Hangfire Documentation

Release 1.7

Hangfire OÜ

Mar 17, 2022

Contents

## Overview

Hangfire allows you to kick off method calls outside of the request processing pipeline in a very easy, but reliable way. These method invocations are performed in a *background thread* and called *background jobs*.

From the 10.000-feet view the library consists of three main components: *client*, *storage* and *server*. Here is a small diagram that describes the main processes in Hangfire:
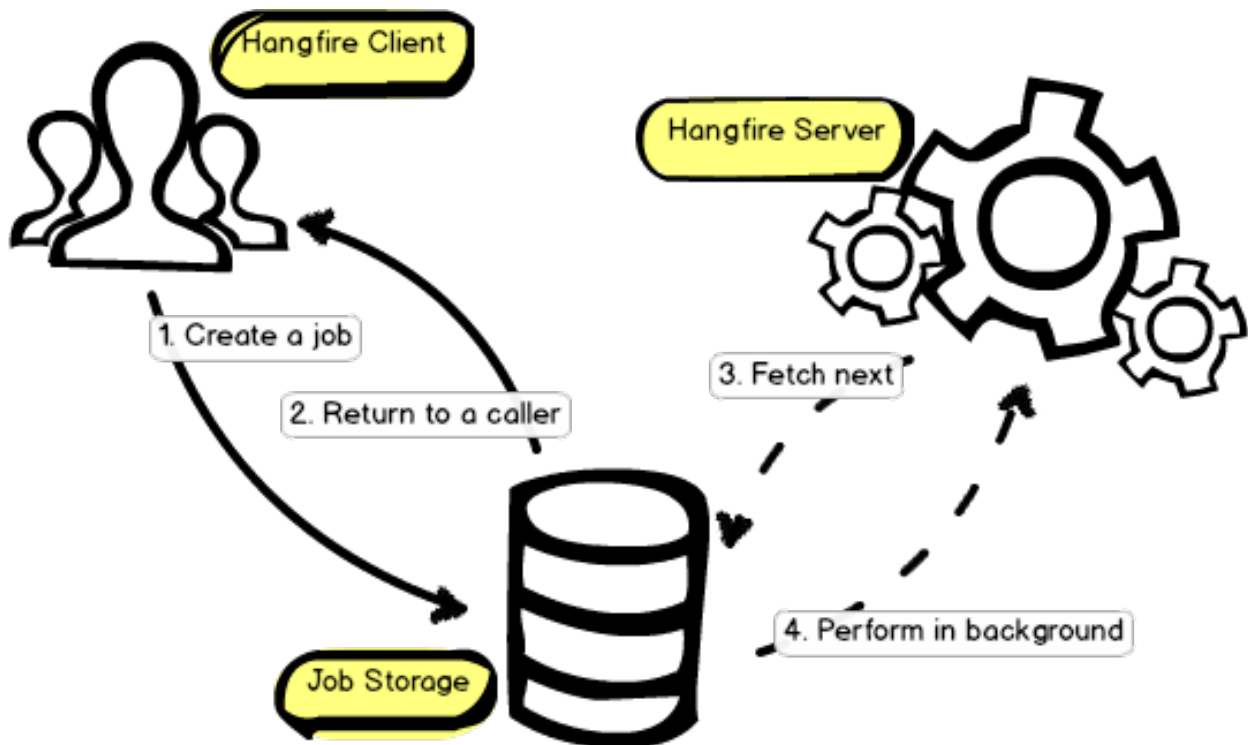
Requirements

Hangfire is not tied to the specific .NET application type. You can use it in ASP.NET *web applications*, non-ASP.NET web applications, in *console applications* or *Windows services*. Here are the requirements:

- .NET Framework 4.5
- Persistent storage (listed below)
- Newtonsoft.Json library 5.0.1

Client

You can create any kind of background jobs using Hangfire: *fire-and-forget* (to offload the method invocation), *delayed* (to perform the call after some time) and *recurring* (to perform methods hourly, daily and so on).

Hangfire does not require you to create special classes. Background jobs are based on regular static or instance methods invocation.

```
var client = new BackgroundJobClient();

client.Enqueue(() => Console.WriteLine("Easy!"));
client.Delay(() => Console.WriteLine("Reliable!"), TimeSpan.FromDays(1));
```

There is also more easy way to create background jobs – the `BackgroundJob` class that allows you to use static methods to perform the creation task.

```
BackgroundJob.Enqueue(() => Console.WriteLine("Hello!"));
```

The control is returned to a caller just after Hangfire serializes the given information and saves it to the *storage*.

CHAPTER 4

Job Storage

Hangfire keeps background jobs and other information that relates to the processing inside a *persistent storage*. Persistence helps background jobs to **survive on application restarts**, server reboots, etc. This is the main distinction between performing background jobs using *CLR's Thread Pool* and *Hangfire*. Different storage backends are supported:

- *SQL Azure, SQL Server 2008 R2* (and later of any edition, including Express)

- *Redis*

SQL Server storage can be empowered with *MSMQ* or RabbitMQ to lower the processing latency.

```
GlobalConfiguration.Configuration.UseSqlServerStorage("db_connection");
```

Server

Background jobs are processed by *Hangfire Server*. It is implemented as a set of dedicated (not thread pool's) background threads that fetch jobs from a storage and process them. Server is also responsible to keep the storage clean and remove old data automatically.

All you need is to create an instance of the `BackgroundJobServer` class and start the processing:

```csharp
using (new BackgroundJobServer())
{
    Console.WriteLine("Hangfire Server started. Press ENTER to exit...");
    Console.ReadLine();
}
```

Hangfire uses reliable fetching algorithm for each storage backend, so you can start the processing inside a web application without a risk of losing background jobs on application restarts, process termination and so on.

Table of Contents

6.1 Getting Started

6.1.1 Requirements

Hangfire works with the majority of .NET platforms: .NET Framework 4.5 or later, .NET Core 1.0 or later, or any platform compatible with .NET Standard 1.3. You can integrate it with almost any application framework, including ASP.NET, ASP.NET Core, Console applications, Windows Services, WCF, as well as community-driven frameworks like Nancy or ServiceStack.

6.1.2 Storage

Storage is a place where Hangfire keeps all the information related to background job processing. All the details like types, method names, arguments, etc. are serialized and placed into storage, no data is kept in a process' memory. The storage subsystem is abstracted in Hangfire well enough to be implemented for RDBMS and NoSQL solutions.

This is the main decision you must make, and the only configuration required before start using the framework. The following example shows how to configure Hangfire with a SQL Server database. Please note that connection string may vary, depending on your environment.

```
GlobalConfiguration.Configuration
    .UseSqlServerStorage(@"Server=.\SQLEXPRESS; Database=Hangfire.Sample; Integrated␣
↪Security=True");
```

6.1.3 Client

The Client is responsible for creating background jobs and saving them into Storage. Background job is a unit of work that should be performed outside of the current execution context, e.g. in background thread, other process, or even on different server – all is possible with Hangfire, even with no additional configuration.

```
BackgroundJob.Enqueue(() => Console.WriteLine("Hello, world!"));
```

Please note this is not a delegate, it's an *expression tree*. Instead of calling the method immediately, Hangfire serializes the type (`System.Console`), method name (`WriteLine`, with all the parameter types to identify it later), and all the given arguments, and places it to Storage.

6.1.4 Server

Hangfire Server processes background jobs by querying the Storage. Roughly speaking, it's a set of background threads that listen to the Storage for new background jobs, and perform them by de-serializing type, method and arguments.

You can place this background job server in any process you want, including dangerous ones like ASP.NET – even if you terminate a process, your background jobs will be retried automatically after restart. So in a basic configuration for a web application, you don't need to use Windows Services for background processing anymore.

```
using (new BackgroundJobServer())
{
    Console.ReadLine();
}
```

6.1.5 Installation

Hangfire is distributed as a couple of NuGet packages, starting from the primary one, Hangfire.Core, that contains all the primary classes as well as abstractions. Other packages like Hangfire.SqlServer provide features or abstraction implementations. To start using Hangfire, install the primary package and choose one of the available storages.

After the release of Visual Studio 2017, a completely new way of installing NuGet packages appeared. So I give up listing all the ways of installing a NuGet package, and fallback to the one available almost everywhere using the `dotnet` app.

```
dotnet add package Hangfire.Core
dotnet add package Hangfire.SqlServer
```

6.1.6 Configuration

Configuration is performed using the `GlobalConfiguration` class. Its `Configuration` property provides a lot of extension methods, both from Hangfire.Core, as well as other packages. If you install a new package, don't hesitate to check whether there are new extension methods.

```
GlobalConfiguration.Configuration
    .SetDataCompatibilityLevel(CompatibilityLevel.Version_170)
    .UseSimpleAssemblyNameTypeSerializer()
    .UseRecommendedSerializerSettings()
    .UseSqlServerStorage("Database=Hangfire.Sample; Integrated Security=True;", new
→SqlServerStorageOptions
    {
        CommandBatchMaxTimeout = TimeSpan.FromMinutes(5),
        SlidingInvisibilityTimeout = TimeSpan.FromMinutes(5),
        QueuePollInterval = TimeSpan.Zero,
        UseRecommendedIsolationLevel = true
    })
```

(continues on next page)

```
        .UseBatches()
        .UsePerformanceCounters();
```

Method calls can be chained, so there's no need to use the class name again and again. Global configuration is made for simplicity, almost every class of Hangfire allows you to specify overrides for storage, filters, etc. In ASP.NET Core environments global configuration class is hidden inside the `AddHangfire` method.

6.1.7 Usage

Here are all the Hangfire components in action, as a fully working sample that prints the "Hello, world!" message from a background thread. You can comment the lines related to server, and run the program several times – all the background jobs will be processed as soon as you uncomment the lines again.

```csharp
using System;
using Hangfire;
using Hangfire.SqlServer;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main()
        {
            GlobalConfiguration.Configuration
                .SetDataCompatibilityLevel(CompatibilityLevel.Version_170)
                .UseColouredConsoleLogProvider()
                .UseSimpleAssemblyNameTypeSerializer()
                .UseRecommendedSerializerSettings()
                .UseSqlServerStorage("Database=Hangfire.Sample; Integrated
→Security=True;", new SqlServerStorageOptions
                {
                    CommandBatchMaxTimeout = TimeSpan.FromMinutes(5),
                    SlidingInvisibilityTimeout = TimeSpan.FromMinutes(5),
                    QueuePollInterval = TimeSpan.Zero,
                    UseRecommendedIsolationLevel = true
                });

            BackgroundJob.Enqueue(() => Console.WriteLine("Hello, world!"));

            using (var server = new BackgroundJobServer())
            {
                Console.ReadLine();
            }
        }
    }
}
```

ASP.NET Applications

You can place the background processing in an ASP.NET application without using additional processes like Windows Services. Hangfire's code is ready for unexpected process terminations, application pool recycles and restarts during the deployment process. Since persistent storages are used, you'll not lose any background job.

Installing Hangfire

Before we start, we'll need a working ASP.NET application, you can use ASP.NET MVC or ASP.NET WebForms, the steps are almost the same. First of all, the following packages should be installed. There are a lot of ways to install NuGet packages, I'll show how to use Package Manager Console window as it doesn't require any screenshots.

```
PM> Install-Package Hangfire.Core
PM> Install-Package Hangfire.SqlServer
PM> Install-Package Hangfire.AspNet
```

Creating a database

As you can see from the snippet above, we'll be using SQL Server as a job storage in this article. Before configuring Hangfire, you'll need to create a database for it or use an existing one. Configuration strings below point to the `HangfireTest` database living in the `SQLEXPRESS` instance on a local machine.

You can use SQL Server Management Studio or any other way to execute the following SQL command. If you are using other database name or instance, ensure you've changed connection strings when configuring Hangfire during the next steps.

```
CREATE DATABASE [HangfireTest]
GO
```

Configuring Hangfire

Depending on the age of your application, we'll make some modification either to the `Startup` class, or the `Global.asax.cs` file. **But not both at the same time**, however nothing terrible will happen in this case, your configuration logic will be executed only once, first invocation wins, but you may get multiple processing servers.

Configuration settings below for new installations only

Some of those settings can be incompatible with existing installations, please see the *Upgrade Guides* instead when upgrading to a newer version.

Using Startup class

If you have a modern (cough, cough) ASP.NET application, then you'd probably have the `Startup.cs` file. This case is the simplest case to bootstrap Hangfire and start using background processing. There are some extension methods and their overloads available for the `IAppBuilder` class.

All you need is to call them, to start using both Hangfire Dashboard and Hangfire Server.

Authorization configuration required for non-local requests

By default only local access is permitted to the Hangfire Dashboard. Dashboard authorization must be configured in order to allow remote access.

```csharp
// Startup.cs
using Hangfire;
using Hangfire.SqlServer;

public class Startup
{
    private IEnumerable<IDisposable> GetHangfireServers()
    {
        GlobalConfiguration.Configuration
            .SetDataCompatibilityLevel(CompatibilityLevel.Version_170)
            .UseSimpleAssemblyNameTypeSerializer()
            .UseRecommendedSerializerSettings()
            .UseSqlServerStorage("Server=.\\SQLEXPRESS; Database=HangfireTest;
→Integrated Security=True;", new SqlServerStorageOptions
            {
                CommandBatchMaxTimeout = TimeSpan.FromMinutes(5),
                SlidingInvisibilityTimeout = TimeSpan.FromMinutes(5),
                QueuePollInterval = TimeSpan.Zero,
                UseRecommendedIsolationLevel = true,
                DisableGlobalLocks = true
            });

        yield return new BackgroundJobServer();
    }

    public void Configuration(IAppBuilder app)
    {
        app.UseHangfireAspNet(GetHangfireServers);
        app.UseHangfireDashboard();

        // Let's also create a sample background job
        BackgroundJob.Enqueue(() => Debug.WriteLine("Hello world from Hangfire!"));

        // ...other configuration logic
    }
}
```

Using Global.asax.cs file

Configured using the Startup class? Skip this section.

If you can't use the Startup class for a reason, just use the HangfireAspNet class and modify the Global.asax.cs file. You'll not have Hangfire Dashboard in this case, but at least you can start the background processing. If you'd like to install the dashboard also, please google how to add the Startup class to your project, and go to the previous section.

```csharp
// Global.asax.cs
using Hangfire;
using Hangfire.SqlServer;

public class MvcApplication : System.Web.HttpApplication
{
    private IEnumerable<IDisposable> GetHangfireServers()
    {
        GlobalConfiguration.Configuration
            .SetDataCompatibilityLevel(CompatibilityLevel.Version_170)
```

(continued from previous page)

```
            .UseSimpleAssemblyNameTypeSerializer()
            .UseRecommendedSerializerSettings()
            .UseSqlServerStorage("Server=.\\SQLEXPRESS; Database=HangfireTest;␣
↪Integrated Security=True;", new SqlServerStorageOptions
            {
                CommandBatchMaxTimeout = TimeSpan.FromMinutes(5),
                SlidingInvisibilityTimeout = TimeSpan.FromMinutes(5),
                QueuePollInterval = TimeSpan.Zero,
                UseRecommendedIsolationLevel = true,
                DisableGlobalLocks = true
            });

        yield return new BackgroundJobServer();
    }

    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        HangfireAspNet.Use(GetHangfireServers);

        // Let's also create a sample background job
        BackgroundJob.Enqueue(() => Debug.WriteLine("Hello world from Hangfire!"));
    }
}
```

You might also need to disable OWIN's Startup class detection, when using initialization based on `Global.asax.cs` file. The problem is `Hangfire.AspNet` package depends on `Microsoft.Owin.SystemWeb` package, and it requires OWIN Startup class to be present in your web application. If the following exception appears, just disable the automatic startup in your `web.config` file as should below.

```
EntryPointNotFoundException: The following errors occurred while attempting to load␣
↪the app.
    - No assembly found containing an OwinStartupAttribute.
    - No assembly found containing a Startup or [AssemblyName].Startup class.
```
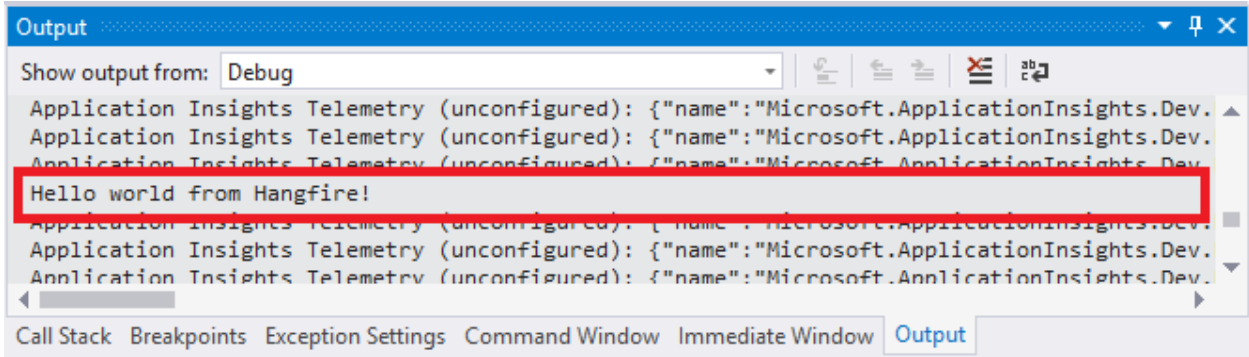
```
<!-- web.config -->
<appSettings>
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  <add key="owin:AutomaticAppStartup" value="false"/>
</appSettings>
```

Running Application

Run your application in the Debug mode by pressing `F5` (this is required to see the output of the `Debug.WriteLine` method). Then check the *Output* window for the following message to see whether background processing has started successfully.

When application is started, open the following URL (assuming your app is running on the 5000 port) to access to the Hangfire Dashboard interface. As we can see, our background job was completed successfully.

Startup class is required for Dashboard UI

Please note, Dashboard UI is available only if you were using the Startup class to configure Hangfire.

```
http://<your-web-app>/hangfire
```


That's all, now you are ready to create other background jobs!

ASP.NET Core Applications

Before we start with our tutorial, we need to have a working ASP.NET Core application. This documentation is devoted to Hangfire, please, read the official ASP.NET Core Documentation to learn the details on how to create and initialize a new web application: Getting Started and Tutorials.

Installing Hangfire

Hangfire is available as a set of NuGet packages, so you need to add them to the `*.csproj` file by adding new `PackageReference` tags as below. Please note that versions in the code snippet below may be outdated, so use versions from the following badges. They are updated in real-time.

```xml
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
  <PackageReference Include="Hangfire.Core" Version="1.7.*" />
  <PackageReference Include="Hangfire.SqlServer" Version="1.7.*" />
  <PackageReference Include="Hangfire.AspNetCore" Version="1.7.*" />
</ItemGroup>
```

Creating a database

As you can see from the snippet above, we'll be using SQL Server as a job storage in this article. Before configuring Hangfire, you'll need to create a database for it, or use an existing one. Configuration strings below point to the `HangfireTest` database living in the `SQLEXPRESS` instance on a local machine.

You can use SQL Server Management Studio or any other way to execute the following SQL command. If you are using an other database name or instance, ensure you've changed the connection strings when configuring Hangfire during the next steps.

```sql
CREATE DATABASE [HangfireTest]
GO
```

Configuring Hangfire

We'll start our configuration process with defining a configuration string for the `Hangfire.SqlServer` package. Consider you have an `sqlexpress` named instance running on localhost, and **just created the "HangfireTest" database**. The current user should be able to create tables, to allow automatic migrations to do their job.

Also, the `Hangfire.AspNetCore` package has a logging integration with ASP.NET Core applications. Hangfire's log messages are sometimes very important and help to diagnose different issues. `Information` level allows to see how Hangfire is working, and `Warning` and higher log levels help to investigate problems.

Configuring Settings

Open the `appsettings.json` file, and add the highlighted lines from the following snippet.

```json
{
  "ConnectionStrings": {
    "HangfireConnection": "Server=.\\sqlexpress;Database=HangfireTest;Integrated
↪Security=SSPI;"
```

```
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning",
      "Hangfire": "Information"
    }
  }
}
```

After updating the application settings, open the `Startup.cs` file. The startup class is the heart of an ASP.NET Core application's configuration. First, we need to import the `Hangfire` namespace.

```
// ...
using Microsoft.Extensions.DependencyInjection;
using Hangfire;
using Hangfire.SqlServer;
```

Registering Services

Dependency Injection is one of the primary techniques introduced in ASP.NET Core. The `Hangfire.AspNetCore` integration package adds an extension method to register all the services, their implementation, as well as logging and a job activator. As a parameter, it takes an action that allows to configure Hangfire itself.

Configuration settings below for new installations only

Some of those settings can be incompatible with existing installations, please see the *Upgrade Guides* instead, when upgrading to a newer version.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add Hangfire services.
    services.AddHangfire(configuration => configuration
        .SetDataCompatibilityLevel(CompatibilityLevel.Version_170)
        .UseSimpleAssemblyNameTypeSerializer()
        .UseRecommendedSerializerSettings()
        .UseSqlServerStorage(Configuration.GetConnectionString("HangfireConnection"),
→new SqlServerStorageOptions
        {
            CommandBatchMaxTimeout = TimeSpan.FromMinutes(5),
            SlidingInvisibilityTimeout = TimeSpan.FromMinutes(5),
            QueuePollInterval = TimeSpan.Zero,
            UseRecommendedIsolationLevel = true,
            DisableGlobalLocks = true
        }));

    // Add the processing server as IHostedService
    services.AddHangfireServer();

    // Add framework services.
    services.AddMvc();
}
```

Adding Dashboard UI

After registering Hangfire types, you can now choose features you need to add to your application. The following snippet shows you how to add the Dashboard UI to use all the Hangfire features immediately. The following lines are fully optional, and you can remove them completely, if your application will only create background jobs, as separate application will process them.

Authorization configuration required for non-local requests

By default only local access is permitted to the Hangfire Dashboard. Dashboard authorization must be configured in order to allow remote access.

```
public void Configure(IApplicationBuilder app, IBackgroundJobClient backgroundJobs,
→IHostingEnvironment env)
{
    // ...
    app.UseStaticFiles();

    app.UseHangfireDashboard();
    backgroundJobs.Enqueue(() => Console.WriteLine("Hello world from Hangfire!"));

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Starting with `Hangfire.AspNetCore 1.7.8`, Hangfire officially supports ASP.NET Core 3.0 endpoint routing. When using `RequireAuthorization` with `MapHangfireDashboard`, be cautious that only local access is allowed by default.

```
// This method gets called by the runtime. Use this method to configure the HTTP
→request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapHangfireDashboard();
    });
}
```

Running Application

Run the following command to start an application, or click the `F5` button in Visual Studio.

```
dotnet run
```

After the application has been started and background processing was started successfully, the following messages should appear.

```
info: Hangfire.SqlServer.SqlServerStorage[0]
    Start installing Hangfire SQL objects...
    Hangfire SQL objects installed.
    Using job storage: 'SQL Server: .\@AspNetCoreTest'
    Using the following options for SQL Server job storage:
        Queue poll interval: 00:00:15.
info: Hangfire.BackgroundJobServer[0]
    Starting Hangfire Server...
    Using the following options for Hangfire Server:
        Worker count: 20
        Listening queues: 'default'
        Shutdown timeout: 00:00:15
        Schedule polling interval: 00:00:15
```

These lines contain messages regarding SQL Server Job Storage that is used to persist background jobs, and the Background Job Server, which is processing all the background jobs.

The following message should also appear, since we created background job, whose only behavior is to write a message to the console.

```
Hello world from Hangfire!
```

When the application has started, open the following URL (assuming your app is running on the 5000 port), to access to the Hangfire Dashboard interface. As we can see, our background job was completed successfully.

```
http://localhost:5000/hangfire
```


When you finished working with the application, press the `Ctrl+C` in your console window to stop the application. The following message should appear telling you that background processing server was stopped gracefully.

```
info: Hangfire.BackgroundJobServer[0]
      Hangfire Server stopped.
```

You can also kill your process, but in this case some background jobs may be delayed in invocation.

6.2 Configuration

Starting from version 1.4, `GlobalConfiguration` class is the preferred way to configure Hangfire. This is an entry point for a couple of methods, including ones from third-party storage implementations or other extensions. The usage is simple, just include `Hangfire` namespace in your application initialization class and discover extension methods for the `GlobalConfiguration.Configuration` property.

For example, in ASP.NET applications, you can place initialization logic to the `Global.asax.cs` file:

```csharp
using Hangfire;

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        // Storage is the only thing required for basic configuration.
        // Just discover what configuration options do you have.
        GlobalConfiguration.Configuration
            .UseSqlServerStorage("<name or connection string>");
            //.UseActivator(...)
            //.UseLogProvider(...)
    }
}
```

For OWIN-based applications (ASP.NET MVC, Nancy, ServiceStack, FubuMVC, etc.), place the configuration lines to the OWIN Startup class.

```csharp
using Hangfire;

[assembly: OwinStartup(typeof(Startup))]
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        GlobalConfiguration.Configuration.UseSqlServerStorage("<name or connection␣
↪string>");
    }
}
```

For other applications, place it somewhere **before** calling other Hangfire methods.

6.2.1 Using Dashboard

Hangfire Dashboard is a place where you could find all the information about your background jobs. It is written as an OWIN middleware (if you are not familiar with OWIN, don't worry), so you can plug it into your ASP.NET, ASP.NET MVC, Nancy, ServiceStack application as well as use OWIN Self-Host feature to host Dashboard inside console applications or in Windows Services.

- *Adding Dashboard (OWIN)*
- *Configuring Authorization*
- *Read-only view*
- *Change URL Mapping*
- *Change* Back to site *Link*
- *Multiple Dashboards*

Adding Dashboard (OWIN)

Additional package required for ASP.NET + IIS

Before moving to the next steps, ensure you have Microsoft.Owin.Host.SystemWeb package installed, otherwise you'll have different strange problems with the Dashboard.

OWIN Startup class is intended to keep web application bootstrap logic in a single place. In Visual Studio 2013 you can add it by right clicking on the project and choosing the *Add / OWIN Startup Class* menu item.
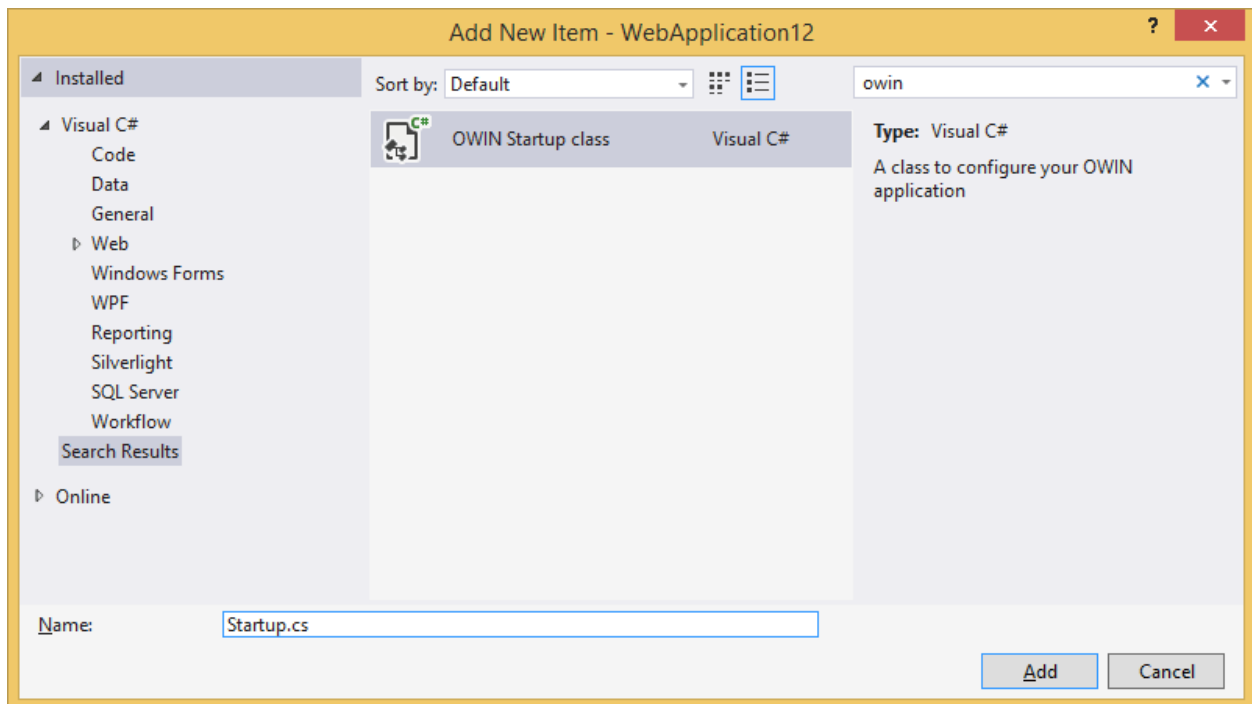
If you have Visual Studio 2012 or earlier, just create a regular class in the root folder of your application, name it `Startup` and place the following contents:

```csharp
using Hangfire;
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(MyWebApplication.Startup))]
```

(continues on next page)

```
namespace MyWebApplication
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Map Dashboard to the `http://<your-app>/hangfire` URL.
            app.UseHangfireDashboard();
        }
    }
}
```

After performing these steps, open your browser and hit the *http://<your-app>/hangfire* URL to see the Dashboard.

Authorization configuration required

By default Hangfire allows access to Dashboard pages **only for local requests**. In order to give appropriate rights for production use, please see the *Configuring Authorization* section.

Configuring Authorization

Hangfire Dashboard exposes sensitive information about your background jobs, including method names and serialized arguments as well as gives you an opportunity to manage them by performing different actions – retry, delete, trigger, etc. So it is really important to restrict access to the Dashboard.

To make it secure by default, only **local requests are allowed**, however you can change this by passing your own implementations of the IDashboardAuthorizationFilter interface, whose Authorize method is used to allow or prohibit a request. The first step is to provide your own implementation.

Don't want to reinvent the wheel?

User, role and claims -based as well as basic access authentication-based (simple login-password auth) authorization filters available as a NuGet package Hangfire.Dashboard.Authorization.

```
public class MyAuthorizationFilter : IDashboardAuthorizationFilter
{
    public bool Authorize(DashboardContext context)
    {
        // In case you need an OWIN context, use the next line, `OwinContext` class
        // is the part of the `Microsoft.Owin` package.
        var owinContext = new OwinContext(context.GetOwinEnvironment());

        // Allow all authenticated users to see the Dashboard (potentially dangerous).
        return owinContext.Authentication.User.Identity.IsAuthenticated;
    }
}
```

For ASP.NET Core environments, use the GetHttpContext extension method defined in the Hangfire. AspNetCore package.

```
public class MyAuthorizationFilter : IDashboardAuthorizationFilter
{
```

```
    public bool Authorize(DashboardContext context)
    {
        var httpContext = context.GetHttpContext();

        // Allow all authenticated users to see the Dashboard (potentially dangerous).
        return httpContext.User.Identity.IsAuthenticated;
    }
}
```

The second step is to pass it to the `UseHangfireDashboard` method. You can pass multiple filters, and the access will be granted only if *all of them* return `true`.

```
app.UseHangfireDashboard("/hangfire", new DashboardOptions
{
    Authorization = new [] { new MyAuthorizationFilter() }
});
```

Method call order is important

Place a call to the `UseHangfireDashboard` method **after other authentication methods** in your OWIN Startup class. Otherwise authentication may not work for you.

```
public void Configuration(IAppBuilder app)
{
    app.UseCookieAuthentication(...); // Authentication – first
    app.UseHangfireDashboard();       // Hangfire – last
}
```

Read-only view

The read-only dashboard view prevents users from changing anything, such as deleting or enqueueing jobs. It is off by default, meaning that users have full control. To enable it, set the `IsReadOnlyFunc` property of the `DashboardOptions`:

```
app.UseHangfireDashboard("/hangfire", new DashboardOptions
{
    IsReadOnlyFunc = (DashboardContext context) => true
});
```

Change URL Mapping

By default, `UseHangfireDashboard` method maps the Dashboard to the `/hangfire` path. If you want to change this for one reason or another, just pass your URL path.

```
// Map the Dashboard to the root URL
app.UseHangfireDashboard("");

// Map to the `/jobs` URL
app.UseHangfireDashboard("/jobs");
```

Change *Back to site* Link

By default, *Back to site* link (top-right corner of Dashboard) leads you to the root URL of your application. In order to change it, use the `DashboardOptions` class.

```
// Change `Back to site` link URL
var options = new DashboardOptions { AppPath = "http://your-app.net" };
// Make `Back to site` link working for subfolder applications
var options = new DashboardOptions { AppPath = VirtualPathUtility.ToAbsolute("~") };

app.UseHangfireDashboard("/hangfire", options);
```

Multiple Dashboards

You can also map multiple dashboards that show information about different storages.

```
var storage1 = new SqlServerStorage("Connection1");
var storage2 = new SqlServerStorage("Connection2");

app.UseHangfireDashboard("/hangfire1", new DashboardOptions(), storage1);
app.UseHangfireDashboard("/hangfire2", new DashboardOptions(), storage2);
```

6.2.2 Using SQL Server

SQL Server is the default storage for Hangfire – it is well known to many .NET developers and used in many project environments. It may be interesting that in the early stage of Hangfire development, Redis was used to store information about jobs, and SQL Server storage implementation was inspired by that NoSql solution. But back to the SQL Server. . .

SQL Server storage implementation is available through the `Hangfire.SqlServer` NuGet package. To install it, type the following command in your NuGet Package Console window:

```
Install-Package Hangfire.SqlServer
```

This package is a dependency of the Hangfire's bootstrapper package `Hangfire`, so if you installed it, you don't need to install the `Hangfire.SqlServer` separately – it was already added to your project.

Supported database engines

Microsoft SQL Server 2008R2 (any edition, including LocalDB) and later, **Microsoft SQL Azure**.

Snapshot isolation is not supported!

Applies only to Hangfire < 1.5.9: Ensure your database doesn't use the snapshot isolation level, and the `READ_COMMITTED_SNAPSHOT` option (another name is *Is Read Committed Snapshot On*) **is disabled**. Otherwise some of your background jobs will not be processed.

Configuration

The package provides extension methods for `GlobalConfiguration` class. Choose either a connection string to your SQL Server or a connection string name, if you have it.

```
GlobalConfiguration.Configuration
    // Use connection string name defined in `web.config` or `app.config`
    .UseSqlServerStorage("db_connection")
    // Use custom connection string
    .UseSqlServerStorage(@"Server=.\sqlexpress; Database=Hangfire; Integrated
→Security=SSPI;");
```

Starting from version 1.7.0 it is recommended to set the following options for new installations (for existing ones, please see *Upgrading to Hangfire 1.7*). These settings will be turned on by default in 2.0, but meanwhile we should preserve backward compatibility.

```
GlobalConfiguration.Configuration
    .UseSqlServerStorage("db_connection", new SqlServerStorageOptions
    {
        CommandBatchMaxTimeout = TimeSpan.FromMinutes(5),
        SlidingInvisibilityTimeout = TimeSpan.FromMinutes(5),
        QueuePollInterval = TimeSpan.Zero,
        UseRecommendedIsolationLevel = true,
        DisableGlobalLocks = true // Migration to Schema 7 is required
    });
```

Installing objects

Hangfire leverages a couple of tables and indexes to persist background jobs and other information related to the processing:

- HangFire.Counter
- HangFire.Hash
- HangFire.Job
- HangFire.JobParameter
- HangFire.JobQueue
- HangFire.List
- HangFire.Schema
- HangFire.Server
- HangFire.Set
- HangFire.State

Some of these tables are used for the core functionality, others fulfill the extensibility needs (making possible to write extensions without changing the underlying schema). Advanced objects like stored procedures, triggers and so on are not used to keep things as simple as possible and allow the library to be used with SQL Azure.

SQL Server objects are **installed automatically** from the `SqlServerStorage` constructor by executing statements described in the `Install.sql` file (which is located under the `tools` folder in the NuGet package). Which contains the migration script, so new versions of Hangfire with schema changes can be installed seamlessly, without your intervention.

If you want to install objects manually, or integrate it with your existing migration subsystem, pass your decision through the SQL Server storage options:

```
var options = new SqlServerStorageOptions
{
    PrepareSchemaIfNecessary = false
};

GlobalConfiguration.Configuration.UseSqlServerStorage("<name or connection string>",
→options);
```

You can isolate HangFire database access to just the HangFire schema. You need to create a separate HangFire user and grant the user access only to the HangFire schema. The HangFire user will only be able to alter the HangFire schema. Below is an example of using a contained database user for HangFire. The HangFire user has least privileges required but still allows it to upgrade the schema correctly in the future.

```
CREATE USER [HangFire] WITH PASSWORD = 'strong_password_for_hangfire'
GO

IF NOT EXISTS (SELECT 1 FROM sys.schemas WHERE [name] = 'HangFire') EXEC ('CREATE
→SCHEMA [HangFire]')
GO

ALTER AUTHORIZATION ON SCHEMA::[HangFire] TO [HangFire]
GO

GRANT CREATE TABLE TO [HangFire]
GO
```

Configuring the Polling Interval

One of the main disadvantage of raw SQL Server job storage implementation – it uses the polling technique to fetch new jobs. Starting from Hangfire 1.7.0 it's possible to use `TimeSpan.Zero` as a polling interval, when `SlidingInvisibilityTimeout` option is set.

```
var options = new SqlServerStorageOptions
{
    SlidingInvisibilityTimeout = TimeSpan.FromMinutes(5),
    QueuePollInterval = TimeSpan.Zero
};

GlobalConfiguration.Configuration.UseSqlServerStorage("<name or connection string>",
→options);
```

This is the recommended value in that version, but you can decrease the polling interval if your background jobs can tolerate additional delay before the invocation.

6.2.3 Using SQL Server with MSMQ

This extension will be deprecated soon

Hangfire.SqlServer 1.7.X versions have long-polling feature implemented when using the recommended settings with the latest schema version. Additional technology like MSMQ complicates the application infrastructure, additional storage like MSMQ brings consistency issues on data loss, and MSMQ itself is a complicated technology, especially with DTC transactions. Therefore, the number of disadvantages outweigh all the advantages of using this extension.

[Hangfire.SqlServer.MSMQ](#) extension changes the way Hangfire handles job queues. Default *[implementation](#)* uses regular SQL Server tables to organize queues, and this extensions uses transactional MSMQ queues to process jobs. Please note that starting from 1.7.0 it's possible to use `TimeSpan.Zero` as a polling delay in Hangfire.SqlServer, so think twice before using MSMQ.

Installation

MSMQ support for SQL Server job storage implementation, like other Hangfire extensions, is a NuGet package. So, you can install it using NuGet Package Manager Console window:

```
PM> Install-Package Hangfire.SqlServer.Msmq
```

Configuration

To use MSMQ queues, you should do the following steps:

1. **Create them manually on each host**. Don't forget to grant appropriate permissions. Please note that queue storage is limited to 1048576 KB by default (approximately 2 millions enqueued jobs), you can increase it through the MSMQ properties window.

2. Register all MSMQ queues in current `SqlServerStorage` instance.

If you are using **only default queue**, call the `UseMsmqQueues` method just after `UseSqlServerStorage` method call and pass the path pattern as an argument.

```
GlobalConfiguration.Configuration
    .UseSqlServerStorage("<connection string or its name>")
    .UseMsmqQueues(@"FormatName:Direct=OS:localhost\hangfire-{0}");
```

To use multiple queues, you should pass them explicitly:

```
GlobalConfiguration.Configuration
    .UseSqlServerStorage("<connection string or its name>")
    .UseMsmqQueues(@"FormatName:Direct=OS:localhost\hangfire-{0}", "critical",
→"default");
```

Limitations

- Only transactional MSMQ queues supported for reliability reasons inside ASP.NET.

- You can not use both SQL Server Job Queue and MSMQ Job Queue implementations in the same server (see below). This limitation relates to Hangfire Server only. You can still enqueue jobs to whatever queues and watch them both in Hangfire Dashboard.

Transition to MSMQ queues

If you have a fresh installation, just use the `UseMsmqQueues` method. Otherwise, your system may contain unprocessed jobs in SQL Server. Since one Hangfire Server instance can not process job from different queues, you should deploy *[multiple instances](#)* of Hangfire Server, one listens only MSMQ queues, another – only SQL Server queues. When the latter finish its work (you can see this in Dashboard – your SQL Server queues will be removed), you can remove it safely.

If you are using default queue only, do this:

```
/* This server will process only SQL Server table queues, i.e. old jobs */
var oldStorage = new SqlServerStorage("<connection string or its name>");
var oldOptions = new BackgroundJobServerOptions();

app.UseHangfireServer(oldOptions, oldStorage);

/* This server will process only MSMQ queues, i.e. new jobs */
GlobalConfiguration.Configuration
    .UseSqlServerStorage("<connection string or its name>")
    .UseMsmqQueues(@"FormatName:Direct=OS:localhost\hangfire-{0}");

app.UseHangfireServer();
```

If you use multiple queues, do this:

```
/* This server will process only SQL Server table queues, i.e. old jobs */
var oldStorage = new SqlServerStorage("<connection string>");
var oldOptions = new BackgroundJobServerOptions
{
    Queues = new [] { "critical", "default" }, // Include this line only if you have
→multiple queues
};

app.UseHangfireServer(oldOptions, oldStorage);

/* This server will process only MSMQ queues, i.e. new jobs */
GlobalConfiguration.Configuration
    .UseSqlServerStorage("<connection string or its name>")
    .UseMsmqQueues(@"FormatName:Direct=OS:localhost\hangfire-{0}", "critical",
→"default");

app.UseHangfireServer();
```

6.2.4 Using Redis

Hangfire Pro subscription required

Starting from Hangfire 1.2, this feature is a part of Hangfire Pro package set

Hangfire with Redis job storage implementation processes jobs much faster than with SQL Server storage. On my development machine I observed more than 4x throughput improvement with empty jobs (method that does not do anything). Hangfire.Pro.Redis leverages the BRPOPLPUSH command to fetch jobs, so the job processing latency is kept to minimum.
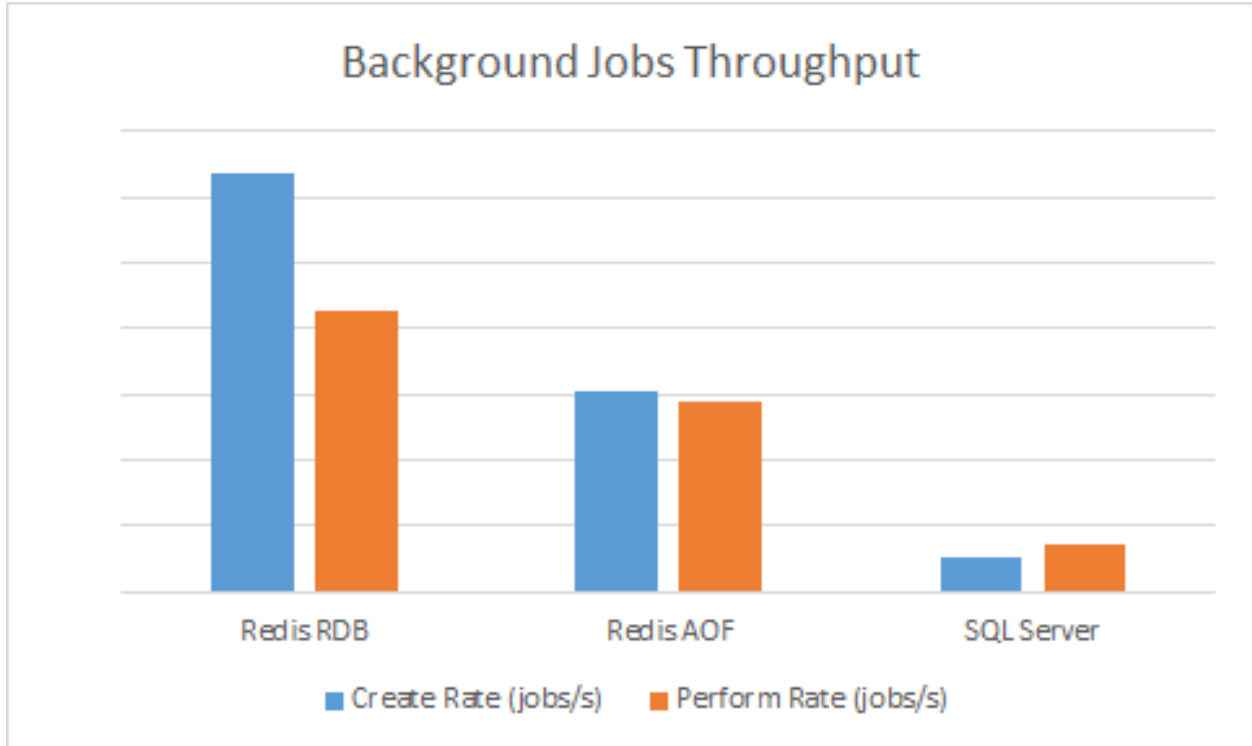
Please, see the downloads page to obtain latest version of Redis. If you unfamiliar with this great storage, please see its documentation. Binaries for Windows are available through NuGet (32-bit, 64-bit) and Chocolatey galleries (64-bit package only).

Limitations

Multiple Redis endpoints are **only supported** in Redis Cluster configuration starting from Hangfire.Pro.Redis 2.1.0. You can't use multiple detached masters or Redis Sentinel configurations.

Redis Configuration

Please read the official Redis documentation to learn how to configure it, especially Redis Persistence and Redis Administration sections to get started with the fundamentals. The following options should be configured to run your background jobs smoothly.

Ensure the following options are configured

These values are default for on-premise Redis installations, but other environments may have different defaults, for example **Azure Redis Cache** and **AWS ElastiCache use non-compatible settings** by default.

```
# Hangfire neither expect that non-expired keys are deleted,
# nor expiring keys are evicted before the expiration time.
maxmemory-policy noeviction

# Non-zero value cause long-running background jobs to be
# processed multiple times due to connection being closed.
# ONLY FOR Hangfire.Pro.Redis 1.X!
timeout 0
```

Hangfire.Pro.Redis 2.x

Redis 2.6.12 is required

Installation

Ensure that you have configured the private Hangfire Pro NuGet feed as written here, and use your favorite NuGet client to install the `Hangfire.Pro.Redis` package:

```
PM> Install-Package Hangfire.Pro.Redis
```

If your project targets .NET Core, just add a dependency in your `*.csproj` file:

```xml
<ItemGroup>
  <PackageReference Include="Hangfire.Pro.Redis" Version="2.8.2" />
</ItemGroup>
```

Configuration

After installing the package, a couple of the `UseRedisStorage` extension method overloads will be available for the `IGlobalConfiguration` interface. They allow you to configure Redis job storage, using both *configuration string* and Hangfire-specific *options*.

Connection string

The basic one is the following, will connect to the Redis on *localhost* using the default port, database and options:

```
GlobalConfiguration.Configuration.UseRedisStorage();
```

For ASP.NET Core projects, call the `UseRedisStorage` method from the `AddHangfire` method delegate:

```
services.AddHangfire(configuration => configuration.UseRedisStorage());
```

You can customize the connection string using the StackExchange.Redis' configuration string format. Please read their documentation for details. The values for the following options have their own defaults in Hangfire, but can be overridden in the *connection string*:

Option	Default
syncTimeout	30000
allowAdmin	true

```
GlobalConfiguration.Configuration
    .UseRedisStorage("contoso5.redis.cache.windows.net,abortConnect=false,ssl=true,
→password=...");
```

Redis Cluster support

You can use a single endpoint to connect to a Redis cluster, Hangfire will detect other instances automatically by querying the node configuration. However, it's better to pass multiple endpoints in order to mitigate connectivity issues, when some of endpoints aren't available, e.g. during the failover process.

Since Hangfire requires transactions, and Redis doesn't support ones that span multiple hash slots, you also need to configure the prefix to assign it to the same hash tag:

```
GlobalConfiguration.Configuration.UseRedisStorage(
    "localhost:6379,localhost:6380,localhost:6381",
    new RedisStorageOptions { Prefix = "{hangfire-1}:" });
```

This will bind all the keys to a single Redis instance. To be able to fully utilize your Redis cluster, consider using multiple `JobStorage` instances and leveraging some load-balancing technique (round-robin is enough for the most cases). To do so, pick different hash tags for different storages and ensure they are using hash slots that live on different masters by using commands `CLUSTER NODES` and `CLUSTER KEYSLOT`.

Passing options

You can also pass the Hangfire-specific options for Redis storage by using the `RedisStorageOptions` class instances:

```
var options = new RedisStorageOptions
{
    Prefix = "hangfire:app1:"
};

GlobalConfiguration.Configuration.UseRedisStorage("localhost", options);
```

The following options are available for configuration:

Option	Default	Description
Prefix	`hangfire:`	Prefix for all Redis keys related to Hangfire.
Database	`null`	Redis database number to be used by Hangfire. When null, then the defaultDatabase option from the configuration string is used.
Max-Suc-ceed-edListLength	`10000`	Maximum visible background jobs in the succeed list to prevent it from growing indefinitely.
MaxDelet-edListLength	`1000`	Maximum visible background jobs in the deleted list to prevent it from growing indefinitely.
Invisibil-ityTime-out	`TimeSpan. FromMinutes(30)`	**Obsolete since 2.4.0** Time interval, within which background job is considered to be still successfully processed by a worker. When a timeout is elapsed, another worker will be able to pick the same background job.
Sub-scrip-tionIn-tegrity-Timeout	`TimeSpan. FromHours(1)`	**Obsolete since 2.1.3** Timeout for subscription-based fetch. The value should be high enough (hours) to decrease the stress on a database. This is an additional layer to provide integrity, because otherwise subscriptions can be active for weeks, and bad things may happen during this time.

Hangfire.Pro.Redis 1.x

This is the old version of Redis job storage for Hangfire. It is based on ServiceStack.Redis 3.71, and has no SSL and .NET Core support. No new features will be added for this version. **This version is deprecated**, switch to the new version to get the new features.

Configuration

Hangfire.Pro.Redis package contains some extension methods for the `GlobalConfiguration` class:

```
GlobalConfiguration.Configuration
    // Use localhost:6379
    .UseRedisStorage();
    // Using hostname only and default port 6379
    .UseRedisStorage("localhost");
    // or specify a port
    .UseRedisStorage("localhost:6379");
    // or add a db number
    .UseRedisStorage("localhost:6379", 0);
    // or use a password
    .UseRedisStorage("password@localhost:6379", 0);

// or with options
var options = new RedisStorageOptions();
GlobalConfiguration.Configuration
    .UseRedisStorage("localhost", 0, options);
```

Connection pool size

Hangfire leverages connection pool to get connections quickly and shorten their usage. You can configure the pool size to match your environment needs:

```
var options = new RedisStorageOptions
{
    ConnectionPoolSize = 50 // default value
};

GlobalConfiguration.Configuration.UseRedisStorage("localhost", 0, options);
```

Using key prefixes

If you are using a shared Redis server for multiple environments, you can specify unique prefix for each environment:

```
var options = new RedisStorageOptions
{
    Prefix = "hangfire:"; // default value
};

GlobalConfiguration.Configuration.UseRedisStorage("localhost", 0, options);
```

6.2.5 Configuring logging

Logging plays an important role in background processing, where work is performed behind the scenes. *Dashboard UI* can greatly help to reveal problems with user code, background jobs themselves. But it works by querying the job storage and requires the information is properly written first, before displaying it.

Even if Hangfire itself doesn't contain any errors, there may be connectivity issues, network blips, problems with the storage instance, different timeout issues and so on. And without logging it's very hard to diagnose those problems and understand what to do – exceptions are thrown in background and may get unnoticed.

Logging subsystem in Hangfire is abstracted by using the LibLog package to allow you to integrate it with any infrastructure. Also, you don't need to configure logging if your application doesn't create any background processing servers – client methods don't log anything, they just throw exceptions on errors.

.NET Core and ASP.NET Core

Hangfire.NetCore and Hangfire.AspNetCore packages provide the simplest way to integrate Hangfire into modern .NET Core applications. It delegates the logging implementation to the Microsoft.Extensions.Logging package, so the only required method to call is the `AddHangfire` method:

Listing 1: Startup.cs

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // ...
    services.AddHangfire(config => config.UseXXXStorage());
}
```

You can also change the minimal logging level for background processing servers to capture lifetime events like "server is starting" and "server is stopping" ones. These events are very important to debug cases when background processing isn't working, because all the processing servers are inactive.

Listing 2: appsettings.json

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning",
      "Hangfire": "Information"
    }
  }
}
```

Once integration is complete, please refer to the official Logging in ASP.NET Core article to learn how to configure the logging subsystem of .NET Core itself.

.NET Framework

If your application uses one of the following libraries, no manual action is required in the most cases. Hangfire knows about these loggers and uses reflection to determine the first available one (in the order defined below) and to call the corresponding methods when logging. And since reflection is used, there are no unnecessary package or assembly references.

1. Serilog
2. NLog
3. Log4Net
4. EntLib Logging
5. Loupe
6. Elmah

Automatic wiring works correctly when your project references only a single logging package. Also, due to breaking changes (rare enough in the packages above), it's possible that wiring doesn't succeed. And to explicitly tell Hangfire what package to use to avoid the ambiguity, you can call one of the following methods (last invocation wins).

```
GlobalConfiguration.Configuration
    .UseSerilogLogProvider()
    .UseNLogLogProvider()
    .UseLog4NetLogProvider()
    .UseEntLibLogProvider()
    .UseLoupeLogProvider()
    .UseElmahLogProvider();
```

If your project doesn't have the required references when calling these methods, you may get a run-time exception.

Of course if you don't have any logging package installed or didn't configure it properly, Hangfire will not log anything, falling back to the internal `NoOpLogger` class. So it's a great time to install one, for example Serilog, as it's the most simple logging package to set up.

Console logger

For simple applications you can use the built-in console log provider, please see the following snippet to learn how to activate it. But please ensure you aren't using it in production environments, because this logger may produce unwanted blocks, since global lock is obtained each time we are writing a message to ensure the colors are correct.

```
GlobalConfiguration.Configuration.UseColouredConsoleLogProvider();
```

Using a custom logger

If your application uses another logging library that's not listed above, you can implement your own logging adapter. Please see the following snippet to learn how to do this – all you need is to implement some interfaces and register the resulting log provider in a global configuration instance.

```csharp
using Hangfire.Logging;

public class CustomLogger : ILog
{
    public string Name { get; set; }

    public bool Log(LogLevel logLevel, Func<string> messageFunc, Exception exception␣
→= null)
    {
        if (messageFunc == null)
        {
            // Before calling a method with an actual message, LogLib first probes
            // whether the corresponding log level is enabled by passing a `null`
            // messageFunc instance.
            return logLevel > LogLevel.Info;
        }

        // Writing a message somewhere, make sure you also include the exception␣
→parameter,
        // because it usually contain valuable information, but it can be `null` for␣
→regular
        // messages.
        Console.WriteLine(String.Format("{0}: {1} {2} {3}", logLevel, Name,␣
→messageFunc(), exception));

        // Telling LibLog the message was successfully logged.
```

```
            return true;
        }
}

public class CustomLogProvider : ILogProvider
{
    public ILog GetLogger(string name)
    {
        // Logger name usually contains the full name of a type that uses it,
        // e.g. "Hangfire.Server.RecurringJobScheduler". It's used to know the
        // context of this or that message and for filtering purposes.
        return new CustomLogger { Name = name };
    }
}
```

After implementing the interfaces above, call the following method:

```
GlobalConfiguration.Configuration.UseLogProvider(new CustomLogProvider());
```

Log level description

There are the following semantics behind each log level. Please take into account that some logging libraries may have slightly other names for these levels, but usually they are almost the same. If you are looking for a good candidate for the minimal log level configuration in your application, choose the `LogLevel.Info`.

Level	Description
Trace	These messages are for debugging Hangfire itself to see what events happened and what conditional branches taken.
Debug	Use this level to know why background processing does not work for you. There are no message count thresholds for this level, so you can use it when something is going wrong. But expect much higher number of messages, comparing to the next levels.
Info	This is the **recommended** minimal level to log from, to ensure everything is working as expected. Processing server is usually using this level to notify about start and stop events – perhaps the most important ones, because inactive server doesn't process anything. Starting from this level, Hangfire tries to log as few messages as possible to not to harm your logging subsystem.
Warn	Background processing may be delayed due to some reason. You can take the corresponding action to minimize the delay, but there will be yet another automatic retry attempt anyway.
Error	Background process or job is unable to perform its work due to some external error which lasts for a long time. Usually a message with this level is logged only after a bunch of retry attempts to ensure you don't get messages on transient errors or network blips. Also usually you don't need to restart the processing server after resolving the cause of these messages, because yet another attempt will be made automatically after some delay.
Fatal	Current processing server will not process background jobs anymore, and manual intervention is required. This log level is almost unused in Hangfire, because there are retries almost everywhere, except in the retry logic itself. Theoretically, ThreadAbortException may cause a fatal error, but only if it's thrown in a bad place – usually thread aborts are being reset automatically. Please also keep in mind that we can't log anything if process is died unexpectedly.

6.3 Background Methods

Background jobs in Hangfire look like regular method calls. Most of its interfaces are using expression trees to define what method should be called and with what arguments. And background jobs can use both instance and static method calls as in the following example.

```
BackgroundJob.Enqueue<IEmailSender>(x => x.Send("hangfire@example.com"));
BackgroundJob.Enqueue(() => Console.WriteLine("Hello, world!"));
```

These lines use *expression trees* – not delegates like `Action` or `Func<T>`. And unlike usual method invocations, they are supposed to be executed asynchronously and even outside of the current process. So the purpose of the method calls above is to collect and serialize the following information.

- Type name, including namespace and assembly.

- Method name and its parameter types.

- Argument values.

Serialization is performed by the Newtonsoft.Json package and resulting JSON, that looks like in the following snippet, is persisted in a storage making it available for other processes. As we can see everything is passed by value, so heavy data structures will also be serialized and consume a lot of bytes in our storage.

```
{"t":"System.Console, mscorlib","m":"WriteLine","p":["System.String"],"a":["Hello,
→world!"]}
```

No other information is preserved

Local variables, instance and static fields and other information isn't available in our background jobs.

6.3.1 Parameters

It is also possible to preserve some context that will be associated with a background job by using *Job Parameters*. This feature is available from *Background Job Filters* and allow us to capture and restore some ambient information. Extension filters use job parameters to store additional details without any intervention to method call metadata.

For example, the CaptureCultureAttribute filter uses job parameters to capture `CurrentCulture` and `CurrentUICulture` when creating a background job and restores it when it is about to be processed.

Anyway, no other context data is preserved

Scopes, globals, `HttpContext` instances and current user IDs aren't preserved automatically.

6.3.2 States

Each background job has a specific state associated with it at every moment in time that defines how and when it will be processed. There is a bunch of built-in states like `Enqueued`, `Scheduled`, `Awaiting`, `Processing`, `Failed`, `Succeeded` and `Deleted`, and custom states can be implemented as well.

During background processing, background jobs are moved from one state into another with executing some side effects. So Hangfire can be considered as a state machine for background jobs. Processed background jobs end in a *final state* (only `Succeeded` and `Deleted` built-in states, but not the `Failed` one) and will be expired automatically after 24 hours by default.

Expiration time can be configured globally in the following way by calling the `WithJobExpirationTimeout` method. But we should ensure to call this method after the `UseXXXStorage` ones, otherwise we'll get a compilation error.

```
GlobalConfiguration.Configuration
    .UseXXXStorage(/* ... */)
    .WithJobExpirationTimeout(TimeSpan.FromHours(6));
```

Calling methods in background

Fire-and-forget method invocation has never been simpler. As you already know from the Quick start guide, you only need to pass a lambda expression with the corresponding method and its arguments:

```
BackgroundJob.Enqueue(() => Console.WriteLine("Hello, world!"));
```

The `Enqueue` method does not call the target method immediately, it runs the following steps instead:

1. Serialize a method information and all its arguments.
2. Create a new background job based on the serialized information.
3. Save background job to a persistent storage.
4. Enqueue background job to its queue.

After these steps were performed, the `BackgroundJob.Enqueue` method immediately returns to a caller. Another Hangfire component, called *Hangfire Server*, checks the persistent storage for enqueued background jobs and performs them in a reliable way.

Enqueued jobs are handled by a dedicated pool of worker threads. The following process is invoked by each worker:

1. Fetch next job and hide it from other workers.
2. Perform the job and all its extension filters.
3. Remove the job from the queue.

So, the job is removed only after processing succeeds. Even if a process was terminated during the performance, Hangfire will perform compensation logic to guarantee the processing of each job.

Each storage has its own implementation for each of these steps and compensation logic mechanisms:

- **SQL Server** implementation uses regular SQL transactions, so in case of a process termination, background job id is placed back on a queue instantly.
- **MSMQ** implementation uses transactional queues, so there is no need for periodic checks. Jobs are fetched almost immediately after enqueueing.
- **Redis** implementation uses blocking `BRPOPLPUSH` command, so jobs are fetched immediately, as with MSMQ. But in case of process termination, they are re-enqueued only after timeout expiration (30 minutes by default).

Calling methods with delay

Sometimes you may want to postpone a method invocation; for example, to send an email to newly registered users a day after their registration. To do this, just call the `BackgroundJob.Schedule` method and pass the desired delay:

```
BackgroundJob.Schedule(
    () => Console.WriteLine("Hello, world"),
    TimeSpan.FromDays(1));
```

Hangfire Server periodically checks the schedule to enqueue scheduled jobs to their queues, allowing workers to execute them. By default, check interval is equal to `15 seconds`, but you can change it by setting the Schedule-PollingInterval property on the options you pass to the `BackgroundJobServer` constructor:

```
var options = new BackgroundJobServerOptions
{
    SchedulePollingInterval = TimeSpan.FromMinutes(1)
};

var server = new BackgroundJobServer(options);
```

If you are processing your jobs inside an ASP.NET application, you should perform the following steps to ensure that your scheduled jobs get executed at the correct time:

- Disable Idle Timeout – set its value to `0`.

- Use the application auto-start feature.

Performing recurrent tasks

Recurring job registration is almost as simple as background job registration – you need to write a single line of code, but you also need to specify an identifier you can use to refer to your job later. The call to `AddOrUpdate` method will create a new recurring job or update existing job with the same identifier.

```
RecurringJob.AddOrUpdate("easyjob", () => Console.Write("Easy!"), Cron.Daily);
```

This line creates a new entry in persistent storage. A special component in Hangfire Server (see *Processing background jobs*) checks the recurring jobs on a minute-based interval and then enqueues them as fire-and-forget jobs. This enables you to track them as usual.

Make sure your app is always running

Your Hangfire Server instance should be always on to perform scheduling and processing logic. If you perform the processing inside an ASP.NET application, please also read the *Making ASP.NET application always running* chapter.

The `Cron` class contains different methods and overloads to run jobs on a minute, hourly, daily, weekly, monthly and yearly basis. You can also use CRON expressions to specify a more complex schedule:

```
RecurringJob.AddOrUpdate("powerfuljob", () => Console.Write("Powerful!"), "0 12 * */2
→");
```

Identifiers should be unique

Use unique identifiers for each recurring job, otherwise you'll end with a single job.

Identifiers may be case sensitive

Recurring job identifier may be **case sensitive** in some storage implementations.

Manipulating recurring jobs

You can remove an existing recurring job by calling the `RemoveIfExists` method. It does not throw an exception when there is no such recurring job.

```
RecurringJob.RemoveIfExists("some-id");
```

To run a recurring job now, call the `Trigger` method. The information about triggered invocation will not be recorded in the recurring job itself, and its next execution time will not be recalculated from this running. For example, if you have a weekly job that runs on Wednesday, and you manually trigger it on Friday it will run on the following Wednesday.

```
RecurringJob.Trigger("some-id");
```

The `RecurringJob` class is a facade for the `RecurringJobManager` class. If you want some more power and responsibility, consider using it:

```
var manager = new RecurringJobManager();
manager.AddOrUpdate("some-id", Job.FromExpression(() => Method()), Cron.Yearly());
```

Passing arguments

You can pass additional data to your background jobs as a regular method arguments. I'll write the following line once again (hope it hasn't bothered you):

```
BackgroundJob.Enqueue(() => Console.WriteLine("Hello, {0}!", "world"));
```

As in a regular method call, these arguments will be available for the `Console.WriteLine` method during the performance of the background job. But since they are marshaled through process boundaries, they are serialized.

The **awesome** Newtonsoft.Json package is used to serialize arguments into JSON strings (since version `1.1.0`). So you can use almost any type as a parameter; including arrays, collections and custom objects. Please see corresponding documentation for more details.

Reference parameters are not supported

You can not pass arguments to parameters by reference – `ref` and `out` keywords are **not supported**.

Since arguments are serialized, consider their values carefully as they can blow up your job storage. Most of the time it is more efficient to store concrete values in an application database and pass their identifiers only to your background jobs.

Remember that background jobs may be processed days or weeks after they were enqueued. If you use data that is subject to change in your arguments, it may become stale – database records may be deleted, the text of an article may be changed, etc. Plan for these changes and design your background jobs accordingly.

Passing dependencies

In almost every job you'll want to use other classes of your application to perform different work and keep your code clean and simple. Let's call these classes as *dependencies*. How to pass these dependencies to methods that will be called in background?

When you are calling static methods in background, you are restricted only to the static context of your application, and this requires you to use the following patterns of obtaining dependencies:

- Manual dependency instantiation through the `new` operator

- Service location

- Abstract factories or builders

- Singletons

However, all of these patterns greatly complicate the unit testability aspect of your application. To fight with this issue, Hangfire allows you to call instance methods in background. Consider you have the following class that uses some kind of `DbContext` to access the database, and `EmailService` to send emails.

```csharp
public class EmailSender
{
    public void Send(int userId, string message)
    {
        var dbContext = new DbContext();
        var emailService = new EmailService();

        // Some processing logic
    }
}
```

To call the `Send` method in background, use the following override of the `Enqueue` method (other methods of `BackgroundJob` class provide such overloads as well):

```csharp
BackgroundJob.Enqueue<EmailSender>(x => x.Send(13, "Hello!"));
```

When a worker determines that it needs to call an instance method, it creates the instance of a given class first using the current `JobActivator` class instance. By default, it uses the `Activator.CreateInstance` method that can create an instance of your class using **its default constructor**, so let's add it:

```csharp
public class EmailSender
{
    private IDbContext _dbContext;
    private IEmailService _emailService;

    public EmailSender()
    {
        _dbContext = new DbContext();
        _emailService = new EmailService();
    }

    // ...
}
```

If you want the class to be ready for unit testing, consider to add constructor overload, because the **default activator can not create instance of class that has no default constructor**:

```csharp
public class EmailSender
{
    // ...

    public EmailSender()
        : this(new DbContext(), new EmailService())
    {
    }

    internal EmailSender(IDbContext dbContext, IEmailService emailService)
```

```
    {
        _dbContext = dbContext;
        _emailService = emailService;
    }
}
```

If you are using IoC containers, such as Autofac, Ninject, SimpleInjector and so on, you can remove the default constructor. To learn how to do this, proceed to the next section.

Using IoC containers

As I said in the *previous section* Hangfire uses the `JobActivator` class to instantiate the target types before invoking instance methods. You can override its behavior to perform more complex logic on a type instantiation. For example, you can tell it to use IoC container that is used in your project:

```
public class ContainerJobActivator : JobActivator
{
    private IContainer _container;

    public ContainerJobActivator(IContainer container)
    {
        _container = container;
    }

    public override object ActivateJob(Type type)
    {
        return _container.Resolve(type);
    }
}
```

Then, you need to register it as a current job activator before starting the Hangfire server:

```
// Somewhere in bootstrap logic, for example in the Global.asax.cs file
var container = new Container();
GlobalConfiguration.Configuration.UseActivator(new ContainerJobActivator(container));
...
app.UseHangfireServer();
```

To simplify the initial installation, there are some integration packages already available on NuGet. Please see the Extensions page on the official site to get the list of all available extension packages: https://www.hangfire.io/extensions. html#ioc-containers.

Some of these activators also provide an extension method for the `GlobalConfiguration` class:

```
GlobalConfiguration.Configuration.UseNinjectActivator(kernel);
```

`HttpContext` is not available

Request information is not available during the instantiation of a target type. If you register your dependencies in a request scope (`InstancePerHttpRequest` in Autofac, `InRequestScope` in Ninject and so on), an exception will be thrown during the job activation process.

So, **the entire dependency graph should be available**. Either register additional services without using the request

scope, or use separate instance of container if your IoC container does not support dependency registrations for multiple scopes.

Using cancellation tokens

Hangfire provides support for cancellation tokens for our background jobs to let them know when a shutdown request was initiated, or job performance was aborted. In the former case the job will be automatically put back to the beginning of its queue, allowing Hangfire to process it after restart.

We should use cancellation tokens as much as possible – they greatly lower the application shutdown time and the risk of the appearance of the `ThreadAbortException`.

CancellationToken

Starting from Hangfire 1.7.0 it's possible to use a regular `CancellationToken` class for this purpose. Unlike the previous `IJobCancellationToken`-based implementation, the new one is fully asynchronous and doesn't lead to immediate storage requests so it's now safe to use it even in tight loops.

```
public async Task LongRunningMethod(CancellationToken token)
{
    for (var i = 0; i < Int32.MaxValue; i++)
    {
        await Task.Delay(TimeSpan.FromSeconds(1), token);
    }
}
```

When creating such a background job, any `CancellationToken` instance can be used, it will be replaced internally just before performing a background job.

```
BackgroundJob.Enqueue<IService>(x => x.LongRunningMethod(CancellationToken.None));
```

Dedicated background process is watching for all the current background jobs that have a cancellation token parameter in a method and polls the storage to watch their current states. When state change is detected or when shutdown is requested, the corresponding cancellation token is canceled.

The polling delay is configurable via the `BackgroundJobServerOptions.CancellationCheckInterval` server option.

```
services.AddHangfireServer(new BackgroundJobServerOptions
{
    CancellationCheckInterval = TimeSpan.FromSeconds(5) // Default value
});
```

IJobCancellationToken

This is the obsolete implementation of cancellation tokens. Although it's implemented asynchronously in the same way as `CancellationToken`-based one, we can't use it in many use cases. Backward compatibility is the only reason for using this interface nowadays, so we should prefer to use `CancellationToken` parameters in the new logic.

The interface contains the `ThrowIfCancellationRequested` method that throws the `OperationCanceledException` when cancellation was requested:

```csharp
public void LongRunningMethod(IJobCancellationToken cancellationToken)
{
    for (var i = 0; i < Int32.MaxValue; i++)
    {
        cancellationToken.ThrowIfCancellationRequested();

        Thread.Sleep(TimeSpan.FromSeconds(1));
    }
}
```

When we want to enqueue such method call as a background job, we can pass the `null` value as an argument for the token parameter, or use the `JobCancellationToken.Null` property to tell code readers that we are doing things right:

```csharp
BackgroundJob.Enqueue(() => LongRunningMethod(JobCancellationToken.Null));
```

The implementation is resolved automatically

Hangfire takes care of passing a proper non-null instance of `IJobCancellationToken` during the job execution at runtime.

Writing unit tests

I will not tell you anything related to unit testing background methods, because Hangfire does not add any specific changes to them (except `IJobCancellationToken` interface parameter). Use your favourite tools and write unit tests for them as usual. This section describes how to test that background jobs were created.

All the code examples use the static `BackgroundJob` class to tell you how to do this or that stuff, because it is simple for demonstrational purposes. But when you want to test a method that invokes static methods, it becomes a pain.

But don't worry – the `BackgroundJob` class is just a facade for the `IBackgroundJobClient` interface and its default implementation – `BackgroundJobClient` class. If you want to write unit tests, use them. For example, consider the following controller that is used to enqueue background jobs:

```csharp
public class HomeController : Controller
{
    private readonly IBackgroundJobClient _jobClient;

    // For ASP.NET MVC
    public HomeController()
        : this(new BackgroundJobClient())
    {
    }

    // For unit tests
    public HomeController(IBackgroundJobClient jobClient)
    {
        _jobClient = jobClient;
    }

    public ActionResult Create(Comment comment)
    {
        ...
```

(continues on next page)

```
        _jobClient.Enqueue(() => CheckForSpam(comment.Id));
        ...
    }
}
```

Simple, yeah. Now you can use any mocking framework, for example, Moq to provide mocks and check the invocations. The `IBackgroundJobClient` interface provides only one method for creating a background job – the `Create` method, that takes a `Job` class instance, that represents the information about the invocation, and a `IState` interface implementation to know the creating job's state.

```csharp
[TestMethod]
public void CheckForSpamJob_ShouldBeEnqueued()
{
    // Arrange
    var client = new Mock<IBackgroundJobClient>();
    var controller = new HomeController(client.Object);
    var comment = CreateComment();

    // Act
    controller.Create(comment);

    // Assert
    client.Verify(x => x.Create(
        It.Is<Job>(job => job.Method.Name == "CheckForSpam" && job.Args[0] == comment.
→Id),
        It.IsAny<EnqueuedState>());
}
```

Note: `job.Method` property points only to background job's method information. If you also want to check a type name, use the `job.Type` property.

Using Batches

Pro Only

This feature is a part of Hangfire Pro package set

Batches allow you to create a bunch of background jobs *atomically*. This means that if there was an exception during the creation of background jobs, none of them will be processed. Consider you want to send 1000 emails to your clients, and they really want to receive these emails. Here is the old way:

```csharp
for (var i = 0; i < 1000; i++)
{
    BackgroundJob.Enqueue(() => SendEmail(i));
    // What to do on exception?
}
```

But what if storage become unavailable on `i == 500`? 500 emails may be already sent, because worker threads will pick up and process jobs once they created. If you re-execute this code, some of your clients may receive annoying duplicates. So if you want to handle this correctly, you should write more code to track what emails were sent.

But here is a much simpler method:

```
BatchJob.StartNew(x =>
{
    for (var i = 0; i < 1000; i++)
    {
        x.Enqueue(() => SendEmail(i));
    }
});
```

In case of exception, you may show an error to a user, and simply ask to retry her action after some minutes. No other code required!

Installation

Batches are available in the Hangfire.Pro package, and you can install it using NuGet Package Manager Console window as usually:

```
PM> Install-Package Hangfire.Pro
```

Batches require to add some additional job filters, some new pages to the Dashboard, and some new navigation menu items. But thanks to the new `GlobalConfiguration` class, it is now as simple as a one method call:

```
GlobalConfiguration.Configuration.UseBatches();
```

Limited storage support

Only official **Hangfire.InMemory**, **Hangfire.SqlServer** and **Hangfire.Pro.Redis** job storage implementations are currently supported. There is nothing special for batches, but some new storage methods should be implemented.

Configuration

The default batch job expiration/retention time if the batch succeeds is 7 days, but you can configure it when calling the `UseBatches` method:

```
GlobalConfiguration.Configuration.UseBatches(TimeSpan.FromDays(2));
```

Chaining Batches

Continuations allow you to chain multiple batches together. They will be executed once *all background jobs* of a parent batch finished. Consider the previous example where you have 1000 emails to send. If you want to make final action after sending, just add a continuation:

```
var id1 = BatchJob.StartNew(/* for (var i = 0; i < 1000... */);
var id2 = BatchJob.ContinueBatchWith(id1, x =>
{
    x.Enqueue(() => MarkCampaignFinished());
    x.Enqueue(() => NotifyAdministrator());
});
```

So batches and batch continuations allow you to define workflows and configure what actions will be executed in parallel. This is very useful for heavy computational methods as they can be distributed to different machines.

Complex Workflows

Create action does not restrict you to create jobs only in *Enqueued* state. You can schedule jobs to execute later, add continuations, add continuations to continuations, etc..

```
var batchId = BatchJob.StartNew(x =>
{
    x.Enqueue(() => Console.Write("1a... "));
    var id1 = x.Schedule(() => Console.Write("1b... "), TimeSpan.FromSeconds(1));
    var id2 = x.ContinueJobWith(id1, () => Console.Write("2... "));
    x.ContinueJobWith(id2, () => Console.Write("3... "));
});

BatchJob.ContinueBatchWith(batchId, x =>
{
    x.Enqueue(() => Console.WriteLine("4..."));
});
```

Nested Batches

Since version 2.0, **batches can consist of other batches**, not only of background jobs. Outer batch is called as *parent*, inner batch is a *child* one (for continuations, it's an *antecedent/continuation* relationship). You can mix both batches and background jobs together in a single batch.

```
BatchJob.StartNew(parent =>
{
    parent.Enqueue(() => Console.WriteLine("First"));
    parent.StartNew(child => child.Enqueue(() => Console.WriteLine("Second")));
});
```

Multiple nesting levels are supported, so each child batch can, in turn, become a parent for another batch, allowing you to create very complex batch hierarchies.

```
BatchJob.StartNew(batch1 =>
{
    batch1.StartNew(batch2 =>
    {
        batch2.StartNew(batch3 => batch3.Enqueue(() => Console.WriteLine("Nested")));
    });
});
```

The whole hierarchy, including parent batch, all of its child batches and background jobs are created in a single transaction. So this feature not only allows you to see a group of related batches on a single dashboard page, but also **create multiple batches atomically**.

```
var antecedentId = BatchJob.StartNew(batch =>
{
    batch.StartNew(inner => inner.Enqueue(() => Console.WriteLine("First")));
    batch.StartNew(inner => inner.Enqueue(() => Console.WriteLine("Second")));
});
```

Parent batch is *succeeded*, if all of its background jobs and batches are *succeeded*. Parent batch is *finished*, if all of its batches and background jobs are in a *final* state. So you can **create continuation for multiple batches**, not just for a single one. Batch continuations also support the nesting feature.

```
BatchJob.ContinueBatchWith(antecedentId, continuation =>
{
    continuation.StartNew(inner => inner.Enqueue(() => Console.WriteLine("First")));
    continuation.StartNew(inner => inner.Enqueue(() => Console.WriteLine("Second")));
});
```

Starting from Hangfire.Pro 2.1.0 it's also possible to use continuations in batches, both standalone and nested ones, for both batches and background jobs.

```
BatchJob.StartNew(parent =>
{
    var nested1 = parent.StartNew(nested =>
    {
        nested.Enqueue(() => Console.WriteLine("Nested 1"));
    });

    var nested2 = parent.ContinueBatchWith(nested1, () => Console.WriteLine("Nested 2
→"));

    var nested3 = parent.ContinueJobWith(nested2, nested =>
    {
        nested.Enqueue(() => Console.WriteLine("Nested 3"));
    });

    string nested5 = null;

    var nested4 = parent.ContinueBatchWith(nested3, nested =>
    {
        nested5 = nested.Enqueue(() => Console.WriteLine("Nested 4"));
    });

    parent.ContinueJobWith(nested5, () => Console.WriteLine("Nested 5"));
});
```

Batch Modification

This is another interesting feature available from version 2.0 that allows you to **modify existing batches** by attaching new background jobs and child batches to them. You can add background jobs in any states, as well as nested batches. If a modified batch has already been finished, it will be moved back to the *started* state.

```
var batchId = BatchJob.StartNew(batch => batch.Enqueue(() => Console.WriteLine("First
→")));
BatchJob.Attach(batchId, batch => batch.Enqueue(() => Console.WriteLine("Second")));
```

This feature helps, if you want a list of records you want to process in parallel, and then execute a continuation. Previously you had to generate a very long chain of continuations, and it was very hard to debug them. So you can create the structure, and modify a batch later.

```
var batchId = BatchJob.StartNew(batch => batch.Enqueue(() => ProcessHugeList(batch.Id,
→ ListId)));
BatchJob.ContinueBatchWith(batchId, batch => batch.Enqueue(() =>
→SendNotification(ListId)));
```

```
// ProcessHugeList
BatchJob.Attach(batchId, batch =>
```

```
{
    foreach (var record in records)
    {
        batch.Enqueue(() => ProcessRecord(ListId, record.Id)));
    }
});
```

Batches can be created without any background jobs. Initially such an empty batches are considered as *completed*, and once some background jobs or child batches are added, they move a batch to the *started* state (or to another, depending on their state).

```
var batchId = BatchJob.StartNew(batch => {});
BatchJob.Attach(batchId, batch => batch.Enqueue(() => Console.WriteLine("Hello, world!
↪")));
```

More Continuations

Since version 2.0 it's possible to **continue batch with a regular background job** without creating a batch that consists only of a single background job. Unfortunately we can't add extension methods for static classes, so let's create a client first.

```
var backgroundJob = new BackgroundJobClient();
var batchId = BatchJob.StartNew(/* ... */);

backgroundJob.ContinueBatchWith(batchId, () => Console.WriteLine("Continuation"));
```

You can use the new feature in other way, and create **batch continuations for regular background jobs**. So you are free to define workflows, where synchronous actions are continued by a group of parallel work, and then continue back to a synchronous method.

```
var jobId = BackgroundJob.Enqueue(() => Console.WriteLine("Antecedent"));
BatchJob.ContinueJobWith(jobId, batch => batch.Enqueue(() => Console.WriteLine(
↪"Continuation")));
```

Cancellation of a Batch

If you want to stop a batch with millions of background jobs from being executed, not a problem, you can call the *Cancel* method, or click the corresponding button in dashboard.

```
var batchId = BatchJob.StartNew(/* a lot of jobs */);
BatchJob.Cancel(batchId);
```

This method **does not** iterate through all the jobs, it simply sets a property of a batch. When a background job is about to execute, job filter checks for a batch status, and move a job to the *Deleted* state, if a batch has cancelled.

6.4 Background Processing

6.4.1 Processing background jobs

Hangfire Server part is responsible for background job processing. The Server does not depend on ASP.NET and can be started anywhere, from a console application to Microsoft Azure Worker Role. Single API for all applications is exposed through the `BackgroundJobServer` class:

```
// Create an instance of Hangfire Server and start it.
// Please look at ctor overrides for advanced options like
// explicit job storage instance.
var server = new BackgroundJobServer();

// Wait for graceful server shutdown.
server.Dispose();
```

Always dispose your background server

Call the `Dispose` method whenever possible to have graceful shutdown features working.

Hangfire Server consists of different components that are doing different work: workers listen to queue and process jobs, recurring scheduler enqueues recurring jobs, schedule poller enqueues delayed jobs, expire manager removes obsolete jobs and keeps the storage as clean as possible, etc.

You can turn off the processing

If you don't want to process background jobs in a specific application instance, just don't create an instance of the `BackgroundJobServer` class.

The `Dispose` method is a **blocking** one, it waits until all the components prepare for shutdown (for example, workers will place back interrupted jobs to their queues). So, we can talk about graceful shutdown only after waiting for all the components.

Strictly saying, you aren't required to invoke the `Dispose` method. Hangfire can handle even unexpected process terminations, and will retry interrupted jobs automatically. However it is better to control the exit points in your methods by using *cancellation tokens*.

6.4.2 Processing jobs in a web application

Ability to process background jobs directly in web applications is a primary goal of Hangfire. No external application like Windows Service or console application is required for running background jobs, however you will be able to change your decision later if you really need it. So, you can postpone architecture decisions that complicate things.

Since Hangfire does not have any specific dependencies and does not depend on `System.Web`, it can be used together with any web framework for .NET:

- ASP.NET WebForms
- ASP.NET MVC
- ASP.NET WebApi
- ASP.NET vNext (through the `app.UseOwin` method)
- Other OWIN-based web frameworks (Nancy, Simple.Web)

- Other non-OWIN based web frameworks (ServiceStack)

Using `BackgroundJobServer` class

The basic way (but not the simplest – see the next section) to start using Hangfire in a web framework is to use host-agnostic `BackgroundJobServer` class that was described in the *previous chapter* and call its `Start` and `Dispose` method in corresponding places.

Dispose the server instance when possible

In some web application frameworks it may be unclear when to call the `Dispose` method. If it is really impossible, you can omit this call as *described here* (but you'll lose the *graceful shutdown* feature).

For example, in ASP.NET applications the best place for start/dispose method invocations is the `global.asax.cs` file:

```csharp
using System;
using System.Web;
using Hangfire;

namespace WebApplication1
{
    public class Global : HttpApplication
    {
        private BackgroundJobServer _backgroundJobServer;

        protected void Application_Start(object sender, EventArgs e)
        {
            GlobalConfiguration.Configuration
                .UseSqlServerStorage("DbConnection");

            _backgroundJobServer = new BackgroundJobServer();
        }

        protected void Application_End(object sender, EventArgs e)
        {
            _backgroundJobServer.Dispose();
        }
    }
}
```

Using OWIN extension methods

Hangfire also provides a dashboard that is implemented on top of OWIN pipeline to process requests. If you have simple set-up and want to keep Hangfire initialization logic in one place, consider using Hangfire's extension methods for OWIN's `IAppBuilder` interface:

Install `Microsoft.Owin.Host.SystemWeb` for ASP.NET + IIS

If you are using OWIN extension methods for ASP.NET application hosted in IIS, ensure you have `Microsoft.Owin.Host.SystemWeb` package installed. Otherwise some features like *graceful shutdown* feature will not work for you.

If you installed Hangfire through the `Hangfire` package, this dependency is already installed.

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.UseHangfireServer();
    }
}
```

This line creates a new instance of the `BackgroundJobServer` class automatically, calls the `Start` method and registers method `Dispose` invocation on application shutdown. The latter is implemented using a `CancellationToken` instance stored in the `host.OnAppDisposing` environment key.

6.4.3 Processing jobs in a console application

To start using Hangfire in a console application, you'll need to install Hangfire packages to your console application first. So, use your Package Manager Console window to install it:

```
PM> Install-Package Hangfire.Core
```

Then, install the needed package for your job storage. For example, you need to execute the following command to use SQL Server:

```
PM> Install-Package Hangfire.SqlServer
```

`Hangfire.Core` package is enough

Please don't install the `Hangfire` package for console applications as it is a quick-start package only and contain dependencies you may not need (for example, `Microsoft.Owin.Host.SystemWeb`).

After installing packages, all you need is to create a new *Hangfire Server* instance and start it as written in the *previous* chapter. However, there are some details here:

- Since the `Start` method is **non-blocking**, we insert a `Console.ReadKey` call to prevent instant shutdown of an application.

- The call to `Stop` method is implicit – it is made through the `using` statement.

```
using System;
using Hangfire;
using Hangfire.SqlServer;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main()
        {
            GlobalConfiguration.Configuration.UseSqlServerStorage("connection_string
↪");

            using (var server = new BackgroundJobServer())
            {
```

```
                Console.WriteLine("Hangfire Server started. Press any key to exit...
↪");

                Console.ReadKey();
            }
        }
    }
}
```

6.4.4 Processing jobs in a Windows Service

To start using Hangfire in a Windows Service, you'll need to install Hangfire packages to your application first. So, use your Package Manager Console window to install it:

```
PM> Install-Package Hangfire.Core
```

Then, install the needed package for your job storage. For example, you need to execute the following command to use SQL Server:

```
PM> Install-Package Hangfire.SqlServer
```

`Hangfire.Core` package is enough

Please don't install the `Hangfire` package for console applications as it is a quick-start package only and contain dependencies you may not need (for example, `Microsoft.Owin.Host.SystemWeb`).

After installing packages, all you need is to create a new *Hangfire Server* instance and start it as written in the *Processing background jobs* chapter. So, open the source code of the file that describes the service and modify it as written below.

```csharp
using System.ServiceProcess;
using Hangfire;
using Hangfire.SqlServer;

namespace WindowsService1
{
    public partial class Service1 : ServiceBase
    {
        private BackgroundJobServer _server;

        public Service1()
        {
            InitializeComponent();

            GlobalConfiguration.Configuration.UseSqlServerStorage("connection_string
↪");
        }

        protected override void OnStart(string[] args)
        {
            _server = new BackgroundJobServer();
        }

        protected override void OnStop()
```

```
        {
            _server.Dispose();
        }
    }
}
```

If you are new to Windows Services in .NET projects, it is always better to google about them first, but for quick-start scenario you'll need only to add an installer and optionally configure it. To perform this step just go back to the design view of the service class, right click on it and choose the `Add Installer` menu item.
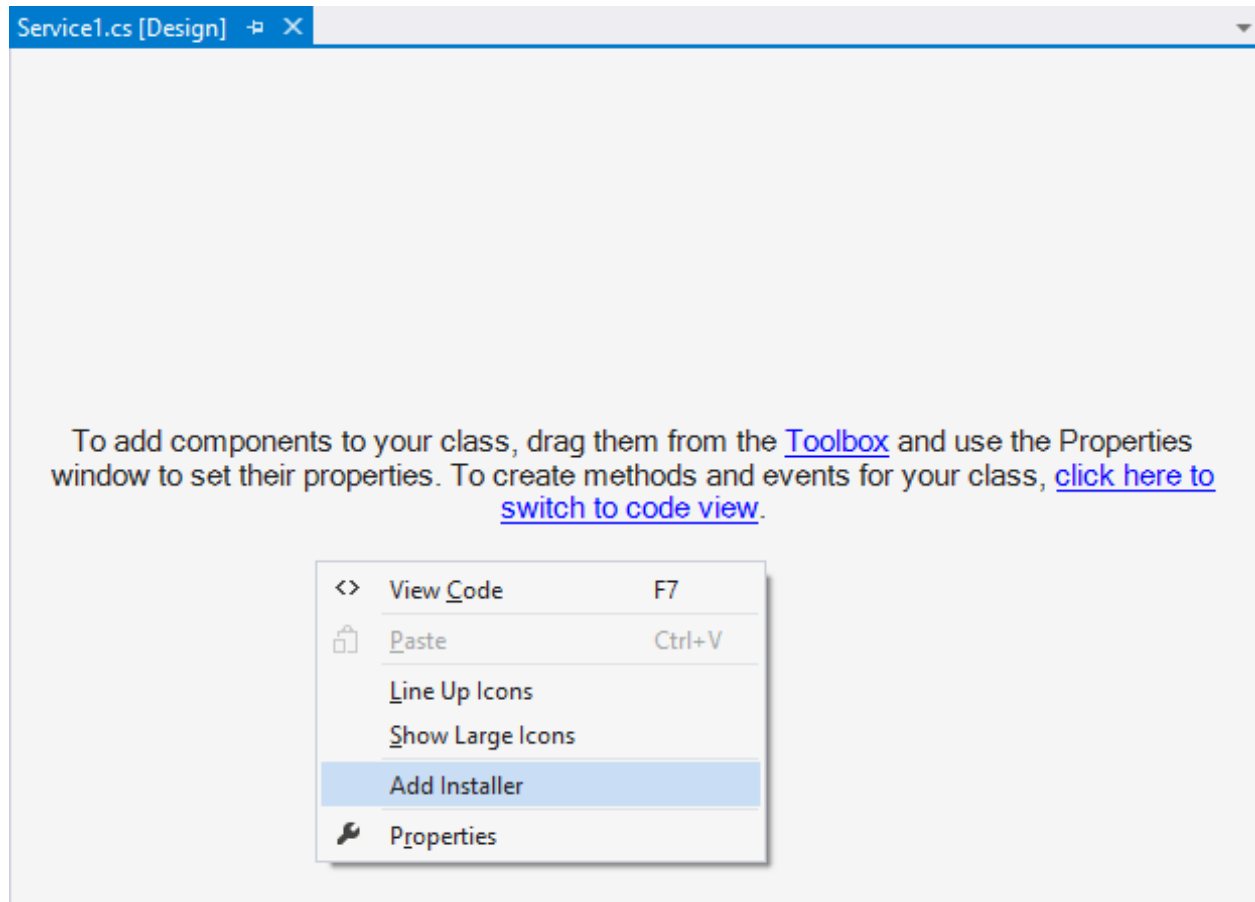
Then build your project, install your Windows Service and run it. If it fails, try look at your Windows Event Viewer for recent exceptions.

```
installutil <yourproject>.exe
```

6.4.5 Dealing with exceptions

Bad things happen. Any method can throw different types of exceptions. These exceptions can be caused either by programming errors that require you to re-deploy the application, or transient errors, that can be fixed without additional deployment.

Hangfire handles all exceptions occurred both in internal (belonging to Hangfire itself), and external methods (jobs, filters and so on), so it will not bring down the whole application. All internal exceptions are logged (so, don't forget

to *enable logging*) and the worst case they can lead – background processing will be stopped after `10` retry attempts with increasing delay modifier.

When Hangfire encounters external exception that occurred during the job performance, it will automatically *try* to change its state to the `Failed` one, and you always can find this job in the Monitor UI (it will not be expired unless you delete it explicitly).

In the previous paragraph I said that Hangfire *will try* to change its state to failed, because state transition is one of places, where *job filters* can intercept and change the initial pipeline. And the `AutomaticRetryAttribute` class is one of them, that schedules the failed job to be automatically retried after increasing delay.

This filter is applied globally to all methods and have 10 retry attempts by default. So, your methods will be retried in case of exception automatically, and you receive warning log messages on every failed attempt. If retry attempts exceeded their maximum, the job will be move to the `Failed` state (with an error log message), and you will be able to retry it manually.

If you don't want a job to be retried, place an explicit attribute with 0 maximum retry attempts value:

```
[AutomaticRetry(Attempts = 0)]
public void BackgroundMethod()
{
}
```

Use the same way to limit the number of attempts to the different value. If you want to change the default global value, add a new global filter:

```
GlobalJobFilters.Filters.Add(new AutomaticRetryAttribute { Attempts = 5 });
```

If you are using ASP.NET Core you can use the IServiceCollection extension method AddHangfire. Note that AddHangfire uses the GlobalJobFilter instance and therefore dependencies should be Transient or Singleton.

```
services.AddHangfire((provider, configuration) =>
{
    configuration.UseFilter(provider.GetRequiredService<AutomaticRetryAttribute>());
}
```

6.4.6 Tracking the progress

There are two ways to implement this task: polling and pushing. Polling is easier to understand, but server push is a more comfortable way, because it helps you to avoid unnecessary calls to server. Plus, SignalR greatly simplifies the

latter task.

I'll show you a simple example, where client only needs to check for a job completion. You can see the full sample in Hangfire.Highlighter project.

Highlighter has the following background job that calls an external web service to highlight code snippets:

```
public void Highlight(int snippetId)
{
    var snippet = _dbContext.CodeSnippets.Find(snippetId);
    if (snippet == null) return;

    snippet.HighlightedCode = HighlightSource(snippet.SourceCode);
    snippet.HighlightedAt = DateTime.UtcNow;

    _dbContext.SaveChanges();
}
```

Polling for a job status

When can we say that this job is incomplete? When the `HighlightedCode` property value *is null*. When can we say it was completed? When the specified property *has value* – this example is simple enough.

So, when we are rendering the code snippet that is not highlighted yet, we need to render a JavaScript that makes ajax calls with some interval to some controller action that returns the job status (completed or not) until the job was finished.

```
public ActionResult CheckHighlighted(int snippetId)
{
    var snippet = _db.Snippets.Find(snippetId);

    return snippet.HighlightedCode == null
        ? new HttpStatusCodeResult(HttpStatusCode.NoContent)
        : Content(snippet.HighlightedCode);
}
```

When code snippet becomes highlighted, we can stop the polling and show the highlighted code. But if you want to track progress of your job, you need to perform extra steps:

- Add a column `Status` to the snippets table.
- Update this column during background work.
- Check this column in polling action.

But there is a better way.

Using server push with SignalR

Why do we need to poll our server? It can say when the snippet becomes highlighted itself. And SignalR, an awesome library to perform server push, will help us. If you don't know about this library, look at it, and you'll love it. Really.

I don't want to include all the code snippets here (you can look at the sources of this sample). I'll show you only the two changes that you need, and they are incredibly simple.

First, you need to add a hub:

```csharp
public class SnippetHub : Hub
{
    public async Task Subscribe(int snippetId)
    {
        await Groups.Add(Context.ConnectionId, GetGroup(snippetId));

        // When a user subscribes a snippet that was already
        // highlighted, we need to send it immediately, because
        // otherwise she will listen for it infinitely.
        using (var db = new HighlighterDbContext())
        {
            var snippet = await db.CodeSnippets
                .Where(x => x.Id == snippetId && x.HighlightedCode != null)
                .SingleOrDefaultAsync();

            if (snippet != null)
            {
                Clients.Client(Context.ConnectionId)
                    .highlight(snippet.Id, snippet.HighlightedCode);
            }
        }
    }

    public static string GetGroup(int snippetId)
    {
        return "snippet:" + snippetId;
    }
}
```

And second, you need to make a small change to your background job method:

```csharp
public void HighlightSnippet(int snippetId)
{
    ...
    _dbContext.SaveChanges();

    var hubContext = GlobalHost.ConnectionManager
        .GetHubContext<SnippetHub>();

    hubContext.Clients.Group(SnippetHub.GetGroup(snippet.Id))
        .highlight(snippet.HighlightedCode);
}
```

And that's all! When user opens a page that contains unhighlighted code snippet, his browser connects to the server, subscribes for code snippet notification and waits for update notifications. When background job is about to be done, it sends the highlighted code to all subscribed users.

If you want to add progress tracking, just add it. No additional tables and columns required, only JavaScript function. This is an example of real and reliable asynchrony for ASP.NET applications without taking much effort to it.

6.4.7 Configuring the degree of parallelism

Background jobs are processed by a dedicated pool of worker threads that run inside Hangfire Server subsystem. When you start the background job server, it initializes the pool and starts the fixed amount of workers. You can specify their number by passing the value to the `UseHangfireServer` method.

```
var options = new BackgroundJobServerOptions { WorkerCount = Environment.
→ProcessorCount * 5 };
app.UseHangfireServer(options);
```

If you use Hangfire inside a Windows service or console app, just do the following:

```
var options = new BackgroundJobServerOptions
{
    // This is the default value
    WorkerCount = Environment.ProcessorCount * 5
};

var server = new BackgroundJobServer(options);
```

Worker pool uses dedicated threads to process jobs separately from requests to let you to process either CPU intensive or I/O intensive tasks as well and configure the degree of parallelism manually.

6.4.8 Placing processing into another process

You may decide to move the processing to the different process from the main application. For example, your web application will only enqueue background jobs, leaving their performance to a Console application or Windows Service. First of all, let's overview the reasons for such decision.

Well scenarios

- Your background processing consumes **too much CPU or other resources**, and this decreases main application's performance. So you want to use separate machine for processing background jobs.

- You have long-running jobs that **are constantly aborted** (retrying, aborted, retried again and so on) due to regular shutdowns of the main application. So you want to use separate process with increased lifetime (and you can't use *always running mode* for your web application).

- *Do you have other suggestions? Please post them in the comment form below*.

You can stop processing background jobs in your main application by simply removing the instantiation of the BackgroundJobServer class (if you create it manually) or removing an invocation of the UseHangfireServer method from your OWIN configuration class.

After accomplishing the first step, you need to enable processing in another process, here are some guides:

- *Using Console applications*
- *Using Windows Services*

Same storage requires the same code base

Ensure that all of your Client/Servers use **the same job storage** and **have the same code base**. If client enqueues a job based on the SomeClass that is absent in server's code, the latter will simply throw a performance exception.

If this is a problem, your client may have references only to interfaces, whereas server provide implementations (please see the *Using IoC containers* chapter).

Doubtful scenarios

- You don't want to consume additional Thread Pool threads with background processing – Hangfire Server uses **custom, separate and limited thread pool**.

- You are using Web Farm or Web Garden and don't want to face with synchronization issues – Hangfire Server is **Web Garden/Web Farm friendly** by default.

6.4.9 Running multiple server instances

Obsolete since 1.5

You aren't required to have additional configuration to support multiple background processing servers in the same process since Hangfire 1.5, just skip the article. Server identifiers are now generated using GUIDs, so all the instance names are unique.

It is possible to run multiple server instances inside a process, machine, or on several machines at the same time. Each server use distributed locks to perform the coordination logic.

Each Hangfire Server has a unique identifier that consists of two parts to provide default values for the cases written above. The last part is a process id to handle multiple servers on the same machine. The former part is the *server name*, that defaults to a machine name, to handle uniqueness for different machines. Examples: `server1:9853`, `server1:4531`, `server2:6742`.

Since the defaults values provide uniqueness only on a process level, you should handle it manually if you want to run different server instances inside the same process:

```
var options = new BackgroundJobServerOptions
{
    ServerName = String.Format(
        "{0}.{1}",
        Environment.MachineName,
        Guid.NewGuid().ToString())
};

var server = new BackgroundJobServer(options);

// or

app.UseHangfireServer(options);
```

6.4.10 Configuring Job Queues

Hangfire can process multiple queues. If you want to prioritize your jobs, or split the processing across your servers (some processes for the archive queue, others for the images queue, etc), you can tell Hangfire about your decisions.

To place a job into a different queue, use the `QueueAttribute` class on your method:

```
[Queue("alpha")]
public void SomeMethod() { }

BackgroundJob.Enqueue(() => SomeMethod());
```

Queue name argument formatting

The Queue name argument must consist of lowercase letters, digits, underscore, and dash (since 1.7.6) characters only.

To begin processing multiple queues, you need to update your `BackgroundJobServer` configuration.

```csharp
var options = new BackgroundJobServerOptions
{
    Queues = new[] { "alpha", "beta", "default" }
};

app.UseHangfireServer(options);
// or
using (new BackgroundJobServer(options)) { /* ... */ }
```

Processing order

Queues are run in the order that depends on the concrete storage implementation. For example, when we are using *Hangfire.SqlServer* the order is defined by alphanumeric order and array index is ignored. When using *Hangfire.Pro.Redis* package, array index is important and queues with a lower index will be processed first.

The example above shows a generic approach, where workers will fetch jobs from the `alpha` queue first, `beta` second, and then from the `default` queue, regardless of an implementation.

ASP.NET Core

For ASP.NET Core, define the queues array with `services.AddHangfireServer` in `Startup.cs`:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // Add the processing server as IHostedService
    services.AddHangfireServer(options =>
    {
        options.Queues = new[] { "alpha", "beta", "default" };
    });
}
```

6.4.11 Concurrency & Rate Limiting

Note: Hangfire.Throttling package is a part of Hangfire.Ace extensibility set and available on the private NuGet feed.

Hangfire.Throttling package contains advanced types and methods to apply concurrency and rate limits directly to our background jobs without touching any logic related to queues, workers, servers or using additional services. So we can control how many particular background jobs are running at the same point of time or within a specific time window.

Throttling is performed asynchronously by rescheduling jobs to a later time or deleting them when throttling condition is met, depending on the configured behavior. And while throttled jobs are waiting for their turn, our workers are free to process other enqueued background jobs.

The primary focus of this package is to provide a simpler way of reducing the load on external resources. Databases or third-party services affected by background jobs may suffer from additional concurrency, causing increased latency

and error rates. While standard solution to use different queues with a constrained number of workers works well enough, it requires additional infrastructure planning and may lead to underutilization. And throttling primitives are much easier to use for this purpose.

Everything works on a best-effort basis

While it can be possible to use this package to enforce proper synchronization and concurrency control over background jobs, it's very hard to achieve it due to the complexity of distributed processing. There are a lot of things to consider, including appropriate storage configuration, and a single mistake will ruin everything.

Throttlers apply only to different background jobs, and there's no reliable way to prevent multiple executions of the same background job other than by using transactions in background job method itself. `DisableConcurrentExecution` may help a bit by narrowing the safety violation surface, but it heavily relies on an active connection, which may be broken (and lock is released) without any notification for our background job.

Hangfire.Throttling provides the following primitives, all of them are implemented as regular state changing filters that run when a worker is starting or completing a background job. They form two groups, depending on their acquire and release behavior.

Concurrency Limiters

- *Mutexes* – allow only a single background job to be running concurrently.

- *Semaphores* – limit how many background jobs are allowed to run concurrently.

Rate Limiters

- *Fixed Window Counters* – limit how many job executions are allowed within a given fixed time interval.

- *Sliding Window Counters* – limit how many job executions are allowed to run within any specific time interval.

- *Dynamic Window Counters* – allow to create window counters dynamically depending on job arguments.

Requirements

Supported only for *Hangfire.SqlServer* (better to use 1.7) and *Hangfire.Pro.Redis* (recommended 2.4.0) storages. Community storage support will be denoted later after defining correctness conditions for storages.

Installation

The package is available on a private Hangfire.Ace NuGet feed (that's different from Hangfire.Pro one), please see the Downloads page to learn how to use it. After registering the private feed, we can install the `Hangfire.Throttling` package by editing our `.csproj` file for new project types:

```
<PackageReference Include="Hangfire.Throttling" Version="1.0.*" />
```

Alternatively we can use Package Manager Console window to install it using the `Install-Package` command as shown below.

```
Install-Package Hangfire.Throttling
```

Configuration

The only configuration method required for throttlers is the `IGlobalConfiguration.UseThrottling` extension method. If we don't call this method, every background job decorated with any throttling filter will be eventually moved to the failed state.

```
GlobalConfiguration.Configuration
    .UseXXXStorage()
    .UseThrottling();
```

The `UseThrottling` method will register all the required filters to make throttling working and add new pages to the Dashboard UI. We can also configure default throttling action to tell the library whether to retry or delete a background job when it's throttled, and specify minimal retry delay (should be greater or equal to 15 seconds) useful for *Concurrency Limiters*.

```
GlobalConfiguration.Configuration
    .UseXXXStorage()
    .UseThrottling(ThrottlingAction.RetryJob, TimeSpan.FromMinutes(1));
```

When using custom `IJobFilterProvider` instance that's resolved via some kind of IoC container, we can use another available overload of the `UseThrottling` method as shown below. It is especially useful for ASP.NET Core applications that's heavy driven by built-in dependency injection.

```
GlobalConfiguration.Configuration
    .UseXXXStorage()
    .UseThrottling(provider.Resolve<IJobFilterProvider>, ThrottlingAction.RetryJob,␣
→TimeSpan.FromMinutes(1));
```

Usage

Most of the throttling primitives are required to be created first using the `IThrottlingManager` interface. Before creating, we should pick a unique *Resource Identifier* we can use later to associate particular background jobs with this or that throttler instance.

Resource Identifier a generic string of maximum 100 characters, just a reference we need to pick to allow Hangfire to know where to get the primitive's metadata. Resource Identifiers are isolated between different primitive types, but it's better not to use same identifiers to not to confuse anyone.

In the following example, a semaphore is created with the `orders` identifier and a limit of 20 concurrent background jobs. Please see later sections to learn how to create other throttling primitives. We'll use this semaphore after a while.

```
using Hangfire.Throttling;

IThrottlingManager manager = new ThrottlingManager();
manager.AddOrUpdateSemaphore("orders", new SemaphoreOptions(limit: 20));
```

Adding Attributes

Throttlers are regular background job filters and can be applied to a particular job by using corresponding attributes as shown in the following example. After adding these attributes, state changing pipeline will be modified for all the methods of the defined interface.

```
using Hangfire.Throttling;

[Semaphore("orders")]
public interface IOrderProcessingJobsV1
{
    int CreateOrder();

    [Mutex("orders:{0}")]
    void ProcessOrder(long orderId);

    [Throttling(ThrottlingAction.DeleteJob, minimumDelayInSeconds: 120)]
    void CancelOrder(long orderId);
}
```

Throttling

Throttling happens when throttling condition of one of the applied throttlers wasn't satisfied. It can be configured either globally or locally, and default throttling action is to schedule background job to run *one minute* (also can be configured) later. After acquiring a throttler, it's not released until job is moved to a final state to prevent part effects.

Before processing the `CreateOrder` method in the example above, a worker will attempt to acquire a semaphore first. On successful acquisition, background job will be processed immediately. But if the acquisition fails, background job is throttled. Default throttling action is `RetryJob`, so it will be moved to the `ScheduledState` with default delay of 1 minute.

For the `ProcessOrder` method, worker will attempt to acquire *both* semaphore and mutex. So if the acquisition of a mutex or semaphore, or both of them fails, background job will be throttled and retried, releasing the worker.

And for the `CancelOrder` method, default throttling action is changed to the `DeleteJob` value. So when semaphore can't be acquired for that job, it will be deleted instead of rescheduled.

Removing Attributes

It's better not to remove the throttling attributes directly when deciding to remove the limits on the particular method, especially for *Concurrency Limiters*, because some of them may not be released properly. Instead, set the `Mode` property to the `ThrottlerMode.Release` value (default is `ThrottlerMode.AcquireAndRelease`) of a corresponding limiter first.

```
using Hangfire.Throttling;

[Semaphore("orders")]
public interface IOrderProcessingJobsV1
{
    [Mutex("orders:{0}", Mode = ThrottlerMode.Release)]
    Task ProcessOrderAsync(long orderId);

    // ...
}
```

In this mode, throttlers will not be applied anymore, only released. So when all the background jobs processed and corresponding limiters were already released, we can safely remove the attribute. *Rate Limiters* don't run anything on the Release stage and are expired automatically, so we don't need to change the mode before their removal.

Strict Mode

Since the primary focus of the library is to reduce pressure on other services, throttlers are released by default when background jobs move out of the *Processing* state. So when you retry or reschedule a running background job, any *Mutexes* or *Semaphores* will be released immediately and let other jobs to acquire them. This mode is called *Relaxed*.

Alternatively you can use *Strict Mode* to release throttlers only when background job was fully completed, e.g. moved to a final state (such as *Succeeded* or *Deleted*, but not the *Failed* one). This is useful when your background job produces multiple side effects, and you don't want to let other background jobs to examine partial effects.

You can turn on the *Strict Mode* by applying the `ThrottlingAttribute` on a method and using its `StrictMode` property as shown below. When multiple throttlers are defined, Strict Mode is applied to all of them. Please note it affects only *Concurrency Limiters* and don't affect *Rate Limiters*, since they don't invoke anything when released.

```
[Mutex("orders:{0}")]
[Throttling(StrictMode = true)]
Task ProcessOrderAsync(long orderId);
```

In either mode, throttler's release and background job's state transition performed in the same transaction.

Concurrency Limiters

Mutexes

Mutex prevents concurrent execution of *multiple* background jobs that share the same resource identifier. Unlike other primitives, they are created dynamically so we don't need to use `IThrottlingManager` to create them first. All we need is to decorate our background job methods with the `MutexAttribute` filter and define what resource identifier should be used.

```
[Mutex("my-resource")]
public void MyMethod()
{
    // ...
}
```

When we create multiple background jobs based on this method, they will be executed one after another on a best-effort basis with the limitations described below. If there's a background job protected by a mutex currently executing, other executions will be throttled (rescheduled by default a minute later), allowing a worker to process other jobs without waiting.

Mutex doesn't prevent simultaneous execution of the same background job

As there are no reliable automatic failure detectors in distributed systems, it is possible that the same job is being processed on different workers in some corner cases. Unlike OS-based mutexes, mutexes in this package don't protect from this behavior so develop accordingly.

`DisableConcurrentExecution` filter may reduce the probability of violation of this safety property, but the only way to guarantee it is to use transactions or CAS-based operations in our background jobs to make them idempotent.

If a background job protected by a mutex is unexpectedly terminated, it will simply re-enter the mutex. It will be held until background job is moved to the final state (Succeeded, Deleted, but not Failed).

We can also create multiple background job methods that share the same resource identifier, and mutual exclusive behavior will span all of them, regardless of the method name.

6.4. Background Processing 65

```
[Mutex("my-resource")]
public void FirstMethod() { /* ... */ }

[Mutex("my-resource")]
public void SecondMethod() { /* ... */ }
```

Since mutexes are created dynamically, it's possible to use a dynamic resource identifier based on background job arguments. To define it, we should use String.Format-like templates, and during invocation all the placeholders will be replaced with actual job arguments. But ensure everything is lower-cased and contains only alphanumeric characters with limited punctuation – no rules except maximum length and case insensitivity is enforced, but it's better to keep identifiers simple.

Maximal length of resource identifiers is 100 characters

Please keep this in mind especially when using dynamic resource identifiers.

```
[Mutex("orders:{0}")]
public void ProcessOrder(long orderId) { /* ... */ }

[Mutex("newsletters:{0}:{1}")]
public void ProcessNewsletter(int tenantId, long newsletterId) { /* ... */ }
```

Throttling Batches

By default background job id is used to identify the current owner of a particular mutex. But since version 1.3 it is possible to use any custom value from a given job parameter. With this feature we can throttle entire batches, since we can pass the `BatchId` job parameter that's used to store the batch identifier. To accomplish this, we need to create two empty methods with `ThrottlerMode.Acquire` and `ThrottlerMode.Release` semantics that will acquire and release a mutex:

```
[Mutex("orders-api", Mode = ThrottlerMode.Acquire, ParameterName = "BatchId")]
public static void StartBatch() { /* Doesn't do anything */ }

[Mutex("orders-api", Mode = ThrottlerMode.Release, ParameterName = "BatchId")]
public static void CompleteBatch() { /* Doesn't do anything */ }
```

And then create a batch as a chain of continuations, starting with the `StartBatch` method and ending with the `CompleteBatch` method. Please note that the last method is created with the `BatchContinuationOptions.OnAnyFinishedState` option to release the throttler even if some of our background jobs completed non-successfully (deleted, for example).

```
BatchJob.StartNew(batch =>
{
    var startId = batch.Enqueue(() => StartBatch());
    var bodyId = batch.ContinueJobWith(startId, nestedBatch =>
    {
        for (var i = 0; i < 5; i++) nestedBatch.Enqueue(() => Thread.Sleep(5000));
    });

    batch.ContinueBatchWith(
        bodyId,
        () => CompleteBatch(),
```

```
        options: BatchContinuationOptions.OnAnyFinishedState);
});
```

In this case batch identifier will be used as an owner, and entire batch will be protected by a mutex, preventing other batches from running simultaneously.

Semaphores

Semaphore limits concurrent execution of multiple background jobs to a certain maximum number. Unlike mutexes, semaphores should be created first using the `IThrottlingManager` interface with the maximum number of concurrent background jobs allowed. The `AddOrUpdateSemaphore` method is idempotent, so we can safely place it in the application initialization logic.

```
IThrottlingManager manager = new ThrottlingManager();
manager.AddOrUpdateSemaphore("newsletter", new SemaphoreOptions(maxCount: 100));
```

We can also call this method on an already existing semaphore, and in this case the maximum number of jobs will be updated. If background jobs that use this semaphore are currently executing, there may be temporary violation that will eventually be fixed. So if the number of background jobs is higher than the new `maxCount` value, no exception will be thrown, but new background jobs will be unable to acquire a semaphore. And when all of those background jobs finished, `maxCount` value will be satisfied.

We should place the `SemaphoreAttribute` filter on a background job method and provide a correct resource identifier to link it with an existing semaphore. If semaphore with the given resource identifier doesn't exist or was removed, an exception will be thrown at run-time, and background job will be moved to the Failed state.

```
[Semaphore("newsletter")]
public void SendNewsletter() { /* ... */ }
```

Multiple executions of the same background job count as 1

As with mutexes, multiple invocations of the same background job aren't respected and counted as 1. So actually it's possible that more than the given count of background job methods are running concurrently. As before, we can use `DisableConcurrentExecution` to reduce the probability of this event, but we should be prepared for this anyway.

As with mutexes, we can apply the `SemaphoreAttribute` with the same resource identifier to multiple background job methods, and all of them will respect the behavior of a given semaphore. However dynamic resource identifiers based on arguments aren't allowed for semaphores as they are required to be created first.

```
[Semaphore("newsletter")]
public void SendMonthlyNewsletter() { /* ... */ }

[Semaphore("newsletter")]
public void SendDailyNewsletter() { /* ... */ }
```

Unused semaphore can be removed in the following way. Please note that if there are any associated background jobs are still running, an `InvalidOperationException` will be thrown (see *Removing Attributes* to avoid this scenario). This method is idempotent, and will simply succeed without performing anything when the corresponding semaphore doesn't exist.

```
manager.RemoveSemaphoreIfExists("newsletter");
```

Rate Limiters

Fixed Window Counters

Fixed window counters limit the number of *background job executions* allowed to run in a specific fixed time window. The entire time line is divided into static intervals of a predefined length, regardless of actual job execution times (unlike in *Sliding Window Counters*).

Fixed window is required to be created first and we can do this in the following way. First, we need to pick some resource identifier unique for our application that will be used later when applying an attribute. Then specify the upper limit as well as the length of an interval (minimum 1 second) via the options.

```
IThrottlingManager manager = new ThrottlingManager();
manager.AddOrUpdateFixedWindow("github", new FixedWindowOptions(5000, TimeSpan.
↪FromHours(1)));
```

After creating a fixed window, simply apply the `FixedWindowAttribute` filter on one or multiple background job methods, and their state changing pipeline will be modified to apply the throttling rules.

```
[FixedWindow("github")]
public void ProcessCommits() { /* ... */ }
```

When background job associated with a fixed window is about to execute, the current time interval is queried to see the number of already performed job executions. If it's less than the limit value, then background job is executed. If not, background job is throttled (scheduled to the next interval by default).

When it's time to stop using the fixed window, we should remove all the corresponding `FixedWindowAttribute` filters first from our jobs, and simply call the following method. There's no need to use the `Release` mode for fixed windows as in *Concurrency Limiters*, because they don't do anything on this phase.

```
manager.RemoveFixedWindowIfExists("github");
```

Fixed window counter is a special case of the Sliding Window Counter described in the next section, with a single bucket. It does not enforce the limitation that *for any given time interval there will be no more than X executions*. So it is possible for one-hour length interval with maximum 4 executions to have 4 executions at 12:59 and another 4 just in a minute at 13:00, because they fall into different intervals.
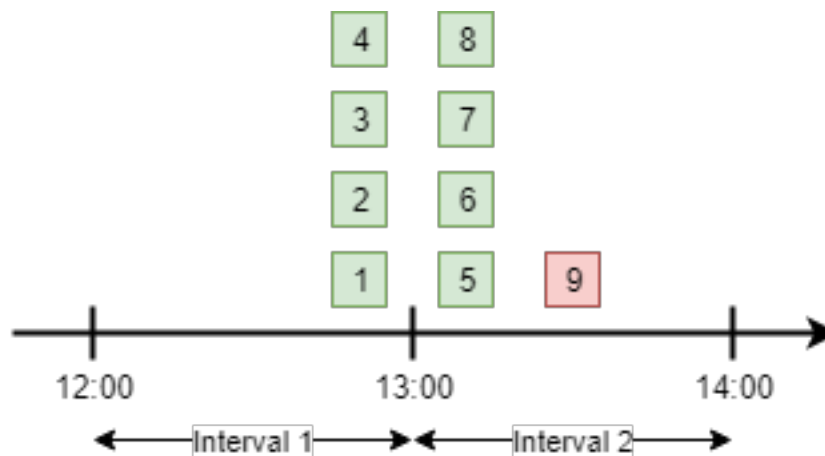
To avoid this behavior, consider using *Sliding Window Counters* described below.

However fixed windows require minimal information to be stored unlike sliding windows discussed next – only timestamp of the active interval to wrap around clock skew issues on different servers and know when to reset the counter,

and the counter itself. As per the logic of a primitive, no timestamps of individual background job executions are stored.

Sliding Window Counters

Sliding window counters are also limiting the number of background job executions over a certain time window. But unlike fixed windows, where the whole timeline is divided into large fixed intervals, intervals in sliding window counters (called "buckets") are more fine grained. Sliding window stores multiple buckets, and each bucket has its timestamp and execution counter.

In the following example we are creating a sliding window counter with one-hour interval and 3 buckets in each interval, and rate limit of 4 executions.

```
manager.AddOrUpdateSlidingWindow("dropbox", new SlidingWindowOptions(
    limit: 4,
    interval: TimeSpan.FromHours(1),
    buckets: 3));
```

After creating a window counter, we need to decorate the necessary background job methods with the `SlidingWindowAttribute` filter with the same resource identifier as in the above code snippet to tell state changing pipeline to inject the throttling logic.

```
[SlidingWindow("dropbox")]
public void ProcessFiles() { /* ... */ }
```

Each bucket participates in multiple intervals as shown in the image below, and the *no more than X executions* requirement is enforced for each of those intervals. So if we had 4 executions at 12:59, all background jobs at 13:00 will be throttled and delayed unlike in a fixed window counter.
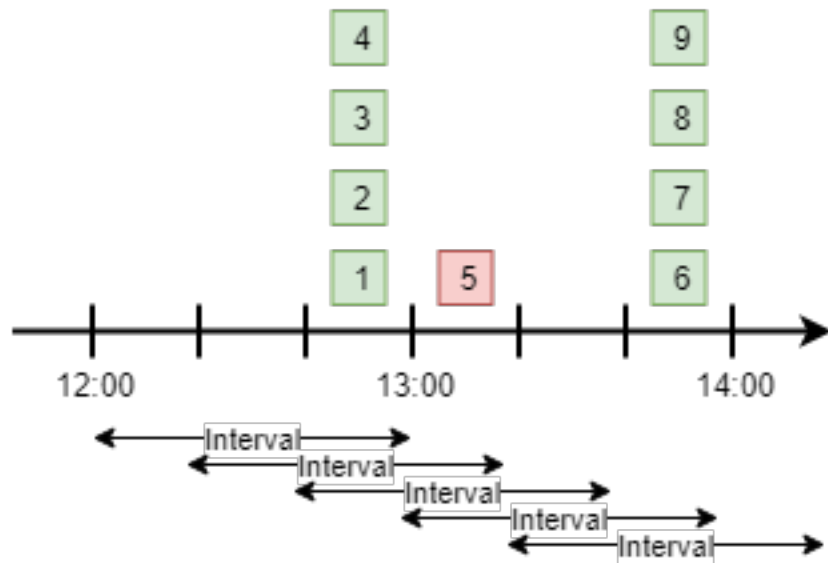
But as we can see in the picture above, background jobs 6-9 will be delayed to 13:40 and executed successfully at that time, although the configured one-hour interval has not passed yet. We can increase the number of buckets to a higher value, but minimal allowed interval of a single bucket is 1 second.

Note: So there's always a chance that limits will be violated, but that's a practical limitation – otherwise we will need to store timestamp for each individual background job that will result in an enormous payload size.

When it's time to remove the throttling on all the affected methods, just remove their references to the `SlidingWindowAttribute` filter and call the following method. Unlike *Concurrency Limiters* it's safe to remove the attributes without changing the mode first, because no work is actually made during the background job completion.

```
manager.RemoveSlidingWindowIfExists("dropbox");
```

Dynamic Window Counters

Dynamic window counter allows us to create sliding window counters dynamically depending on background job arguments. It's also possible to set up an upper limit for all of its sliding windows, and even use some rebalancing strategies. With all of these features we can get some kind of fair processing, where one participant can't capture all the available resources that's especially useful for multi-tenant applications.

`DynamicWindowAttribute` filter is responsible for this kind of throttling, and along with setting a resource identifier we need to specify the window format with String.Format-like placeholders (as in *Mutexes*) that will be converted into dynamic window identifiers at run-time based on job arguments.

Maximal length of resource identifiers is 100 characters

Please keep this in mind especially when using dynamic resource identifiers.

```
[DynamicWindow("newsletter", "tenant:{0}")]
public void SendNewsletter(long tenantId, string template) { /* ... */ }
```

Dynamic Fixed Windows

The following code snippet demonstrates the simplest form of a dynamic window counter. Since there's a single bucket, it will create a fixed window of one-hour length with maximum 4 executions per each tenant. There will be up to 1000 fixed windows to not to blow up the data structure's size.

```
IThrottlingManager manager = new ThrottlingManager();

manager.AddOrUpdateDynamicWindow("newsletter", new DynamicWindowOptions(
    limit: 4,
    interval: TimeSpan.FromHours(1),
    buckets: 1));
```

Dynamic Sliding Windows

If we increase the number of buckets, we'll be able to use sliding windows instead with the given number of buckets. Limitations are the same as in sliding windows, so minimum bucket length is 1 second. As with fixed windows, there will be up to 1000 sliding windows to keep the size under control.

```
manager.AddOrUpdateDynamicWindow("newsletter", new DynamicWindowOptions(
    limit: 4,
    interval: TimeSpan.FromHours(1),
    buckets: 60));
```

Limiting the Capacity

Capacity allows us to control how many fixed or sliding sub-windows will be created dynamically. After running the following sample, there will be maximum 5 sub-windows limited to 4 executions. This is useful in scenarios when we don't want a particular background job to take all the available resources.

```
manager.AddOrUpdateDynamicWindow("newsletter", new DynamicWindowOptions(
    capacity: 20,
    limit: 4,
    interval: TimeSpan.FromHours(1),
    buckets: 60));
```

Rebalancing Limits

When the capacity is set, we can also define dynamic limits for individual sub-windows in the following way. When rebalancing is enabled, individual limits depend on a number of active sub-windows and the capacity.

```
manager.AddOrUpdateDynamicWindow("newsletter", new DynamicWindowOptions(
    capacity: 20,
    minLimit: 2,
    maxLimit: 20,
    interval: TimeSpan.FromHours(1),
    buckets: 60));
```

So in the example above, if there are background jobs only for a single tenant, they will be performed at full speed, 20 per hour. But if other participant is trying to enter, existing ones will be limited in the following way.

- 1 participant: 20 per hour

- 2 participants: 10 per hour for each

- 3 participants: 7 per hour for 2 of them, and 6 per hour for the last

- 4 participants: 5 per hour for each

- …

- 10 participants: 2 per hour for each

Removing the Throttling

As with other rate limiters, you can just remove the `DynamicWindow` attributes from your methods and call the following methods. There's no need to change the mode to `Release` as with *Concurrency Limiters*, since no logic is running on background job completion.

```
manager.RemoveDynamicWindowIfExists("newsletter");
```

6.5 Best Practices

Background job processing can differ a lot from a regular method invocation. This guide will help you keep background processing running smoothly and efficiently. The information given here is based off of this blog post.

6.5.1 Make job arguments small and simple

Method invocation (i.e. a job) is serialized during the background job creation process. Arguments are converted into JSON strings using the *TypeConverter* class. If you have complex entities and/or large objects; including arrays, it is better to place them into a database, and then pass only their identities to the background job.

Instead of doing this:

```
public void Method(Entity entity) { }
```

Consider doing this:

```
public void Method(int entityId) { }
```

6.5.2 Make your background methods reentrant

Reentrancy means that a method can be interrupted in the middle of its execution and then safely called again. The interruption can be caused by many different things (i.e. exceptions, server shut-down), and Hangfire will attempt to retry processing many times.

You can have many problems, if you don't prepare your jobs to be reentrant. For example, if you are using an email sending background job and experience an error with your SMTP service, you can end with multiple emails sent to the addressee.

Instead of doing this:

```
public void Method()
{
    _emailService.Send("person@example.com", "Hello!");
}
```

Consider doing this:

```
public void Method(int deliveryId)
{
    if (_emailService.IsNotDelivered(deliveryId))
    {
        _emailService.Send("person@example.com", "Hello!");
        _emailService.SetDelivered(deliveryId);
    }
}
```

To be continued. . .

6.6 Deployment to Production

6.6.1 Making ASP.NET application always running

By default, Hangfire Server instance in a web application will not be started until the first user hits your site. Even more, there are some events that will bring your web application down after some time (I'm talking about Idle Timeout and different app pool recycling events). In these cases your *recurring tasks* and *delayed jobs* will not be enqueued, and *enqueued jobs* will not be processed.

This is particularly true for smaller sites, as there may be long periods of user inactivity. But if you are running critical jobs, you should ensure that your Hangfire Server instance is always running to guarantee the in-time background job processing.

On-Premise applications

For web applications running on servers under your control, either physical or virtual, you can use the auto-start feature of IIS 7.5 shipped with Windows Server 2008 R2. Full setup requires the following steps to be done:

1. Enable automatic start-up for Windows Process Activation (WAS) and World Wide Web Publishing (W3SVC) services (enabled by default).

2. Configure Automatic Startup for an Application pool (enabled by default).

3. Enable Always Running Mode for Application pool and configure Auto-start feature as written below.

Creating classes

First, you'll need a special class that implements the `IProcessHostPreloadClient` interface. It will be called automatically by Windows Process Activation service during its start-up and after each Application pool recycle.

```
public class ApplicationPreload : System.Web.Hosting.IProcessHostPreloadClient
{
    public void Preload(string[] parameters)
    {
        HangfireBootstrapper.Instance.Start();
    }
}
```

Then, update your `global.asax.cs` file as described below. *It is important* to call the `Stop` method of the `BackgroundJobServer` class instance, and it is also important to start Hangfire server in environments that don't have auto-start feature enabled (for example, on development machines) also.

```
public class Global : HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        HangfireBootstrapper.Instance.Start();
    }

    protected void Application_End(object sender, EventArgs e)
    {
        HangfireBootstrapper.Instance.Stop();
    }
}
```

Then, create the `HangfireBootstrapper` class as follows. Since both `Application_Start` and `Preload` methods will be called in environments with auto-start enabled, we need to ensure that the initialization logic will be called exactly once.

```
public class HangfireBootstrapper : IRegisteredObject
{
    public static readonly HangfireBootstrapper Instance = new HangfireBootstrapper();

    private readonly object _lockObject = new object();
    private bool _started;
```

(continues on next page)

```csharp
    private BackgroundJobServer _backgroundJobServer;

    private HangfireBootstrapper()
    {
    }

    public void Start()
    {
        lock (_lockObject)
        {
            if (_started) return;
            _started = true;

            HostingEnvironment.RegisterObject(this);

            GlobalConfiguration.Configuration
                .UseSqlServerStorage("connection string");
                // Specify other options here

            _backgroundJobServer = new BackgroundJobServer();
        }
    }

    public void Stop()
    {
        lock (_lockObject)
        {
            if (_backgroundJobServer != null)
            {
                _backgroundJobServer.Dispose();
            }

            HostingEnvironment.UnregisterObject(this);
        }
    }

    void IRegisteredObject.Stop(bool immediate)
    {
        Stop();
    }
}
```

And optionally, if you want to map Hangfire Dashboard UI, create an OWIN startup class:

```csharp
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
            var options = new DashboardOptions
                {
            AuthorizationFilters = new[]
            {
                new LocalRequestsOnlyAuthorizationFilter()
            }
                };
```

```
        app.UseHangfireDashboard("/hangfire", options);
    }
}
```

Enabling Service Auto-start

After creating above classes, you should edit the global `applicationHost.config` file (`%WINDIR%\System32\inetsrv\config\applicationHost.config`). First, you need to change the start mode of your application pool to `AlwaysRunning`, and then enable Service AutoStart Providers.

Save only after all modifications

After making these changes, the corresponding application pool will be restarted automatically. Make sure to save changes **only after** modifying all elements.

```xml
<applicationPools>
    <add name="MyAppWorkerProcess" managedRuntimeVersion="v4.0" startMode=
↪"AlwaysRunning" />
</applicationPools>

<!-- ... -->

<sites>
    <site name="MySite" id="1">
        <application path="/" serviceAutoStartEnabled="true"
                             serviceAutoStartProvider="ApplicationPreload" />
    </site>
</sites>

<!-- Just AFTER closing the `sites` element AND AFTER `webLimits` tag -->
<serviceAutoStartProviders>
    <add name="ApplicationPreload" type="WebApplication1.ApplicationPreload,␣
↪WebApplication1" />
</serviceAutoStartProviders>
```

Note that for the last entry, `WebApplication1.ApplicationPreload` is the full name of a class in your application that implements `IProcessHostPreloadClient` and `WebApplication1` is the name of your application's library. You can read more about this here.

There is no need to set IdleTimeout to zero – when Application pool's start mode is set to `AlwaysRunning`, idle timeout does not work anymore.

Ensuring auto-start feature is working

If something went wrong. . .

If your app won't load after these changes made, check your Windows Event Log by opening **Control Panel** → **Administrative Tools** → **Event Viewer**. Then open *Windows Logs* → *Application* and look for a recent error records.

The simplest method - recycle your Application pool, wait for 5 minutes, then go to the Hangfire Dashboard UI and check that current Hangfire Server instance was started 5 minutes ago. If you have problems – don't hesitate to ask

them on forum.

Azure web applications

Enabling always running feature for application hosted in Microsoft Azure is simpler a bit: just turn on the `Always On` switch on the Configuration page and save settings.

This setting does not work for free sites.

If nothing works for you. . .

. . . because you are using shared hosting, free Azure website or something else (btw, can you tell me your configuration in this case?), then you can use the following ways to ensure that Hangfire Server is always running:

1. Use *separate process* to handle background jobs either on the same, or dedicated host.

2. Make HTTP requests to your website on a recurring basis by external tool (for example, Pingdom).

3. *Do you know any other ways? Let me know!*

Making ASP.NET Core application always running on IIS

Follow these directions in IIS:

1. Set application pool under which the application runs to:

   a. **.NET CLR version: .NET CLR Version v4.0.30319**

      i. Normally, for a .NET Core app, you'd use *No managed code*, but if you do that, the application preload option won't work.

   b. Managed pipeline mode: **Integrated**

2. Right-click on the same application pool and select "Advanced Settings". Update the following values:

   a. **Set start mode to "Always Running".**

      i. Setting the start mode to "Always Running" tells IIS to start a worker process for your application right away, without waiting for the initial request.
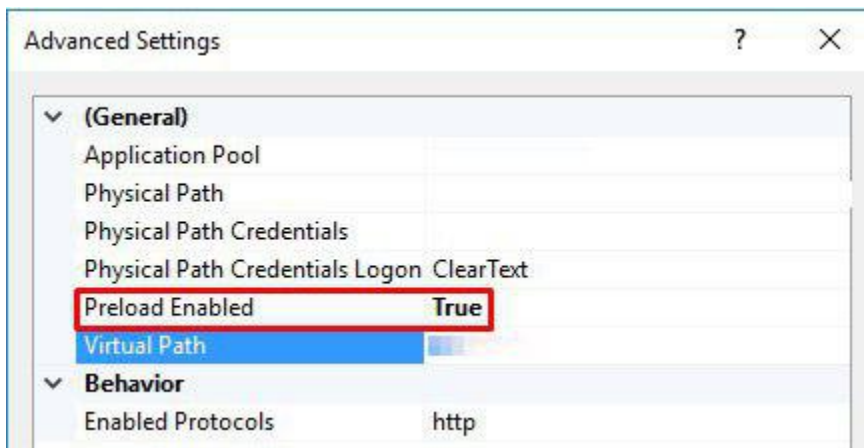
   b. Set Idle Time-Out (minutes) to 0.
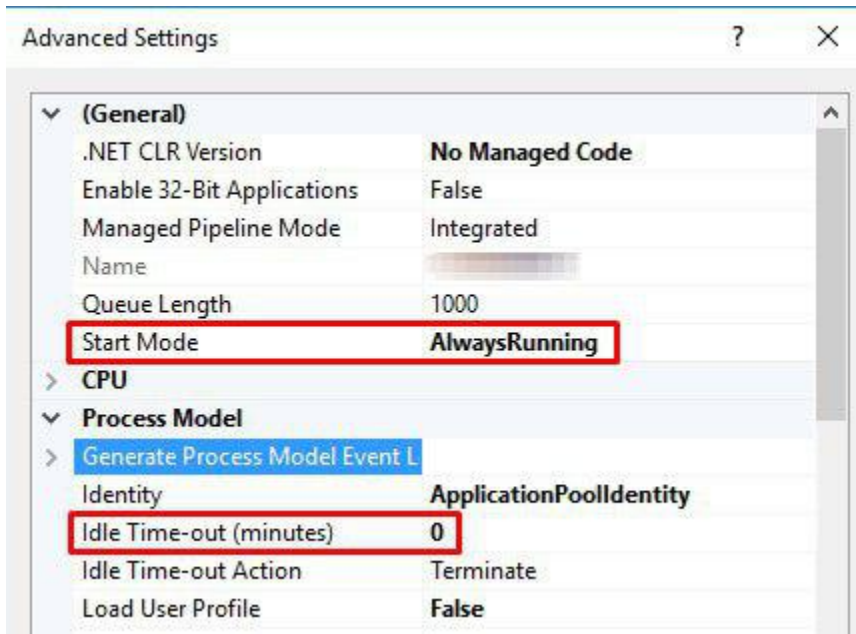
3. Open Advanced Settings on your application:

4. Set Preload Enabled = True:

5. Go to the *Configuration Editor* on your app, and navigate to *system.webServer/applicationInitialization*. Set the following settings:

    a. doAppInitAfterRestart: **True**

    b. **Open up the *Collection...* ellipsis. On the next window, click *Add* and enter the following:**

        i. hostName: **{URL host for your Hangfire application}**

   ii. initializationPage: **{path for your Hangfire dashboard, like /hangfire}**

You can check this page for more documentation about it.

6.6.2 Using performance counters

Pro Only

This feature is a part of Hangfire Pro package set

Performance Counters is a standard way to measure different application metrics on a Windows platform. This package enables Hangfire to publish performance counters so you can track them using different tools, including Performance Monitor, Nagios, New Relic and others.
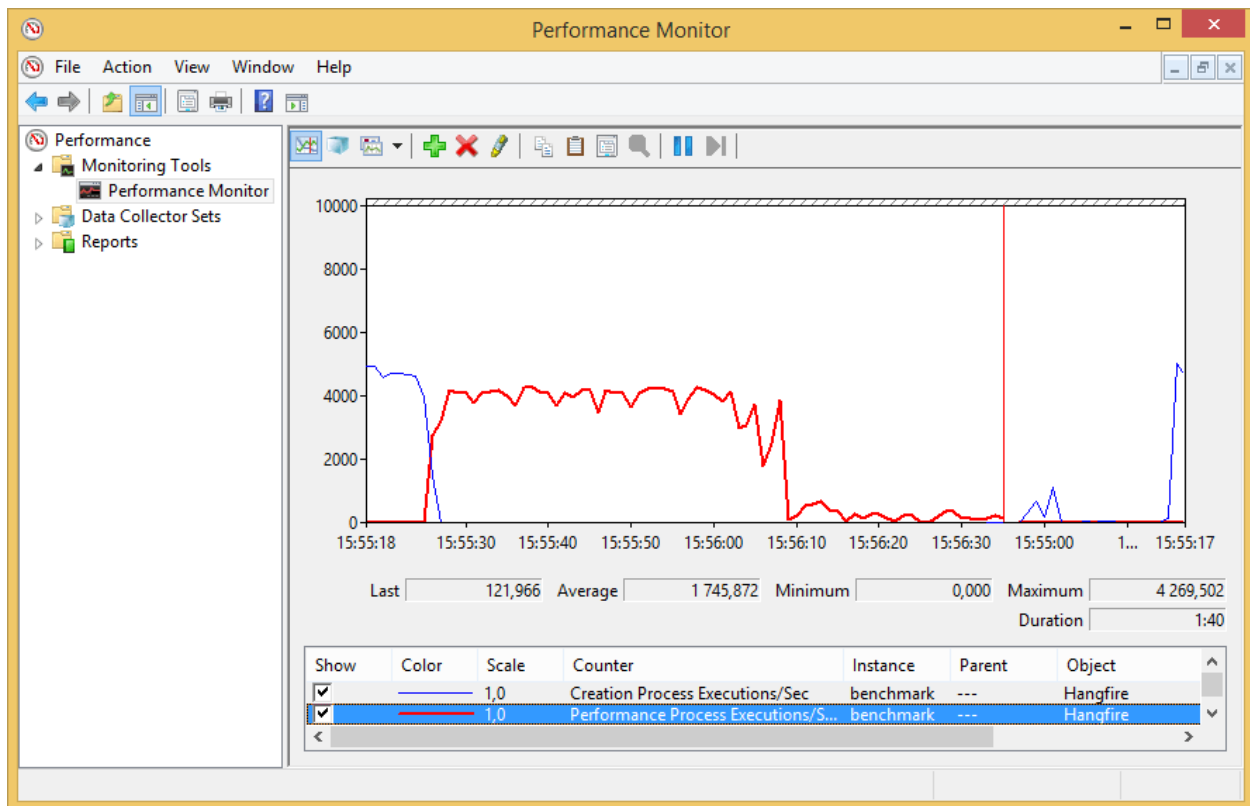
Installation

Before configuring Hangfire and starting to publish performance counters, you need to add them to *every machine* you use by running `hangfire-perf.exe` program with the `ipc` argument (for both install and update actions):

```
hangfire-perf ipc
```

To uninstall performance counters, use the `upc` command:

```
hangfire-perf upc
```

Configuration

Performance counters are exposed through the `Hangfire.Pro.PerformanceCounters` package. After adding it to your project, you need only to initialize them by invoking the following method:

```csharp
using Hangfire.PerformanceCounters;

PerformanceCounters.Initialize("unique-app-id");
```

Initialization logic is much easier within your OWIN Startup class:

```csharp
using Hangfire.PerformanceCounters;

public void Configure(IAppBuilder app)
{
    app.UseHangfirePerformanceCounters();
}
```

Membership Configuration

Also, ensure your IIS/ASP.NET user is a member of the "Performance Monitor Users" group.

Performance counters

Here is the list of performance counters currently exposed:

- Creation Process Executions
- Creation Process Executions/Sec
- Performance Process Executions
- Performance Process Executions/Sec
- Transitions to Succeeded State
- Transitions to Succeeded State/Sec
- Transitions to Failed State/Sec

Want more? Just open a GitHub Issue and describe what metric you want to see.

6.7 Extensibility

6.7.1 Using job filters

All processes are implemented with Chain-of-responsibility pattern and can be intercepted like with ASP.NET MVC Action Filters.

Define the filter

```csharp
public class LogEverythingAttribute : JobFilterAttribute,
    IClientFilter, IServerFilter, IElectStateFilter, IApplyStateFilter
{
    private static readonly ILog Logger = LogProvider.GetCurrentClassLogger();

    public void OnCreating(CreatingContext context)
    {
        Logger.InfoFormat("Creating a job based on method `{0}`...", context.Job.
→Method.Name);
    }

    public void OnCreated(CreatedContext context)
    {
        Logger.InfoFormat(
            "Job that is based on method `{0}` has been created with id `{1}`",
            context.Job.Method.Name,
            context.BackgroundJob?.Id);
    }

    public void OnPerforming(PerformingContext context)
    {
        Logger.InfoFormat("Starting to perform job `{0}`", context.BackgroundJob.Id);
    }

    public void OnPerformed(PerformedContext context)
    {
        Logger.InfoFormat("Job `{0}` has been performed", context.BackgroundJob.Id);
    }

    public void OnStateElection(ElectStateContext context)
    {
        var failedState = context.CandidateState as FailedState;
        if (failedState != null)
        {
            Logger.WarnFormat(
                "Job `{0}` has been failed due to an exception `{1}`",
                context.BackgroundJob.Id,
                failedState.Exception);
        }
    }

    public void OnStateApplied(ApplyStateContext context, IWriteOnlyTransaction␣
→transaction)
    {
        Logger.InfoFormat(
            "Job `{0}` state was changed from `{1}` to `{2}`",
            context.BackgroundJob.Id,
            context.OldStateName,
            context.NewState.Name);
    }

    public void OnStateUnapplied(ApplyStateContext context, IWriteOnlyTransaction␣
→transaction)
    {
        Logger.InfoFormat(
            "Job `{0}` state `{1}` was unapplied.",
```

(continues on next page)

```
            context.BackgroundJob.Id,
            context.OldStateName);
    }
}
```

Apply it

Like ASP.NET filters, you can apply filters on method, class and globally:

```
[LogEverything]
public class EmailService
{
    [LogEverything]
    public static void Send() { }
}

GlobalJobFilters.Filters.Add(new LogEverythingAttribute());
```

If your service is resolved via interface, then the filter should be applied to your interface instead of class:

```
[LogEverything]
public interface IEmailService
{
    [LogEverything]
    void Send() { }
}
```

6.8 Tutorials

6.8.1 Sending Mail in Background with ASP.NET MVC

Table of Contents

Let's start with a simple example: you are building your own blog using ASP.NET MVC and want to receive an email notification about each posted comment. We will use the simple but awesome Postal library to send emails.

Tip: I've prepared a simple application that has only comments list, you can download its sources to start work on tutorial.

You already have a controller action that creates a new comment, and want to add the notification feature.

```csharp
// ~/HomeController.cs

[HttpPost]
public ActionResult Create(Comment model)
{
    if (ModelState.IsValid)
    {
        _db.Comments.Add(model);
        _db.SaveChanges();
    }

    return RedirectToAction("Index");
}
```

Installing Postal

First, install the `Postal` NuGet package:

```
Install-Package Postal.Mvc5
```

Then, create `~/Models/NewCommentEmail.cs` file with the following contents:

```csharp
using Postal;

namespace Hangfire.Mailer.Models
{
    public class NewCommentEmail : Email
    {
        public string To { get; set; }
        public string UserName { get; set; }
        public string Comment { get; set; }
    }
}
```

Create a corresponding template for this email by adding the `~/Views/Emails/NewComment.cshtml` file:

```
@model Hangfire.Mailer.Models.NewCommentEmail
To: @Model.To
From: mailer@example.com
Subject: New comment posted

Hello,
There is a new comment from @Model.UserName:

@Model.Comment

<3
```

And call Postal to send email notification from the `Create` controller action:

```csharp
[HttpPost]
public ActionResult Create(Comment model)
{
    if (ModelState.IsValid)
```

```
    {
        _db.Comments.Add(model);
        _db.SaveChanges();

        var email = new NewCommentEmail
        {
            To = "yourmail@example.com",
            UserName = model.UserName,
            Comment = model.Text
        };

        email.Send();
    }

    return RedirectToAction("Index");
}
```

Then configure the delivery method in the `web.config` file (by default, tutorial source code uses `C:\Temp` directory to store outgoing mail):

```xml
<system.net>
  <mailSettings>
    <smtp deliveryMethod="SpecifiedPickupDirectory">
      <specifiedPickupDirectory pickupDirectoryLocation="C:\Temp\" />
    </smtp>
  </mailSettings>
</system.net>
```

That's all. Try to create some comments and you'll see notifications in the pickup directory.

Further considerations

But why should a user wait until the notification was sent? There should be some way to send emails asynchronously, in the background, and return a response to the user as soon as possible.

Unfortunately, asynchronous controller actions do not help in this scenario, because they do not yield response to the user while waiting for the asynchronous operation to complete. They only solve internal issues related to thread pooling and application capacity.

There are great problems with background threads also. You should use Thread Pool threads or custom ones that are running inside ASP.NET application with care – you can simply lose your emails during the application recycle process (even if you register an implementation of the `IRegisteredObject` interface in ASP.NET).

And you are unlikely to want to install external Windows Services or use Windows Scheduler with a console application to solve this simple problem (we are building a personal blog, not an e-commerce solution).

Installing Hangfire

To be able to put tasks into the background and not lose them during application restarts, we'll use Hangfire. It can handle background jobs in a reliable way inside ASP.NET application without external Windows Services or Windows Scheduler.

```
Install-Package Hangfire
```

Hangfire uses SQL Server or Redis to store information about background jobs. So, let's configure it. Add a new class Startup into the root of the project:

```
   public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        GlobalConfiguration.Configuration
            .UseSqlServerStorage(
                "MailerDb",
                new SqlServerStorageOptions { QueuePollInterval = TimeSpan.
→FromSeconds(1) });


        app.UseHangfireDashboard();
        app.UseHangfireServer();



    }

}
```

The `SqlServerStorage` class will install all database tables automatically on application start-up (but you are able to do it manually).

Now we are ready to use Hangfire. It asks us to wrap a piece of code that should be executed in background in a public method.

```
[HttpPost]
public ActionResult Create(Comment model)
{
    if (ModelState.IsValid)
    {
        _db.Comments.Add(model);
        _db.SaveChanges();

        BackgroundJob.Enqueue(() => NotifyNewComment(model.Id));
    }

    return RedirectToAction("Index");
}
```

Note, that we are passing a comment identifier instead of a full comment – Hangfire should be able to serialize all method call arguments to string values. The default serializer does not know anything about our `Comment` class. Furthermore, the integer identifier takes less space in serialized form than the full comment text.

Now, we need to prepare the `NotifyNewComment` method that will be called in the background. Note that `HttpContext.Current` is not available in this situation, but Postal library can work even outside of ASP.NET request. But first install another package (that is needed for Postal 0.9.2, see the issue). Let's update package and bring in the RazorEngine

```
Update-Package -save
```

```
public static void NotifyNewComment(int commentId)
{
    // Prepare Postal classes to work outside of ASP.NET request
    var viewsPath = Path.GetFullPath(HostingEnvironment.MapPath(@"~/Views/Emails"));
    var engines = new ViewEngineCollection();
```
(continues on next page)

```csharp
    engines.Add(new FileSystemRazorViewEngine(viewsPath));

    var emailService = new EmailService(engines);

    // Get comment and send a notification.
    using (var db = new MailerDbContext())
    {
        var comment = db.Comments.Find(commentId);

        var email = new NewCommentEmail
        {
            To = "yourmail@example.com",
            UserName = comment.UserName,
            Comment = comment.Text
        };

        emailService.Send(email);
    }
}
```

This is a plain C# static method. We are creating an `EmailService` instance, finding the desired comment and sending a mail with Postal. Simple enough, especially when compared to a custom Windows Service solution.

> **Warning:** Emails now are sent outside of request processing pipeline. As of Postal 1.0.0, there are the following limitations: you can not use layouts for your views, you MUST use `Model` and not `ViewBag`, embedding images is not supported either.

That's all! Try to create some comments and see the `C:\Temp` path. You also can check your background jobs at `http://<your-app>/hangfire`. If you have any questions, you are welcome to use the comments form below.

> **Note:** If you experience assembly load exceptions, please, please delete the following sections from the `web.config` file (I forgot to do this, but don't want to re-create the repository):

```xml
<dependentAssembly>
  <assemblyIdentity name="Newtonsoft.Json" publicKeyToken="30ad4fe6b2a6aeed" culture=
→"neutral" />
  <bindingRedirect oldVersion="0.0.0.0-6.0.0.0" newVersion="6.0.0.0" />
</dependentAssembly>
<dependentAssembly>
  <assemblyIdentity name="Common.Logging" publicKeyToken="af08829b84f0328e" culture=
→"neutral" />
  <bindingRedirect oldVersion="0.0.0.0-2.2.0.0" newVersion="2.2.0.0" />
</dependentAssembly>
```

Automatic retries

When the `emailService.Send` method throws an exception, Hangfire will retry it automatically after a delay (that is increased with each attempt). The retry attempt count is limited (10 by default), but you can increase it. Just apply the `AutomaticRetryAttribute` to the `NotifyNewComment` method:

```
[AutomaticRetry( Attempts = 20 )]
public static void NotifyNewComment(int commentId)
{
    /* ... */
}
```

Logging

You can log cases when the maximum number of retry attempts has been exceeded. Try to create the following class:

```
public class LogFailureAttribute : JobFilterAttribute, IApplyStateFilter
{
    private static readonly ILog Logger = LogProvider.GetCurrentClassLogger();

    public void OnStateApplied(ApplyStateContext context, IWriteOnlyTransaction
→transaction)
    {
        var failedState = context.NewState as FailedState;
        if (failedState != null)
        {
            Logger.ErrorException(
                String.Format("Background job #{0} was failed with an exception.",
→context.JobId),
                failedState.Exception);
        }
    }

    public void OnStateUnapplied(ApplyStateContext context, IWriteOnlyTransaction
→transaction)
    {
    }
}
```

And add it:

Either globally by calling the following method at application start:

```
public void Configuration(IAppBuilder app)
{
    GlobalConfiguration.Configuration
        .UseSqlServerStorage(
            "MailerDb",
            new SqlServerStorageOptions { QueuePollInterval = TimeSpan.FromSeconds(1)
→})
            .UseFilter(new LogFailureAttribute());

    app.UseHangfireDashboard();
    app.UseHangfireServer();
}
```

Or locally by applying the attribute to a method:

```
[LogFailure]
public static void NotifyNewComment(int commentId)
{
    /* ... */
}
```

You can see the logging is working when you add a new breakpoint in LogFailureAttribute class inside method On-StateApplied

If you like to use any of common logger and you do not need to do anything. Let's take NLog as an example. Install NLog (current version: 4.2.3)

```
Install-Package NLog
```

Add a new Nlog.config file into the root of the project.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      autoReload="true"
      throwExceptions="false">

  <variable name="appName" value="HangFire.Mailer" />

  <targets async="true">
    <target xsi:type="File"
            name="default"
             layout="${longdate} - ${level:uppercase=true}: ${message}${onexception:$
→{newline}EXCEPTION\: ${exception:format=ToString}}"
            fileName="${specialfolder:ApplicationData}\${appName}\Debug.log"
            keepFileOpen="false"
            archiveFileName="${specialfolder:ApplicationData}\${appName}\Debug_$
→{shortdate}.{##}.log"
            archiveNumbering="Sequence"
            archiveEvery="Day"
            maxArchiveFiles="30" />

    <target xsi:type="EventLog"
            name="eventlog"
            source="${appName}"
            layout="${message}${newline}${exception:format=ToString}"/>
  </targets>
  <rules>
    <logger name="*" writeTo="default" minlevel="Info" />
    <logger name="*" writeTo="eventlog" minlevel="Error" />
  </rules>
</nlog>
```

run application and new log file could be find on cd %appdata%HangFire.MailerDebug.log

Fix-deploy-retry

If you made a mistake in your `NotifyNewComment` method, you can fix it and restart the failed background job via the web interface. Try it:

```
// Break background job by setting null to emailService:
EmailService emailService = null;
```

Compile a project, add a comment and go to the web interface by typing `http://<your-app>/hangfire`. Exceed all automatic attempts, then fix the job, restart the application, and click the `Retry` button on the *Failed jobs* page.

Preserving current culture

If you set a custom culture for your requests, Hangfire will store and set it during the performance of the background job. Try the following:

```
// HomeController/Create action
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("es-ES");
BackgroundJob.Enqueue(() => NotifyNewComment(model.Id));
```

And check it inside the background job:

```
public static void NotifyNewComment(int commentId)
{
    var currentCultureName = Thread.CurrentThread.CurrentCulture.Name;
    if (currentCultureName != "es-ES")
    {
        throw new InvalidOperationException(String.Format("Current culture is {0}",
→currentCultureName));
    }
    // ...
```

6.8.2 Highlighter Tutorial

| Sample | http://demo.hangfire.io, sources |

Table of Contents

- *Overview*
- *Setting up the project*
  - *Prerequisites*
  - *Creating a project*
  - *Hiliting the code*
- *The problem*
- *Solving a problem*
  - *Installing Hangfire*
  - *Moving to background*
- *Conclusion*

Overview

Consider you are building a code snippet gallery web application like GitHub Gists, and want to implement the syntax highlighting feature. To improve user experience, you also want it to work even if a user has disabled JavaScript in her browser.

To support this scenario and to reduce the project development time, you chosen to use a web service for syntax highlighting, such as http://pygments.org/ or http://www.hilite.me.

Note: Although this feature can be implemented without web services (using different syntax highlighter libraries for .NET), we are using them just to show some pitfalls regarding to their usage in web applications.

You can substitute this example with real-world scenario, like using external SMTP server, another services or even long-running CPU-intensive task.

Setting up the project

Tip: This section contains steps to prepare the project. However, if you don't want to do the boring stuff or if you have problems with project set-up, you can download the tutorial source code and go straight to *The problem* section.

Prerequisites

The tutorial uses **Visual Studio 2012** with Web Tools 2013 for Visual Studio 2012 installed, but it can be built either with Visual Studio 2013.

The project uses **.NET 4.5**, **ASP.NET MVC 5** and **SQL Server 2008 Express** or later database.

Creating a project

Let's start from scratch. Create an *ASP.NET MVC 5 Empty Project* and name this awesome web application `Hangfire.Highlighter` (you can name it as you want, but prepare to change namespaces).

I've included some screenshots to make the project set-up not so boring:
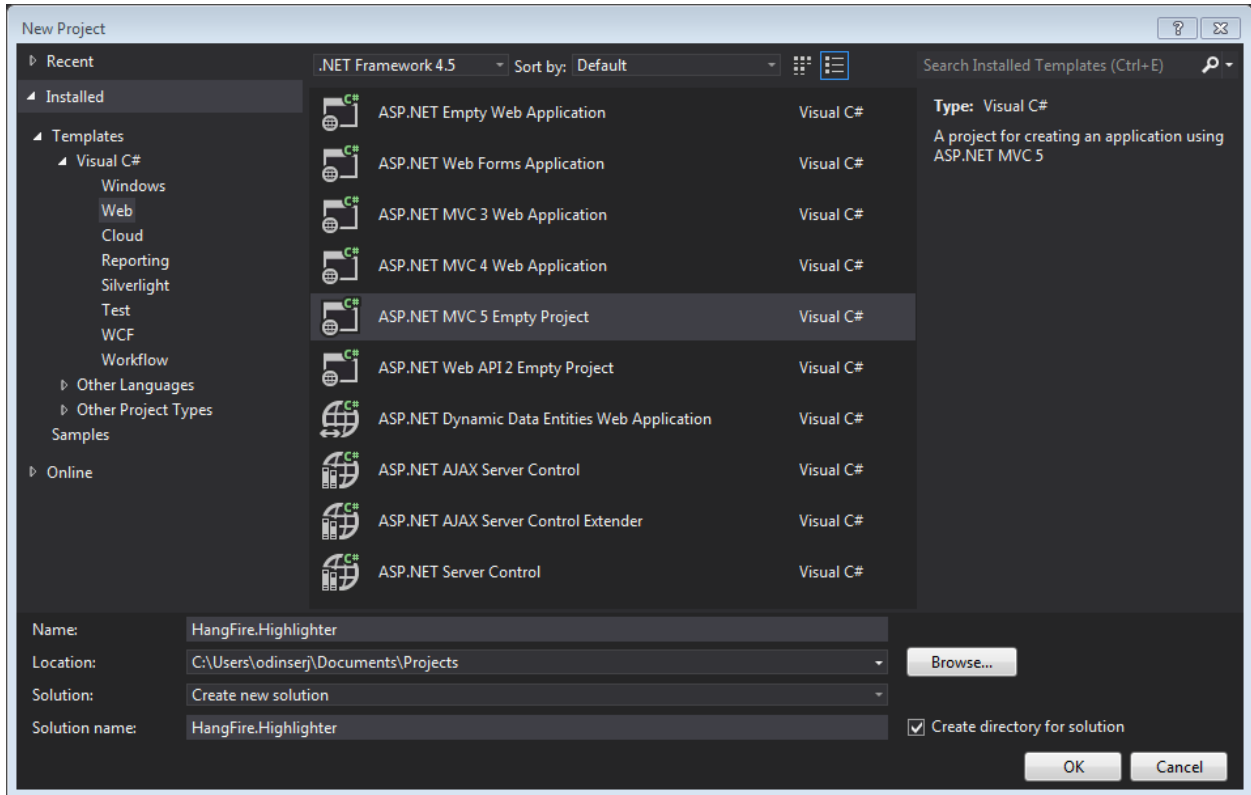
Then, we need a controller to handle web requests. So scaffold an **MVC 5 Controller - Empty** controller and call it `HomeController`:
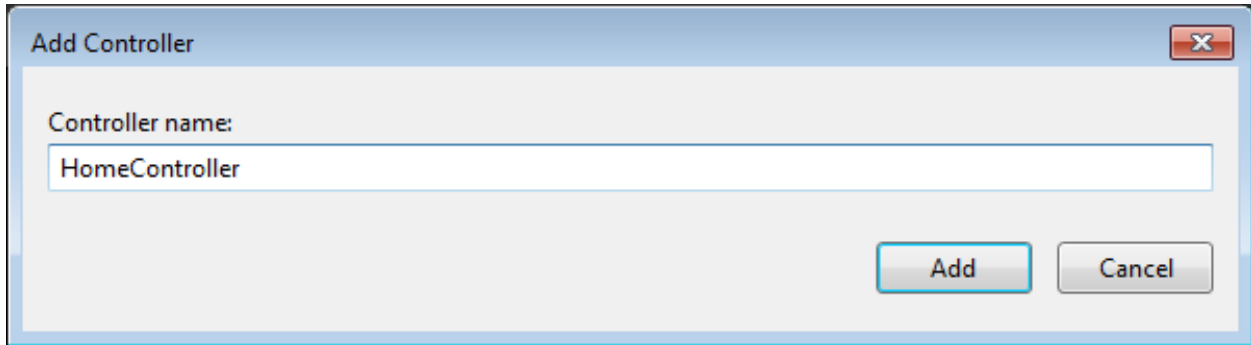
Our controller contains now only `Index` action and looks like:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

We have a controller with a single action. To test that our application is working, scaffold an **empty view** for `Index` action.

The view scaffolding process also adds additional components to the project, like *Bootstrap*, *jQuery*, etc. After these steps my solution looks like:
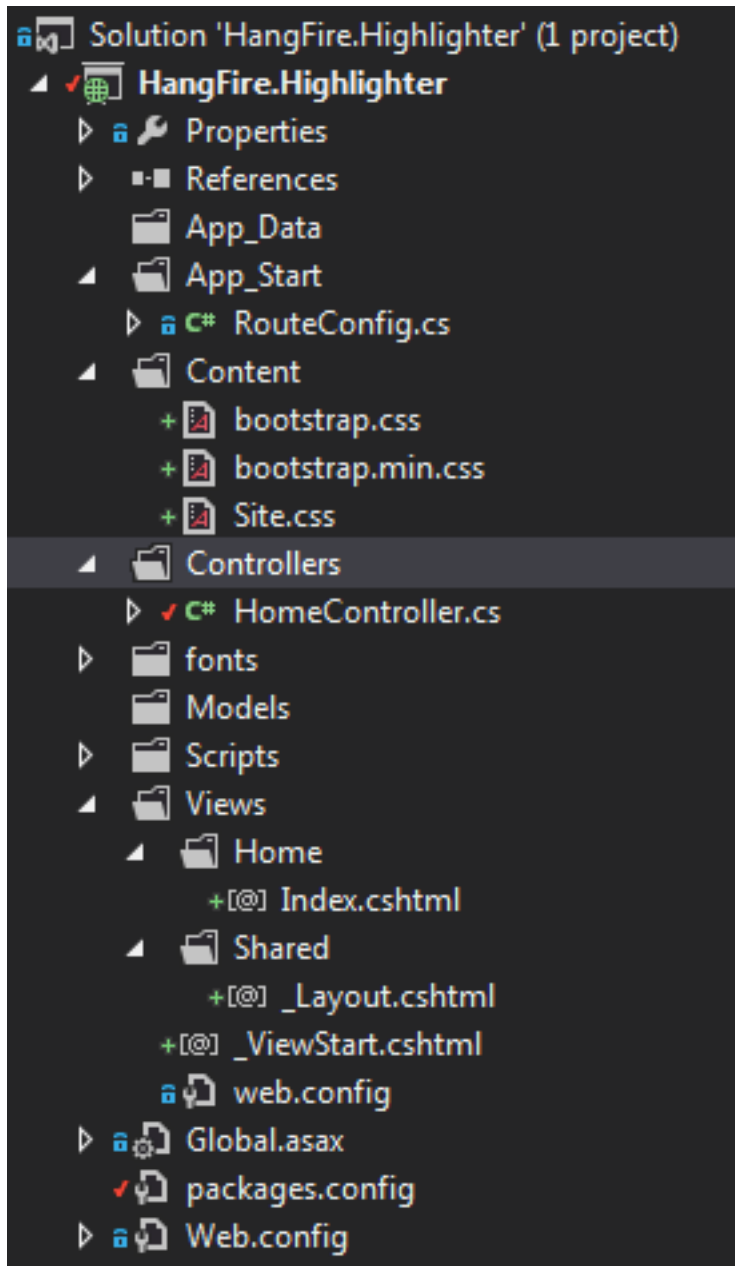
Let's test the initial setup of our application. Press the F5 key to start debugging and wait for your browser. If you encounter exceptions or don't see the default page, try to reproduce all the given steps, see the tutorial sources or ask a question in the comments below.

Defining a model

We should use a persistent storage to preserve snippets after application restarts. So, we'll use **SQL Server 2008 Express** (or later) as a relational storage, and **Entity Framework** to access the data of our application.

Installing Entity Framework

Open the Package Manager Console window and type:

```
Install-Package EntityFramework
```

After the package installed, create a new class in the `Models` folder and name it `HighlighterDbContext`:

```csharp
// ~/Models/HighlighterDbContext.cs

using System.Data.Entity;

namespace Hangfire.Highlighter.Models
{
    public class HighlighterDbContext : DbContext
    {
        public HighlighterDbContext() : base("HighlighterDb")
        {
        }
    }
}
```

Please note, that we are using undefined yet connection string name `HighlighterDb`. So, lets add it to the `web.config` file just after the `</configSections>` tag:

```xml
<connectionStrings>
  <add name="HighlighterDb" connectionString="Server=.\sqlexpress; Database=Hangfire.
→Highlighter; Trusted_Connection=True;" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Then enable **Entity Framework Code First Migrations** by typing in your *Package Manager Console* window the following command:

```
Enable-Migrations
```

Adding code snippet model

It's time to add the most valuable class in the application. Create the `CodeSnippet` class in the `Models` folder with the following code:

```csharp
// ~/Models/CodeSnippet.cs

using System;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace Hangfire.Highlighter.Models
{
    public class CodeSnippet
    {
        public int Id { get; set; }

        [Required, AllowHtml, Display(Name = "C# source")]
        public string SourceCode { get; set; }
        public string HighlightedCode { get; set; }

        public DateTime CreatedAt { get; set; }
        public DateTime? HighlightedAt { get; set; }
    }
}
```

Don't forget to include the following property in the `HighlighterDbContext` class:

```csharp
// ~/Models/HighlighterDbContext.cs
public DbSet<CodeSnippet> CodeSnippets { get; set; }
```

Then add a database migration and run it by typing the following commands into the Package Manager Console window:

```
Add-Migration AddCodeSnippet
Update-Database
```

Our database is ready to use!

Creating actions and views

Now its time to breathe life into our project. Please, modify the following files as described.

```csharp
// ~/Controllers/HomeController.cs

using System;
using System.Linq;
using System.Web.Mvc;
using Hangfire.Highlighter.Models;

namespace Hangfire.Highlighter.Controllers
{
    public class HomeController : Controller
    {
        private readonly HighlighterDbContext _db = new HighlighterDbContext();

        public ActionResult Index()
        {
            return View(_db.CodeSnippets.ToList());
        }

        public ActionResult Details(int id)
        {
            var snippet = _db.CodeSnippets.Find(id);
            return View(snippet);
        }

        public ActionResult Create()
        {
            return View();
        }

        [HttpPost]
        public ActionResult Create([Bind(Include="SourceCode")] CodeSnippet snippet)
        {
            if (ModelState.IsValid)
            {
                snippet.CreatedAt = DateTime.UtcNow;

                // We'll add the highlighting a bit later.

                _db.CodeSnippets.Add(snippet);
```

(continues on next page)

```
            _db.SaveChanges();

            return RedirectToAction("Details", new { id = snippet.Id });
        }

        return View(snippet);
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            _db.Dispose();
        }
        base.Dispose(disposing);
    }
}
}
```

```
@* ~/Views/Home/Index.cshtml *@

@model IEnumerable<Hangfire.Highlighter.Models.CodeSnippet>
@{ ViewBag.Title = "Snippets"; }

<h2>Snippets</h2>

<p><a class="btn btn-primary" href="@Url.Action("Create")">Create Snippet</a></p>
<table class="table">
    <tr>
        <th>Code</th>
        <th>Created At</th>
        <th>Highlighted At</th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                <a href="@Url.Action("Details", new { id = item.Id })">@Html.Raw(item.
→HighlightedCode)</a>
            </td>
            <td>@item.CreatedAt</td>
            <td>@item.HighlightedAt</td>
        </tr>
    }
</table>
```

```
@* ~/Views/Home/Create.cshtml *@

@model Hangfire.Highlighter.Models.CodeSnippet
@{ ViewBag.Title = "Create a snippet"; }

<h2>Create a snippet</h2>

@using (Html.BeginForm())
{
```

```
    @Html.ValidationSummary(true)

    <div class="form-group">
        @Html.LabelFor(model => model.SourceCode)
        @Html.ValidationMessageFor(model => model.SourceCode)
        @Html.TextAreaFor(model => model.SourceCode, new { @class = "form-control",␣
→style = "min-height: 300px;", autofocus = "true" })
    </div>

    <button type="submit" class="btn btn-primary">Create</button>
    <a class="btn btn-default" href="@Url.Action("Index")">Back to List</a>
}
```

```
@* ~/Views/Home/Details.cshtml *@

@model Hangfire.Highlighter.Models.CodeSnippet
@{ ViewBag.Title = "Details"; }

<h2>Snippet <small>#@Model.Id</small></h2>

<div>
    <dl class="dl-horizontal">
        <dt>@Html.DisplayNameFor(model => model.CreatedAt)</dt>
        <dd>@Html.DisplayFor(model => model.CreatedAt)</dd>
        <dt>@Html.DisplayNameFor(model => model.HighlightedAt)</dt>
        <dd>@Html.DisplayFor(model => model.HighlightedAt)</dd>
    </dl>

    <div class="clearfix"></div>
</div>

<div>@Html.Raw(Model.HighlightedCode)</div>
```

Adding MiniProfiler

To not to profile our application by eye, we'll use the `MiniProfiler` package available on NuGet.

```
Install-Package MiniProfiler
```

After installing, update the following files as described to enable profiling.

```
// ~/Global.asax.cs

public class MvcApplication : HttpApplication
{
    /* ... */

    protected void Application_BeginRequest()
    {
        StackExchange.Profiling.MiniProfiler.Start();
    }

    protected void Application_EndRequest()
    {
```

```
            StackExchange.Profiling.MiniProfiler.Stop();
    }
}
```

```
@* ~/Views/Shared/_Layout.cshtml *@

<head>
    <!-- ... -->
    @StackExchange.Profiling.MiniProfiler.RenderIncludes()
</head>
```

You should also include the following setting to the `web.config` file, if the `runAllManagedModulesForAllRequests` is set to `false` in your application (it is by default):

```
<!-- ~/web.config -->

<configuration>
  ...
  <system.webServer>
    ...
    <handlers>
      <add name="MiniProfiler" path="mini-profiler-resources/*" verb="*" type="System.
→Web.Routing.UrlRoutingModule" resourceType="Unspecified" preCondition=
→"integratedMode" />
    </handlers>
  </system.webServer>
</configuration>
```

Hiliting the code

It is the core functionality of our application. We'll use the http://hilite.me service that provides HTTP API to perform highlighting work. To start to consume its API, install the `Microsoft.Net.Http` package:

```
Install-Package Microsoft.Net.Http
```

This library provides simple asynchronous API for sending HTTP requests and receiving HTTP responses. So, let's use it to make an HTTP request to the *hilite.me* service:

```
// ~/Controllers/HomeController.cs

/* ... */

public class HomeController
{
    /* ... */

    private static async Task<string> HighlightSourceAsync(string source)
    {
        using (var client = new HttpClient())
        {
            var response = await client.PostAsync(
                @"http://hilite.me/api",
                new FormUrlEncodedContent(new Dictionary<string, string>
                {
```

```
                    { "lexer", "c#" },
                    { "style", "vs" },
                    { "code", source }
                }));

            response.EnsureSuccessStatusCode();

            return await response.Content.ReadAsStringAsync();
        }
    }

    private static string HighlightSource(string source)
    {
        // Microsoft.Net.Http does not provide synchronous API,
        // so we are using wrapper to perform a sync call.
        return RunSync(() => HighlightSourceAsync(source));
    }

    private static TResult RunSync<TResult>(Func<Task<TResult>> func)
    {
        return Task.Run<Task<TResult>>(func).Unwrap().GetAwaiter().GetResult();
    }
}
```

Then, call it inside the `HomeController.Create` method.

```
// ~/Controllers/HomeController.cs

[HttpPost]
public ActionResult Create([Bind(Include = "SourceCode")] CodeSnippet snippet)
{
    try
    {
        if (ModelState.IsValid)
        {
            snippet.CreatedAt = DateTime.UtcNow;

            using (StackExchange.Profiling.MiniProfiler.StepStatic("Service call"))
            {
                snippet.HighlightedCode = HighlightSource(snippet.SourceCode);
                snippet.HighlightedAt = DateTime.UtcNow;
            }

            _db.CodeSnippets.Add(snippet);
            _db.SaveChanges();

            return RedirectToAction("Details", new { id = snippet.Id });
        }
    }
    catch (HttpRequestException)
    {
        ModelState.AddModelError("", "Highlighting service returned error. Try again␣
→later.");
    }

    return View(snippet);
}
```

Note: We are using synchronous controller action method, although it is recommended to use asynchronous one to make network calls inside ASP.NET request handling logic. As written in the given article, asynchronous actions greatly increase application CAPACITY (The maximum throughput a system can sustain, for a given workload, while maintaining an acceptable response time for each individual transaction. – from "Release It" book written by Michael T. Nygard), but does not help to increase PERFORMANCE (How fast the system processes a single transaction. – from "Release It" book written by Michael T. Nygard). You can test it by yourself with a sample application – there are no differences in using sync or async actions with a single request.

This sample is aimed to show you the problems related to application performance. And sync actions are used only to keep the tutorial simple.
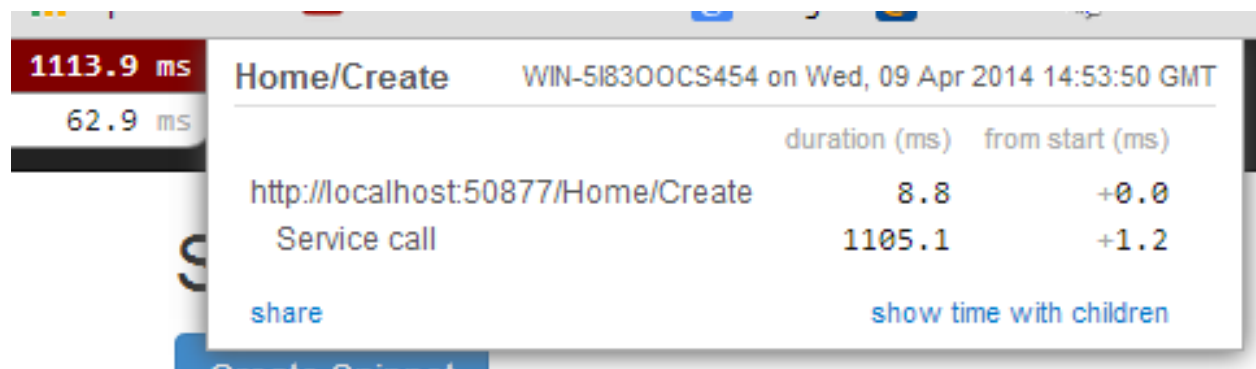
The problem

Tip: You can use the hosted sample to see what's going on.

Now, when the application is ready, try to create some code snippets, starting from a smaller ones. Do you notice a small delay after you clicked the *Create* button?

On my development machine it took about 0.5s to redirect me to the details page. But let's look at *MiniProfiler* to see what is the cause of this delay:
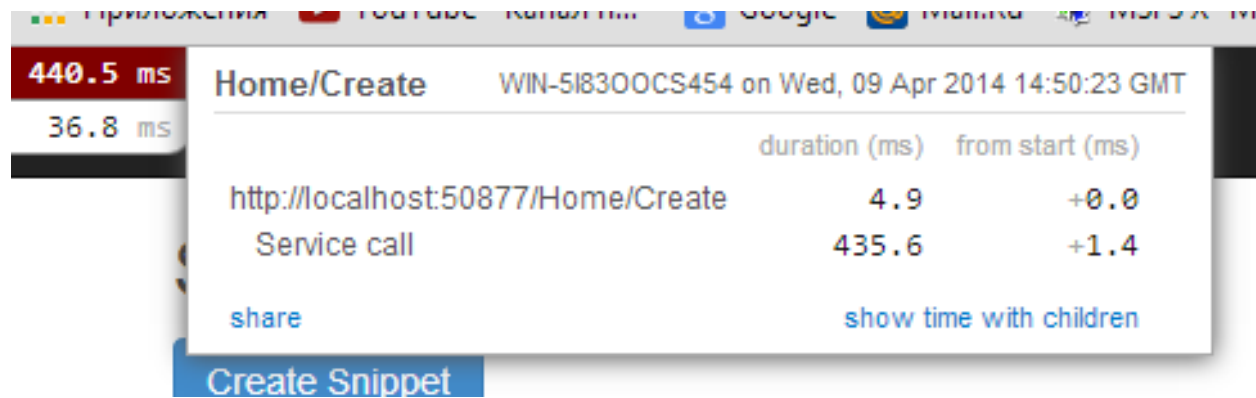
As we see, call to web service is our main problem. But what happens when we try to create a medium code block?

And finally a large one:

The lag is increasing when we enlarge our code snippets. Moreover, consider that syntax highlighting web service (that is not under your control) experiences heavy load, or there are latency problems with network on their side. Or consider heavy CPU-intensive task instead of web service call that you can not optimize well.

Your users will be annoyed with un-responsive application and inadequate delays.

Solving a problem

What can you do with a such problem? Async controller actions will not help, as I said *earlier*. You should somehow take out web service call and process it outside of a request, in the background. Here is some ways to do this:

- **Use recurring tasks** and scan un-highlighted snippets on some interval.
- **Use job queues**. Your application will enqueue a job, and some external worker threads will listen this queue for new jobs.

Ok, great. But there are several difficulties related to these techniques. The former requires us to set some check interval. Shorter interval can abuse our database, longer interval increases latency.

The latter way solves this problem, but brings another ones. Should the queue be persistent? How many workers do you need? How to coordinate them? Where should they work, inside of ASP.NET application or outside, in Windows Service? The last question is the sore spot of long-running requests processing in ASP.NET application:

> **Warning:** **DO NOT** run long-running processes inside of your ASP.NET application, unless they are prepared to **die at any instruction** and there is mechanism that can re-run them.
>
> They will be simple aborted on application shutdown, and can be aborted even if the `IRegisteredObject` interface is used due to time out.

Too many questions? Relax, you can use Hangfire. It is based on *persistent queues* to survive on application restarts, uses *reliable fetching* to handle unexpected thread aborts and contains *coordination logic* to allow multiple worker threads. And it is simple enough to use it.

> **Note:** **YOU CAN** process your long-running jobs with Hangfire inside ASP.NET application – aborted jobs will be restarted automatically.

Installing Hangfire

To install Hangfire, run the following command in the Package Manager Console window:

```
Install-Package Hangfire
```

After the package installed, add or update the OWIN Startup class with the following lines of code.

```csharp
public void Configuration(IAppBuilder app)
{
    GlobalConfiguration.Configuration.UseSqlServerStorage("HighlighterDb");

    app.UseHangfireDashboard();
    app.UseHangfireServer();
}
```

That's all. All database tables will be created automatically on first start-up.

Moving to background

First, we need to define our background job method that will be called when worker thread catches highlighting job. We'll simply define it as a static method inside the `HomeController` class with the `snippetId` parameter.

```csharp
// ~/Controllers/HomeController.cs

/* ... Action methods ... */

// Process a job
public static void HighlightSnippet(int snippetId)
{
    using (var db = new HighlighterDbContext())
    {
        var snippet = db.CodeSnippets.Find(snippetId);
        if (snippet == null) return;

        snippet.HighlightedCode = HighlightSource(snippet.SourceCode);
        snippet.HighlightedAt = DateTime.UtcNow;

        db.SaveChanges();
    }
}
```

Note that it is simple method that does not contain any Hangfire-related functionality. It creates a new instance of the `HighlighterDbContext` class, looks for the desired snippet and makes a call to a web service.

Then, we need to place the invocation of this method on a queue. So, let's modify the `Create` action:

```csharp
// ~/Controllers/HomeController.cs

[HttpPost]
public ActionResult Create([Bind(Include = "SourceCode")] CodeSnippet snippet)
{
    if (ModelState.IsValid)
    {
        snippet.CreatedAt = DateTime.UtcNow;

        _db.CodeSnippets.Add(snippet);
        _db.SaveChanges();

        using (StackExchange.Profiling.MiniProfiler.StepStatic("Job enqueue"))
```

(continues on next page)

```
        {
            // Enqueue a job
            BackgroundJob.Enqueue(() => HighlightSnippet(snippet.Id));
        }

        return RedirectToAction("Details", new { id = snippet.Id });
    }

    return View(snippet);
}
```

That's all. Try to create some snippets and see the timings (don't worry if you see an empty page, I'll cover it a bit later):

Good, 6ms vs ~2s. But there is another problem. Did you notice that sometimes you are redirected to the page with no source code at all? This happens because our view contains the following line:

```
<div>@Html.Raw(Model.HighlightedCode)</div>
```

Why the `Model.HighlightedCode` returns null instead of highlighted code? This happens because of **latency** of the background job invocation – there is some delay before a worker fetch the job and perform it. You can refresh the page and the highlighted code will appear on your screen.

But empty page can confuse a user. What to do? First, you should take this specific into a place. You can reduce the latency to a minimum, but **you can not avoid it**. So, your application should deal with this specific issue.

In our example, we'll simply show the notification to a user and the un-highlighted code, if highlighted one is not available yet:

```
@* ~/Views/Home/Details.cshtml *@

<div>
    @if (Model.HighlightedCode == null)
    {
        <div class="alert alert-info">
            <h4>Highlighted code is not available yet.</h4>
            <p>Don't worry, it will be highlighted even in case of a disaster
                (if we implement failover strategies for our job storage).</p>
            <p><a href="javascript:window.location.reload()">Reload the page</a>
                manually to ensure your code is highlighted.</p>
        </div>

        @Model.SourceCode
    }
```

```
    else
    {
        @Html.Raw(Model.HighlightedCode)
    }
</div>
```

But instead you could poll your application from a page using AJAX until it returns highlighted code:

```
// ~/Controllers/HomeController.cs

public ActionResult HighlightedCode(int snippetId)
{
    var snippet = _db.Snippets.Find(snippetId);
    if (snippet.HighlightedCode == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.NoContent);
    }

    return Content(snippet.HighlightedCode);
}
```

Or you can also use send a command to users via SignalR channel from your `HighlightSnippet` method. But that's another story.

Note: Please, note that user still waits until its source code will be highlighted. But the application itself became more responsive and he is able to do another things while background job is processed.

Conclusion

In this tutorial you've seen that:

- Sometimes you can't avoid long-running methods in ASP.NET applications.

- Long running methods can cause your application to be un-responsible from the users point of view.

- To remove waits you should place your long-running method invocation into background job.

- Background job processing is complex itself, but simple with Hangfire.

- You can process background jobs even inside ASP.NET applications with Hangfire.

Please, ask any questions using the comments form below.

6.9 Upgrade Guides

6.9.1 Upgrading to Hangfire 1.7

For Beta users

If you are upgrading from beta versions, please follow the same steps as described here, and update your configuration only when all your instances already updated to the newest version.

Hangfire 1.7.0 brings a number of new features and great improvements for different aspects of background processing, including increased efficiency and better interoperability. We always consider backward compatibility when introducing new changes to ensure all the existing data can be processed by a newer version.

But during upgrades in distributed environments it's also important to have the forward compatibility property, where older versions can co-exist with the newer ones without causing any troubles. In this case you can perform upgrades gradually, updating instances one-by-one without stopping the whole processing first.

Read the following sections carefully to minimize the risks during the upgrade process, but here are the main points:

1. You should call `SetDataCompatibilityLevel(CompatibilityLevel.Version_170)` and use other new features **only after** all of your servers migrated to the new version. Otherwise you will get only exceptions in the best case, or undefined behavior caused by custom serializer settings.

2. `Schema 6` and `Schema 7` migrations added to SQL Server, and you'll need to perform them either automatically or manually. Hangfire.SqlServer 1.6.23 is forward compatible with those schemas, and 1.7.0 is backward compatible with `Schema 4` (from version 1.5.0) and later.

3. New migrations for SQL Server will not be performed automatically unless `EnableHeavyMigrations` option is set. If your background processing is quite intensive, you should apply the migration manually with setting `SINGLE_USER` mode for the database to avoid deadlocks and reduce migration time.

If you have any issues with an upgrade process, please post your thoughts to GitHub Issues.

Data Compatibility

To allow further development without sacrificing forward compatibility, a new concept was added to version 1.7 – *Data Compatibility Level* that defines the format of the data that is written to a storage. It can be specified by calling the `IGlobalConfiguration.SetDataCompatibilityLevel` method and provides two options: `CompatibilityLevel.Version_110` (default value, every version starting from 1.1.0 understands it) and `CompatibilityLevel.Version_170`.

The latest compatibility level contains the following changes:

- Background job payload is serialized using a more compact format.

- Serialization of internal data is performed with `TypeNameHandling.Auto`, `TypeNameAssemblyFormat.Simple`, `DefaultValueHandling.IgnoreAndPopulate` and `NullValueHandling.Ignore` settings that can't be affected by user settings set by `UseSerializerSettings` method and even by custom `JsonConvert.DefaultOptions`.

- `DateTime` arguments are serialized using regular JSON serializer, instead of `DateTime.ToString("o")` method.

Backward Compatibility

Hangfire.Core, Hangfire.SqlServer:

New version can successfully process background jobs created with both `Version_110` and `Version_170` data compatibility levels. However if you change the `UseSerializerSettings` with incompatible options, the resulting behavior is undefined.

Hangfire.SqlServer:

All queries are backward compatible even with `Schema 5` from versions 1.6.X, so you can run the schema migration manually after some time, for example during off-hours.

Forward Compatibility

Warning

No new features or configuration options, except those mentioned in upgrade steps below, should be used to preserve the forward compatibility property.

Hangfire.Core, Hangfire.SqlServer:

Forward compatibility is supported on the `CompatibilityLevel.Version_110` (default value) data compatibility level. In this case no data in the new format will be created in the storage, and servers of previous versions will be possible to handle the new data.

Hangfire.SqlServer:

Hangfire.SqlServer 1.6.23 **is forward compatible** with `Schema 6` and `Schema 7` schemas. Previous versions don't support the new schemas and may lead to exceptions. Anyway it's better to upgrade all your servers first, and only then apply the migration.

Code Compatibility

Breaking Changes in API

Unfortunately there's need to update your code if you are using one of the following features during upgrade to the newest version. I understand such changes are not welcome when migrating between minor versions, but all of them are required to fix problems. These changes cover only the low level API surface and don't relate to background jobs.

Hangfire.Core:

- Added `IBackgroundJobFactory.StateMachine` property to enable transactional behavior of `RecurringJobScheduler`.
- Changed the way of creating recurring job. `CreatingContext.InitialState` and `CreatedContext.InitialState` properties for `IClientFilter` implementations will return `null` now, instead of an actual value. Use `IApplyStateFilter` or `IElectStateFilter` to access that value.

Hangfire.AspNetCore:

- Removed registrations for `IBackgroundJobFactory`, `IBackgroundJobPerformer` and `IBackgroundJobStateChanger` interfaces. Custom implementations of these interfaces now applied only if **all of them** are registered. Previously it was unclear what `JobActivator` is used – from registered service or from options, and lead to errors.

Hangfire.SqlServer:

- `IPersistentJobQueueMonitoringApi.Get**JobIds` methods now return `IEnumerable<long>`. If you are using non-default persistent queue implementations, upgrade those packages as well. This change is required to handle bigger identifier format.

Breaking Changes in Code

There are no breaking changes for your background jobs in this release, unless you explicitly changed the following configuration options.

- `IGlobalConfiguration.UseRecommendedSerializerSettings` (disabled by default) may affect argument serialization and may be incompatible with your current JSON settings if you've changed them using the `JobHelper.SetSerializerSettings` method or `DefaultValueAttribute` on your argument classes or different date/time formats.

- Setting `BackgroundJobServerOptions.TaskScheduler` to `null` (`TaskScheduler.Default` is used by default) will force async continuations to be processed by the worker thread itself, reducing the number of required threads (that's good). But if you are using non-recommended and dangerous `Task.Result` or `Task.GetAwaiter().GetResult()` methods, your async background jobs can be deadlocked.

Upgrade Steps

Steps related to the Hangfire.SqlServer package are optional

This guide covers upgrade details also for the `Hangfire.SqlServer` package, because its versioning scheme is closely related to the `Hangfire.Core` package. If you are using another storage, simply skip information related to SQL Server, because nothing is changed for other storages in this release.

1. Upgrading Packages

First upgrade all the packages without touching any new configuration and/or new features. Then deploy your application with the new version until all your servers are successfully migrated to the newer version. 1.6.X and 1.7.0 servers can co-exist in the same environment just fine, thanks to forward compatibility.

a. Upgrade your NuGet package references using your own preferred way. If you've referenced Hangfire using a single meta-package, just upgrade it:

```xml
<PackageReference Include="Hangfire" Version="1.7.*" />
```

If you reference individual packages upgrade them all, here is the full list of packages that come with this release. Please note that versions in the code snippet below may be outdated, so use versions from the following badges, they are updated in real-time.

```xml
<PackageReference Include="Hangfire.Core" Version="1.7.*" />
<PackageReference Include="Hangfire.AspNetCore" Version="1.7.*" />
<PackageReference Include="Hangfire.SqlServer" Version="1.7.*" />
<PackageReference Include="Hangfire.SqlServer.Msmq" Version="1.7.*" />
```

b. Fix breaking changes mentioned in the previous section if they apply to your use case.

c. **Optional.** If your background processing sits mostly idle and you are already using Hangfire 1.6.23, you can run the schema migration for SQL Server during this step. Otherwise I'd highly encourage you to perform the migration manually as written in the following section, because it may take too long if there are outstanding queries.

```
GlobalConfiguration.Configuration.UseSqlServerStorage("connection_string", new␣
→SqlServerStorageOptions
{
    CommandBatchMaxTimeout = TimeSpan.FromMinutes(5),
    QueuePollInterval = TimeSpan.Zero,
```

(continues on next page)

```
        SlidingInvisibilityTimeout = TimeSpan.FromMinutes(5),
        UseRecommendedIsolationLevel = true,
        PrepareSchemaIfNecessary = true, // Default value: true
        EnableHeavyMigrations = true     // Default value: false
});
```

d. Set the `StopTimeout` for your background processing servers to give your background jobs some time to be processed during the shutdown event, instead of instantly aborting them.

```
new BackgroundJobServerOptions
{
    StopTimeout = TimeSpan.FromSeconds(10)
}
```

2. Migrating the Schema

Schema migration can be postponed to off-hours

Hangfire.SqlServer 1.7 package can talk with all schemas, starting from `Schema 4` from version 1.5.0, so you can wait for some time before applying the new ones.

`Schema 6` and `Schema 7` migrations that come with the new `Hangfire.SqlServer` package version will not be applied automatically, unless you set the `EnableHeavyMigrations` options as written above. This option was added to prevent uncontrolled upgrades that may lead to long downtime or deadlocks when applied in processing-heavy environments or during the peak load.

To perform the manual upgrade, obtain the DefaultInstall.sql migration script from the repository and wrap it with the lines below to reduce the migration downtime. Please note this will abort all the current transactions and prevent new ones from starting until the upgrade is complete, so it's better to do it during off-hours.

```
ALTER DATABASE [HangfireDB] SET SINGLE_USER WITH ROLLBACK IMMEDIATE;

-- DefaultInstall.sql / Install.sql contents

ALTER DATABASE [HangfireDB] SET MULTI_USER;
```

If you are using non-default schema, please get the Install.sql file instead and replace all the occurrences of the `$(HangFireSchema)` token with your schema name without brackets.

3. Updating Configuration

Ensure all your processing servers upgraded to 1.7

Before performing this step, ensure all your processing servers successfully migrated to the new version. Otherwise you may get exceptions or even undefined behavior, caused by custom JSON serialization settings.

When all your servers can understand the new features, you can safely enable them. The new version understands all the existing jobs even in previous data format, thanks to backward compatibility. All these settings are recommended, but **optional** – you can use whatever you have currently.

a. Set the new data compatibility level and type serializer to have more compact payloads for background jobs.

```
GlobalConfiguration.Configuration
    // ...
    .SetDataCompatibilityLevel(CompatibilityLevel.Version_170)
    .UseSimpleAssemblyNameTypeSerializer();
```

b. If you don't use custom JSON settings before by calling `JobHelper.SetSerializerSettings` or by using `JsonConvert.DefaultOption` or by using attributes on your job argument classes, you can set the recommended JSON options that lead to more compact payloads. **Otherwise you can get breaking changes.**

```
GlobalConfiguration.Configuration
    // ...
    .UseRecommendedSerializerSettings();
```

If you do use custom settings, you can call the `UseSerializerSettings` method instead:

```
GlobalConfiguration.Configuration
    // ...
    .UseSerializerSettings(new JsonSerializerSettings { /* ... */ });
```

c. Update SQL Server options to have better locking scheme, more efficient dequeue when using Sliding Invisibility Timeout technique and disable heavy migrations in future to prevent accidental deadlocks.

```
GlobalConfiguration.Configuration
    // ...
    .UseSqlServerStorage("connection_string", new SqlServerStorageOptions
    {
        // ...
        DisableGlobalLocks = true,    // Migration to Schema 7 is required
        EnableHeavyMigrations = false // Default value: false
    });
```

After setting new configuration options, deploy the changes to your servers when needed.