonly, processes (programs) use more than one thread
  + In .NET, there is always more than one thread (e.g. GC)
  + Each thread has a **memory area** associated with it known as a **Call Stack**
  + Stores **local variables**
  + Stores the **currently invoked methods** in order of invocation
* **Threads in C#**
  + Threads in C# can be created using the **System.Thread** class
  + Constructor accepts a **method** (delegate) to execute on a separate thread
* **System.Thread**
  + **Start()** – schedules the thread for execution
  + **Join()** – waits for the thread to finish its work (blocks the calling thread)
* **Thread – Example**
* **Thread Stack**
  + Each thread has its own **stack**
  + The start (bottom) of the stack is the method from which the thread began execution
  + Each method (frame) stores local variables
* **Thread Race Conditions**
  + A **race condition** occurs when two or more threads access shared data and they try to change it at the same time
* **Thread Safety**
  + A thread-safe resource can be safely accessed by multiple threads
  + **lock** keyword grants access to only one thread at a time
    - Avoids race conditions
    - Blocks any other threads until the lock is released
* **Exception Handling**
  + Exceptions cannot be handled outside a thread
* **Exception Handling – the Right Way**

### Q&A [Sli.do] (~5 min)

### BREAK: 10 min

### Tasks (~15 min)

* **Tasks in C#**
  + A task is a high-level representation of concurrent work
    - Runs in **parallel** with the main thread
    - May not run on a new thread (the CLR decides)
    - Offers several operations
      * Creating, running and **returning** result
      * Continuing with another task (**chaining several operations**)
      * Proper exception handling
      * Progress/state reports
* **Creating Tasks in C#**
  + Creating tasks can be done in several ways
    - Initialize a new **Task** object
    - **Task.Run()**
    - **Task.Factory.StartNew()** – enables additional task customization
* **Generic Tasks**
  + **Task<T>** is a task that will return a result sometime in the future
* **Task Exception Handling**
  + **Exceptions** that have occurred within the body of a **Task** can be captured and handled outside of it

### Q&A [Sli.do] (~5 min)

### Async and Await (~15 min)

* **Tasks with Async and Await**
  + The keywords **async** and **await** are **always** used together
  + **async** hints the compiler that the method might run in parallel
  + Does not make a method run asynchronously (**await** makes it)
  + Tells the compiler "**this method could wait for a resource or operation**"
    - If it starts waiting, return to the calling method
    - When the wait is over, go back to called method
  + **await** is used in a method which has the **async** keyword
    - Saves the context in a state machine
    - Marks waiting for a resource (a task to complete)
      * Resource should be a **Task<T>**
      * Returns **T** result from **Task<T>** when it completes
* **Async and Await – Example**

### Q&A [Sli.do] (~5 min)

**Exercise**

Because this is a part theoretical topic, there are no practical exercises.

**Evaluation & Exam**

Because this is a part theoretical topic, only the asynchronous processing part is included in the exam.