Virtual Scrum Board

# Abstract

This document describes what Virtual Scrum Board is all about, and provides software architecture for implementing it.

# Overview

A Scrum Board is a tool that helps Teams make Sprint Backlog items visible. The board can take many physical and virtual forms but it performs the same function regardless of how it looks. The board is updated by the Team and shows all items that need to be completed for the current Sprint.

In this exercise, we will create a Virtual Scrum Board, which can be shared and operated by all team members, remotely and concurrently (multiple clients working against the service same time, so changes in one can be seen by others).

## Virtual Board

The physical scrum board is a big canvas hang on a wall or painted, where team members place or draw tasks as small notes. These notes are usually grouped by task status, in columns for example: Todo, Doing, Done. Moving a note from one group to another is changing the task status.

The virtual board can be thought of a virtual place where Tasks are the data kept by the board in the database.

The virtual board is accessed through a remote service and visualized by the client application.

## Task

A task represents a single, small mission a developer, DebOp, etc. should complete within the sprint time period.

A task has the following attributes:

**Status** – a single task can be in a single status out of common three states: *Todo*, *Doing*, *Done*. Starting a sprint, a task is in a Todo status. When someone starts working on it, he/she move it to Doing, then finally Done when fully completed.

**Assigned To** – the team member this task is assigned to.

**Priority** – this number in range 1-5 represents the priority of a task among others to reflect its importance comparing to other tasks in sprint. Higher number represents higher importance.

**Remaining Work** – A number represents the work remains on this task until fully completed. This can be in days, hours, etc.

**Comments** – a list of comments added to this task by the team members. This can also be used for a discussion.

# Architecture

In general, we will have a single remote service that implements the Scrum Board logic and Data.

The board state will be persisted in a MongoDB server, and the operations will be exposed by the service as RESTful APIs using ASP.NET Core 2.0 on top of .NET Core 2.0.

The client application will be implemented as Windows Desktop application, based on WPF stack, on top of the latest Full .NET Framework, leveraging C# 7.x language.

We will use DI (dependency injection), in both Service and Client sides, and Rx on client side to handle service data changes.

For simplicity, changes to data will flow from the service to the clients using Polling instead of Pushing. Meaning, each client will have well defined timer that polls the data repeatedly on a time basis. For example, each five second or less.

## Data Entities

The following is a list of entities describe the data structure of the Virtual Scrum Board. This data will be persisted in MongoDB instance collections.

### User

Represents a user registered in the system, such as a Team Member, Admin, etc. For simplicity, we won’t have roles or permissions. All users have same role and permission to do anything. In a real system, we may want to provide users roles so for example, only project manager can add new Task Group, or update Project details or create/delete boards, etc.

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Description |
| Id | ObjectId (mongo db type) | Unique ID of the user, auto generated by mongo db |
| CreationTimeUtc | DateTime | The date time this object created |
| Name | String | A unique name of the user |
| Secret | String | Salted password of the user |
| Image | Byte[] | Binary serialization of user avatar image 32x32, PNG format. |

### Board

Represents a scrum board details. Each board contains project details and team members assigned to this board.

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Description |
| Id | ObjectId (mongo db type) | Unique ID of the board, auto generated by mongo db |
| CreationTimeUtc | DateTime | The date time this object created |
| Version | Integer | Number represents change version |
|  |  |  |
| Name | String | The name of the board |
| CreatedBy | String | The name of the user who created this board |
| Description | String | A short description of this board, can be used as a title |
| TeamMembers | String[] | The name of the users who are assigned to this board |

### Task

Represents a task assigned to a single board. A task contains details, status and other relevant data representing work estimation and whom this task assigned to.

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Description |
| Id | ObjectId (mongo db type) | Unique ID of the task, auto generated by mongo db |
| CreationTimeUtc | DateTime | The date time this object created |
| Version | Integer | Number represents change version |
| Description | String | A short description of this task, can be used as the task’s title |
| CreatedBy | String | The name of the user who created this task |
| AssignedTo | String | The name of the user who is assigned to this task |
| Status | TaskStatus | An enum represents the task status: Todo = 10, Doing = 20, Done = 30. |
| Priority | Integer | Represents the priority of the task |
| RemainingWork | Integer | Represents the remaining work |
| Comments | Comment[] | A collection of comment IDs commented on this task |

### Comment

Represents a comment attached with a task.

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Description |
| Id | ObjectId (mongo db type) | Unique ID of the comment, auto generated by mongo db |
| CreationTimeUtc | DateTime | The date time this object created |
| Content | String | A text content of this comment |
| CommentedBy | String | The name of the user who commented this |

## Accessors

To abstract the access to the underlying resource repository (DB, File, Registry, etc.), and to decouple changes to it from the upper layer, we may want to create an accessor, that actually uses the underlying resource repository APIs.

**In this exercise, we will skip this layer for simplicity and time considerations**. **We will access the MongoDB driver directly from the managers.**

## Controllers and Managers

We will have a single remote service which has the following controllers:

### User Controller

Provides user registration, authentication and user details.

**Register user** – register a new user.

**Authenticate user** – authenticate user and provides token back.

### Board Controller

Represents the virtual board. This controller provides APIs for managing boards (for example, create boards, query board details, etc.).

The following are the APIs should be provided by the board controller:

**Create board** – creates a new board and returns its unique ID generated by the DB.

**Delete board** – deletes a board given its unique ID.

**Get all boards** – returns description of all boards. This not includes the tasks data.

**Get changes** – return all changes given array of pair: board id, version.

**Update board details** – update board details, such as the board name and description.

**Get all tasks in board** – returns all the tasks belongs to this board.

### Task Controller

Represents the board tasks. This controller provides APIs for creating, deleting, querying and updating tasks.

**Create task** – creates a new task instance given its board ID and other details, such as Assigned To, Remaining Time, etc. A new task is auto set to ‘Todo’ status. This returns the task’s unique ID created by the DB.

**Get changes** – return all changes given array of pair: task id, version.

**Delete task** – deletes an existing task given its unique ID.

**Update task field** – updates one of the task fields, such as Assigned To, Status, Priority, Remaining Work, etc.

**Add task comment** – adds a new comment.

**Update task comment** – updates an existing comment.

**Delete comment** – deletes a comment.

**NOTE**: each controller only passes the call to an underlying Manager instance which does the Job. For example, the Task controller may call TaskManager.AssignTaskTo whenever a PUT operation is called with relevant data.

Working like this, we decouple the “communication” stack from the business stack, so each can change independently.

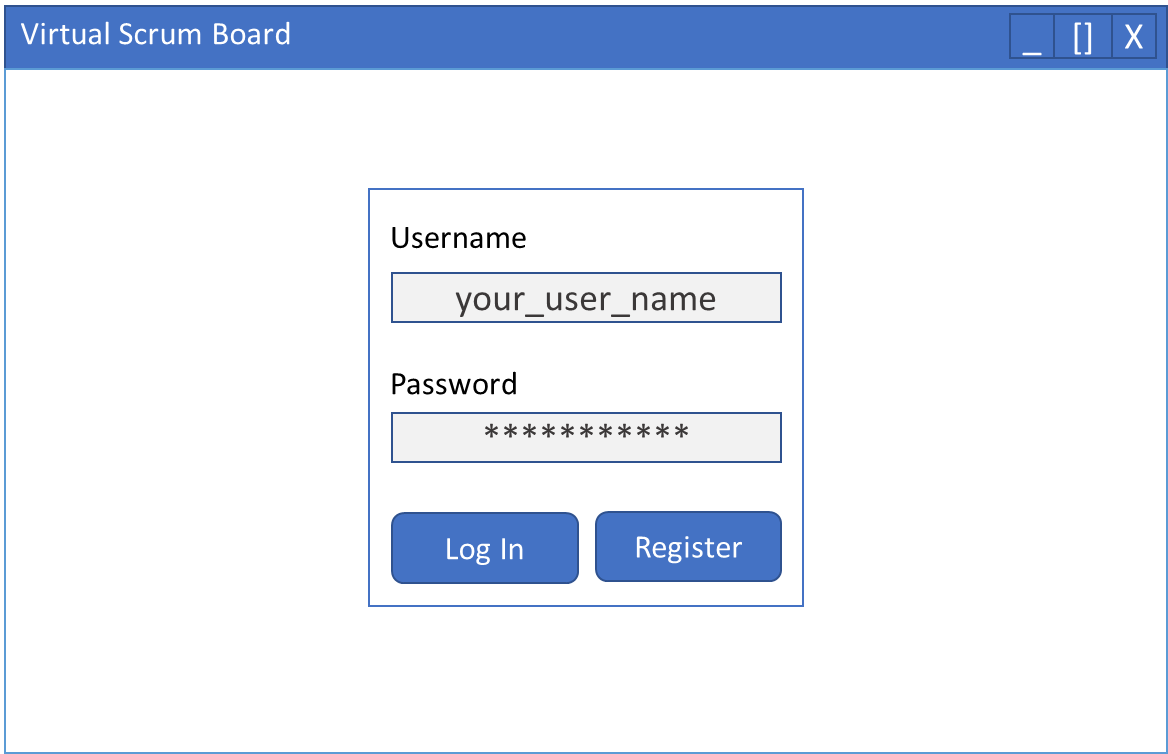
## Client and UI

The desktop client, will be developed with WPF stack, using Caliburn.Micro as MVVM framework.

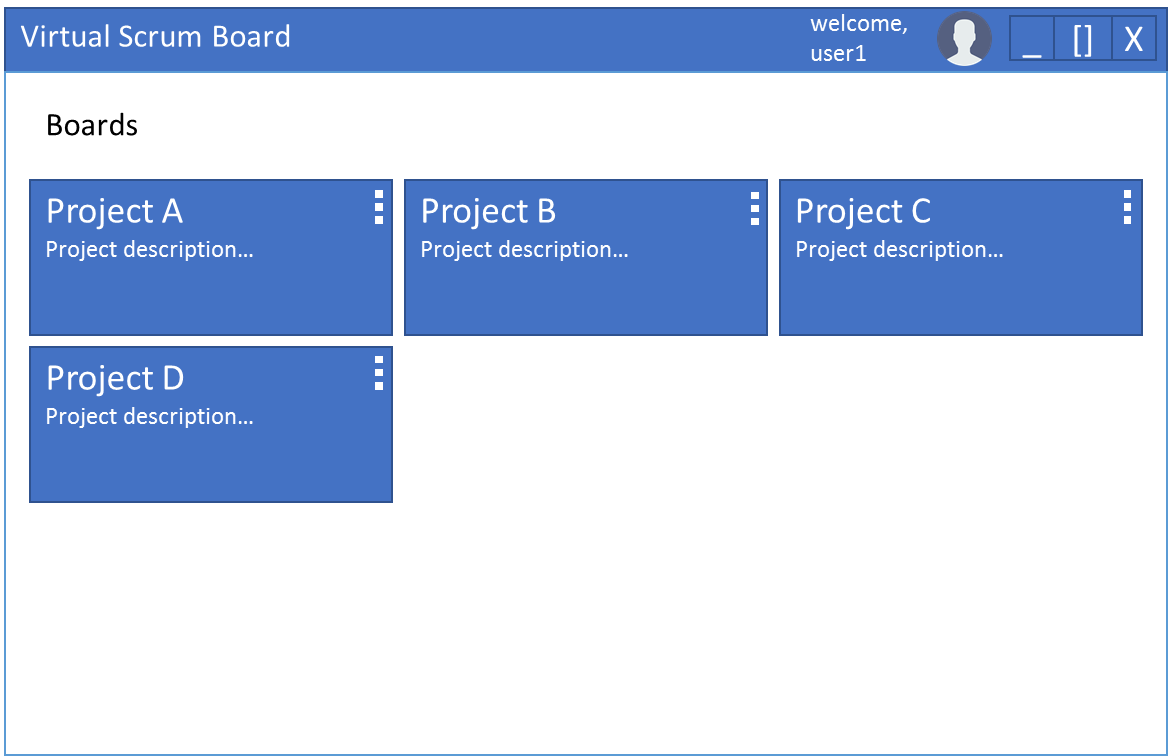
The client logic and UI should provide visual way to interact with the above service and data as described next.

* When client starts, it polls for changes on Boards, Tasks and update them accordingly. Change can be observed if the Version of the object held by the client is different than that on the server (See Version field on each data entity).
* The client uses Refit library to access REST apis.

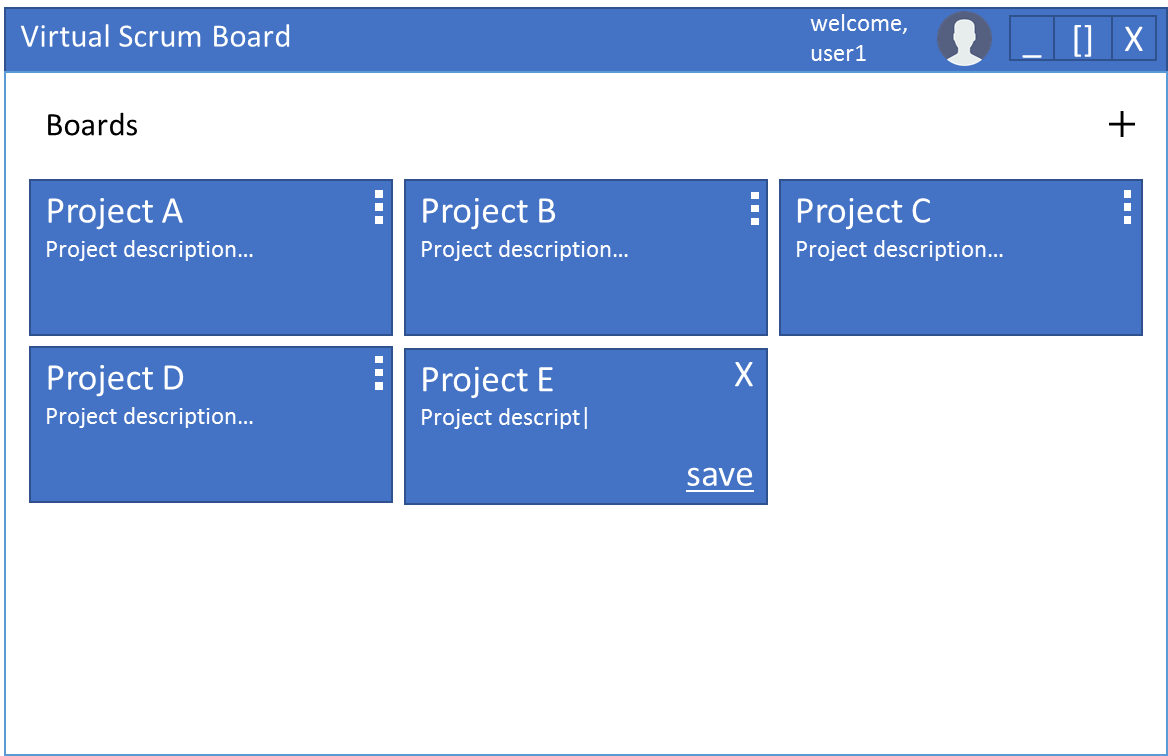
As the user runs the application, a login view is displayed. The user fills in the details or register with a registration form if first time. Clicking register, quickly registers the user with the user name and password provided. Then auto login the user.



Once the user logged in, there is a list of all existing board (or none if no boards).

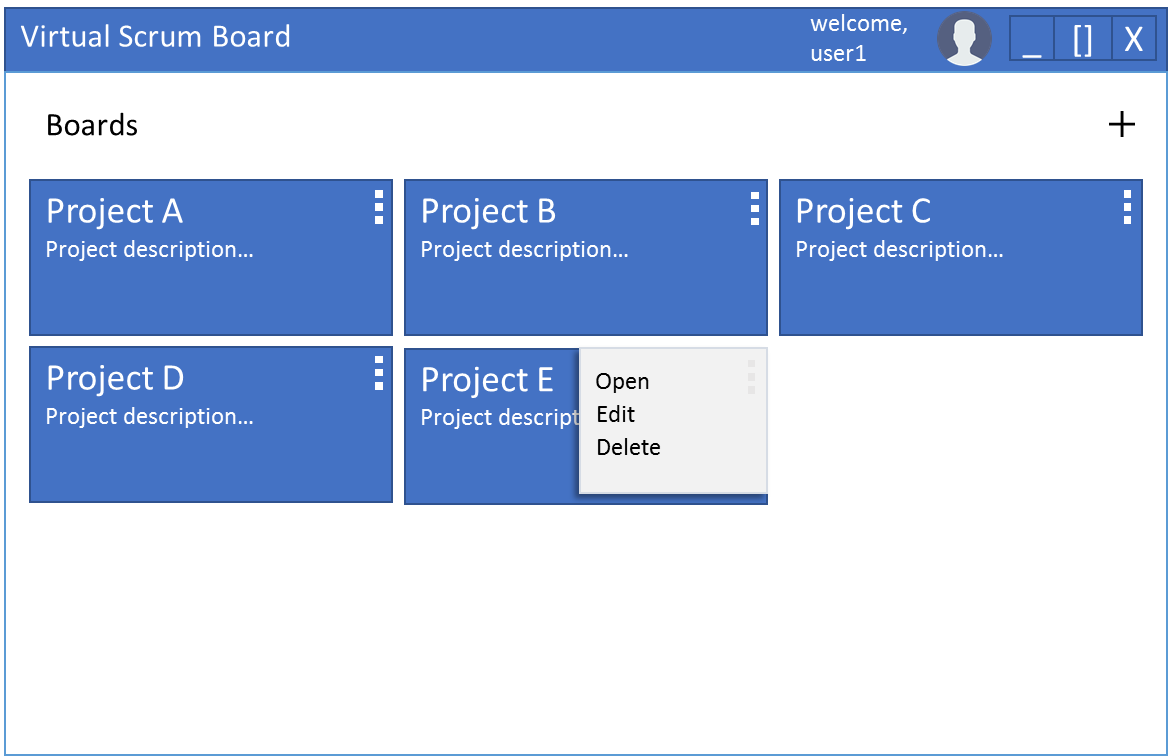


Clicking ‘+’ sign on the right, creates a new board:

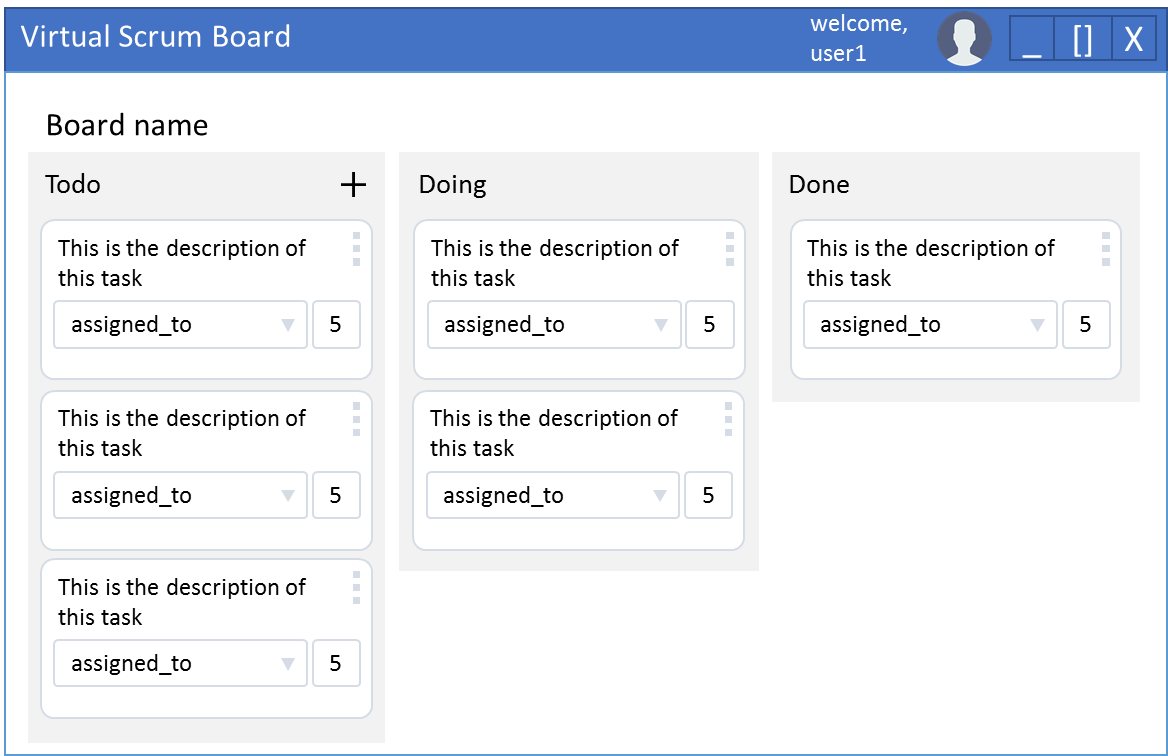


Clicking the board menu, the user can:

1. Open the board
2. Update board details
3. Delete the board



Double click a board – opens it and move to tasks screen.



In the tasks screen, a user can:

1. Add Todo task by clicking the ‘+’. This creates a new task on the top of the list, and the user can edit its details, in place, and then click save, or ‘X’ to remove. Same as creating new board (see above).
2. By clicking the Assigned to combo, a list of all users assigned to same board are listed, the user can pick one to assign the task to.
3. Clicking the Remaining Work number, the user can change it.
4. Dragging the task to other columns, changes the task status accordingly.
5. Clicking the menu button, opens a menu with:
   1. Edit – edits the task.
   2. Delete – deletes the task.

Clicking edit in menu, navigates to Task Edit view:

