Charon Framework

Developer Documentation v1.0 by Wij Skinner

Contents

[Source Control 3](#_Toc461718215)

[Solution\Project Structure & Descriptions 3](#_Toc461718216)

[Core 3](#_Toc461718217)

[Charon.Core.Entities 3](#_Toc461718218)

[Charon.Core.Messaging 3](#_Toc461718219)

[Charon.Core.Services 4](#_Toc461718220)

[Charon.Core.Utilities 4](#_Toc461718221)

[Data 4](#_Toc461718222)

[Charon.Data 4](#_Toc461718223)

[Messaging 5](#_Toc461718224)

[Charon.Messaging 5](#_Toc461718225)

[Infrastructure 5](#_Toc461718226)

[Charon.Infrastructure.PersistenceService 5](#_Toc461718227)

[Charon.Infrastructure.PersistenceServicePackage 5](#_Toc461718228)

[Engines 6](#_Toc461718229)

[PersistenceEngine 6](#_Toc461718230)

[ProductEngine 6](#_Toc461718231)

[ExecutionManager 6](#_Toc461718232)

[Configuration Information 7](#_Toc461718233)

[Virtual Machines 7](#_Toc461718234)

[RabbitMQ 7](#_Toc461718235)

[Persistence Engine Service 8](#_Toc461718236)

[SQL Server 8](#_Toc461718237)

[Team City 9](#_Toc461718238)

[In Depth 9](#_Toc461718239)

[Queue Structure 9](#_Toc461718240)

[Adding Entity Types 9](#_Toc461718241)

[Adding Request/Response Types 9](#_Toc461718242)

[Tricky Bits 9](#_Toc461718243)

[BusManager 9](#_Toc461718244)

[QueueHandlerManager 9](#_Toc461718245)

[ListConverter 9](#_Toc461718246)

[UpdateConverter 9](#_Toc461718247)

[ExecutionManager 9](#_Toc461718248)

# Source Control

We are using a git repository for the framework and it is located on the Snugg’s codebase account:

<http://code.thesnugg.com/projects/charon-framework/repositories/charon-framework/tree/master>

Currently we are just using the master branch. Use the link below to clone the branch:

<https://snugg.codebasehq.com/charon-framework/charon-framework.git>

# Solution\Project Structure & Descriptions

The solution’s folder is structured as follows:

## Core

Contains core objects and interfaces shared across most of the rest of the solution. It comprises following projects:

### Charon.Core.Entities

Contains:

* Low level base entities, such as EntityBase, the base object all persisted objects inherit from
* RequestBase and ResponseBase which all request or response message types inherit from
* All POCOs which inherit from EntityBase and are therefore persistable via the persistence engine
* Various other shared interfaces and enumerations

### Charon.Core.Messaging

Contains:

* Messaging request and response types. This is where new request\response types will be added as they are created. Currently we just have requests and responses used by the persistence engine:
  + EntityChangeRequest
  + EntityListRequest
  + PagedEntityListRequest
  + EntitiesChangedResponse
  + EntityListResponse
  + PagedEntityListResponse
* Message bus definitions. These are used to define and setup queues\exchanges for the RabbitMQ instance.
  + TopicQueues – Queue definitions that use RabbitMQ topic base routing. See <https://www.rabbitmq.com/tutorials/tutorial-five-dotnet.html>
  + Exchanges – Exchange definitions. Currently just includes standard erp.topic exchange and erp.fanout.products.updated – a fanout queue used to notify interested nodes of product entity updates
  + Direct queues – Not currently used
* EntityTypes – These are key collections that define the set of types that can be used for a particular purpose:
  + EntityBaseTypes – This is the collection of types inheriting from EntityBase that the persistence engine can use. This collection is used in numerous places to setup message type subscriptions (for example). If adding a new entity type that we wish to persist it must be in this collection for the messaging subsystem to be aware of it.
  + TypesWithExpressions – This is a list of types that contain expressions, and is used by our custom JSON serialiser to work out which types will need the use of the Serialize.Linq library to transmit serialised expressions over the message bus.
  + EntityRequestTypes – This is a collection of request types used by the persistence engine to setup its request type/entity type subscriptions
  + EntityResponseTypes – Collection of all response types
  + EntityRequestAndResponseTypes – Concatenated collection of the above two collections

### Charon.Core.Services

Contains:

* Interface definitions for current and future services
  + Currently just contains IPersistenceService

### Charon.Core.Utilities

Contains:

* Various mainly static utility classes such as:
  + GenericHelpers – This is a set of methods used to identify and invoke run time types versions of generic methods
  + TypeSwitch – A utility class used when you want a switch equivalent based on type

## Data

Contains all data access classes used to communicate with a persistent database. We have chosen not to use the code first approach to entity framework development for a number of reasons. The database design was created first, and a tool was used to generate mappings and POCO classes. From that initial start, new entities are now created manually as it is a simple structure to copy.

### Charon.Data

Contains:

* CharonContext – The entity framework context that handles database access. Currently set to use a SQL server database located on staging.thesnugg.com called Pluto. Lazy loading and proxy creation are both disabled.
* DataAccess
  + The framework uses a generic entity framework data access layer. This provides the ability to add/remove and update database tables based on disconnected/unattached POCOs that inherit from the EntityBase class.
  + This works by analysing the State property of each object in an object graph, to decide the correct action and persist this to the database via the CharonContext.
* Mappings - The entities inheriting from EntityBase are mapped to the database context via the fluent API in the Mappings folder within the data project.
  + Each object that inherits from EntityBase must have its own mapping class
  + Navigation properties are represented as virtual, and both the ID and the navigation objects themselves are included on the POCO. The navigation property ID is used as a foreign key in the fluent API mapping classes. This allows us to set navigation properties on entities using just their IDs for clients using the persistence service/persistence engine.

## Messaging

Contains all components used to communicate via the RabbitMQ message bus. We are currently using the open source EasyNetQ messaging library which sits on top of the RabbitMQ .net components.

### Charon.Messaging

Contains:

* BusManager – This is the class used to connect to the message bus. It is used to subscribe to queues, publish messages to queue and to maintain queues and exchanges.
* QueueHandlerManager – EasyNetQ specifies that queue subscriptions are based on the type (as in C# type) of the message. The queue handler manager is way of keeping track of what handlers of which type are subscribed to which queues.
* EasyNetQCustomisation – EasyNetQ allows most of its internal components to be replaced via dependency injection. This allows us to customise various parts of it. We use this facility to provider a custom serialiser within which we can utilise the Serialize.Linq expression serialisation library.
* DTOs – The data architecture allows the retrieval and editing of data using linq expressions sent over the message bus. Because the resulting data queries can form large object graphs with circular references we use two classes here to prune our object graphs into just the required data. This is split into two main classes:
  + ListConverter – This prunes GetEntity object graphs so that we only send/receive objects with the specified navigation properties with no circular references.
  + UpdateConverter – This prunes UpdateEntity object graphs so that we only send/receive navigation properties that are in a modified state or are connected to modified objects beneath them in the object graph.

## Infrastructure

Contains services that are provided to other software (nodes) that require a connection to the message bus. Currently just contains the persistence service, but in future will contain the other services offered such as product service, order service etc.

### Charon.Infrastructure.PersistenceService

Contains:

* PersistenceService – This is the main entry point for the RED ERP system (or any other node that will require the use of the persistence engine), to communicate with the persistence service. It enables the retrieval, editing and deletion of data via the message bus. It implements the previously designed IPersistenceService class.

### Charon.Infrastructure.PersistenceServicePackage

Contains:

* Nuget package builder – To enable the persistence service to be added easily to other applications, in this project it is built into a nuget package referencing all its external dependencies (other nuget packages) and our own binaries required to use the persistence service.
  + The built nuget package (currently named Charon.Infrastructure.PersistenceService.1.0.0.nupkg) is at the moment then copied to a local nuget package source directory and can be accessed from there by connected applications. In future we aim to use our own nuget server for such purposes.

## Engines

The engines are the heart of the framework that do work requested by their appropriate service. Currently we have just one functional engine which is the persistence engine. It does, however provide a structured example for the production of future engines.

### PersistenceEngine

The persistence engine facilitates the persistence and retrieval of data from external nodes on the message bus, to the new framework’s one central data source (currently the Pluto sql server database). It achieves this with the use of synchronous and asynchronous multi-threading, allowing long running tasks to be assigned to their own thread for the lifetime of the operation.

It is a console application that can be installed as a windows service. This allows it to run in console mode (ie just double click the executable) for debugging purposes, so that all messages are echoed directly to the screen. In normal service mode operation, it uses the NLog framework to log to the central database.

Contains:

* EngineConfiguration – Configuration info the determines the location of the RabbitMQ instance, the async processing interval (how often the application checks to see what tasks are waiting to be executed and assigns them to threads), and the number of threads to be made available for asynchronous processing.
* EntityOperations – This class contains the operations that the persistence engine can execute as tasks. Currently there are only two operations. They are UpdateEntities, GetEntities and GetPagedEntities. They all use Charon.Data generic data access classes to get/set data to/from the database.
* RequestTasks – This contains the three classes that inherit from PersistenceRequestTaskBase and they are:
  + EntityChangeRequestTask – Task that calls the UpdateEntities operation
  + EntityListRequestTask – Task that calls the GetEntities operation above
  + PagedEntityListRequest – Task that calls the GetPagedEntities operation
* Window service components – The services folder in the project folder contains the entry point for the engine (Program.cs), the ServiceWrapper class that enables the project to run as a Windows service, and the ProjectInstaller class which is used to allow you to install the service using the .NET framework InstalllUtil.exe executables.

### ProductEngine

The product engine project structure is currently the same as the persistence engine – without any functionality within it. This will eventually provide the ability to maintain listings on the various marketplace APIs and provide product catalogue information.

### ExecutionManager

The execution manager is responsible for managing threads in the engines. It provides a structured way of managing a thread pool with a configurable number of threads for task execution that waits on a configurable timer to execute queued tasks asynchronously. It also enables synchronous operations to be immediately executed on their own thread.

# Configuration Information

## Virtual Machines

The framework currently uses one virtual machine. This machine is located on Microsoft azure and is managed through the azure portal - <https://portal.azure.com>

* Machine name: rabbit-hole.westeurope.cloudapp.azure.com
* IP address: 40.115.27.43
* Remote login\administrator user:
  + Username: rabbit
  + Password: Tastycarrots1

This machine hosts the RabbitMQ message bus instance, the Teamcity build system and the persistence engine windows service.

## RabbitMQ

We run RabbitMQ on a windows machine. To re-install/ or create a new instance, follow these steps:

1. Download and install the appropriate Windows 64/32-bit Erlang libs from here - <https://www.erlang.org/downloads>
2. Download and install the appropriate Windows 64/32-bit Rabbit MQ server installation from here - <https://www.rabbitmq.com/download.html>
3. Run the “RabbitMQ Command Prompt” start menu item.
4. Execute the following command to enable the web management portal and shovel plugins for RabbitMQ (without quotes) “rabbitmq-plugins enable rabbitmq\_shovel rabbitmq\_management”
5. The management portal will now be available on the machine you have installed it on default port 15672. For a local instance that will be the address <http://localhost:15672>
6. Login with the default username guest, password guest.
7. Setup required users for our implementation. Go to admin tab in management portal.
8. We will be setting up the following usernames and passwords which are required for the new frameworks current setup:
   1. admin – Tastyalfalfa1
   2. persistenceengine - Tastypersistence1
   3. productengine – Tastyproducts1
   4. red – Tastyred1
9. Click add a user, add username and password, then click add user
10. Once added, click the user in the table above to give access rights.
11. Leave default all host access and click set permission.
12. The admin user must have the tag “administrator” added so that it can programmatically manage exchanges and queues and it is the login we use to access the web control panel.
13. Once 12. has been completed the guest user can and should be removed.

And that’s it. If the persistence engine is configured to point to the newly created instance it will create all the exchanges and queues it needs on startup.

Further configuration of the RabbitMQ instance (for example ports used etc) is described on the RabbitMQ installation pages - <https://www.rabbitmq.com/configure.html#customise-windows-environment>

## Persistence Engine Service

* Location: rabbit-hole.westeurope.cloudapp.azure.com
* Install directory: C:\Program Files\Charon\PersistenceEngine

The persistence engine is currently deployed manually using the following steps:

1. Remote access to rabbit-hole.westeurope.cloudapp.azure.com
2. Run services.msc
3. Find “Charon Persistence Service” and stop it
4. Delete all files from C:\Program Files\Charon\PersistenceEngine
5. Switch to local machine
6. Build Persistence Engine on local machine
7. Copy all files from <repository root>\Engines\PersistenceEngine\bin\Release on local machine
8. Switch to remote machine rabbit-hole.westeurope.cloudapp.azure.com
9. Paste all files into C:\Program Files\Charon\PersistenceEngine
10. Start Charon Persistence Service

If the service needs to be setup on a machine where it has not been installed before then the windows service entry will not be present and it will need to be installed first. This is done via the following steps (assuming latest persistence engine files have been copied to the path listed above):

1. The service entry is created and removed using the .NET framework InstallUtil.exe executable. For .NET framework 4, this is located at: %Windows%\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe
2. To add the service entry for the Persistence Engine
   1. Open a command prompt with Administrative Privileges.
   2. Navigate to the folder containing the PersistenceEngine.exe binary
   3. Run the command InstallUtil.exe PersistenceEngine.exe
3. To uninstall the service:
   1. Open a command prompt with Administrative Privileges.
   2. Navigate to the folder containing the PersistenceEngine.exe binary
   3. Run the command InstallUtil.exe PersistenceEngine.exe /u

## Persistence Service Nuget Package

The persistence service is used by other nodes to communicate with the persistence engine over the RabbitMQ message bus. Currently the only application that is using it is the RED ERP solution.

The persistence service is automatically built into a nuget package within visual studio using a post-build event specified in the PersistenceServicePackage project properties. The build event calls the Nuget.exe package building executable located in the PersistenceServicePackage\nuget folder.

The build event calls the command:

nuget pack $(ProjectDir)nuget\Charon.Infrastructure.PersistenceService.nuspec

This uses the configuration defined in Charon.Infrastructure.PersistenceService.nuspec

to build a nuget package that can then be released to a nuget server, or as is currently done for the RED ERP system, added to a local repository, checked out and added as a local nuget source in Visual Studio’s package manager configuration.

For the RED ERP system, the local package source that must be added for the package to be restored correctly is at <repository root> \CharonSystem.Erp.Presentation.Mvc\nugetSource

## SQL Server

The persistence engine is currently configured to use a SQL server database. Recent backups of the data can be found in the CharonFramework repository in the <root>/Data/Backup folder in .bacpac format. These can be imported directly into a local SQL instance using the ”Import Data Tier Application” context menu command that is available when you right-click on the SQL instance’s Databases node in SQL server management studio.

Currently the persistence engine uses the staging.thesnugg.com SQL instance and the database is called Pluto.

The login used by the persistenceengine is:

* Username: persistenceengine
* Password: Tastysql1

# Other Areas

## Queue Structure

### Exchanges

Currently we are using two types of RabbitMQ exchange, these are the topic exchange (<https://www.rabbitmq.com/tutorials/tutorial-five-dotnet.html>) and the fanout exchange.

These two exchanges are created by the BusManager class (used by both persistence engine and persistence service) when it is instantiated if they are found to not exist:

* erp.topic (topic exchange)
* erp.fanout (fanout exchange)

### Queues

The following topic exchanges are created by the BusManager class when it is instantiated if they are found to not exist:

* Charon.ProductEntities.List – Queue that is subscribed to by the persistence engine that all EntityListRequest objects are sent to
* Charon.ProductEntities.Update – Queue that is subscribed to by the persistence engine that all EntityChangeRequest objects are sent to
* Charon.ProductEntities.\* - Monitoring queue that captures all list and update requests
* Charon.\*.Update – Monitoring queue that captures all updates
* # - Monitoring queue using the # wildcard that captures all messages published to erp.topic exchange

### Temporary Queues

Consumers of the message bus use temporary queues to get responses from the persistence engine sent back that are only relevant for themselves. Request types that allow a private response provide a PrivateResponseQueue which the persistence engine uses to send the response back. These queues will be deleted once the response has been received by the consumer. Temporary queues will have a guid identifier appended to the topic queue that the request has been sent to, and this guid is taken from the id property of the request message.

The process of temporary queue creation and deletion is handled in the PersistenceService class for the consumer. This does mean that if an application using the PersistenceService is killed without allowing its dispose method to fully complete, temporary queues can be left hanging around. A future enhancement to the framework when the node monitoring application will be built should enable some kind of clean up of these orphaned queues (working on the assumption that private queues that haven’t had any activity for even a short period of time can be disposed of). At the moment the only consumer node is the RED ERP system, which does generally shutdown gracefully allowing the dispose method to complete, therefore it is not a major issue at this time

Fanout queues are temporary queues declared on the fanout exchange by a consumer and will receive all messages sent to that exchange. We use this to allow consumer applications to monitor updates that may have been instantiated by a different consumer.

## Adding Entity Types

Adding a new entity to the persistence engine requires the following tasks to be completed:

## Adding Request/Response Types

## Tricky Bits

### BusManager

### QueueHandlerManager

### ListConverter

### UpdateConverter

### ExecutionManager