InfinniPlatform

Release 2.0

August 24, 2017

Оглавление

InfinniPlatform is an infrastructure framework designed to solve several most common problems when you try to build a scalable web-application. For example, authentication and authorization, data storage, caching, logging, messaging, push notification and etc. It offers to a developer a unified solution which covers majority of challenges the developer may encounter while working on the project. Ready-to-use infrastructure is what makes the platform tick right out of the box.

InfinniPlatform is a cross-platform open source project based on .NET Core and well integrated with ASP.NET Core. Moreover, InfinniPlatform was designed as set of loosely coupled flexible components so that you can use them separately. In our work we use the most modern and advanced industry-grade open source components such as MongoDB[1], Redis[2], RabbitMQ[3] and etc. This approach ensures you avoid vendor lock-in for the core parts of both middleware and application. It is worth noting, however, that flexibility of the framework allows you using any other components.

This solution is distributed under AGPLv3[4] license which means you may use it free of charge and exceptionally all components employed are free to use as well to all.

[1] https://www.mongodb.com/download-center
[2] http://redis.io/download
[3] https://www.rabbitmq.com/download.html
[4] https://raw.githubusercontent.com/InfinniPlatform/InfinniPlatform/master/LICENSE

Getting Started

Below you will find the steps to build your first ASP.NET Core app powered by InfinniPlatform.

Let's start by building a simple "Hello, world!" app.

1. Install .NET Core[5]

2. Create a new ASP.NET Core project:

```
mkdir myapp
cd myapp
dotnet new web
```

3. Install InfinniPlatform.Core package:

```
dotnet add package InfinniPlatform.Core -s https://www.myget.org/F/infinniplatform/ -v 2.3.8-*
```

4. Create MyHttpService.cs and define an HTTP service:

Listing 1.1: MyHttpService.cs

```
using System.Threading.Tasks;

using InfinniPlatform.Http;

namespace myapp
{
    class MyHttpService : IHttpService
    {
        public void Load(IHttpServiceBuilder builder)
        {
            builder.Get["/hello"] = async request =>
                await Task.FromResult("Hello from InfinniPlatform!");
        }
    }
}
```

5. Create MyAppContainerModule.cs and register the HTTP service:

Listing 1.2: MyAppContainerModule.cs

```
using InfinniPlatform.Http;
using InfinniPlatform.IoC;

namespace myapp
{
    class MyAppContainerModule : IContainerModule
```

[5] https://dot.net/core

```
    {
        public void Load(IContainerBuilder builder)
        {
            builder.RegisterType<MyHttpService>().As<IHttpService>().SingleInstance();
        }
    }
}
```

6. Update the code in Startup.cs to use InfinniPlatform:

Listing 1.3: Startup.cs

```
using System;

using InfinniPlatform.AspNetCore;
using InfinniPlatform.IoC;

using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace myapp
{
    public class Startup
    {
        public IServiceProvider ConfigureServices(IServiceCollection services)
        {
            services.AddContainerModule(new MyAppContainerModule());

            return services.BuildProvider();
        }

        public void Configure(IApplicationBuilder app, IContainerResolver resolver)
        {
            app.UseDefaultAppLayers(resolver);
        }
    }
}
```

7. Restore the packages:

```
dotnet restore -s https://www.myget.org/F/infinniplatform/
```

8. Run the app (the dotnet run command will build the app when it's out of date):

```
dotnet run
```

9. Browse to http://localhost:5000/hello

10. Press Ctrl+C to stop the app

Dynamic Objects

Dynamic objects expose members such as properties and methods at run time, instead of in at compile time. This enables you to create objects to work with structures that do not match a static type or format. Having created an instance of such object makes possible to bind a set of properties to it. This behavior is made possible by 'late binding' with using of keyword dynamic. Basic usage of dynamic objects is processing of non-structured and non-formalized data.

Creating Dynamic Objects

InfinniPlatform has dynamic object represented by DynamicDocument.

```
dynamic instance = new DynamicDocument();
```

Setting Properties of Dynamic Objects

Dynamic object instance can be created with pre-defined properties:

```
dynamic instance = new DynamicDocument
                {
                    { "Property1", 123 },
                    { "Property2", "Abc" },
                    { "Property3", DateTime.Now },
                    {
                        "Property4", new DynamicDocument
                                {
                                    { "SubProperty1", 456 },
                                    { "SubProperty2", "Def" }
                                }
                    }
                };
```

or define properties later:

```
instance.Property1 = 123;
instance.Property2 = "Abc";
instance.Property3 = DateTime.Now;
instance.Property4 = new DynamicDocument();
instance.Property4.SubProperty1 = 456;
instance.Property4.SubProperty2 = "Def";
```

Value property may define a link to delegate:

```
instance.Sum = new Func<int, int, int>((a, b) => a + b);
```

Getting Properties of Dynamic Objects

Defining properties of dynamic object is identical to defining properties of regular classes:

```
var property1 = instance.Property1; // 123
var property2 = instance.Property2; // "Abc"
var property3 = instance.Property3; // DateTime.Now
var property4 = instance.Property4;
var subProperty1 = property4.SubProperty1; // 456
var subProperty2 = instance.Property4.SubProperty2; // "Def"
var sum = instance.Sum(2, 3); // 5
```

Recommendations to work with Dynamic Objects

Dynamic objects simplify processing of non-structured data and at the same time increases chance of error due to the fact that expressions workin with dynamic objects being built are not affected by syntax analysis. Any expression's result which formed by calling to either dynamic object or its properties is the dynamic object itself. Thus if the result type of dynamic object is not defined may cause large code blocks which is uncontrollable at the building stage. Also you should bear in mind about lack of information in exception stack that may arise while building dynamic code.

This is very important, due to mentioned reasons, to exactly define the result type of dynamic expression and use keyword dynamic where it is indeed applicable. In case you don't use non-structured data objects, particular properties of data types can be often easily defined. Good rule is to define type in advance to avoid errors of type conversion and even in case of getting one you will be aware of its reasons.

```
int property1 = instance.Property1; // 123
string property2 = instance.Property2; // "Abc"
DateTime property3 = instance.Property3; // DateTime.Now
dynamic property4 = instance.Property4;
int subProperty1 = property4.SubProperty1; // 456
string subProperty2 = instance.Property4.SubProperty2; // "Def"
int sum = instance.Sum(2, 3); // 5
```

Serialization of Dynamic Objects

Class instances DynamicDocument can be serialized and deserialized to/from JSON. You my find additional info here Data Serialization (c. 31).

IoC Container

Modern applications have of configurable, related components. The more complicated app logic is, the more complicated components and its relations are. The key to the successful app development is the principle of writing decoupled code. This rule underlines that every component must be isolated and ideally must not rely on dependencies from the other components. Decoupled apps are the most flexible and easily maintainable. They can be tested with less efforts and time.

All InfinniPlatform app components get managed by IoC[6]-container which main goal is to simplify and automate writing of decoupled code. IoC[7]-container stores list of app components and automatically defines relations between them and controls lifetime of each one.

IoC Container Module

Before an application will be run you need to register all components in the IoC container. Modules can add a set of related components to a container. Each module implements the IContainerModule interface and contains only the Load() method for registering components.

Loading of IoC Container Module

Method Load() designed to register app components and must not contain any other logic due to the fact it is posed in inconsistent state. To register components into Load() an instance of the IContainerBuilder interface is passed.

Note: If there is necessity to execute some logic immediately after the app is run one should use methods described in the article Application Events (с. 17).

Common structure of IoC-container module may look like this:

```
public class MyAppContainerModule : IContainerModule
{
    public void Load(IContainerBuilder builder)
    {
        // Registering components...
    }
}
```

[6] http://martinfowler.com/articles/injection.html
[7] http://martinfowler.com/articles/injection.html

Configuration IoC Container on Startup

To configure the IoC container in an ASP.NET Core application you need to create an instance of the IServiceProvider[8] interface and return one from the ConfigureServices()[9] method. For features that require substantial setup there are Add[Component] extension methods on IServiceCollection[10]. User defined modules are added by the AddContainerModule() extension method. The BuildProvider() extension method builds and returns an instance of the IServiceProvider[11] interface.

```
public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddContainerModule(new MyAppContainerModule());

        return services.BuildProvider();
    }

    // ...
}
```

Registration Concepts

Interface IContainerBuilder represents a few overloadings of the method Register(), designed to register IoC-container components. Registration methods also define the way of creating instances of components. Instances components may be made via reflection[12] by means IoC-container itself; may be represented by beforehand created instance; may be created by a factory function or a lambda[13]-expression. Each component may represent one or a few services defined with using one of the methods As().

```
public interface IMyService
{
    // ...
}


public class MyComponent : IMyService
{
    // ...
}

public class ContainerModule : IContainerModule
{
    public void Load(IContainerBuilder builder)
    {
        builder.RegisterType<MyComponent>().As<IMyService>();

        // ...
    }
}
```

[8] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection
[9] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup#the-configureservices-method
[10] https://docs.microsoft.com/en-us/aspnet/core/api/microsoft.extensions.dependencyinjection.iservicecollection
[11] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection
[12] https://msdn.microsoft.com/en-us/library/f7ykdhsy(v=vs.110).aspx
[13] https://msdn.microsoft.com/en-US/library/bb397687.aspx

Register Types

Component instances registered with method RegisterType() created by reflection[14] and class constructor with the most number of parameters retrievable from container.

```
// Способ 1
builder.RegisterType<MyComponent>().As<IMyService>();

// Способ 2
builder.RegisterType(typeof(MyComponent)).As(typeof(IMyService));
```

Register Generic Types

If component presented as generic[15]-type to register one should use method RegisterGeneric(). As in case of regular types, instances of generic-components created by reflection[16] and class constructor with the most number of parameters retrievable from container.

```
public interface IRepository<T> { /* ... */ }

public class MyRepository<T> : IRepository<T> { /* ... */ }

// ...

builder.RegisterGeneric(typeof(MyRepository<>)).As(typeof(IRepository<>));
```

Register Instances

In some cases you may want to register an instance component created beforehand. For example, if creation of the component requires a lot of resources or is a technically complicated task. To register such components one should use method RegisterInstance().

```
builder.RegisterInstance(new MyComponent()).As<IMyService>();
```

Register Factory Functions

Component may be registered by a factory function or lambda[17]-expression. This way suits well when creation of component instance should be accompanied by preliminary calculations or is impossible to be created by class constructor. Such components should be registered via method RegisterFactory().

```
builder.RegisterFactory(r => new MyComponent()).As<IMyService>();
```

Input parameter r represents context of IoC-container (c. 12), which can be used to get all dependencies required to create component. This approach is the most fitting rather than obtaining dependencies via closure because this ensures a unified way of managing the life cycle of all dependencies.

```
builder.RegisterFactory(r => new A(r.Resolve<B>()));
```

Resolving dependencies

Once components registered (c. 8) they can be retrieved. Retrieving process of a single component instance by another using IoC-container is called dependency resolving. In InfinniPlatform apps all

[14] https://msdn.microsoft.com/en-us/library/f7ykdhsy(v=vs.110).aspx
[15] https://msdn.microsoft.com/en-US/library/512aeb7t.aspx
[16] https://msdn.microsoft.com/en-us/library/f7ykdhsy(v=vs.110).aspx
[17] https://msdn.microsoft.com/en-US/library/bb397687.aspx

dependencies passed via class constructors.

Resolving Direct Dependency

In most cases a direct dependency is defined between components. Next example component A depends on component B. In the very moment when app requests component A, first IoC-container creates component B then pass newly created component into constructor of component A. If component B depends on other components they will be created beforehand.

```
public class A
{
    private readonly B _b;

    public A(B b)
    {
        _b = b;
    }

    public void SomeMethod()
    {
        _b.DoSomething();
    }
}
```

Resolving Enumeration of Dependencies

Dependencies of an enumerable type provide multiple implementations of the same service (interface). Next example component A is dependant on all components of type B. All components of type B will be created and passed to component A via constructor as an instance of IEnumerable<T>[18].

```
public class A
{
    private readonly IEnumerable<B> _list;

    public A(IEnumerable<B> list)
    {
        _list = list;
    }

    public void SomeMethod()
    {
        foreach (var b in _list)
        {
            b.DoSomething();
        }
    }
}
```

Resolving with Delayed Instantiation

A lazy dependency is not instantiated until its first use. This appears where the dependency is infrequently used, or expensive to construct. To take advantage of this, use a Lazy<T>[19] in the constructor. Next example shows component A depends on component B but gets that dependency via lazy initialization while requesting a property Lazy<T>.Value[20] for the first time.

[18] https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1?view=netcore-1.1
[19] https://docs.microsoft.com/en-us/dotnet/api/system.lazy-1?view=netcore-1.1
[20] https://docs.microsoft.com/en-us/dotnet/api/system.lazy-1.value?view=netcore-1.1#System_Lazy_1_Value

```
public class A
{
    private readonly Lazy<B> _b;

    public A(Lazy<B> b)
    {
        _b = b;
    }

    public void SomeMethod()
    {
        _b.Value.DoSomething();
    }
}
```

Resolving Factory Functions

Using an auto-generated factory is applicable in case if it is required to create more than one instance of dependency or decision to create dependency can be done in runtime. Next example shows that component A depends on component B however it gets this dependency right before its usage.

```
public class A
{
    private readonly Func<B> _b;

    public A(Func<B> b)
    {
        _b = b;
    }

    public void SomeMethod()
    {
        var b = _b();

        b.DoSomething();
    }
}
```

Resolving Parameterized Factory Functions

Using an auto-generated factory is also applicable in case if there are strongly-typed parameters in the resolution function. Next example shows that component A depends on component B but gets this dependency right before its usage having passed to the factory function parameter values required to create component B.

```
public class A
{
    private readonly Func<int, B> _b;

    public A(Func<int, B> b)
    {
        _b = b;
    }

    public void SomeMethod()
    {
        var b = _b(42);

        b.DoSomething();
```

```
    }
}


public class B
{
    public B(int v) { /* ... */ }

    public void DoSomething() { /* ... */ }
}
```

If factory function has duplicate types in the input parameter list one should define its delegate.

```
public class A
{
    private readonly FactoryB _b;

    public A(FactoryB b)
    {
        _b = b;
    }

    public void SomeMethod()
    {
        var b = _b(42, 43);

        b.DoSomething();
    }
}


public class B
{
    public B(int v1, int v2) { /* ... */ }

    public void DoSomething() { /* ... */ }
}


public delegate B FactoryB(int v1, int v2);
```

Getting Direct Access to IoC Container

In case if it is required to make a universal factory of components which type is knowable in runtime, for example as in generic-type case, or working component logic depends on configuration of IoC-container, one can obtain a direct access to container using IContainerResolver. Next example shows component A acquires access to IoC-container because component type becomes known in runtime.

```
public class A
{
    private readonly IContainerResolver _resolver;

    public A(IContainerResolver resolver)
    {
        _resolver = resolver;
    }

    public void SomeMethod<T>()
    {
        var b = _resolver.Resolve<B<T>>();
```

```
        b.DoSomething();
    }
}


public class B<T>
{
    public void DoSomething() { /* ... */ }
}
```

Resolving dependencies at Runtime

The IContainerResolver interface lets get dependency by any of afore mentioned way. Resolve() serves those purposes and has two reloads.

```
// Way 1
IMyService myService = resolver.Resolve<IMyService>();

// Way 2
object myService = resolver.Resolve(typeof(IMyService));
```

If service is not registered, method Resolve() will throw an exception. This can be bypassed two ways, first one is to use method TryResolve().

```
// Way 1

IMyService myService;

if (resolver.TryResolve<IMyService>(out myService))
{
    // ...
}

// Way 2

object myService;

if (resolver.TryResolve(typeof(IMyService), out myService))
{
    // ...
}
```

Second is to use method ResolveOptional().

```
// Way 1

IMyService myService = resolver.ResolveOptional<IMyService>();

if (myService != null)
{
    // ...
}

// Way 2

object myService = resolver.ResolveOptional(typeof(IMyService));

if (myService != null)
{
    // ...
}
```

Checking registrations

To check the configuration of IoC-container one may call a list of registered services Services. To check the status of registration of a particular service one should use method IsRegistered().

```
// Way 1

if (resolver.IsRegistered<IMyService>())
{
    // ...
}

// Way 2

if (resolver.IsRegistered(typeof(IMyService)))
{
    // ...
}
```

Controlling Lifetime

Lifetime of the component is defined by the fact how long the component instance are available to use in application, from the moment of its creation and to the moment of its disposal (c. 15). Accordingly to lifetime of the InfinniPlatform app components may be divided into the following types:

- Created at each retrieving
- Created for the time of request processing
- Created for the time of app execution

If component has no internal state and being used during app execution then it makes sense to create shareable component instance at the start of app execution and dispose it in the end. Otherwise if component has an internal state but not bound by request processing such instance should be created before first call and be disposed right after its usage. It is recommended to created stateless components so it will decrease a number of error and reduce resources utilized.

Defining Component Lifetime

IoC-container performs automatic lifetime components control thus their lifetime is defined during registration (c. 8). All registered components are created each time they are received by default.

```
// Component will be created at each retrieving (by default)
builder.RegisterType<MyComponent>().As<IMyService>().InstancePerDependency();

// Component will be created for the time of HTTP-request execution
builder.RegisterType<MyComponent>().As<IMyService>().InstancePerLifetimeScope();

// Component will be created once for the time of the app execution
builder.RegisterType<MyComponent>().As<IMyService>().SingleInstance();
```

In the end of lifetime cycle IoC-container disposes (c. 15) component instance which makes it no longer available for further usage. This is the reason that definition of the lifetime must take into account their dependency. For example, component SingleInstance() is not able to directly be dependant on component InstancePerDependency().

Listing 3.1: Possible direct dependencies

Initial type	May refer to
InstancePerDependency()	• InstancePerDependency() • InstancePerLifetimeScope() • SingleInstance()
InstancePerLifetimeScope()	• InstancePerLifetimeScope() • SingleInstance()
SingleInstance()	• SingleInstance()

If component's lifetime is more than lifetime of the component it depends on to retrieve dependency one should use factory function (c. 11). Next example shows component A depends on component B but retrieves its dependency right before usage due to the fact that the lifetime of component A is longer than lifetime of component B.

```
builder.RegisterType<A>().AsSelf().SingleInstance();
builder.RegisterType<B>().AsSelf().InstancePerDependency();

// ...

public class A
{
    private readonly Func<B> _b;

    public A(Func<B> b)
    {
        _b = b;
    }

    public void SomeMethod()
    {
        var b = _b();

        b.DoSomething();
    }
}
```

Components Disposing

App may get resources which temporary created for the time of execution. For example a connection to a database, file stream an so on. .NET model offers IDisposable[21] interface which brings all resources to be disposed.

In the end of component lifetime IoC-container checks whether it implements IDisposable[22] interface and if it does then it calls method Dispose()[23]. Afterwards the current component instance becomes unavailable for further usage.

To deny automatic disposal one should directly call method ExternallyOwned(). This may be frequently used when the component lifetime is owned by external component.

```
public class DisposableComponent : IDisposable { /* ... */ }
```

[21] https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netcore-1.1
[22] https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netcore-1.1
[23] https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose?view=netcore-1.1#System_IDisposable_Dispose

```
// ...

builder.RegisterType<DisposableComponent>().ExternallyOwned();
```

Application Events

Some applications may require initialization at the start and deinitialization in the end of app execution. First this implies that app may have pre-defined settings, for instance allocation of particular resources, data migration, cache pre-filling and so on. Second, at the stage of deinitialization, a reverse process take place, that is disposing allocated resources. Both stages are optional and depend on app logic and resources it manipulates.

Application Events Types

InfinniPlatfrom apps may handle the following events:

- After the application host has fully started and is about to wait for a graceful shutdown
- After the application host has fully stopped and is not wait for new requests

When the application host has fully started you have a chance to execute any kind of background tasks such as pre-filling data cache (c. 73), data indexing and etc. When the application host has fully stopped you may dispose resources, save data retaining in memory and etc.

Note: You should pay attention that application may stop by exception or forcefully unloaded by administrative tools. Don't rely that aforementioned event handlers will be invoked anyway. Instead of this the restoration logic should be implemented to help handle those emergencies.

Application Event Handler

To handle the application events there are two type of handlers represented by IAppStartedHandler and IAppStoppedHandler interfaces. The first is invoked when the application has started, the last is invoked when the application has stopped. All you need is to implement an appropriate handler and register (c. 8) it in IoC container (c. 7).

Next example shows a handler which handles the application startup event.

```
public class MyAppStartedHandler : IAppStartedHandler
{
    public void Handle()
    {
        // App initialization code
    }
}

// ...

builder.RegisterType<MyAppStartedHandler>().As<IAppStartedHandler>().SingleInstance();
```

Asynchronous Event Handling

The application events is handled synchronously that is they don't return result until completed. Such behavior is intentionally predefined so the application could control the launch-stop-launch transitions on its own. For instance, in the case when status of event handling is unnecessary you may enclose event handling in try/catch block, nevertheless it is highly recommended to recorded exception into the application log (c. 23). If part of logics can be executed asynchronously it is recommended to run it in a new thread.

Note: It is the good practice when you minimize duration of the application start and stop. Accordingly this will improve the speed of app deployment and its re-launch.

Application Configuration

Developers and administrators use configuration files to customize their applications without programming. Usually configuration file is a text file which contains an application settings. But files are not the only way to configure applications there are a lot of other configuration sources at least command-line arguments and environment variables. The configuration API in ASP.NET Core[24] provides a way of configuring an app based on a list of name-value pairs that can be read at runtime from multiple sources. The name-value pairs can be grouped into a multi-level hierarchy. There are configuration providers for:

- File formats (INI[25], JSON[26], and XML[27])
- Command-line arguments
- Environment variables
- In-memory .NET objects
- An encrypted user store[28]
- Azure Key Vault[29]
- Custom providers[30]

Each configuration value maps to a string key. There's built-in binding support to deserialize settings into a custom POCO[31] object (a simple .NET class with properties). All of these tools can make your application highly flexible and simplify deploying and supporting.

Application Configuration File

Configuration files are one of the most common way to configure an application. Recently JSON[32] became popular format to representing an application settings. The settings are represented as JSON object with many defined properties. Properties of the first level are configuration sections which described by "key-value" pairs. Key is a name of property while value as a rule is a JSON object of any complexity. Each configuration section reflects settings of an subsystem or a particular component.

You can see an example of configuration file common structure below. This contains two sections section1 and section2, each one has its own set of properties. Section properties can be of any JSON compatible type (string type in example). Number, name and content of configuration section is defined by the app developer however there are a few pre-defined InfinniPlatform configuration sections.

[24] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration
[25] https://en.wikipedia.org/wiki/INI_file
[26] http://json.org/
[27] http://www.w3.org/XML/
[28] https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets
[29] https://docs.microsoft.com/en-us/aspnet/core/security/key-vault-configuration
[30] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration#custom-config-providers
[31] https://en.wikipedia.org/wiki/Plain_Old_CLR_Object
[32] http://json.org/

Listing 5.1: AppConfig.json

```
{
  "section1": {
    "Property11": "Value11",
    "Property12": "Value12",
    "Property13": "Value13"
  },
  "section2": {
    "Property21": "Value21",
    "Property22": "Value22",
    "Property23": "Value23",
  }
}
```

The following highlighted code hooks up the JSON file configuration provider to one source.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    private readonly IConfigurationRoot _configuration;

    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("AppConfig.json", optional: true, reloadOnChange: true);

        _configuration = builder.Build();
    }

    // ...
}
```

It's typical to have different configuration settings for different environments[33], for example, development, test and production. Thus you can improve previous example by adding a source of the environment.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    private readonly IConfigurationRoot _configuration;

    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("AppConfig.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"AppConfig.{env.EnvironmentName}.json", optional: true);

        _configuration = builder.Build();
    }

    // ...
}
```

See AddJsonFile()[34] for an explanation of the parameters.

[33] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments
[34] https://docs.microsoft.com/ru-ru/aspnet/core/api/microsoft.extensions.configuration.jsonconfigurationextensions

Note: Configuration sources are read in the order they are specified and the latest overrides previous.

After configuration settings it is time to define an application options. The options pattern uses custom options classes to represent a group of related settings. We recommended that you create decoupled classes for each feature within your app. Decoupled classes follow:

- The Interface Segregation Principle[35] : Classes depend only on the configuration settings they use.
- Separation of Concerns[36] : Settings for different parts of your app are not dependent or coupled with one another.

The options class must be non-abstract with a public parameterless constructor. For the abovementioned section1 an appropriate options class can have next form:

```
public class MyOptions
{
    public string Property11 { get; set; }
    public string Property12 { get; set; }
    public string Property13 { get; set; }
}
```

There is a way reading options directly nevertheless the more elegant method is using dependency injection (c. 7) mechanism.

```
// Direct reading of the configuration section
MyOptions options = _configuration.GetSection("section1").Get<MyOptions>();
```

In the following code, the MyOptions class is added to the service container and bound to configuration.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    private readonly IConfigurationRoot _configuration;

    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("AppConfig.json", optional: true, reloadOnChange: true)
            .AddJsonFile($"AppConfig.{env.EnvironmentName}.json", optional: true);

        _configuration = builder.Build();
    }

    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        // Register the configuration section which MyOptions binds against
        services.Configure<MyOptions>(_configuration.GetSection("section1"));

        // ...
    }

    // ...
}
```

The following component uses dependency injection (c. 7) on IOptions<TOptions>[37] to access settings:

[35] https://en.wikipedia.org/wiki/Interface_segregation_principle
[36] https://en.wikipedia.org/wiki/Separation_of_concerns
[37] https://docs.microsoft.com/en-us/aspnet/core/api/microsoft.extensions.options.ioptions-1

```
public class MyComponent
{
    private readonly MyOptions _options;

    public MyComponent(IOptions<MyOptions> optionsAccessor)
    {
        _options = optionsAccessor.Value;
    }

    // ...
}
```

Environment Variables

Environment Variables are yet another popular way to configure an application.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    private readonly IConfigurationRoot _configuration;

    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("AppConfig.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"AppConfig.{env.EnvironmentName}.json", optional: true)
                .AddEnvironmentVariables();

        _configuration = builder.Build();
    }

    // ...
}
```

Configuration sources are read in the order they are specified. In the code above, the environment variables are read last. Any configuration values set through the environment would replace those set in the two previous providers.

Note: A best practice is to specify environment variables last, so that the local environment can override anything set in deployed configuration files.

Monitoring and Diagnostics

InfinniPlatform is fully integrated with ASP.NET Core and supports a logging API that works with a variety of logging providers. Built-in providers let you send logs to one or more destinations, and you can plug in a third-party logging framework. This article shows how to use the logging API and providers in your code.

Using ILogger<T>

To create logs, get an ILogger<T>[38] object using dependency injection (с. 7) mechanism and store it in a field, then call logging methods on that logger object.

```
public class MyComponent
{
    private readonly ILogger<MyComponent> _logger;

    public MyComponent(ILogger<MyComponent> logger)
    {
        _logger = logger;
    }

    public void DoSomethig()
    {
        _logger.LogInformation("Hello!");
    }
}
```

Using IPerformanceLogger<T>

To log performance metrics you can use IPerformanceLogger<T>. Using this component is as easy as using ILogger<T>[39]. The implementation of the IPerformanceLogger<T> is based on ILogger<T>[40] and use typeof(IPerformanceLogger<T>) as a logger category (a pointer to an event source). So you can recognize log events of the IPerformanceLogger<T> and route them to a separate writer.

```
public class MyComponent
{
    private readonly IPerformanceLogger<MyComponent> _perfLogger;

    public MyComponent(IPerformanceLogger<MyComponent> perfLogger)
    {
        _perfLogger = perfLogger;
```

[38] https://docs.microsoft.com/ru-ru/aspnet/core/api/microsoft.extensions.logging.ilogger-1
[39] https://docs.microsoft.com/ru-ru/aspnet/core/api/microsoft.extensions.logging.ilogger-1
[40] https://docs.microsoft.com/ru-ru/aspnet/core/api/microsoft.extensions.logging.ilogger-1

```
    }

    public void DoSomethig()
    {
        var startTime = DateTime.Now;

        Exception exception = null;

        try
        {
            // Some code which duration is logged
        }
        catch (Exception e)
        {
            exception = e;
        }
        finally
        {
            _perfLogger.Log(nameof(DoSomethig), startTime, exception);
        }
    }
}
```

Note: The above example is quite wordy and inconvenient. Nonetheless you can reduce this difficulties using, for instance, the Delegation pattern[41] or some kind of AOP[42] tools.

The Logger Name

By default ILogger<T>[43] uses T as the type who's name is used for the logger name. And with it the logger name is produced as the fully qualified name of the T, including its namespace and excluding any generic arguments even if they exist. So if you, for instance, have two classes A and its generic analogue A<T>, the logger names for the both will be the same. In practice it is inconvenient because it can be difficult to define who is the real source of an event without looking at the code. Also if the component has a unique name using its full name is unnecessary and decreases readability of the logs.

To solve above problems InfinniPlatform integrates with the logging API and sets the own rules of building logger names. According to these rules the logger name of the ILogger<T> is produced as C#-like representation of the type T, including its generic arguments if they exist.

For example, say you have the MyComponent type as below which tries to get the logger for itself. In this case the logger name will be Namespace.To.The.MyComponent and that is no differences with the default logic.

```
namespace Namespace.To.The
{
    class MyComponent
    {
        public MyComponent(ILogger<MyComponent> logger)
        {
            // ...
        }

        // ...
    }
}
```

[41] https://en.wikipedia.org/wiki/Delegation_pattern
[42] https://en.wikipedia.org/wiki/Aspect-oriented_programming
[43] https://docs.microsoft.com/ru-ru/aspnet/core/api/microsoft.extensions.logging.ilogger-1

Differences begin with generic types. Suppose the MyComponent is generic. In this case the logger name depends on T and will be Namespace.To.The.MyComponent<T> where T is C#-like representation of the type T. For example, the name of the ILogger<MyComponent<SomeType>> will be Namespace.To.The.MyComponent<Other.Namespace.To.The.SomeType>. Thus you know exactly which component is the event source and can make right decision looking at the event.

```
namespace Namespace.To.The
{
    class MyComponent<T>
    {
        public MyComponent(ILogger<MyComponent<T>> logger)
        {
            // ...
        }

        // ...
    }
}
```

Note: In any case the logger name for generic types will contain information about generic arguments because it can be important during analysis.

LoggerNameAttribute

Some components can have unique name or represent known abstraction in your system so there are no reasons to have full qualified name for the logger of these components. In these cases you can use the LoggerNameAttribute and define the own component name.

Note: The LoggerNameAttribute can be useful feature during refactoring which includes renaming of the types. Also it force you think harder when you choose a name for your component and imagine how it will be represented in the log.

In the next example the name of the ILogger<MyComponent> will be MySubsystem.

```
namespace Namespace.To.The
{
    [LoggerName("MySubsystem")]
    class MyComponent
    {
        public MyComponent(ILogger<MyComponent> logger)
        {
            // ...
        }

        // ...
    }
}
```

In case of the generic types the behavior is similar. For example, the name of the ILogger<MyComponent<SomeType>> will be MySubsystem<Other.Namespace.To.The.SomeType>.

```
namespace Namespace.To.The
{
    [LoggerName("MySubsystem")]
    class MyComponent<T>
    {
        public MyComponent(ILogger<MyComponent<T>> logger)
        {
```

```
            // ...
        }

        // ...
    }
}
```

Also you can apply LoggerNameAttribute to SomeType then the logger name will be shorter - MySubsystem<SomeType>.

```
[LoggerName("SomeType")]
class SomeType
{
    // ...
}
```

How to configure Serilog

Note: Serilog[44] is one of the most popular logging framework and has lots of ways to configuration. Here is one of them and we give it as an example.

Serilog[45] provides sinks for writing log events to storage in various formats. In our example we split log events by two streams. The first - the application event log - catches all events, excepting events of IPerformanceLogger<T>. The second - the application performance log - catches the only events of IPerformanceLogger<T>. The first stream writes to one file the second to another, both use the Serilog.Sinks.RollingFile.RollingFileSink.

1. Define the log output template.

```
string outputTemplate =
    "{Timestamp:o}|{Level:u3}|{RequestId}|{UserName}|{SourceContext}|{Message}{NewLine}{Exception}";
```

This template uses a number of built-in properties:

TimeStamp The event's timestamp, as a DateTimeOffset (o means using ISO 8601, see format strings and properties[46]).

Level The log event level, formatted as the full level name. For more compact level names, use a format such as {Level:u3} or {Level:w3} for three-character upper- or lowercase level names, respectively.

SourceContext The logger name (c. 24). Usually it is full name of the component type who is the event source.

Message The log event's message, rendered as plain text.

NewLine A property with the value of System.Environment.NewLine[47].

Exception The full exception message and stack trace, formatted across multiple lines.

Also there are two our properties which will be described later:

RequestId The unique identifier of the HTTP request during which the event occurred.

UserName The user name of the HTTP request during which the event occurred.

[44] https://serilog.net/
[45] https://serilog.net/
[46] https://docs.microsoft.com/en-us/dotnet/api/system.globalization.datetimeformatinfo?view=netcore-1.1
[47] https://docs.microsoft.com/en-us/dotnet/api/system.environment.newline?view=netcore-1.1#System_Environment_NewLine

2. Define the function which splits the application event log and the application performance log:

```
Func<LogEvent, bool> performanceLoggerFilter =
    Matching.WithProperty<string>(
        Constants.SourceContextPropertyName,
        p => p.StartsWith(nameof(IPerformanceLogger)));
```

3. Configure Serilog[48] logger:

```
Log.Logger = new LoggerConfiguration()
    // Configures the minimum level - Information
    .MinimumLevel.Information()
    // It will be described later
    .Enrich.With(new HttpContextLogEventEnricher(httpContextAccessor))
    // Writes log events to Console (all events)
    .WriteTo.LiterateConsole(outputTemplate: outputTemplate)
    // The application event log
    .WriteTo.Logger(lc => lc.Filter.ByExcluding(performanceLoggerFilter)
                          .WriteTo.RollingFile("logs/events-{Date}.log",
                                               outputTemplate: outputTemplate))
    // The application performance log
    .WriteTo.Logger(lc => lc.Filter.ByIncludingOnly(performanceLoggerFilter)
                          .WriteTo.RollingFile("logs/performance-{Date}.log",
                                               outputTemplate: outputTemplate))
    // Create a logger using the above configuration
    .CreateLogger();
```

4. Add to the Startup class registration Serilog[49]:

```
public class Startup
{
    // ...

    public void Configure(IApplicationBuilder app,
                          IContainerResolver resolver,
                          ILoggerFactory loggerFactory,
                          IApplicationLifetime appLifetime,
                          IHttpContextAccessor httpContextAccessor)
    {
        // Configure logger (see above)...

        // Register Serilog
        loggerFactory.AddSerilog();

        // Ensure any buffered events are sent at shutdown
        appLifetime.ApplicationStopped.Register(Log.CloseAndFlush);

        // ...
    }
}
```

5. Declare the HttpContextLogEventEnricher class which provides RequestId and UserName properties.

```
using Microsoft.AspNetCore.Http;

using Serilog.Core;
using Serilog.Events;

class HttpContextLogEventEnricher : ILogEventEnricher
{
    private const string RequestIdProperty = "RequestId";
```

[48] https://serilog.net/
[49] https://serilog.net/

```
    private const string UserNameProperty = "UserName";


    public HttpContextLogEventEnricher(IHttpContextAccessor httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor;
    }


    private readonly IHttpContextAccessor _httpContextAccessor;


    public void Enrich(LogEvent logEvent, ILogEventPropertyFactory propertyFactory)
    {
        var context = _httpContextAccessor.HttpContext;

        if (context != null)
        {
            var requestId = context.TraceIdentifier ?? "";
            var requestIdProperty = propertyFactory.CreateProperty(RequestIdProperty, requestId);
            logEvent.AddPropertyIfAbsent(requestIdProperty);

            var userName = context.User?.Identity?.Name ?? "";
            var userNameProperty = propertyFactory.CreateProperty(UserNameProperty, userName);
            logEvent.AddPropertyIfAbsent(userNameProperty);
        }
    }
}
```

Here's an example of what you see in the the application event log:

Listing 6.1: logs/events-20170609.log

```
2017-06-09T17:01:50.1335297+05:00|INF|0HL5F5T7R11QM||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request starting H
2017-06-09T17:01:50.6985948+05:00|INF|0HL5F5T7R11QM||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request finished in
2017-06-09T17:01:50.7546117+05:00|INF|0HL5F5T7R11QN||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request starting H
2017-06-09T17:01:50.7856106+05:00|INF|0HL5F5T7R11QN||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request finished in
2017-06-09T17:01:54.3309882+05:00|INF|0HL5F5T7R11QO||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request starting H
2017-06-09T17:01:54.4770051+05:00|INF|0HL5F5T7R11QO||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request finished in
2017-06-09T17:01:54.5060098+05:00|INF|0HL5F5T7R11QP||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request starting H
2017-06-09T17:01:54.5140225+05:00|INF|0HL5F5T7R11QP||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request finished in
2017-06-09T17:01:57.8511674+05:00|INF|0HL5F5T7R11QQ||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request starting H
2017-06-09T17:01:58.0921584+05:00|INF|0HL5F5T7R11QQ||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request finished in
2017-06-09T17:01:58.1196721+05:00|INF|0HL5F5T7R11QR||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request starting H
2017-06-09T17:01:58.1281824+05:00|INF|0HL5F5T7R11QR||Microsoft.AspNetCore.Hosting.Internal.WebHost|Request finished in
```

And in the application performance log:

Listing 6.2: logs/performance-20170609.log

```
2017-06-09T17:01:54.4129957+05:00|INF|0HL5F5T7R11QO||IPerformanceLogger<JobScheduler>|{ "IsStarted": 2 }
2017-06-09T17:01:54.4179953+05:00|INF|0HL5F5T7R11QO||IPerformanceLogger<JobScheduler>|{ "GetStatus": 1 }
2017-06-09T17:01:54.4189952+05:00|INF|0HL5F5T7R11QO||IPerformanceLogger<JobScheduler>|{ "GetStatus": 0 }
2017-06-09T17:01:54.4204952+05:00|INF|0HL5F5T7R11QO||IPerformanceLogger<JobScheduler>|{ "GetStatus": 0 }
2017-06-09T17:01:54.4250023+05:00|INF|0HL5F5T7R11QO||IPerformanceLogger<IHttpService>|{ "GET::/info/scheduler": 91 }
2017-06-09T17:01:54.4740019+05:00|INF|0HL5F5T7R11QO||IPerformanceLogger<GlobalHandlingAppLayer>|{ "GET::/info/sch
2017-06-09T17:01:54.5090235+05:00|INF|0HL5F5T7R11QP||IPerformanceLogger<IHttpService>|{ "GET::/{id}": 0 }
2017-06-09T17:01:54.5120237+05:00|INF|0HL5F5T7R11QP||IPerformanceLogger<GlobalHandlingAppLayer>|{ "GET::/favicon.i
2017-06-09T17:01:58.0791573+05:00|INF|0HL5F5T7R11QQ||IPerformanceLogger<JobScheduler>|{ "GetStatus": 1 }
2017-06-09T17:01:58.0811566+05:00|INF|0HL5F5T7R11QQ||IPerformanceLogger<IHttpService>|{ "GET::/scheduler/jobs": 221
2017-06-09T17:01:58.0891579+05:00|INF|0HL5F5T7R11QQ||IPerformanceLogger<GlobalHandlingAppLayer>|{ "GET::/schedule
2017-06-09T17:01:58.1231732+05:00|INF|0HL5F5T7R11QR||IPerformanceLogger<IHttpService>|{ "GET::/{id}": 0 }
```

2017-06-09T17:01:58.1261729+05:00|INF|0HL5F5T7R11QR||IPerformanceLogger<GlobalHandlingAppLayer>|{ "GET::/favicon.i

Data Serialization

Data Serialization is one of the main parts when we talk about transferring data via network. InfinniPlatfrom provides necessary instruments to serialize and deserialize data using JSON format. Also there are abstractions for extend and customize this mechanism.

InfinniPlatfrom uses Json.NET[50] - popular high-performance JSON framework for .NET. So you get all capabilities and features this library. Besides InfinniPlatfrom provides a range high-level abstractions which are well integrated with other services of InfinniPlatfrom. This basic and powerful mechanism is used in most of cases especially on the system layer but at the same time you do not need to care about it usually.

Data Serialization is represented as JsonObjectSerializer class which implements two interfaces from the same namespace - IObjectSerializer and IJsonObjectSerializer. The first - IObjectSerializer declares common methods of serializers. The second - IJsonObjectSerializer extends the first and contains few special methods which are appropriate to JSON format.

JsonObjectSerializer class is thread-safe and it has two singleton instances - Default and Formatted. Default instance uses UTF-8 encoding (without BOM) and serializes objects without formatting. Formatted instance is the same as Default but serializes objects with formatting (for getting easy-to-read JSON). Both of them do not have any other settings which you can pass to the constructor of JsonObjectSerializer. You can use Default and Formatted instances explicitly but if you have access to IoC Container (c. 7) we strongly recommend getting instance IJsonObjectSerializer via IoC. For example acquire IJsonObjectSerializer through a constructor (c. 9) of the class where you need this dependency. It allows to use the same settings of the data serialization in an application and customize them in one place.

Serialization Attributes

Attributes can be used to control how JsonObjectSerializer serializes and deserializes .NET objects.

NonSerializedAttribute

The NonSerializedAttribute[51] excludes a field or property from serialization. By default public fields and properties are included to serialization. Standard .NET serialization attribute NonSerializedAttribute[52] allows to exclude specific properties from a resultant JSON. Usually it is useful to exclude properties which can be calculated from other fields.

```
public class Person
{
    public string FirstName { get; set; }
```

[50] http://www.newtonsoft.com/json
[51] https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute?view=netcore-1.1
[52] https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute?view=netcore-1.1

```
    public string LastName { get; set; }

    public DateTime Birthday { get; set; }

    [NonSerialized]
    public int Age { get; set; }
}
```

SerializerVisibleAttribute

The SerializerVisibleAttribute includes a field or property to serialization. By default non-public fields and properties as well as properties with non-public setters (or without them) are excluded from serialization. This attribute is the opposite NonSerializedAttribute[53].

```
public class Document
{
    [SerializerVisible]
    public object _id { get; internal set; }

    [SerializerVisible]
    public DocumentHeader _header { get; internal set; }
}
```

SerializerPropertyNameAttribute

The SerializerPropertyNameAttribute sets specified property name while serialization. By default, the JSON property will have the same name as the .NET property. This attribute allows the name to be customized.

```
public class Person
{
    [SerializerPropertyName("forename")]
    public string FirstName { get; set; }

    [SerializerPropertyName("surname")]
    public string LastName { get; set; }
}
```

Serialization Known Types

By default JsonObjectSerializer does not include any type information into resultant JSON. So if a serializable type contains a property with an abstract type serialization will be successful but not deserialization. It is because JsonObjectSerializer does not have any information about specific type of the property.

```
public interface I
{
}


public class A : I
{
    public string PropertyA { get; set; }
}
```

[53] https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute?view=netcore-1.1

```
public class B : I
{
    public string PropertyB { get; set; }
}


public class C
{
    public I Property1 { get; set; }
    public I Property2 { get; set; }
}


var value = new C
            {
                Property1 = new A { PropertyA = "ValueA" },
                Property2 = new B { PropertyB = "ValueB" }
            };

var serializer = new JsonObjectSerializer(withFormatting: true);

var json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{
//  "Property1": {
//    "PropertyA": "ValueA"
//  },
//  "Property2": {
//    "PropertyB": "ValueB"
//  }
//}

serializer.Deserialize<C>(json);
// JsonSerializationException: Could not create an instance of type I.
// Type is an interface or abstract class and cannot be instantiated.
```

To solve this problem you can use IKnownTypesSource and pass it into the JsonObjectSerializer constructor directly or via IoC Container (c. 7). Known types allow to include type information into resultant JSON during serialization and rely on it during deserialization. All you need is to add an unique alias for each type which can be use as value of a property with abstract type.

```
class MyKnownTypesSource : IKnownTypesSource
{
    public void AddKnownTypes(KnownTypesContainer knownTypesContainer)
    {
        knownTypesContainer
            .Add<A>("A")
            .Add<B>("B");
    }
}

// ...

var value = new C
            {
                Property1 = new A { PropertyA = "ValueA" },
                Property2 = new B { PropertyB = "ValueB" }
            };

var serializer = new JsonObjectSerializer(withFormatting: true,
                    knownTypes: new[] { new MyKnownTypesSource() });
```

```
var json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{
//  "Property1": {
//    "A": {
//      "PropertyA": "ValueA"
//    }
//  },
//  "Property2": {
//    "B": {
//      "PropertyB": "ValueB"
//    }
//  }
//}

var result = serializer.Deserialize<C>(json);

Console.WriteLine(((A)result.Property1).PropertyA);
// ValueA

Console.WriteLine(((B)result.Property2).PropertyB);
// ValueB
```

Serialization Converters

In some cases JSON view of an object must have a little different representation than it is described in the object type. The differences can be related to the data schema or the data type of certain fields or properties. There are many reasons for that. For example to store data in a database or to transfer data by network it is more convenient to use different representation than it is described in the data type. Also perhaps you cannot change a data type because of using an external library or you have to use different format because of communicating with an external system.

Serialization Converters provide a way to customize how an object will be serialized and deserialized. For that you need to implement the interface IMemberValueConverter and pass it into the JsonObjectSerializer constructor directly or via IoC Container (с. 7).

Note: The XmlDateMemberValueConverter implements IMemberValueConverter for DateTime members which have XmlElementAttribute[54] with DataType property equals date. In this case we have to use only the date part of DateTime value (without the time part). The XmlDateMemberValueConverter handles these cases and converts DateTime value to Date which can be serialized as the Unix time. It can be useful during integration with SOAP services.

The IMemberValueConverter has three methods:

- CanConvert() - Checks whether this converter can be applied to specified member.
- Convert() - Converts an original member value to new format during serialization.
- ConvertBack() - Converts an original member value back from new format during deserialization.

Next example converts all DateTime members to the Unix time during serialization and then converts them back to DateTime during deserialization.

```
public class UnixDateTimeConverter : IMemberValueConverter
{
    private static readonly DateTime UnixTimeZero
```

[54] https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlelementattribute?view=netcore-1.1

```
        = new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc);


    public bool CanConvert(MemberInfo member)
    {
        var property = member as PropertyInfo;

        return (property != null)
            && (property.PropertyType == typeof(DateTime)
                || property.PropertyType == typeof(DateTime?));
    }

    public object Convert(object value)
    {
        var date = value as DateTime?;

        if (date != null)
        {
            var unixTime = (long)date.Value.Subtract(UnixTimeZero).TotalSeconds;

            return unixTime;
        }

        return null;
    }

    public object ConvertBack(Func<Type, object> value)
    {
        var unixTime = (long?)value(typeof(long?));

        if (unixTime != null)
        {
            var date = UnixTimeZero.AddSeconds(unixTime.Value);

            return date;
        }

        return null;
    }
}


public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public DateTime Birthday { get; set; }
}


var value = new Person
            {
                FirstName = "John",
                LastName = "Smith",
                Birthday = new DateTime(2000, 1, 1)
            };

var valueConverters = new IMemberValueConverter[]
                      {
                          new UnixDateTimeConverter()
                      };
```

```
var serializer = new JsonObjectSerializer(withFormatting: true, valueConverters: valueConverters);

var json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{
//  "FirstName": "John",
//  "LastName": "Smith",
//  "Birthday": 946684800
//}

var result = serializer.Deserialize<Person>(json);

Console.WriteLine("{0:yyyy/MM/dd}", result.Birthday);
//2000/01/01
```

Serialization Error Handling

InfinniPlatform supports error handling during serialization and deserialization. Error handling lets you catch an error and choose whether to handle it and continue with serialization or let the error bubble up and be thrown in your application.

To handle serialization errors you need to implement the interface ISerializerErrorHandler and pass it into the JsonObjectSerializer constructor directly or via IoC Container (с. 7). The ISerializerErrorHandler has the only one method Handle(). It is called whenever an exception is thrown while serializing or deserializing JSON.

Note: The IgnoreSerializerErrorHandler implements ISerializerErrorHandler and ignores all exceptions. This handler allows to skip properties whose getters and setters can throw exceptions.

Next example ignores all exceptions during serialization and deserialization.

```
public class IgnoreSerializerErrorHandler : ISerializerErrorHandler
{
    public bool Handle(object target, object member, Exception error)
    {
        return true;
    }
}


public class BadGetter
{
    public string Property1
    {
        get;
        set;
    }

    public string Property2
    {
        get { throw new Exception(); }
        set { }
    }
}


var value = new BadGetter
```

```
        {
            Property1 = "Value1",
            Property2 = "Value2"
        };

var errorHandlers = new ISerializerErrorHandler[]
                {
                    new IgnoreSerializerErrorHandler()
                };

var serializer = new JsonObjectSerializer(withFormatting: true, errorHandlers: errorHandlers);

var json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{
//  "Property1": "Value1"
//}
```

Serialization Dates and Times

The problem comes from the JSON spec itself: there is no literal syntax for dates in JSON. The spec has objects, arrays, strings, integers, and floats, but it defines no standard for what a date looks like. The default format used by JsonObjectSerializer is the ISO 8601[55] standard.

```
var value = new Person
        {
            FirstName = "John",
            LastName = "Smith",
            Birthday = new DateTime(2000, 1, 2, 3, 4, 5) // It will be serialized as ISO 8601
        };

var serializer = new JsonObjectSerializer(withFormatting: true);

var json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{
//  "FirstName": "John",
//  "LastName": "Smith",
//  "Birthday": "2000-01-02T03:04:05"
//}
```

But sometimes we need to work only with either date part or time part. For these goals there are two special types: Date and Time. The JsonObjectSerializer supports these types and serializes them using next rules.

- Date is serialized as a 64-bit signed integer[56] which is the Unix time, defined as the number of seconds that have elapsed since 00:00:00 (UTC), 1 January 1970.
- Time is serialized as a double-precision floating-point number[57] which is the number of seconds that have elapsed since 00:00:00.

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
```

[55] https://en.wikipedia.org/wiki/ISO_8601
[56] https://docs.microsoft.com/en-us/dotnet/api/system.int64?view=netcore-1.1
[57] https://docs.microsoft.com/en-us/dotnet/api/system.double?view=netcore-1.1

```
    public Date BirthDay { get; set; }
    public Time BirthTime { get; set; }
}

var value = new Person
            {
                FirstName = "John",
                LastName = "Smith",
                BirthDay = new Date(2000, 1, 2),
                BirthTime = new Time(3, 4, 5)
            };

var serializer = new JsonObjectSerializer(withFormatting: true);

var json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{
//  "FirstName": "John",
//  "LastName": "Smith",
//  "BirthDay": 946771200,
//  "BirthTime": 11045.0
//}
```

Serialization Dynamic Objects

The JsonObjectSerializer supports dynamic objects (с. 5).

```
var value = new DynamicWrapper
            {
                { "FirstName", "John" },
                { "LastName", "Smith" },
                { "Birthday", new DateTime(2000, 1, 2, 3, 4, 5) }
            };

var serializer = new JsonObjectSerializer(withFormatting: true);

var json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{
//  "FirstName": "John",
//  "LastName": "Smith",
//  "Birthday": "2000-01-02T03:04:05"
//}

dynamic result = serializer.Deserialize(json);

Console.WriteLine(result.FirstName);
//John

Console.WriteLine(result.LastName);
//Smith

Console.WriteLine(result.Birthday);
//1/2/2000 3:04:05 AM
```

Reducing Serialized JSON Size

One of the common problems encountered when serializing .NET objects to JSON is that the JSON ends up containing a lot of unwanted properties and values. This can be especially significant when returning JSON to the client. More JSON means more bandwidth and a lower performance. To solve the issue of unwanted JSON, InfinniPlatform has a range of built-in options to fine-tune what gets written from a serialized object.

By default public fields and properties are included to serialization. Adding the NonSerializedAttribute[58] to a property tells the serializer to always skip writing it to the JSON result.

```
public class Person
{
    // Included in JSON
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthday { get; set; }

    // Ignored
    [NonSerialized]
    public int Age { get; set; }
}
```

If a class has many properties and you only want to serialize a small subset of them, then adding NonSerializedAttribute[59] to all the others will be tedious and error prone. The way to solve this scenario is to add the DataContractAttribute[60] to the class and DataMemberAttribute[61] to the properties to serialize. Only the properties you mark up will be serialized.

```
[DataContract]
public class Person
{
    // Included in JSON
    [DataMember]
    public string FirstName { get; set; }
    [DataMember]
    public string LastName { get; set; }
    [DataMember]
    public DateTime Birthday { get; set; }

    // Ignored
    public int Age { get; set; }
}
```

Also you can change property names and make them shorter using SerializerPropertyNameAttribute (c. 32) (but it can influence on readability your JSON).

```
public class Person
{
    [SerializerPropertyName("fn")]
    public string FirstName { get; set; }
    [SerializerPropertyName("ln")]
    public string LastName { get; set; }
    [SerializerPropertyName("bd")]
    public DateTime Birthday { get; set; }
}
```

The JsonObjectSerializer allows to format JSON which is easy-to-read. It is great for readability when you are developing. Disabling formatting on the other hand keeps the JSON result small, skipping all

[58] https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute?view=netcore-1.1
[59] https://docs.microsoft.com/en-us/dotnet/api/system.nonserializedattribute?view=netcore-1.1
[60] https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datacontractattribute?view=netcore-1.1
[61] https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datamemberattribute?view=netcore-1.1

unnecessary spaces and line breaks to produce the most compact and efficient JSON possible.

```
var value = new Person
            {
                FirstName = "John",
                LastName = "Smith"
            };

var serializer = new JsonObjectSerializer(withFormatting: true);

var json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{
//  "FirstName": "John",
//  "LastName": "Smith"
//}

serializer = new JsonObjectSerializer(withFormatting: false);

json = serializer.ConvertToString(value);

Console.WriteLine(json);
//{"FirstName":"John","LastName":"Smith"}
```

For more complex cases you can use serialization converters (с. 34) which provide a way to customize how an object will be serialized and deserialized, including changing serialization behavior at runtime.

HTTP Services

InfinniPlatform provides lightweight customizable API for building HTTP based services. With it you can handle GET, POST, PUT, PATCH and DELETE requests. It is very easy to create new HTTP service just look at the following code.

```
class MyHttpService : IHttpService
{
    public void Load(IHttpServiceBuilder builder)
    {
        builder.Get["/hello"] = async request =>
            await Task.FromResult("Hello from InfinniPlatform!");
    }
}
```

Defining Modules

Modules are key concept and the one thing which you have to know to develop HTTP services. A module is created by inheriting from the IHttpService interface. Each module implements the Load() method where you can define the behaviors of your HTTP service, in the form of routes and the actions they should perform if they are invoked.

The Load() method gets an instance of the IHttpServiceBuilder interface which with using DSL[62] style allows define routes and their actions. Each action gets an information about request and handles it asynchronously. Requests are represented as the IHttpRequest interface and give a comprehensive information to their handling.

Next example shows the registration of two handlers for GET requests.

```
public class MyHttpService : IHttpService
{
    public void Load(IHttpServiceBuilder builder)
    {
        builder.Get["/resource1"] = request => Task.FromResult<object>("Resource1");
        builder.Get["/resource2"] = request => Task.FromResult<object>("Resource2");
    }
}
```

Note: The Load() method will be invoked only once on an application startup.

Modules can be declared anywhere you like just register them in IoC Container (с. 7).

[62] https://en.wikipedia.org/wiki/Domain-specific_language

```
builder.RegisterType<MyHttpService>()
       .As<IHttpService>()
       .SingleInstance();
```

Note: All modules should be registered as a single instance (c. 14).

If you have lots of modules in an assembly you can register them all using RegisterHttpServices().

```
builder.RegisterHttpServices(assembly);
```

Asynchronous Handling

Request handlers are asynchronous by default so you can use async/await keywords.

```
public class MyHttpService : IHttpService
{
    public void Load(IHttpServiceBuilder builder)
    {
        builder.Get["/resource1"] = OnResource1;
        builder.Get["/resource2"] = OnResource2;
    }

    private async Task<object> OnResource1(IHttpRequest request)
    {
        // Do something asynchronously
        return await Task.FromResult<object>("Resource1");
    }

    private async Task<object> OnResource2(IHttpRequest request)
    {
        // Do something asynchronously
        return await Task.FromResult<object>("Resource2");
    }
}
```

Declaring Service Path

Usually modules combine some common functionality which are available on the same base path. So you can define a module path and each route will be subordinate to the path of the module. This saves you from having to repeat the common parts of the route patterns and also to nicely group your routes together based on their relationship.

```
public class MyHttpService : IHttpService
{
    public void Load(IHttpServiceBuilder builder)
    {
        builder.ServicePath = "/base/path/to";
        builder.Get["/resource1"] = OnResource1;
        builder.Get["/resource2"] = OnResource2;
    }

    // ...
}
```

Defining Routes

Routes are defined in the Load() method. In order to define a route you need to specify a Method + Pattern + Action.

```
public class ProductsHttpService : IHttpService
{
    public void Load(IHttpServiceBuilder builder)
    {
        builder.Get["/products/{id}"] = async request =>
        {
            // Do something
        };
    }
}
```

Method

The Method is the HTTP method[63] that is used to access the resource. You can handle GET, POST, PUT, PATCH and DELETE methods. The IHttpServiceBuilder interface contains a definition for each of these methods.

Pattern

The Pattern declares the application-relative URL that the route answers to.

- Literal segment, /some, requires an exact match.
- Capture segment, /{name}, captures whatever is passed into the given segment.
- Capture optional segment, /{name?}, by adding ? at the end of the segment name the segment can be made optional.
- Capture optional/default segment, /{name?unnamed}, by adding a value after ? we can turn an optional segment into a segment with a default value.
- RegEx segment, /(?<id>[\d]{1,2}), using Named Capture Grouped[64] Regular Expressions, you can get a little more control out of the segment pattern.
- Greedy segment, /{name*}, by adding * to the end of the segment name, the pattern will match any value from the current forward slash onward.
- Multiple captures segment, /{file}.{extension} or /{file}.ext, a segment containing a mix of captures and literals.

Pattern segments can be combined, in any order, to create a complex Pattern for a route.

Note: It's worth noting that capture segments are greedy, meaning they will match anything in the requested URL until another segment matches or until the end of the URL is reached. Sometimes you may end up with two routes which end up giving a positive match. Each pattern has a score which is used to resolve the conflicts. But we do not recommend to use conflicted routes.

Action

A route Action is the behavior which is invoked when a request is matched to a route. It is represented as a delegate of type Func<IHttpRequest,Task<object>> where the input argument is an information

[63] https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
[64] http://www.regular-expressions.info/named.html

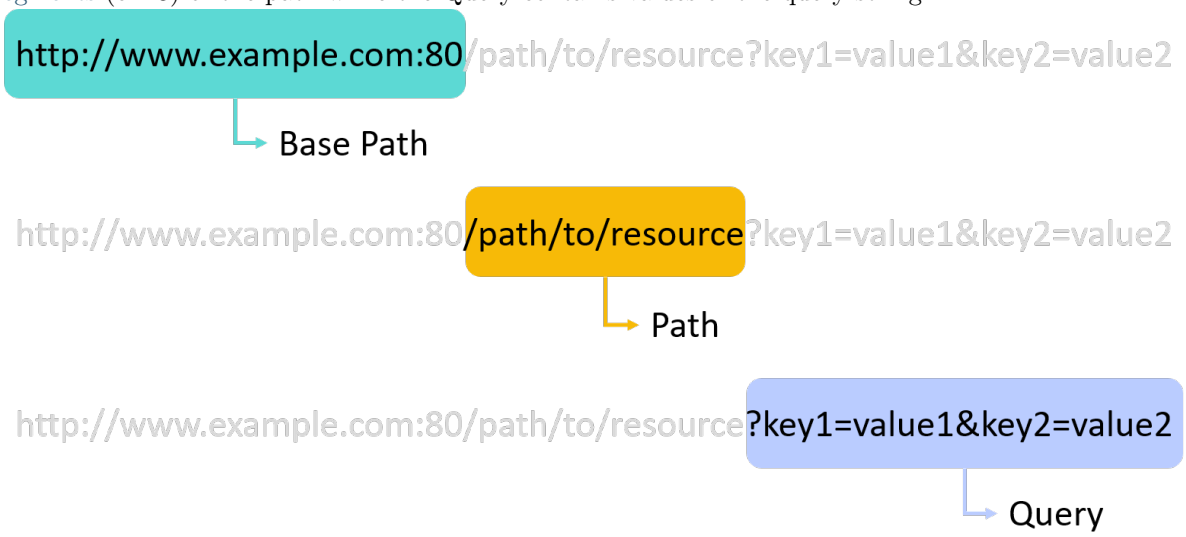
about request (IHttpRequest) and the result is a task (Task<object>[65]) to retrieving response. The response can be any model (c. 46).

Request Handling

Actions handle requests getting an instance of the IHttpRequest which gives a comprehensive information about current request. At least it allows to get a request data from the URL or from the message body.

Parameters and Query

To retrieve information from a URL you can use properties Parameters and Query. Both return dynamic object (c. 5) with information from different parts of the URL. The Parameters contains values of named segments (c. 43) of the path while the Query contains values of the query string.



Next example shows how to extract values from a URL which looks like on previous image.

```
builder.Get["/{segment1}/{segment2}/{segment3}"] = async request =>
{
    string segment1 = request.Parameters.segment1; // segment1 == "path"
    string segment2 = request.Parameters.segment2; // segment2 == "to"
    string segment3 = request.Parameters.segment3; // segment3 == "resource"

    string key1 = request.Query.key1; // key1 == "value1"
    string key2 = request.Query.key2; // key2 == "value2"

    // ...
};
```

Form and Content

Such methods as POST, PUT and PATCH can contains the body. Depending on the Content-Type[66] header the message body can be parsed as an object. In this case the Form property returns dynamic object (c. 5). For instance the request body will be considered as an object if the Content-Type[67] equals application/json (see Media Types[68]).

[65] https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1?view=netcore-1.1
[66] https://www.w3.org/Protocols/rfc1341/4_Content-Type.html
[67] https://www.w3.org/Protocols/rfc1341/4_Content-Type.html
[68] http://www.iana.org/assignments/media-types/media-types.xhtml

```
// curl -X POST -H 'Content-Type: application/json' \
// -d '{"Title":"Title1","Content":"Content1"}' \
// http://www.example.com:80/articles

builder.Post["/articles"] = async request =>
{
    string title = request.Form.Title; // title == "Title1"
    string content = request.Form.Content; // content == "Content1"

    // ...
};
```

If you have a strongly typed model you can deserialize (c. 31) it from the request body. For example if the Content-Type[69] equals application/json the deserialization can be performed using IJsonObjectSerializer. The Content property allows you to get a Stream[70] object representing the incoming HTTP body. Also important to note the Content can be used for a custom handling of the request body.

```
public class ArticlesHttpService : IHttpService
{
    private readonly IJsonObjectSerializer _serializer;

    public ArticlesHttpService(IJsonObjectSerializer serializer)
    {
        _serializer = serializer;
    }

    public void Load(IHttpServiceBuilder builder)
    {
        // curl -X POST -H 'Content-Type: application/json' \
        // -d '{"Title":"Title1","Content":"Content1"}' \
        // http://www.example.com:80/articles

        builder.Post["/articles"] = async request =>
        {
            var article = _serializer.Deserialize<Article>(request.Content);

            // ...
        };
    }
}


public class Article
{
    public string Title { get; set; }
    public string Content { get; set; }
}
```

Note: Both the Form and the Content can not be used at the same time because they read the same request stream. So if you get the Form the request stream will be read and the Content will poit to the end.

Files

In more complex cases a request can contains one or few files which are available via the Files property. The Files returns an enumerable items of type IHttpRequestFile and each of them allows to get the file name and the file data stream.

[69] https://www.w3.org/Protocols/rfc1341/4_Content-Type.html
[70] https://docs.microsoft.com/en-us/dotnet/api/system.io.stream?view=netcore-1.1

```
builder.Post["/albums/{id}"] = async request =>
{
    foreach (IHttpRequestFile photo in request.Files)
    {
        // Do something
    }

    // ...
};
```

Response Building

The response is represented as the IHttpResponse interface which defines HTTP status code[71], HTTP headers[72] and HTTP body content. The basic implementation of the HttpResponse provides universal constructors to build any type of response. Also there are a few implementations to increase usability.

- TextHttpResponse represents a text response;
- JsonHttpResponse represents a JSON response;
- StreamHttpResponse represents a stream response of a given Content-Type[73];
- RedirectHttpResponse represents an HTTP redirect response;
- PrintViewHttpResponse represents a response of the Print View (c. 99).

Besides several prepared responses were added which are used very often:

- HttpResponse.Ok represents the 200 OK[74] response;
- HttpResponse.Unauthorized represents the 401 Unauthorized[75] response;
- HttpResponse.Forbidden represents the 403 Forbidden[76] response;
- HttpResponse.NotFound represents the 404 Not Found[77] response.

Result Converters

The response can be any model and the final result will be determined by the ResultConverter which defines conversion rules from the source model to the IHttpResponse instance. If a module (c. 41) does not set the ResultConverter then the default conversion rules are used. They are represented in the DefaultHttpResultConverter class:

- IHttpResponse will be returned as is;
- null will be interpreted as HttpResponse.Ok;
- int[78] will be interpreted as a HTTP status code[79];
- string[80] will be interpreted as TextHttpResponse;
- byte[][81], Stream[82] and Func<Stream>[83] will be interpreted as StreamHttpResponse;

[71] https://tools.ietf.org/html/rfc7231#section-6
[72] http://www.iana.org/assignments/message-headers/message-headers.xml
[73] https://www.w3.org/Protocols/rfc1341/4_Content-Type.html
[74] https://tools.ietf.org/html/rfc7231#section-6.3.1
[75] https://tools.ietf.org/html/rfc7235#section-3.1
[76] https://tools.ietf.org/html/rfc7231#section-6.5.3
[77] https://tools.ietf.org/html/rfc7231#section-6.5.4
[78] https://docs.microsoft.com/en-us/dotnet/api/system.int32?view=netcore-1.1
[79] https://tools.ietf.org/html/rfc7231#section-6
[80] https://docs.microsoft.com/en-us/dotnet/api/system.string?view=netcore-1.1
[81] https://docs.microsoft.com/en-us/dotnet/api/system.byte?view=netcore-1.1
[82] https://docs.microsoft.com/en-us/dotnet/api/system.io.stream?view=netcore-1.1
[83] https://docs.microsoft.com/en-us/dotnet/api/system.io.stream?view=netcore-1.1

- Exception[84] will be interpreted as 500 Internal Server Error[85] with the exception message;
- other objects will be interpreted as JsonHttpResponse.

Next converter wraps a result to the JSON object with a single property Result.

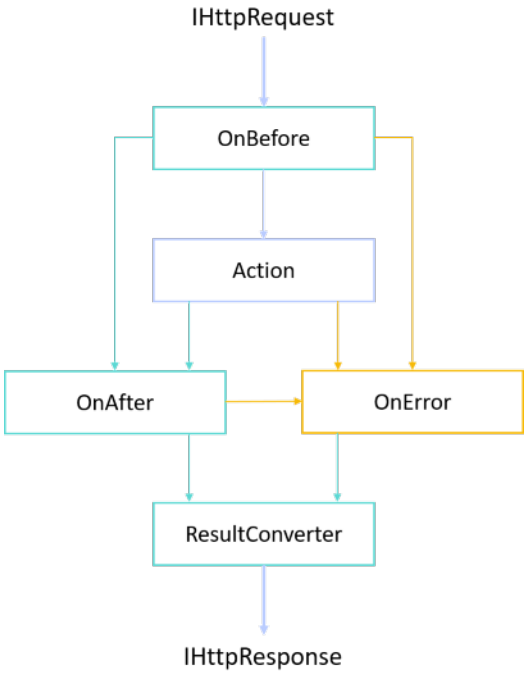
```
builder.ResultConverter = result =>
{
    return (result is IHttpResponse)
        ? (IHttpResponse)result
        : new JsonHttpResponse(new { Result = result });
};

builder.Get["/some"] = request =>
{
    return Task.FromResult<object>(123); // {"Result":123 }
};
```

Intercepting Requests

Besides defining handlers for specific routes (c. 43), a module (c. 41) can also intercept requests that match one of its routes, both before and after the route is invoked. It is important to understand that these interceptors will only be invoked if the incoming request matches one of the routes in the module.

Note: The interceptors are very useful when you want to perform tasks, per-request, on a module level for things like security, caching and rewriting requests and responses.

Below is shown the statechart with existing extension points. Each point is a stage of the request handling pipeline and you can intercept all of them to customize the handling. The ResultConverter (c. 46) is described previously so here OnBefore (c. 48), OnAfter (c. 48) and OnError (c. 48) are described.



[84] https://docs.microsoft.com/en-us/dotnet/api/system.io.stream?view=netcore-1.1
[85] https://tools.ietf.org/html/rfc7231#section-6.6.1

OnBefore interceptor

The OnBefore interceptor enables you to intercept the request before it is passed to the appropriate route handler - Action. This gives you a couple of possibilities such as modifying parts of the request or even prematurely aborting the request by returning a response that will be sent back to the caller.

```
builder.OnBefore = async (IHttpRequest request) =>
{
    // Do something asynchronously and return null or a result object
};
```

Since the interceptor will be invoked for all routes in the module, there is no need to define a pattern to match. The parameter that is passed into the interceptor is an instance of the current IHttpRequest. A return value of null means that no action is taken by the interceptor and that the request should proceed to be processed by the matching route. However, if the interceptor returns some result of its own, the route will never be processed by the route and the response will be sent back to the client.

OnAfter interceptor

The OnAfter gets an instance of the current IHttpRequest and a result from previous stage. While an OnBefore interceptor is called before the route handler an OnAfter interceptor is called when the route has already been handled and a response has been generated. So here you can modify or replace the result.

```
builder.OnAfter = async (IHttpRequest request, object result) =>
{
    // Do something asynchronously and return a modified result
};
```

OnError interceptor

The OnError interceptor enables you to execute code whenever an exception occurs in any of the module routes that are being invoked. It gives you access to the current IHttpRequest and the exception that took place. So here you can handle an exception and build an error result.

```
builder.OnError = async (IHttpRequest request, Exception exception) =>
{
    // Do something asynchronously and return an error result
};
```

Global interceptors

The application pipelines enable you to perform tasks before and after routes are executed, and in the event of an error in any of the routes in the application. They behave the same way as the module pipelines (see the statechart above (c. 47)) but they are executed for all invoked routes, not just for the ones that are the module of the route that is being invoked.

To define the application level HTTP handler you need to implement the IHttpGlobalHandler interface and register (c. 8) the implementation in IoC Container (c. 7).

Document Storage

Document storage is a computer program designed for storing, retrieving and managing document-oriented information, also known as semi-structured data. Document-oriented databases are one of the main categories of NoSQL[86] databases, and the popularity of the term document-oriented database has grown with the use of the term NoSQL itself.

InfinniPlatform provides abstraction layer to work with a document-oriented database and its implementation based on MongoDB[87].

Using Document Storage

Next instruction shows how to use the document storage API in conjunction with MongoDB[88].

Configuring Document Storage

1. Install MongoDB[89]

2. Install InfinniPlatform.DocumentStorage.MongoDB package:

```
dotnet add package InfinniPlatform.DocumentStorage.MongoDB \
  -s https://www.myget.org/F/infinniplatform
```

3. Call AddMongoDocumentStorage() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddMongoDocumentStorage();

        // ...

        return services.BuildProvider();
    }
```

[86] https://en.wikipedia.org/wiki/NoSQL
[87] https://www.mongodb.com/
[88] https://www.mongodb.com/
[89] https://www.mongodb.com/

```
    // ...
}
```

Next using of the document storage differs depending on context: typed (c. 50) or dynamic (c. 50).

Typed Context

When a document can be presented as a normal .NET class then better use the typed context. In this case you will have all the power static code analysis.

1. Create MyDocument.cs and define a descendant of the Document class:

```
using InfinniPlatform.DocumentStorage;

class MyDocument : Document
{
    public string Property1 { get; set; }

    // ...
}
```

2. Request the IDocumentStorageFactory instance in the constructor:

```
using InfinniPlatform.DocumentStorage;

class MyComponent
{
    private readonly IDocumentStorage<MyDocument> _storage;

    public MyComponent(IDocumentStorageFactory factory)
    {
        _storage = factory.GetStorage<MyDocument>();
    }

    // ...
}
```

3. To access to the documents use IDocumentStorage<TDocument>:

```
var document = new MyDocument { _id = 1, Property1 = "Hello!" };

// Create
_storage.InsertOne(document);

// Read
var document = _storage.Find(i => i._id.Equals(1)).First();

// Update
_storage.UpdateOne(u => u.Set(i => i.Property1, "Hello, World!"), i => i._id.Equals(1));

// Delete
_storage.DeleteOne(i => i._id.Equals(1));
```

Dynamic Context

When a document can not be presented as a normal .NET class because semi-structured data, you can use the dynamic context. In this case you you will have more flexibility but there is a chance to make mistake and find it only at runtime.

1. Use DynamicDocument to declare dynamic objects

2. Request the IDocumentStorageFactory instance in the constructor:

```
using InfinniPlatform.DocumentStorage;

class MyComponent
{
    private readonly IDocumentStorage _storage;

    public MyComponent(IDocumentStorageFactory factory)
    {
        _storage = factory.GetStorage("MyDocument");
    }

    // ...
}
```

3. To access to the documents use IDocumentStorage:

```
var document = new DynamicDocument { { "_id", 1 }, { "Property1", "Hello!" } };

// Create
_storage.InsertOne(document);

// Read
var document = _storage.Find(f => f.Eq("_id", 1)).First();

// Update
_storage.UpdateOne(u => u.Set("Property1", "Hello, World!"), f => f.Eq("_id", 1));

// Delete
_storage.DeleteOne(f => f.Eq("_id", 1));
```

Document Storage Interceptors

Document Storage Interceptors allow to intercept invocations to the IDocumentStorage<TDocument> or IDocumentStorage instances. This useful feature can help you implement some infrastructure mechanism, for instance, log an audit event when user accessing a document, validate documents and arguments of the accessing, inject or restrict data and so forth.

Intercept Typed Documents

When a document is presented as a normal .NET class and working with it goes through the IDocumentStorage<TDocument>, to define the interceptor for this document implement the IDocumentStorageInterceptor<TDocument> interface:

```
class MyDocumentStorageInterceptor : DocumentStorageInterceptor<MyDocument>
{
    public override void OnAfterInsertOne(DocumentInsertOneCommand<MyDocument> command,
                                           DocumentStorageWriteResult<object> result,
                                           Exception exception)
    {
        // Handling the invocations IDocumentStorage<MyDocument>.InsertOne()...
    }

    // ...
}
```

After that register the interceptor in IoC-container (c. 8):

```
builder.RegisterType<MyDocumentStorageInterceptor>()
       .As<IDocumentStorageInterceptor>()
       .SingleInstance();
```

Intercept Dynamic Documents

When a document is presented as the DynamicDocument class and working with it goes through the IDocumentStorage, to define the interceptor for this document implement the IDocumentStorageInterceptor interface:

```
class MyDocumentStorageInterceptor : DocumentStorageInterceptor
{
   public override void OnAfterInsertOne(DocumentInsertOneCommand command,
                                          DocumentStorageWriteResult<object> result,
                                          Exception exception)
   {
      // Handling the invocations IDocumentStorage<MyDocument>.InsertOne()...
   }

   // ...
}
```

After that register the interceptor in IoC-container (c. 8):

```
builder.RegisterType<MyDocumentStorageInterceptor>()
       .As<IDocumentStorageInterceptor>()
       .SingleInstance();
```

Document Storage Management

Each storage can have additional settings at least each storage has unique name. Besides it there is support of the indexes. Indexes support the efficient execution of queries. Without indexes the storage must scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, the storage can use the index to limit the number of documents it must inspect.

Indexes are special data structures that store a small portion of the storage's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations.

Default Indexes

The IDocumentStorage<TDocument> interface implementation works with descendants of the Document class which defines two mandatory fields: _id and _header, respectively, the unique identifier and the system information of the document. By default the _id is Guid[90] but it is possible to use any other type. The _header is described using DocumentHeader class.

The DocumentHeader class contains system information which is used by an implementation of the storage to organize and control access to the documents. For example, the _tenant field helps to organize multitenancy[91] and that field is used upon each query execution as an additional filter. Thus each document contains at least two indexes:

- _id - for primary key

[90] https://docs.microsoft.com/en-us/dotnet/api/system.guid?view=netcore-1.1
[91] https://en.wikipedia.org/wiki/Multitenancy

- _header._tenant and _header._deleted - for multitenancy[92]

Note: The IDocumentStorage interface implementation works the same but instead of the Document it uses DynamicDocument.

Additional Indexes

To define additional indexes you must create an instance of the DocumentMetadata class and set the Indexes property. Next example defines index by UserName field.

```
var userNameIndex = new DocumentIndex
                {
                    Key = new Dictionary<string, DocumentIndexKeyType>
                        {
                            { "UserName", DocumentIndexKeyType.Asc }
                        }
                };

var userStoreMetadata = new DocumentMetadata
                    {
                        Type = "UserStore",
                        Indexes = new[] { userNameIndex }
                    };
```

After that you must get IDocumentStorageManager and invoke the CreateStorageAsync() method:

```
IDocumentStorageManager _ storageManager;

// ...

storageManager.CreateStorageAsync(userStoreMetadata);
```

Storage Provider

The IDocumentStorage<TDocument> and IDocumentStorage are used to access documents of the storage but besides data access they do additional logic. For example, providing interception (c. 51) mechanism and organizing multitenancy[93]. And this logic can be a barrier to realize some system procedures such as data migration. To direct access to the storage without additional logic (interception, multitenancy and etc.), use IDocumentStorageProvider<TDocument> or IDocumentStorageProvider. To get them, use IDocumentStorageProviderFactory.

Document Attributes

There are several helpful attributes that can make work with documents more convenient.

DocumentTypeAttribute

The DocumentTypeAttribute attribute defines the storage name of the specified document type. By default without its attribute the storage name equals an appropriate type name. It is not always reasonable especially you use existing database or other naming convention. Also this attribute eliminates possibility of accidental renaming the storage during renaming the document class name.

[92] https://en.wikipedia.org/wiki/Multitenancy
[93] https://en.wikipedia.org/wiki/Multitenancy

```
[DocumentType("orders")]
class Order : Document
{
    // ...
}
```

DocumentIgnoreAttribute

The DocumentIgnoreAttribute attribute forces the storage to ignore the specified property of the document class. Properties are marked with this attribute will not be stored in a database. It is most applicable for calculated properties, that is properties which values can be calculated based on values of other properties; or when data can not or should not be serialized to be stored in a database.

```
class Order : Document
{
    public double Count { get; set; }
    public double Price { get; set; }

    [DocumentIgnore]
    public double Total => Count * Price;

    // ...
}
```

DocumentPropertyNameAttribute

The DocumentPropertyNameAttribute attribute forces the storage to use other name for the specified property of the document class. By default without its attribute field names in a database equal appropriate property names. It is not always reasonable especially you use existing database or other naming convention. Also this attribute eliminates possibility of accidental renaming the field during renaming the document property name.

```
class Order : Document
{
    [DocumentPropertyName("count")]
    public double Count { get; set; }

    [DocumentPropertyName("price")]
    public double Price { get; set; }

    // ...
}
```

Specifications

Direct access to the IDocumentStorage<TDocument> or IDocumentStorage provokes uncontrolled increasing number of query types. If you still working in this style after for a while you will see that the same queries are implemented in different ways and one query works fine but its analogue is very slow. Then you will lose control and not able to enumerate which queries go to the storage. All of these provokes bugs and makes support difficult. In fact an application as a rule has only few typical queries and it will be fine having them in one place.

Repository[94] and Specifications[95] patterns come to the rescue. InfinniPlatform implements these patterns and provides two base classes: Specification<TDocument> (for typed context (c. 50)) and Specification

[94] https://martinfowler.com/eaaCatalog/repository.html
[95] https://www.martinfowler.com/apsupp/spec.pdf

(for dynamic context (c. 50)).

Specifications for Typed Documents

Suppose we need to organize a storage of some articles and we know each article has date, author, title, text and can be in two states: draft and published.

```
class Article : Document
{
    public DateTime Date { get; set; }
    public string Author { get; set; }
    public string Title { get; set; }
    public string Text { get; set; }
    public ArticleState State { get; set; }

    // ...
}

enum ArticleState
{
    Draft = 0,
    Published = 1
}
```

And in the most of the cases we need to fetch published articles for the specified period:

```
class GetPublishedArticles : Specification<Article>
{
    private readonly DateTime _start;
    private readonly DateTime _end;

    public GetPublishedArticles(DateTime start, DateTime end)
    {
        _start = start;
        _end = end;
    }

    public override Expression<Func<Article, bool>> Filter =>
        a => a.Date >= _start && a.Date <= _end
            && a.State == ArticleState.Published;
}
```

After that it will be right to create appropriate repository of the articles:

```
interface IArticleRepository
{
    IEnumerable<Article> GetArticles(ISpecification<Article> specification);

    // ...
}

class ArticleRepository : IArticleRepository
{
    private readonly IDocumentStorage<Article> _storage;

    public ArticleRepository(IDocumentStorageFactory factory)
    {
        _storage = factory.GetStorage<Article>();
    }

    public IEnumerable<Article> GetArticles(ISpecification<Article> specification)
    {
```

```
        return _storage.Find(specification.Filter).ToList();
    }

    // ...
}
```

And use it to fetch articles:

```
IArticleRepository repository;
DateTime start, end;

// ...

ISpecification<Article> specification = new GetPublishedArticles(start, end);
IEnumerable<Article> publishedArticles = repository.GetArticles(specification);
```

Specifications for Dynamic Documents

Let's consider the same example as above but in dynamic context. Declare specification:

```
class GetPublishedArticles : Specification
{
    private readonly DateTime _start;
    private readonly DateTime _end;

    public GetPublishedArticles(DateTime start, DateTime end)
    {
        _start = start;
        _end = end;
    }

    public override Func<IDocumentFilterBuilder, object> Filter =>
        a => a.And(a.Gte("Date", _start), a.Lte("Date", _end),
                a.Eq("State", ArticleState.Published));
}
```

Declare the repository of the articles:

```
interface IArticleRepository
{
    IEnumerable<DynamicDocument> GetArticles(ISpecification specification);

    // ...
}

class ArticleRepository : IArticleRepository
{
    private readonly IDocumentStorage _storage;

    public ArticleRepository(IDocumentStorageFactory factory)
    {
        _storage = factory.GetStorage("Article");
    }

    public IEnumerable<DynamicDocument> GetArticles(ISpecification specification)
    {
        return _storage.Find(specification.Filter).ToList();
    }

    // ...
}
```

And use it to fetch articles:

```
IArticleRepository repository;
DateTime start, end;

// ...

ISpecification specification = new GetPublishedArticles(start, end);
IEnumerable<DynamicDocument> publishedArticles = repository.GetArticles(specification);
```

Composing Specifications

The specification classes Specification<TDocument> and Specification override !, & and | operators and implement, respectively, negation, conjunction and disjunction. Thus you can compose existing specifications instead of creating new.

```
// Not
var notSpecification = !specification;

// And
var andSpecification = specification1 & specification2 & specification3;

// Or
var orSpecification = specification1 | specification2 | specification3;
```

Transactions

The document storage does not assume supporting transactions in terms of ACID[96] but rely that a write operation is atomic on the level of a single document, even if the operation modifies multiple embedded documents within a single document. When a single write operation modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic and other operations may interleave. It is common behavior most of NoSQL[97] databases.

Two Phase Commit

Lack of transactions is known problem and you can solve it using the Two Phase Commit[98], for example, in a way which is offered by MondoDB[99]. But there is no silver bullet and if you really need ACID[100] transactions you should use appropriate database.

Unit of Work

Yet another solution of transactions absence can be the Unit of Work[101] pattern which is presented as the IUnitOfWork interface. You can accumulate changes in the memory of the web server and then commit them. Until the commit, all changes of the documents remain in the memory and nobody see them. Thus if during the changes an exception appears, the documents will not be changed. To some degree it works like transactions but if there is some functional relation between the changes, the unit of work can be unsuitable solution.

To create instance of the IUnitOfWork use the IUnitOfWorkFactory:

[96] https://en.wikipedia.org/wiki/ACID
[97] https://en.wikipedia.org/wiki/NoSQL
[98] https://en.wikipedia.org/wiki/Two-phase_commit_protocol
[99] https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/
[100] https://en.wikipedia.org/wiki/ACID
[101] http://martinfowler.com/eaaCatalog/unitOfWork.html

```
IUnitOfWorkFactory factory;

// ...

IUnitOfWork unitOfWork = factory.Create();
```

Note: The IUnitOfWork can be nested during the HTTP request handling.

The IUnitOfWork implements the IDisposable[102] interface so it is better to use using operator:

```
IUnitOfWorkFactory factory;

// ...

using (IUnitOfWork unitOfWork = factory.Create())
{
    // Inserts, updates, deletes...

    unitOfWork.Commit();
}
```

After all changes are applied, invoke Commit() or CommitAsync().

Now let's consider full example of using the IUnitOfWork:

```
IUnitOfWorkFactory factory;

// ...

using (IUnitOfWork unitOfWork = factory.Create())
{
    unitOfWork.InsertOne(...);
    unitOfWork.InsertMany(...);

    unitOfWork.UpdateOne(...);
    unitOfWork.UpdateMany(...);

    unitOfWork.DeleteOne(...);
    unitOfWork.DeleteMany(...);

    unitOfWork.Commit();
}
```

Document HTTP Service

There is possibility to expose the storage via HTTP as is. Be careful, it provide powerful mechanism for quick start but to build clear and understandable RESTful API better create own HTTP services (с. 41).

Configuring Document HTTP Service

1. Install InfinniPlatform.DocumentStorage.HttpService package:

```
dotnet add package InfinniPlatform.DocumentStorage.HttpService \
    -s https://www.myget.org/F/infinniplatform/
```

2. Call AddDocumentStorageHttpService() in ConfigureServices():

[102] https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netframework-4.7

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddDocumentStorageHttpService();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

3. Register in IoC-container (c. 8) the document HTTP service:

```
builder.RegisterDocumentHttpService<MyDocument>();
```

4. Run application and browse to http://localhost:5000/documents/MyDocument/

Interception of Document HTTP Service

The document HTTP service provides default behavior which may be a little different than what you want. In this case you can use interceptor of the document HTTP service. Implement IDocumentStorageInterceptor<TDocument> (typed context) or IDocumentStorageInterceptor (dynamic context) and register the implementation in IoC-container (c. 8).

Document HTTP Service API

GET /documents/(string: documentType)/
string: id Returns the document of the specified type and with given identifier.

Parameters

- documentType (string) – The document type name.
- id (string) – The document unique identifier.

Response Headers

- Content-Type[103] – application/json

Status Codes

- 200 OK[104] – OK
- 400 Bad Request[105] – Validation Error
- 500 Internal Server Error[106] – Internal Server Error

GET /documents/(string: documentType)/
Returns documents of the specified type.

Parameters

[103] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[104] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[105] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[106] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1

- documentType (string) – The document type name.

Query Parameters

- search (string) – Optional. The text for full text search.
- filter (string) – Optional. The filter query (c. 61).
- select (string) – Optional. The select query (c. 69).
- order (string) – Optional. The order query (c. 69).
- count (boolean) – Optional. By default - false. The flag whether to return the number of documents.
- skip (int) – Optional. By default - 0. The number of documents to skip before returning the remaining elements.
- take (int) – Optional. By default - 10, maximum - 1000. The number of documents to return.

Response Headers

- Content-Type[107] – application/json

Status Codes

- 200 OK[108] – OK
- 400 Bad Request[109] – Validation Error
- 500 Internal Server Error[110] – Internal Server Error

POST /documents/(string: documentType)/
Creates or updates specified document.

Parameters

- documentType (string) – The document type name.

Form Parameters

- body – The document and optionally the document attachments (files).

Request Headers

- Content-Type[111] – application/json
- Content-Type[112] – multipart/form-data
- Content-Type[113] – application/x-www-form-urlencoded

Response Headers

- Content-Type[114] – application/json

Status Codes

- 200 OK[115] – OK
- 400 Bad Request[116] – Validation Error
- 500 Internal Server Error[117] – Internal Server Error

[107] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[108] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[109] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[110] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1
[111] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[112] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[113] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[114] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[115] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[116] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[117] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1

DELETE /documents/(string: documentType)/
string: id Deletes the document of the specified type and with given identifier.

Parameters

- documentType (string) – The document type name.
- id (string) – The document unique identifier.

Response Headers

- Content-Type[118] – application/json

Status Codes

- 200 OK[119] – OK
- 400 Bad Request[120] – Validation Error
- 500 Internal Server Error[121] – Internal Server Error

DELETE /documents/(string: documentType)/
Deletes documents of the specified type.

Parameters

- documentType (string) – The document type name.

Query Parameters

- filter (string) – Optional. The filter query (c. 61).

Response Headers

- Content-Type[122] – application/json

Status Codes

- 200 OK[123] – OK
- 400 Bad Request[124] – Validation Error
- 500 Internal Server Error[125] – Internal Server Error

Filter Query

The filter query is a string with contains a filter expression:

```
func(args)
```

where func - the filter function name, args - the function arguments.

There are a lot of function, most of them accepts a document property name as the first parameter and an appropriate value as the second parameter which is used to compare with the property. Some functions can accept other functions as arguments such as composing function - and and or, other functions can have no arguments, have one or any amount. Below the filter query functions are presented.

Logical Query Functions

not(filter)
The logical negation of the specified expression.

Example:

[118] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[119] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[120] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[121] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1
[122] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[123] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[124] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[125] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1

```
not(eq('status', 'published'))
```

Arguments

- filter – The filter query.

and(filters)
The logical conjunction of the specified expression.

Example:

```
and(eq('status', 'published'), eq('author', 'John Smith'))
```

Arguments

- filters – The list of filter queries separated by comma.

or(filters)
The logical disjunction of the specified expression.

```
and(eq('status', 'published'), eq('status', 'signed'))
```

Arguments

- filters – The list of filter queries separated by comma.

Element Query Functions

exists(field[, exists = true])
When exists is true, matches the documents that contain the field, including documents where the field value is null; if exists is false, the query returns only the documents that do not contain the field.

Arguments

- field (string) – The document field name.
- exists (boolean) – The flag of existings.

type(field, valueType)
Selects the documents where the value of the field is an instance of the specified type. Querying by data type is useful when dealing with highly unstructured data where data types are not predictable.

Available Types:

•Boolean

•Int32

•Int64

•Double

•String

•DateTime

•Timestamp

•Binary

•Object

•Array

Example:

```
type('zipCode', 'String')
```

Arguments

- field (string) – The document field name.
- valueType (string) – The document field type.

Comparison Query Functions

in(field, values)
Selects the documents where the value of a field equals any value in the specified array.

Example:

```
in('tags', '.net', 'asp.net', 'c#')
```

Arguments

- field (string) – The document field name.
- values – The field values.

notIn(field, values)
Selects the documents where the field value is not in the specified array or the field does not exist.

Example:

```
notIn('tags', '.net', 'asp.net', 'c#')
```

Arguments

- field (string) – The document field name.
- values – The field values.

eq(field, value)
Specifies equality condition, matches documents where the value of a field equals the specified value.

Example:

```
eq('status', 'published')
```

Arguments

- field (string) – The document field name.
- value – The field value.

notEq(field, value)
Selects the documents where the value of the field is not equal to the specified value. This includes documents that do not contain the field.

Example:

```
notEq('status', 'published')
```

Arguments

- field (string) – The document field name.
- value – The field value.

gt(field, value)
Selects those documents where the value of the field is greater than the specified value.

Example:

```
gt('price', 9.99)
```

Arguments

- field (string) – The document field name.
- value – The field value.

gte(field, value)
Selects the documents where the value of the field is greater than or equal to a specified value.

Example:

```
gte('price', 9.99)
```

Arguments

- field (string) – The document field name.
- value – The field value.

lt(field, value)
Selects the documents where the value of the field is less than the specified value.

Example:

```
lt('price', 9.99)
```

Arguments

- field (string) – The document field name.
- value – The field value.

lte(field, value)
Selects the documents where the value of the field is less than or equal to the specified value.

Example:

```
lte('price', 9.99)
```

Arguments

- field (string) – The document field name.
- value – The field value.

regex(field, pattern)
Selects documents where the value of the field matches a specified regular expression.

Example:

```
regex('phone', '^+123')
```

Arguments

- field (string) – The document field name.
- pattern (string) – The regular expression.

startsWith(field, value[, ignoreCase = true])
Selects documents where the value of the field starts with a specified substring.

Example:

```
startsWith('phone', '+123')
```

Arguments

- field (string) – The document field name.
- value (string) – The substring to matching.
- ignoreCase (boolean) – The flag of ignoring case.

endsWith(field, value[, ignoreCase = true])
Selects documents where the value of the field ends with a specified substring.

Example:

```
endsWith('phone', '789')
```

Arguments

- field (string) – The document field name.
- value (string) – The substring to matching.
- ignoreCase (boolean) – The flag of ignoring case.

contains(field, value[, ignoreCase = true])
Selects documents where the value of the field contains a specified substring.

Example:

```
contains('phone', '456')
```

Arguments

- field (string) – The document field name.
- value (string) – The substring to matching.
- ignoreCase (boolean) – The flag of ignoring case.

Array Query Functions

match(arrayField, filter)
Selects documents where the value of the field is an array which contains elements that satisfy the specified filter.

Example:

```
match('addresses', eq('street', 'Broadway'))
```

Arguments

- arrayField (string) – The document field which contains an array.
- filter – The filter query.

all(arrayField, elements)
Selects the documents where the value of a field is an array that contains all the specified elements.

Example:

```
all('tags', '.net', 'asp.net', 'c#')
```

Arguments

- arrayField (string) – The document field which contains an array.
- elements – The list of elements to matching.

anyIn(arrayField, elements)
Selects the documents where the value of a field is an array that contains at least one of the specified elements.

Example:

```
anyIn('tags', '.net', 'asp.net', 'c#')
```

Arguments

- arrayField (string) – The document field which contains an array.
- elements – The list of elements to matching.

anyNotIn(arrayField, elements)
Selects the documents where the value of a field is an array that does not contains the specified elements.

Example:

```
anyNotIn('tags', '.net', 'asp.net', 'c#')
```

Arguments

- arrayField (string) – The document field which contains an array.
- elements – The list of elements to matching.

anyEq(arrayField, element)
Selects the documents where the value of a field is an array that contains at least one element that equals the specified.

Example:

```
anyEq('tags', '.net')
```

Arguments

- arrayField (string) – The document field which contains an array.
- element – The element to matching.

anyNotEq(arrayField, element)
Selects the documents where the value of a field is an array that contains at least one element that does not equal the specified.

Example:

```
anyNotEq('tags', '.net')
```

Arguments

- arrayField (string) – The document field which contains an array.
- element – The element to matching.

anyGt(arrayField, element)
Selects the documents where the value of a field is an array that contains at least one element that is greater than the specified.

Example:

```
anyGt('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- element – The element to matching.

anyGte(arrayField, element)
Selects the documents where the value of a field is an array that contains at least one element that is greater than or equal to the specified.

Example:

```
anyGte('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- element – The element to matching.

anyLt(arrayField, element)
Selects the documents where the value of a field is an array that contains at least one element that is less than the specified.

Example:

```
anyLt('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- element – The element to matching.

anyLte(arrayField, element)
Selects the documents where the value of a field is an array that contains at least one element that is less than or equal to the specified.

Example:

```
anyLte('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- element – The element to matching.

sizeEq(arrayField, size)
Selects the documents where the value of a field is an array which size equals the specified.

Example:

```
sizeEq('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- size (int) – The element to matching.

sizeGt(arrayField, size)
Selects the documents where the value of a field is an array which size is greater than the specified.

Example:

```
sizeGt('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- size (int) – The element to matching.

sizeGte(arrayField, size)
Selects the documents where the value of a field is an array which size is greater than or equal to the specified.

Example:

```
sizeGte('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- size (int) – The element to matching.

sizeLt(arrayField, size)
Selects the documents where the value of a field is an array which size is less than the specified.

Example:

```
sizeLt('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- size (int) – The element to matching.

sizeLte(arrayField, size)
Selects the documents where the value of a field is an array which size is less than or equal to the specified.

Example:

```
sizeLte('scores', 42)
```

Arguments

- arrayField (string) – The document field which contains an array.
- size (int) – The element to matching.

Constant Query Functions

date(value)
Specifies a date and time constant using ISO 8601[126] format.

Example:

[126] https://en.wikipedia.org/wiki/ISO_8601

```
date('2017-06-21')
```

Arguments

- value (string) – The date and time in ISO 8601[127] format.

Select Query

The select query allows to request the only specified fields of the document or vice versa exclude specified fields of the document. The select query is a string with contains a list of expressions separated by comma:

```
func(args), func(args), ...
```

where func - the select function name, args - the function arguments.

Below the select query functions are presented.

include(field)
: Specifies that the specified field should be included in the response.

  Example:

```
include('addresses')
```

  Arguments

  - field (string) – The document field name.

exclude(field)
: Specifies that the specified field should be excluded from the response.

  Example:

```
exclude('addresses')
```

  Arguments

  - field (string) – The document field name.

Order Query

The order query specifies the order in which the query returns matching documents. The order query is a string with contains a list of expressions separated by comma:

```
func(args), func(args), ...
```

where func - the order function name, args - the function arguments.

Below the order query functions are presented.

asc(field)
: Specifies an ascending sort for the specified field.

  Example:

```
asc('creationDate')
```

  Arguments

  - field (string) – The document field name.

[127] https://en.wikipedia.org/wiki/ISO_8601

desc(field)
Specifies an descending sort for the specified field.

Example:

```
desc('creationDate')
```

Arguments

- field (string) – The document field name.

BLOB Storage

BLOB storage service can be used to store and retrieve Binary Large Objects (BLOBs), or what are more commonly known as files. In this part we will cover how to get files into and out of the storage, how you can add metadata to your files and more.

There are many reasons why you should consider using BLOB storage. Perhaps you want to share files with clients, or off-load some of the static content from your web servers to reduce the load on them.

Using BLOB Storage

Next instruction shows how to use the BLOB storage API.

1. Install InfinniPlatform.BlobStorage.FileSystem package:

```
dotnet add package InfinniPlatform.BlobStorage.FileSystem \
  -s https://www.myget.org/F/infinniplatform/
```

2. Call AddFileSystemBlobStorage() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddFileSystemBlobStorage();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

3. Request the IBlobStorage instance in the constructor:

```
class MyComponent
{
    private readonly IBlobStorage _storage;

    public MyComponent(IBlobStorage storage)
    {
```

```
        _storage = storage;
    }

    // ...
}
```

4. Use IBlobStorage to access BLOBs:

```
// Create
Stream blobStream;
BlobInfo blobInfo = _storage.CreateBlob("document.pdf", "application/pdf", blobStream);

// Read
string blobId = blobInfo.Id;
BlobData blobData = _storage.GetBlobData(blobId);

// Update
Stream newBlobStream;
BlobInfo newBlobInfo = UpdateBlob(blobId, "new_document.pdf", "application/pdf", newBlobStream);

// Delete
_storage.DeleteBlob(blobId);
```

BLOB HTTP Service

There is possibility to expose the storage via HTTP as is. Be careful, it provide powerful mechanism for quick start but to build clear and understandable RESTful API better create own HTTP services (с. 41).

1. Install InfinniPlatform.BlobStorage.HttpService package:

```
dotnet add package InfinniPlatform.BlobStorage.HttpService \
    -s https://www.myget.org/F/infinniplatform/
```

2. Call AddBlobStorageHttpService() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddBlobStorageHttpService();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

3. Run application and browse to http://localhost:5000/blob/<id> (where id is BLOB's identifier)

Data Caching

Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently. Caching makes a copy of data that can be returned much faster than from the original source. You should write and test your app to never depend on cached data.

Using IInMemoryCache

The simplest cache is based on the IInMemoryCache, which represents a cache stored in the memory of the web server. Apps which run on a server farm of multiple servers should ensure that sessions are sticky when using the in-memory cache. Sticky sessions ensure that subsequent requests from a client all go to the same server.

Note: Some HTTP servers allow sticky sessions. For example, nginx[128] supports session persistence[129], but there is no guarantee that the same client will be always directed to the same server. Thus we recommend using IInMemoryCache if and only if you have a single app server otherwise you should use some kind distributed cache, for instance, ISharedCache or ITwoLayerCache. Nonetheless, the in-memory cache can store any object; the distributed cache is limited by a database format.

To work with IInMemoryCache you need to make next steps.

1. Install InfinniPlatform.Cache.Memory package:

```
dotnet add package InfinniPlatform.Cache.Memory -s https://www.myget.org/F/infinniplatform/
```

2. Call AddInMemoryCache() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddInMemoryCache();

        // ...

        return services.BuildProvider();
```

[128] http://nginx.org/
[129] http://nginx.org/en/docs/http/load_balancing.html#nginx_load_balancing_with_ip_hash

```
    }

    // ...
}
```

3. Request the IInMemoryCache instance in the constructor:

```
class MyComponent
{
    private readonly IInMemoryCache _cache;

    public MyComponent(IInMemoryCache cache)
    {
        _cache = cache;
    }

    // ...
}
```

Using ISharedCache

Apps which run on a server farm of multiple servers should ensure that sessions are sticky when using the in-memory cache. Sticky sessions ensure that subsequent requests from a client all go to the same server. Non-sticky sessions in a web farm require a distributed cache to avoid cache consistency problems. For some apps, a distributed cache can support higher scale out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.

InfinniPlatform has an abstraction for distributed caching - ISharedCache, which assumes keeping data in a database. There is one implementation based on Redis[130] and to use this implementation you need to make next steps.

1. Install Redis[131]

2. Install InfinniPlatform.Cache.Redis package:

```
dotnet add package InfinniPlatform.Cache.Redis -s https://www.myget.org/F/infinniplatform/
```

3. Call AddRedisSharedCache() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddRedisSharedCache();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

[130] https://redis.io/
[131] https://redis.io/

4. Request the ISharedCache instance in the constructor:

```
class MyComponent
{
    private readonly ISharedCache _cache;

    public MyComponent(ISharedCache cache)
    {
        _cache = cache;
    }

    // ...
}
```

Using ITwoLayerCache

If cached data that changes infrequently that the best solution can be ITwoLayerCache, which assumes keeping data in a database but duplicates it in the memory of the web server. Thus you reduce interactions with a database and consequently improve performance. There is one implementation of this abstraction - TwoLayerCache. The TwoLayerCache depends on IInMemoryCache and ISharedCache implicitly. The former is the first layer of the caching, the latter is the second. To avoid cache consistency problems you also need to provide ITwoLayerCacheStateObserver implementation.

Next instruction shows how to use ITwoLayerCache based on Redis[132] as the second layer of caching and RabbitMQ[133] for the cache synchronization.

1. Install Redis[134]

2. Install RabbitMQ[135]

3. Install next packages:

```
dotnet add package InfinniPlatform.Cache.Memory -s https://www.myget.org/F/infinniplatform/
dotnet add package InfinniPlatform.Cache.Redis -s https://www.myget.org/F/infinniplatform/
dotnet add package InfinniPlatform.Cache.TwoLayer -s https://www.myget.org/F/infinniplatform/
dotnet add package InfinniPlatform.MessageQueue.RabbitMQ -s https://www.myget.org/F/infinniplatform/
```

4. Update ConfigureServices() as below:

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddInMemoryCache();
        services.AddRedisSharedCache();
        services.AddTwoLayerCache();
        services.AddRabbitMqMessageQueue();

        // ...

        return services.BuildProvider();
    }
```

[132] https://redis.io/
[133] https://www.rabbitmq.com/
[134] https://redis.io/
[135] https://www.rabbitmq.com/

```
    // ...
}
```

5. Request the ITwoLayerCache instance in the constructor:

```
class MyComponent
{
    private readonly ITwoLayerCache _cache;

    public MyComponent(ITwoLayerCache cache)
    {
        _cache = cache;
    }

    // ...
}
```

Message Queue

Message queues are the linking part between various processes and provide reliable and scalable interface to interact with other connected systems and devices. There are several advantages using queues and further is listed the most important.

Firstly the queues allow to implement loosely coupled[136] systems and thus satisfy one of GRASP[137] pattern. Components in a loosely coupled system can be replaced with alternative implementations that provide the same services. Also such components are less constrained to the same platform, language, operating system, or build environment.

Secondly the queues provide a way of horizontal scalability[138]. An important advantage of horizontal scalability is that it can provide administrators with the ability to increase capacity on the fly. Another advantage is that in theory, horizontal scalability is only limited by how many entities can be connected successfully. Thereby you can implement fault tolerant and do not have a single point of failure.

Thirdly the queues ensure asynchronous processing. Asynchronous processing enables various processes to run at the same time. In general processes might be processed faster and the uniform load distribution is ensured.

Also there are disadvantages of the queues. If systems are decoupled in time, it is difficult to also provide transactional integrity; additional coordination protocols are required. Furthermore, if the order is important for some processes the queues are not appropriate tool.

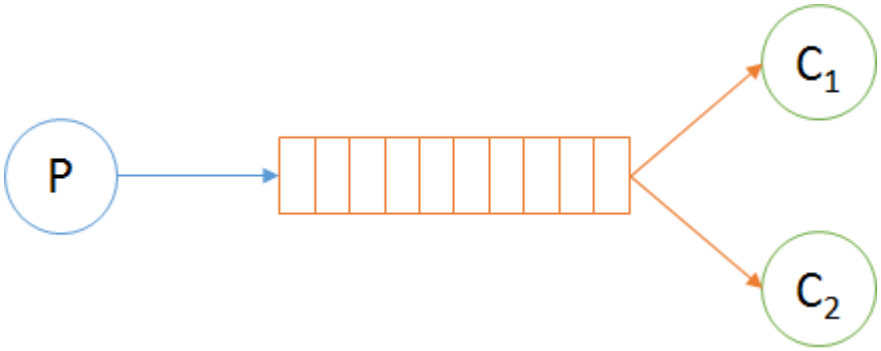
Message Queue Types

In general may select two types of the queues: task queues (c. 77) and broadcast queues (c. 78).

Task Queue

Messages from the task queue are distributed among all subscribers but each message is received and handled by only one subscriber. This queue may be used to organize parallel processing of tasks or data.

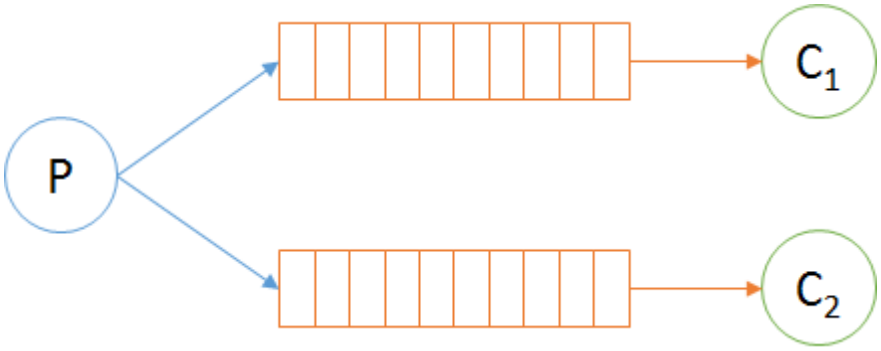
[136] https://en.wikipedia.org/wiki/Loose_coupling
[137] https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)#Low_coupling
[138] https://en.wikipedia.org/wiki/Scalability



In InfinniPlatform the ITaskProducer interface represents producer and the ITaskConsumer interface represents consumer for the task queues.

Broadcast Queue

Messages from the broadcast queue are received and handled by each subscriber. This queue may be used for mixed processing of the same message.



In InfinniPlatform the IBroadcastProducer interface represents producer and the IBroadcastConsumer interface represents consumer for the task queues.

Using Message Queue

InfinniPlatform has an abstraction for work with message queue. Currently there is one implementation based on RabbitMQ[139]. To use this implementation you need to make next steps.

1. Install Erlang[140]

2. Install RabbitMQ[141]

3. Install InfinniPlatform.MessageQueue.RabbitMQ package:

```
dotnet add package InfinniPlatform.MessageQueue.RabbitMQ -s https://www.myget.org/F/infinniplatform/
```

4. Call AddRabbitMqMessageQueue() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;
```

[139] https://www.rabbitmq.com/
[140] http://www.erlang.org/
[141] https://www.rabbitmq.com/

```
public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddRabbitMqMessageQueue();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

Using Task Queue

To send messages in a task queue (c. 77) use the ITaskProducer interface:

```
public class MyMessage
{
    // ...
}

public class MyComponent
{
    private readonly ITaskProducer _producer;

    public MyComponent(ITaskProducer producer)
    {
        _producer = producer;
    }

    public async Task DoSomething(MyMessage message)
    {
        MyMessage message;

        // ...

        await _producer.PublishAsync(message);

        // ...
    }
}
```

To receive messages from a task queue (c. 77) implement the ITaskConsumer interface:

```
public class MyConsumer : TaskConsumerBase<MyMessage>
{
    protected override async Task Consume(Message<MyMessage> message)
    {
        // Message handling
    }
}
```

Consumers of the task queue (c. 77) must be registered in IoC-container (c. 8):

```
builder.RegisterType<MyConsumer>().As<ITaskConsumer>().SingleInstance();
```

Using Broadcast Queue

To send messages in a broadcast queues (c. 78) use the IBroadcastProducer interface:

```
public class MyMessage
{
    // ...
}

public class MyComponent
{
    private readonly IBroadcastProducer _producer;

    public MyComponent(IBroadcastProducer producer)
    {
        _producer = producer;
    }

    public async Task DoSomething(MyMessage message)
    {
        MyMessage message;

        // ...

        await _producer.PublishAsync(message);

        // ...
    }
}
```

To receive messages from a broadcast queues (c. 78) implement the IBroadcastConsumer interface:

```
public class MyConsumer : BroadcastConsumerBase<MyMessage>
{
    protected override async Task Consume(Message<MyMessage> message)
    {
        // Message handling
    }
}
```

Consumers of the broadcast queues (c. 78) must be registered in IoC-container (c. 8):

```
builder.RegisterType<MyConsumer>().As<IBroadcastConsumer>().SingleInstance();
```

Security

InfinniPlatform is fully integrated with ASP.NET Core Security API and provides several helpful services to automate security aspects implementation. This chapter gives a brief how to use these services.

Internal Authentication

Internal Authentication provides the authentication mechanism based on its own user storage. Thus you must install the Document Storage (с. 49) or define own storage for the users.

1. Install InfinniPlatform.Auth package:

```
dotnet add package InfinniPlatform.Auth -s https://www.myget.org/F/infinniplatform/
```

2. Call AddAuthInternal() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddAuthInternal();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

Auth HTTP Service

There is predefined HTTP service for authentication.

1. Install InfinniPlatform.Auth.HttpService package:

```
dotnet add package InfinniPlatform.Auth.HttpService -s https://www.myget.org/F/infinniplatform/
```

2. Call AddAuthHttpService() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddAuthHttpService();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

After that next authentication API will be available via HTTP.

POST /auth/SignIn/

Authenticates the user based on the specified user key and password and starts user session. User's id, username, email or phone number can be used as user key. This method will try to find user using key one-by-one, so can be less effective than methods for specific user key (see below).

Example:

```
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{"UserKey":"user1","password":"qwerty"}' \
    http://localhost:5000/auth/SignIn
```

Request Headers

- Content-Type[142] – application/json

Response Headers

- Content-Type[143] – application/json
- Set-Cookie[144] – User Cookies

Status Codes

- 200 OK[145] – OK
- 400 Bad Request[146] – Validation Error
- 500 Internal Server Error[147] – Internal Server Error

POST /auth/SignInById/

Authenticates the user based on the specified name and password and starts user session.

Example:

```
curl -X POST \
    -H "Content-Type: application/json" \
```

[142] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[143] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[144] http://tools.ietf.org/html/rfc2109#section-4.2.2
[145] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[146] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[147] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1

```
-d '{"Id":"9d63e3d2-cf06-4c85-a8d3-ca634dfc0131","password":"qwerty"}' \
http://localhost:5000/auth/SignInById
```

Request Headers

- Content-Type[148] – application/json

Response Headers

- Content-Type[149] – application/json
- Set-Cookie[150] – User Cookies

Status Codes

- 200 OK[151] – OK
- 400 Bad Request[152] – Validation Error
- 500 Internal Server Error[153] – Internal Server Error

POST /auth/SignInByUserName/

Authenticates the user based on the specified id and password and starts user session.

Example:

```
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{"UserName":"user1","password":"qwerty"}' \
    http://localhost:5000/auth/SignInInternal
```

Request Headers

- Content-Type[154] – application/json

Response Headers

- Content-Type[155] – application/json
- Set-Cookie[156] – User Cookies

Status Codes

- 200 OK[157] – OK
- 400 Bad Request[158] – Validation Error
- 500 Internal Server Error[159] – Internal Server Error

POST /auth/SignInByEmail/

Authenticates the user based on the specified email and password and starts user session.

Example:

[148] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[149] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[150] http://tools.ietf.org/html/rfc2109#section-4.2.2
[151] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[152] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[153] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1
[154] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[155] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[156] http://tools.ietf.org/html/rfc2109#section-4.2.2
[157] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[158] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[159] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1

```
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{"Email":"user1@infinni.ru","password":"qwerty"}' \
    http://localhost:5000/auth/SignInByEmail
```

Request Headers

- Content-Type[160] – application/json

Response Headers

- Content-Type[161] – application/json
- Set-Cookie[162] – User Cookies

Status Codes

- 200 OK[163] – OK
- 400 Bad Request[164] – Validation Error
- 500 Internal Server Error[165] – Internal Server Error

POST /auth/SignInByPhoneNumber/

Authenticates the user based on the specified phone number and password and starts user session.

Example:

```
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{"PhoneNumber":"+73216549877","password":"qwerty"}' \
    http://localhost:5000/auth/SignInByPhoneNumber
```

Request Headers

- Content-Type[166] – application/json

Response Headers

- Content-Type[167] – application/json
- Set-Cookie[168] – User Cookies

Status Codes

- 200 OK[169] – OK
- 400 Bad Request[170] – Validation Error
- 500 Internal Server Error[171] – Internal Server Error

POST /auth/SignOut/

Terminates the user session.

Example:

[160] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[161] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[162] http://tools.ietf.org/html/rfc2109#section-4.2.2
[163] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[164] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[165] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1
[166] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[167] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[168] http://tools.ietf.org/html/rfc2109#section-4.2.2
[169] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[170] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[171] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1

```
curl -X POST http://localhost:5000/auth/SignOut
```

Response Headers

- Set-Cookie[172] –

Status Codes

- 200 OK[173] – OK
- 400 Bad Request[174] – Validation Error
- 500 Internal Server Error[175] – Internal Server Error

[172] http://tools.ietf.org/html/rfc2109#section-4.2.2
[173] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[174] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1
[175] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5.1

Job Scheduler

Some applications may require execution of jobs accordingly particular time schedule. These tasks can be done by using job scheduler. The scheduler can run jobs in accordance with specific time or period. There are two key things of scheduler: schedule (с. 89) and job handler (с. 93). Those can be bound by "one to many" relation. Schedule describes a specific handler call time while the handler may be used in many schedules. So you can use one job handler in different schedules.

Job handlers relate with a specific context of job handling (с. 93), which contain information about schedule, handler fire time, the previous and the next fire time, additionally it may contain extra data defined by developer. Processing context is specified as job or job instance. Each job handling is executed in background thread.

InfinniPlatform job scheduler may run scheduled tasks in cluster infrastructure. This feature is delivered by message queue (с. 77) that guarantees processing of the job by one of the clusters nodes while computing power is equally distributed among the cluster nodes.

Getting Started with Job Scheduler

This is a brief manual to get started with the InfinniPlatform job scheduler.

1. Install InfinniPlatform.Scheduler.Quartz package:

```
dotnet add package InfinniPlatform.Scheduler.Quartz -s https://www.myget.org/F/infinniplatform/
```

2. Call AddQuartzScheduler() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddQuartzScheduler();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

3. Create MyJobHandler.cs and define job handler (с. 93):

Listing 14.1: MyJobHandler.cs

```
class MyJobHandler : IJobHandler
{
    public async Task Handle(IJobInfo jobInfo, IJobHandlerContext context)
    {
        await Console.Out.WriteLineAsync($"Greetings from {nameof(MyJobHandler)}!");
    }
}
```

4. Create MyJobInfoSource.cs and define job info source (c. 94):

Listing 14.2: MyJobInfoSource.cs

```
class MyJobInfoSource : IJobInfoSource
{
    public Task<IEnumerable<IJobInfo>> GetJobs(IJobInfoFactory factory)
    {
        var jobs = new[]
                {
                    // Job will be handled every 5 seconds
                    factory.CreateJobInfo<MyJobHandler>("MyJob",
                        b => b.CronExpression(e => e.Seconds(i => i.Each(0, 5))))
                };

        return Task.FromResult<IEnumerable<IJobInfo>>(jobs);
    }
}
```

5. Register in IoC-container (c. 8) the job handler and the job source:

```
builder.RegisterType<MyJobHandler>().AsSelf().As<IJobHandler>().SingleInstance();
builder.RegisterType<MyJobInfoSource>().As<IJobInfoSource>().SingleInstance();
```

Job Info

To plan a job one should create an information which includes at least handler type and its call time. The IJobInfo interface has next properties.

Job Info Properties

Id Unique job identifier. Required attribute. Formed automatically and represents joint properties for Group and Name, divided by . symbol. Used to form an unique identifier of job instance (c. 93) during the call of handler (c. 93).

Group Job group name. Required attribute. Used to form an unique job Id and may logically group jobs. If a group is not defined then Default is used.

Name Job name. Required attribute. Used to form an unique job Id, must be unique in Group. No default value.

State Job execution state. Required attribute. By default the job is always planned to be executed - Planned however one can define job in paused state - Paused to further resume it by a trigger or request (c. 95).

MisfirePolicy Misfire job policy. Required attribute. By default all misfired jobs are ignored - DoNothing however scheduler can execute all jobs right away and can further proceed in accordance with schedule - FireAndProceed.

HandlerType Job handler type. Required attribute. Job handler full name including a namespace and assembly name in which handler is declared. Used to invoke the handler.

Description Job description. Optional attribute. For example, detailed job logic description. Anything that can be needed for understanding of proceedings.

StartTimeUtc Start time job planning (UTC). Optional attribute. Planned immediately to be executed as put in the scheduler otherwise from the defined time. Start time should not exceed end time EndTimeUtc.

EndTimeUtc End time job planning (UTC). Optional attribute. Planned immediately until the end of app execution otherwise till the defined time. End time should be less than its start time StartTimeUtc.

CronExpression Job handler schedule in CRON (c. 89) style. Optional attribute. Defines schedule in calendar style. If it is not defined a first fire time coincides with start time StartTimeUtc.

Data Job data. Optional attribute. Value of this attribute is available in job data context (c. 93). Value of this attribute must be serializable (c. 31).

Note: Job scheduler considers values of all attributes merging all conditions by logical conjunction.

Creating Job Info

To create information about job the following factory is used IJobInfoFactory, offering a few overloads of method CreateJobInfo(). Method signature CreateJobInfo() uses DSL[176] (Domain Specific Language) which is represented as fluent interface[177].

```
IJobInfoFactory factory;

...

// Job "MyJob" will be executed daily
// at 10:35 by job handler MyJobHandler
factory.CreateJobInfo<MyJobHandler>("MyJob",
    b => b.CronExpression(e => e.AtHourAndMinuteDaily(10, 35)))
```

CRON Expression

Schedule of job execution is defined as CRON[178] style expression . cron is a classical job scheduler in UNIX like OSes utilized to run periodical tasks. CRON expressions define schedule in calendar style. For instance, “at 8:00 each day from Monday to Friday” or “at 13:30 each last Friday of month”. CRON expressions is a powerful and simple tool which require not much of efforts to get accustomed to.

CRON Expression Syntax

CRON expression is a single line consists of 6 or 7 parts divided by spaces. Each part is a condition for schedule and all of them get joined in accordance by logical multiplication (AND). The scheme below shows the structure of CRON expression. Expression parts are marked with * and lines show descriptions.

```
------------ Second
| ------------ Minute
| | ------------ Hour
| | | ------------ Day of month
| | | | ------------ Month
```

[176] https://en.wikipedia.org/wiki/Domain-specific_language
[177] http://martinfowler.com/bliki/FluentInterface.html
[178] https://en.wikipedia.org/wiki/Cron

```
| | | | | ---------------- Day of week
| | | | | | ---------------- Year
| | | | | | |
* * * * * * *
```

CRON expression parts can contain any of the allowed values, along with various combinations of the allowed special characters for that part. Table below describes allowed values and allowed special characters for each part.

Part Name	Mandatory	Allowed Values	Allowed Special Characters
Second	Yes	0-59	, - * /
Minute	Yes	0-59	, - * /
Hour	Yes	0-23	, - * /
Day of Month	Yes	1-31	, - * / ? L W
Month	Yes	1-12 (1 - January)	, - * /
Days of week	Yes	1-7 (1 - Sunday)	, - * / ? L #
Year	No	1970-2099	, - * /

CRON Special Characters

* Used to define all possible values. For example, * for minutes means “each minute”.

, Used to enumerate values. For example, 2,4,6 for week days means “Monday, Wednesday and Friday”.

\- Used to define values range. For example, 10-12 for hours means “10, 11 and 12 hours”.

/ Used to define reiteration periods. For example, 0/15 for seconds means “0, 15, 30 и 45 seconds” while 5/15 means “5, 20, 35 and 50 seconds”; 1/3 for month days means “every 3 days from 1st day of month”.

? Represents lack of specific value. Using of this character is allowed in one of the two parts - day of month or day of week, but not in both at once. For example, to plan a job execution on 10th day of each month and each week then day of month is defined as 10 while day of week is defined as ?.

L Has different meaning in each of the two parts - day of month and day of week. For example, value L for day of month means “the last day of the month” (31 for January, 29 for February in leap year). The same value for day of week means “the last day of week” - 7 (Saturday). However if used in the day of week parts after another value, it means “the last defined week day of month”. For example, 6L means “the last Friday of month”. You can also specify an offset from the last day of the month, such as L-3. For example, L-3 for day of month means “before 3 days until last day of the month”.

W Used to specify the weekday (Monday-Friday) nearest the given day of month. For example, 15W for day of month means “the nearest weekday to the 15th of the month”. For example, if 15th is Saturday, job will be executed on 14th on Friday. If 15th is Sunday, job will be executed on 16th on Monday. If 15th is Thursday, job will be execute on 15th on Thursday. However if value is equal 1W and 1st is Saturday then job will be executed on 3rd on Monday, because this rule defines jobs to be executed within one month. Combination LW is allowed and means “last weekday day of the month”.

Used to define the n-th day of week in a month. For example, 6#3 for day of week means “3rd Friday of month”, 2#1 - “the 1st Monday of the month”, 4#5 - “the 5th Wednesday of the month”.

Defining CRON Expression

CRON expression can be used when job info (с. 87) is created. In this case one of CronExpression() overloads can be used.

```
IJobInfoFactory factory;

...

// Job "MyJob" will be executed daily
// at 10:35 by MyJobHandler handler
factory.CreateJobInfo<MyJobHandler>("MyJob",
    b => b.CronExpression("0 35 10 * * ?"))
```

As you can see CRON expressions are simple and main principle of building expressions is quite clear. But it is quite easy to forget meaning of parts CRON expression or some rules of building expressions. So the CronExpression() method has a few overloads which uses DSL[179] (Domain Specific Language) concept. DSL is represented as fluent interface[180]. Next example shows recently reviewed example but with using DSL-version of the CronExpression() method.

```
IJobInfoFactory factory;

...

// Job "MyJob" will be executed daily
// at 10:35 by MyJobHandler handler
factory.CreateJobInfo<MyJobHandler>("MyJob",
    b => b.CronExpression(e => e.AtHourAndMinuteDaily(10, 35)))
```

CRON Expressions Examples

You can see examples of CRON expressions below: left - original CRON expression, right - lambda-expression to build the same expression with using ICronExpressionBuilder.

* * * * * ?

```
// Each second.
b => { }
```

0 0 12 * * ?

```
// Daily at 12:00.
b => b.AtHourAndMinuteDaily(12, 00)
```

0 15 10 * * ?

```
// Daily at 10:15.
b => b.AtHourAndMinuteDaily(10, 15)
```

0 * 14 * * ?

```
// Daily each minute from 14:00 to 14:59.
b => b.Hours(i => i.Each(14))
      .Minutes(i => i.Every())
      .Seconds(i => i.Each(0))
```

0 0/5 14 * * ?

```
// Daily each 5 minute from 14:00 to 14:55.
b => b.Hours(i => i.Each(14))
      .Minutes(i => i.Each(0, 5))
      .Seconds(i => i.Each(0))
```

0 0/5 14,18 * * ?

[179] https://en.wikipedia.org/wiki/Domain-specific_language
[180] http://martinfowler.com/bliki/FluentInterface.html

```
// Daily each 5 minutes from 14:00 to 14:55 and from 18:00 to 18:55.
b => b.Hours(i => i.EachOfSet(14, 18))
      .Minutes(i => i.Each(0, 5))
      .Seconds(i => i.Each(0))
```

0 0-5 14 * * ?

```
// Daily each minute с 14:00 по 14:05.
b => b.Hours(i => i.Each(14))
      .Minutes(i => i.EachOfRange(0, 5))
      .Seconds(i => i.Each(0))
```

0 10,44 14 ? 3 4

```
// Each Wednesday of March at 14:10 and 14:44.
b => b.Hours(i => i.Each(14))
      .Minutes(i => i.EachOfSet(10, 44))
      .Seconds(i => i.Each(0))
      .Month(i => i.Each(Month.March))
      .DayOfWeek(i => i.Each(DayOfWeek.Wednesday))
```

0 15 10 ? * 2-6

```
// Each day from Monday to Friday at 10:15.
b => b.AtHourAndMinuteDaily(10, 15)
      .DayOfWeek(i => i.EachOfRange(DayOfWeek.Monday, DayOfWeek.Friday))
```

0 15 10 15 * ?

```
// 15th each month at 10:15.
b => b.AtHourAndMinuteDaily(10, 15)
      .DayOfMonth(i => i.Each(15))
```

0 15 10 L * ?

```
// Last day of month each month at 10:15.
b => b.AtHourAndMinuteDaily(10, 15)
      .DayOfMonth(i => i.EachLast())
```

0 15 10 L-2 * ?

```
// Before 2 days until last day of every month at 10:15.
b => b.AtHourAndMinuteDaily(10, 15)
      .DayOfMonth(i => i.EachLast(2))
```

0 15 10 ? * 6L

```
// Each last Friday of every month at 10:15.
b => b.AtHourAndMinuteDaily(10, 15)
      .DayOfWeek(i => i.EachLast(DayOfWeek.Friday))
```

0 15 10 ? * 6L 2016-2020

```
// Each last Friday of every month at 10:15 from 2016 to 2020 год.
b => b.AtHourAndMinuteDaily(10, 15)
      .DayOfWeek(i => i.EachLast(DayOfWeek.Friday))
      .Year(i => i.EachOfRange(2016, 2020))
```

0 15 10 ? * 6#3

```
// Each 3rd Friday of every month at 10:15.
b => b.AtHourAndMinuteDaily(10, 15)
      .DayOfWeek(i => i.EachNth(DayOfWeek.Friday, 3))
```

0 0 12 1/5 * ?

```
// Each 5 days from 1st day of every month at 12:00.
b => b.AtHourAndMinuteDaily(12, 00)
      .DayOfMonth(i => i.Each(1, 5))
```

0 11 11 11 11 ?

```
// Every 11th November at 11:11.
b => b.AtHourAndMinuteDaily(11, 11)
      .DayOfMonth(i => i.Each(11))
      .Month(i => i.Each(Month.November))
```

0 15 10 ? * 2,4,6

```
// Each Monday, Wednesday and Friday at 10:15.
b => b.AtHourAndMinuteOnGivenDaysOfWeek(10, 15,
          DayOfWeek.Monday,
          DayOfWeek.Wednesday,
          DayOfWeek.Friday)
```

0 15 10 1,10,15 * ?

```
// 1th, 10th and 15th day at 10:15.
b => b.AtHourAndMinuteMonthly(10, 15,
          1, 10, 15)
```

Job Handler

Each schedule must relate to an implementation of the IJobHandler interface. As a result the handler will be invoked according to its schedule.

Note: A job handler must analyze all possible situation. For example, when the handler was invoked but previous handling has not finished yet; there are misfire tasks because the application was stopped for a while; the handler was invoked early than it was planned; the handler was not invoked at the exact time, and so forth.

Registering Job Handlers

Job handlers are created and managed by IoC container (c. 7) so the handlers must be registered (c. 8):

```
builder.RegisterType<MyJobHandler>().AsSelf().As<IJobHandler>().SingleInstance();
```

To register all handlers of an assembly use the RegisterJobHandlers() helper:

```
builder.RegisterJobHandlers(assembly);
```

Job Handler Context

When a job handler is invoked it gets an information to process task. This information is called the job handler context and presented as the IJobHandlerContext interface. The IJobHandlerContext interface has next properties.

InstanceId The unique job instance identifier. Formed automatically with using IJobInfo. Each instance is handled only once by some cluster node.

FireTimeUtc The actual time the trigger fired. For instance the scheduled time may have been 10:00:00 but the actual fire time may have been 10:00:03 if the scheduler was too busy.

ScheduledFireTimeUtc The scheduled time the trigger fired for. For instance the scheduled time may have been 10:00:00 but the actual fire time may have been 10:00:03 if the scheduler was too busy.

PreviousFireTimeUtc Gets the previous fire time or null if the handler was invoked the first time.

NextFireTimeUtc Gets the next fire time ot null if the handler will not be invoked anymore.

Data The job data which is defined by the job info (c. 87).

Job Handler Example

To define a job handler you need to implement the IJobHandler interface.

```
class MyJobHandler : IJobHandler
{
    public async Task Handle(IJobInfo jobInfo, IJobHandlerContext context)
    {
        await Console.Out.WriteLineAsync($"Greetings from {nameof(MyJobHandler)}!");
    }
}
```

Job Info Source

To schedule jobs you need to define the source, that is, implement the IJobInfoSource interface. After an application is started the scheduler gets all registered (c. 8) sources and inquires them to get scheduled jobs and make initialization.

Registering Job Info Source

Job sources are created and managed by IoC container (c. 7) so the sources must be registered (c. 8):

```
builder.RegisterType<MyJobInfoSource>().As<IJobInfoSource>().SingleInstance();
```

To register all sources of an assembly use the ‘RegisterJobHandlers()‘_ helper:

```
builder.RegisterJobInfoSources(assembly);
```

Persistent Job Info Source

The scheduler allows to add jobs and manage them at runtime using IJobScheduler. By default added at runtime jobs are stored in the memory of the web server so they will be lost after restarting the application. To store jobs in a persistent storage you should install an implementation of the document storage (c. 49) or implement the IJobSchedulerRepository interface, then the jobs will be scheduled even after restarting the application.

Job Info Source Example

To define a job info source you need to implement the IJobInfoSource interface.

```
class MyJobInfoSource : IJobInfoSource
{
    public Task<IEnumerable<IJobInfo>> GetJobs(IJobInfoFactory factory)
    {
        var jobs = new[]
                {
                    // Job will be handled every 5 seconds
                    factory.CreateJobInfo<MyJobHandler>("MyJob",
                        b => b.CronExpression(e => e.Seconds(i => i.Each(0, 5))))
                };

        return Task.FromResult<IEnumerable<IJobInfo>>(jobs);
    }
}
```

Job Scheduler Management

To manage the scheduler use the IJobScheduler_.

Adding and Updating Jobs

The AddOrUpdateJob() method adds or updates a job information (c. 88). By default added at runtime jobs are stored in the memory of the web server so they will be lost after restarting the application. To store jobs in a persistent storage you should install an implementation of the document storage (c. 49) or implement the IJobSchedulerRepository interface, then the jobs will be scheduled even after restarting the application. To add several jobs at once use the AddOrUpdateJobs() method. Also you can add a Paused job and not start planning it not immediately after the addition.

Deleting Jobs

The DeleteJob() method deletes the specified job. After that the job will be unscheduled and removed from the storage (c. 94). Thus you cannot resume resume (c. 95) the job after its the deletion. Nonetheless, jobs are declared in the code (c. 94) will be stopped just until restarting the application. To delete several jobs at once use DeleteJobs() or DeleteAllJobs() methods.

Pausing Jobs

The PauseJob() method stops scheduling the specified job. To pause several jobs at once use PauseJobs() or PauseAllJobs() methods.

Resuming Jobs

The ResumeJob() method starts scheduling the specified job. To pause several jobs at once use ResumeJobs() or ResumeAllJobs() methods.

Triggering Jobs

The TriggerJob() method invokes processing the specified job despite its schedule. Before triggering a job make sure it is Planned. To trigger several jobs at once use TriggerJobs() or TriggerAllJob() methods.

Getting Job Scheduler Status

There are two additional methods getting the scheduler status: IsStarted() and GetStatus(). The IsStarted() method checks whether the scheduler is started and returns true if it is started. The GetStatus() method allows to check status of the current jobs.

```
IJobScheduler jobScheduler;

// ...

var plannedCount = await jobScheduler.GetStatus(i => i.Count(j => j.State == JobState.Planned));
```

Job Scheduler via REST Services

There is the REST service to manage the job scheduler. By security reasons this service available only on a host where the application works.

GET /scheduler/
Checks whether the scheduler is started and returns the number of planned and paused jobs.

Response Headers

- Content-Type[181] – application/json

Status Codes

- 200 OK[182] – OK

GET /scheduler/jobs
Returns a list of jobs which is in specified state.

Query Parameters

- state (string) – Optional. One of the two values: planned or paused.
- skip (int) – Optional. By default - 0.
- take (int) – Optional. By default - 10.

Response Headers

- Content-Type[183] – application/json

Status Codes

- 200 OK[184] – OK

GET /scheduler/jobs/(string: id)
Returns status (c. 88) of the specified job.

Parameters

- id (string) – The job unique identifier.

Response Headers

- Content-Type[185] – application/json

Status Codes

- 200 OK[186] – OK

[181] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[182] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[183] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[184] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[185] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[186] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1

POST /scheduler/jobs/(string: id)
Adds or updates the specified job.

Parameters

- id (string) – The job unique identifier.

Form Parameters

- body – The job info (c. 88).

Request Headers

- Content-Type[187] – application/json

Response Headers

- Content-Type[188] – application/json

Status Codes

- 200 OK[189] – OK

DELETE /scheduler/jobs/(string: id)
Deletes the specified job.

Parameters

- id (int) – The job unique identifier.

Response Headers

- Content-Type[190] – application/json

Status Codes

- 200 OK[191] – OK

POST /scheduler/pause
Pauses the specified jobs.

Query Parameters

- ids (string) – Optional. Job identifiers, listed by comma.

Response Headers

- Content-Type[192] – application/json

Status Codes

- 200 OK[193] – OK

POST /scheduler/resume
Resumes the specified jobs.

Query Parameters

- ids (string) – Optional. Job identifiers, listed by comma.

Response Headers

- Content-Type[194] – application/json

Status Codes

[187] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[188] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[189] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[190] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[191] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[192] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[193] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[194] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17

- 200 OK[195] – OK

POST /scheduler/trigger

Invokes processing the specified jobs despite their schedule.

Query Parameters

- ids (string) – Optional. Job identifiers, listed by comma.

Form Parameters

- body – The data to job processing.

Request Headers

- Content-Type[196] – application/json

Response Headers

- Content-Type[197] – application/json

Status Codes

- 200 OK[198] – OK

[195] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1
[196] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[197] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17
[198] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1

Print View

Print View is the mechanism for generating a textual representation of data. Data can be any format, structured and semi-structured as well as binary. The textual representation is generated with predefined template and specified data. The template is an instance of the PrintDocument class which is usually stored as a JSON file. To creating templates is used the special WYSIWYG editor - Print View Designer (c. 100).

Unlike most of reporting engines Print View is more flexible but at the same time is low-level mechanism. The main concept is the flow content, i.e. generation a textual representation depending on data, while the reporting engines based on fixed templates. In other words, Print View allows to build human-readable representation of the specified data.

Currently Print View provides two formats: HTML and PDF.

Using Print View

To using Print View you have to go through next steps.

1. Install wkhtmltopdf[199] v0.12.2.4

Note: While installing wkhtmltopdf[200] better do not change the default installation directory, it allows to avoid needless configuration.

Note: If you install wkhtmltopdf[201] on Linux without X Server, wkhtmltopdf[202] should run via xvfb as below:

Listing 15.1: /usr/local/bin/wkhtmltopdf.sh

```
#!/bin/bash
xvfb-run -a -s "-screen 0 640x480x16" wkhtmltopdf "$@"
```

Also do not forget to set the permission for execution:

```
chmod a+x /usr/local/bin/wkhtmltopdf.sh
```

2. Install InfinniPlatform.PrintView package:

[199] https://wkhtmltopdf.org/
[200] https://wkhtmltopdf.org/
[201] https://wkhtmltopdf.org/
[202] https://wkhtmltopdf.org/

```
dotnet add package InfinniPlatform.PrintView -s https://www.myget.org/F/infinniplatform/
```

3. Call AddPrintView() in ConfigureServices():

```
using System;

using InfinniPlatform.AspNetCore;

using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        services.AddPrintView();

        // ...

        return services.BuildProvider();
    }

    // ...
}
```

4. Request the IPrintViewBuilder instance in the constructor:

```
class MyComponent
{
    private readonly IPrintViewBuilder _builder;

    public MyComponent(IPrintViewBuilder builder)
    {
        _builder = builder;
    }

    // ...
}
```

5. Create the template using Print View Designer (c. 100)

6. Use the Build() method to generate the document:

```
Func<Stream> template;
object dataSource;
Stream outStream;

// ...

await _builder.Build(outStream, template, dataSource, PrintViewFileFormat.Pdf);
```

Print View Designer

Print View Designer is the special WYSIWYG editor for creating Print View templates. Currently it works only on Windows 7+ with pre-installed .NET 4.5.

Print View Designer is an open-source project and available here:

https://github.com/InfinniPlatform/InfinniPlatform.PrintViewDesigner

1. Download a Windows installation script InfinniPlatform.PrintViewDesigner here[203].

[203] https://raw.githubusercontent.com/InfinniPlatform/InfinniPlatform.PrintViewDesigner/master/InfinniPlatform.PrintViewDesigner/Insta

2. Run Install.bat:

```
Install.bat
```

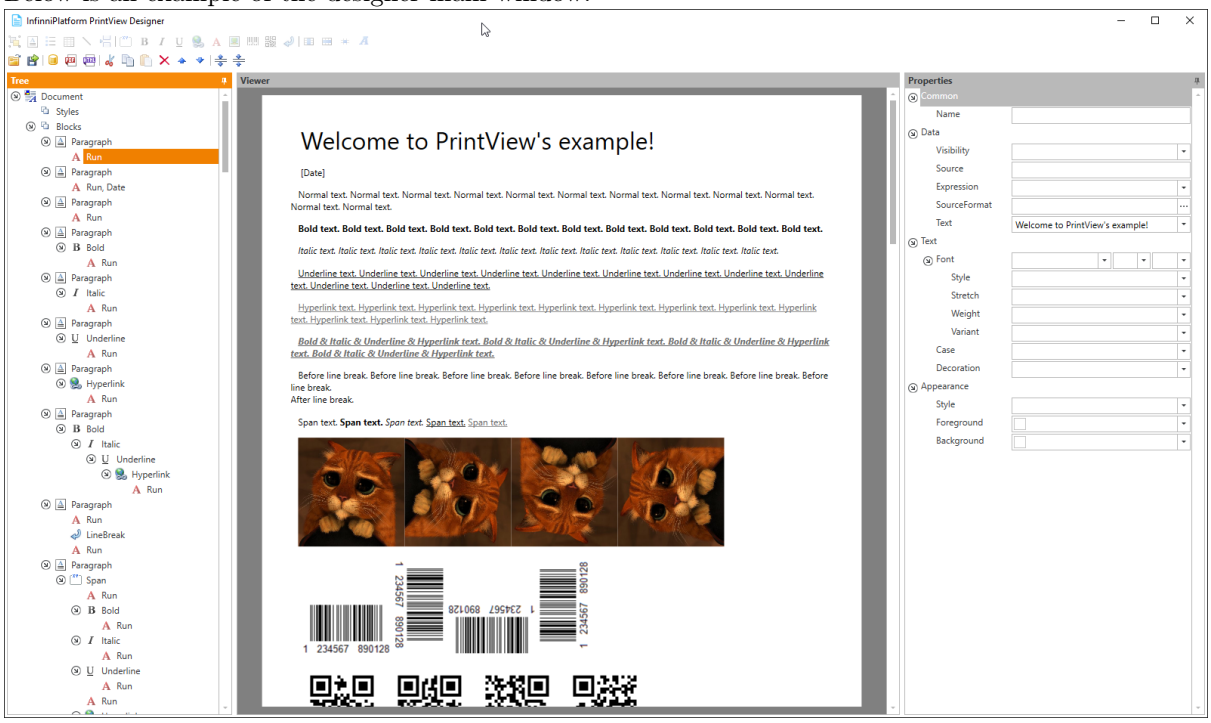
3. Go to the designer folder (where X - version number of the designer):

```
cd InfinniPlatform.PrintViewDesigner.X
```

4. Run the designer:

```
InfinniPlatform.PrintViewDesigner.exe
```

Below is an example of the designer main window.



Release Notes

This sections describes changes in InfinniPlatform versions.

InfinniPlatform 1.11.0

Данный выпуск ориентирован на улучшение общей структуры проекта и стабилизацию существующего функционала. Основной целью при этом является выделение специфической и зависимой от окружения функциональности в отдельные пакеты, которые можно будет подключать в своем решении, если в этом есть необходимость. До настоящего момента большая часть функциональности платформы предоставлялась "из коробки" целиком и включалась в приложение даже в том случае, если в этом не было необходимости.

Print View

Механизм печатных представлений (с. 99) был исключен из стандартного функционала InfinniPlatform и теперь предоставляется в виде отдельного NuGet-пакета InfinniPlatform.PrintView. Теперь, чтобы добавить функциональность печатных представлений в свой проект, необходимо установить данный пакет, выполнив следующую команду в Package Manager Console[204].

```
PM> Install-Package InfinniPlatform.PrintView
```

В связи с указанными изменениями ранее используемый интерфейс InfinniPlatform.Sdk.PrintView.IPrintViewApi был удален. Вместо него следует использовать InfinniPlatform.PrintView.Contract.IPrintViewBuilder, определенный в пакете InfinniPlatform.PrintView. Новая функциональность не требует от разработчика хранения шаблонов печатных представлений в определенном каталоге на диске. Теперь шаблон печатного представления можно хранить где угодно, например, в ресурсах самого приложения.

```
IPrintViewBuilder printViewBuilder;

...

// Поток для записи печатного представления
Stream printView;

// Данные печатного представления
object dataSource;

// Сборка, содержащая в ресурсах шаблон печатного представления
Assembly resourceAssembly;

...
```

[204] http://docs.nuget.org/consume/package-manager-console

```
// Получение шаблона печатного представления по имени ресурса
Func<Stream> template = () => resourceAssembly.GetManifestResourceStream("<PRINT VIEW RESOURCE>");

// Создание печатного представления по шаблону и данным
await printViewBuilder.Build(printView, template, dataSource, PrintViewFileFormat.Pdf);
```

Для оптимизации работы HTTP-обработчиков, результатом работы которых являются печатные представления, был добавлен класс InfinniPlatform.PrintView.Contract.PrintViewHttpResponse. Благодаря этому формируемое представление сразу выводится в выходной поток запроса, минимизируя время ожидания ответа и потребление памяти на сервере (по сравнению с предыдущей реализацией).

```
IPrintViewBuilder printViewBuilder;

...

// Шаблон печатного представления
Func<Stream> template;

// Данные печатного представления
object dataSource;

...

// HTTP-ответ с печатным представлением
var response = new PrintViewHttpResponse(printViewBuilder, template, dataSource);
```

Редактор Print View

С учетом изменений был доработан редактор печатных представлений[205].

- Добавлено отображение размера страницы в области предпросмотра.
- Добавлена подсветка в области предпросмотра выделенного элемента в дереве.
- Добавлена перекрестная навигация между областью предпросмотра и деревом (контекстное меню).
- Добавлено контекстное меню для очистки редакторов свойств (Size, Border, Margin, Padding).
- Добавлена возможность осуществлять одновременный предпросмотр в PDF и HTML.
- Добавлена возможность использовать тестовый набор данных для предпросмотра.

Static Content

Появилась возможность хостинга встроенных ресурсов приложения. Это позволит, например, хранить файлы представлений не в отдельных файлах на диске, а в соответствующей сборке (Embedded Resource). Подробнее о настройке см. статью “Хостинг статического контента”.

InfinniPlatform.Watcher

Расширение InfinniPlatform.Watcher перенесено в проект платформы в качестве пакета расширения. Теперь новые версии этого пакета будут выходить вместе с новыми версиями InfinniPlatform.

[205] https://github.com/InfinniPlatform/InfinniPlatform.PrintViewDesigner

SerializerVisibleAttribute

Появился атрибут InfinniPlatform.Sdk.Serialization.SerializerVisibleAttribute, позволяющий добавлять в список сериализуемых членов типа закрытые поля и свойства. Например, вы можете запретить изменять значения определенного свойства в коде или сделать его недоступным на уровне пользователя вашего типа, используя модификаторы доступа private или internal, но при этом разрешить сериализацию этого свойства с помощью атрибута SerializerVisibleAttribute.

```
public class Document
{
    [SerializerVisible]
    public DocumentHeader _header { get; internal set; }

    // ...
}
```

Пакеты аутентификации

До недавнего времени механизм аутентификации был неотъемлемой частью InfinniPlatform. Как показала практика, это решение не оправдало себя. Более того это требовало установки множества зависимостей, которые в большинстве случаев не использовались. По этой причине было принято решение реализовать механизм аутентификации в виде набора расширений, представленных в виде NuGet-пакетов. На данный момент реализованы следующие виды аутентификации:

- InfinniPlatform.Auth.Cookie - аутентификация с помощью Cookie;
- InfinniPlatform.Auth.Internal - аутентификация с помощью базы данных приложения;
- InfinniPlatform.Auth.Google - аутентификация с помощью учетной записи Google;
- InfinniPlatform.Auth.Facebook - аутентификация с помощью учетной записи Facebook;
- InfinniPlatform.Auth.Vk - аутентификация с помощью учетной записи ВКонтакте;
- InfinniPlatform.Auth.Adfs - аутентификация с помощью учетной записи ADFS.

Также есть возможность добавить свой способ аутентификации. Для этого нужно:

- подключить в свое решение NuGet-пакет InfinniPlatform.Http;
- реализовать интерфейс InfinniPlatform.Http.Middlewares.IHttpMiddleware с нужной логикой аутентификации;
- зарегистрировать реализацию интерфейса в контейнере зависимостей (с. 7).

Переименование

В целях улучшения структуры проекта часть типов была перенесена в другое пространство имен.

- InfinniPlatform.Sdk.Services → InfinniPlatform.Sdk.Http.Services
- InfinniPlatform.Sdk.Metadata.Documents → InfinniPlatform.Sdk.Documents.Metadata

ITenantScope

Добавлена концепция области работы с данными определенной организации InfinniPlatform.Sdk.Session.ITenantScope. Для ее использования достаточно получить зависимость InfinniPlatform.Sdk.Session.ITenantScopeProvider и определить границы области работы с помощью оператора using, как в примере ниже. Ранее доступ к данным организации осуществлялся на основе учетных данных пользователя и, таким образом, не было возможности выполнять какую-либо логику за пределами обработки запроса пользователя, кроме как напрямую обращаться к данным, минуя все высокоуровневые абстракции.

```
ITenantScopeProvider scopeProvider;

// ...

using (scopeProvider.BeginTenantScope("<Your Tenant ID>"))
{
    // Work with IDocumentStorage or IDocumentStorage<T>
}
```

Также обеспечена поддержка выполнения асинхронных операций (async/await) внутри области (using).

Плагины к InfinniPlatform и Razor view engine

Введен новый тип NuGet-пакетов - Plugins. В отличии от пакетов-расширений (Extensions), плагины устанавливаются непосредственно в папку platform и расширяют возможности InfinniPlatform. Это изменение позволило уменьшить количество зависимостей ядра платформы.

Движок отображения Razor-представлений теперь исключен из ядра платформы и представляет собой отдельный пакет InfinniPlatform.Plugins.ViewEngine.

Команда Init

Добавлена команда Init в InfinniPlatform.ServiceHost.exe и Infinni.Node.exe. Выполняет логику инициализации приложения, реализованную в методе OnInit() интерфейса IAppEventHandler:

```
public class ExampleAppInitializer : AppEventHandler
{
    public override void OnInit()
    {
        // Declare initialization logic here...
    }
}
```

Позволяет выполнить тяжелые, требовательные ко времени выполнения операции, которые необходимо выполнить при первом старте приложения (инициализация БД, миграция данных и т.п.), без запускаа самого приложения. Команда поддерживается в утилитах Infinni.Node и InfinniPlatform.ServiceHost:

```
# Infinni.Node

# Выполнить `только` инициализацию приложения
Infinni.Node.exe init -i <AppName>

# InfinniPlatform.ServiceHost

# Выполнить `только` запуск приложения
InfinniPlatform.ServiceHost.exe
InfinniPlatform.ServiceHost.exe -s
InfinniPlatform.ServiceHost.exe --start

# Выполнить `только` инициализацию приложения
InfinniPlatform.ServiceHost.exe -i
InfinniPlatform.ServiceHost.exe --init

# Выполнить инициализацию, затем старт приложения
InfinniPlatform.ServiceHost.exe -i -s
InfinniPlatform.ServiceHost.exe --init --start
```

JSON-схемы конфигурационных файлов

В файлы конфигурации можно подключить JSON-схему, что позволяет использовать автодополнение в редакторах кода (например Visual Studio, Visual Studio Code), а также получать информацию о секциях, параметрах и их возможных значений. Схемы хранятся в репозитории InfinniPlatform[206]. Для подключения, достаточно добавить в файл конфигурации поле $schema с адресом общей схемы (Common.json[207]).

```
{
   "$schema": "https://raw.githubusercontent.com/InfinniPlatform/InfinniPlatform/master/Files/Config/Schema/Common.json"
   ...
}
```

Также схему можно расширить для прикладного проекта и подключать уже расширенный вариант, достаточно унаследоваться от общей схемы (Common.json[208]), т.е. указать её в поле allOf (см. пример ниже):

```
{
   "id": "Custom.json",
   "$schema": "http://json-schema.org/draft-04/schema#",
   "title": "AppCustom",
   "description": "Custom application settings.",
   "allOf": [{
      "$ref": "https://raw.githubusercontent.com/InfinniPlatform/InfinniPlatform/master/Files/Config/Schema/Common.json"
   }],
   "properties": {
      "customProperty": {
      "type": "object",
      "description": "Some custom application setting."
      }
   }
}
```

Подробнее о JSON-схемах см. JSON schema[209] и - Understanding JSON Schema[210].

Инструменты для администрирования приложений (Infinni.Agent + Infinni.Server)

Выпущены первые версии инструментов для администрирования приложений в кластере: Infinni.Agent и Infinni.Server.

Infinni.Agent - предоставляет REST API для обращения к утилите Infinni.Node.

Infinni.Server - предоставляет REST API, а также Web-интерфейс для обращения к экземплярам приложения Infinni.Agent, установленных на машинах в кластере.

Подробнее см. соответствующие главы документации: Infinni.Agent и про Infinni.Server.

InfinniPlatform 1.10.0

Версия посвящена выпуску планировщика заданий (с. 87) и улучшению механизмов по работе с очередями сообщений (с. 77).

[206] https://github.com/InfinniPlatform/InfinniPlatform/tree/master/Files/Config/Schema
[207] https://raw.githubusercontent.com/InfinniPlatform/InfinniPlatform/master/Files/Config/Schema/Common.json
[208] https://raw.githubusercontent.com/InfinniPlatform/InfinniPlatform/master/Files/Config/Schema/Common.json
[209] http://json-schema.org/
[210] https://spacetelescope.github.io/understanding-json-schema/index.html

Job Scheduler

Появилась возможность добавлять задачи, которые будут выполняться в фоновом режиме по заданному расписанию. При этом учитывается возможность развертывания приложения в кластерной инфраструктуре, благодаря чему выполнение заданий распределяется по узлам кластера.

Для диспетчеризации заданий используется популярная и очень мощная библиотека Quartz.NET[211]. Недавно авторами этого проекта был анонсирован переход на .NET Core, что способствовало выбору данного инструмента. Благодаря Quartz.NET стало возможным использование такого гибкого инструмента планирования, как выражения CRON (с. 89).

Ко всему прочему добавлен механизм управления планировщиком заданий (с. 95) во время работы приложения, а также гибкие инструменты для его администрирования.

Для начала начала работы с планировщиком заданий InfinniPlatform можно воспользоваться статьей, доступной по следующей ссылке (с. 87).

Message Queue

Добавлена возможность управлять количеством сообщений, единовременно передаваемых получателю, что позволит быстрее освобождать очередь, а также снизить время на сетевое взаимодействие между сервером приложений и сервером очереди сообщений.

Добавлена возможность устанавливать масимальное количество потоков обработки сообщений получателем. При работе с асинхронными методами без такого ограничения может возникнуть слишком большое количество потоков, что может привести к ошибкам и негативно сказаться на производительности.

.NET Framework 4.5.2

Выполнен переход на версию 4.5.2 .NET Framework, т.к. эта версия требуется для последних версий используемых сторонних Nuget-пакетов. При переходе на новый релиз нужно изменить значение используемой версии .NET в файле InstallPlatform.ps1:

```
[Parameter(HelpMessage = "Version of the .NET.")]
[String] $framework = 'net452'
```

[211] https://www.nuget.org/packages/Quartz

Indices and tables

- genindex
- search

J

L

M

N

O

P

R

S

T