HangFire Documentation

Release 1.0

Sergey Odinokov

August 27, 2015

Contents

Incredibly easy way to perform **fire-and-forget**, **delayed** and **recurring jobs** inside **ASP.NET applications**. No Windows Service / Task Scheduler required. Backed by Redis, SQL Server, SQL Azure or MSMQ.

Hangfire provides unified programming model to handle background tasks in a **reliable way** and run them on shared hosting, dedicated hosting or in cloud. You can start with a simple setup and grow computational power for background jobs with time for these scenarios:

- mass notifications/newsletter;
- batch import from xml, csv, json;
- creation of archives;
- firing off web hooks;
- deleting users;
- building different graphs;
- image/video processing;
- purge temporary files;
- recurring automated reports;
- database maintenance;
- *. . . and so on.*

Hangfire is a .NET Framework alternative to Sidekiq, Resque, delayed_job.

CHAPTER 1

Contents

1.1 Project pages

Site	http://hangfire.io
Documentation	http://docs.hangfire.io
Forum	http://discuss.hangfire.io
NuGet Packages	https://www.nuget.org/packages?q=hangfire
Development	https://github.com/HangfireIO/Hangfire
Issues	https://github.com/HangfireIO/Hangfire/issues
Release Notes	https://github.com/HangfireIO/Hangfire/releases

1.2 Quick start

1.2.1 Installation

There are a couple of packages for Hangfire available on NuGet. To install Hangfire into your **ASP.NET application** with **SQL Server** storage, type the following command into the Package Manager Console window:

```
PM> Install-Package Hangfire
```

1.2.2 Configuration

After installing the package, add or update the OWIN Startup class with the following lines:

```
public void Configure(IAppBuilder app)
{
    app.UseHangfire(config =>
    {
        config.UseSqlServerStorage("<connection string or its name>");
        config.UseServer();
    });
}
```

Then open the Hangfire Dashboard to test your configuration. Please, build the project and open the following URL in a browser:

1.2.3 Usage

Add a job. . .

Hangfire handles different types of background jobs, and all of them are invoked on a separate execution context.

Fire-and-forget

This is the main background job type, persistent message queues are used to handle it. Once you create a fire-and-forget job, it is being saved to its queue (`"default"` by default, but multiple queues supported). The queue is listened by a couple of dedicated workers that fetch a job and perform it.

```
BackgroundJob.Enqueue(() => Console.WriteLine("Fire-and-forget"));
```

Delayed method invocation

If you want to delay the method invocation for a certain type, call the following method. After the given delay the job will be put to its queue and invoked as a regular fire-and-forget job.

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

Recurring tasks

To call a method on a recurrent basis (hourly, daily, etc), use the `RecurringJob` class. You are able to specify the schedule using CRON expressions to handle more complex scenarios.

```
RecurringJob.AddOrUpdate(() => Console.Write("Recurring"), Cron.Daily);
```

. . . and relax

Hangfire saves your jobs into persistent storage and processes them in a reliable way. It means that you can abort Hangfire worker threads, unload application domain or even terminate the process, and your jobs will be processed anyway [1]. Hangfire flags your job as completed only when the last line of your code was performed, and knows that the job can fail before this last line. It contains different auto-retrying facilities, that can handle either storage errors or errors inside your code.

This is very important for generic hosting environment, such as IIS Server. They can contain different optimizations, timeouts and error-handling code (that may cause process termination) to prevent bad things to happen. If you are not using the reliable processing and auto-retrying, your job can be lost. And your end user may wait for its email, report, notification, etc. indefinitely.

1.3 Features

1.3.1 Queue-based processing

Instead of invoking a method synchronously, place it on a persistent queue, and Hangfire worker thread will take it and perform within its own execution context:

[1] But when your storage becomes broken, Hangfire can not do anything. Please, use different failover strategies for your storage to guarantee the processing of each job in case of a disaster.

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

This method creates a job in the storage and immediately returns control to the caller. Hangfire guarantees that the specified method will be called even after the abnormal termination of the host process.

1.3.2 Delayed method invocation

Instead of invoking a method right now, you can postpone its execution for a specified time:

```
BackgroundJob.Schedule(() => Console.WriteLine("Hello, world!"), TimeSpan.FromMinutes(5));
```

This call also saves a job, but instead of placing it to a queue, it adds the job to a persistent schedule. When the given time elapsed, the job will be added to its queue. Meanwhile you can restart your application – it will be executed anyway.

1.3.3 Recurring tasks

Recurring job processing was never been easier. All you need is to call a single line of code:

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

Hangfire uses NCrontab library to perform scheduling tasks, so you can use more complex CRON expressions:

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

1.3.4 Integrated web interface

Web interface will help you to track the execution of your jobs. See their processing state, watch the statistics. Look at screenshots on http://hangfire.io, and you'll love it.

1.3.5 SQL Server and Redis support

Hangfire uses persistent storage to store jobs, queues and statistics and let them survive application restarts. The storage subsystem is abstracted enough to support both classic SQL Server and fast Redis.

- SQL Server provides simplified installation together with usual maintenance plans.
- Redis provides awesome speed, especially comparing to SQL Server, but requires additional knowledge.

1.3.6 Automatic retries

If your method encounters a transient exception, don't worry – it will be retried automatically in a few seconds. If all retry attempts are exhausted, you are able to restart it manually from integrated web interface.

You can also control the retry behavior with the `AutomaticRetryAttribute` class. Just apply it to your method to tell Hangfire the number of retry attempts:

```
[AutomaticRetry(100)]
public static void GenerateStatistics() { }

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

1.3.7 Guaranteed processing

Hangfire was made with the knowledge that the hosting environment can kill all the threads on each line. So, it does not remove the job until it was successfully completed and contains different implicit retry logic to do the job when its processing was aborted.

1.3.8 Instance method calls

All the examples above uses static method invocation, but instance methods are supported as well:

```
public class EmailService
{
    public void Send() { }
}

BackgroundJob.Enqueue<EmailService>(x => x.Send());
```

When a worker sees that the given method is an instance-method, it will activate its class first. By default, the `Activator.CreateInstace` method is being used, so only classes with default constructors are supported by default. But you can plug in your IoC container and pass the dependencies through the constructor.

1.3.9 Culture capturing

When you marshal your method invocation into another execution context, you should be able to preserve some environment settings. Some of them – `Thread.CurrentCulture` and `Thread.CurrentUICulture` are automatically being captured for you.

It is done by the `PreserveCultureAttribute` class that is applied to all of your methods by default.

1.4 Advanced Features

1.4.1 Cancellation tokens

Hangfire can tell your methods were aborted or canceled due to shutdown event, so you can stop them gracefully using job cancellation tokens that are similar to the regular `CancellationToken` class.

```
public void Method(IJobCancellationToken token)
{
    for (var i = 0; i < Int32.MaxValue; i++)
    {
        token.ThrowIfCancellationRequested();
        Thread.Sleep(1000);
    }
}
```

1.4.2 IoC Containers

In case you want to improve the testability of your job classes or simply don't want to use a huge amount of different factories, you should use instance methods instead of static ones. But you either need to somehow pass the dependencies into these methods and the default job activator does not support parameterful constructors.

Don’t worry, you can use your favourite IoC container that will instantiate your classes. There are two packages, `Hangfire.Ninject` and `Hangfire.Autofac` for their respective containers. If you are using another container, please, write it yourself (on a basis of the given packages) and contribute to Hangfire project.

1.4.3 Logging

Hangfire uses the `Common.Logging` library to log all its events. It is a generic library and you can plug it to your logging framework using adapters. Please, see the list of available adapters on NuGet Gallery.

1.4.4 Web Garden and Web Farm friendly

You can run multiple Hangfire instances, either on the same or different machines. It uses distributed locking to prevent race conditions. Each Hangfire instance is redundant, and you can add or remove instances seamlessly (but control the queues they listen).

1.4.5 Multiple queues processing

Hangfire can process multiple queues. If you want to prioritize your jobs or split the processing across your servers (some processes the `archive` queue, others – 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("critical")]
public void SomeMethod() { }

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

To start to process multiple queues, you need to update your OWIN bootstrapper’s configuration action:

```
app.UseHangfire(config =>
{
    config.UseServer("critical", "default");
});
```

The order is important, workers will fetch jobs from the `critical` queue first, and then from the `default` queue.

1.4.6 Concurrency level control

Hangfire uses its own fixed worker thread pool to consume queued jobs. Default worker count is set to `Environment.ProcessorCount * 5`. This number is optimized both for CPU-intensive and I/O intensive tasks. If you experience excessive waits or context switches, you can configure amount of workers manually:

```
app.UseHangfire(config =>
{
    config.UseServer(100);
});

// or
var server = new BackgroundJobServer(100);
```

1.4.7 Process jobs anywhere

By default, the job processing is made within an ASP.NET application. But you can process jobs either in the console application, Windows Service or anywhere else.

1.4.8 Extensibility

Hangfire is build to be as generic as possible. You can extend the following parts:

- storage implementation;
- states subsystem (including the creation of new states);
- job creation process;
- job performance process;
- state changing process;
- job activation process.

Some of core components are made as extensions: `QueueAttribute`, `PreserveCultureAttribute`, `AutomaticRetryAttribute`, `SqlServerStorage`, `RedisStorage`, `NinjectJobActivator`, `AutofacJobActivator`, `ScheduledState`.

1.5 Tutorials

1.5.1 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.

```
// ~/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
```

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

```
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 sent email notification from the `Create` controller action:

```
[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):

```
<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 does 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 or update the OWIN Startup class as written here.

```
public void Configure(IAppBuilder app)
{
    app.UseHangfire(config =>
    {
        app.UseSqlServerStorage("MailerDb");
        app.UseServer();
    });
}
```

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).

```
Install-Package RazorEngine
```

```
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();
    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 being 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):

```
<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(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 = LogManager.GetCurrentClassLogger();

    public void OnStateApplied(ApplyStateContext context, IWriteOnlyTransaction transaction)
    {
        var failedState = context.NewState as FailedState;
        if (failedState != null)
        {
            Logger.Error(
                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:

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

Or locally by applying the attribute to a method:

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

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:
var 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}", currentCultureNar
    }
    // ...
```

1.5.2 Highlighter Tutorial

Simple sample	https://github.com/odinserj/Hangfire.Highlighter
Full sample	http://highlighter.hangfire.io, sources

Table of Contents

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 are also want it to work even if a user disabled JavaScript in her browser.

To support this scenario and to reduce the project development time, you choosed to use a web service for syntax highlighting, such as http://pygments.appspot.com 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

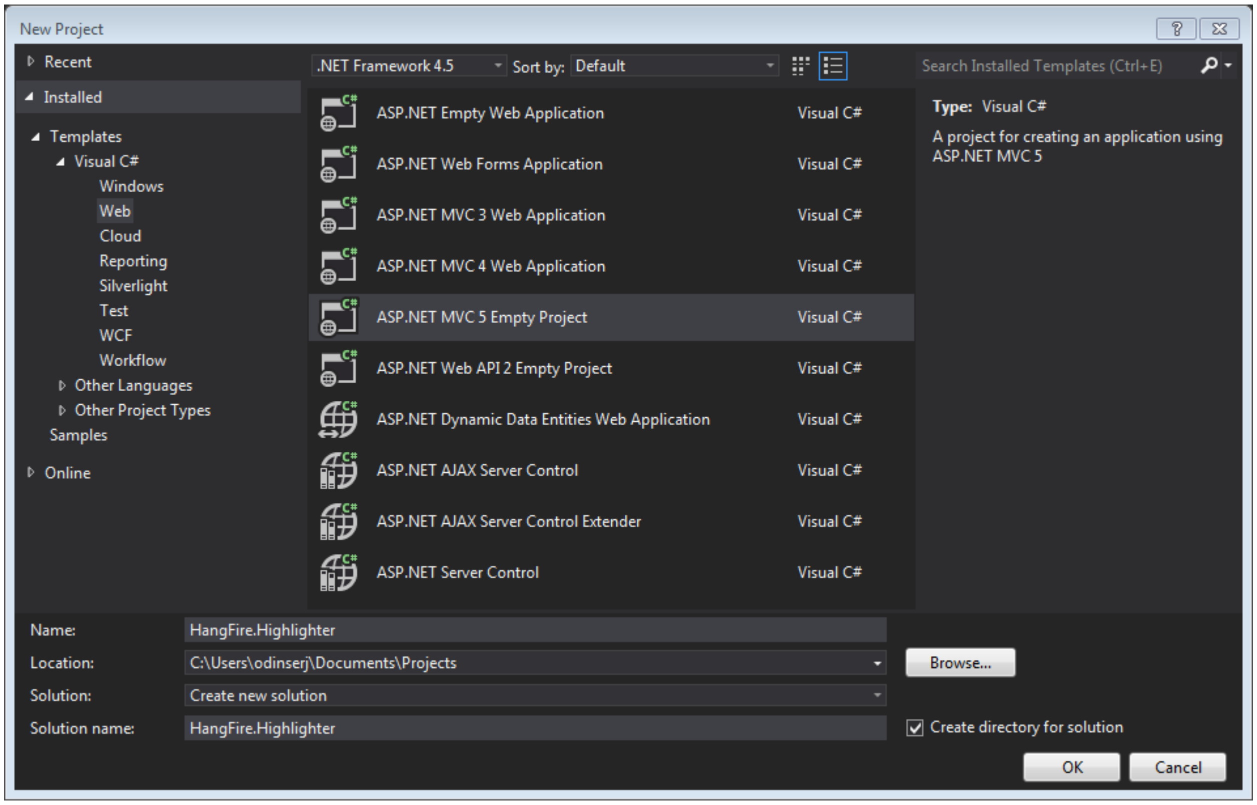
The tutorial uses **Visual Studio 2012** with Web Tools 2013 for Visual Studio 2012 installed, but it can be built either with Visual Studio 2013.

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:



ASP.NET Empty Web Application	Visual C#
ASP.NET Web Forms Application	Visual C#
ASP.NET MVC 3 Web Application	Visual C#
ASP.NET MVC 4 Web Application	Visual C#
ASP.NET MVC 5 Empty Project	Visual C#
ASP.NET Web API 2 Empty Project	Visual C#
ASP.NET Dynamic Data Entities Web Application	Visual C#
ASP.NET AJAX Server Control	Visual C#
ASP.NET AJAX Server Control Extender	Visual C#
ASP.NET Server Control	Visual C#

Type: Visual C#

A project for creating an application using ASP.NET MVC 5

Name: HangFire.Highlighter

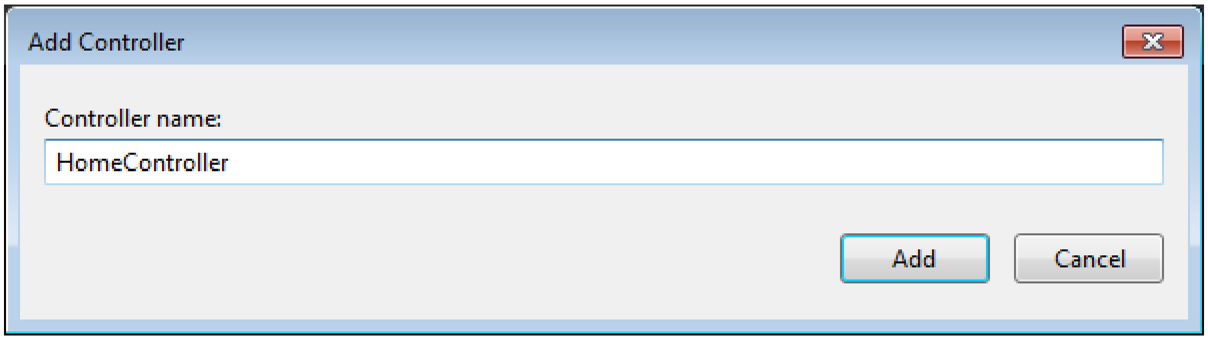
Location: C:\Users\odinserj\Documents\Projects

Solution: Create new solution

Solution name: HangFire.Highlighter

✓ Create directory for solution

Then, we need a controller to handle web requests. So scaffold an **MVC 5 Controller - Empty** controller and call it `HomeController`:

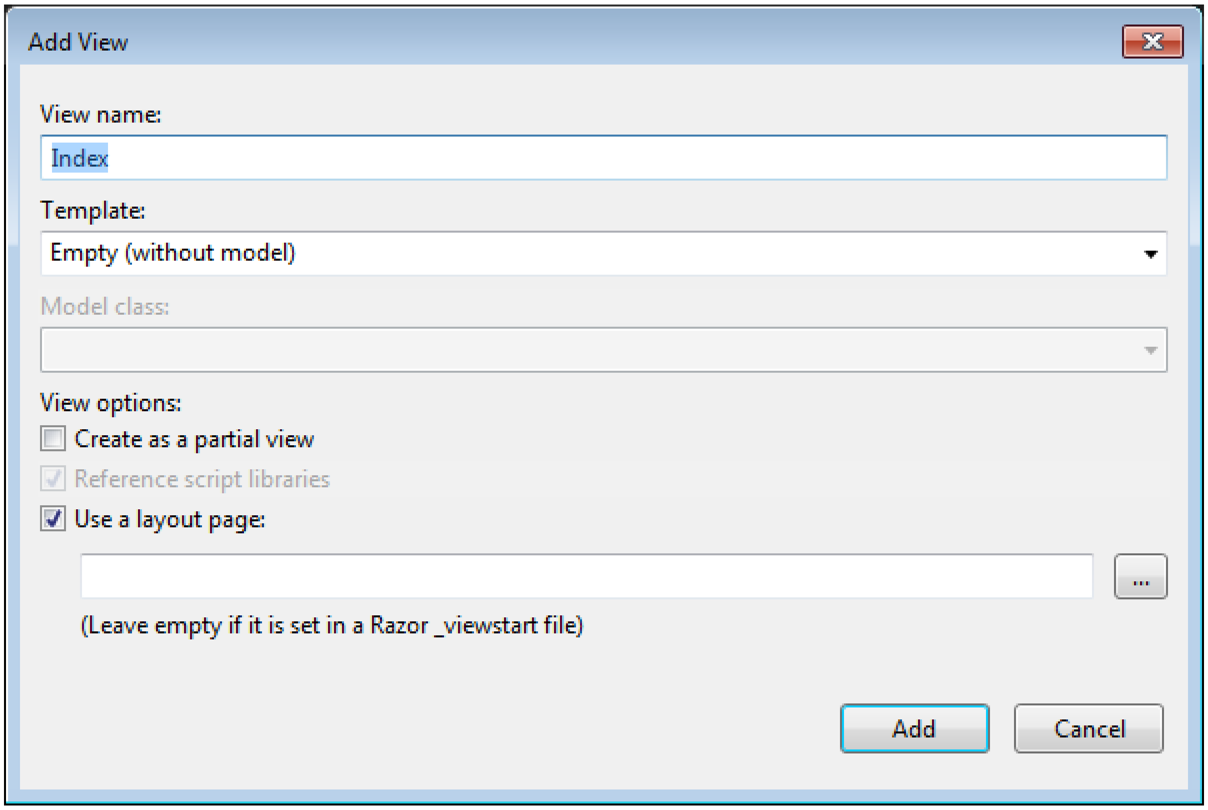


Controller name: HomeController

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.



View name: Index

Template: Empty (without model)

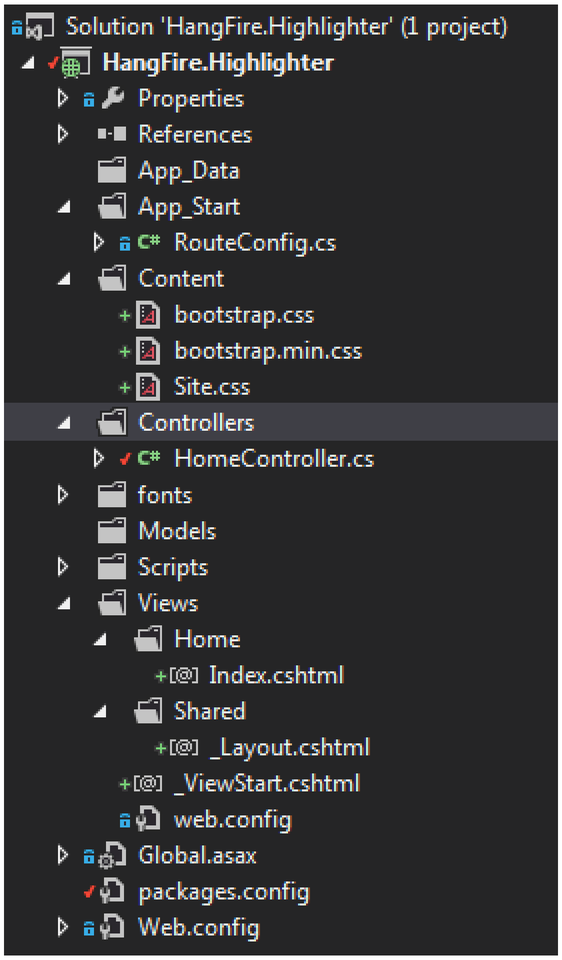
Model class:

View options:

- [ ] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

The view scaffolding process also adds additional components to the project, like *Bootstrap*, *jQuery*, etc. After these steps my solution looks like:


```
Solution 'HangFire.Highlighter' (1 project)
  HangFire.Highlighter
    Properties
    References
    App_Data
    App_Start
      RouteConfig.cs
    Content
      bootstrap.css
      bootstrap.min.css
      Site.css
    Controllers
      HomeController.cs
    fonts
    Models
    Scripts
    Views
      Home
        Index.cshtml
      Shared
        _Layout.cshtml
      _ViewStart.cshtml
      web.config
    Global.asax
    packages.config
    Web.config
```

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`:

```
// ~/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:

```
<connectionStrings>
  <add name="HighlighterDb" connectionString="Server=.\sqlexpress; Database=Hangfire.Highlighter; Tru
</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:

```
// ~/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:

```
// ~/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.

```
// ~/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);
                _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
            </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-heig
    </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.Url
    </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.

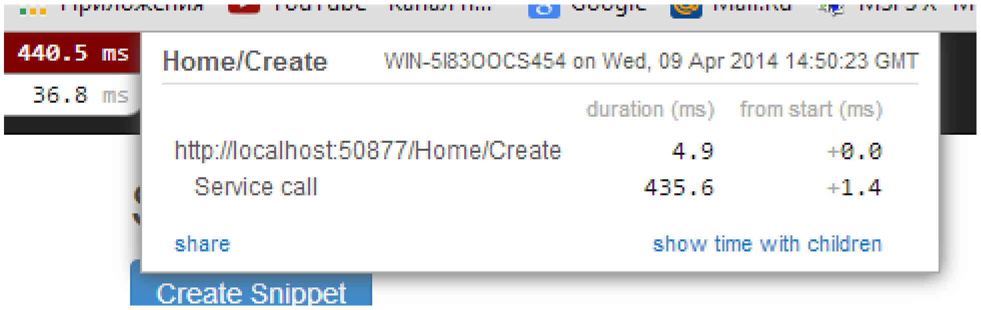
This sample is aimed to show you the problems related to application performance. And sync actions are used only to keep the tutorial simple.

The problem

Tip: You can use the hosted sample to see what's going on.

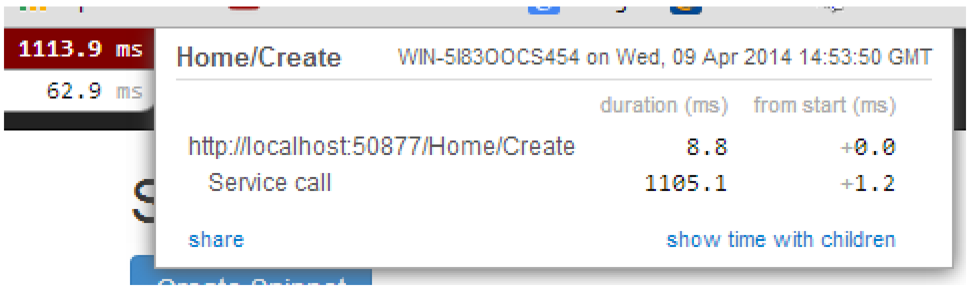
Now, when the application is ready, try to create some code snippets, starting from a smaller ones. Do you notice a small delay after you clicked the *Create* button?

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:



```
440.5 ms
 36.8 ms

Home/Create          WIN-5I83OOCS454 on Wed, 09 Apr 2014 14:50:23 GMT

                                      duration (ms)   from start (ms)
http://localhost:50877/Home/Create          4.9            +0.0
  Service call                            435.6            +1.4
```

As we see, call to web service is our main problem. But what happens when we try to create a medium code block?

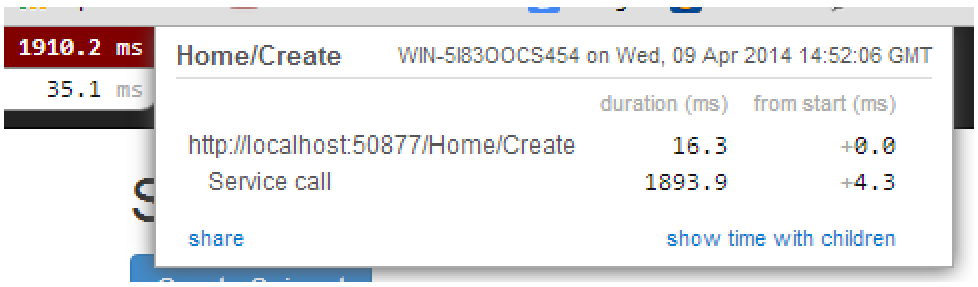


1113.9 ms
62.9 ms

Home/Create WIN-5I83OOCS454 on Wed, 09 Apr 2014 14:53:50 GMT

	duration (ms)	from start (ms)
http://localhost:50877/Home/Create	8.8	+0.0
Service call	1105.1	+1.2

And finally a large one:



1910.2 ms
35.1 ms

Home/Create WIN-5I83OOCS454 on Wed, 09 Apr 2014 14:52:06 GMT

	duration (ms)	from start (ms)
http://localhost:50877/Home/Create	16.3	+0.0
Service call	1893.9	+4.3

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 unresponsible 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 being 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 as `written here` with the following lines of code.

```
public void Configure(IAppBuilder app)
{
    app.UseHangfire(config =>
    {
        config.UseSqlServerStorage("HighlighterDb");
        config.UseServer();
    });
}
```

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.

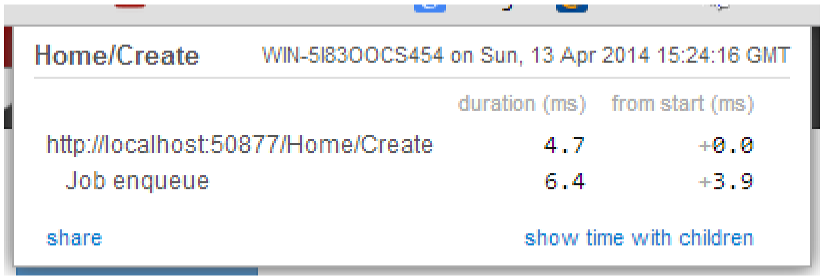
```
// ~/Controllers/HomeController.cs

/* ... 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:

```
// ~/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"))
        {
            // 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):



```
Home/Create                      WIN-5I83OOCS454 on Sun, 13 Apr 2014 15:24:16 GMT

                                   duration (ms)   from start (ms)
http://localhost:50877/Home/Create           4.7              +0.0
  Job enqueue                                6.4              +3.9

share                                          show time with children
```

Good, 6ms vs ~2s. But there is another problem. Did you notice that sometimes you are being 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 responsible and he is able to do another things while background job is being 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.

1.6 User's Guide

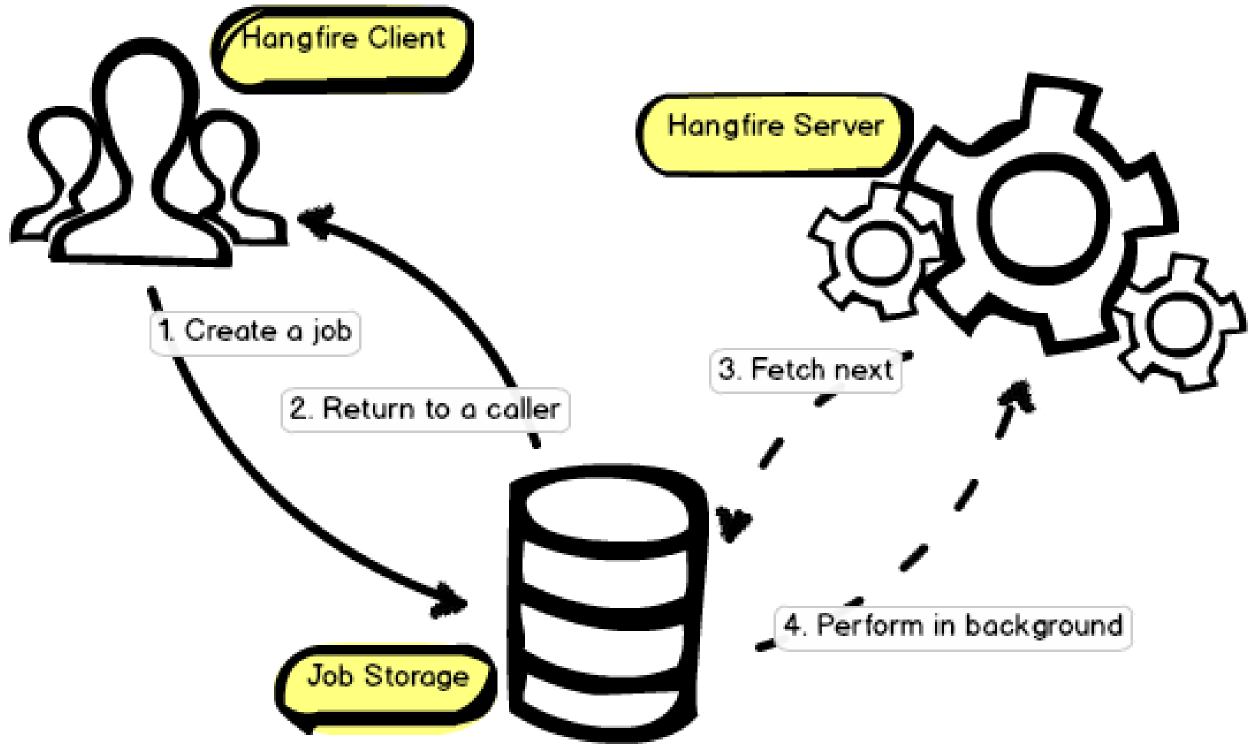
Warning: User's Guide is still under construction. Please, do not link to inner sections (link to the guide itself), since the structure can be changed.

1.6.1 Getting started

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 being performed in a *background thread* and called *background jobs*.

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



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

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.

```
JobStorage.Current = new SqlServerStorage("db_connection");
```

Server

Background jobs are being 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:

```
var server = new BackgroundJobServer();
server.Start();
```

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.

Installation

Hangfire project consists of a couple of NuGet packages available on NuGet Gallery site. Here is the list of basic packages you should know about:

- Hangfire – bootstrapper package that is intended to be installed **only** for ASP.NET applications that uses SQL Server as a job storage. It simply references to Hangfire.Core, Hangfire.SqlServer and Microsoft.Owin.Host.SystemWeb packages.
- Hangfire.Core – basic package that contains all core components of Hangfire. It can be used in any project type, including ASP.NET application, Windows Service, Console, any OWIN-compatible web application, Azure Worker Role, etc.

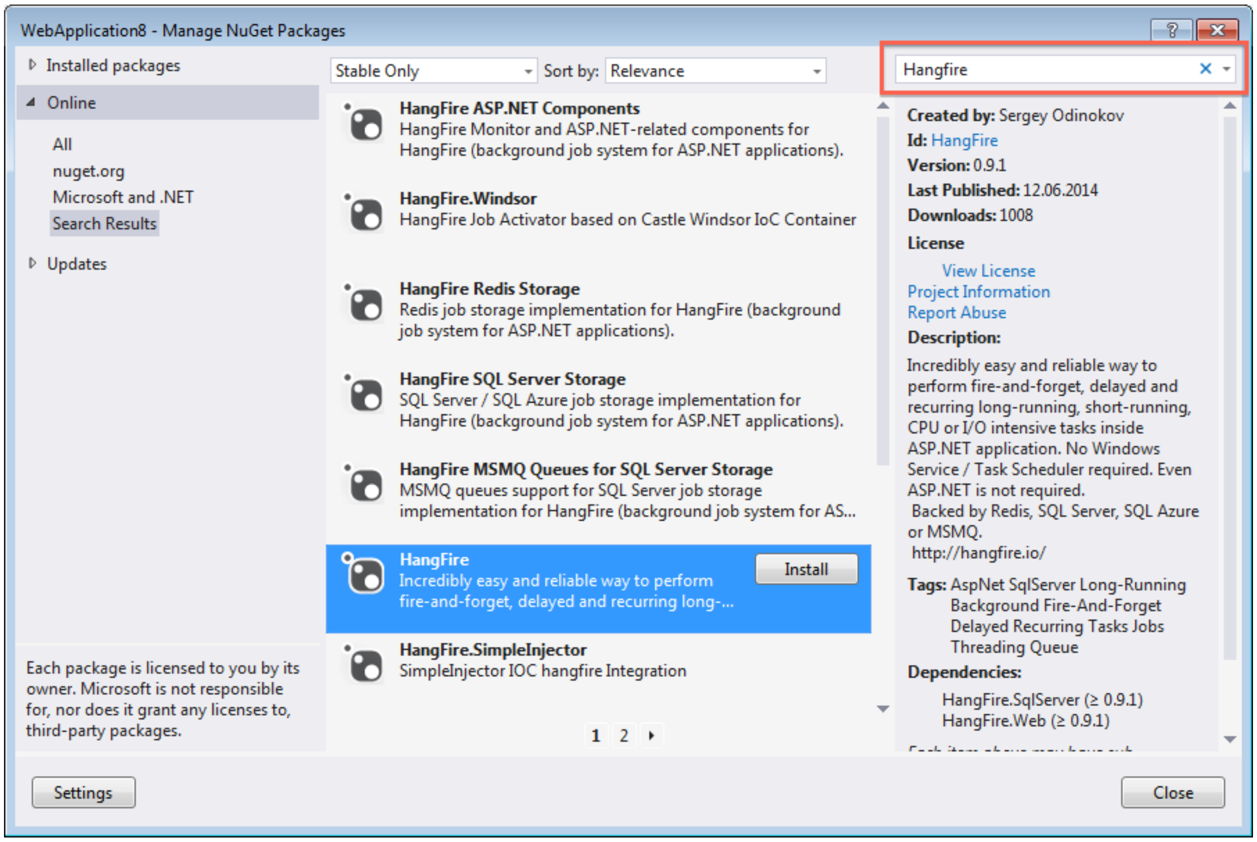
Warning: If you are using custom installation within a web application hosted in IIS, do not forget to install the Microsoft.Owin.Host.SystemWeb package. Otherwise some features, like graceful shutdown may not work.

Using Package Manager Console

```
PM> Install-Package Hangfire
```

Using NuGet Package Manager

Right-click on your project in Visual Studio and choose the `Manage NuGet Packages` menu item. Search for `Hangfire` and install the choosed package:



Hangfire

HangFire ASP.NET Components
HangFire Monitor and ASP.NET-related components for HangFire (background job system for ASP.NET applications).

HangFire.Windsor
HangFire Job Activator based on Castle Windsor IoC Container

HangFire Redis Storage
Redis job storage implementation for HangFire (background job system for ASP.NET applications).

HangFire SQL Server Storage
SQL Server / SQL Azure job storage implementation for HangFire (background job system for ASP.NET applications).

HangFire MSMQ Queues for SQL Server Storage
MSMQ queues support for SQL Server job storage implementation for HangFire (background job system for AS...

HangFire
Incredibly easy and reliable way to perform fire-and-forget, delayed and recurring long-...

HangFire.SimpleInjector
SimpleInjector IOC hangfire Integration

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Created by: Sergey Odinokov
Id: HangFire
Version: 0.9.1
Last Published: 12.06.2014
Downloads: 1008
License
View License
Project Information
Report Abuse
Description:
Incredibly easy and reliable way to perform fire-and-forget, delayed and recurring long-running, short-running, CPU or I/O intensive tasks inside ASP.NET application. No Windows Service / Task Scheduler required. Even ASP.NET is not required.
Backed by Redis, SQL Server, SQL Azure or MSMQ.
http://hangfire.io/
Tags: AspNet SqlServer Long-Running Background Fire-And-Forget Delayed Recurring Tasks Jobs Threading Queue
Dependencies:
HangFire.SqlServer (≥ 0.9.1)
HangFire.Web (≥ 0.9.1)

OWIN bootstrapper

In OWIN based web application frameworks, such as ASP.NET MVC, FubuMVC, Nancy, ServiceStack and many others, you can use OWIN bootstrapper methods to simplify the configuration task.

Adding OWIN Startup class

Note: If your project already have the OWIN Startup class (for example if you have SignalR installed), go to the next section.

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:

```
using HangFire;
using HangFire.SqlServer;
using Microsoft.Owin;
using Owin;

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

namespace MyWebApplication
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            /* configuration goes here */
        }
    }
}
```

Configuring Hangfire

Hangfire provides an extension method for the `IAppBuilder` interface called `UseHangfire` – an entry point to the configuration. Storage, Job activator, Authorization filters, Job filters can be configured here, check the available methods through the intellisence. Job storage is the only required configuration option, all others are optional.

Note: Prefer to use the `UseServer` method over manual `BackgroundJobServer` instantiation to process background jobs inside a web application. The method registers a handler of the application's shutdown event to perform the graceful shutdown for your jobs.

```
public void Configuration(IAppBuilder app)
{
    app.UseHangfire(config =>
    {
        // Basic setup required to process background jobs.
        config.UseSqlServerStorage("<your connection string or its name>");
        config.UseServer();
    });
}
```

The order of `Use*` methods inside the configuration action is not important, all configuration logic is being performed after all calls to these methods. The `UseHangfire` method also registers the *Hangfire Dashboard* middleware at the `http://<your-app>/hangfire` default url (but you can change it).

1.6.2 Storage configuration

Using SQL Server

Note: Supported database engines: **Microsoft SQL Server 2008R2** (any edition) and later, **Microsoft SQL Azure**.

SQL Server is the default storage for Hangfire – it is well known to many .NET developers and being 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.

Configuration

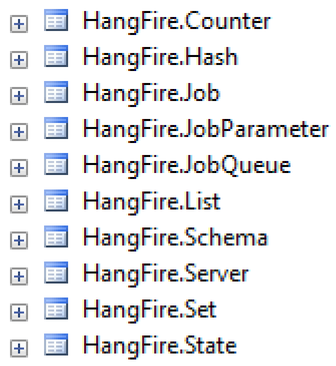
The package provides an extension method for the Hangfire's OWIN bootstrapper method. Choose either a connection string to your SQL Server or a connection string name, if you have it.

```
app.UseHangfire(config =>
{
    // Connection string defined in web.config
    config.UseSqlServerStorage("db_connection");

    // Define the conenction string
    config.UseSqlServerStorage("Server=.\\sqlexpress; Database=Hangfire; Integrated Security=SSPI;");
});
```

If you want to use Hangfire outside of web application, where OWIN Startup class is not applicable, create an instance of the `SqlServerStorage` manually and pass it to the `JobStorage.Current` static property. Parameters are the same.

```
JobStorage.Current = new SqlServerStorage("connection string or its name");
```

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 being used to keep things as simple as possible and allow the library to be used with SQL Azure.

SQL Server objects are being **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
};

var storage = new SqlServerStorage("<name or connection string>", options);
```

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. You can adjust the polling interval, but, as always, lower intervals can harm your SQL Server, and higher interval produce too much latency, so be careful.

```
var options = new SqlServerStorageOptions
{
    QueuePollInterval = TimeSpan.FromSeconds(15) // Default value
};

var storage = new SqlServerStorage("<name or connection string>", options);
```

If you want to remove the polling technique, consider using the MSMQ extensions or Redis storage implementation.

Configuring the Invisibility Timeout Default SQL Server job storage implementation uses a regular table as a job queue. To be sure that a job will not be lost in case of unexpected process termination, it is being deleted only from a queue only upon a successful completion.

To make it invisible from other workers, the `UPDATE` statement with `OUTPUT` clause is being used to fetch a queued job and update the `FetchedAt` value (that signals for other workers that it was fetched) in an atomic way. Other workers see the fetched timestamp and ignore a job. But to handle the process termination, they will ignore a job only during a specified amount of time (defaults to 30 minutes).

Although this mechanism ensures that every job will be processed, sometimes it may cause either long retry latency or lead to multiple job execution. Consider the following scenario:

1. Worker A fetched a job (runs for a hour) and started it at 12:00.
2. Worker B fetched the same job at 12:30, because the default invisibility timeout was expired.
3. Worker C fetched the same job at 13:00, because

If you are using cancellation tokens, it will be set for Worker A at 12:30, and at 13:00 for Worker B. This may lead to the fact that your long-running job will never be executed. If you aren't using cancellation tokens, it will be concurrently executed by WorkerA and Worker B (since 12:30), but Worker C will not fetch it, because it will be deleted after successful performance.

So, if you have long-running jobs, it is better to configure the invisibility timeout interval:

```
var options = new SqlServerStorageOptions
{
    InvisibilityTimeout = TimeSpan.FromMinutes(30) // default value
};

var storage = new SqlServerStorage("<name or connection string>", options);
```

If you want to forget about invisibility interval, take a look at MSMQ extension, it uses transactional queues that return a job to its queue immediately upon a process termination.

Using SQL Server with MSMQ

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 in a more efficient way:

Feature	Raw SQL Server	SQL Server + MSMQ
Retry after process termination	Timeout (30 minutes by default)	Immediate after restart
Worst job fetch time	Polling Interval (15 seconds by default)	Immediate

So, if you want to lower background job processing latency with SQL Server storage, consider switching to 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.
2. Register all MSMQ queues in current `SqlServerStorage` instance.

If you are using only default queue, pass the path pattern to the `UseMsmqQueues` extension method (it also applicable to the OWIN bootstrapper):

```
app.UseHangfire(config =>
{
    config.UseSqlServerStorage("<connection string or its name>")
          .UseMsmqQueues(@".\hangfire-{0}");
});
```

To use multiple queues, you should pass them explicitly:

```
config.UseSqlServerStorage("<connection string or its name>")
      .UseMsmqQueues(@".\hangfire-{0}", "critical", "default");
```

If you are running Hangfire outside of web application, and OWIN context is not accessible, call the extension method on an instance of the `SqlServerStorage` class (parameters are the same):

```
var storage = new SqlServerStorage("<connection string or its name>");
storage.UseMsmqQueues(@".\hangfire-{0}");
```

Limitations

- Only transactional MSMQ queues supported for reability 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.

The following case will not work: the `critical` queue uses MSMQ, and the `default` queue uses SQL Server to store job queue. In this case job fetcher can not make the right decision.

```
config.UseSqlServerStorage("<name or connection string>")
      .UseMsmqQueues(@"hangfire-{0}", "critical");

config.UseServer("critical", "default");
```

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 from HangFire.Monitor – 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
{
    ServerName = "OldQueueServer" // Pass this to differentiate this server from the next one
};

config.UseServer(oldStorage, oldOptions);

/* This server will process only MSMQ queues, i.e. new jobs */
config.UseSqlServerStorage("<connection string or its name>")
      .UseMsmqQueues(@".\hangfire-{0}");

config.UseServer();
```

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
    ServerName = "OldQueueServer" // Pass this to differentiate this server from the next one
};

config.UseServer(oldStorage, oldOptions);

/* This server will process only MSMQ queues, i.e. new jobs */
config.UseSqlServerStorage("<connection string or its name>")
      .UseMsmqQueues(@".\hangfire-{0}", "critical", "default");

config.UseServer("critical", "default");
```

Using Redis

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.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.

Redis also supports Windows platform, but this is unofficial fork made by clever Microsoft guys. Here are GitHub repository branches for versions: 2.6, 2.8. Redis binaries are available through NuGet (32-bit, 64-bit) and Chocolate galleries (64-bit only). To install it as a Windows Service, check the rgl/redis repository, install it and update with binaries given above. *Don't use Redis 2.4 for Windows version for production environments (it is slow).*

Installation

Redis job storage implementation is available through the Hangfire.Redis NuGet package. So, install it using the NuGet Package Manager Console window:

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

Configuration

If you are using Hangfire in a web application, you can use extension methods for OWIN bootstrapper:

```
app.UseHangfire(config =>
{
    // Using hostname only and default port 6379
   config.UseRedisStorage("localhost");

   // or specify a port
   config.UseRedisStorage("localhost:6379");

   // or add a db number
   config.UseRedisStorage("localhost:6379", 0);

   // or use a password
   config.UseRedisStorage("password@localhost:6379", 0);

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

   /* ... */
})
```

When OWIN configuration is not applicable, you can create an instance of the `RedisStorage` class and pass it to the static `JobStorage.Current` property. All connection strings and options are same.

```
JobStorage.Current = new RedisStorage("password@localhost:6379", 0);
```

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
};

app.UseRedisStorage("localhost", 0, options);
```

1.6.3 Background methods

Calling methods in background

Fire-and-forget method invocation has never been simpler. As you can already know from the Quick start guide, you should only 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 being handled by a dedicated pool of worker threads. The following process is being invoked by each worker:

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

So, the job is being removed only after processing succeeded. Even if a process was terminated during the performance, Hangfire will perform a 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 periodical checks to fetch next jobs. If a process was terminated, the job will be re-queued only after `InvisibilityTimeout` expiration (30 minutes by default).
- **MSMQ** implementation uses transactional queues, so there is no need for periodic checks. Jobs are being fetched almost immediately after enqueueing.
- **Redis** implementation uses blocking `BRPOPLPUSH` command, so jobs are being fetched immediately, as with MSMQ. But in case of process termination, they are being enqueued only after timeout expiration that defaults to 15 minutes.

Calling methods with delay

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

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

Hangfire Server periodically check the schedule to enqueue scheduled jobs to their queues, allowing workers to perform them. By default, check interval is equal to `15 seconds`, but you can change it, just pass the corresponding option to the `BackgroundJobServer` ctor (also applicable for OWIN bootstrapper):

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

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

If you are processing your jobs inside an ASP.NET application, you should be warned about some setting that may prevent your scheduled jobs to be performed in-time. To avoid that behavour, perform the following steps:

- Disable Idle Timeout – set its value to `0`.
- Use the application auto-start feature.

Performing recurrent tasks

Recurring job registration is pretty simple – you need to write only a single line of code in application initialization logic:

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

This line creates a new entry in the storage. Special component in Hangfire Server (see Processing background jobs) will check the recurring jobs on a minute-based interval and enqueue them as fire-and-forget jobs, so you can track them as usual.

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

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

Each recurring job has its own unique identifier. In previous examples it is being generated implicitly, using the type name and method name of the given method call expression (resulting in `"Console.Write"`). The `RecurringJob` class contains other methods that take the recurring job identifier, so you can use define it explicitly to be able to use it later.

```
RecurringJob.AddOrUpdate("some-id", () => Console.WriteLine(), Cron.Hourly);
```

The call to `AddOrUpdate` method will create a new recurring job or update existing job with the same identifier.

Note: Recurring job identifier is **case sensitive** in some storage implementations.

You can remove 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 to recurring job itself, and its next execution time will not be recalculated.

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

The `RecurringJob` class is a facade for the `RecurringJobManager` class. If you want some more power, consider to use 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 a background job. But since they are being marshaled through the process boundaries, they are being serialized.

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

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

Since arguments are being 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 identifiers only to your background jobs.

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

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.

```
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):

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

When a worker determines that it need 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:

```
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**:

```
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 being 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:

```
// Somewhere in bootstrap logic, for example in the Global.asax.cs file
var container = new Container();

JobActivator.Current = new ContainerJobActivator(container);
```

To simplify the initial installation, there are some integration packages already available on NuGet:

- Hangfire.Autofac
- Hangfire.Ninject
- Hangfire.SimpleInjector
- Hangfire.Windsor

Some of these activators also provide an extension method for OWIN bootstrapper:

```
app.UseHangfire(config =>
{
    config.UseNinjectActivator(kernel)
});
```

Warning: 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 your 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.

Cancellation tokens are exposed through the `IJobCancellationToken` interface. It contains the `ThrowIfCancellationRequested` method that throws the `OperationCanceledException` when cancellation was requested:

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

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

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

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

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

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 being used to enqueue background jobs:

```
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)
    {
        ...
        _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.

```
[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.Arguments[0] == comment.Id.ToStri
        It.IsAny<EnqueuedState>());
}
```

1.6.4 Background processing

Processing background jobs

Background job processing is performed by Hangfire Server component that is exposed as the `BackgroundJobServer` class. To start the background job processing, you need to start it:

```
var server = new BackgroundJobServer();
server.Start();
```

You should also call the `Stop` method to shutdown all background thread gracefully, making cancellation tokens work on shutdown event.

```
server.Stop();
```

If you want to process background jobs inside a web application, you should use the OWIN bootstrapper's `UseServer` method instead of manual instantiation of the `BackgroundJobServer` class. If you are curious why, you should know that this method registers the callback for `host.OnAppDisposing` application event that stops the Server gracefully.

```
app.UseHangfire(config =>
{
    config.UseServer();
});
```

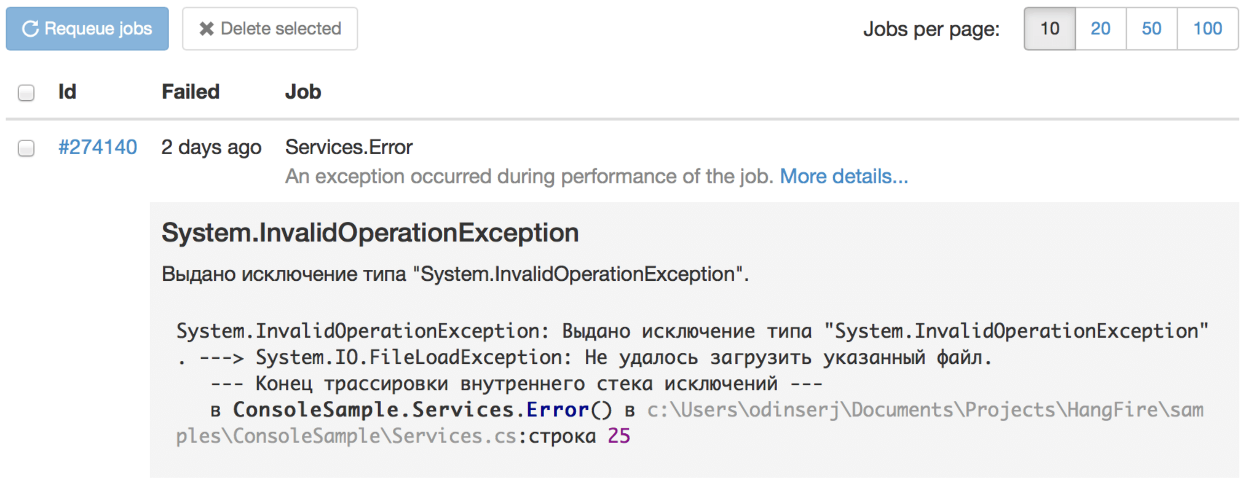
Warning: If you are using custom installation within a web application hosted in IIS, do not forget to install the Microsoft.Owin.Host.SystemWeb package. Otherwise some features, like graceful shutdown may not work.

Dealing with exceptions

Bad things happen. Every methods can throw exception of different types. 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 occured 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 being 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 occured 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).



Id	Failed	Job
#274140	2 days ago	Services.Error An exception occurred during performance of the job. More details...

System.InvalidOperationException

Выдано исключение типа "System.InvalidOperationException".

```
System.InvalidOperationException: Выдано исключение типа "System.InvalidOperationException"
. ---> System.IO.FileLoadException: Не удалось загрузить указанный файл.
   --- Конец трассировки внутреннего стека исключений ---
   в ConsoleSample.Services.Error() в c:\Users\odinserj\Documents\Projects\HangFire\sam
ples\ConsoleSample\Services.cs:строка 25
```

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 });
```

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 help you to avoid unnecessary calls to server. Plus, SignalR greatly simplifies the latter task.

I'll show you a simple example, where client needs only 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 become 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 we need to poll our server? It can say when the snippet become highlighted himself. 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:

```
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:

```
public void Highlight(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 a progress tracking, just add it. No additional tables and columns required, only JavaScript function. This is example of real and reliable asynchrony for ASP.NET applications without taking much effort to it.

Configuring the degree of parallelism

Background jobs are being 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 through the OWIN bootstrapper:

```
app.UseHangfire(config =>
{
    config.UseServer(Environment.ProcessorCount * 5);
});
```

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 separatly from requests to let you to process either CPU intensive or I/O intensive tasks as well and configure the degree of parallelism manually.

Running multiple server instances

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 consist 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 unqueness for different machines. Examples: `server1:9853`, `server1:4531`, `server2:6742`.

Since the defaults values provide uniqueness only on a process level, you should to 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.UseHangfire(config => config.UseServer(options));
```

1.6.5 Best practices

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

Make job arguments small and simple

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

Instead this:

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

Consider doing this:

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

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 different exceptions, and Hangfire will attempt to retry it many times.

You can face with different problems, if you didn't prepared your method to this feature. For example, if you are using email sending background job and experience errors with SMTP service, you can end with multiple letters sent to the single

Instead this:

```
public void Method()
{
    _emailService.Send("person@exapmle.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. . .

1.6.6 Deployment to production

Configuring Dashboard authorization

By default Hangfire allows access to Dashboard pages only for local requests. In order to give appropriate rights for production users, install the following package:

```
Install-Package Hangfire.Dashboard.Authorization
```

And configure authorization filters in the OWIN bootstrapper's configuration action:

```
using Hangfire.Dashboard;

app.UseHangfire(config =>
{
    config.UseAuthorizationFilters(new AuthorizationFilter
    {
        Users = "admin, superuser", // allow only specified users
        Roles = "admins" // allow only specified roles
    });

    // or

    config.UseAuthorizationFilters(
        new ClaimsBasedAuthorizationFilter("hangfire", "access"));
});
```

Or implement your own authorization filter:

```
using Hangfire.Dashboard;

public class MyRestrictiveAuthorizationFilter : IAuthorizationFilter
{
     public bool Authorize(IOwinContext context)
     {
         return false;
     }
}
```

Adding logging

Hangfire uses the Common.Logging library to log different events in an abstract way. If your application already have a logging framework installed, you need only to install the corresponding logging adapter and plug it in your application. You can check the list of supported adapters on NuGet Gallery site.

If your application does not have any logging framework installed, you need to choose and install it first. There are different logging frameworks, such as NLog, log4net, etc., but their description should not be in this article.

Hangfire does not produce many log messages and uses different logging levels to separate different types of messages. All logger names start with the `Hangfire` prefix, so you can use wildcarding feature of your logging framework to make Hangfire logging separate from your application logging.

Installing support for NLog

This section is for demonstration purposes only – to show that logging feature is easy to install. Consider you have an application with NLog library that is already configured. You only need to

1. Install the logging adapter for `Common.Logging` library:

```
PM> Install-Package Common.Logging.NLog
```

2. Configure the installed logging adapter:

```
var properties = new NameValueCollection();
properties["configType"] = "INLINE";

LogManager.Adapter = new NLogLoggerFactoryAdapter(properties);
```

For more information, please refer to the Common.Logging library's documentation.

Log level description

- **Trace** – for debugging Hangfire itself.
- **Debug** – for know why background processing does not work for you.
- **Info** – to see that everything is working as expected: *Hangfire was started or stopped*, *Hangfire components performed useful work*. This is the **recommended** level to log.
- **Warn** – to know about potential problems early: *performance failed, but automatic retry attempt will be made*, *thread abort exceptions*.
- **Error** – to know about problems that may lead to temporary background processing disruptions or problems you should know about: *performance failed, you need either to retry or delete a job manually*, *storage connectivity errors, automatic retry attempt will be made*.
- **Fatal** – to know that background job processing does not work partly or entirely, and requires manual intervention: *storage connectivity errors, retry attempts exceeded*, *different internal issues, such as OutOfMemoryException and so on*.

1.6.7 Extensibility

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

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

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

    public void OnCreated(CreatedContext filterContext)
    {
        Logger.InfoFormat(
            "Job that is based on method `{0}` has been created with id `{1}`",
            filterContext.Job.MethodData.MethodInfo.Name,
            filterContext.JobId);
    }

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

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

    public void OnStateElection(ElectStateContext context)
    {
        var failedState = context.CandidateState as FailedState;
        if (failedState != null)
        {
            Logger.WarnFormat(
                "Job `{0}` has been failed due to exception `{1}` but will be retried automatically u
                context.JobId,
                failedState.Exception);
        }
    }

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

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