DOWNLOAD PDF HTTP AND BACKGROUND TASKS

Chapter 1 : Background Tasks in radare2 Â· The Official Radare Blog

In calendrierdelascience.com Core, background tasks can be implemented as hosted services. A hosted service is a class with background task logic that implements the IHostedService interface. This topic provides three hosted service examples: Background task that runs on a timer. Hosted service that activates a.

In this blog post, we will give you some pointers on some pitfalls. You will have a good starting point to do your implementation. Background processing for tasks When running a background task in ASP. When using the hosted service, you do need to keep in mind to handle the dependency injection correctly. The IHostedService runs as a singleton for your task processing. When starting a scheduled task, the task has to be given an independent dependency injection scope. NET Core background processing. The ScopedProcessor class from this article is a good starting point for implementing a scheduled task. The important part of the dependency injection is the Process method: In the basic implementation, the ScopedProcessor class adds a 5-second delay between the processing of the task. Adding scheduling is the next step. Scheduling the background task with Cron expression A Cron expression is a format that let you specify when to trigger the next execution of your task. A Cron expression enables you to precisely specify when to start a task. You have to override the ExecuteAsync to start processing based on the Cron expression. For parsing the Cron expression we use a standard library nuget package NCrontab. This package can parse the Cron expression and determine the next run. WriteLine "Processing starts here" ; return Task. AddSingleton ; Now you are ready for the basic scenario where you only have one instance and do not need advanced monitoring and can miss some processing rounds. When the task runs longer than the interval between the scheduled moments, it will skip starting the process. Gracefully canceling a running task on shutdown, error handling and handling processing when the service was restarted or had downtime can also be improved.

Chapter 2 : calendrierdelascience.com - Should I Block It? (Background Task Host)

Learn how to debug a background task, including background task activation and debug tracing in the Windows event log. Declare background tasks in the application manifest Enable the use of background tasks by declaring them as extensions in the app manifest.

User Not Present User Present Using conditions allows you to ensure that your background task is successful, and also prevents unnecessary usage of system resources. For instance, using the Internet Available condition comes in handy if your background task requires Internet access. Combining an Internet Not Available condition with an Internet Available trigger, for instance, will result in your task failing to ever run. Background Tasks and the Lock Screen Before adding background tasks to your Windows Store app, take into account the impact of the lock screen on background tasks. A user can place up to seven apps on his lock screen. Once an app is added to the lock screen, it gets a higher priority over other apps. This higher priority has a direct affect on background tasks. I mentioned earlier that the app must be on the lock screen before the Timer trigger would work. In fact, the Control Channel and Push Notification triggers also require the app to be on the lock screen. Similarly, several of the System Event triggers also require the app to be added to the lock screen. In addition to having more triggers available, apps placed on the lock screen get more system resources for their background tasks. Each background task associated with a lock screen app is given a two-second CPU quota and can be refreshed every 15 minutes. Apps not on the lock screen only have a one-second quota with a two-hour refresh. Data throughput is scaled based on the average network speed of the device, but lock screen apps are allowed approximately six times more throughput than others. This could involve notifying the user that certain features will be disabled, or having an alternate workflow to implement when the app has been removed from the lock screen. Fortunately, there are two system events that you can attach a background task to in order to manage this: To add the reference to your app, right-click on the References folder under the BackgroundTaskDemo project in the Solutions Explorer and select Add Reference. In the left column, under Solution, select Projects. Check the BackgroundTasks project and select OK. Now your solution should have a new BackgroundTasks library containing a single class, Class1. This is done through the BackgroundTaskBuilder class. The role of the BackgroundTaskBuilder is to match an implementation of IBackgroundTask with a trigger, a unique name and any optional conditions. Each registered task must have a unique name within the scope of the app. Listing 2 shows the modified OnNavigatedTo method. The OnNavigatedTo method with the background registration code added. The TaskEntryPoint is set to the full path to your implementation of IBackgroundTask, which is the namespace followed by the class name. The SystemTrigger constructor also has a second parameter in addition to the trigger type, OneShot, which allows you to specify if this task will be fired more than once. There are no conditions in this example, but you can easily add them through the SetCondition method. After your background task has been configured, a call to the Register method will register the task with Windows. You can use the same instance of BackgroundTaskBuilder to register more than one task, but remember to set the Name property to a new identifier. The final step to adding a background task to your solution is to declare it in the App Manifest. Open the App Manifest Designer by double-clicking the Package. At the top of the design, select the Declarations tab. Each class that implements IBackgroundTask must be declared before Windows will give it permission to register. To add a declaration, select Background Tasks from the Available Declarations dropdown and press the Add button. There are two pieces of information required: The supported task type. This value should be set to the full path of the IBackgroundTask implementation, which is identical to the TaskEntryPoint previously set. Figure 1 shows what the final declaration should look like. Typically, each background task is executed in a special system process called BackgroundTaskHost. However, for some triggers, running in the context of your app makes more sense for instance, a Control Channel trigger. Note that all background tasks will run in the system process for JavaScript apps. In addition, the Start page property is used to set the entry point of a background task in a JavaScript app. With the declaration added to the project, you can now build and deploy the app to your device. Press F5 to build and run the project in

debug mode. Once the app loads, you get an empty black screen because no visual elements have been added to the UI. Opening the Windows Start screen shows a new default tile added that looks similar to Figure 2. The default BackgroundTask tile. To trigger the background task, you need to change the network status of the device. This can be done by turning off the Wi-Fi, unplugging the device from the network or turning on airplane mode. Regardless of the approach, you need to disconnect your device from the network. The live tile for the app should now show the current network status and a date and time stamp. Because Windows controls the timing of live tile updates, it might take several seconds before the new tile appears. Figure 3 shows an example of the new tile. The BackgroundTask tile with a date and time stamp. Also, each task is given a short period of time to complete its execution and will be terminated early if that time limit is reached. This provides an excellent way to extend the life of the background task. The collection will only contain tasks for the current app, and not from any other app. The IBackgroundTaskRegistration allows you to do a couple of things. First, the interface exposes two events: By attaching to these events, your app can respond to status changes of each background task during its execution. In order to effectively use the Progress event, your background task needs to update the Progress property of the IBackgroundTaskInstance passed into the Run method. Listing 2 shows that the first few lines of code cycle through all of the registered tasks and unregisters them. However, there are certain cases when triggering the background task for testing purposes might be a bit challenging. For instance, the minimum refresh time you may set for a Time trigger is 15 minutes. Can you imagine having to wait 15 minutes each time you wanted to test a background test? You can add the Debug Location toolbar by right-clicking on the Visual Studio toolbar and selecting Debug Location from the list of available toolbars. This toolbar is only available -- and will only be visible -- when an app is currently running inside the debugger. Within this toolbar is a Suspend dropdown box. Expanding the dropdown reveals a list of all background tasks registered, as shown in Figure 4. The name listed will be the one you assigned each task in the BackgroundTaskBuilder. On a side note, this toolbar is a great way to test app behavior when suspended and resumed by Windows. All current background tasks, visible from the Suspend dropdown box. Selecting a background task from this list will manually trigger the background task. To test this, put a break point in the Run method of the MyBackgroundTask class. From the Suspend dropdown, select MySampleTask. Visual Studio will trigger your background task and stop at the added break point. Deep Background Background tasks are an important part of many Windows Store apps.

DOWNLOAD PDF HTTP AND BACKGROUND TASKS

Chapter 3 : Run scheduled background tasks in calendrierdelascience.com Core â€“ { Think Rethink }

The Background Task Alert should now show whenever there is an issue with an export/save. Of course you can still click that option and alerts will only show in the Background Tasks panel, but it's important to know how to get it back.

NET Core, background tasks can be implemented as hosted services. A hosted service is a class with background task logic that implements the IHostedService interface. This topic provides three hosted service examples: Background task that runs on a timer. Hosted service that activates a scoped service. The scoped service can use dependency injection. Queued background tasks that run sequentially. View or download sample code how to download The sample app is provided in two versions: Web Host â€” The Web Host is useful for hosting web apps. The example code shown in this topic is from the Web Host version of the sample. For more information, see the Web Host topic. For more information, see the Generic Host topic. Package Reference the Microsoft. App metapackage or add a package reference to the Microsoft. The interface defines two methods for objects that are managed by the host: StopAsync CancellationToken - Triggered when the host is performing a graceful shutdown. StopAsync contains the logic to end the background task and dispose of any unmanaged resources. The hosted service is activated once at app startup and gracefully shutdown at app shutdown. When IDisposable is implemented, resources can be disposed when the service container is disposed. Timed background tasks A timed background task makes use of the System. The timer is disabled on StopAsync and disposed when the service container is disposed on Dispose: LogInformation "Timed Background Service is starting. FromSeconds 5 ; return Task. LogInformation "Timed Background Service is working. LogInformation "Timed Background Service is stopping. Infinite, 0 ; return Task. ConfigureServices with the AddHostedService extension method: No scope is created for a hosted service by default. In the following example, an ILogger is injected into the service: LogInformation "Scoped Processing Service is working. LogInformation "Queued Hosted Service is starting. LogInformation "Queued Hosted Service is stopping. QueueBackgroundWorkItem is called to enqueue the work item:

DOWNLOAD PDF HTTP AND BACKGROUND TASKS

Chapter 4 : web development - Servicing background tasks on a large site - Software Engineering Stack E

How to handle background tasks (fire and forget) and reoccurring background tasks inside calendrierdelascience.com Web Applications, Console or Windows Service with Hangfire. It seems like every application I've ever written, at some point needs to run a task in the background that is to be scheduled at some point in the future.

In a nut shell, my approach for some time now has been to develop a Windows Service. I used to use WCF but getting a distributed transaction to work correctly over WCF always seemed like a pain in the ass. NServiceBus did the trick, I could commit data and create tasks in a transaction and not worry whether my service was up and running at the time. As a simple example, if ever I needed to send an email for example a registration email I would create the user account and fire off a signal to my Windows Service to send the email in a transaction. The message handler on the service side would pick up the message and process accordingly. I have started doing this now for a number of reasons biggest one being deployment is no longer a pain in the ass: This is extremely useful to see what is happening at runtime. Your deployment model for your web apps will work for your service application. IIS provides a few neat features for handling application failures similar in some respects to a Windows Service. It provides a number of alternatives to exposing an API for other apps to consume. If you go this route forgive me for copying and pasting from my original post I would definitely consider running the background logic in a separate web application. There are number of reasons for this: There may be a different security model for the UI displaying information about the running background processes. I would not want to expose this UI to anyone else but the ops team. Also, the web application may run as a different user which has an elevated set of permissions. Furthermore, the application processing the background tasks could be deployed to a separate server if required. Doing this gets back to the marshaling aspect. If you are using Windows Workflow 4. The web hosting approach for services is still fairly new to me, only time will tell if it was the correct choice. So far so good though. Stay away from the applicationhost. I had originally posted a few more links in this message but alas, this is my first post to this exchange and only one link is supported! There was basically two others, to get them Google "Death to Windows Services

Chapter 5 : App Lifecycle - Keep Apps Alive with Background Tasks and Extended Execution

In the previous blog post called background tasks with calendrierdelascience.com Core using the IHostedService Peter described how to use the IHostedInterface for background tasks. In this post, we continue on this subject and add some pointers on how to perform scheduled background tasks.

Background Tasks in radare2 July 3, Recently, I have been working on improving performance in Cutter, the radare2 GUI, especially when working with larger binaries. One major issue was that almost everything that accessed r2, such as updating the list of functions, strings, etc. While this is barely noticeable with smaller binaries, it can lead to a severe impact on usability for larger ones. The obvious solution for this is to somehow run the work in the background and update the UI when it is done. Unfortunately, r2 was never designed to be multi-threaded. This may intuitively seem like a poor design choice, however there are clear reasons behind it: First, apart from potential speed-up, multi-threading would have almost no practical advantages when accessing r2 from its classic command line interface. And second, ensuring all necessary parts of the code are thread-safe would introduce an enormous additional overhead, possibly even lowering the overall performance. I talked to pancake about this topic and he explained me the rough concept he had in mind on how to implement background tasks in r2 as some sort of a compromise between pure single-threading and full multi-threading. The idea is that multiple tasks can be running at the same time, but only one can actually do any work. Some sort of scheduling algorithm would then stop and dispatch the tasks repeatedly, making them essentially run in an interleaved fashion. The obvious disadvantage of this method is that there will be absolutely no speedup when running tasks in parallel, however it does allow executing long-running tasks in the background while still being able to execute shorter-running commands without having to wait, which is, after all, the single most important issue that background tasks in r2 are supposed to solve, so I eventually decided to implement this method and it is now ready to use and available since radare2 version 2. I will now first explain how to use the new tasks feature in r2 from the command line interface and then go further into the actual implementation details to provide some documentation for other r2 developers. The first one, with id 0, is the main task, which is the one on which everything that is not explicitly run in a background task is executed on. The second one, having id 1, is the one we have manually started above. It should be noted, however, that not all commands are currently safe to be run in background tasks. See the section below about task-safety for more information on this. The interface for using tasks currently looks like this: Even though only one task is doing actual work at any point in time, it is still necessary to run each of them on a separate thread in order to be able to switch between tasks while retaining their full callstacks. When the thread is started, the task first checks whether there are any other tasks which are currently running. If that is not the case, it can start executing its command. The result of this behaviour is that the tasks schedule and dispatch each other automatically in a cyclic manner. These functions operate on a global instance of RCons, in which the output is accumulated and can eventually be printed to stdout or used further internally. In its original form, this behaviour is unsuited for concurrent tasks, because they would alternately write into the same buffer, thus creating a corrupt result. However, because these functions are extremely frequently used throughout the entire codebase, this would require refactoring each of these locations. Even worse, the context pointer would have to be passed through all functions that require it in any function they are calling, which would also require changing their signatures. The Main Task As already mentioned above, there is always a single task called the main task, on which everything is executed that is not explicitly run in a background task. Having this task is necessary for scheduling between the main work and any background tasks to work. The main task has no thread explicitly attached to it since its work is generally simply run on the main thread. In the normal radare2 executable, these functions are called once at the beginning and the end of main , respectively. Sleeping Sometimes a task may need to wait for something else to complete while not being able to do anything in this time. The most prominent example for this right now is when r2 waits for user input on the command line. When a task is sleeping, it will be ignored in the scheduling process and all other tasks will be able to continue running. Task-Safety While this approach to implement background tasks prevents any

issues arising from actual multi-threading, certain kinds of race conditions may still occur. Some commands may, for example, temporarily change some eval vars and reset them before exiting. Running such a command, interleaved with another command that accesses one of these eval vars, will thus result in some undesired behaviour. Next Steps Since the basic implementation of tasks now works fairly reliable, the next step will be to use them in Cutter wherever it makes sense. It will be necessary to carefully go through the code of each command that should be run in the background in Cutter in order to ensure task-safety. A few kinds of processes are already executed in tasks in Cutter 1. One feature that is not yet implemented is the possibility to interrupt background tasks while they are still running. It should, however, be possible to implement that quite easily and I plan to do it soon.

Chapter 6 : Background Tasks in Windows Store Apps -- Visual Studio Magazine

While these background tasks are more limited than Celery tasks, they allow you to run tasks in the background while still receiving requests. A Simple Example I have written code, which provides you with a simple example of how you can use such a background task.

The content you requested has been removed. When a user launched an app, it could run wild on the system: Nowadays, things are tougher. In a mobile-first world, apps are restricted to a defined application lifecycle. For apps, the OS is enforcing what has always been good practice, that applications enter a quiesced steady state when not in active use. This is especially important because the application model in the Universal Windows Platform UWP needs to scale from the lowest-end devices, like phones and Internet of Things IoT devices, all the way to the most powerful desktops and Xboxes. For complex apps, the modern application model can seem restrictive at first, but as this article will describe, there are a few ways applications can expand the box and run completing tasks and delivering notifications , even when not in the foreground. The Application Lifecycle In traditional Win32 and. NET desktop development, apps are typically in one of two states: This may seem obvious, but think of an instant messaging application like Skype or a music app like Spotify: All the while, Skype sits in the background waiting for messages to come in, and Spotify keeps playing music. This contrasts with modern apps such as Windows apps built on the UWP , which spend most of their time in a state other than running. Windows apps are always in one of three states: When a user launches a Windows app, for example by tapping on a Tile on the Start menu, the app is activated and enters the running state. As long as the user is interacting with the app, it stays in the running state. If the user navigates away from the app or minimizes it, a suspended event is fired. When the user navigates back to the app, the application threads are unfrozen and the application resumes the running state. From an app execution perspective, the difference between the suspended and not running states is whether the application is allowed to stay resident in memory. When an application is merely suspended, its execution is frozen, but all of its state information stays in memory. Figure 2 shows what happens to resource usage as applications transition through the lifecycle. When an application is activated, it begins to consume memory, typically reaching a relatively stable plateau. When an application is suspended, its memory consumption typically goes downâ€”buffers and used resources are released, and CPU consumption goes to zero enforced by the OS. When an app moves from suspended to not running, both memory and CPU use go to zero, again enforced by the OS. The Windows Emulator that ships with Visual Studio can be immensely helpful for this; it allows developers to target devices with as little MB of memory. For example, a typical use case for a social or communications app is to sign in and sync a bunch of cloud data contacts, feeds, conversation history and so forth. With the basic application lifecycle as described, in order to download contacts, the user would need to keep the app open and in the foreground the entire time! As soon as the user switched to a different app or put the device in their pocket, this would no longer work. There needs to be a mechanism to allow applications to run a bit longer. The Universal Windows Platform introduces the concept of extended execution to help with these types of scenarios. There are two cases where extended execution can be used: At any point during regular foreground execution, while the application is in the running state. The code for these two cases is the same, but the application behaves a little differently in each. In the first case, the application stays in the running state, even if an event that normally would trigger suspension occurs for example, the user navigating away from the application. The application will never receive a suspending event while the execution extension is in effect. When the extension is disposed, the application becomes eligible for suspension again. With the second case, if the application is transitioning to the suspended state, it will stay in a suspending state for the period of the extension. Once the extension expires, the application enters the suspended state without further notification. Figure 3 shows how to use extended execution to extend the uspending state of an application. The application then hooks up a revocation event handler, which is called if Windows can no longer support the extension, for example, if another high-priority task, like a foreground application or an incoming VoIP call, needs the resources. Finally, the application requests the extension and, if successful,

begins its save operation. When an application gets a suspension event, it begins to release or serialize resources. If the app determines it needs more time and takes an extension, it remains in the suspending state until the extension is revoked. This is useful for when the application knows beforehand it will need to continue running in the background, as with the navigation app mentioned earlier. The code is very similar to the previous example, and is shown in Figure 5. This happens because, in this case, the extension is typically revoked only due to resource pressure, a situation that can only be mitigated by releasing resources that is, removing the app from memory. Background tasks are separate components in an application that implement the IBackgroundTask interface. These components can be executed without heavyweight UI frameworks, and typically execute in a separate process although they can also be run in-proc with the primary application executable. A background task is executed when its associated trigger is fired. There are many trigger types supported by Windows, including these background trigger types: Using a background task is a three-step process: The component needs to be created, then declared in the application manifest and then registered at run time. Background tasks are typically implemented in separate Windows Runtime WinRT component projects in the same solution as the UI project. This allows the background task to be activated in a separate process, reducing the memory overhead required by the component. A simple implementation of an IBackgroundTask is shown in Figure 7. IBackgroundTask is a simple interface that defines just one method, Run. When the Run method completes, the background task is terminated. This registration tells Windows the trigger type, the entry point and the executable host of the task, as follows: Finally, the task must also be registered at run time, and this is shown in Figure 8. Here I use the BackgroundTaskBuilder object to register the task with a TimeTrigger that will fire every 30 minutes, if the Internet is available. This is the ideal type of trigger for common operations like updating a tile or periodically syncing small amounts of data. Because background tasks run in the background when the user might be doing something important in the foreground or the device is in standby , they are tightly restricted in the amount of memory and CPU time they can use. Background tasks should be tested on a variety of devices, especially low-end devices, before you publish an application. There are a couple of other things to note, as well: If Battery Saver is available and active typically when the battery is below a certain charge threshold , background tasks are prevented from running until the battery is recharged past the battery-saver threshold. This is no longer the case in Windows 10, but an app must always call BackgroundExecutionManger. RequestAcessAsync to declare its intent to run in the background. A Better Way to Do Contact Syncing There are many background operations that can be completed with either extended execution or with background tasks. ApplicationTrigger like DeviceUseTrigger belongs to a special class of triggers that are triggered directly from the foreground portion of the application. This is done by explicitly calling RequestAsync on the trigger object.

Chapter 7 : Aiohttp â€“ Background Tasks â€“ Edmund Martin

Django Background TasksÂ¶. Django Background Task is a databased-backed work queue for Django, loosely based around Ruby's DelayedJob library. This project was adopted and adapted from this repo.

Imagine a magical processing pixie makes your browser work. Like most of us, the pixie can only do one thing at a time. If we throw many tasks at the pixie, they get added to a big to-do list and are processed in order. Everything else stops when the pixie encounters a script tag or has to run a JavaScript function. The code is downloaded if required and run immediately before further events or rendering can be handled. This is necessary because your script could do anything: Even if there were two or more pixies, the others would need to stop work while the first processed your code. One option is to use Web Workers which can run code concurrently in a separate thread. Another possibility is setTimeout, e. The browser will execute the doSomething function once other immediately-executing tasks have completed. Unfortunately, the function will be called regardless of processing demand. You can read more about requestAnimationFrame here: The tasks are stored in an array as function references: As Paul Lewis notes in his blog post on the subject , the work you do in a requestIdleCallback should be in small chunks. It is not suitable for anything with unpredictable execution times such as manipulating the DOM, which is better done using a requestAnimationFrame callback. You should also be wary of resolving or rejecting Promises, as the callbacks will execute immediately after the idle callback has finished, even if there is no more time remaining. Opera should also gain the feature imminently. Microsoft and Mozilla are both considering the API and it sounds promising. Paul Lewis mentioned above created a simple requestIdleCallback shim. While support is limited today, requestIdleCallback could be an interesting facility to help you maximize web page performance. But what do you think?

Chapter 8 : How to run Background Tasks in calendrierdelascience.com - Scott Hanselman

Easy Background Tasks in calendrierdelascience.com by Jeff Atwood on July 18, As I work on the badge implementation for Stack Overflow, I needed a way to call the code that detects and awards the badges out of band.

Celery can be installed from pip, version 3. I downloaded the installer Redis-x The Flask application factory: More than just this, it sets out a more standardised approach to designing an application. Destiny Vault Raider app structure: Flask application with Redis and Celery. From the diagram, we can see: How the Flask application connects to the Redis message broker. The Message broker talks to the Celery worker. Now, lets tun these ideas into code! Creating the Celery worker: Create an instance of the Celery worker, add the Celery configuration. The Celery configuration will be defined a little later in config. Adding the Celery worker to the app instance: First, I create the setup for the Celery beat schedule, I set the schedule for 5 minutes, which is seconds. Create Celery beat schedule: Next, I create the Config object, with the Celery and Redis settings, for both production and development. Connecting to the Celery and Redis server: I also ping the Redis server to check the connection. I tried to manually add the hostname, port and password as strings and populate the redis. However, the important thing to note is the celery. This is the periodic task that is run every 5 minutes. Starting the Celery workers: To start the Celery workers, you need both a Celery worker and a Beat instance running in parallel. Here are the commands for running them: Calling the asynchronous task: In this case I return back to the index. I had a system in place where I would receive update messages on a private Slack channel â€" depending on how the update went. The Periodic task will be executed every 5 minutes when the Celery Beat scheduler is running. Here I can check the progress from the Celery output: I get updates from the Slack messages. Creating a development Start up script: This can save a lot of time as opening 4 command windows and starting each process separately. The first line changes to our working directory. Then I wait for 5 seconds to allow the Redis server to shutdown. The next 4 commands are used to start the Redis server, Celery worker, Celery Beat worker, and Flask server â€" each started in their own command shell. Redis server, Celery workers and Flask server started via the Startup. Creating a Redis broker and adding it to the app: To start the Celery worker and Beat processes, add the following to your procfile: To start the process, you need to enable the Celery worker Dyno: Note on running Celery and Redis on Heroku: For example the pricing for the package I wanted worked out like this as of November , all figures are per month:

Chapter 9 : Running a long background task in calendrierdelascience.com | The calendrierdelascience.com

A few years back Phil Haack wrote a great article on the dangers of recurring background tasks in calendrierdelascience.com it he points out a few gotchas that are SO common when folks try to do work in the background.

If you have any questions, please follow me on Twitter. Follow codeopinion Hangfire I stumbled upon Hangfire a couple years ago when trying to find solution to running background tasks in a. NET console application running as a service with Topshelf. When I was trying to find a solution, I need to have tasks distributed across multiple worker services. Hangfire has this problem solved. An easy way to perform fire-and-forget, delayed and recurring tasks inside ASP. No Windows Service required. Backed by persistent storages. Open and free for commercial use. Background method calls and their arguments are serialized and may overcome the process boundaries. You can use Hangfire on different machines to get more processing power with no configuration â€" synchronization is performed automatically. I do however just want to give you enough to show you how simple it is to use and let you decide if its a viable solution for you. There are two ways in which you can get started. If you plan on using Hangfire within an ASP. Install-Package HangFire -Version 1. NET Web Application or any other project type. What differs is where you put this configuration. But it is incredibly straight forward. You simply use the GlobalConfiguration class to configure the entry point. Server The server is what processes the background tasks we will define later. In order to create a server, you simply create a BackgroundJobServer. This can be called from any project you wish to act as a Hangfire server. As with configuration, where you place create the BackgroundJobServerwill depend on the project type. These calls do not need to be made from the same application as the server. Fire-and-forget Delay Recurring Any type of Tasks can be called from any client and it will be executed on a Hangfire server. Fire-and-forget You simply call the Enqueue with your action you want to invoke on the server. The Enqueue method does not call the target method immediately, it runs the following steps instead: Serialize a method information and all its arguments. Create a new background job based on the serialized information. Save background job to a persistent storage. Enqueue background job to its queue. When you run this application, here is the console output. You can also create reoccurring tasks and supports CRON expressions. I highly recommend you check out this project if you are looking for this type of functionality.