MonoTouch.Dialog

DEVELOPMENTOR
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Who Am I?

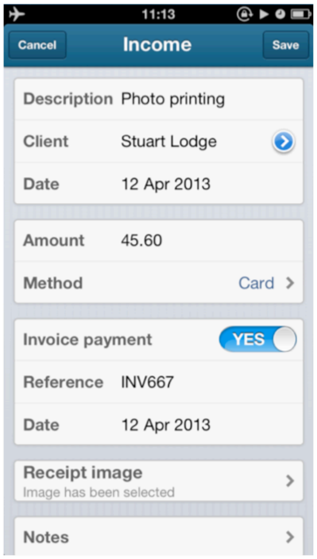
- My Name is Mark Smith
  - mark@julmar.com
  - @marksm
  - julmar.com/blogs/mark
- DEVELOPMENTOR author and instructor
- Consultant through my company julmar.com – focused primarily on UI design, parallel programming and debugging
- Microsoft MVP

Agenda

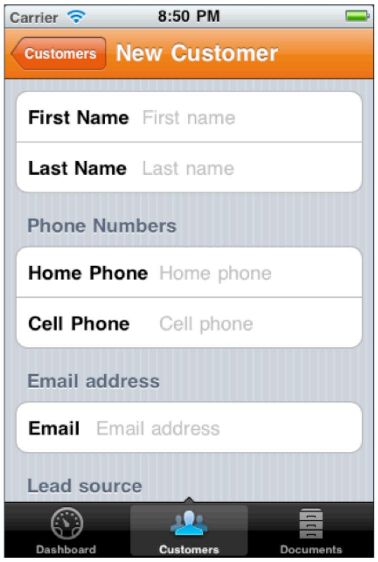
- What is MonoTouch.Dialog?
- Providing data
- Data Styles
- Customizing elements
- Navigation
- Pull to refresh support
- Pulling state out of the DVC

What is MonoTouch.Dialog?

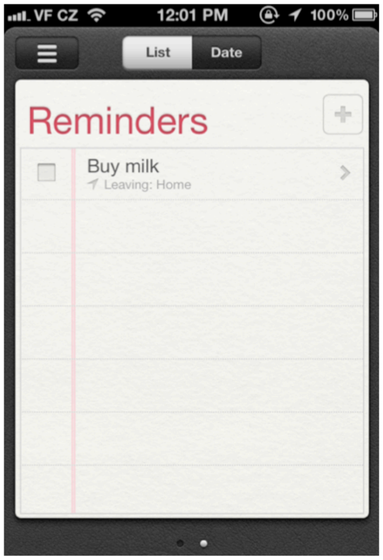
- MonoTouch.Dialog is an API to build **`UITableView`** dialogs



Description Photo printing
Client Stuart Lodge
Date 12 Apr 2013
Amount 45.60
Method Card
Invoice payment YES
Reference INV667
Date 12 Apr 2013
Receipt image
Image has been selected
Notes



First Name First name
Last Name Last name
Phone Numbers
Home Phone Home phone
Cell Phone Cell phone
Email address
Email Email address
Lead source

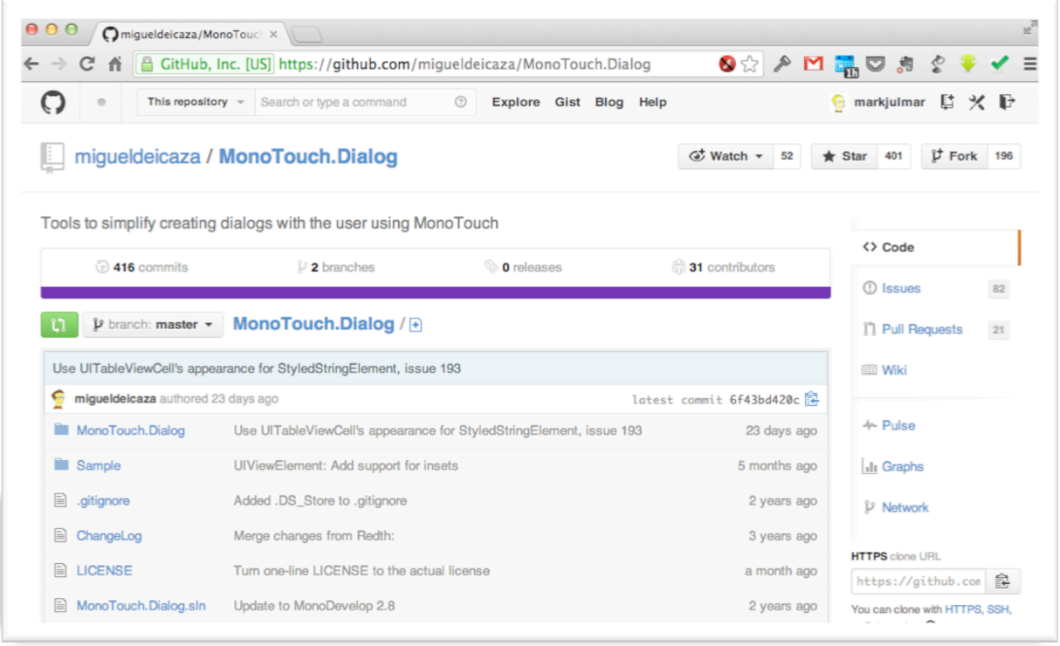


Reminders
Buy milk
Leaving: Home

It was built to support all the features and capabilities of **`UITableView`** without the tedium of creating all the necessary support classes, instead you provide the data and it generates the proper views for you automatically

Where is it?

- MonoTouch.Dialog is an open-source project that was originally shipped separately but has now been added into Xamarin.iOS



migueldeicaza / MonoTouch.Dialog

Tools to simplify creating dialogs with the user using MonoTouch

416 commits 2 branches 0 releases 31 contributors

MonoTouch.Dialog /

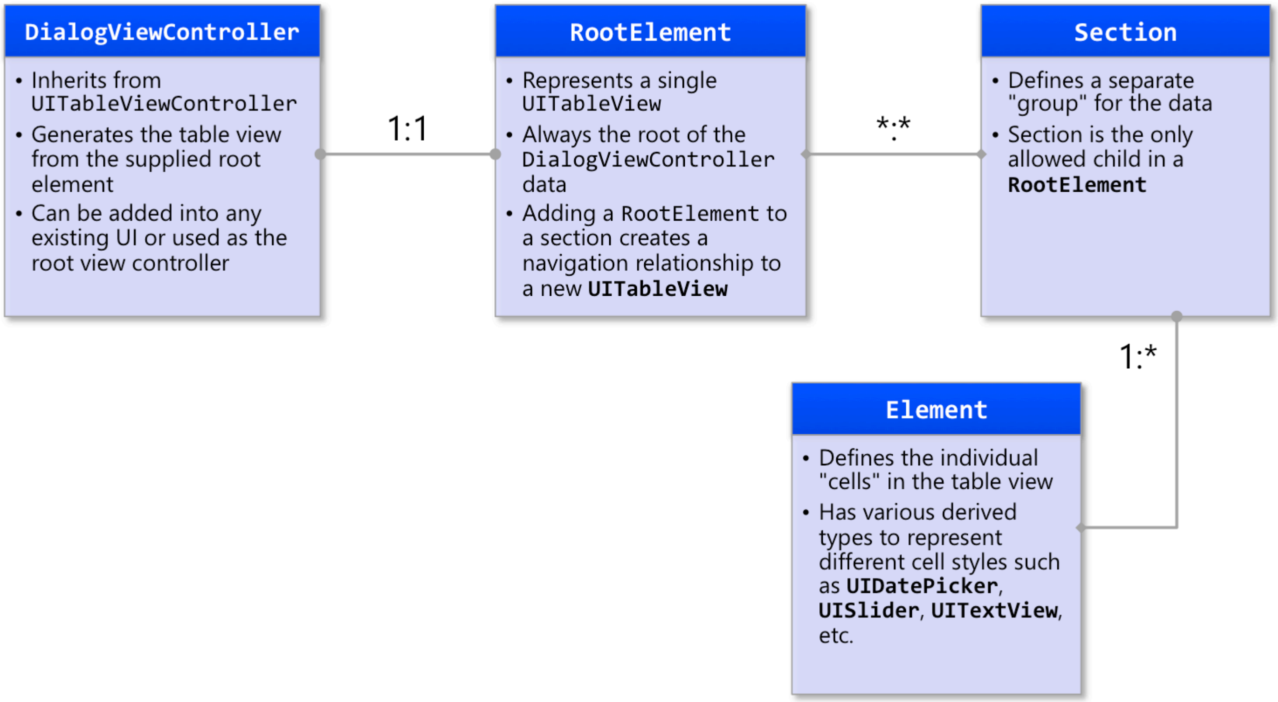
Use UITableViewCell's appearance for StyledStringElement, issue 193

migueldeicaza authored 23 days ago latest commit 6f43bd420c

MonoTouch.Dialog	Use UITableViewCell's appearance for StyledStringElement, issue 193	23 days ago
Sample	UIViewElement: Add support for insets	5 months ago
.gitignore	Added .DS_Store to .gitignore	2 years ago
ChangeLog	Merge changes from Redth:	3 years ago
LICENSE	Turn one-line LICENSE to the actual license	a month ago
MonoTouch.Dialog.sln	Update to MonoDevelop 2.8	2 years ago

Providing data

- MT.d builds screens using four composable parts



`DialogViewController`

- Inherits from `UITableViewController`
- Generates the table view from the supplied root element
- Can be added into any existing UI or used as the root view controller

1:1

`RootElement`

- Represents a single `UITableView`
- Always the root of the `DialogViewController` data
- Adding a `RootElement` to a section creates a navigation relationship to a new **`UITableView`**

:

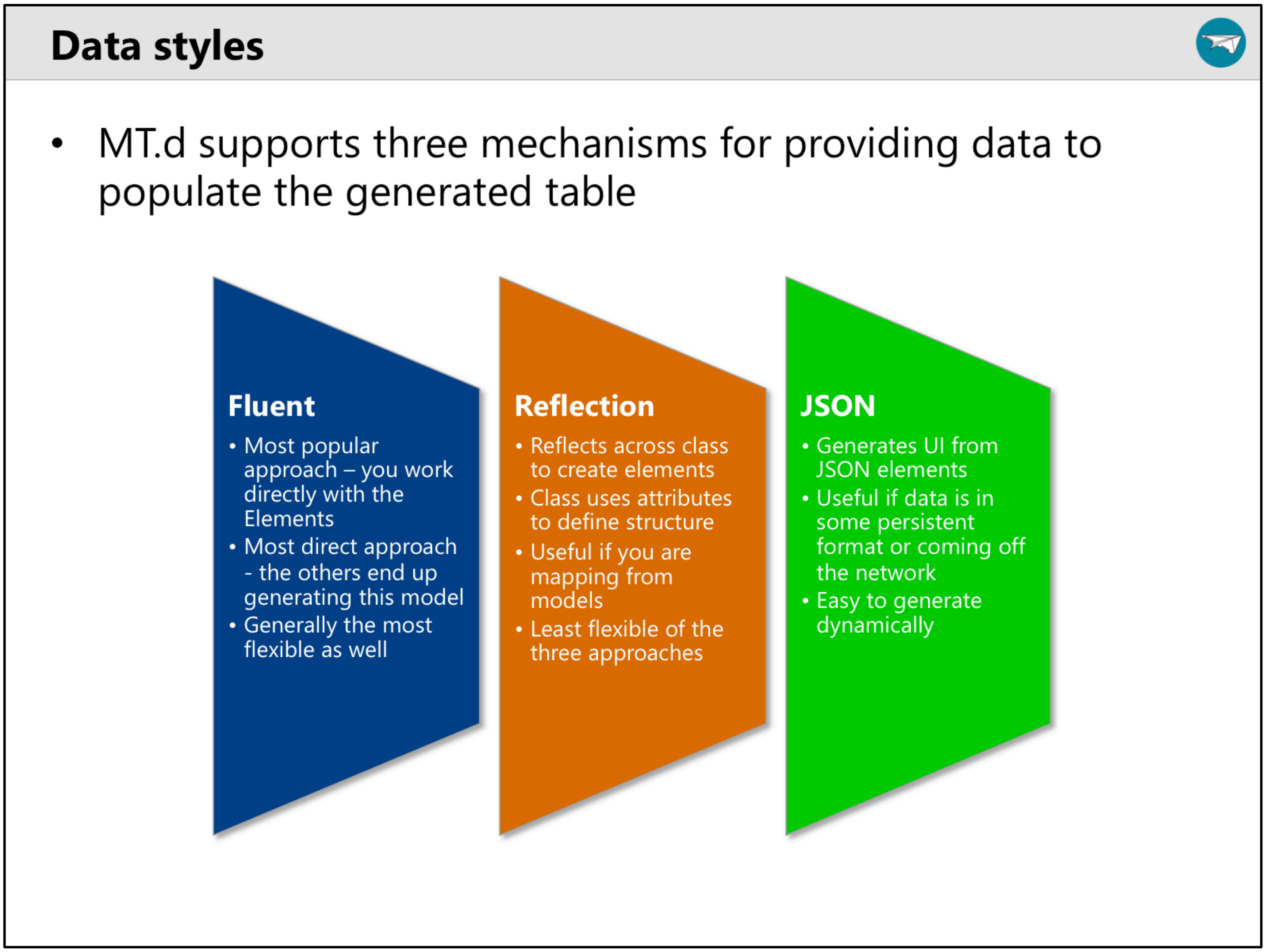
`Section`

- Defines a separate "group" for the data
- Section is the only allowed child in a **`RootElement`**

1:*

`Element`

- Defines the individual "cells" in the table view
- Has various derived types to represent different cell styles such as **`UIDatePicker`**, **`UISlider`**, **`UITextView`**, etc.



Data styles

- MT.d supports three mechanisms for providing data to populate the generated table

Fluent

- Most popular approach – you work directly with the Elements
- Most direct approach - the others end up generating this model
- Generally the most flexible as well

Reflection

- Reflects across class to create elements
- Class uses attributes to define structure
- Useful if you are mapping from models
- Least flexible of the three approaches

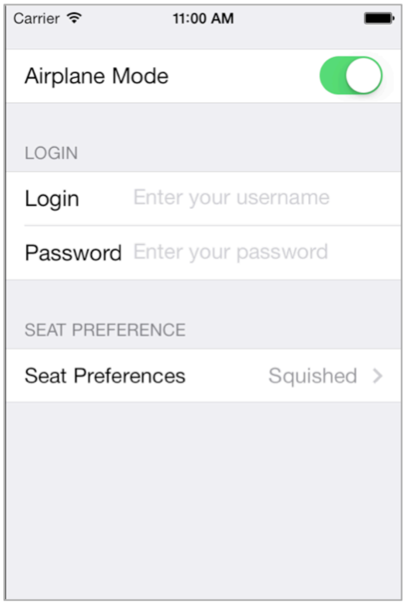
JSON

- Generates UI from JSON elements
- Useful if data is in some persistent format or coming off the network
- Easy to generate dynamically

Fluent model

- Fluent model builds up the screen by manually composing the element graph

```
var root = new RootElement ("Account") {
   new Section() {
      new BooleanElement("Airplane Mode", true),
   },
   new Section("Login") {
       new EntryElement("Login", "Enter your username", ""),
       new EntryElement("Password", "Enter your password", "",
                           true),
   },
   new Section("Seat Preference") {
       new RootElement("Seat Preferences",
                        new RadioGroup(1))
       {
          new Section() {
             new RadioElement("Window"),
             new RadioElement("Squished"),
             new RadioElement("Aisle"),
          }
       }
   }
};
```



Carrier 11:00 AM

Airplane Mode

LOGIN

Login: Enter your username

Password: Enter your password

SEAT PREFERENCE

Seat Preferences: Squished

Reflection model

- Reflection model works off a decorated class instance

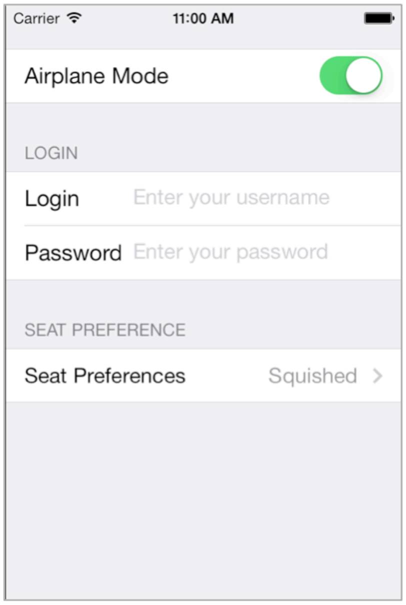
```
using MonoTouch.Dialog;

public sealed class AirPrefs
{
    [Section]
    public bool AirplaneMode { get; set; }

    [Section("Login")]
    [Entry("Enter your username")]
    public string UserName { get; set; }

    [Password("Enter your password")]
    public string Password { get; set; }

    [Section("Seat Preference")]
    public SeatPreference Preference;
}
```



Airplane Mode ●

LOGIN

Login Enter your username

Password Enter your password

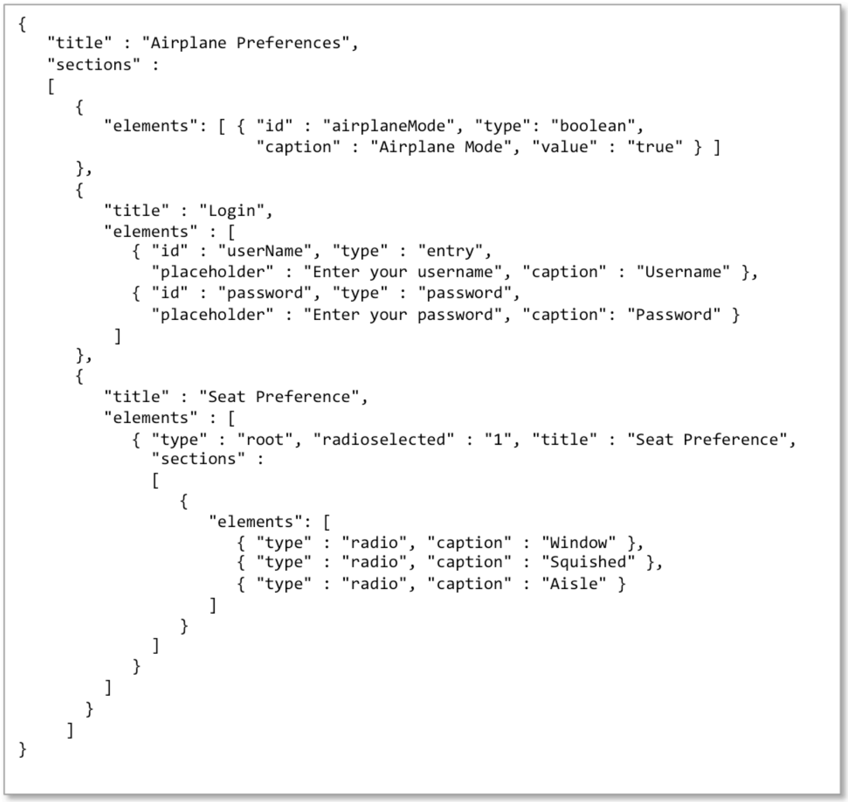
SEAT PREFERENCE

Seat Preferences Squished >

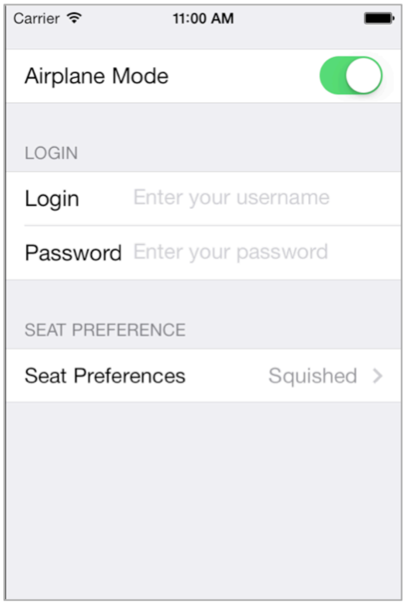
MT.d includes a **`BindingContext`** class to support this model which generates an element graph from a supplied object instance

JSON model

- JSON model creates elements from JSON object



```
{
   "title" : "Airplane Preferences",
   "sections" :
   [
      {
         "elements": [ { "id" : "airplaneMode", "type": "boolean",
                         "caption" : "Airplane Mode", "value" : "true" } ]
      },
      {
         "title" : "Login",
         "elements" : [
            { "id" : "userName", "type" : "entry",
              "placeholder" : "Enter your username", "caption" : "Username" },
            { "id" : "password", "type" : "password",
              "placeholder" : "Enter your password", "caption": "Password" }
          ]
      },
      {
         "title" : "Seat Preference",
         "elements" : [
            { "type" : "root", "radioselected" : "1", "title" : "Seat Preference",
              "sections" :
              [
                 {
                    "elements": [
                       { "type" : "radio", "caption" : "Window" },
                       { "type" : "radio", "caption" : "Squished" },
                       { "type" : "radio", "caption" : "Aisle" }
                    ]
                 }
              ]
            }
         ]
       }
     ]
}
```



Airplane Mode

LOGIN

Login Enter your username

Password Enter your password

SEAT PREFERENCE

Seat Preferences Squished

Using MonoTouch.Dialog

- First step is to create a **`DialogViewController`** – this is a table view controller that loads the data from some supplied source

Constructor parameter is the **`RootElement`** used to create the table view

```
public override bool FinishedLaunching (UIApplication app,
                                        NSDictionary options)
{
   window = new UIWindow (UIScreen.MainScreen.Bounds);

   RootElement root = ...;
   DialogViewController rootVC = new DialogViewController (root);

   window.RootViewController = rootVC;
   window.MakeKeyAndVisible ();

   return true;
}
```

You can make it the root view controller like this, or use it as a child view controller

Root Element

- **`RootElement`** is used to create a single table view – it holds a set of **`Section`** elements which are used for grouping

Optional parameter provides caption

Section can have headers and footers which can either be a string, or a **`UIView`** – these are just passed to the constructor

```
var root = new RootElement ("Account")
{
   new Section("Login")
   {
      ...
   },
   ...
   new Section() { ... }
};
```

Sections

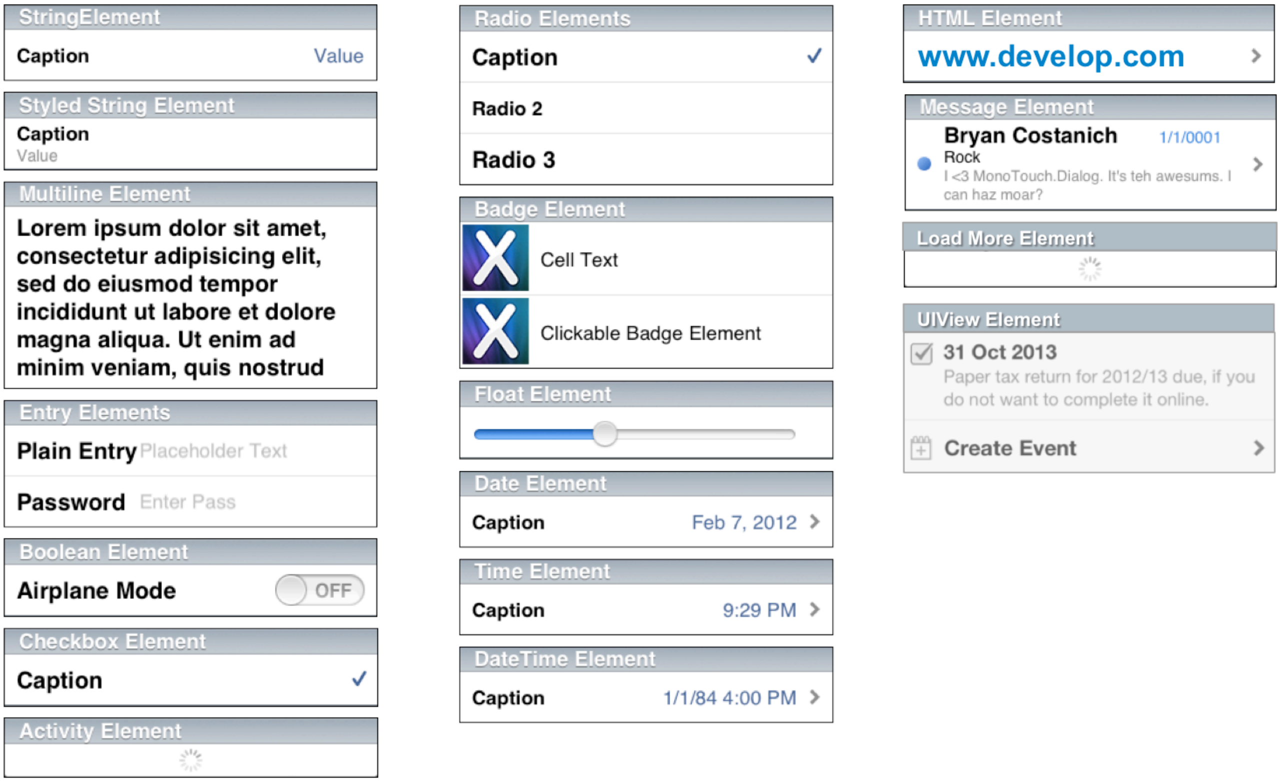
- Sections are populated with elements – each element will render a cell in the table

```
new Section("Login")
{
   new EntryElement("Login", "Enter your username", ""),
   new EntryElement("Password", "Enter your password", "", true),
   new BooleanElement("Remember Login", true),
   ...
},
...
```

Element graph is mutable – i.e. if you change elements at runtime the changes will be reflected in the UI

Element Types

- System includes variety of common UI element types



StringElement
Caption Value

Styled String Element
Caption
Value

Multiline Element
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud

Entry Elements
Plain Entry Placeholder Text
Password Enter Pass

Boolean Element
Airplane Mode OFF

Checkbox Element
Caption ✓

Activity Element

Radio Elements
Caption ✓
Radio 2
Radio 3

Badge Element
Cell Text
Clickable Badge Element

Float Element

Date Element
Caption Feb 7, 2012

Time Element
Caption 9:29 PM

DateTime Element
Caption 1/1/84 4:00 PM

HTML Element
www.develop.com

Message Element
Bryan Costanich 1/1/0001
Rock
I <3 MonoTouch.Dialog. It's teh awesums. I can haz moar?

Load More Element

UIView Element
31 Oct 2013
Paper tax return for 2012/13 due, if you do not want to complete it online.
Create Event

Customizing elements

- Can create new element types to visualize custom **`UITableViewCell`** and provide rich UI cells

Derive from **`Element`** type closest to what you want to do

```
public class ChatBalloonElement : Element
{
   static NSString key = new NSString ("balloonElement");
   private InstantMessage _message;

   public ChatBalloonElement(InstantMessage msg) : base (null)
   {
      _message = msg;
   }

   public override UITableViewCell GetCell (UITableView tableViewCell)
   {
      var cell = tv.DequeueReusableCell (key) as BalloonTableViewCell;
      if (cell == null)
         cell = new BalloonTableViewCell(_message, key);
      else
         cell.SetMessageData(_message);
      return cell;
   }
}
```

Navigation

- Adding a **`RootElement`** into a section creates a navigation page that displays a new table view – requires that the root controller is a **`UINavigationController`**

```
var root = new RootElement ("Account")
{
   ...
   new Section()
   {
       new RootElement("Page Title")
       {
          new Section() { ... }
          new Section() { ... }
       }
   }
};
```

Just define the secondary view with the same fluent syntax

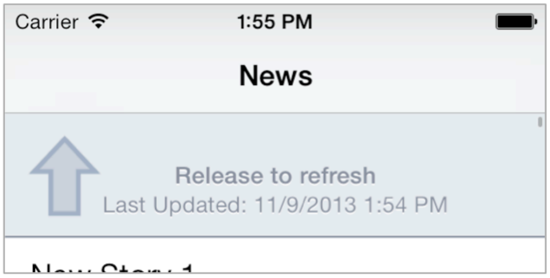
Responding to actions

- Can use a **`StringElement`** in the view to provide a "button" and then wire up an action to present a secondary view (or perform any runtime action)

```
new Section()
{
    new StringElement("Tap Me", () =>
    {
        rootVC.PresentViewController(
            new SecondaryViewController(), true, null);
    }),
    ...
},
...
```

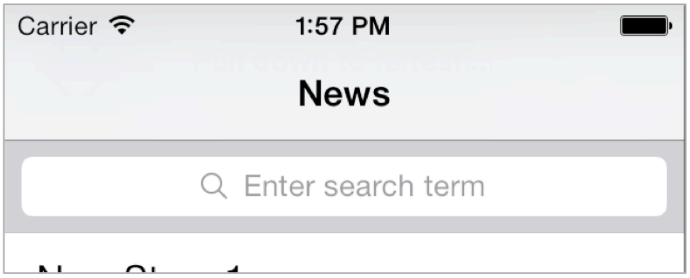
Higher level functions

- MT.d supports two main-stream features useful for most table view applications



Release to refresh
Last Updated: 11/9/2013 1:54 PM

"pull-to-refresh"



search bar

Pull to refresh support

- Can easily add "pull-to-refresh" support to the application by handling event, refreshing data and then reloading UI

```
public override bool FinishedLaunching (UIApplication app,
                                        NSDictionary options)
{
   ...
   DialogViewController rootVC = new DialogViewController (root);
   rootVC.RefreshRequested += OnRefreshRequested;
   ...
}

private void OnRefreshRequesed(object sender, EventArgs e)
{
   // Reload data
   ...
   ((DialogViewController)sender).ReloadComplete();
}
```

call `ReloadComplete` on the UI thread once the refresh is complete – this will remove the "refreshing" UI that is added when the swipe occurs

Search support

- Search is enabled by setting property, MT.d will automatically search all visible fields and constrain the view to only contain matching terms – you can also provide custom filtering or trigger the search programmatically

```
public override bool FinishedLaunching (UIApplication app,
                                        NSDictionary options)
{
   ...
   DialogViewController rootVC = new DialogViewController (root);
   rootVC.EnableSearch = true;
   rootVC.SearchPlaceholder = "Enter Search Term";
   ...
}
```

Reading state out of the UI

- MT.d is really oriented towards *live data display* – it has limited support for reading state out of the table view
- When using the Fluent or JSON model, you can always read the current value
- When using the Reflection model, the **`BindingContext`** has a **`Fetch`** method which will refresh the object state from the UI graph – this can be called when the view is being dismissed for example
- Some elements have events which are raised when values are changed – this allows for live change monitoring, however the support is not complete

Reading state – Reflection API

- Reflection API uses **`BindingContext`** to provide binding from elements to an object and vice-versa

```
AirPrefs preferences = new AirPrefs ()
{ AirplaneMode = true, Preference = SeatPreference.Squished };

var context = new BindingContext(null, preferences,
                                 "Preferences");
...
var childController = new DialogViewController(context);
childController.ViewDisappearing += (sender, e) =>
{
   context.Fetch(); // pull values from UI into object
};
PresentViewController(childController, true, null);
```

Reading state – elements model

- Can read the state of the UI directly from the elements – this works for every model

```
BooleanElement airplaneMode;

var root = new RootElement ("Account") {
   new Section()
   {
      (airplaneMode = new BooleanElement("Airplane Mode", true)),
      ...
   },
   ...
};

airplaneMode.ValueChanged += (sender, e) =>
{
   if (airplaneMode.Value == true)
      TurnOnAirplaneMode();
   else
      TurnOffAirplaneMode();
};
```

Detecting live changes

- Not all elements support change notification – can always derive a new class to provide missing notification if you really need it

Element	Interaction Event
`BooleanElement` `ImageBooleanElement`	`ValueChanged`

`StringElement`
`ImageStringElement`
`RadioElement`
`CheckboxElement`
`DateTimeElement`
	`Tapped` (not really a changed event – but indicates the item was activated)

`StyledStringElement` `StyledMultilineElement`	`AccessoryTapped` `Tapped`
`EntryElement`	`Changed`

Summary

- MonoTouch.Dialog takes the burden out of creating table view user interfaces
- Can read data from Elements API, Reflection API or JSON
- UI is composed of sections and elements
- Can create multi-level master/detail views
- Can customize the UI however you desire
- Even supports "pull-to-refresh" and search capabilities out of the box