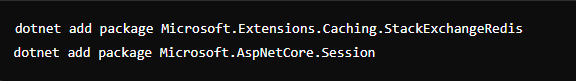
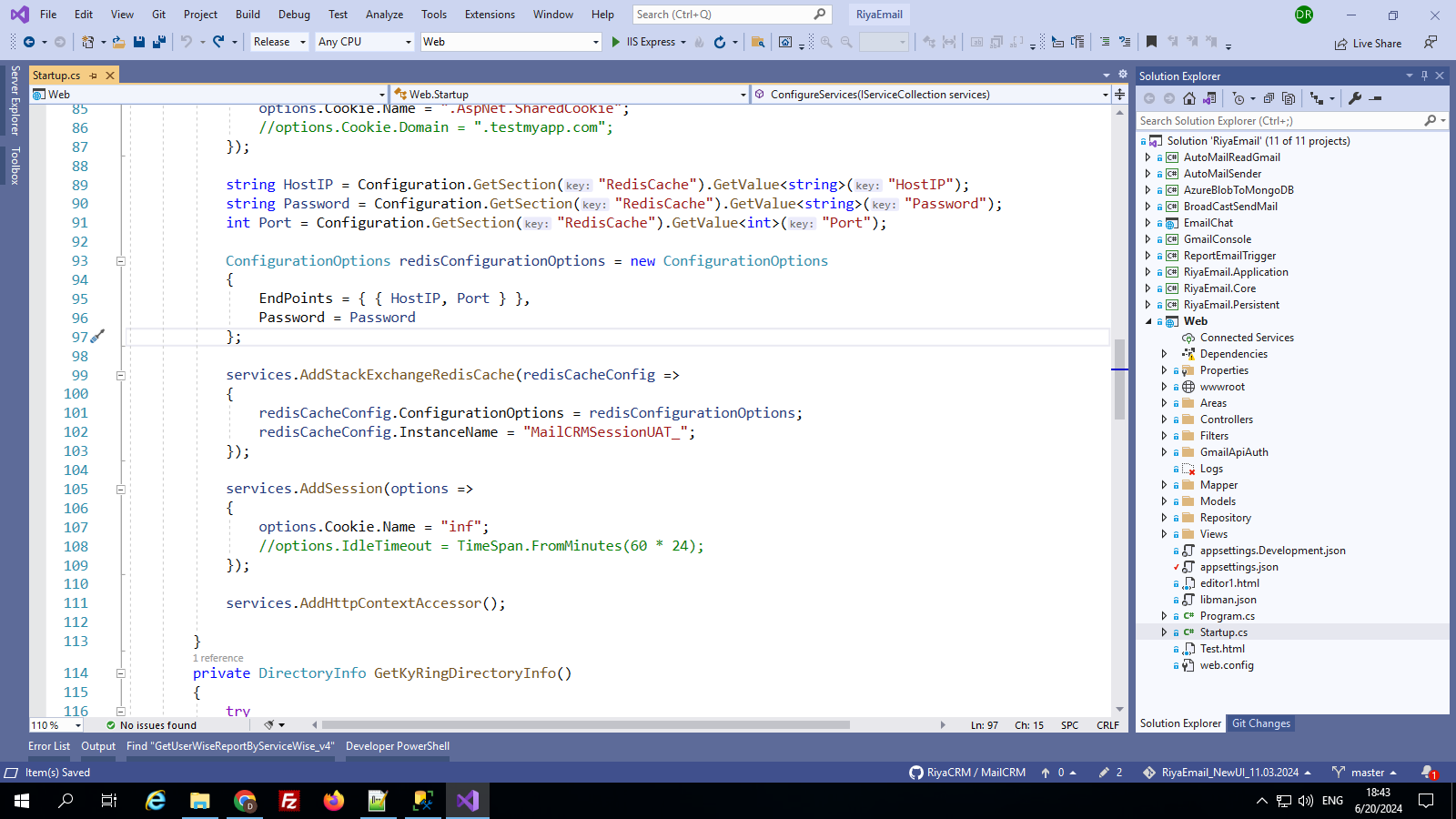
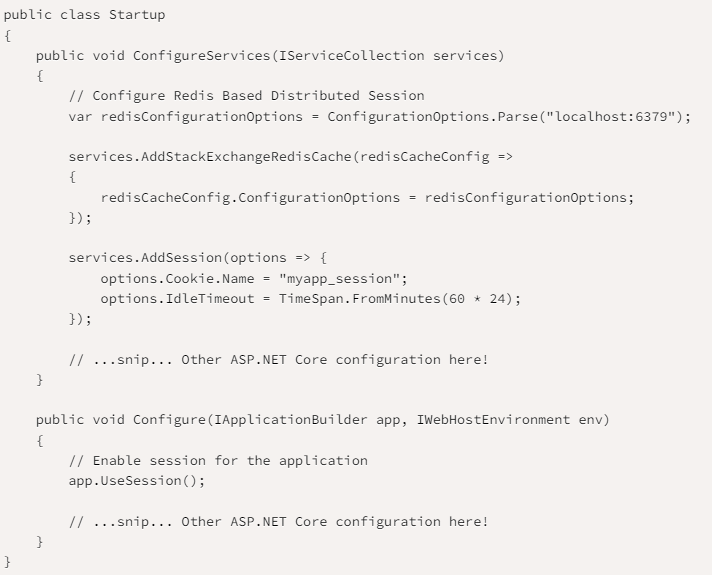
1. Business Concept
   * We need distributed session state because load balancing with sticky sessions is whack. Spoiler: We DON’T roll it ourselves. ASP.NET session storage is useful for storing state across page views. In single server situations it’s simple to set up because ASP.NET supports in-memory session out of the box. In-memory sessions stop working as soon as there is more than one server. Most production environments have more than one server so the session issue needs to be dealt with.
   * There are two options for sessions in a web farm. First, a load balancer can be used to lock each user on a specific box (so-called “sticky sessions” or “session affinity”). This lets us continue to use in-memory session. The second is switching from in-memory to distributed session storage.
   * We will use a distributed session using Redis because we have existing Redis infrastructure and we don’t want a complex load balancer added to the mix.
2. Language Async
   * load balancing: the process of distributing traffic among multiple servers to improve a service or application's performance and reliability
3. Core Business Logic
   * **Session Management:**
     1. **Session Storage:** Instead of storing session data in the memory of a single server, Redis stores session data centrally, allowing any server in a web farm to access it. This supports scalability and load balancing.
     2. **Data Persistence:** Redis can persist session data to disk, providing durability beyond in-memory storage.
   * **Integration with ASP.NET Core:**
     1. **Configuration:** Configure ASP.NET Core to use Redis as the session store.
     2. **Middleware Setup:** Set up middleware to use Redis for session management.
   * **Scalability and Performance:**
     1. **High Availability:** Redis supports clustering and replication, ensuring high availability and failover capabilities.
     2. **Low Latency:** Redis is known for its high-speed read and write operations, ensuring low latency in session retrieval and updates.
   * **Security and Session Expiration:**
     1. **Data Security:** Redis provides mechanisms for securing data, such as encryption and access control.
     2. **Expiration Policies:** Set session expiration policies to manage the lifecycle of session data efficiently.
4. Core technical aspects
   * Install Required Packages:
     1. Add the necessary NuGet packages to your ASP.NET Core project



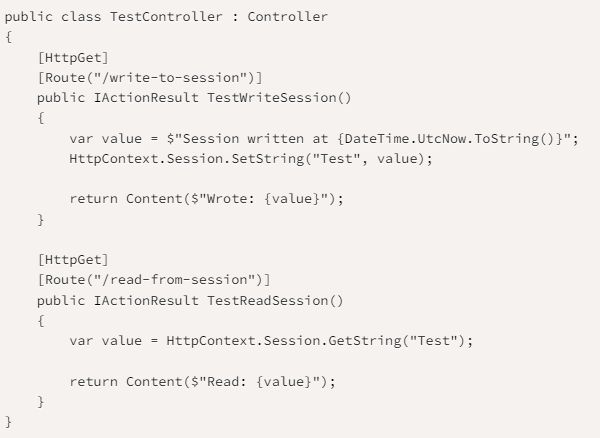
* Configure Services in **Startup.cs** :
  1. Using Host Ip, Port and Password



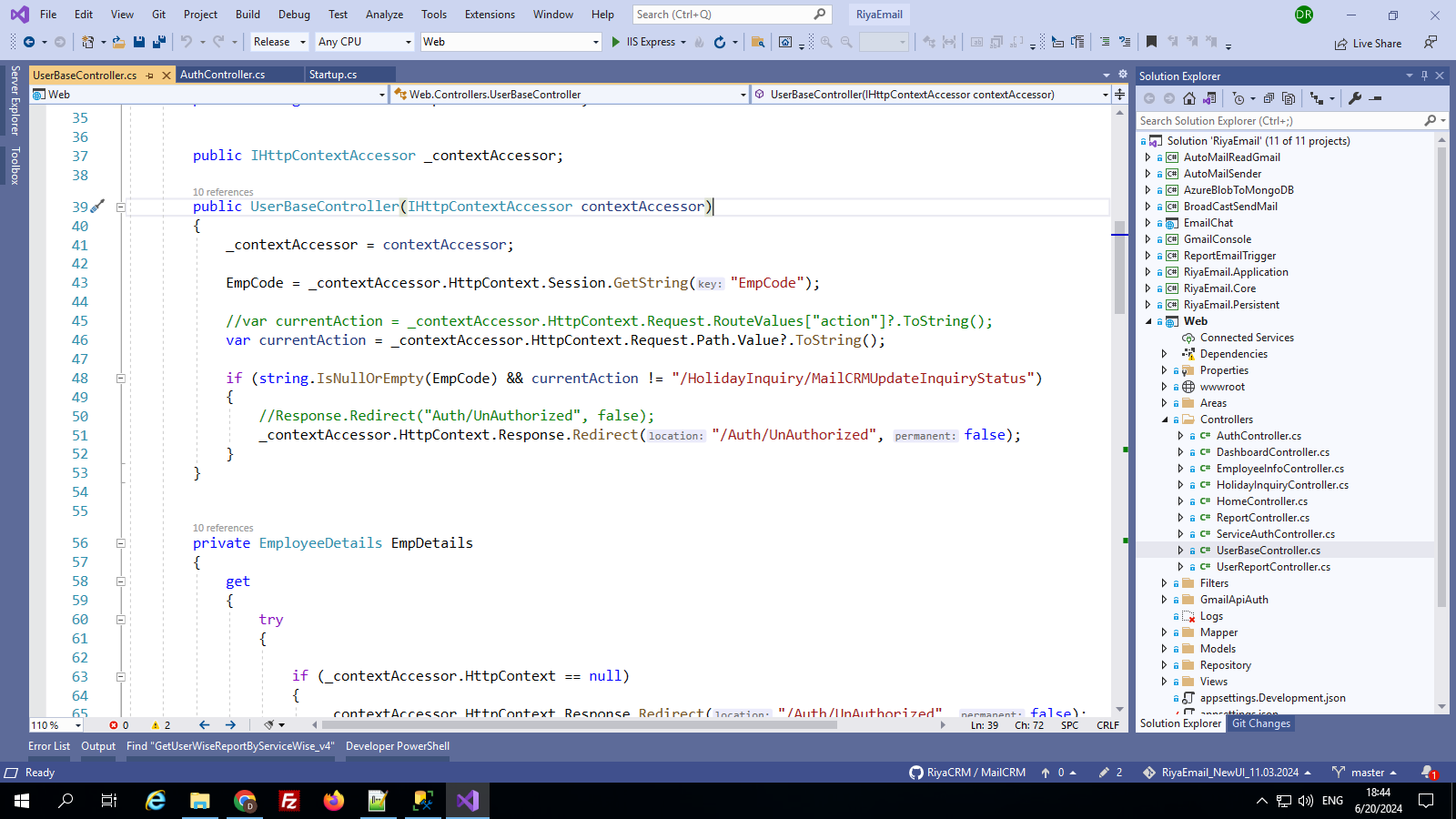
* 1. Using localhost



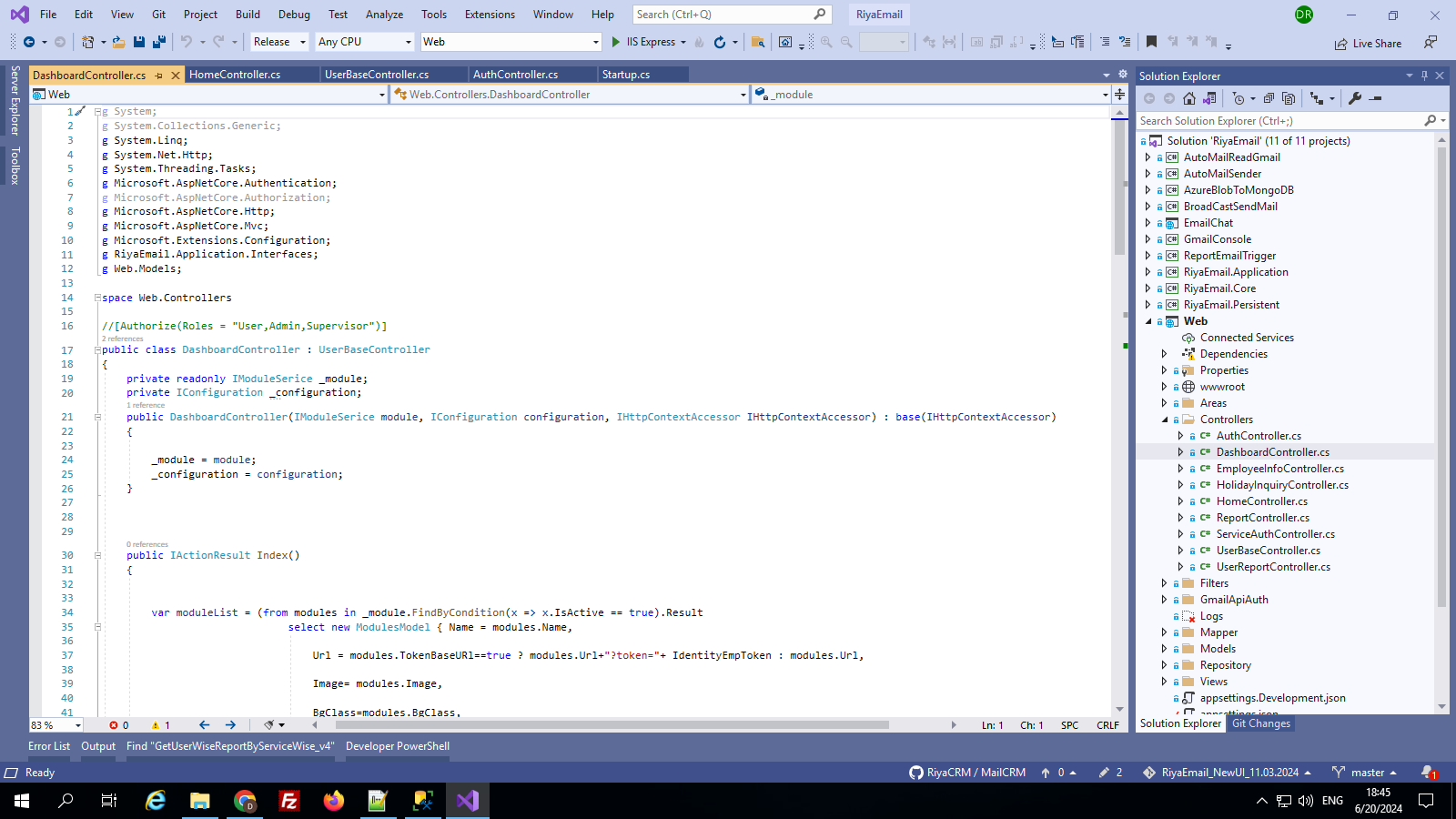
* Read and Write to the User’s Session:



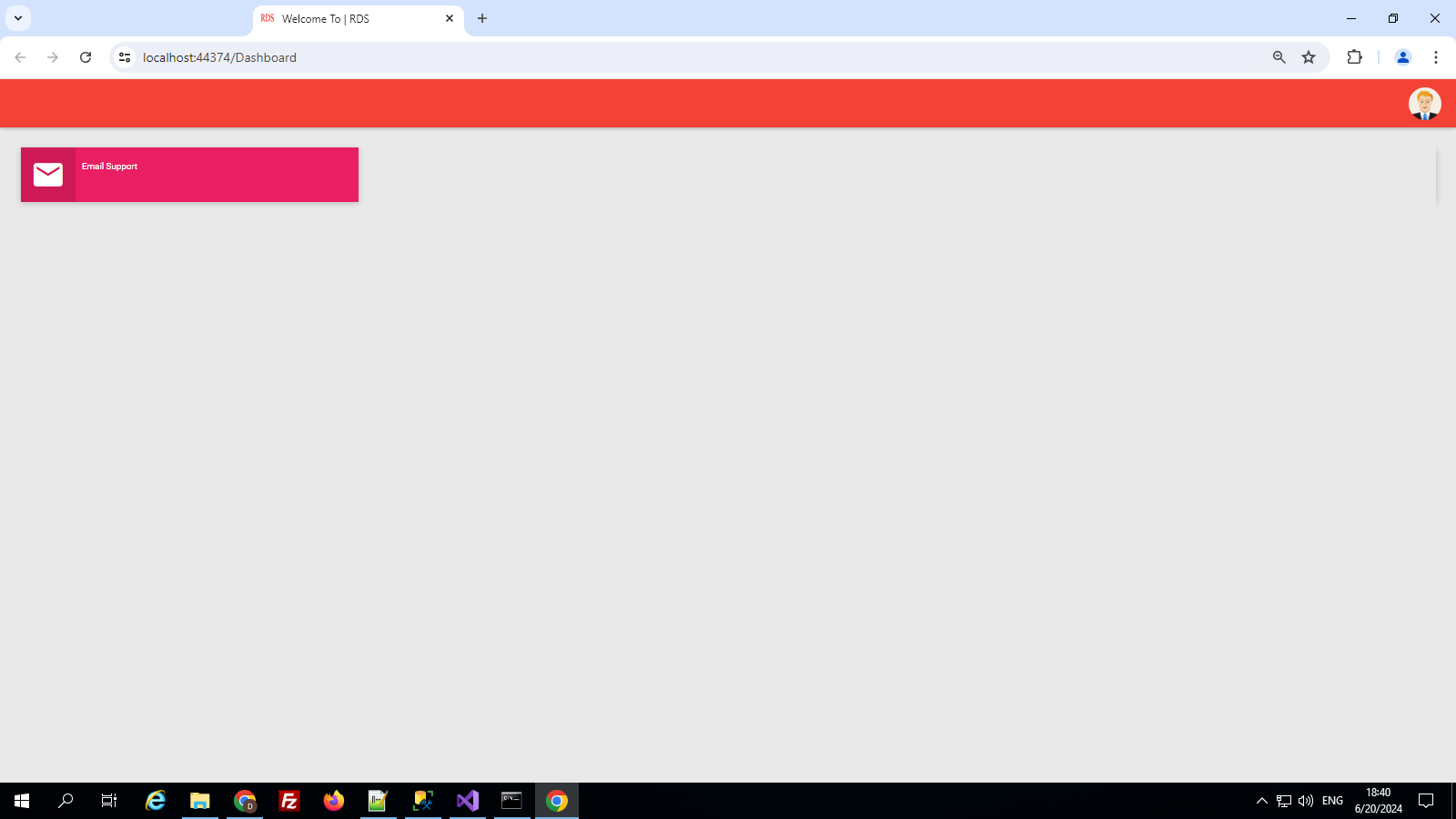
* Use For Session Authentication:
  1. Set session authentication code in inherited controller like UserBase controller



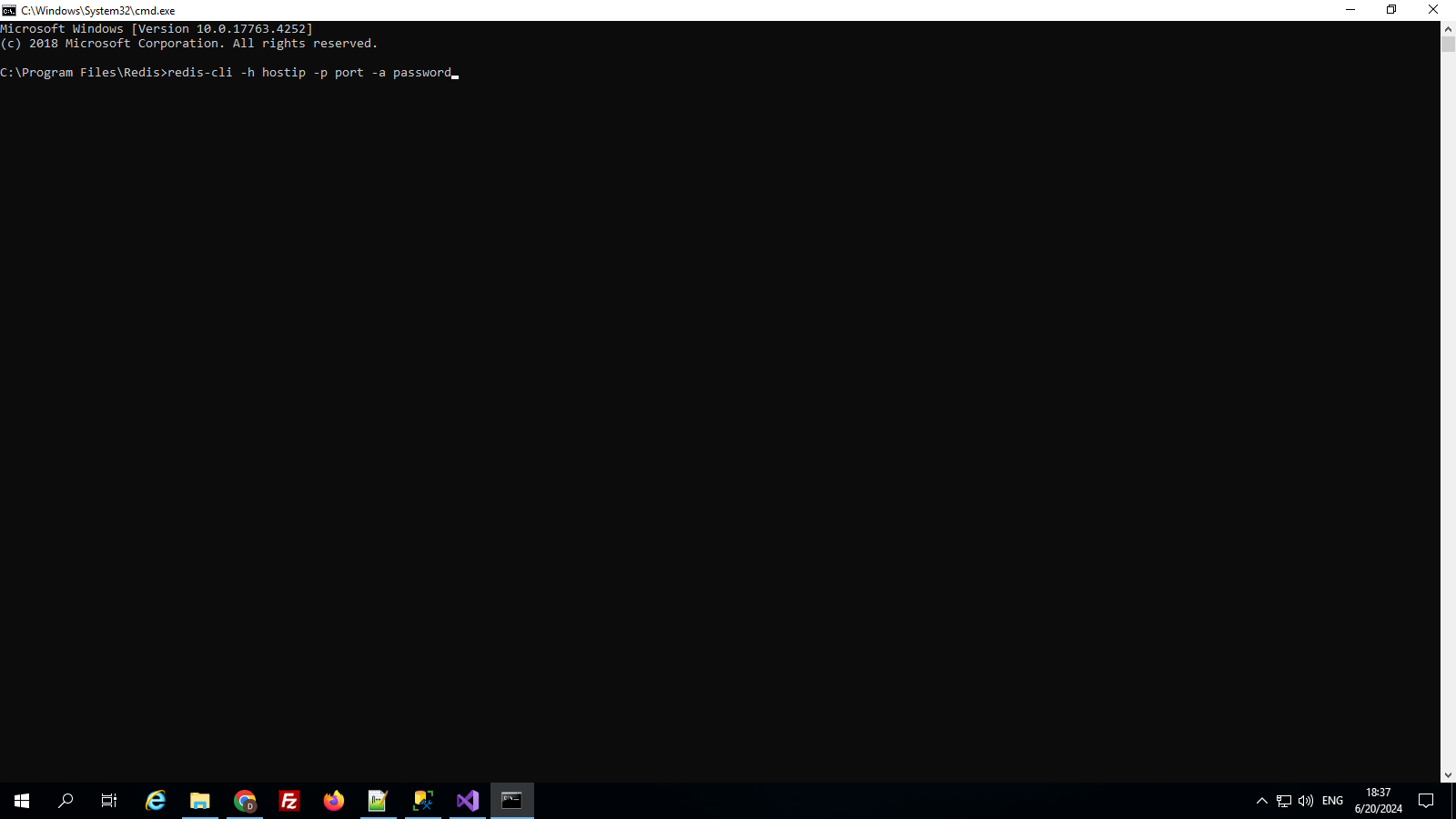
* 1. Now UserBase controller inherits with other controller.



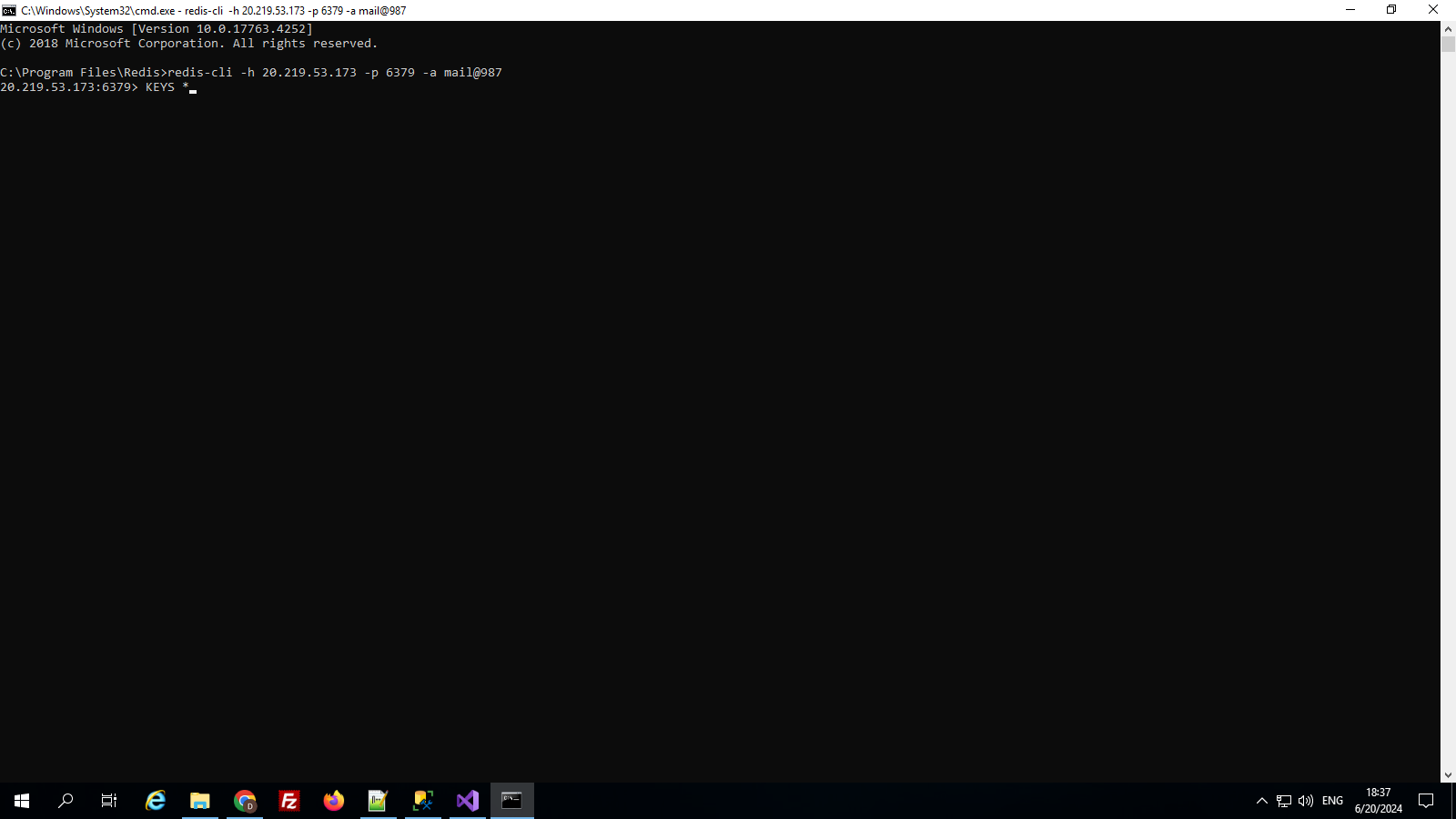
* Run the project after the code is set.



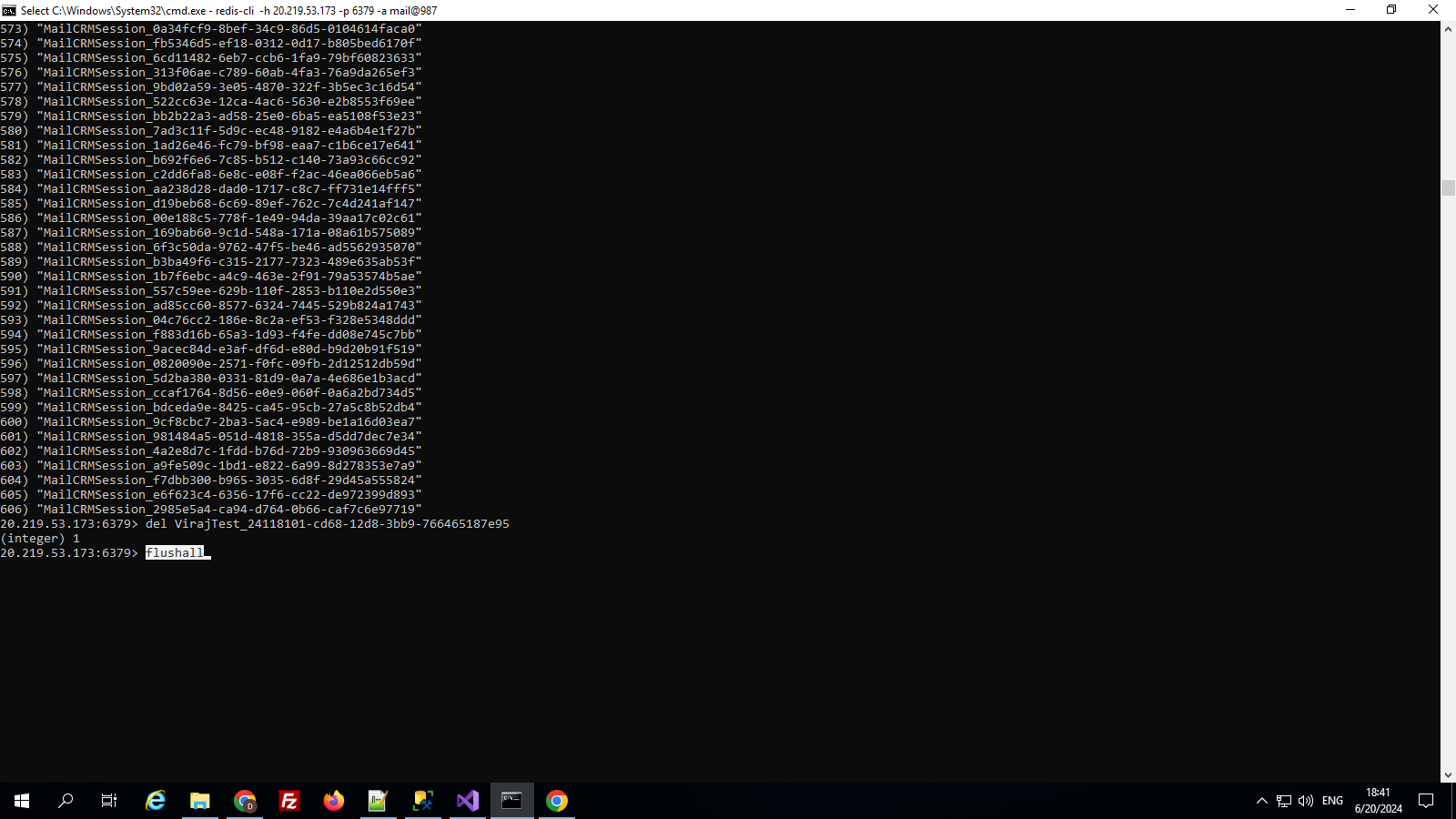
* Then install **redis-cli exe** andopen through **cmd.** Also connect Redis server using HostIp, port, and password.



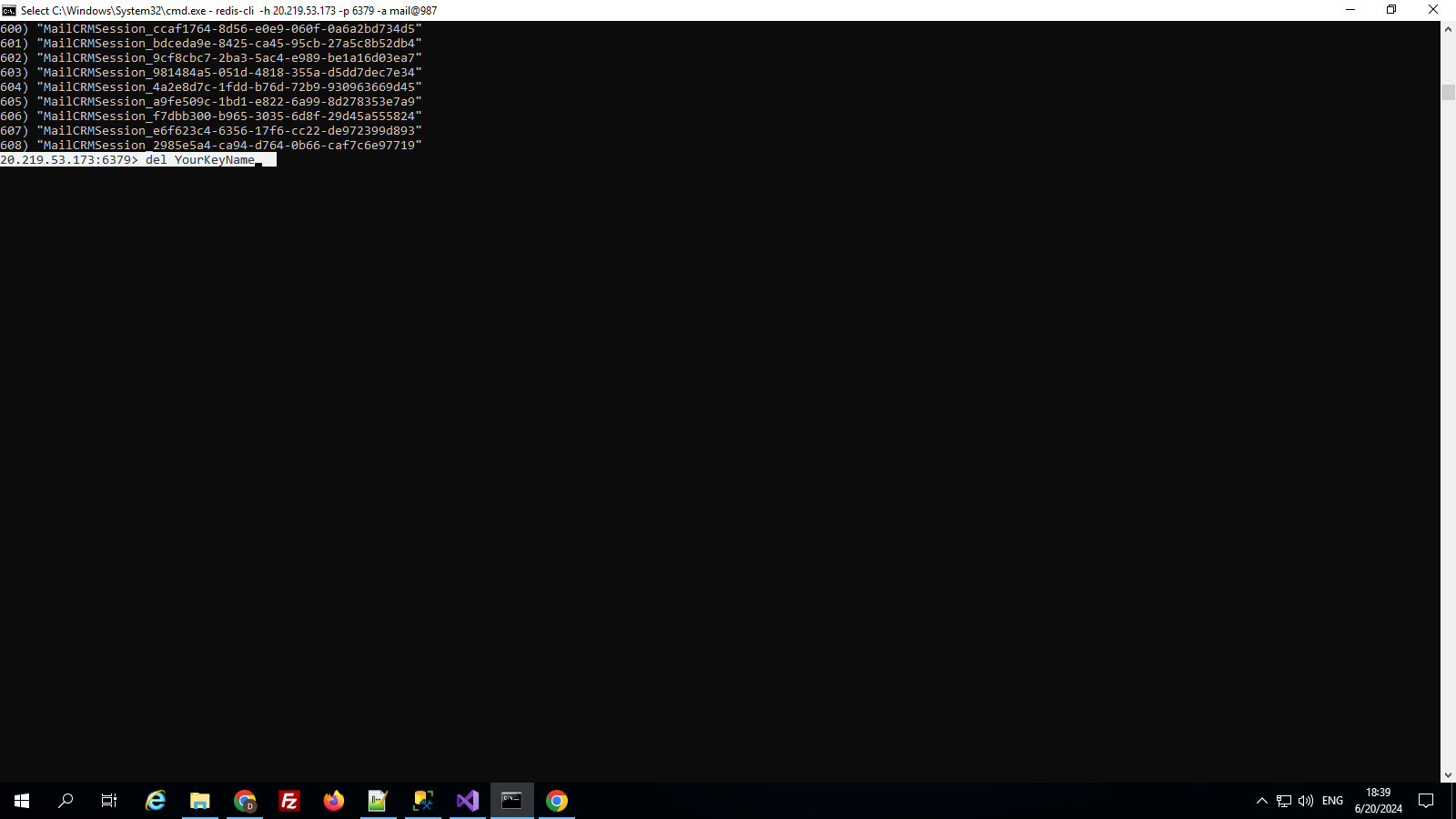
* After connecting redis-cli server, you can check generated session keys using **“KEYS \*”** command.



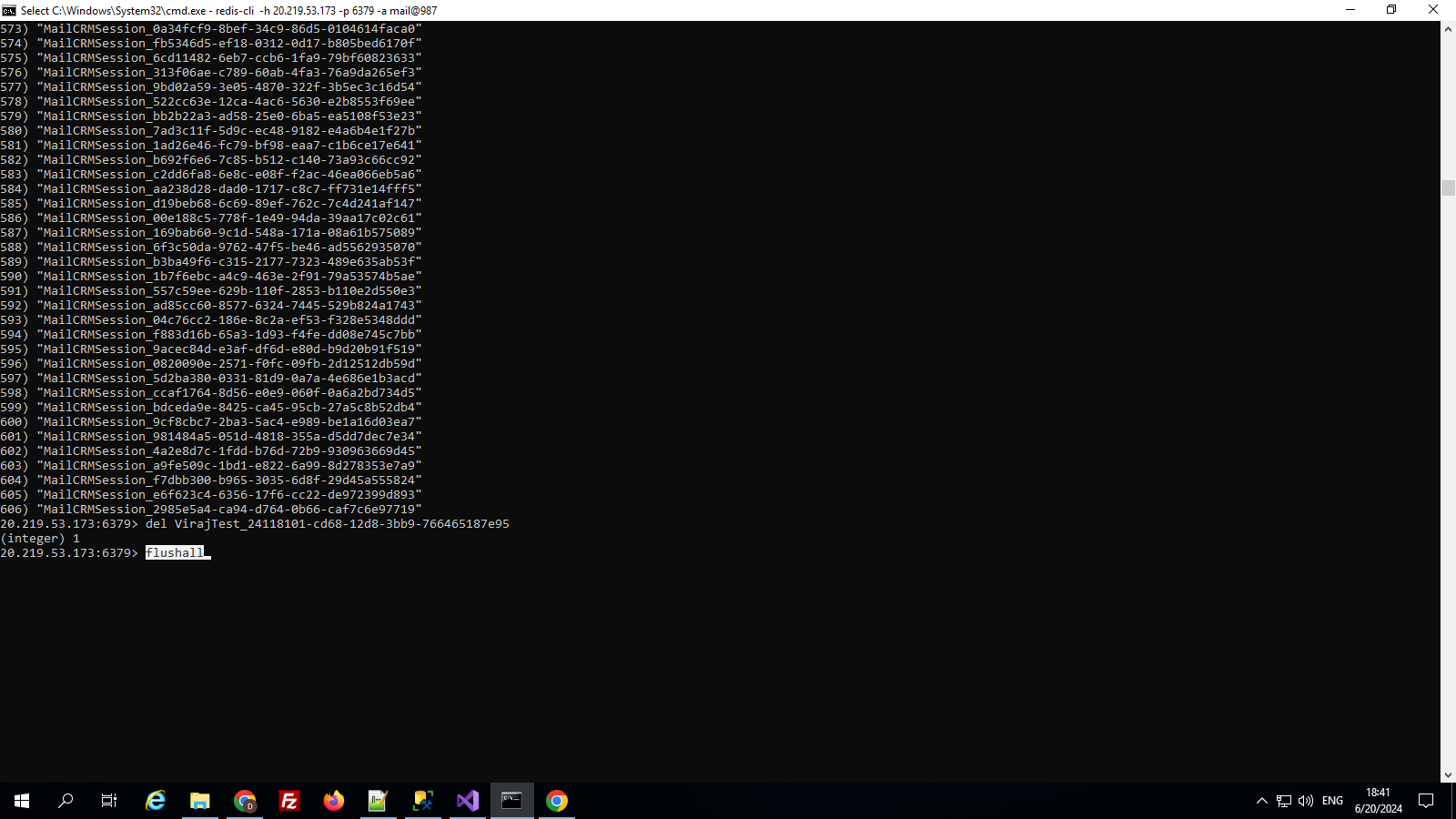
* now you can see your keys.



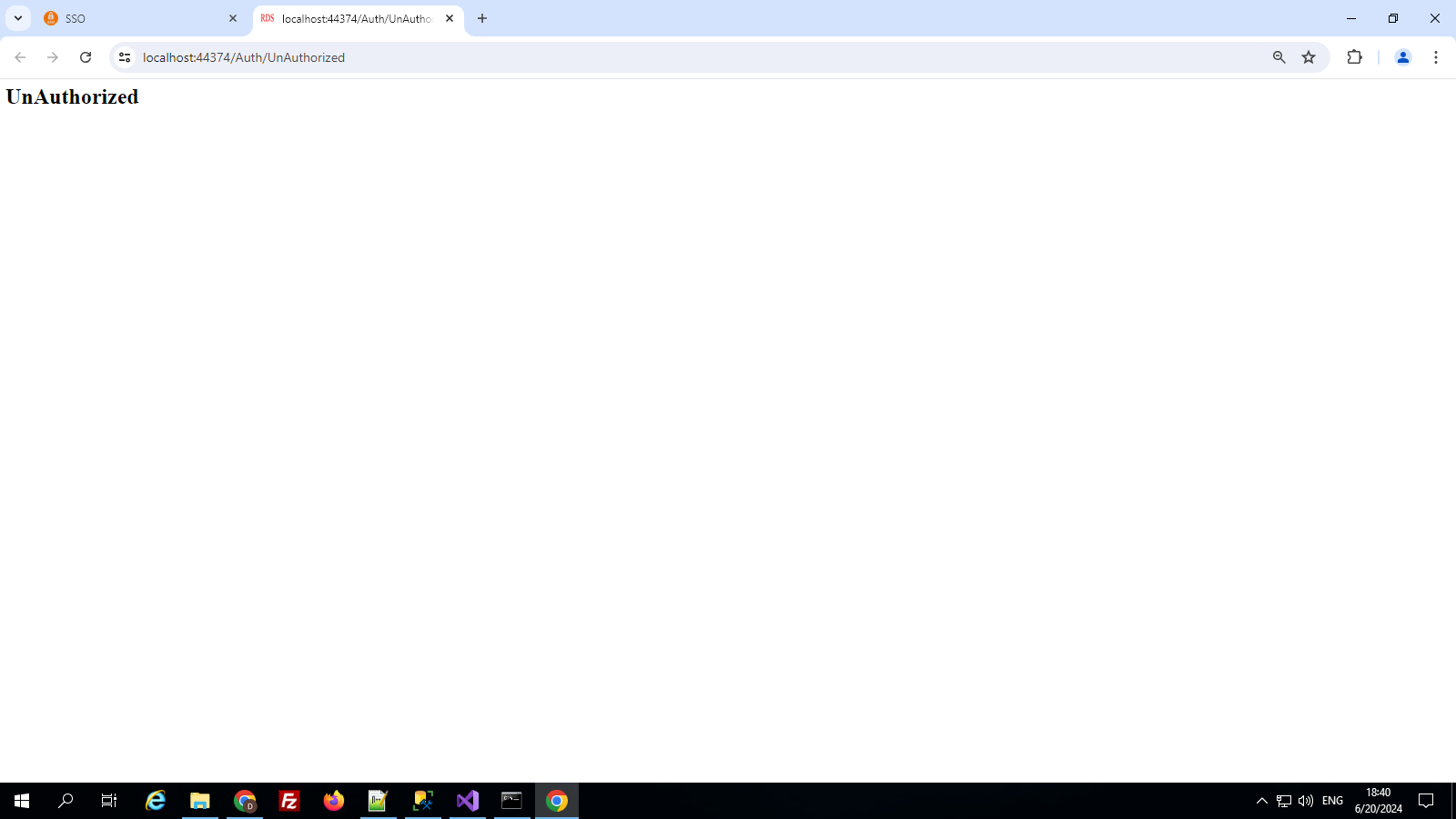
* Also, you can delete a particular one redis key using **“del YourKeyName”** command.



* Also, you can delete all the redis key using **“flushall”** command.



* As soon as the redis key is deleted, the session of the application will be destroyed and the login will be unauthorized in the application.



1. Security parameters
   * –
2. Absolute to do
   * Add NuGet package and their name is **microsoft.extensions.caching.stackexchangeredis**
3. Error handling
   * Not getting the error.
4. Incident Book
   * –
5. Q&A Session
   * -