Автоматизированный сервис для исполнения процессов в нотации BPMN

Инструкция пользователя

ООО «Ак Барс Цифровые Технологии»
2021 г.

Оглавление

Ограничение использования документа

Настоящий документ является результатом интеллектуальной деятельности, исключительное право на которое принадлежит Обществу с ограниченной ответственностью «Ак Барс Цифровые Технологии» (именуемое далее правообладатель).

Любое использование (как полностью, так и в части) настоящего документа (в частности: копирование, воспроизведение, распространение, доведение до всеобщего сведения и т.д., в цифровой форме и/или на бумажных носителях) допускается только по соглашению с правообладателем. Нарушение исключительного права преследуется в соответствии с законодательством Российской Федерации, нормами международного права.

Правообладатель вправе вносить изменения в Программный Продукт, настоящую документацию без предварительного уведомления Лицензиата.

Описание основных функций продукта

Назначение продукта

Платформа предназначена для автоматизации прикладных бизнес-процессов, а именно для оркестрации вызовов различных сервисов и приложений. В том числе, когда в процессе есть необходимость выполнения «ручных» этапов человеком.

Платформа определяет последовательность запусков, условия запуска (в том числе, на основе итогов работы одного из сервисов), режим сервисов (синхронный/асинхронный) и тому подобное.

Сервисы могу иметь неограниченную длительность работы, запускаться через указанные промежутки времени и зависеть от действий, выполненных человеком (выбор варианта, ввод текста, и т.д.). Также возможно участие в процессе внешних систем, результаты работы которых влияют на процесс.

Потенциальным потребителем данного сервиса является любое сочетание систем (сервисов), работающих в рамках единого бизнес-процесса и связанных друг с другом.

Функциональные характеристики продукта

1. Подключение SDK к приложению и конфигурирование.
2. Запуск процессов по указанным в виде схемы BPMN правилам и последовательности.
3. Исполнение задачи, полученной от BPM Engine.
4. Обработка событий извне и связывание с ожидающей задачей в BPM Engine.
5. Исполнение асинхронной задачи, полученной от BPM Engine (паттерн Async External Task).
6. Обработка событий извне и завершение асинхронной задачи в BPM Engine (паттерн Async External Task).

Особенности продукта

Автоматизированный сервис для исполнения процессов в нотации BPMN не имеет визуального интерфейса и предполагает свое использование только посредством SDK и API.

Термины и определения

Термин	Определение\Полное наименование
BPMN (Business Process Management Notation)	Язык моделирования бизнес-процессов.
Process Definition (схема, BPMN-схема, схема процесса)	Экземпляр конкретной версии BPMN-схемы.
Бизнес-сервис	Соответствует определению ArchiMate.
Бизнес-процесс	Последовательность этапов исполнения бизнес-сервиса и логики перехода между ними, отражающая исключительно точку зрения бизнеса. Бизнес-процесс не зависит от прикладных систем и технических деталей интеграции между ними. В контексте данного решения необходимо четко различать бизнес-процессы уровней front/ middle/ back и не допускать размещение логики разных уровней в рамках одного процесса. Передача управления от одного уровня к другому осуществляется аналогично интеграции с внешней системой.
Системный процесс	Сценарий последовательности вызовов между различными системами.
Рабочий процесс/ workflow	Исполняемый экземпляр конкретного процесса, описанного в нотации BPMN и имеющий текущее состояние. Рабочий процесс может выполнять как бизнес-процесс, так и системный. При правильном подходе к реализации оркестрации эти два типа процесса не смешиваются на одной схеме. Системные процессы могут реализовываться в виде подпроцессов основного бизнес-процесса.
Variables (Process Variables, переменные)	Именованный список строковых переменных конкретного экземпляра процесса.
Routing variable	Обычная переменная, на основе которой строится маршрут исполнения процесса по схеме.
Контекст процесса	Минимально необходимый набор переменных и их значений в рамках определенного экземпляра рабочего процесса, который используется для принятия логических решений и запуска этапов.
Entity (сущность)	Исходная сущность, в рамках которого исполняется процесс (например, заявка на кредит, заказ на доставку).

Данные процесса	Бизнес-значимые данные, сущности предметной области бизнес-процесса, которые читаются, создаются или изменяются при исполнении процесса. Срок жизни таких данных обычно превышает срок жизни отдельного экземпляра процесса.
Этап процесса	Атомарный с точки зрения процесса шаг (действие).
Системный этап	Действие, не требующее интеграции с внешними системами или участия пользователя. Например, вычислительная обработка данных или контекста процесса.
Интеграционный этап	Действие, требующее для исполнения вызова внешней системы.
Ручной этап	Действие, требующее участия человека (пользователя).
External Task (внешняя задача)	Внешняя задача. В терминологии BPMN это подтип Service Task. Ставится BPM-движком на исполнение, а исполняет ее отдельный интеграционный сервис (.Net). Подробнее тут и тут.
Receive Task (задача на ожидание)	Задача на ожидание event-а, для продолжения процесса. Подробнее тут.
Event (событие)	Некое событие, позволяющее продолжить работу данного процесса (который ранее «встал на ожидание» данного события).
Handler (обработчик)	Общее название C#-классов, зона ответственности которых состоит в том, чтобы вызвать некий интеграционный код и/или сформировать на выходе Variables процесса (для использования на следующих этапах).
Task Handler	Подтип Handler-а специализированный под исполнение External Task.
Event Handler	Подтип Handler-а специализированный под исполнение Receive Task.
Async Task Handler	Подтип Handler-а специализированный под исполнение External Task в асинхронном режиме.
Async Event Handler	Подтип Handler-а специализированный под исполнение External Task в асинхронном режиме.
Handler Context (контекст обработчика, контракт обработчика)	Типизированный набор свойств, минимально необходимый для исполнения обработчика.
Проект	Набор разворачиваемых и исполняемых компонентов, который рассматривается как одна система. Может поддерживать несколько бизнес-процессов, однако развитие проекта должно осуществляться в рамках одной команды.

Инструкция пользователя

Общий порядок разработки приложения на основе платформы

Разработка приложения на основе SDK платформы (далее SDK) предполагает несколько обязательных шагов:

- Разворачивание SDK.
- Подключение SDK в host сборку и конфигурирование.
- Разработка схемы процесса и его подключение к платформе.
- Разработка обработчиков:
  - Для обработки событий BPM Engine при движении по схеме процесса (Task Handler);
  - Для обработки внешних событий (из очереди, из пользовательского UI) (Event Handler).
- Подключение SDK в domain сборку, в т. ч. регистрация обработчиков.

Далее обеспечивается возможность запуска разработанного процесса и, при необходимости, отслеживание событий SDK.

Разворачивание SDK

Действия выполняются согласно руководству по установке платформы.

Подключение SDK в host сборку (web, console)

host-сборка - это запускаемая библиотека, задачи которой сводятся к:

- хостингу функционала приложения (например, в рамках ASP.Net Core);
- подключение набора domain-сборок;
- регистрация зависимостей, сторонних библиотек, HTTP обработчиков;
- авто-прогон миграций БД;
- глобальная обработка ошибок и т.д.;
- в контексте платформы в рамках этой сборки проводится регистрация Handler-ов и их запуск в работу.

Пакет Abdt.LiveBpm.All обеспечивает:

- подключение всего SDK целиком и всех его сервисов;
- запуск среды исполнения обработчиков;

- гибкое конфигурирование всего SDK целиком.

Метод AddLiveBpm

Регистрирует сервисы для среды исполнения обработчиков (handlers). Используемые параметры:

1. Обязательные

- AppName - уникальное имя приложения;
- CamundaUrl - Базовый URL Camunda;
- EventProducers - именованный список EventProducer-ов и параметров подключения к RabbitMq (обязательно при использовании Event Handlers).

2. Настройки обработчиков

- DefaultTaskHandlerSettings - свойство для чтения/изменения объекта настроек по умолчанию для Task Handlers (для перекрытия целиком всех настроек) (см. Task Handler Settings);
- ConfigureDefaultTaskHandlerSettings - дополнительный метод для перекрытия настроек по умолчанию для Task Handlers (для перекрытия отдельных полей) (см. Task Handler Settings);
- DefaultEventHandlerSettings - свойство для чтения/изменения объекта настроек по умолчанию для Event Handlers (для перекрытия целиком всех настроек) (см. Event Handler Settings);
- ConfigureDefaultEventHandlerSettings - дополнительный метод для перекрытия настроек по умолчанию для Event Handlers (для перекрытия отдельных полей) (см. Event Handler Settings).

3. Общие настройки

- CamundaTimeout - таймаут HTTP запросов к Camunda (по умолчанию 5 мин);
- PrivateFields - поля для маскирования логов (запросы в Camunda и т.д.);
- LogMaskingEnabled - маскирование включено (по умолчанию true);
- DatabaseConnectionString - подключение SQL Server/PostgreSQL для сохранения ActionLogs;
- DatabaseType - тип БД (SQLServer, PostgreSQL), по умолчанию – SQLServer;
- RedisConnectionString - подключение к Redis для реализации глобальных блокировок по данному процессу;

- AutoMigrate - автомиграция БД ActionLogs при старте приложения (можно использовать «вручную» IDatabaseMigrationsRunner);
- MasterDataJsonDataSet - подключение метаданных MasterData;
- AutoDeploySchemes - автоматический деплой схем при старте сервиса (по умолчанию true);
- DeploySchemesFolder - относительный путь к папке, содержащей BPMN-схемы (*.bpmn) для деплоя (поддерживается рекурсивный вложенный поиск, по умолчанию "resources/bpmn");
- DeployChangedOnly - деплоить только реально измененные схемы (по умолчанию true);
- DeployDuplicateFiltering – исключать дублирование ресурсов при деплое (по умолчанию true).

Состав Task Handler Settings

- WorkerReplicas - количество реплик Workers. По умолчанию = 1;
- MaxExternalTasksPerIteration - максимальное количество задач, обрабатываемых в параллели. По умолчанию = 1;
- LongPollingTimeout - таймаут (максимальное время ожидания) для Long polling. По умолчанию = 25 sec. (почему так: таймаут OpenShift 30 sec. минус 5 sec.);
- DistributedLockEnabled - флаг включения глобальных блокировок via Redis по каждому процессу. Для исключения выполнения в параллели нескольких этапов одного и того же процесса (исключаются гонки по БД и т. д.). По умолчанию = false;
- DistributedLockTimeout - таймаут на снятие глобальной блокировки. По умолчанию = 5 min.;
- DistributedLockWaitTimeout – таймаут между попытками получения распределенной блокировки. По умолчанию = 10 sec.;
- ExternalTaskLockDuration - время жизни распределенной блокировки External task в Camunda (при fetchAndLock). По умолчанию = 5 min.;
- ExtendExternalTaskLockDuration - время жизни распределенной блокировки External task в Camunda (при extendLock). По умолчанию = 1 hour;
- RetryCount - количество попыток повторного исполнения External Task. По умолчанию = 3;

- RetryTimeout - таймаут для повторных попыток исполнения External Task. По умолчанию = 5 min.;
- IdleExponent - экспонента в зависимости от количества ошибок для Idle;
- LocalRetryCount - количество попыток для локальных (via Polly) повторных вызовов внешнего API (like Camunda REST API). По умолчанию = 3;
- LocalRetryTimeout - таймаут для локальных (via Polly) попыток. По умолчанию = 1 sec.;
- BpmEngineRetryCount - количество локальных (via Polly) попыток в контексте вызовов BPM Engine Camunda. По умолчанию = 5;
- BpmEngineMinRetryTimeout - минимальный таймаут для локальных (via Polly) попыток в контексте вызовов BPM Engine Camunda. По умолчанию = 1 sec.;
- BpmEngineMaxRetryTimeout - максимальный таймаут для локальных (via Polly) попыток в контексте вызовов BPM Engine Camunda. По умолчанию = 5 sec.;
- MaxHandlerTimeout - максимальное время исполнения Handler-a. По умолчанию = 5 min.

Состав Event Handler Settings

- ConsumerReplicas - количество реплик Consumers. По умолчанию = 1;
- RetryCount - не используется;
- RetryTimeout - не используется;
- DistributedLockEnabled - флаг включения глобальных блокировок via Redis по каждому процессу. Для исключения выполнения в параллели нескольких этапов одного и того же процесса (исключаются гонки по БД и т.д.). По умолчанию = false;
- DistributedLockTimeout - таймаут на снятие глобальной блокировки. По умолчанию = 5 min.;
- DistributedLockWaitTimeout – таймаут между попытками получения распределенной блокировки. По умолчанию = 10 sec.;
- LocalRetryCount - количество повторных попыток обработки сообщения. По умолчанию = 10;
- LocalRetryTimeout - количество таймаута между повторными попытками обработки сообщения. По умолчанию = 10 sec.;

- MaxHandlerTimeout - max время исполнения Handler-а. По умолчанию = 5 min.

Метод UseLiveBpm

Запускает сервисы для среды исполнения обработчиков (handlers).

Пример

Подключение пакета:

```
dotnet add package Abdt.LiveBpm.All
```

Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    services.AddMvc(opt =>
        {
            opt.Filters.Add<BusinessOperationLoggingFilter<ClientException>>();
            opt.Filters.Add<TracingOperationFilter>();
        })
        .AddLiveBpm() // #SDK API (optional)
        .AddApi()
        .AddControllersAsServices();

    // ...

    // #application handlers
    services.AddHandlers();

    // #SDK Core
    services.AddLiveBpm(c =>
    {
        c.AppName = "Abdt.MyDomain.MyApplication";
        c.CamundaUrl = Configuration["App:CamundaUrl"];
        c.EventProducers = Configuration.GetSection("EventProducers").Get<EventProducers>();
        c.DatabaseType = DatabaseType.PostgreSql;
        c.DatabaseConnectionString = Configuration["App:DbConnectionString"];

        // перекрытие настроек по умолчанию для всех Handler-ов
            c.ConfigureDefaultTaskHandlerSettings = settings =>
            {
                settings.MaxExternalTasksPerIteration = 5;
                settings.RetryCount = 20;
                settings.LocalRetryCount = 20;
            };

        // перекрытие настроек по умолчанию для всех Handler-ов
            c.ConfigureDefaultEventHandlerSettings = settings =>
            {
                settings.ConsumerReplicas = 5;
                settings.LocalRetryCount = 20;
                settings.RetryCount = 20;
```

```
        };
    });
}

[UsedImplicitly]
public void Configure(IApplicationBuilder app, IHostingEnvironment env, IApplicationLifetime lifetime,
    ILogManager logManager, IOptions<SwaggerOptions> swaggerOptions, IServiceProvider serviceProvider)
{
    MigrationsRunner.ApplyMigrations(logManager.GetLogger("Migrator"),
serviceProvider, "Abdt.MyDomain.MyApplication.Web").Wait();

    // #Start
    app.UseLiveBpm(serviceProvider);

    // ...
}
```

Пример EventProducers в appSettings.json

```
{
  "App": {
    "DbConnectionString": "Data Source=.\\SQLEXPRESS;Initial Catalog=Abdt.LiveBpm.TestHost;Integrated
Security=SSPI;",
    "RedisCacheConnection": "localhost:6379",
    "RedisCachePrefix": "Abdt.LiveBpm.TestHost",
    "CamundaUrl": "http://localhost:8082",
    "OrderHttpRepositoryUrl": "http://localhost:5000/health"
  },
  "UseSwagger": true,
  "GlobalPrefix": "",
  "Kestrel": {
    "EndPoints": {
      "Http": {
        "Url": "http://0.0.0.0:5000"
      }
    }
  },
  "EventProducers": {
    "ConnectionStrings": {
      "default":
"host=127.0.0.1:5672;publisherConfirms=true;username=rabbit;password=rabbit;requestedHeartbeat=3600",
      "CRM":
"host=127.0.0.1:5672;publisherConfirms=true;username=rabbit;password=rabbit;requestedHeartbeat=3600"
    }
  }
```

Подключение SDK в domain-сборку (с обработчиками)

domain-сборка — это стандартная библиотека классов, её задачи:

- реализация конкретного набора бизнес-функций (например, вызов функции другой системы для списания средств со счета, авторизация средств и т. д.);
- в итоге подключается в host-сборку;

- в контексте платформы в рамках этой сборки имплементируется нужный набор Handler-ов.

Пакет Abdt.LiveBpm.Core

- Описывает основные интерфейсы обработчиков (ITaskHandler, IEventHandler, IAsyncTaskHandler, IAsyncEventHandler) и их контракты;
- Обеспечивает регистрацию обработчиков;
- Обеспечивает гибкое конфигурирование данного отдельного обработчика и среды его исполнения;
- Также предоставляет несколько public сервисов (например, IStartProcessService).

Метод AddHandler

Регистрирует обработчик для исполнения External Task. При регистрации Handler-а SDK поднимает HostedService, который периодически опрашивает BPM-engine на предмет наличия новых External Tasks к исполнению.

Возможно задание параметров на этапе регистрации:

- TopicName - TopicName (из BPMN схемы);
- Settings - метод для перекрытия настроек по умолчанию (для данного Handler) (см. Task Handler Settings);
- GroupName - имя группы для группировки (группировка в один Worker);
- MediatorProviderType - тип BPM Engine.

Метод AddEventHandler

Регистрирует обработчик event-а для исполнения Receive Task. При регистрации Handler-а SDK поднимает Consumer-а, который подписывает на канал (например, очередь в RabbitMq).

Возможно задание параметров на этапе регистрации:

- MessageName - MessageName (из BPMN схемы);
- Settings - метод для перекрытия настроек по умолчанию (для данного Handler) (см. Event Handler Settings);
- TopicName - TopicName исходной Async External Task pattern (из BPMN схемы);

- ProducerKey - ProducerKey для получения ConnectionString & группировки (группировка в один HostedService);
- ChannelType - тип канала (RabbitMQ, Own);
- Channel - описание канала для получения event-а;
- MediatorProviderType - тип BPM Engine;
- SerializerType - Ттип серилиализатора.

Метод AddAsyncHandler

Регистрирует обработчик для исполнения "Async External Task" (паттерн на основе External Task). Параметры аналогично AddHandler.

Метод AddAsyncEventHandler

Регистрирует обработчик event-а для исполнения "Async External Task" (паттерн на основе External Task). Параметры аналогично AddEventHandler

Пример

Подключение пакета:

```
dotnet add package Abdt.LiveBpm.Core
```

Entry.cs:

```
public static IServiceCollection AddHandlers(this IServiceCollection services)
{
   // регистрация Task Handlers
   services.AddHandler<TaskHandler1>(topicName: "TaskHandler1");
   services.AddHandler<TaskHandler2>(topicName: "TaskHandler2");
   services.AddHandler<MultiInstanceTaskHandler>(c =>
   {
      c.TopicName = "MultiInstanceTask";
      c.GroupName = "Group #1"; // для объединения обработчиков в один Worker

      // перекрытие настроек по умолчанию
      c.Settings = settings =>
      {
         settings.LocalRetryCount = 10;
         settings.LocalRetryTimeout = TimeSpan.FromMinutes(25);
         settings.ExternalTaskLockDuration = TimeSpan.FromMinutes(15);
      };
   });

   // Регистрация Event Handlers
   services.AddEventHandler<EventHandler1>(messageName: "MessageName_EventHandler1");
   services.AddEventHandler<EventHandler1>(c =>
   {
      c.ProducerKey = "SomeProducer"; // ключ Producer-а из настроек (мета-данные + ConnectionStrings)
      c.MessageName = "AsyncTaskHandler2";

      // параметры очереди в RabbitMq
```

```
        c.Channel.ExchangeName = "ExchangeName";
        c.Channel.QueueName = "QueueName";
        c.Channel.RoutingKey = "RoutingKey";

        // перекрытие настроек по умолчанию
        c.Settings = settings =>
        {
            settings.ConsumerReplicas = 2;
            settings.LocalRetryCount = 3;
        };
    });

    // Регистрация пары Task Handler + Event Handler (для Async External Task)
    services.AddAsyncHandler<AsyncTaskHandler2>(topicName: "AsyncTaskHandler2");
    services.AddAsyncEventHandler<AsyncEventHandler2>();

    return services;
}
```

Разработка обработчиков

Основные принципы разработки обработчиков:

- Представляет собой реализацию Command Handler Pattern
  - https://exceptionnotfound.net/implementing-cqrs-in-net-part-2-handling-commands-and-events/;
  - https://buildplease.com/pages/fpc-10/;
  - https://blogs.cuttingedge.it/steven/posts/2011/meanwhile-on-the-command-side-of-my-architecture/;
  - Представляет собой довольно небольшой кусок атомарного кода, очень похожий на AWS Lambda function.
- «Не знает» ничего про окружение (где он запускается), но довольно много «знает» о том, как хорошо исполнить свою атомарную задачу (например, вызов стороннего сервиса).
- Всегда есть строго определенный «входящий контракт» для исполнения своей работы (т. н. параметры команды, контекст и т. д.). Контракт является минимально необходимым.
- Почти всегда есть четко определенный «исходящий контракт» факта исполнения своей работы (переменные процесса, поля в сущности).
- Возможность запуска «вне-процесса».
- Быстро создать и легко «выкинуть».
- Декораторы и окружение
  - Handler «оборачивается» набором декораторов (логирование, аудит, вызов Camunda по http, обработка ошибок и т. д.) для реализации сквозной функциональности.

  - Выполняется в рамках какого-либо Worker-а или Consumer-а.
  - Поэтому не рекомендуется вносить в handlers код реализации сквозной функциональности (если она, конечно, есть в SDK)
- Должен поддерживать retry (т.е. код handler может быть вызван повторно via SDK).
- Устанавливается политика по вызову Cancel (для CancellationToken) через MaxHandlerTimeout (настройка).
- Данные в переменных процесса или БД
  - Вы можете хранить все данные в Process variables, не используя вообще свою внутреннюю БД;
  - Но плата за это – производительность;
  - Также возможно труднее будет отлавливать «потерянные изменения» (а можно бы было через database concurrency exception);
  - И Process variables это временные данные, время жизни которых равна времени жизни процесса;
  - Поэтому рекомендации по хранению просты:
    - Исходную модель документа (например, клиентские данные по заявке) нужно хранить, используя сущность базового документа (своя внутренняя БД);
    - Все новые свойства и атрибуты, которые появляются в процессе исполнения, хранятся в Process variables;
    - Новое свойство и атрибут можно продублировать в свою БД отдельным полем;
    - Если новой свойство имеет большой размер - не рекомендуется его хранить в Process variables;
    - Если это routing-переменная (т. е. которая влияет на маршрут исполнения процесса) - 100% это Process variables.
  - Целевая БД задается через параметр DatabaseType (SQLServer, PostgreSQL). По умолчанию SQLServer.

Функции Task Handler (и его асинхронной версии)

- Основная функция - вызов другой системы (т. е. интеграционный код), процессинг переменных, заполнение переменных для routing-а на схеме и т.д.;
- Также внутри может создавать «нужный» request-а для API вызова (адаптера или системы напрямую) через AutoMapper и т. д.;

- Метод MapIn - создает экземпляр «входящего контракта» (контекста для исполнения своей работы) используя Process variables + свое внешнее хранилище;
- Метод Validate - валидирует свой контекст исполнения;
- Метод Handle - создает API request и исполняет интеграционный код, выполняет процессинг. Результат складывается либо в переменные, либо в out-свойства контекста (включая переменные для routing-а);
- Метод MapOut (optional) – «выплевывает» наружу результат своей работы (переменные либо out-свойства контекста) в какое-либо внутреннее хранилище (если необходимо).

Функции Event Handler (и его асинхронной версии)

- Основная функция - Event-Handling (не только обработчика, но и SDK) - оповестить BPM Engine о получении нужного события, чтобы BPM Engine смог двинуть далее процесс;
- Также может провести небольшой процессинг сообщения, с целью наполнить Process variables;
- Не рекомендуется делать здесь тяжелый процессинг и вызов внешних систем (т. к. retry будет только «локальный, в рамках одного потока» и довольно через малые промежутки времени);
- Метод Correlate – «коррелирует» (связывает) сообщение с запущенным процессом либо с асинхронной операций;
- Метод Handle - исполняет процессинг сообщения и при необходимости, заполняет routing-переменные.

Рекомендации при разработке

- Нет зависимостей на другие handlers;
- Нет шаринга (sharing, переиспользования) кода бизнес-логики между другими handlers (т. е. лучше выбрать copy-paste);
- Общий интеграционный код (маппинг в API requests, HTTP clients) - можно и нужно переиспользовать;
- Поддержка retry;
- Принцип единственной ответственности (Single Responsibility Principle, SRP);

- Поддержка исполнения в параллели с другими handlers в рамках одного процесса;
- Новая версия handler-а вводится через новый Topic + новый класс (для Task handler);
- Все новые свойства и атрибуты, которые появляются в процессе исполнения, хранятся в Process variables;
- Минимально-возможный контракт контекста исполнения (исключая конечно задачи вида «Положить целиком всю заявку в ОКЗ» - здесь контракт должен содержать заявку целиком отдельным свойством);
- «Условный» (т.е. на IF-ах) интеграционный код лучше разделить на два handler + routing variable;
- Предпочитайте вызов адаптеров прямым вызовам систем;
- В методе MapOut не нужно использовать сущность из контекста (ранее полученную из БД) для сохранения в БД. Нужно заново ее получить по ID из БД (для возможности retry по database concurrency exception);
- Внутренний storage provider (для работы с внутренней сущностью) должен поддерживать оптимистичную блокировку;
- https://docs.microsoft.com/ru-ru/ef/core/saving/concurrency;
- https://www.mssqltips.com/sqlservertip/6115/how-to-handle-concurrency-in-entity-framework-core/;
- рекомендуется подход с полем: public byte[] RowVersion {get; set;};
- Также рекомендуется установить жизненный цикл DbContext вручную "ServiceLifetime.Transient" для исключения ошибок concurrency;
- Не усложняйте БД и схему хранения сущности «третьими нормальными формами» - рекомендуется вложенные сущности хранить как JSON (в string полях).

Связывание этапов BPMN-схемы (элементов) и соответствующего им .Net кода

Осуществляется двумя путями:

- Для элемента External Task
  - На элементе указывается TopicName;
  - При регистрации .Net Handler-а - ему указывается тот же самый TopicName;
  - .Net Handler является по сути удаленным исполнителем задачи для элемента External Task;

  - https://docs.camunda.org/manual/latest/reference/bpmn20/tasks/service-task/;
  - https://docs.camunda.org/manual/latest/user-guide/process-engine/external-tasks/.
- Для элемента Receive Task
  - На элементе указывается MessageName;
  - При регистрации .Net Handler-a - ему указывается тот-же самый MessageName;
  - .Net Handler является по сути удаленным оповещателем о получении некоего события для Receive Task;
  - https://docs.camunda.org/manual/latest/reference/bpmn20/tasks/receive-task/.

Остальные элементы BPMN-схемы исполняются только в контексте BPM-engine (Camunda) и связи с .Net кодом не имеют (т. е. не имеют исполнителей - Handlers).

Пример TaskHandler1.cs (Простой обработчик)

Реализует интерфейс ITaskHandler.

```
/// <summary>
/// Обработчик этапа с topicName = "TaskHandler1"
/// </summary>
public class TaskHandler1 : ITaskHandler<TaskHandler1CommandParams>
{
    private readonly SomeHttpClient _someHttpClient;

    public TaskHandler1(SomeHttpClient someHttpClient)
    {
        _someHttpClient = someHttpClient;
    }

    /// <summary>
    /// Создание локального контекста выполнения задачи (параметров команды)
    /// </summary>
    public Task<TaskHandler1CommandParams> MapIn(TaskHandleContext processContext)
    {
        var result = new TaskHandler1CommandParams()
        {
            SomeId = processContext.ExternalTask.Variables.GetVariableValue("SomeId"),
        };

        return Task.FromResult(result);
    }

    /// <summary>
    /// Валидация контекста выполнения задачи
    /// </summary>
    public Task<ValidateResult> Validate(TaskHandler1CommandParams context)
```

```
    {
        Guard.NotNull(context, "context != null");
        Guard.NotEmpty(context.SomeId, "context.SomeId != null");

        return Task.FromResult(ValidateResult.Success());
    }

    /// <summary>
    /// Код вызова
    /// </summary>
    public async Task<HandleResult> Handle(TaskHandler1CommandParams context, TaskHandleContext processContext, CancellationToken cancellationToken)
    {
        var request = new { someId = context.SomeId, };
        var response = await _someHttpClient.Post(request);

        var vars = new[]
        {
            ("TaskHandler1WasCompleted", true.ToString())
        };

        var result = new HandleResult(request, response, vars);
        return result;
    }

    /// <summary>
    /// Сохранение результата исполнения задачи "во-вне" при необходимости
    /// </summary>
    public Task MapOut(TaskHandler1CommandParams context, HandleResult handleResult, CancellationToken cancellationToken)
    {
        return Task.CompletedTask;
    }
}

/// <summary>
/// Контекст выполнения задачи TaskHandler1 (параметры команды)
/// </summary>
public class TaskHandler1CommandParams
{
    public string SomeId { get; set; }
}
```

Пример TaskHandler2.cs (с использованием внутренней БД)

Реализует интерфейс ITaskHandler.

```
/// <summary>
/// Обработчик этапа с topicName = "TaskHandler2"
/// </summary>
public class TaskHandler2 : ITaskHandler<TaskHandler2CommandParams>
{
    private readonly SomeHttpClient _someHttpClient;
    private readonly IOrderRepository _orderRepository;

    public TaskHandler2(SomeHttpClient someHttpClient, IOrderRepository orderRepository)
    {
        _someHttpClient = someHttpClient;
        _orderRepository = orderRepository;
    }
```

```
/// <summary>
/// Создание локального контекста выполнения задачи (параметров команды)
/// </summary>
public async Task<TaskHandler2CommandParams> MapIn(TaskHandleContext processContext)
{
    // подгружаем часть необходимых данных из БД (т.к. не храним их в variables по разным причинам)
    var order = await _orderRepository.Get(processContext.Document.DocumentId);

    var result = new TaskHandler2CommandParams()
    {
        SomeId = processContext.ExternalTask.Variables.GetVariableValue("SomeId"),
        OrderId = order?.Id,
        OrderName = order?.Name,
    };

    return result;
}

/// <summary>
/// Валидация контекста выполнения задачи
/// </summary>
public Task<ValidateResult> Validate(TaskHandler2CommandParams context)
{
    Guard.NotNull(context, "context != null");
    Guard.NotEmpty(context.SomeId, "context.SomeId != null");
    Guard.NotEmpty(context.OrderName, "context.OrderName != null");

    return Task.FromResult(ValidateResult.Success());
}

/// <summary>
/// Код вызова
/// </summary>
public async Task<HandleResult> Handle(TaskHandler2CommandParams context, TaskHandleContext processContext, CancellationToken cancellationToken)
{
    var request = new { someId = context.SomeId, orderName = context.OrderName, };
    var response = await _someHttpClient.Post(request);

    var vars = new[]
    {
        ("TaskHandler2WasCompleted", true.ToString()),
        ("SomeData", "foo"),
    };

    var result = new HandleResult(request, response, vars);
    return result;
}

/// <summary>
/// Сохранение результата исполнения задачи "во-вне" при необходимости
/// </summary>
public async Task MapOut(TaskHandler2CommandParams context, HandleResult handleResult, CancellationToken cancellationToken)
{
    // запрашиваем сущность каждый раз для исключения database concurrency exception (через Retry)
    var order = await _orderRepository.Get(context.OrderId);
```

```
        // обновляем данные сущности
        order.SomeData = handleResult.Result.Variables.GetVariableValue("SomeData");

        // commit
        await _orderRepository.Update(order);
    }
}

/// <summary>
/// Контекст выполнения задачи TaskHandler2 (параметры команды)
/// </summary>
public class TaskHandler2CommandParams
{
    public string SomeId { get; set; }
    public string OrderId { get; set; }
    public string OrderName { get; set; }
}
```

Пример EventHandler1.cs

Реализует интерфейс IEventHandler.

```
/// <summary>
/// Обработчик receive task с MessageName = MessageName_EventHandler1
/// </summary>
public class EventHandler1 : IEventHandler<SomeExternalEventMessage>
{
    private readonly IOrderRepository _orderRepository;

    public EventHandler1(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public async Task<CorrelateResult> Correlate(SomeExternalEventMessage message, EventHandleContext processContext)
    {
        // Коррелируем полученный event с запущенным процессом (т.е. любыми средствами и правдами/неправдами вычисляем ProcessId)
        var order = await _orderRepository.Get(message.OrderId);
        var result = new CorrelateResult(order.ProcessId);
        return result;
    }

    public Task<HandleResult> Handle(SomeExternalEventMessage message, EventHandleContext processContext)
    {
        var vars = new[]
        {
            ("EventHandler1WasCompleted", true.ToString())
        };
        var result = new HandleResult(message, null, vars);
        return Task.FromResult(result);
    }
}
```

Пример Async Task + Event Handlers

Async Task Handler это, по сути, паттерн, реализуется комбинацией двух компонентов: Task Handler & Event Handler. Цель паттерна состоит в том, чтобы упростить кодовую базу для асинхронных операций типа «Запрос-Ответ».

Для реализации данного паттерна необходимо создать два класса, реализующих интерфейсы IAsyncTaskHandler и IAsyncEventHandler.

```
/// <summary>
/// Обработчик этапа с topicName = "TaskHandler2"
/// </summary>
public class AsyncTaskHandler2 : IAsyncTaskHandler<AsyncTaskHandler2CommandParams>
{
    private readonly SomeHttpClient _someHttpClient;

    public AsyncTaskHandler2(SomeHttpClient someHttpClient)
    {
        _someHttpClient = someHttpClient;
    }

    /// <summary>
    /// Создание локального контекста выполнения задачи (параметров команды)
    /// </summary>
    public Task<AsyncTaskHandler2CommandParams> MapIn(AsyncTaskHandleContext processContext)
    {
        var result = new AsyncTaskHandler2CommandParams()
        {
            SomeId = processContext.ExternalTask.Variables.GetVariableValue("SomeId"),
        };

        return Task.FromResult(result);
    }

    /// <summary>
    /// Валидация контекста выполнения задачи
    /// </summary>
    public Task<ValidateResult> Validate(AsyncTaskHandler2CommandParams context)
    {
        Guard.NotNull(context, "context != null");
        Guard.NotEmpty(context.SomeId, "context.SomeId != null");

        return Task.FromResult(ValidateResult.Success());
    }

    /// <summary>
    /// Код вызова запрос асинхронной операции
    /// </summary>
    public async Task<AsyncHandleResult> Handle(AsyncTaskHandler2CommandParams context,
AsyncTaskHandleContext processContext, CancellationToken cancellationToken)
    {
        // формируем ID асинхронной операции
        var asyncOperationId = Guid.NewGuid().ToString();

        // передаем ID асинхронной операции во внешнюю систему
        var request = new { someId = context.SomeId, asyncOperationId, };
        var response = await _someHttpClient.Post(request);

        var vars = new[]
        {
```

```
            ("AsyncTaskHandler2WasCompleted", true.ToString()),
        };

        var result = AsyncHandleResult.Success(asyncOperationId, request, response, vars);
        return result;
    }

    public Task MapOut(AsyncTaskHandler2CommandParams context, AsyncHandleResult handleResult,
CancellationToken cancellationToken)
    {
        return Task.CompletedTask;
    }
}

/// <summary>
/// Контекст выполнения задачи TaskHandler2 (параметры команды)
/// </summary>
public class AsyncTaskHandler2CommandParams
{
    public string SomeId { get; set; }
}
public class AsyncEventHandler2 : IAsyncEventHandler<AsyncEventHandler2Message>
{
    /// <summary>
    /// Поиск ID-а асинхронной операции
    /// </summary>
    public Task<AsyncCorrelateResult> Correlate(AsyncEventHandler2Message message,
AsyncEventHandleContext processContext)
    {
        var correlateResult = AsyncCorrelateResult.Success(message.AsyncOperationId);
        return Task.FromResult(correlateResult);
    }

    /// <summary>
    /// Обработка сообщения
    /// </summary>
    public Task<HandleResult> Handle(AsyncEventHandler2Message message, AsyncEventHandleContext
processContext)
    {
        var vars = new[]
        {
            ("AsyncEventHandler2WasCompleted", true.ToString())
        };
        var result = new HandleResult(message, null, vars);
        return Task.FromResult(result);
    }
}

public class AsyncEventHandler2Message
{
    /// <summary>
    /// ID-а асинхронной операции
    /// </summary>
    public string AsyncOperationId { get; set; }
}
```

Разработка схемы процесса и его загрузка в платформу

Схема разрабатывается в приложении Camunda Modeler (https://camunda.com/products/camunda-bpm/modeler/) в нотации BPMN.

Разработанная схема в формате ".bpmn" добавляется в папку "resources\bpmn" (либо в папку, указанную при подключении SDK в параметре DeploySchemesFolder, см. метод AddLiveBpm), устанавливается режим «Copy always on build».

Для корректного деплоя необходимо копирование BPMN-схем в output directory при билде проекта. Для этого необходимо выполнить глобальную настройку проекта для копирования какой-либо папки в output. Например, папки схем из директории resources\bpmn (+вложенные). Данная настройка является настройкой по умолчанию для проектов, созданных на основе шаблона .Net сервиса, либо по шаблону платформы.

```
<ItemGroup>
  <Content Include="resources\bpmn\**\*.bpmn">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </Content>
</ItemGroup>
```

Запуск процесса

Исполнение процесса всегда производится над какой-либо сущностью (заявка на кредит, заказ карты и т. д.). Для формализации требований к сущности создан интерфейс IDocument. Сущность должна обладать набором обязательных и опциональных атрибутов.

Старт процесса строится на базе двух параметров - описание сущности (документ, базовый документ и т. д.) и Tag-а схемы.

Интерфейс IDocument

Обязательные поля:

- DocumentId - уникальный ИД документа (для получения соответствия между процессом и документом).

Рекомендуемые поля к заполнению после старта процесса:

- ProcessId - ИД запущенного процесса.

Опциональные поля:

- BusinessKey - Ключ для данного процесса (инстанса) в BPM Engine (использование - по усмотрению разработчика). По умолчанию = DocumentId;
- DocumentType - Тип документа (любой);
- DocumentCode - Код/номер документа (любой);
- ProcessDefinitionId - ID конкретной версии схемы, по которой был запущен процесс (рекомендуемое поле к заполнению после старта процесса);
- ExternalSystem - ID системы-инициатора процесса (сайт и т.д.);
- ExternalProcessId - ID документа системы-инициатора процесса (ID заявки сайта и т.д.).

Сервис для старта процесса IStartProcessService

Метод StartProcess - старт процесса по данному документу и tag-у схемы

Параметры:

- document (IDocument, либо documentId + businessKey) - описание исходного документа
- schemeKey - tag-схемы, по которой нужно стартовать процесс (берется последняя версия)
- variables - дополнительные переменные для инициализации процесса (по умолчанию пусто)
- mediatorProviderType - тип BPM-engine (по умолчанию = Camunda)

Рекомендуется для создания endpoint-а для старта процесса.

Пример

Service

```
public class OrderService : IOrderService
{
   private readonly IOrderRepository _orderRepository;
   private readonly IStartProcessService _startProcessService;

   public OrderService(IOrderRepository orderRepository, IStartProcessService startProcessService)
   {
      _orderRepository = orderRepository;
      _startProcessService = startProcessService;
   }

   public async Task<StartProcessResponse> StartProcess(string schemeKey)
   {
```

```
        var order = await _orderRepository.Add(new OrderEntity()
        {
            Id = Guid.NewGuid().ToString(),
        });

        var document = new DocumentInfo(order.Id);
        var processInfo = await _startProcessService.StartProcess(document, schemeKey);

        order.ProcessId = processInfo.ProcessId;
        await _orderRepository.Update(order);
        return processInfo;
    }
}
```

Endpoint для старта процесса

```
[Route("api/[controller]")]
public class OrderController : ControllerBase
{
    private readonly IOrderService _orderService;

    public OrderController(IOrderService orderService)
    {
        _orderService = orderService;
    }

    [HttpPost("{schemeKey}/startProcess")]
    public Task<StartProcessResponse> StartProcess(string schemeKey)
    {
        return _orderService.StartProcess(schemeKey);
    }
}
```

При разворачивании приложения согласно инструкции (см. Разворачивание платформы) вместе с SDK разворачивается тестовое приложение, на котором можно протестировать работу SDK.

Для этого необходимо открыть swagger (см. раздел «Проверка запуска» в руководстве по установке), найти там метод api/Test/LiveBpm.TestHost.ProcessLogger/startProcess и по кнопке «Try it out» проверить запуск процесса. Убедиться, что процесс запустился можно с использованием приложения Cockpit (https://camunda.com/products/camunda-bpm/cockpit/).

Отслеживание событий от SDK

Для возможности дополнительной обработки инцидентов, ошибок и т.д. существуют механизмы подписки на соответствующие события от SDK. Подписка реализуется средствами библиотеки MediatR.

Список событий:

- IncidentEvent - событие об инциденте;
- MediatorTimeoutEvent - событие об TimeoutException на FetchAndLock (Camunda, are you okay?);
- MessageNotHandledEvent - событие об невозможности обработать Event;
- ProcessFinalizedEvent - событие о завершении процесса. Событие должно быть отправлено обработчиком финального этапа (т.е. событие не отправляется ядром автоматически);
- SupportErrorEvent - событие об ошибке для Support. Событие должно быть отправлено обработчиком этапа для Support (т.е. событие не отправляется ядром автоматически).