Hangfire Documentation

Release 1.6

Sergey Odinokov

Jun 07, 2018

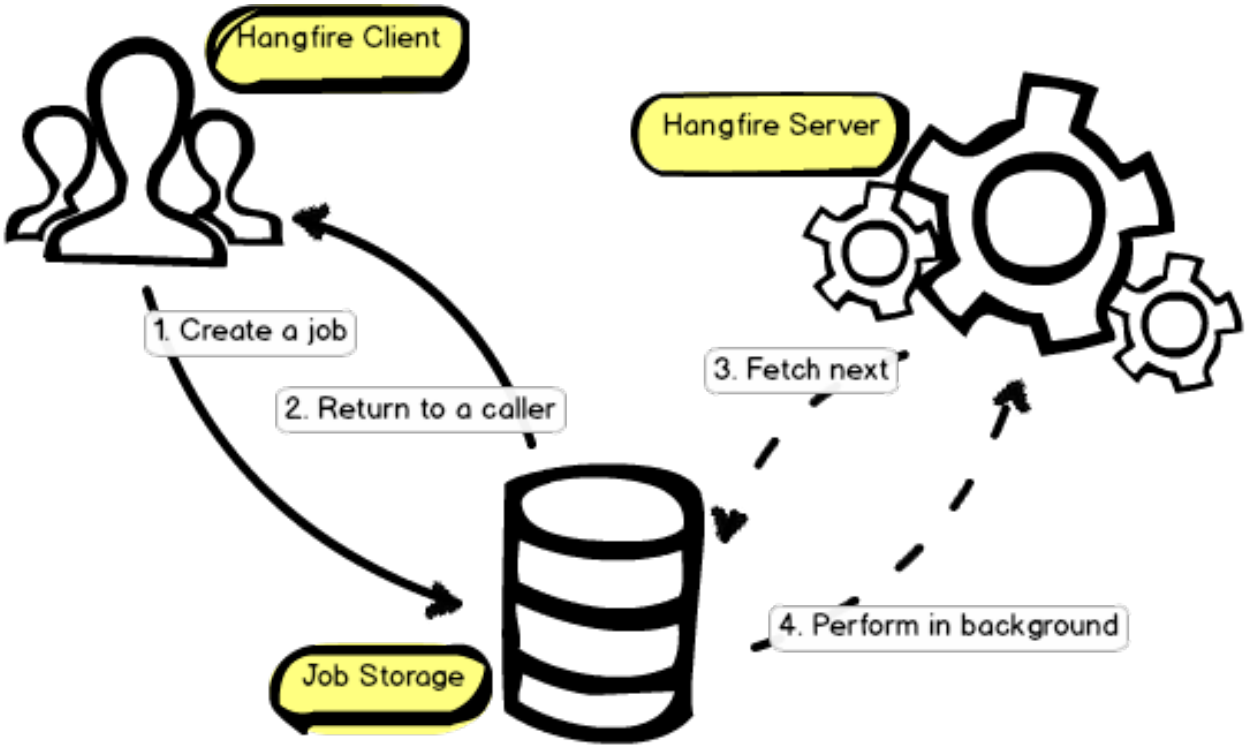
Contents

CHAPTER 1

Overview

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

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



CHAPTER 2

Requirements

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

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

CHAPTER 3

Client

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

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

```
var client = new BackgroundJobClient();

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

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

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

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

CHAPTER 4

Job Storage

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

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

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

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

CHAPTER 5

Server

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

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

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

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

CHAPTER 6

Table of Contents

6.1 Quick start

6.1.1 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
```

6.1.2 Configuration

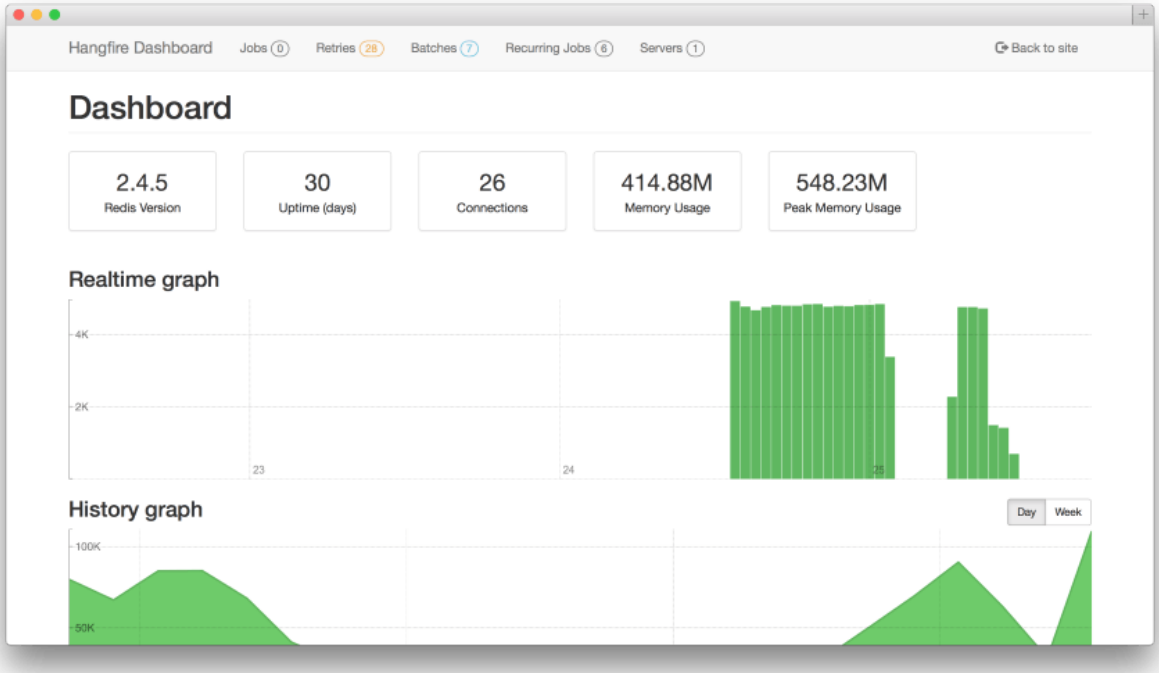
After installing the package, add or update the OWIN Startup class with the following lines:

```
using Hangfire;

// ...

public void Configuration(IAppBuilder app)
{
    GlobalConfiguration.Configuration.UseSqlServerStorage("<connection string or its
↪name>");

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

Authorization configuration required

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

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



6.1.3 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 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

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

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.WriteLine("Daily Job"), Cron.Daily);
```

Continuations

Continuations allow you to define complex workflows by chaining multiple background jobs together.

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

... 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.

6.2 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.

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

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.

---

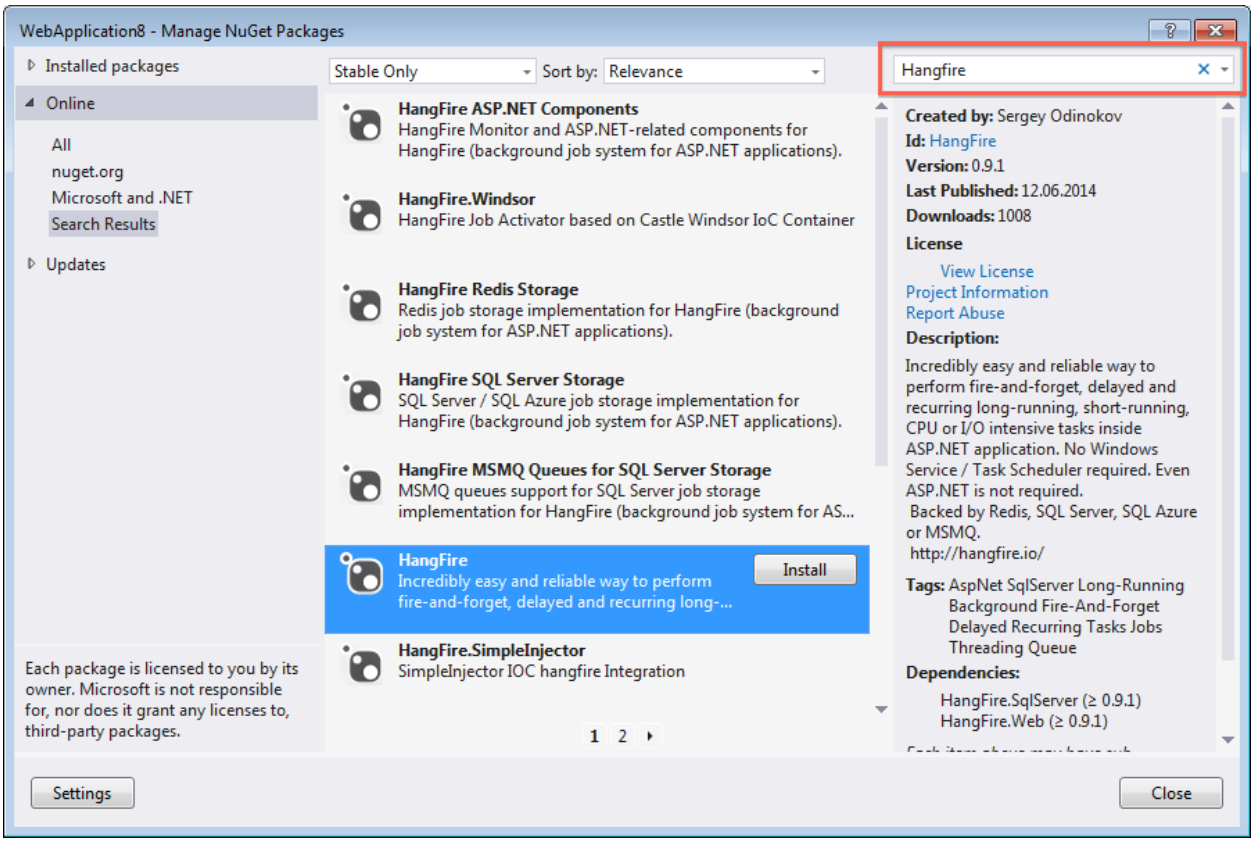
[1] But when your storage becomes broken, Hangfire can not do anything. Please, use different failover strategies for your storage to guarantee the processing of each job in case of a disaster.

6.2.1 Using Package Manager Console

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

6.2.2 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 chosen package:



6.3 Configuration

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

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

```
using Hangfire;

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
```

(continues on next page)

(continued from previous page)

```
        // Storage is the only thing required for basic configuration.
        // Just discover what configuration options do you have.
        GlobalConfiguration.Configuration
            .UseSqlServerStorage("<name or connection string>");
            //.UseActivator(...)
            //.UseLogProvider(...)
    }
}
```

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

```
using Hangfire;

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

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

6.3.1 Using Dashboard

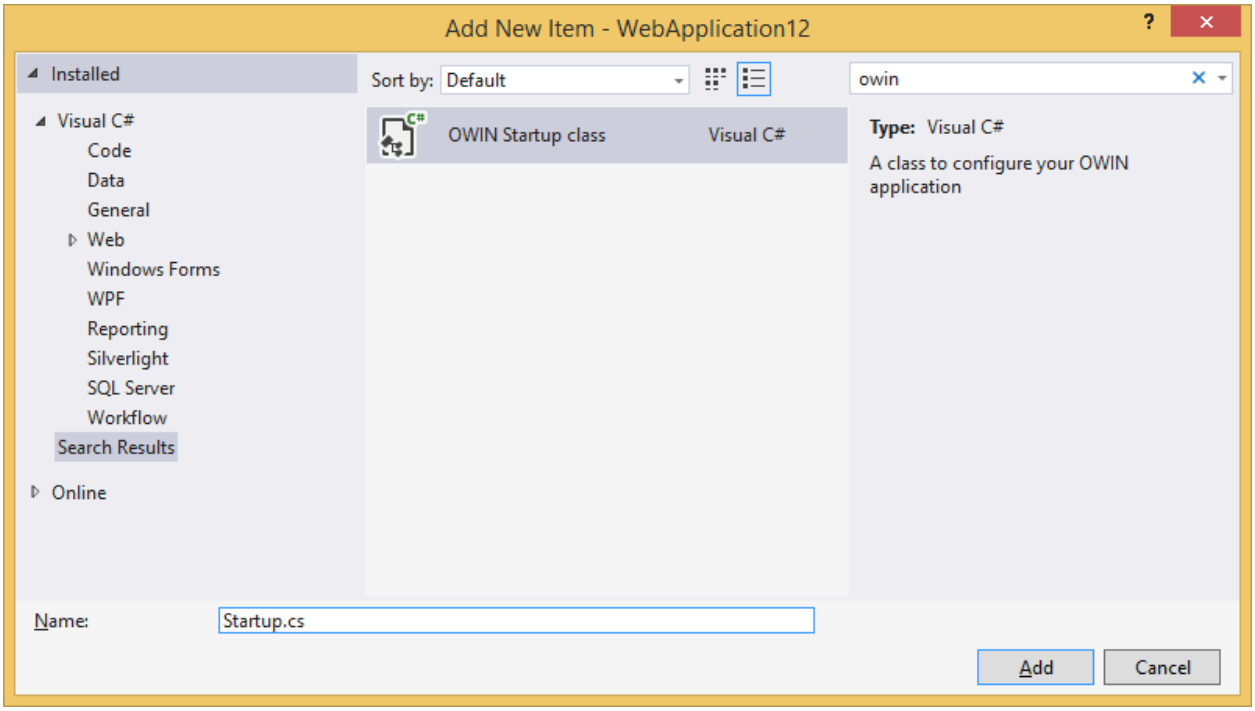
Hangfire Dashboard is a place where you could find all the information about your background jobs. It is written as an OWIN middleware (if you are not familiar with OWIN, don't worry), so you can plug it into your ASP.NET, ASP.NET MVC, Nancy, ServiceStack application as well as use OWIN Self-Host feature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

Adding Dashboard

Additional package required for ASP.NET + IIS

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

OWIN Startup class is intended to keep web application bootstrap logic in a single place. In Visual Studio 2013 you can add it by right clicking on the project and choosing the *Add / OWIN Startup Class* menu item.



If you have Visual Studio 2012 or earlier, just create a regular class in the root folder of your application, name it `Startup` and place the following contents:

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

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

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

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

Authorization configuration required

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

Configuring Authorization

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

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

Don't want to reinvent the wheel?

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

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

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

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

```
public class MyAuthorizationFilter : IDashboardAuthorizationFilter
{
    public bool Authorize(DashboardContext context)
    {
        var httpContext = context.GetHttpContext();

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

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

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

Method call order is important

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

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

Read-only view

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

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

Change URL Mapping

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

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

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

Change *Back to site* Link

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

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

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

Multiple Dashboards

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

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

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

6.3.2 Using SQL Server

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

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

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

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

Supported database engines

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

Snapshot isolation is not supported!

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

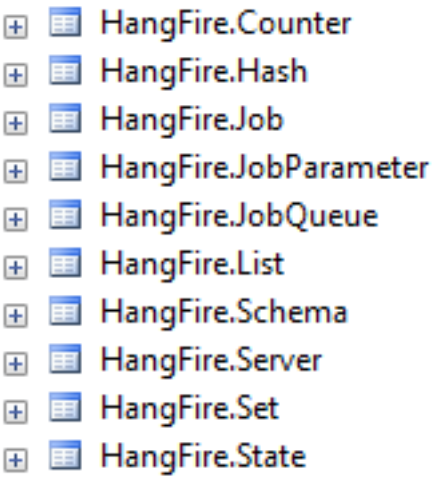
Configuration

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

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

Installing objects

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



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

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

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

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

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

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

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

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

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

GRANT CREATE TABLE TO [HangFire]
GO
```

Configuring the Polling Interval

One of the main disadvantage of raw SQL Server job storage implementation – it uses the polling technique to fetch new jobs. 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.

Please note that **millisecond-based intervals aren't supported**, you can only use intervals starting from *1 second*.

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

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

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

6.3.3 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	Immediate after restart	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. Please note that queue storage is limited to 1048576 KB by default (approximately 2 millions enqueued jobs), you can increase it through the MSMQ properties window.
2. Register all MSMQ queues in current `SqlServerStorage` instance.

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

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

To use multiple queues, you should pass them explicitly:

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

Limitations

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

Transition to MSMQ queues

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

If you are using default queue only, do this:

```
/* This server will process only SQL Server table queues, i.e. old jobs */
var oldStorage = new SqlServerStorage("<connection string or its name>");
var oldOptions = new BackgroundJobServerOptions
{
    ServerName = "OldQueueServer" // Pass this to differentiate this server from the␣
↪next one
};

app.UseHangfireServer(oldOptions, oldStorage);

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

app.UseHangfireServer();
```

If you use multiple queues, do this:

```
/* This server will process only SQL Server table queues, i.e. old jobs */
var oldStorage = new SqlServerStorage("<connection string>");
var oldOptions = new BackgroundJobServerOptions
{
    Queues = new [] { "critical", "default" }, // Include this line only if you have␣
↪multiple queues
    ServerName = "OldQueueServer" // Pass this to differentiate this server from the␣
↪next one
};

app.UseHangfireServer(oldOptions, oldStorage);
```

(continues on next page)

(continued from previous page)

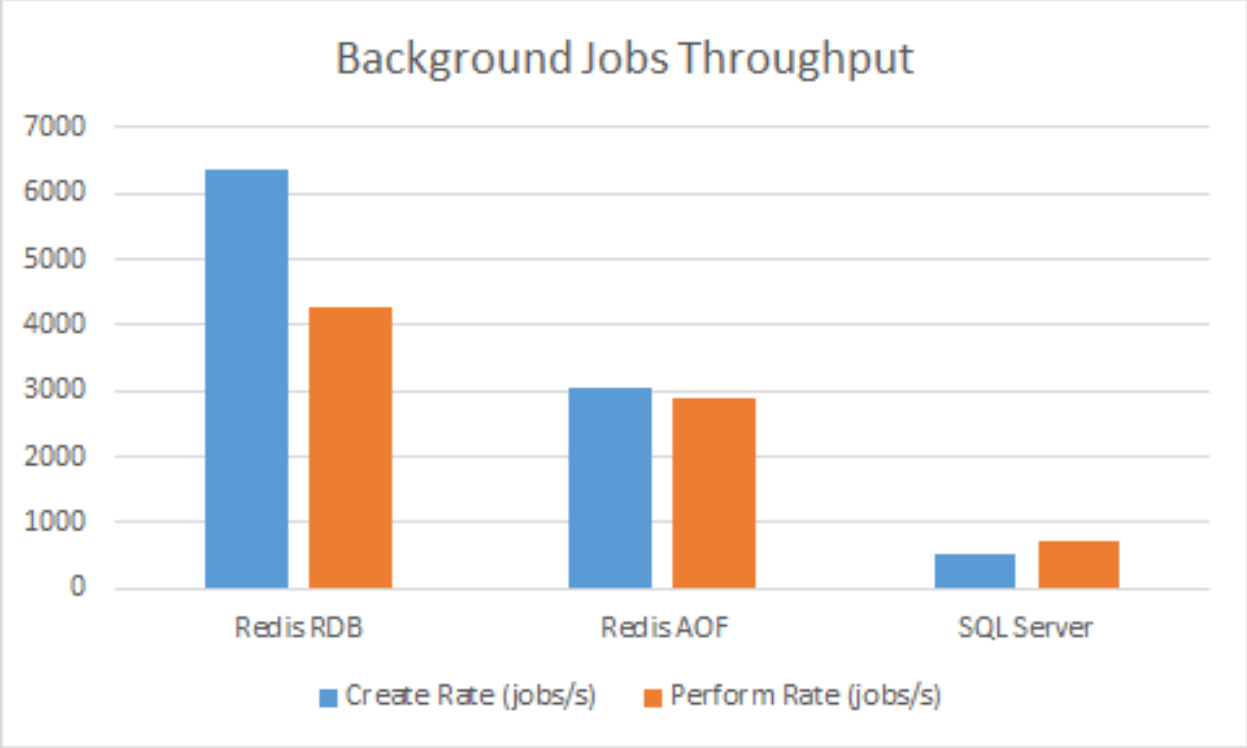
```
/* This server will process only MSMQ queues, i.e. new jobs */
GlobalConfiguration.Configuration
    .UseSqlServerStorage("<connection string or its name>")
    .UseMsmqQueues(@"FormatName:Direct=OS:localhost\hangfire-{0}", "critical",
↪"default");

app.UseHangfireServer();
```

6.3.4 Using Redis

Hangfire Pro subscription required

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

Hangfire with Redis job storage implementation processes jobs much faster than with SQL Server storage. On my development machine I observed more than 4x throughput improvement with empty jobs (method that does not do anything). `Hangfire.Pro.Redis` leverages the `BRPOPLPUSH` command to fetch jobs, so the job processing latency is kept to minimum.



Please, see the downloads page to obtain latest version of Redis. If you unfamiliar with this great storage, please see its documentation. Binaries for Windows are available through NuGet (32-bit, 64-bit) and Chocolatey galleries (64-bit only).

Limitations

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

Redis Configuration

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

Ensure the following options are configured

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

```
# Non-zero value cause long-running background jobs to be
# processed multiple times due to connection was closed.
# NOTE: This setting is only required for Hangfire.Pro.Redis 1.x!
timeout 0

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

Hangfire.Pro.Redis 2.x

Redis 2.6.12 is required

Installation

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

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

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

```
"dependencies": {
    "Hangfire.Pro.Redis": "2.0.2"
}
```

Configuration

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

Connection string

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

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

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

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

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

Option	Default
`syncTimeout`	`30000`
`allowAdmin`	`true`

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

In .NET Core you need to use IP addresses instead, because DNS lookup isn't available in StackExchange.Redis for .NET Core.

```
GlobalConfiguration.Configuration
    .UseRedisStorage("127.0.0.1");
```

Redis Cluster support

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

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

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

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

Passing options

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

```
var options = new RedisStorageOptions
{
    Prefix = "hangfire:app1:",
    InvisibilityTimeout = TimeSpan.FromHours(3)
};

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

The following options are available for configuration:

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

Hangfire.Pro.Redis 1.x

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

Configuration

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

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

// or with options
var options = new RedisStorageOptions();
```

(continues on next page)

(continued from previous page)

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

Connection pool size

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

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

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

Using key prefixes

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

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

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

6.3.5 Configuring logging

Starting from Hangfire 1.3.0, you are **not required to do anything**, if your application already uses one of the following libraries through the reflection (so that Hangfire itself does not depend on any of them). Logging implementation is **automatically chosen** by checking for the presence of corresponding types in the order shown below.

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

If you want to log Hangfire events and have no logging library installed, please choose one of the above and refer to its documentation.

Console logger

For console applications and sandbox installations, there is the `ColouredConsoleLogProvider` class. You can start to use it by doing the following:

```
LogProvider.SetCurrentLogProvider(new ColouredConsoleLogProvider());
```

Adding a custom logger

It is very simple to add a custom logger implementation. If your application uses another logging library that is not listed above, you only need to implement the following interfaces:

```
public interface ILog
{
    /// <summary>
    /// Log a message the specified log level.
    /// </summary>
    /// <param name="logLevel">The log level.</param>
    /// <param name="messageFunc">The message function.</param>
    /// <param name="exception">An optional exception.</param>
    /// <returns>true if the message was logged. Otherwise false.</returns>
    /// <remarks>
    /// Note to implementers: the message func should not be called if the loglevel␣
↪is not enabled
    /// so as not to incur performance penalties.
    ///
    /// To check IsEnabled call Log with only LogLevel and check the return value, no␣
↪event will be written
    /// </remarks>
    bool Log(LogLevel logLevel, Func<string> messageFunc, Exception exception = null);
}

public interface ILogProvider
{
    ILog GetLogger(string name);
}
```

After implementing the interfaces above, call the following method:

```
LogProvider.SetCurrentLogProvider(new CustomLogProvider());
```

Log level description

- **Trace** – for debugging Hangfire itself.
- **Debug** – to 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 learn 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*.

6.4 Background methods

6.4.1 Calling methods in background

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

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

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

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

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

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

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

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

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

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

6.4.2 Calling methods with delay

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

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

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

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

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

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

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

6.4.3 Performing recurrent tasks

Recurring job registration is just as simple as background job registration – you only need to write a single line of code:

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

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

Make sure your app is always running

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

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

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

Specifying identifiers

Each recurring job has its own unique identifier. In the previous examples it was generated implicitly, using the type and method names of the given call expression (resulting in `"Console.Write"` as the identifier). The `RecurringJob` class contains overloads that take an explicitly defined job identifier. So that you can refer to the job 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.

Identifiers should be unique

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

Identifiers may be case sensitive

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

Manipulating recurring jobs

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

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

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

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

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

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

6.4.4 Passing arguments

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

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

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

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

Reference parameters are not supported

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

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

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

6.4.5 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()
```

(continues on next page)

(continued from previous page)

```
        : 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.

6.4.6 Using IoC containers

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

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

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

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

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

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

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

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

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

`HttpContext` is not available

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

will be thrown during the job activation process.

So, **the entire dependency graph should be available**. Either register additional services without using the request scope, or use separate instance of container if your IoC container does not support dependency registrations for multiple scopes.

6.4.7 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));
```

The implementation is resolved automatically

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

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

6.4.8 Writing unit tests

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

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

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

```
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.Args[0] == comment.
↪Id),
        It.IsAny<EnqueuedState>());
}
```

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

6.4.9 Using Batches

Pro Only

This feature is a part of Hangfire Pro package set

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

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

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

But here is a much simpler method:

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

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

Installation

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

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

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

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

Limited storage support

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

Configuration

The default batch job expiration/retention time if the batch succeeds is 7 days.

```
var defaultBatchJobRetentionPeriod = new TimeSpan(2, 0, 0, 0); //2 day retention
Hangfire.GlobalConfiguration.Configuration.UseBatches(defaultBatchJobRetentionPeriod);
```

Chaining Batches

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

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

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

Complex Workflows

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

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

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

6.5 Background processing

6.5.1 Processing background jobs

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

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

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

Always dispose your background server

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

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

You can turn off the processing

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

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

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

6.5.2 Processing jobs in a web application

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

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

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

Using `BackgroundJobServer` class

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

Dispose the server instance when possible

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

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

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

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

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

            _backgroundJobServer = new BackgroundJobServer();
        }

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

Using OWIN extension methods

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

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

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

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

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

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

6.5.3 Processing jobs in a console application

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

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

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

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

`Hangfire.Core` package is enough

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

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

- Since the `Start` method is **non-blocking**, we insert a `Console.ReadKey` call to prevent instant shutdown of an application.
- The call to `Stop` method is implicit – it is made through the `using` statement.

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

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

            using (var server = new BackgroundJobServer())
            {
                Console.WriteLine("Hangfire Server started. Press any key to exit...
↪");

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

6.5.4 Processing jobs in a Windows Service

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

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

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

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

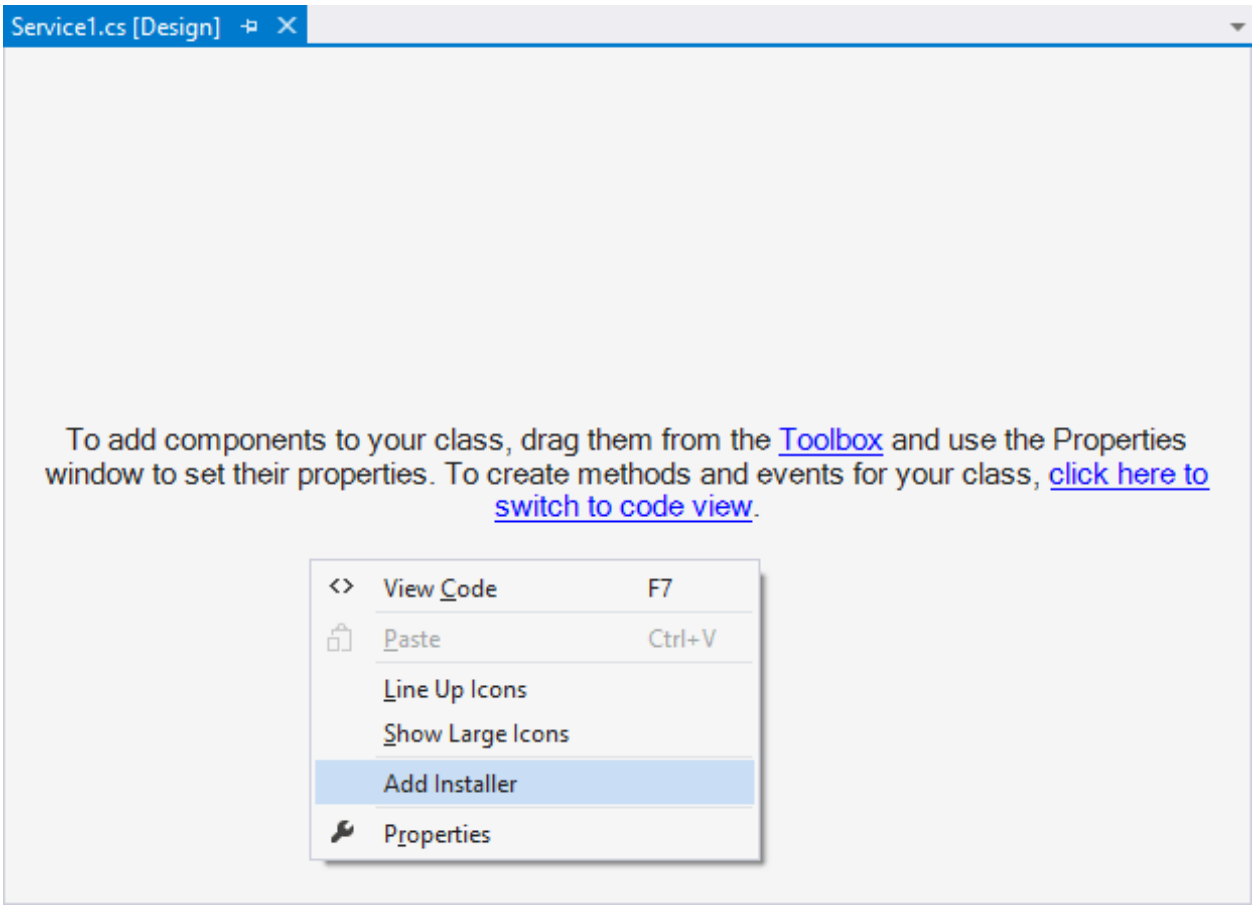
`Hangfire.Core` package is enough

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

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

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

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

        public Service1()
        {
            InitializeComponent();

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

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

        protected override void OnStop()
        {
            _server.Dispose();
        }
    }
}
```

If you are new to Windows Services in .NET projects, it is always better to google about them first, but for quick-start scenario you'll need only to add an installer and optionally configure it. To perform this step just go back to the design view of the service class, right click on it and choose the `Add Installer` menu item.



Then build your project, install your Windows Service and run it. If it fails, try look at your Windows Event Viewer for recent exceptions.

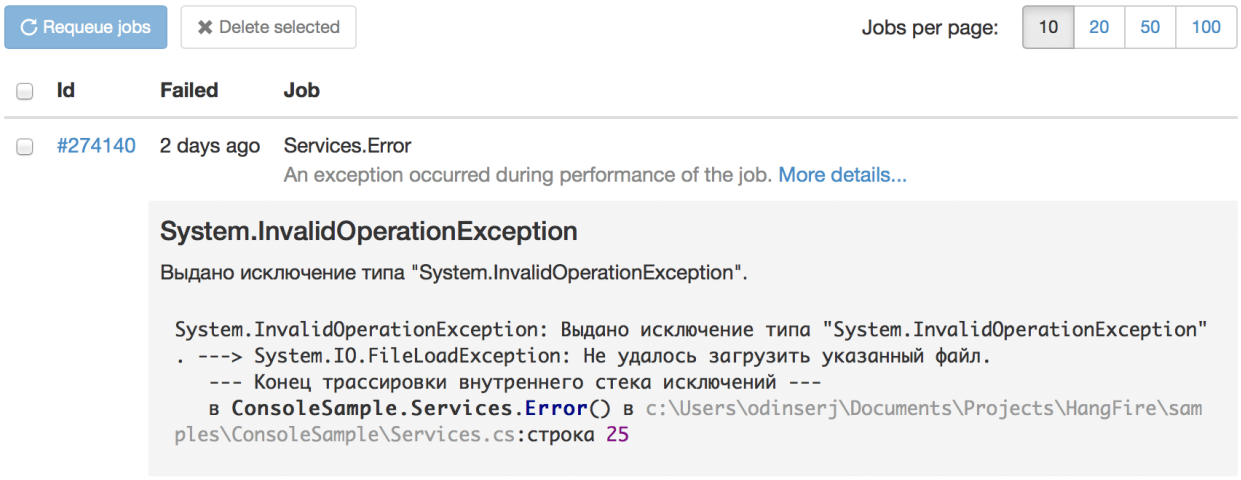
```
installutil <yourproject>.exe
```

6.5.5 Dealing with exceptions

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

Hangfire handles all exceptions 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 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).



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 });
```

6.5.6 Tracking the progress

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    _dbContext.SaveChanges();
}
```

Polling for a job status

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

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

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

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

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

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

But there is a better way.

Using server push with SignalR

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

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

First, you need to add a hub:

```
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)
```

(continues on next page)

(continued from previous page)

```
                .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 HighlightSnippet(int snippetId)
{
    ...
    _dbContext.SaveChanges();

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

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

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

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

6.5.7 Configuring the degree of parallelism

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

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

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

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

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

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

6.5.8 Placing processing into another process

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

Well scenarios

- Your background processing consumes **too much CPU or other resources**, and this decreases main application's performance. So you want to use separate machine for processing background jobs.
- You have long-running jobs that **are constantly aborted** (retrying, aborted, retried again and so on) due to regular shutdowns of the main application. So you want to use separate process with increased lifetime (and you can't use *always running mode* for your web application).
- *Do you have other suggestions? Please post them in the comment form below*.

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

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

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

Same storage requires the same code base

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

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

Doubtful scenarios

- You don't want to consume additional Thread Pool threads with background processing – Hangfire Server uses **custom, separate and limited thread pool**.
- You are using Web Farm or Web Garden and don't want to face with synchronization issues – Hangfire Server is **Web Garden/Web Farm friendly** by default.

6.5.9 Running multiple server instances

Obsolete since 1.5

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

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

Each Hangfire Server has a unique identifier that 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 uniqueness for different machines. Examples: `server1:9853`, `server1:4531`, `server2:6742`.

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

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

var server = new BackgroundJobServer(options);

// or

app.UseHangfireServer(options);
```

6.5.10 Configuring Job Queues

Hangfire can process multiple queues. If you want to prioritize your jobs or split the processing across your servers (some processes 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());
```

Queue name argument formatting

The Queue name argument must consist of lowercase letters, digits and underscore characters only.

To start to process multiple queues, you need to update your `BackgroundJobServer` configuration.

```
var options = new BackgroundJobServerOptions
{
    Queues = new[] { "critical", "default" }
};

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

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

6.6 Best practices

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

6.6.1 Make job arguments small and simple

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

Instead of doing this:

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

Consider doing this:

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

6.6.2 Make your background methods reentrant

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

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

Instead of doing this:

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

Consider doing this:

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

To be continued...

6.7 Deployment to production

6.7.1 Making ASP.NET application always running

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

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

On-Premise applications

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

1. Enable automatic start-up for Windows Process Activation (WAS) and World Wide Web Publishing (W3SVC) services (enabled by default).
2. Configure Automatic Startup for an Application pool (enabled by default).
3. Enable Always Running Mode for Application pool and configure Auto-start feature as written below.

Creating classes

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

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

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

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

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

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

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

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

    private BackgroundJobServer _backgroundJobServer;

    private HangfireBootstrapper()
    {
    }

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

            HostingEnvironment.RegisterObject(this);

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

            _backgroundJobServer = new BackgroundJobServer();
        }
    }

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

            HostingEnvironment.UnregisterObject(this);
        }
    }

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

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

```
public class Startup
{
    public void Configuration(IAppBuilder app)
```

(continues on next page)

(continued from previous page)

```
    {
            var options = new DashboardOptions
                {
            AuthorizationFilters = new[]
            {
                new LocalRequestsOnlyAuthorizationFilter()
            }
                };

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

Enabling Service Auto-start

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

Save only after all modifications

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

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

<!-- ... -->

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

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

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

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

Ensuring auto-start feature is working

If something went wrong...

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

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

Azure web applications

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

This setting does not work for free sites.

If nothing works for you...

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

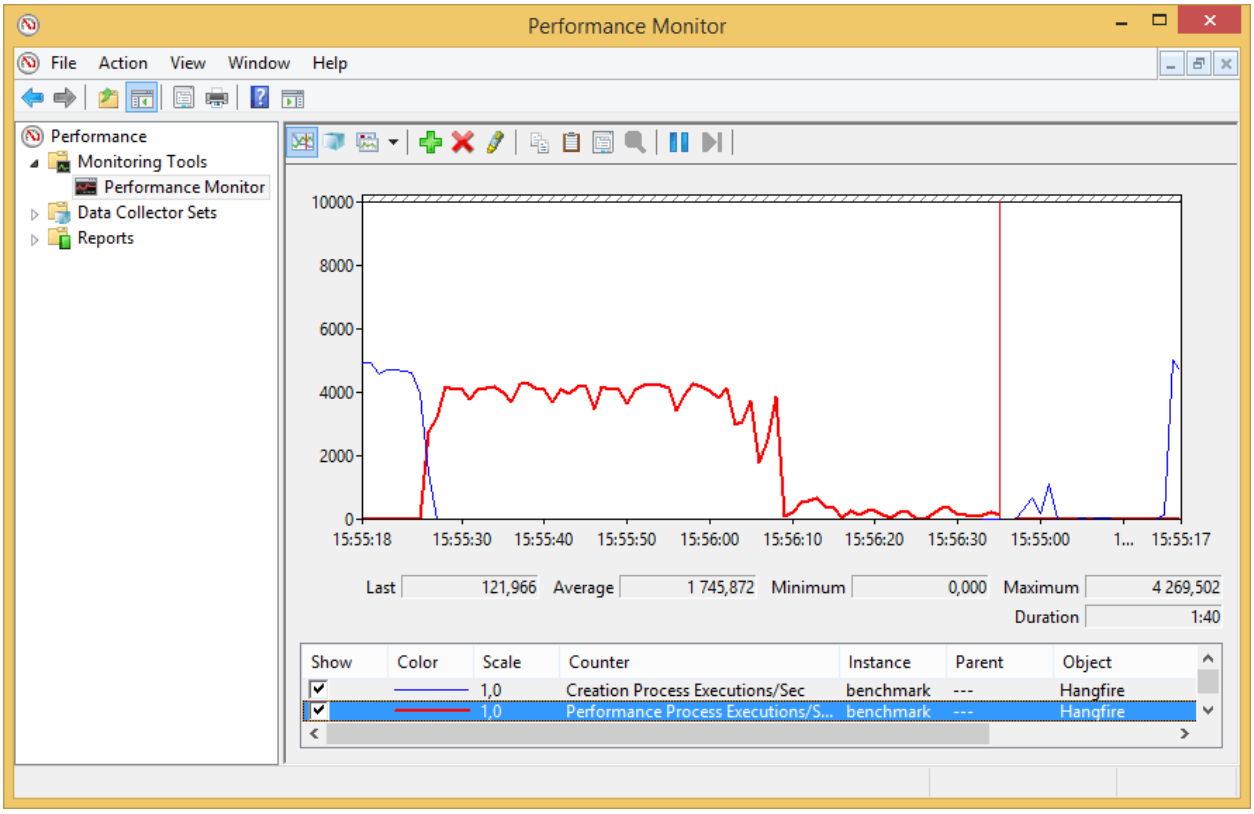
1. Use *separate process* to handle background jobs either on the same, or dedicated host.
2. Make HTTP requests to your web site on a recurring basis by external tool (for example, Pingdom).
3. *Do you know any other ways? Let me know!*

6.7.2 Using performance counters

Pro Only

This feature is a part of Hangfire Pro package set

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



Installation

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

```
hangfire-perf ipc
```

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

```
hangfire-perf upc
```

Configuration

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

```
using Hangfire.PerformanceCounters;

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

Initialization logic is much easier within your OWIN Startup class:

```
using Hangfire.PerformanceCounters;

public void Configure(IAppBuilder app)
{
```

(continues on next page)

(continued from previous page)

```
    app.UseHangfirePerformanceCounters();
}
```

Membership Configuration

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

Performance counters

Here is the list of performance counters currently exposed:

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

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

6.8 Extensibility

6.8.1 Using job filters

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

Define the filter

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

Apply it

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

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

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

6.9 Tutorials

6.9.1 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.Mvc5
```

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

```
using Postal;

namespace Hangfire.Mailer.Models
{
```

(continues on next page)

(continued from previous page)

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

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

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

Installing Hangfire

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

```
Install-Package Hangfire
```

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

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


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


    }

}
```

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

```
Update-Package -save
```

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

    var emailService = new EmailService(engines);

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

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

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

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

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

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

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

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

Automatic retries

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

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

Logging

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

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

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

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

And add it:

Either globally by calling the following method at application start:

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

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

Or locally by applying the attribute to a method:

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

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

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

```
Install-Package NLog
```

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

```

```

<?xml version="1.0" encoding="utf-8" ?> <nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" autoReload="true" throwExceptions="false">

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

<targets async="true">

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

<target xsi:type="EventLog" name="eventlog" source="${appName}" layout="${message}${newline}${exception:format=ToString}"/>

</targets> <rules>

<logger name="*" writeTo="default" minlevel="Info" /> <logger name="*" writeTo="eventlog" minlevel="Error" />

</rules>

</nlog>

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

Fix-deploy-retry

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

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

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

Preserving current culture

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

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

And check it inside the background job:

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

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

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

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

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

Setting up the project

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

Prerequisites

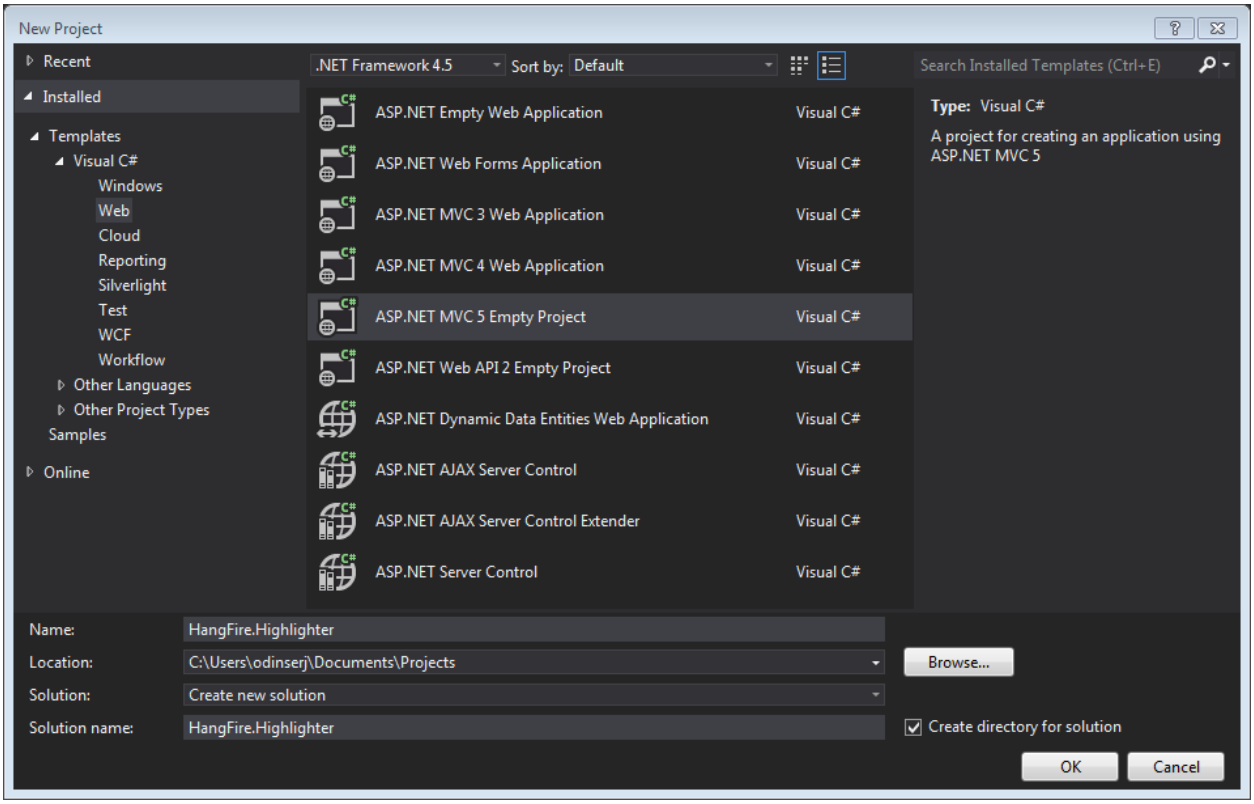
The tutorial uses **Visual Studio 2012** with Web Tools 2013 for Visual Studio 2012 installed, but it can be built either with Visual Studio 2013.

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

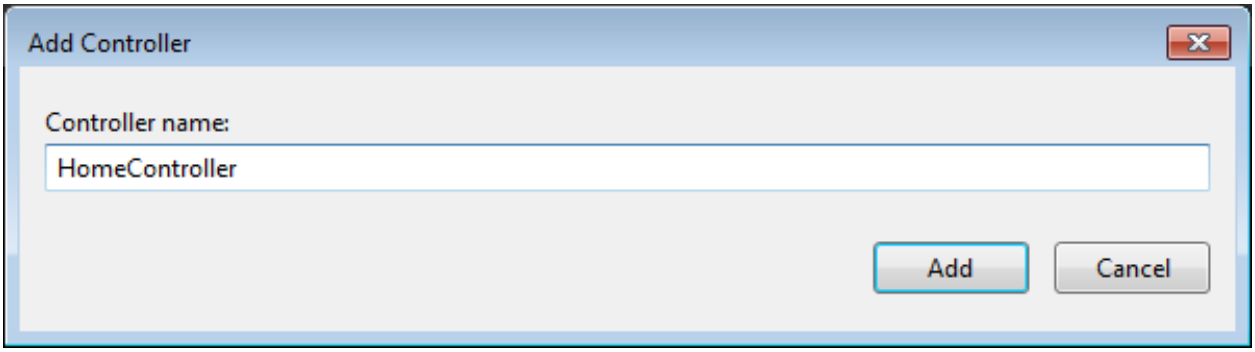
Creating a project

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

I've included some screenshots to make the project set-up not so boring:



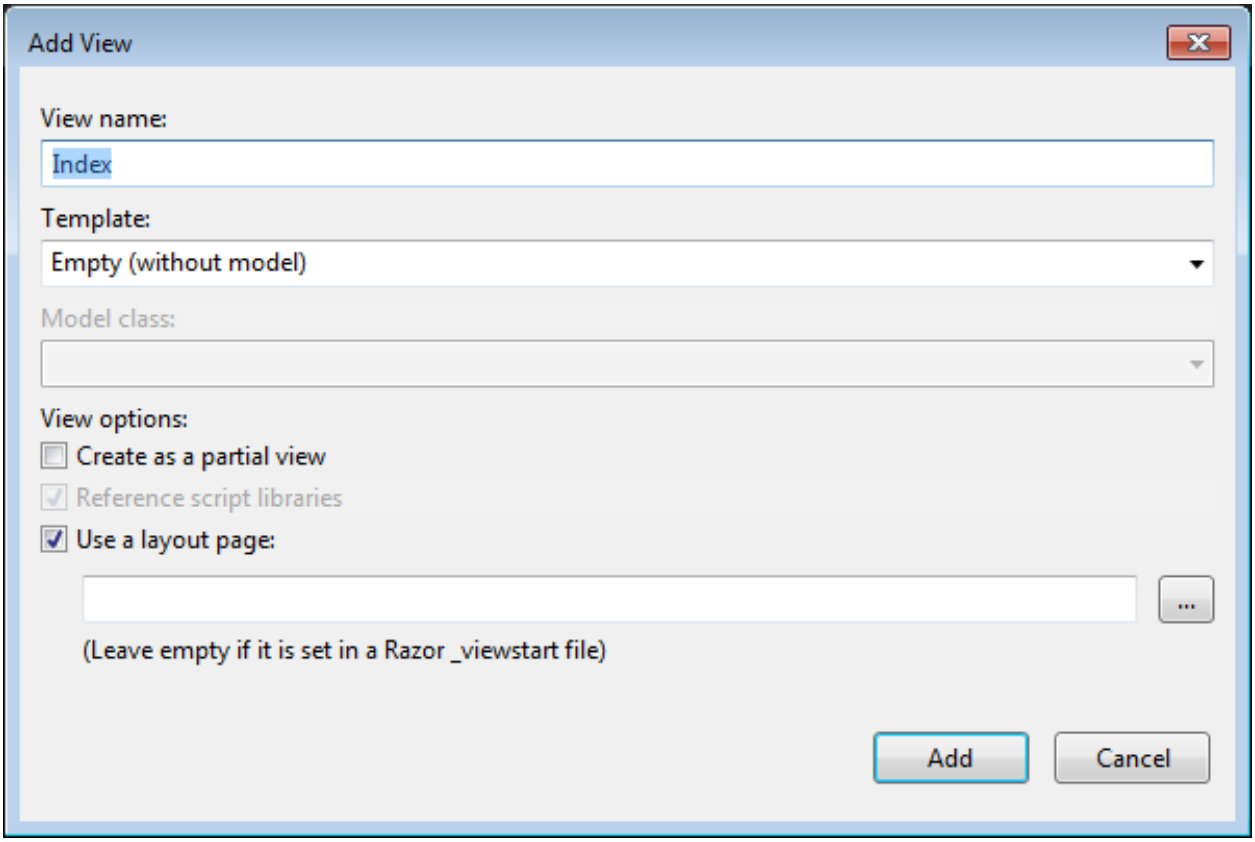
Then, we need a controller to handle web requests. So scaffold an **MVC 5 Controller - Empty** controller and call it `HomeController`:



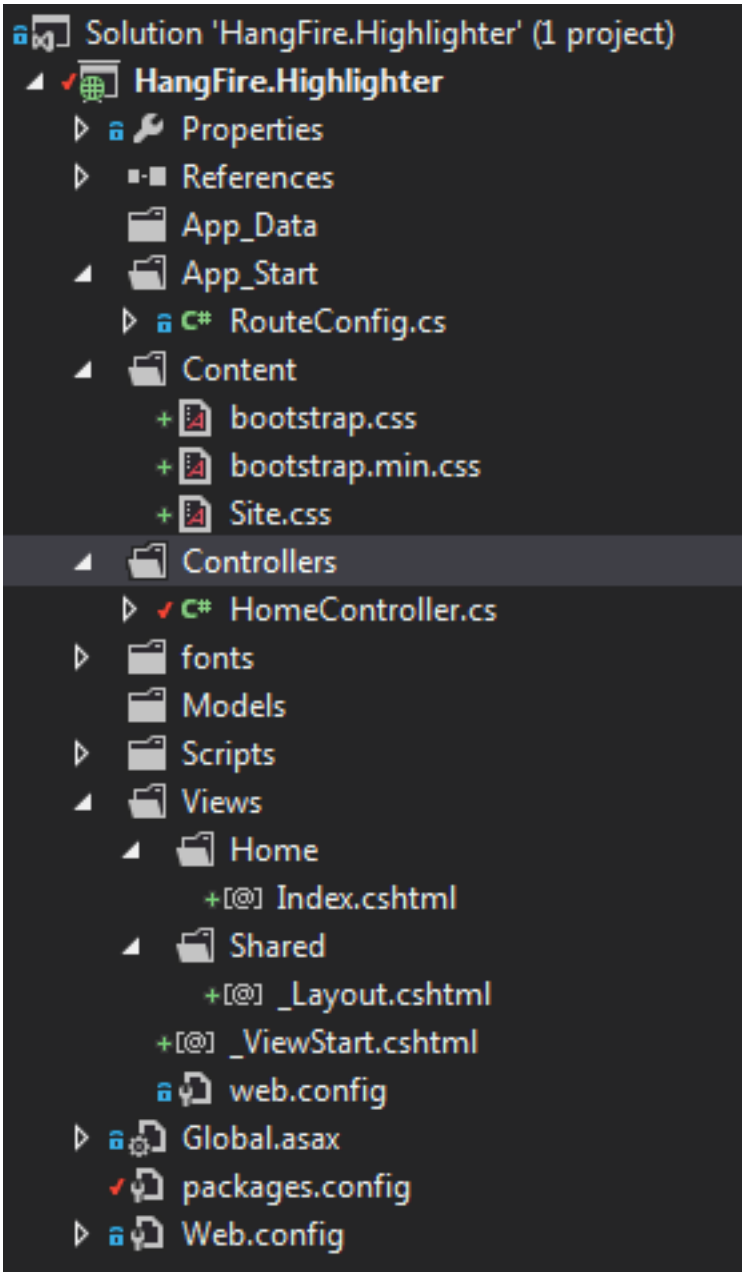
Our controller contains now only `Index` action and looks like:

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

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



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



Let's test the initial setup of our application. Press the `F5` key to start debugging and wait for your browser. If you encounter exceptions or don't see the default page, try to reproduce all the given steps, see the tutorial sources or ask a question in the comments below.

Defining a model

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

Installing Entity Framework

Open the Package Manager Console window and type:

```
Install-Package EntityFramework
```

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

```
// ~/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; Trusted_Connection=True;" providerName="System.Data.SqlClient" />
</connectionStrings>
```

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

```
Enable-Migrations
```

Adding code snippet model

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

```
// ~/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);
```

(continues on next page)

(continued from previous page)

```
                _db.SaveChanges();

                return RedirectToAction("Details", new { id = snippet.Id });
            }

            return View(snippet);
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                _db.Dispose();
            }
            base.Dispose(disposing);
        }
    }
}
```

```
@* ~/Views/Home/Index.cshtml *@

@model IEnumerable<Hangfire.Highlighter.Models.CodeSnippet>
@{ ViewBag.Title = "Snippets"; }

<h2>Snippets</h2>

<p><a class="btn btn-primary" href="@Url.Action("Create")">Create Snippet</a></p>
<table class="table">
    <tr>
        <th>Code</th>
        <th>Created At</th>
        <th>Highlighted At</th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                <a href="@Url.Action("Details", new { id = item.Id })">@Html.Raw(item.
↪HighlightedCode)</a>
            </td>
            <td>@item.CreatedAt</td>
            <td>@item.HighlightedAt</td>
        </tr>
     }
</table>
```

```
@* ~/Views/Home/Create.cshtml *@

@model Hangfire.Highlighter.Models.CodeSnippet
@{ ViewBag.Title = "Create a snippet"; }

<h2>Create a snippet</h2>

@using (Html.BeginForm())
{
```

(continues on next page)

(continued from previous page)

```
    @Html.ValidationSummary(true)

    <div class="form-group">
        @Html.LabelFor(model => model.SourceCode)
        @Html.ValidationMessageFor(model => model.SourceCode)
        @Html.TextAreaFor(model => model.SourceCode, new { @class = "form-control",␣
↪style = "min-height: 300px;", autofocus = "true" })
    </div>

    <button type="submit" class="btn btn-primary">Create</button>
    <a class="btn btn-default" href="@Url.Action("Index")">Back to List</a>
}
```

```
@* ~/Views/Home/Details.cshtml *@

@model Hangfire.Highlighter.Models.CodeSnippet
@{ ViewBag.Title = "Details"; }

<h2>Snippet <small>#@Model.Id</small></h2>

<div>
    <dl class="dl-horizontal">
        <dt>@Html.DisplayNameFor(model => model.CreatedAt)</dt>
        <dd>@Html.DisplayFor(model => model.CreatedAt)</dd>
        <dt>@Html.DisplayNameFor(model => model.HighlightedAt)</dt>
        <dd>@Html.DisplayFor(model => model.HighlightedAt)</dd>
    </dl>

    <div class="clearfix"></div>
</div>

<div>@Html.Raw(Model.HighlightedCode)</div>
```

Adding MiniProfiler

To not to profile our application by eye, we'll use the `MiniProfiler` package available on NuGet.

```
Install-Package MiniProfiler
```

After installing, update the following files as described to enable profiling.

```
// ~/Global.asax.cs

public class MvcApplication : HttpApplication
{
    /* ... */

    protected void Application_BeginRequest()
    {
        StackExchange.Profiling.MiniProfiler.Start();
    }

    protected void Application_EndRequest()
    {
```

(continues on next page)

(continued from previous page)

```
        StackExchange.Profiling.MiniProfiler.Stop();
    }
}
```

```
@* ~/Views/Shared/_Layout.cshtml *@

<head>
  <!-- ... -->
  @StackExchange.Profiling.MiniProfiler.RenderIncludes()
</head>
```

You should also include the following setting to the `web.config` file, if the `runAllManagedModulesForAllRequests` is set to `false` in your application (it is by default):

```
<!-- ~/web.config -->

<configuration>
  ...
  <system.webServer>
    ...
    <handlers>
      <add name="MiniProfiler" path="mini-profiler-resources/*" verb="*" type="System.
↪Web.Routing.UrlRoutingModule" resourceType="Unspecified" preCondition=
↪"integratedMode" />
    </handlers>
  </system.webServer>
</configuration>
```

Hiliting the code

It is the core functionality of our application. We'll use the http://hilite.me service that provides HTTP API to perform highlighting work. To start to consume its API, install the `Microsoft.Net.Http` package:

```
Install-Package Microsoft.Net.Http
```

This library provides simple asynchronous API for sending HTTP requests and receiving HTTP responses. So, let's use it to make an HTTP request to the *hilite.me* service:

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

/* ... */

public class HomeController
{
    /* ... */

    private static async Task<string> HighlightSourceAsync(string source)
    {
        using (var client = new HttpClient())
        {
            var response = await client.PostAsync(
                @"http://hilite.me/api",
                new FormUrlEncodedContent(new Dictionary<string, string>
                {
```

(continues on next page)

(continued from previous page)

```
                    { "lexer", "c#" },
                    { "style", "vs" },
                    { "code", source }
                }));

            response.EnsureSuccessStatusCode();

            return await response.Content.ReadAsStringAsync();
        }
    }

    private static string HighlightSource(string source)
    {
        // Microsoft.Net.Http does not provide synchronous API,
        // so we are using wrapper to perform a sync call.
        return RunSync(() => HighlightSourceAsync(source));
    }

    private static TResult RunSync<TResult>(Func<Task<TResult>> func)
    {
        return Task.Run<Task<TResult>>(func).Unwrap().GetAwaiter().GetResult();
    }
}
```

Then, call it inside the `HomeController.Create` method.

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

[HttpPost]
public ActionResult Create([Bind(Include = "SourceCode")] CodeSnippet snippet)
{
    try
    {
        if (ModelState.IsValid)
        {
            snippet.CreatedAt = DateTime.UtcNow;

            using (StackExchange.Profiling.MiniProfiler.StepStatic("Service call"))
            {
                snippet.HighlightedCode = HighlightSource(snippet.SourceCode);
                snippet.HighlightedAt = DateTime.UtcNow;
            }

            _db.CodeSnippets.Add(snippet);
            _db.SaveChanges();

            return RedirectToAction("Details", new { id = snippet.Id });
        }
    }
    catch (HttpRequestException)
    {
        ModelState.AddModelError("", "Highlighting service returned error. Try again␣
↪later.");
    }

    return View(snippet);
}
```

Note: We are using synchronous controller action method, although it is recommended to use asynchronous one to make network calls inside ASP.NET request handling logic. As written in the given article, asynchronous actions greatly increase application CAPACITY (The maximum throughput a system can sustain, for a given workload, while maintaining an acceptable response time for each individual transaction. – from "Release It" book written by Michael T. Nygard), but does not help to increase PERFORMANCE (How fast the system processes a single transaction. – from "Release It" book written by Michael T. Nygard). You can test it by yourself with a sample application – there are no differences in using sync or async actions with a single request.

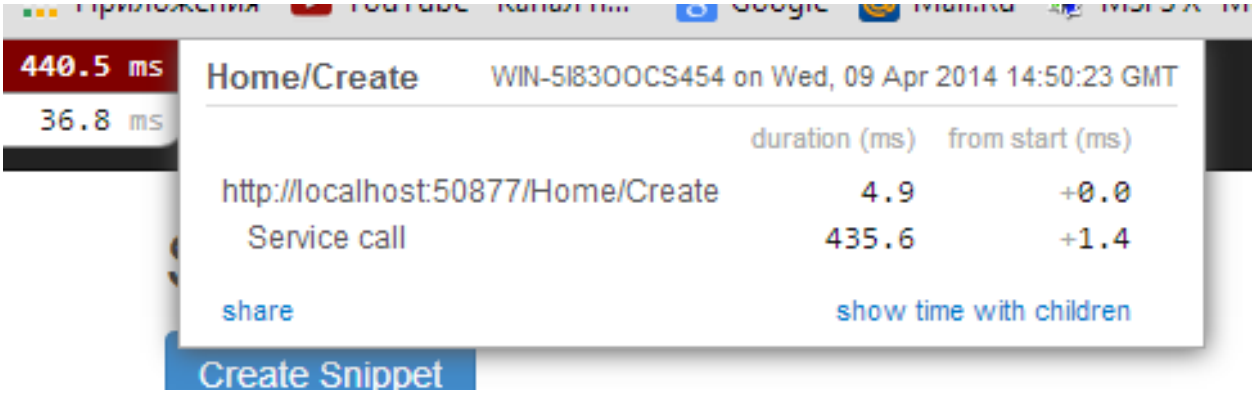
This sample is aimed to show you the problems related to application performance. And sync actions are used only to keep the tutorial simple.

The problem

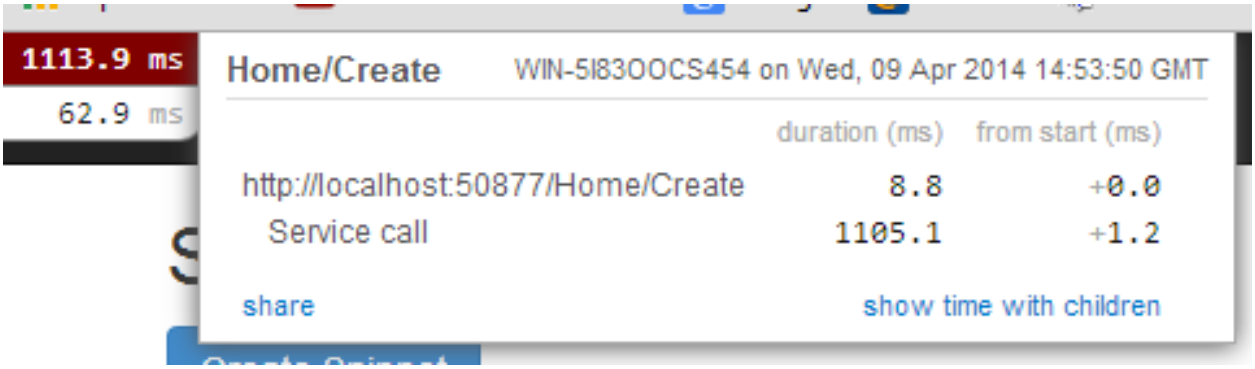
Tip: You can use the hosted sample to see what's going on.

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

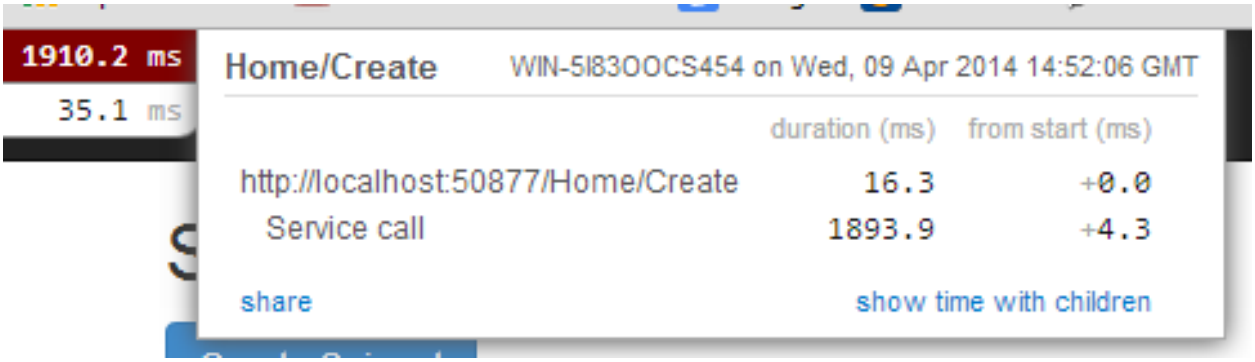
On my development machine it took about 0.5s to redirect me to the details page. But let's look at *MiniProfiler* to see what is the cause of this delay:



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



And finally a large one:



The lag is increasing when we enlarge our code snippets. Moreover, consider that syntax highlighting web service (that is not under your control) experiences heavy load, or there are latency problems with network on their side. Or consider heavy CPU-intensive task instead of web service call that you can not optimize well.

Your users will be annoyed with un-responsive application and inadequate delays.

Solving a problem

What can you do with a such problem? Async controller actions will not help, as I said *earlier*. You should somehow take out web service call and process it outside of a request, in the background. Here is some ways to do this:

- **Use recurring tasks** and scan un-highlighted snippets on some interval.
- **Use job queues**. Your application will enqueue a job, and some external worker threads will listen this queue for new jobs.

Ok, great. But there are several difficulties related to these techniques. The former requires us to set some check interval. Shorter interval can abuse our database, longer interval increases latency.

The latter way solves this problem, but brings another ones. Should the queue be persistent? How many workers do you need? How to coordinate them? Where should they work, inside of ASP.NET application or outside, in Windows Service? The last question is the sore spot of long-running requests processing in ASP.NET application:

> **Warning:** **DO NOT** run long-running processes inside of your ASP.NET application, unless they are prepared to **die at any instruction** and there is mechanism that can re-run them.
>
> They will be simple aborted on application shutdown, and can be aborted even if the `IRegisteredObject` interface is used due to time out.

Too many questions? Relax, you can use Hangfire. It is based on *persistent queues* to survive on application restarts, uses *reliable fetching* to handle unexpected thread aborts and contains *coordination logic* to allow multiple worker threads. And it is simple enough to use it.

Note: **YOU CAN** process your long-running jobs with Hangfire inside ASP.NET application – aborted jobs will be restarted automatically.

Installing Hangfire

To install Hangfire, run the following command in the Package Manager Console window:

```
Install-Package Hangfire
```

After the package installed, add or update the OWIN Startup class with the following lines of code.

```
public void Configuration(IAppBuilder app)
{
    GlobalConfiguration.Configuration.UseSqlServerStorage("HighlighterDb");

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

That's all. All database tables will be created automatically on first start-up.

Moving to background

First, we need to define our background job method that will be called when worker thread catches highlighting job. We'll simply define it as a static method inside the `HomeController` class with the `snippetId` parameter.

```
// ~/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:

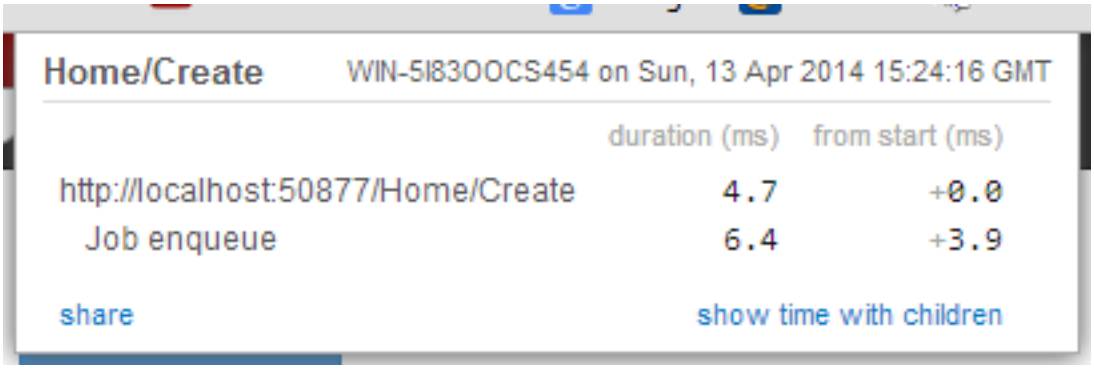
```
// ~/Controllers/HomeController.cs

[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"))
```

(continues on next page)

(continued from previous page)

```
        {
            // Enqueue a job
            BackgroundJob.Enqueue(() => HighlightSnippet(snippet.Id));
        }

        return RedirectToAction("Details", new { id = snippet.Id });
    }

    return View(snippet);
}
```

That's all. Try to create some snippets and see the timings (don't worry if you see an empty page, I'll cover it a bit later):



Good, 6ms vs ~2s. But there is another problem. Did you notice that sometimes you are redirected to the page with no source code at all? This happens because our view contains the following line:

```
<div>@Html.Raw(Model.HighlightedCode)</div>
```

Why the `Model.HighlightedCode` returns null instead of highlighted code? This happens because of **latency** of the background job invocation – there is some delay before a worker fetch the job and perform it. You can refresh the page and the highlighted code will appear on your screen.

But empty page can confuse a user. What to do? First, you should take this specific into a place. You can reduce the latency to a minimum, but **you can not avoid it**. So, your application should deal with this specific issue.

In our example, we'll simply show the notification to a user and the un-highlighted code, if highlighted one is not available yet:

```
@* ~/Views/Home/Details.cshtml *@

<div>
    @if (Model.HighlightedCode == null)
    {
        <div class="alert alert-info">
            <h4>Highlighted code is not available yet.</h4>
            <p>Don't worry, it will be highlighted even in case of a disaster
                (if we implement failover strategies for our job storage).</p>
            <p><a href="javascript:window.location.reload()">Reload the page</a>
                manually to ensure your code is highlighted.</p>
        </div>

        @Model.SourceCode
    }
```

(continues on next page)

(continued from previous page)

```
    else
    {
        @Html.Raw(Model.HighlightedCode)
    }
</div>
```

But instead you could poll your application from a page using AJAX until it returns highlighted code:

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

public ActionResult HighlightedCode(int snippetId)
{
    var snippet = _db.Snippets.Find(snippetId);
    if (snippet.HighlightedCode == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.NoContent);
    }

    return Content(snippet.HighlightedCode);
}
```

Or you can also use send a command to users via SignalR channel from your `HighlightSnippet` method. But that's another story.

Note: Please, note that user still waits until its source code will be highlighted. But the application itself became more responsive and he is able to do another things while background job is processed.

Conclusion

In this tutorial you've seen that:

- Sometimes you can't avoid long-running methods in ASP.NET applications.
- Long running methods can cause your application to be un-responsible from the users point of view.
- To remove waits you should place your long-running method invocation into background job.
- Background job processing is complex itself, but simple with Hangfire.
- You can process background jobs even inside ASP.NET applications with Hangfire.

Please, ask any questions using the comments form below.