#### Realizing Modular Construction with NInject

NInject is integrated already with ASP.NET MVC / ASP.NET WebApi / WCF. Based on this integration, those frameworks use NInject to create controllers and service instances. NInject automatically injects then all dependencies of those instances using the root container.

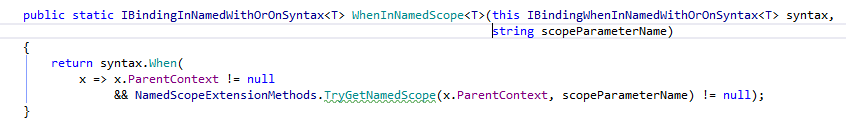
NInject organizes the definition of bindings into modules. NInject modules must be used to define the bindings for the business fields. Those modules should be loaded by reflection over the assemblies.

The NInject extension “Ninject.Extensions.NamedScope” allows to define child scopes for every business field, which allows to separate instances of different business fields (=make sure, that internal instances of one business field are not resolvable by another one).

The named scope (i.e. business field context) must be defined on the root instances needed for the request, e.g. the controller. To define this for a controller, the controller should be registered as in the following example for HomeController and business field “BusinessField1”:



To associate dependencies with this named scope, the following code is used:



By adding a call to WhenInNamedScope to any binding, this binding is only valid in the context of this scope, e.g.:



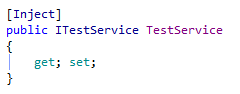
The lifetime of the instances is not affected by this, i.e. it’s possible to have all kind of lifetimes inside the business field. The condition just makes sure, that on resolution, only the bindings for the correct business field are considered.

Because the root instances (=entry points to the business field) define the scope and don’t have a scope constraint, it’s possible to resolve the entry points via their interface in different business fields. To make sure, that the context of the current resolver is not passed on to the other business fields, the interface of the other business field should be registered in the consumer like this:



This specifies, that the ITestService from business field 1 should be resolved context independent in business field 2.

To allow circular dependencies of those entry points, the injection for those entry points should be done via property injection and not via constructor injection, e.g.:



#### Using NHibernate inside the business fields

The data model of the business fields must be kept separate, i.e. business field 1 must not directly access the data model of business field 2. Instead, this must be done using services.

Rationale: The services of the corresponding business fields can make sure, that the business rules are met before an update is done on the data of this business field. Additionally, this allows to keep the data model of the business fields separate from each other (see also 8.1), which helps in realizing a modular Deployment.

To separate the data model, a separate session factory is used per business field. In NInject, the session factory of the business fields would be bound as singleton like this:

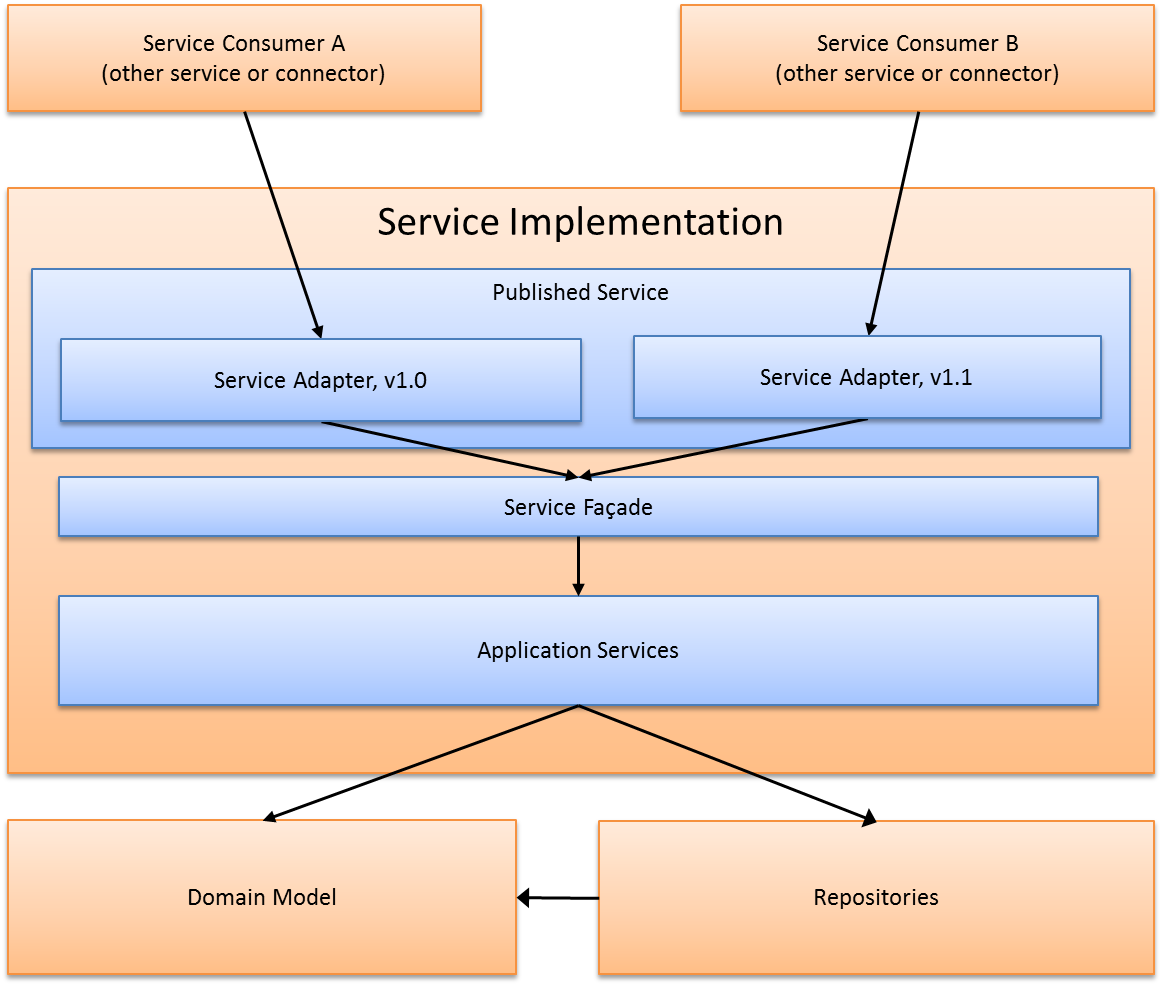


To inject this session factory, NInject allows to resolve constructor arguments explicitly:



Using this pattern, it’s always clear, which session factory is used and which part of the data model is accessed.

Note: This separation of data models also implies, that no foreign keys among the different data models are allowed.



The service adapter is responsible for mapping between DTOs of a specific version and the newest version. After mapping, it delegates the call to the real service implementation (the service façade).

The service façade implements the internal service interface. It realizes the service logic, which is implemented based on a set of internal application services and a domain model. The application services contain the processing logic of the requested service operation. They interact with other application services and/or the domain model to fulfill their task

Domain Models (see [PAEE], “Service Layer pattern”) are used to organize domain logic with classical OO design patterns. It helps to avoid code duplication between different application services dealing with the same domain concepts. It also increases the readability of the code, because the well-known business terms can also be found in the domain model.

##### Domain Model

The domain model is implemented using the PONO (plain old .NET objects) approach.

Objects in the domain model can be persisted to the database using NHibernate. The persistence logic is encapsulated by repositories to allow reusing persistence logic (e.g. querying for certain domain objects).

A domain object can have the following states in respect to the database:

* Transient  
  The object has never been persisted into the database or has been deleted from the database
* Persistent  
  The object is persistent in the database

Transient objects can be normally created by just calling the new operator. If the construction of a transient object is complex, this operation is also be done by the repository.

Note: It’s also sometimes useful to have domain objects, which are not persisted to the database, i.e. it’s not mandatory, that a domain object can be (directly) persisted to the database.

##### Repositories

Repositories are called by the application services to load/save/delete domain objects (see also [DDC], “Putting NHibernate into action”).

The repositories provide the following operations to load/save/delete domain objects:

* Add  
  This operation associates a root domain object with the NHibernate session. From this moment, the state of the object is tracked by NHibernate. Changes to the object are automatically persisted to the database on next Flush.
* Get  
  Different versions of get exists on the repository to load a single instance or a list of domain objects. The different get methods should have a specific name indicating what the purpose of this specific get is.

Delete  
This method deletes the provided domain objects and it’s children

Child domain objects can be just attached to root domain objects to persistent them. By default, the NHibernate mapping should define cascade dependencies, so that child objects are automatically saved/deleted too.

Note: The repository doesn’t contain a Save method to “Save” a persistent object. This is just not necessary, because the NHibernate session takes the responsibility for objects attached to the session (see also 4.1.4.2).

Hint: For implementation of batch functionality, the batch itself must handle session flush / detaching of objects from the session to achieve a good performance (see chapter 5.5.2).

##### Application-Services

As services don’t maintain session state between service consumer and provider, the application services don’t maintain a NHibernate session between different consumer/provider interactions. Because of this, the application services interact in the following way with the repositories/domain model in the update case (see also 6.2):

* Load the current domain model from the database by calling appropriate get methods on the repository. This associates the domain model object with the NHibernate session.
* Apply changes from the user to the current version of the domain model, performing a version check at the same time (to make sure, that the user changes have not been done on a now already stale entity)
* Persist only the changes to the database. By committing the NHibernate session, NHibernate automatically updates the database with all the changes and performs an additional version check with the current state of the database.

Rationales: Keeping session state inside the services would mean, that the domain model must be serialised into the distributed cache and must be transferred to all servers, which is not preferable.   
Because the SQL Server is setup as an always on availability group, writing would lead to an update of more than one SQL Server. Therefore, this pattern favours read access over write access.

Note: Session state can be held either in the browser application or in the service consumer (which can store it inside the distributed cache). Service consumers can be e.g. Web Apis or web services for external systems.

##### Service Contract guidelines

Serivce contracts should follow these guidelines:

* Service methods should provide coarse grained functionality.   
  Rationale: More round-trips with the server add additional latency and therefore impact user experience.
* Service methods have exactly one argument (a message) and return a message.  
  Rationale: It’s possible to easily adapt them without breaking changes for minor versions. Additionally, it’s easier to design coarse grained interfaces by focusing on messages.
* In case, the message could contain an unbounded list of entries, the service contract must support paging, i.e. the client must be able to just retrieve a portion of the result set.
* Service methods may be adapted with non-breaking changes. Breaking changes require to implement a new service published with a different url.

Non-Breaking changes are:

* Adding new methods
* Adapting messages with non-breaking changes, i.e. adding non-mandatory properties.

##### Publishing / Consuming Services

The business services are published using WCF. The WCF services are self-hosted (not hosted in IIS, because IIS doesn’t bring additional features in comparision to a simpler self-hosting solution).

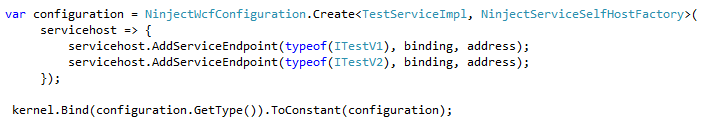
The service hosts are created by the NInject extension NInject.Extensions.WCF. The configuration of the endpoints is done in the code (because XML configuration become difficult to manage for many services and it’s difficult to ensure a consistent publication of the services). The bindings/behaviours are configured in the App Config, so that bindings and behaviours can be reconfigured without any code change.

The NinjectSelfHostBootstrapper is responsible for creating the NInject kernel and for instantiating, configuring and starting the WCF services:



All business services assemblies (e. g. Viacar.[Area1].\*.dll, see also 9.1.1) are discovered by the application container and loaded into the NInject kernel. The contained NInject modules define the instance bindings for each business field. Additionally, those NInject modules define, how the WCF services are published by the NInject WCF extension, i.e. the NInject WCF Hosts configurations should be organized by business fields and therefore be implemented inside the Business Services assemblies.

This is achieved by registering a NinjectWcfConfiguration for every WCF service in the NInject container. The NinjectSelfHostBootstrapper will, after creating the kernel, resolve all configurations registered in the kernel to define and start the WCF services. The following code will create and bind such a configuration:



Minor Versions of service interfaces are implemented in the same published WCF service class. Every minor version has its own interface, i.e. they don’t share operations but share operation arguments. This allows to easily consume a newer minor version (just change the service interface used). The following picture shows an example of a WCF service with two minor versions:

The PublishedWcfService class delegates all calls to the service façade. It implements all the service contracts of a major version (in the example: major version 1). To allow to implement the same interface method multiple times, it implements all interfaces explicitly. In the above example, the Operation1 was already present in version 1.0 of the interface and is therefore implemented once for version 1.0 and once for version 1.1 by the PublishedWcfService class.

In NInject, the published Wcf Service class is registered once with a self-binding as well as a binding to the implemented interfaces. The first registration is used, if the services is called from a out-of-process consumer, whereas the other registrations are used for in-process calls to the service.

Because service consumers and service providers are implemented both by the viacar system, the contract is shared by a common assembly (see also 9.1.1). Therefore, clients are created using the ChannelFactory .NET class. To simplify correct disposal, exception handling and caching of the channelfactory, a dynamic proxy is used. This dynamic proxy is generated by a NInject provider using the Linfu.DynamicProxy library. The following diagram shows this:

Such a provider is registered in the following way with NInject (for a service with a contract ITestV1):



Note: The configuration of the bindings/behaviours is done by application configuration (shared, included config file between consumer and provider named Binding.config and Behaviours.config), so that it can be adapted without a rebuild. Everything else is done by code.

#### UnitOfWork and Transactions

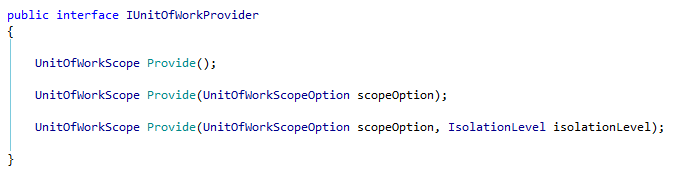
A UnitOfWork encapsulates the NHibernate-Session and defines the scope of a transaction. A transaction is automatically started, when a UnitOfWork is started.

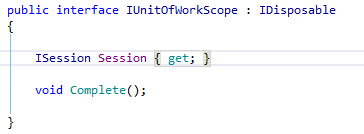
It’s possible to break out from a UnitOfWork and perform part of the logic in a different UnitOfWork, by explicitly requiring an indepedenant UnitOfWork.

The current UnitOfWork is provided to the application by a UnitOfWork Provider, which offers the following functionality:

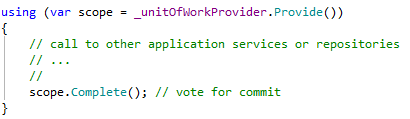
* Provide the current unit of work
* Provide a new unit of work with the default or specified transaction isolation level.

The UnitOfWorkProvider returns a new instance of IUnitOfWorkScope, which can be used to access the NHibernate session and to vote for commit/abort of the current transaction. The corresponding interfaces look like this:

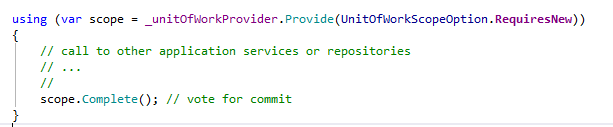




To participate in the current UnitOfWork, the application service uses then following code:



To request an independent UnitOfWork, the application services uses the following code



The default transaction isolation level used is ReadCommitted Snapshot.   
(to enable: alter database [DB\_NAME] set read\_committed\_snapshot on)

Rationale: The users of the Viacar system should not work a lot concurrently with the same data. Therefore, a higher isolation level is not needed. Conflicts should be managed by the application. However to prevent locking/deadlock issues with select statements, read committed snapshot is used instead of read committed with locks.

Note: A UnitOfWork can only span actions inside one business field. Cross-business field interactions take not part of the same transaction/UnitOfWork. Cross-business field interactions are described in the chapter 6.1.

#### Delivery Channels

This chapter provides some patterns to realize the StVA business application as well as the self-service application.

Concepts related to external interface for B2B interactions are described in the chapter 4.1.5.

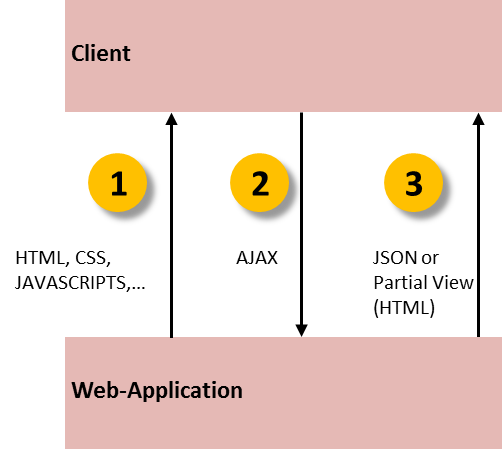
Those applications will be implemented using the Single Page web application pattern. This pattern allows to build a web application which behaves similar than a traditional smart client application by preventing a re-rendering of the whole UI in most cases. Because of this, the application doesn’t look like a set of independent pages and provides an improved usability over traditional web applications.

To support different types of devices, these applications will be built using responsive design (to a certain amount as needed by the different application kinds). This allows the application to adapt to the available space. For the self-service application, all types of devices should be supported, for the internal part, it’s enough, if the application adapts to a certain range of screen sizes (not all StVA use the same screens).

The web applications are implemented based on the ASP.NET MVC and the angularJS framework.

In the ASP.NET MVC framework, UIs are created from Views (together with Stylesheets, javascript, …), controllers and a view model. The controller is responsible for accepting requests from the browser, process them and return the needed information for the browser to update the UI. The controller can return only data or partial pages, thus allowing the browser to only refresh part of the UI.

A typical interaction between the browser and the web application looks like this:



In the first step, the browser requests the main application page (1). The ASP.NET controller renders the view and returns it to the browser. All interactions with this page are handled using asynchronous requests to the server (2), where the controller processes these requests and returns information back to update the user interface in the browser (3). Angular.js helps in interactions (2) and (3). It is responsible for:

* Communication with the Sever   
  By using promises, the asynchronous return of results can be handled in a much simpler way (including error handling and retries). Communication with the server is encapsulated in client side services, this allowing to reuse/organize code in a better way.
* Handling user actions (angular.js controllers)   
  Controllers organize the logic which reacts to actions of the user. They manipulate models and initiate communication with the server and handle the results.
* Data binding  
  As a result of an action inside a view, it’s many time necessary to update only some parts of the view. With databinding (1 way by default), Angular.js performs all the necessary updates to the DOM (no JavaScript DOM manipulation needed). If the application needs to send data to the server, the databinding (in this case, 2 way is needed) also helps in synchronizing data from view back to component.

Routing  
Angular.js can replace part of the application page by a different view. For this, the application changes, as a response to a user action, to a different route. Angular.js then requests this partial view from the ASP.NET MVC controller and replaces the current view by the new view.  
ASP.NET MVC is responsible for:

* Delivering main application as well as angular views  
  The ASP.NET framework delivers the views. They will be cached in the template cache of angularJS.
* Enable/Disable part of the UI according to user rights / tenant configuration  
  The views delivered by ASP.NET MVC only contain the parts, which are relevant for the user (see 10.9.4.2 as well as 10.6.2.1)
* Multi-Language support (see also 10.9.3)  
  ASP.NET MVC delivers the views in the correct language.  
  Rationale: Changing a language is normally performed by a profile configuration option and can lead to a reload of the whole application. There is no need to switch the language in the middle of the work.   
  Translations on the angular side cost also more performance, because it must be done by databinding and filtering every time the view is shown in contrast to once for ASP.NET MVC.

A more detailed description of the usage of ASP.NET MVC together with AngularJS can be found here: [MvCAJs].

##### Guidelines for AngularJS

To create a maintainable AngularJS application, the following guidelines must be followed:

* Angular code must be all written in TypeScript to enable compile-time-type-checking and to improve the readability/maintainability of the implementation.
* Coding guidelines to be used for developing with TypeScript:
  + <https://github.com/Platypi/style_typescript>:   
    The code is automatically checked before the compile step, if it’s respecting this guideline: https://github.com/Platypi/style\_typescript#tslint.
  + <https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines>  
    The coding guidelines from platypi are extended by the Microsoft typescript guidelines (the non-typescript project specific guidelines only). The checks for these additional guidelines are integrated in the checks for platypi.
* Angular JS components must not manipulate views, i.e. they must not know the structure of the view. Directives should be used, if the view must be modified by java script code.
* Javascript code must be valid to prevent issues with different browser types.
* Javascript files must be versioned, i.e. all javascript files must be referenced using a version number to allow caching on the client side (this ensures, that when a new version of the javascript is available on the server, the cached version is no longer valid).

##### Guidelines for CSS

To create a maintainable web application, the following guidelines must be followed:

* CSS files must be versioned, i.e. all css files must be referenced using a version number to allow caching on the client side.
* CSS must be used in a way to support the templating of the UI as described in chapter 10.9.4.2.3.
* CSS is not written directly, instead it is generated from SASS.

##### Session State

The session state of the web-application is kept inside the redis distributed cache and can therefore be accessed from all web servers (see 5.6).

State concerning the interaction with the user must be kept in the SPA on the client side. The session state contains only state related to performance optimization (like tokens, authorization information and profile information).

To ensure, that a zero-downtime deployment works, the session state of the current release must be usable by the next release and the session state of the next release must be usable by the current release. To ensure this:

* Keep session state minimal
* Items of basic types inside the session (e.g. string, int, guid, …) are safe to use for multiple versions and should be preferred (if it doesn’t affect readability and maintainability)
* When storing complex types, custom type converters help to convert from/to a string representation, which can take into account changes between versions (i.e. a new version doesn’t has an issue if a part in the serialized string is missing). Old versions of the type converter should preserve new serialized parts (don’t just throw them away).

More details about session serialization / deserialization can be found here:  
https://msdn.microsoft.com/en-us/library/aa479041.aspx

Sessions have an absolute lifetime (don’t allow to keep a session forever). For the StVA application, this lifetime is around 12 hours, to make sure, that the session doesn’t time out during a working day. After a session timeout, the application needs to be reloaded in the browser.

Additionally to the absolute timeout, the session has also an idle timeout. If the application is not used for some time, the session will timeout and require a reload of the application afterwards.

, the web applications will observe the session lifetime themselves. If the session timeout approaches, the application will inform the user and will afterwards redirect to the start page.

### Interface Hubs

An interface hub is a concept to handle all internal and external interfaces in a homogenous way. It is based on the following principles:

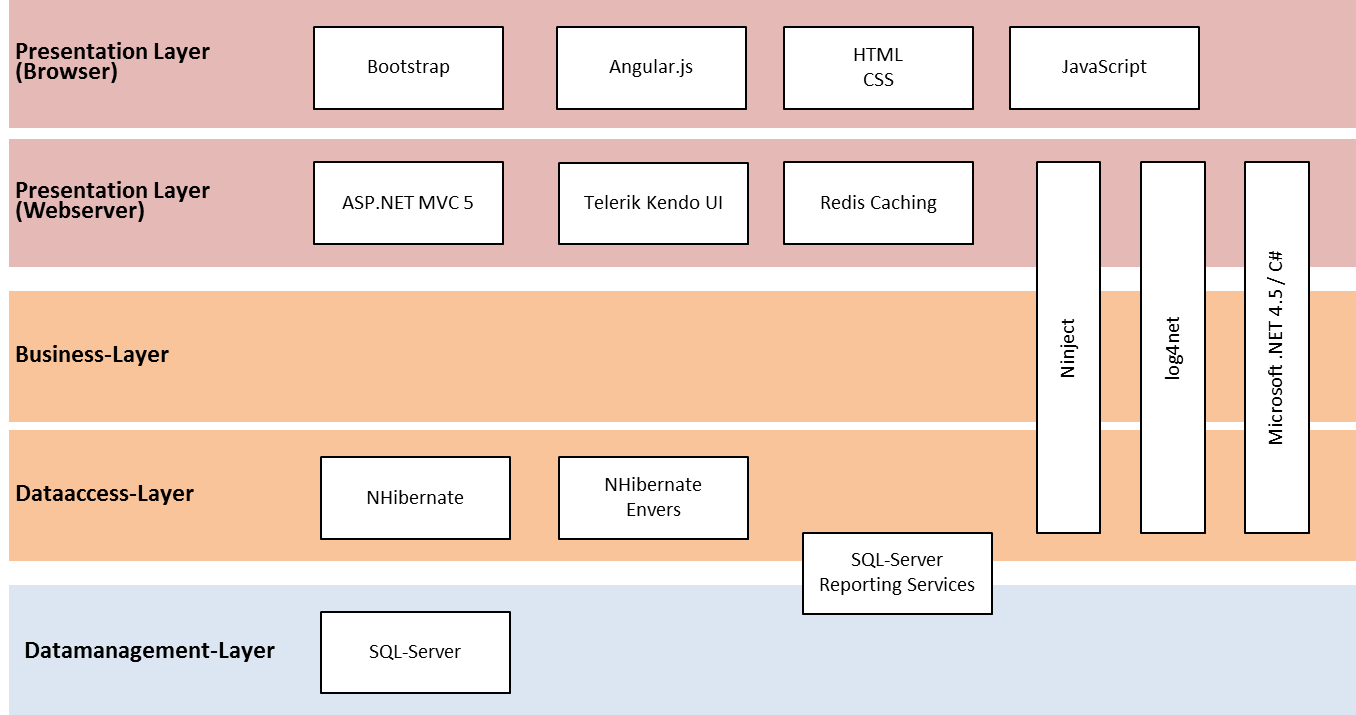
* All interfaces are versioned.  
  This ensures, that an evolution in an interface doesn’t break interface clients. The interface hub supports the new as well as the old version of the interface.
* Interactions with interfaces are generally implemented asynchronously  
  Using asynchronous communication with external systems ensures, that the interactions with the interface can be distributed over multiple nodes. It also ensures, that in case of an unavailability of an external system, that interactions with the external system can be delayed until the system is up again.   
  Additionally, the system can control the throughput to ensure, that external systems are not overwhelmed with interactions.
* Interactions should be reliable  
  If an interaction with another party is requested via an interface hub, the requestor does not need to track the status of the request. It can be sure, that the interaction will eventually happen.  
  Note: For this to work, the interface must provide idempotent operations. Otherwise, a manual intervention is needed in the error case.
* The Viacar system defines a canonical model  
  Depending on the tenant, the Viacar system has to interact with different tenant system for the same interface. Therefore, it uses internally a canonical model and maps it to the tenant model during the interaction with the external system. That is, the differences in external systems are hidden behind the canonical Viacar model.

The implementation of those principles is described in the following chapters.

Note: Interface hubs is a concept used for backend system integration. The interaction between Frontends and services are described in the chapter 4.1.4.1.

## Technologies, components and reuse

The following picture gives and overview of the used components for building the web application. The table afterwards describe those components in more detail.



In the picture above, the presentation layer is used to realize the delivery channels architecture layer, the business- and dataccess-layer is used to realize the business

| Technology / product / component | | Type / License | | Role in system / justification for choice | | |
| --- | --- | --- | --- | --- | --- | --- |
| SQL Server 2014 | | Commercial | | Database to permanently store business data of the application | | |
| NHibernate 4.0.4 GA | | LGPL | | OR Mapping library to access data stored in SQL Server 2014.  NHibernate is a very mature product, which simplifies the access to date stored in SQL Server from C#. | | |
| NHibernate.Envers 2.0 | | LGPL | | Powerful audit trail integrated with NHibernate. The application needs to keep track of modifications done by the users. It must be able to report, which user has done what change (see 2.7) | | |
| NHibernate Validator | | LGPL | | Powerful framework for implementing validation. It doesn’t support only validating domain objects persisted by NHibernate, but it’s a general purpose validation framework, which provides a deep integration with NHibernate.  The Viacar system needs to perform a lot of validations of user input, therefore, a framework with NHibernate Validator is needed to efficiently implement those validations. | | |
| NInject 3.2  Together with Ninject.Extensions.NamedScope, NInject.Web.Mvc,  NInject.Web.WebApi,  Ninject.Extensions.Interception,  Ninject.Extensions.  Interception.LinFu,  NInject.Extensions.Wcf, NInject.Extensions.Wcf.SelfHost  Ninject.Extensions.Logging | | Apache 2.0 | | NInject is one of the most used Dependency Injection frameworks in .NET. It is used to wire up the application at runtime. It improves the testability of the application by unit tests. | | |
| LinFu.DynamicProxy | | LGPL | | Used for interceptors with NInject as well as for generating dynamic proxies for WCF service consumers. | | |
| Log4Net | | Apache 2.0 | | Standard library for logging in .NET. To operate an application successfully, it must be able to log information about what happens in the application. | | |
| log4net.ElasticSearch | | MIT | | Library which contains a log4net appender for Elastic search. | | |
| AutoMapper | | MIT | | Library of convention-based object-object mapper.  This library reduces the amount of code needed to map between objects. It is e.g. useful to map internal dtos to external dtos (see e.g. 4.1.4.1) | | |
| Microsoft.NET 4.6 | | Commercial | | Current version of the .NET framework. Microsoft technology is required for this project (see 2.6) | | |
| ASP.NET MVC 5 | | Commercial | | Main Framework for building web applications with .NET using the MVC pattern. | | |
| Telerik Kendo UI | | Commercial | | Kendo UI provides advanced UI components to realize complex UIs. UX concept requires more complex components than bootstrap provides out of the box. | | |
| Redis Caching  (Windows Version from MS Open Tech)  Version 3.0.500 | | BSD | | To achieve the usability / performance quality goals, the application needs to have a very good response time.  Session state maintained in a distributed cache can improve the performance of session access in comparison to sql server session storage while maintaining high availability.  Reference data can be accessed much faster from a distributed cache than from the database. Therefore introducing such a cache can help to increase response time. | | |
| RedisSessionStateProvider version 2.0 (to support redis clusters) | | Microsoft Open Source (MIT) | | Standard Microsoft Redis session state provider, which uses StackExchange.Redis to access the Redis cache. | | |
| StackExchange.Redis | | MIT | | Redis Client for C#. This allows to access the redis cache from the applications. | | |
| CircuitBreaker.NET | | MIT | | Simple implementation of the circuit breaker pattern needed for external interfaces.  (see https://github.com/alexandrnikitin/CircuitBreaker.Net) | | |
| Bootstrap 3.3.5 | | MIT | | Bootstrap is one of the main frameworks used to build responsive web sites (it’s expected, that at least the public parts of the application are accessed using multiple device types).  Bootstrap also provides a lot of basic controls. It works also well with Kendo UI. | | |
| jQuery 2.1.4 | | MIT | | JavaScript library, which makes using JavaScript easier and helps with browser compatibility. | | |
| AngularJS 2.0.0-beta.0 | | MIT | | Angular JS helps with building complex UIs with web technology.  To achieve the high requirements on usability, the application user interface must use advanced javascript functionality.  AngularJS provides an MVC model for the javascript part to organise the javascript code in a maintainable/structured way. If focuses on declaring the user-interface behaviour instead of writing low level javascript code. This should lead to a less error prone and more efficient implantation process. | | |
| TypeScript 1.6 | | Apache | | TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.  Angular 2 fully supports and recommends to use TypeScript.  Typescript also brings refactoring support for javascript, which improves maintainability of the code.  Note: Typescript can be already used for angular 1.x to write controllers, services, … | | |
| Quartz.NET 2.3.3 | | Apache | | The system needs to process a lot of tasks asynchronously, e.g. for the communication with external systems.  This component supports to run those asynchronous tasks.  This component also supports a cluster deployment to fulfil availability requirements.  It also allows to fail-over jobs due to a shutdown of the scheduler, thus providing a high reliability.  Note: It also provides an integration with NInject by using NInject.Extensions.Quartz. | | |
| NUnit 2.6.4 | | NUnit specific (open source) | | NUnit is a standard framework to test .NET applications.  It is used to write unit/integration tests and run those tests. | | |
| NSubstitute | | BSD | | Mocking framework for testing classes with dependencies in isolation. This is required to create unit tests. | | |
| Selenium and Selenium Grid | | Apache | | Allows to test the UI of the application automatically. Selenium Grid provides a way to run those tests from within the CI build on a build server. It manages all the browser instances and prevents e.g. browser instances leaking. | | |
| SQL Server Reporting Services | | Commercial | | The StVAs need to report a lot of numbers concerning their activities. They also want to edit the report templates. SSRS provides an efficient way to solve those requirements. | | |
| RoundhouseE | | Apache | | This datamirgration engine is used to upgrade the database from the software factory. | | |
| Id | Description of change | | Rationale, impact | | Validated by/when |
| C1 | AppFabric Caching is deprecated now. It is replaced by Redis Caching (also in Azure) | | The project should not use a no longer maintained product.  To allow running the application in the azure cloud, the application should use a cloud compatible technology.  The impact should be low, because Redis provides equivalent features to AppFabric Caching. | |  |
| C2 | Kendo UI required | | Using Bootstrap only for UI components is not enough. For more advanced user interfaces, Kendo UI should be used. | | UX Designer, 03.11.2015 |
| C3 | SQL Server Always On is not used. Instead, the high availability feature provided by the hoster are used | | Always On requires much more SQL Server licenses / more virtual machines than the solution offered by the provider by default, therefore based on cost calculation, the Always On features is no longer used for ensuring high availability | | Thomas Vogt |
| C4 | Online Help | | Additonal requirements have been discussed. The discussion is still ongoing. | |  |

## Distributed Cache

The distributed cache caches reference data and session data (i.e. the ASP.NET session). Reference data is data, which is more or less static and is only edited by administrators using configuration screens (e.g. the list of cantons, the possible colours for vehicles, …) or in some cases imported from external systems.

For ASP.NET session data, the session provider is responsible for loading/storing session data in the cache.

Reference data is stored in the cache on first access and is periodically refreshed. If reference data is edited using a configuration screen, the corresponding cache entry is also refreshed. The caching of reference data is performed by the business service. For the web application, it is transparent, if the cache data is retrieved from the cache of from the database.

In certain cases, it could be interesting to store also some business related data in the cache (as performance optimization). The principle used is the same as for reference data, with the exception, that business related data should expire and therefore will be removed from the cache after some amount of time (no refresh).

### Technology / architecture

The distributed cache is powered by the Redis cache product. To store/load the session state from the redis cache, the standard Microsoft RedisSessionStateProvider is used. This session state provider will use internally the StackExchange redis cache client. For caching reference data, the application also uses the StackExchange redis client. The following diagram shows this:

The redis cache deployment is described in chapter 7.2.1.1.

### Security

Access to redis server is restricted by using password authentication (defined in the redis service configuration). The communication with the redis cache is encrypted using TLS (which prevents to send the password as clear text over the wire).

Note: The session state configuration section in the web.config file must be encrypted to protect the password in the DMZ for portal web applications.

### Security

Access to elastic search is restricted by using the Shield extension (<https://www.elastic.co/products/shield>). This allows to encrypt the communication with elastic search as well as to restrict access to indexes to only a certain set of users.

The user authentication for end-users is integrated with the WAF by writing a custom authentication plugin (<https://www.elastic.co/guide/en/shield/current/custom-realms.html>). Services are authenticated by authenticating the STS tokens from Viacar also using a custom authentication plugin.

To ensure, that a user of one tenant can’t access data of a different tenant, document level security is used to apply a filter for tenant to all indices accessed by a user (<https://www.elastic.co/guide/en/shield/current/setting-up-field-and-document-level-security.html#document-level-security>).

#### Provided Interfaces

Elastic search provides a REST API for inserting/searching log data:

* <https://www.elastic.co/guide/en/elasticsearch/reference/current/docs.html>
* https://www.elastic.co/guide/en/elasticsearch/reference/current/search.html

## Service call from presentation layer

The following sequence diagram shows an exemplary interaction from the web browser up to the database and describes the dynamic aspects of the patterns introduced in chapter 4.1.4:

The first interaction from the browser shows, how a view template is loaded from the MVC controller. For this, normally no interaction with the WCF services is required.

The second interaction from the browser shows how a javascript AJAX call is handled by the WebApi. The WebApi forwards the call to the WCF service responsible for searching customers using a service proxy. The Published Web service forwards the call to the service façade. The façade uses one or more application service to implement its functionality, in this case it uses a customer application service for interacting with customers. This application service uses a repository for customers and retrieves by this a domain object representing the found customer from the database.

|  |  |  |
| --- | --- | --- |
| **Directories in viacar/trunk/src** | **Folder content** | |
| App | | Contains code not related to a business field |
| |--Viacar.WebApp | | Contains the application container for the StVA application |
| |--|--Areas | | Parent folder for the web parts of the business fields. |
| |--|--|--Viacar.<Field1>.Web | | Contains the web project for <field1> |
| … | |  |
| |--|--|--Viacar.<FieldN>.Web | | Contains the web project for <fieldN> |
| |--|--App\_Data, App\_Start | | Standard ASP.NET MVC project folders. |
| |--|--fonts | | Contains all the common glyphs of the application. |
| |--|--Content | | Contains all the common stylesheets, images, … of the application. The organization of static content is described in more detail in 9.1.1.3 |
| |--|--Scripts | | Contains all the common javascript part of the application. The organization of javascript code is described in more detail in in 9.1.1.3 |
| |--|--Views | | Contains all the ASP.NET MVC views for the application frame. The organization of web projects is described in more detail in in 9.1.1.3. |
| |--|--Controllers | | Contains all the ASP.NET MVC controllers for the application frame. The organization of web projects is described in more detail in in 9.1.1.3. |
| |--|--Models | | Contains all the ASP.NET MVC Models for the application frame. The organization of web projects is described in more detail in in 9.1.1.3. |
| |--Viacar.<Portal>App | | Contains the application container for a Portal application (<Portal> is the placeholder for a specifc portal like Infocar). It is organized internally equivalent to the Viacar.WebApp. |
| |--Viacar.Services | | Contains the application container for the services. |
| |--Viacar.Sts | | Contains the STS used for creating tokens (see 10.6.1). |
| |--Viacar.JobRunner | | Contains the application container for the job runner |
| |--Viacar.ExternalInterfaces | | Contains the container for external interfaces |
| <Field1> | | Contains the projects for the specific business field <field1> except the web part. |
| |--Viacar.<Field1>.Contracts | | Contains the Viacar.<Field>.Contracts project |
| |--Viacar.<Field1>.Domain | | Contains the Viacar.<Field>.Domain project |
| |--Viacar.<Field1>.Persistence | | Contains the Viacar.<Field>.Persistence project |
| |--Viacar.<Field1>.Services | | Contains the Viacar.<Field>.Services project |
| |--Test | | The folder for test projects |
| |--|--Viacar.<Field1>.UnitTests | | Contains the Viacar.<Field>.UnitTests project |
| |--|--Viacar.<Field1>.Integration-Tests | | Contains the Viacar.<Field>.IntegrationTests project |
| |--|--Viacar.<Field1>.RegressionTests | | Contains the Viacar.<Field>.RegressionTests project |
| … | |  |
| <FieldN> | | Contains the assemblies for the specific business field <fieldN> except the web part. |
| Misc | | Contains various files used by the source code in some way, e.g. common configurations as well as common source files linked into every project (e.g. SharedAssemblyInfo.cs) |

For the UI Layer, the validation rules are generated based on the NHibernate Validation annotations placed on either the service layer entities (if equivalent to the Web API) or based on the specific Web API validation rules (if not equivalent).

This rule generation is performed in the following way:

* The HtmlHelper is extended by custom methods to generate all the UI controls, e.g. NgEditorFor.   
  Those methods define the binding between the UI element and the WebAPI model property by lambda expression. For this, the viewmodel must be either set to the WebAPI model class or the viewmodel must contain a property of the type of the WebAPI model class.  
  Note: The model will not contain the real content which is rendered on the client side in the end. The content is provided by a call to the WebAPI.
* A custom ModelMetaDataProvider (see <http://www.dotnetcurry.com/aspnet-mvc/715/model-metadata-provider-aspnet-mvc>) will investigate the NHibernate Validator attributes on the Web API model and will fill into ModelMetaData.[AdditionalValues](https://msdn.microsoft.com/en-us/library/system.web.mvc.modelmetadata.additionalvalues(v=vs.118).aspx#P:System.Web.Mvc.ModelMetadata.AdditionalValues) everything needed to render the validation directives for angular in the HtmlHelper.
* The angular directives will ensure, that angular performs the same validation as the WebAPI. The validation will be performed when the user leaves the field.
* For every NHibernateValidator attribute, directive custom Angular validator (if not provided by default) must be implemented which implements the validation logic of such an attribute in typescript.

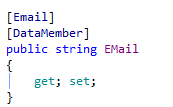
Field validation rules spanning multiple UI fields are specifically implemented in a first step (it’s not expected to have a lot of these).

### Validations

Syntactic validations (single field validation) are implemented using the NHibernate Validator framework. They are performed at least on the service interface level, to check, that the clients of the shared business services only call the services with valid input (this is e.g. needed for protection against XSS attacks as defined in 10.5.5.2).

Note: NHibernate Validator is a general purpose validation framework, so validations of e.g. values read from configuration files can be done using this framework too.

NHibernate Validator provides multiple ways to define validations (config file, attributes or fluent API). Because validation can be seen as part of the service contract, the attribute approach is used to define the validation rules directly on the DTO classes. This looks e.g. like in the following code snippet:



In the above code, a NHibernate Validator constraint is used to define, that the property must contain a valid Email address. To implement RegEx validations, the attribute “Pattern” is provided by NHibernate Validator. NHibernate Validator provides a lot of additional validators out of the box. If no suitable validator is available, custom validators can also be written by implementing an own validation attribute (which implements the IValidator as well as IRuleArgs interface). To support validation of child objects, the property containing the child / List of children must be decorated with the “Valid” attribute.  
Validations concerning multiple properties together must be implemented as custom validation rule on class level.

Validations with NHibernate Validator are performed using the ValidatorEngine. This ValidatorEngine must have a singleton scope, because it internally caches all the reflection information of the entities to validate, i.e. it must be bound to a constant in the NInject kernel (see also here: <http://fabiomaulo.blogspot.ch/2009/02/diving-in-nhibernatevalidator.html>):  
  


The validation is performed in a generic way by implementing a WCF IParameterInspector, which validates all WCF service calls. For this to work, the classes implementing the messages must be correctly decorated with the “Valid” attribute, so that NHibernate Validator recursively investigates the whole object graph.

Note: Business Rules are not validated by NHibernate Validator, but they are implemented as described in the chapter 10.2.5.

The chapter 10.1.8 describes, how those service level validations are performed on the client side too.

#### Integrating Log4Net

Logs are written using Log4Net. This chapter defines the guidelines, which apply to logging with Log4net:

* Logging should use specific categories (i.e. logger names) to identify the location (e.g. the class), where the log is produced. For logs, which are created by interceptors, the logger name should be derived from the real call performed and not from the interceptor class.
* Appenders should use the following layout:  
  %date [%-6.6thread] %-6.6level %-70.70logger (%property{Correlationid})- (%property{TenantID})| %message%newline%exception
* Logging should use the correct log level as defined at the beginning of the chapter.
* Logs should contain contextual information as defined above.   
  Those are passed to log4net by using specific properties (ThreadContext.Properties).  
  Context information is passed in message headers over process bounderies.
* Where appropriate (e.g. for request logging) a WCF, WebApi or NInject interceptor should be used to produce the required logs
* Expensive log statements should first check, if the corresponding log level is enabled before producing a log (e.g. when performing message logging in wcf calls).

Log4Net is integrated by using the Ninject.Extensions.Logging extension. This allows to unit test, that certain logging messages are created (especially important for logs concerning errors).

To send logs to elastic search, the elastic search appender is used. No detour via file is used for this. For the case, that elastic search is not reachable despite the high availability setup of it, a rolling file logger is used to have logs in emergency case (in a protected location).  
Rationale: The elastic search appender is asynchronous, so it doesn’t affect the response time negatively. It also batches multiple log events to ensure an efficient interaction with elastic search.

##### Changing Log Level at runtime

To support debugging production issues, log levels should be changeable without requiring a restart of the log producer applications, because this could remedy the error condition and thus preventing the possibility to generate more detailed logs.

To allow changing the log level of log4net at runtime, the application uses a customized log4net LevelEvaluator (see http://svn.apache.org/viewvc/logging/log4net/trunk/src/Core/LevelEvaluator.cs?revision=1707180&view=markup). This evaluator periodically loads the configured log level from the database. LevelEvalutors work on the appender level and therefore restrict the maximally logged level. The levels of the logger hierarchy itself can’t be changed by this.