

dotNet5782_9171_7973

- [dotNet5782_9171_7973](#)
 - [Bonus Review](#)
 - [General](#)
 - [New c# Features](#)
 - [Dal](#)
 - [Structure](#)
 - [Logic Deletion](#)
 - [Extensive Use of Generic](#)
 - [PL](#)
 - [Custom Window Layout - Docking](#)
 - [Regular Expression](#)
 - [User Interface](#)
 - [Full Support of All Data Queries](#)
 - [MVVM](#)
 - [PO Entities](#)
 - [Custom UserControl](#)
 - [Miscellaneous](#)
 - [External dictionary](#)
 - [The application sends an email using Smtp object:](#)
 - [Simulator](#)
 - [Location Update](#)
 - [Parallel Activation](#)
 - [Busy Indicator](#)
 - [Prevent Application Closing](#)
 - [Maps](#)
 - [Design patterns](#)
 - [Factory - Full structure](#)
 - [Singleton](#)
 - [Last But Not Least - Well Neat, Organized and Detailed README](#)

Bonus Review

General

New c# Features

- [Record](#)
- [Switch Expression](#)
- [Tuples](#)
- [Init Only Setters](#)
- [Using Statement](#)
- [Range Operator](#)

Dal

Structure

We have implemented the layers model in the second structure (The Bonus structure), So we have a config file which follows the given format:

```
<config>
  <dal>[chosen-dal]</dal>
  <dal-packages>
    <[package-1]>
      <class-name>[package-1-class-name]</class-name>
      <namespace>[package-1-namespace]</namespace>
    </[package-1]>
    ...
  </dal-packages>
</config>
```

This format allows to specify the namespace in addition to the class name.

Logic Deletion

The dal deletion is just a *Logic Deletion* rather than *Real Deletion*. All the `Dal` entities implement the interface `IDeetable` which consists of just one property as follows

```
interface IDeetable
{
    bool IsDeleted { get; set; }
}
```

From now, *deletion* is changing the entity's `IsDeleted` property to `true`. Only non-deleted entities are allowed to perform actions.

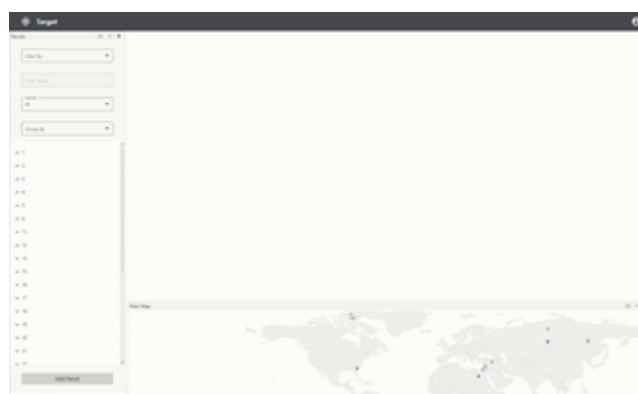
Extensive Use of Generic

In order to Avoid repetition according to the **DRY** principle, We implemented all our `Dal` methods as generic methods. So, Instead of having `AddDrone`, `AddParcel`, `AddBaseStation` and `AddCustomer` for example, We only have `AddItem<T>` method.

PL

Custom Window Layout - Docking

Layout is very flexible and easy to use.

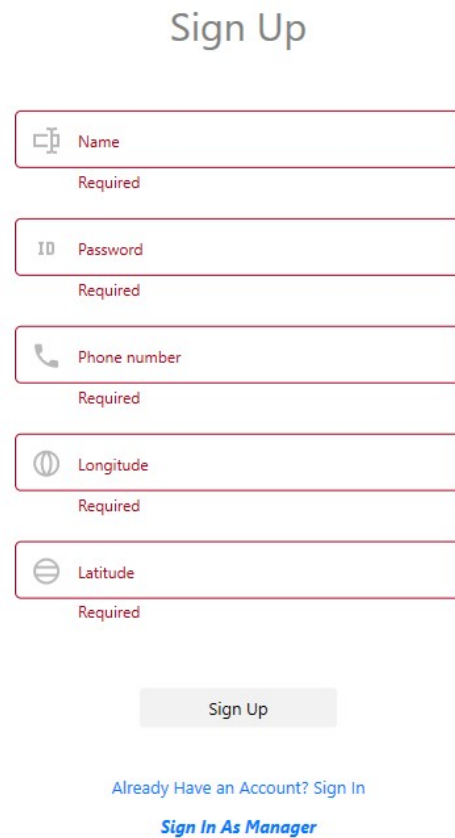


Regular Expression

We used `Regex` for validity checking. [example](#)

User Interface

Our project supports two modes: customer mode and manager mode. When running the program the following screen shows up:



The image shows a 'Sign Up' form with a title 'Sign Up' at the top. Below the title are five input fields, each with a red border and a red 'Required' label underneath. The fields are: 1. Name (with a person icon), 2. Password (with a key icon), 3. Phone number (with a phone icon), 4. Longitude (with a globe icon), and 5. Latitude (with a globe icon). Below the fields is a grey 'Sign Up' button. At the bottom, there is a link 'Already Have an Account? Sign In' and a blue link 'Sign In As Manager'.

Sign Up

Name
Required

Password
Required

Phone number
Required

Longitude
Required

Latitude
Required

Sign Up

[Already Have an Account? Sign In](#)

[Sign In As Manager](#)

Then pressing the `Sign In As Managar` button enters the program in manager mode.

Signing up enters with a new customer account in customer mode.


Clicking on `Already Have an...` gives sign in page like this screen:

Sign In

ID

Id

Required



Name

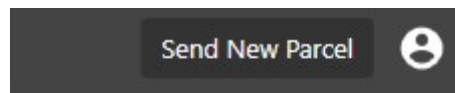
Required

Sign In

[Do Not Have an Account? Sign Up](#)

[Sign In As Manager](#)

Clicking `Log out` any time brings back to register window, where reconnecting is available again.



Full Support of All Data Queries

Very easy way to accsses accurate data. (uses reflection)

- Filter (notice the dynamic input)

Drones

Filter By

Filter Value

Sort By

Id

Group By

▶ fKde3

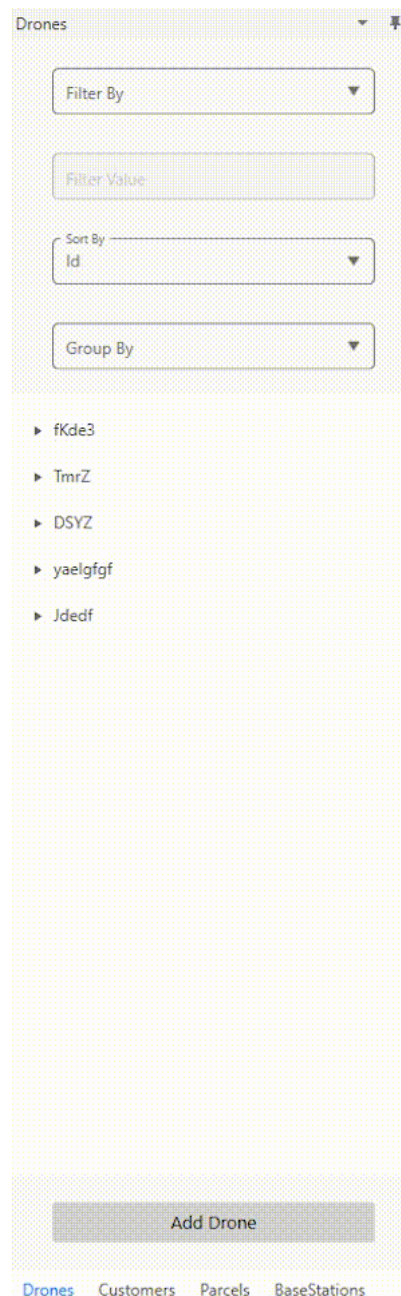
▶ TmrZ

▶ DSYZ

▶ yaelgfgf

▶ Jdedf

- Sort and Group



MVVM

We used **FULL** MVVM, with full binding.

PO Entities

PO entities are mainly used as Model s in MVVM.

Custom UserControl

Extensive use in UserControl rather than window control.

Makes the user experience better, and makes the application look better.

Miscellaneous

- Triggers
 - Event Trigger [example](#)
- Behaviors examples: [definition](#) [use](#)
- Converters [example](#)

- Commands (We implemented A `RelayCommand` class and used it as properties in our `PL` classes) [example](#)
- Data templates [example](#)
- `ObservableCollection` [example](#)
- `CollectionView` [example](#)
- `ContextMenu` [example](#)

External dictionary

We used dictionary for style definitions, outer `Data templates`, etc.

The application sends an email using `Smtp` object:

- To sender- when his parcel is sent
- To reciever- when he gets a parcel

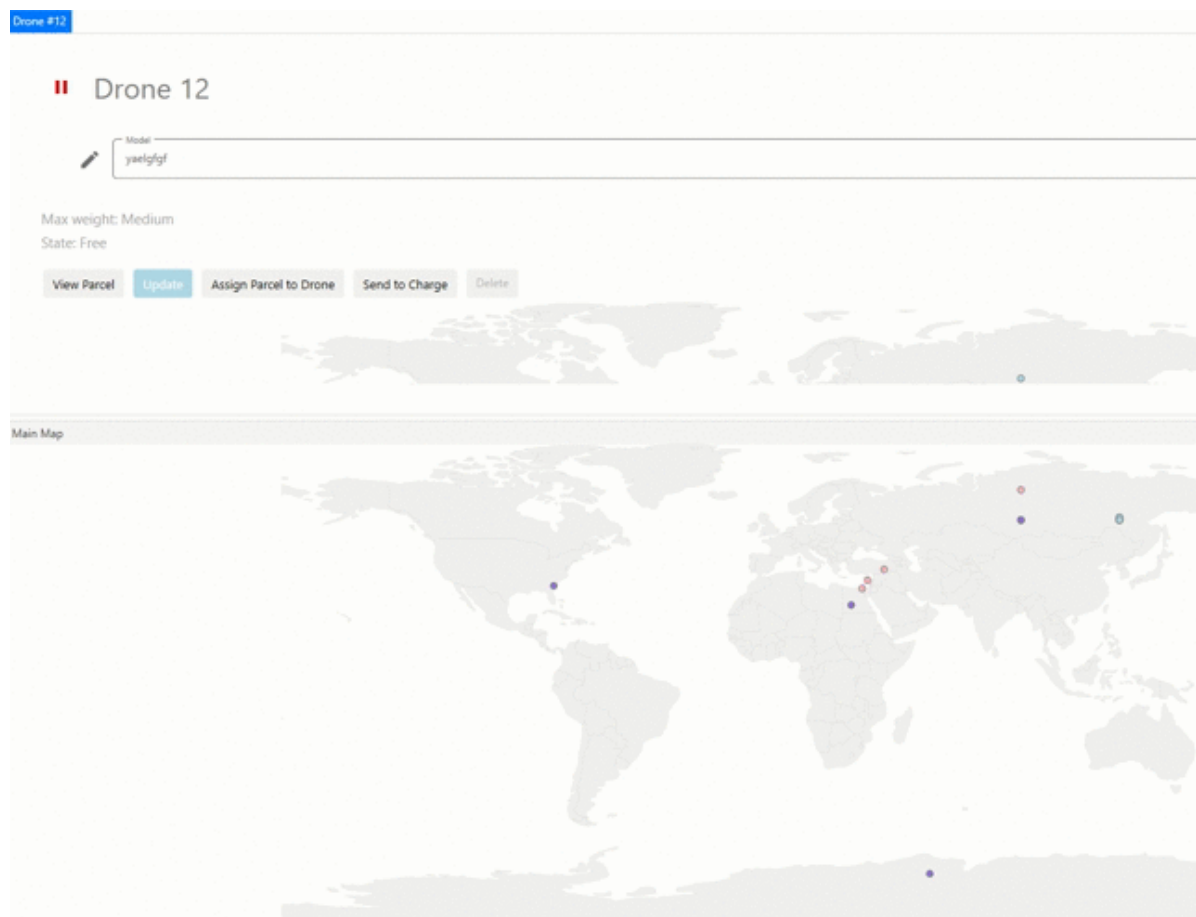
[see code](#)

Simulator

Location Update

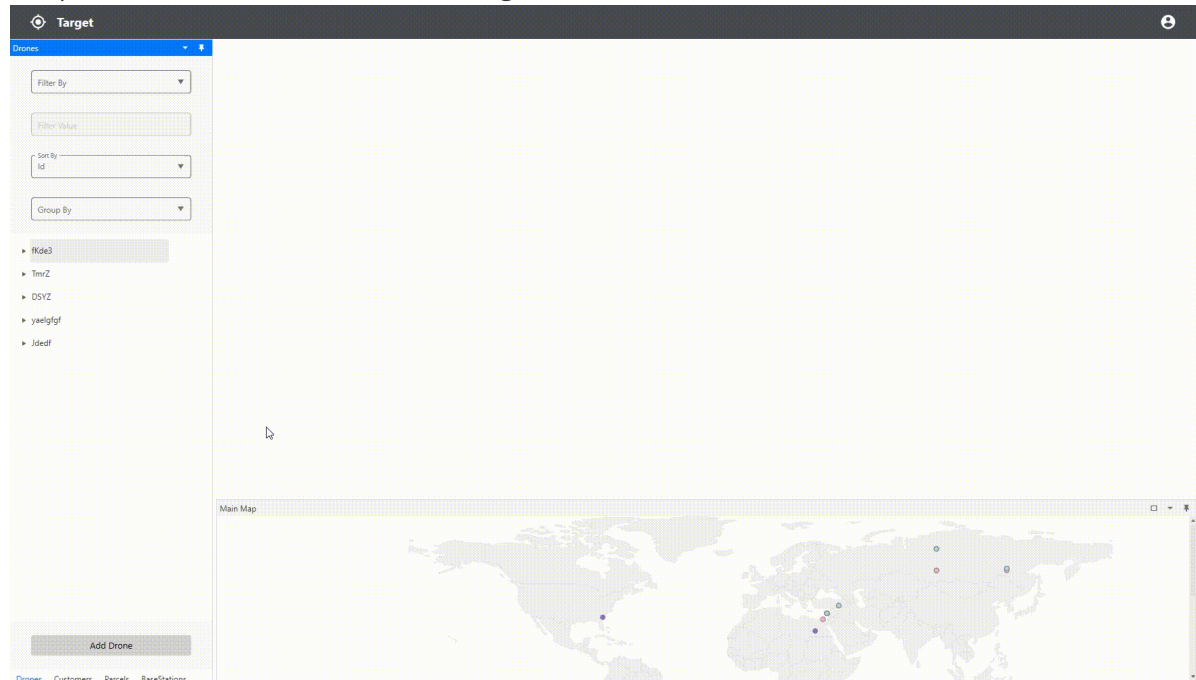
Location updates in all related items while running.





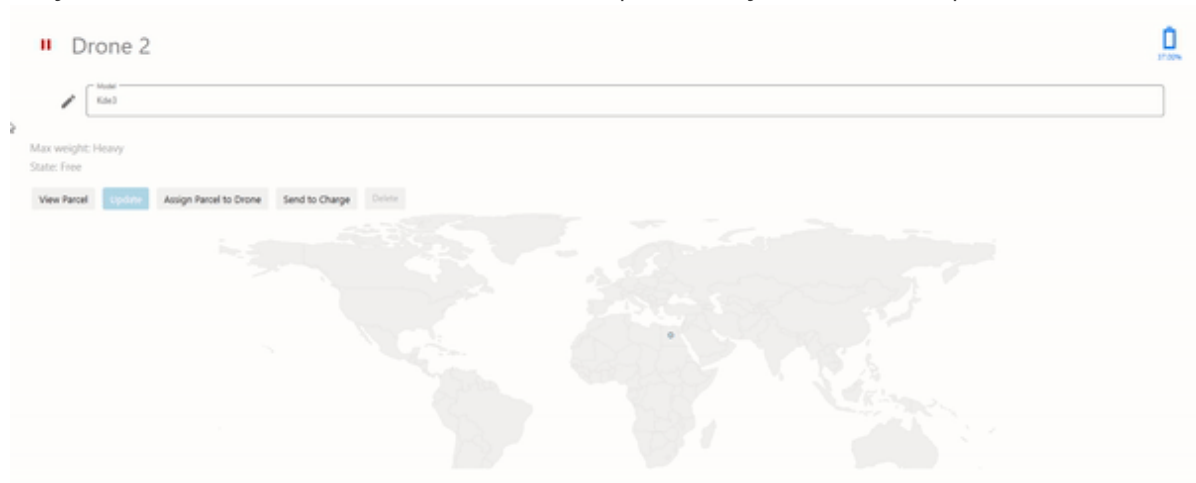
Parallel Activation

It is possible to run several simulators together. (not limited)



Busy Indicator

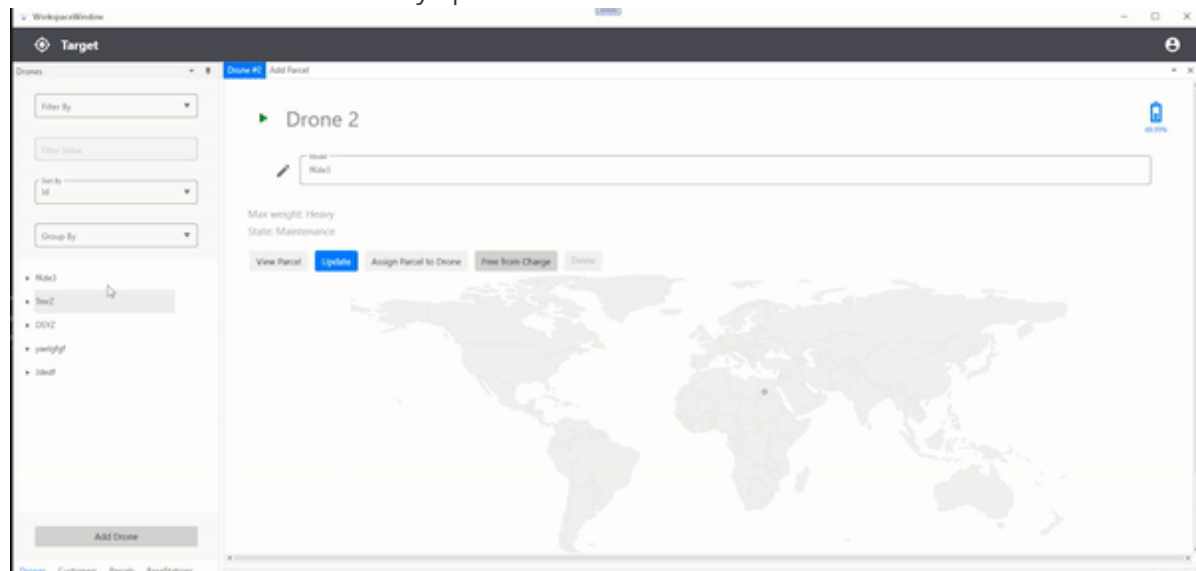
Busy indicator is on while simulator is about to stop its activity, after user's request.



Prevent Application Closing

The Application Prevent Closing As long as Simulators are On.

This is to make sure all data is fully updated.



Maps

Each entity has its map to represent its location, Besides, there is a **Main Map** for all the entities together.

Design patterns

Factory - Full structure

We used the full (bonus) structure for our **Factory**.

It finds the requested implementation of the service contract and supplies its **Instance**.

To get it the following piece of code alone is necessary:

```
dalApi.IDal dal { get; } = dalApi.DalFactory.GetDal();
```


Singleton

We implemented an abstract class `Singleton` which has lazy initialization and is thread-safe. The `Da1` and `BL` layers just inherit it.

Last But Not Least - Well Neat, Organized and Detailed [README](#).