# SIS

SIS is a combination of a Web Server and a MVC Framework. Ultimately it is designed to mimic Microsoft’s IIS and ASP.NET Core. Following several Lab documents you will build all components of the SIS.

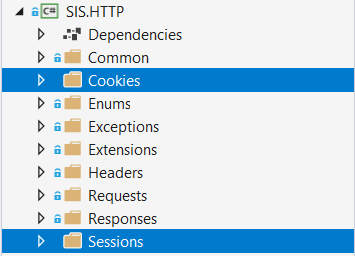
# SIS: Handmade HTTP Server

Following to the end this document will help you to create your own very simple HTTP Server. Later in the course we will extend it by adding sessions, cookies etc. We will eventually build a MVC Framework, with which we can build MVC Web Application which will be hosted on the Handmade HTTP Server.

# State management

In this lab you will configure the Server to be stateful. This means that we will add a few classes for Cookies and Sessions, in order to maintain states about our clients.

Start by **adding** the following **2 namespaces** to the SIS.HTTP project.



And now let’s get to the main thing...

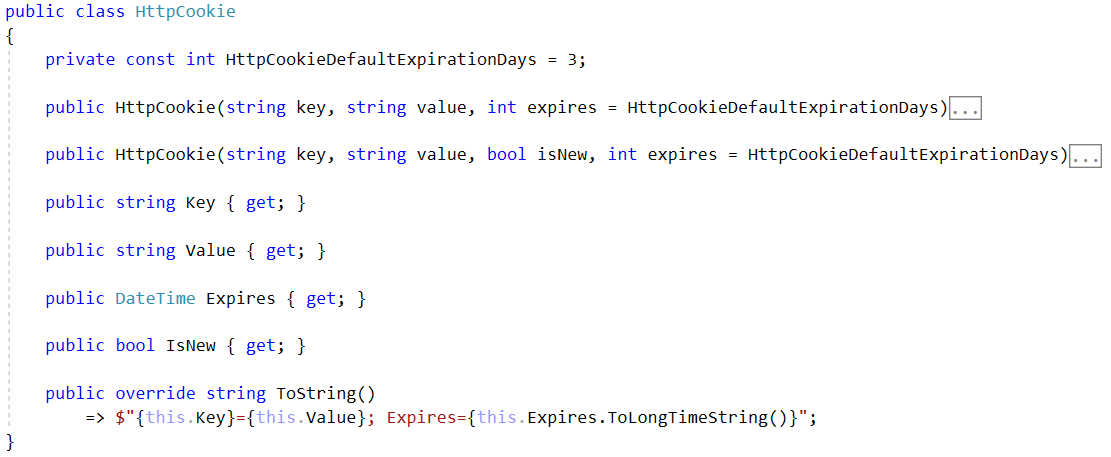
## Cookies

### HttpCookie Class

The first thing we need to do is add the functionality for the **Cookies**, they will be the most **primitive element** to our **State Management**.

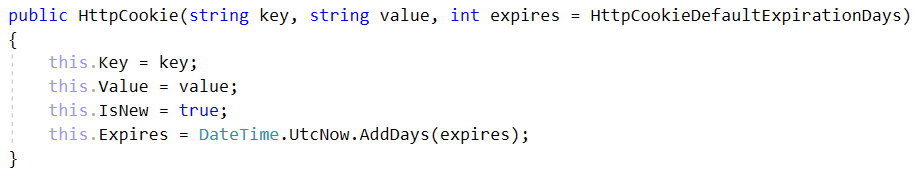
Create a class, called HttpCookie, in the Cookies namespace. The class should have the following properties:

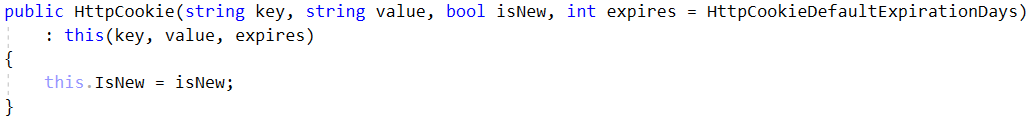
* Key – a **string**, representing the **key** (or the **name**) of the **Cookie**.
* Value – a **string**, representing the **value** (or the **content**) of the **Cookie**.
* Expires – a **DateTime**, representing the **expiration time** of the **Cookie**.
  + This **property** will be initialized with an **integer**, which will represent **DAYS**, from the **current moment**.
* IsNew – a **boolean**, which will be used to **define** if the **Cookie** is **freshly created**. This way we will know if the **Server** has **created** the **Cookie** (for example when the **Client logs in**) or the **Cookie** comes from the **Client**.



The ToString() method formats the **Cookie** **parameters** in **Web-ready format**.

There are **2 constructors** which control the way the IsNew property is **initialized**, due to its behaviour:

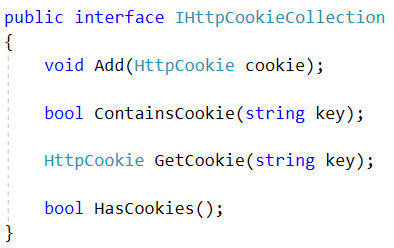




And with this we have completed our HttpCookie class. Now it’s time to create a **Repository-like** class for it.

### HttpCookieCollection Class

Create an interface, called IHttpCookieCollection in the Cookies namespace. It should look like this.

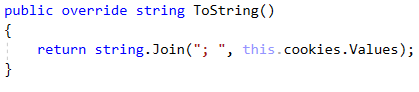


The classes that implement the interface and its methods should have the following functionality:

* Add() – **adds** the given Cookie to a **collection** of HttpCookies.
* ContainsCookie() – returns a **boolean** result, on whether the **given key** is **contained** in the HttpCookie collection.
* GetCookie() – **extracts**, form the HttpCookie collection, the Cookie with the **given key**, and **returns it**.
* HasCookies() – returns a **boolean** result, on **whether** there are **ANY** **cookies** in the HttpCookie collection.

The class is yours to implement. : )

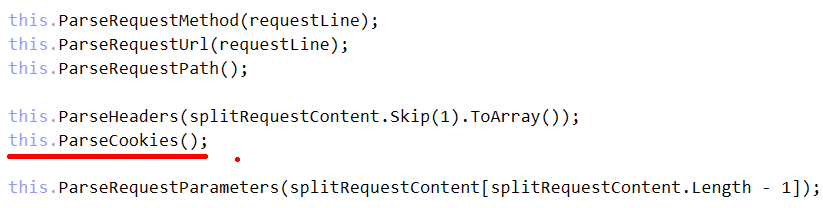
The HttpCookieCollection class should also have a ToString() override method, that should format the Cookies in Web-Ready format:



Now let’s add the Cookies to our main communication classes – The HttpRequest & HttpResponse.

### HttpRequest

Add an IHttpCookieCollection property to the HttpRequest class. Intiialize it from the constructor. Write a method ParseCookies(), which checks the HttpHeadersCollection for a Header with name “Cookie” and if there is, extracts its **string value**, **formats** it, **parses** it and **adds** it to the HttpCookieCollection.



### HttpResponse

Add an IHttpCookieCollection property to the HttpResponse class. Intiialize it from the constructor. Write a public method AddCookie(), which **adds** the **given** Cookie to the HttpCookieCollection.

Reformat the ToString() method so that it includes the Cookies, if there are any, with a “Set-Cookie” **name**, and **values** – separated by a **semi colon** and a **space**.



And with this we are done with the Cookies’ implementation in our HTTP Server. This will be enough for now. As the time passes by, we will obviously refactor them, extend them, optimize them and manipulate them in many ways, but let’s be patient.

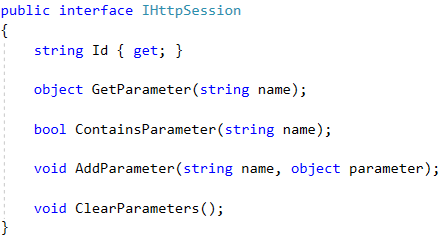
**NOTE**: Each of the **public methods** and **properties** you’ve **added** should **be** **also added** to the **interfaces** of the **corresponding classes**.

## Sessions

The next big thing we need to implement is the Sessions. They are **Server-side State Management** mechanism, and are the most important element of the Stateful functionality.

### IHttpSession

Let’s start with the interface. Create an interface, called IHttpSession, in the Sessions namespace. The HttpSession should have a **collection** of **parameters** and a Id which is a **string**. The interface should look like this:



As you can see there are **several methods** for **accessing** the **collection** of **parameters**, as it will be **private**. This is everything we need as public behaviour.

The classes that implement the **interface** and its **methods** should have the following functionality:

* Id – just a property with a getter.
* GetParameter() – **extracts**, form the **parameter collection**, the **parameter** with the **given name**, and **returns it**.
* ContainsParameter() – returns a **boolean** result, on whether a **parameter** with **given name** is **contained** in the collection.
* AddParameter() – **adds** the given parameter with the **given name** to a **key-value-pair** **collection** of **parameters**.
* ClearParameters() – **clears** the **collection**, emptying it.

The class is yours to implement. : )

**Note**: The **constructor** of the **class** should **initialize** the **collection** and the Id. Here’s a hint on how it should look:



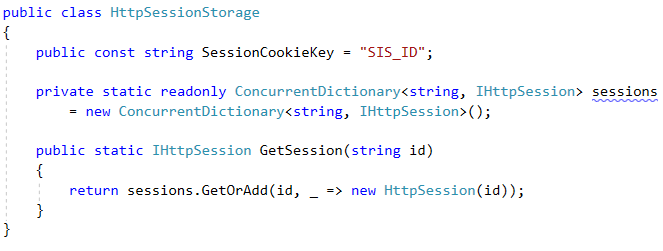
The HttpSessions are implemented, but there is something missing. The **Server** needs to store the Sessions somewhere.

### HttpSessionStorage

Create a class named HttpSessionStorage, in the Sessions namespace. We will use this **class** to **store** our **sessions**, in a **Dictionary-like** collection. But our **Server** will work with **many Clients parallelly**, which means that the collection must be **async-friendly**, or **thread-safe**.

Well, there is a collection that just does the trick.

We would also need a **Session Key**, something with which our Session will be sent as a Cookie to the Client.   
Let’s call it “SIS\_ID”. This will be the **SIS**’s **Session Key**.



The GetSession() method **retrieves** a Session from the **Session Storage collection** if it **exists**, or **adds** it and **then retrieves** it, if it **does NOT exist**.

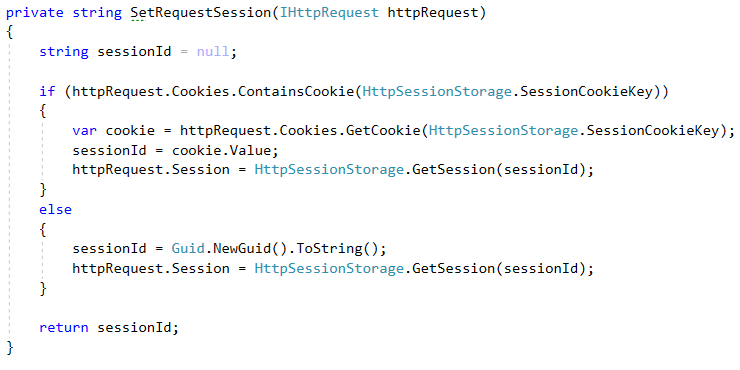
And with this we are ready with our HttpSessionStorage. Now it’s time to include all we’ve created so far in the main business logic of our **Server**.

## Server

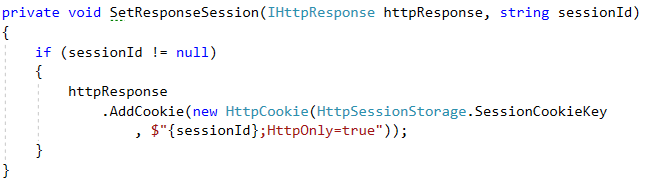
We have to add the Sessions functionality to our ConnectionHandler, in order for it to be linked with the **Client**.

The following things must be included in our logic:

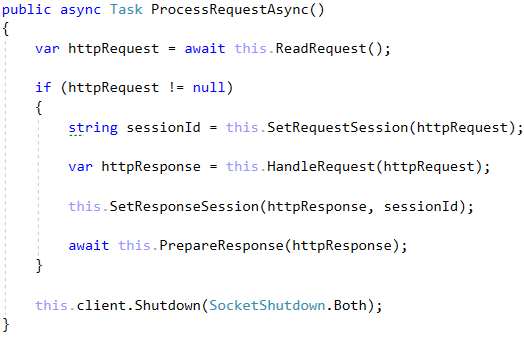
* **Initialize** the Request Session – **Check** if the Request contains a Cookie with the **SIS Session Key**.
  + If **there is**:
    - **Extract** the **value** of the Cookie (which will be the **id** of the Session).
    - **Use** the **id** to **extract** the Session from the Session storage.
    - **Set** it to the Request’s Session property.
  + If **there isn’t**:
    - **Generate** a **new GuID**, create a **new Session** with it.
    - **Add** the new Session to the Session storage.
    - **Set** it to the Request’s Session property.
  + This should be achieved by **adding** the **following method** to the ConnectionHandler class:



* **Initialize** the Response Session – **Add** a Cookie with the **SIS Session Key** as **Cookie key**, and the Request’s session id, as **Cookie value**
  + This should be achieved by **adding** the **following method** to the ConnectionHandler class:



The **invocation** of these methods should be **performed** while **processing** the Request. Modify the ProcessRequest() method like this:



And this should do for now. We have **implemented** a very simple **State Management** mechanism in our **SIS**’s **HTTP Server**.

As the Server is implement, currently, it generates a Session for every connected client. That Session however is **manipulatable**, and we can **add parameters** to it. **Parameters** such as username for example. Such parameters may be used to **indicate** if the **Client** is **logged in**, or what are his **permissions** ot our application.

## Test it out

