# Simple Twitter

Contents

[Simple Twitter 1](#_Toc416474790)

[Brief design notes 2](#_Toc416474791)

[UI (frontend): 2](#_Toc416474792)

[Middle Layer 2](#_Toc416474793)

[Back-end 3](#_Toc416474794)

[Quering data 3](#_Toc416474795)

[Further development: 3](#_Toc416474796)

[Running application 4](#_Toc416474797)

[Senior Web Server Engineer Test 6](#_Toc416474798)

[Important Note 6](#_Toc416474799)

[You are not expected to write a full blown scalable web application. What we are looking for is a solid architecture, clean and elegant code. 6](#_Toc416474800)

[General Guidelines 6](#_Toc416474801)

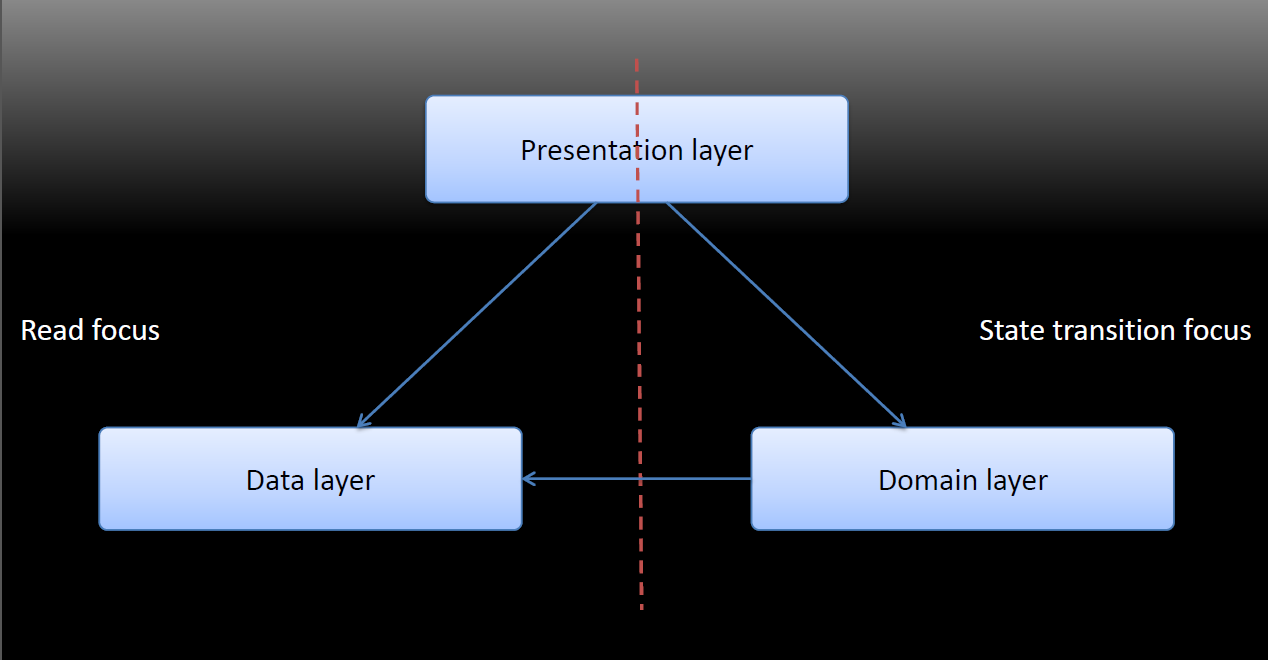
[The Test Assignment 6](#_Toc416474802)

# Brief design notes

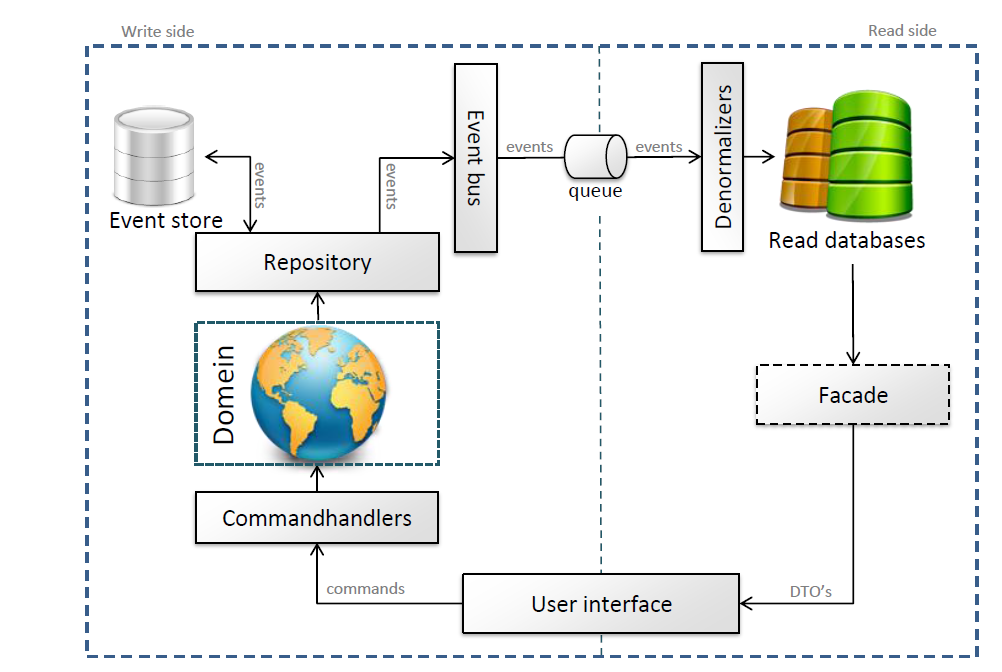
For inspiration was used the Twitter architectural review (by Matthijs Neppelenbroek) <http://www.timdeboer.eu/paper_publishing/Twitter_An_Architectural_Review.pdf>

To achieve the main goal - messaging architecture was selected based on NServiceBus combined with CQRS approach *(without Event sourcing, as the ES is not needed in this app)*.

Design concept:



CQRS architecture (by Microsoft):



### UI (frontend):

ASP.NET MVC + Angular JS, uses RESTfull web api *(see SimpleTwitter.API project).*

### Middle Layer

The middle layer is basically used as a queueing system to not overload the back-end service layer. With CQRS pattern we separate queries from state transition commands. We can quickly query read data stored in Redis and when inserting new data or changing the existing data we just create and put command into the queue. CommandServiceBus uses NServiceBus (with underliying MSMQ) sends state transition commands to a Write Side, which handles the commands and when it is done publishes an events for a ReadSide to update ReadModels.

We can create as many write side and read side nodes as we will need to handle all the user requests. NServiceBus also allows creating command handler and subscribe it on a specific commands, thus we can add search node, separate RDBMS on several databases etc.

### Back-end

The back-end service layer of SimpleTwitter stores all the users and tweets. It is used 2 back-end storages – RDBMS for write side and a key-value storage for read side models.

WriteSide persists data in memory using Fake DbSet, which can be easily replaced with EF implementation (MSSQL Server or MYSQL).

Read Side persists data in Redis.

### Quering data

WebApi web application communicates with Read database (read models) through ReadModelFacade which provides a read access to the data stored in Redis.

Read models are de-normalized and optimized for queries, for example we will query list of followers and followings, so we have 2 read models to quickly access followers and followings by a key (userid), instead of joining tables in RDBMS storage.

*Note in commercial application, console applications better to convert into IIS hosted web service (WCF app) and windows services (write side and read side apps)*

**Patterns used**: CQRS pattern without Event sourcing, Repository + GenericRepository, Unit of Work.

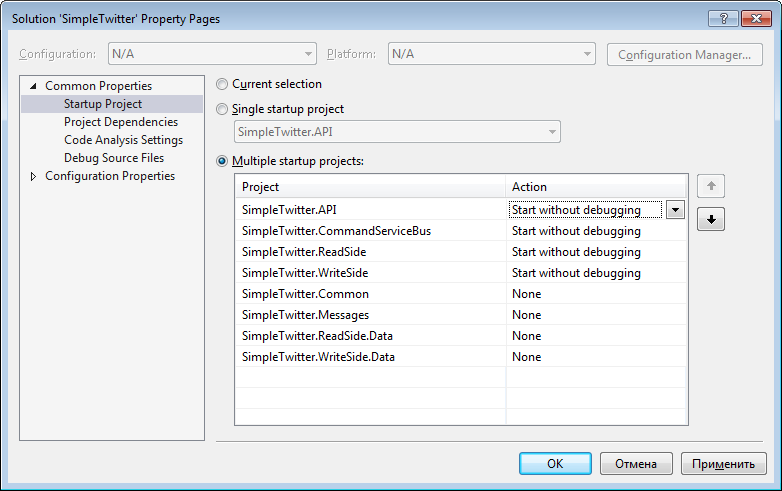
Technologies used: NServiceBus (sender-> publisher -> subscriber), Redis (with StackExchange client), WCF, WebAPI

### Further development:

Read models could be separated on recent data stored in on logical redis database (limited set contains last 100 posts(tweets) of each user) and the archived data stored in another logical redis db, b/c users will not list more than 10 pages of tweets (10 tweets per page) so the scan can be faster if the set will be limited. Read side should be able to restore state communi WriteSide.

# Running application

1. **IDE – Visual Studio 2013**
2. **When open the solution use NuGet Manager to Restore Packages**
3. **Ensure that multiple start up projects are configured (or run console apps manually):**



1. **Install and run Redis server. I used Redis 2.8.19, with a default configuration:**

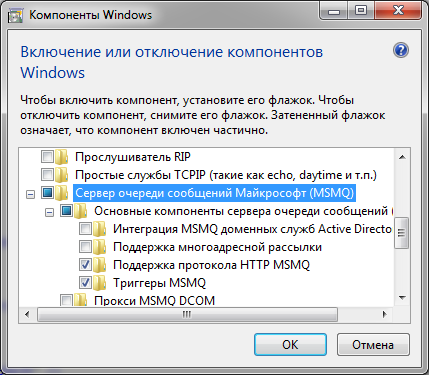
<appSettings>

<add key="RedisConfig" value="localhost:6379,connectTimeout=5000" />

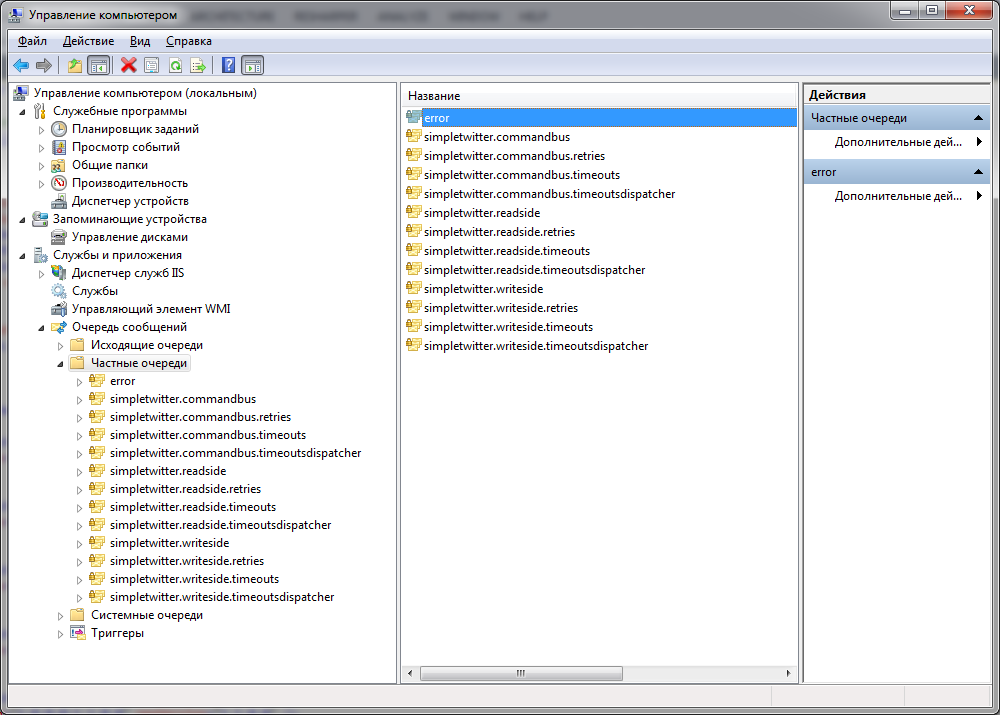
<add key="RedisDatabaseNumber" value="0" />

</appSettings>

1. **Ensure that MSMQ windows component is installed:**



**After running 3 console applications you should have the following queues:**



1. **Run steps order:**

* **Run Redis server**
* **Run WCFCommandBusService app**
* **Run WriteSide app**
* **Run ReadSide app**
* **Run WebAPI web app *(at first run it seeds an initial data sending commands to WCFCommandBus)***

# Senior Web Server Engineer Test

## Important Note

## You are not expected to write a full blown scalable web application. What we are looking for is a solid architecture, clean and elegant code.

**You should see it as a Proof Of Concept for your architecture proposal.**

## General Guidelines

1. Please provide
   * working code, ready for production
   * a brief design document explaining what you did
   * a link to a server where your code can be tested.

1. Feel free to search the Web for inspiration, but please do not copy code from anywhere, keep it 100% yours.
2. Use C# as your language of implementation and Asp.Net MVC 4.\* or 5.\*
3. Please remember that the design should be ready for heavy traffic web application, with many users and interactions. The design need to be ready for this situation.

## The Test Assignment

You are to design and implement the API and server side modules for a simplified

twitter-like social updates site:

* The application contains users
* Each user can post short text messages (140 chars)
* Each user can follow other users, and get a feed of their latest updates.
* Each user can also get a global feed for all the users.

Implement an HTTP based (ReSTful in the loose sense of the term) API that exposes

the following calls (no need for authentication, choose the format you like):

* CreateUser [UserName]
* PostMessage [UserId, MessageText]
* Follow [FollowingUser, FollowedUser]
* Unfollow [FollwingUser, UnfollowedUser]
* GetFeed [ForUserId]
* GetGlobalFeed

Please bear in mind the scalability of this system and try to find a good solution for a large scale application.