Business Continuity Mobility System iPhone Front End Application Module Design Specification

Author	Revision Number	Date
architect	1.0	

Contents

Application Design Specification

1. Design

We are looking for a solution for key teams within our company to keep critical services working in the event of a major disaster (ranging from major technical infrastructure failures to fires and other local environmental disasters). The Business Continuity Mobility Solution should allow us to manage at least three key functions:

- λ Incident alert and notification
- λ Contact information
- λ Business continuity plans and checklists

This contest will take the provided ARS, storyboard, and services architecture and will define the architecture to implement the front end controllers for the iPhone application.

1.1 Work Flow Description

The workflow of this application is like this:
The user starts the application, perform login, and see the main menu. From the main menu, the user chooses a sub-menu of the application or logs out like this:

If the user chooses Incidents for example, the application will display a list of Incidents like this:

Then the user can choose a particular incident to view it, update it, create a new incident, etc. Other functions of the application are similar. The application is basically a platform for the user to view certain information records, and optionally update them. It is a navigation-controller based application.

All data-related operations are delegated to the back-end services, which maintain a local cache transparent to this module. When performing a back-end operation, the user is shown with a modal view with a spinner informing him/her that an operation is underway. So the application will never hang due to network and cache lookup operations.

The local cache is refreshed at a fixed interval to ensure that the data is fresh.

1.2 Component Requirements

1.2.1 TopCoder Software Components

None

1.2.1.1 Existing TopCoder Components

None

1.2.2 Third Party Components

None

1.2.3 Assemblies

Business Continuity Mobility System iPhone Front End Infrastructure Assembly –this assembly should implement some basic infrastructure classes, and the controllers for the login page and menu page. The XIB files of the controllers are also in scope.

Business Continuity Mobility System iPhone Front End Incidents Assembly – this assembly is responsible for the controllers that show incident list, incident details, incident filter, and incident notes. The XIB files of the controllers are also in scope.

Business Continuity Mobility System iPhone Front End Incident Add and Update Assembly –

this assembly is responsible for the controllers that allow the user to add an incident and update an incident. The XIB files of the controllers are also in scope.

Business Continuity Mobility System iPhone Front End Convene Rooms Assembly - this assembly is responsible for the controllers that allow the user to perform convene room-related use cases. The XIB files of the controllers are also in scope.

Business Continuity Mobility System iPhone Front End Contacts Assembly - this assembly is responsible for the Contact-related pages. The XIB files of the controllers are also in scope.

Business Continuity Mobility System iPhone Front End More and Settings Assembly - this assembly is responsible for the controllers that show the More menu and the Settings menu, including all the related pages. The XIB files of the controllers are also in scope.

Business Continuity Mobility System iPhone Front End Integration Assembly - this assembly should integrate all other assemblies of this module to make sure testing is successful.

The dependencies between assemblies are as follows:

The Infrastructure module is the most fundamental one and must be developed first. All other modules depend on it. Although the diagram shows that Infrastructure module also <<uses>> several other modules, that's SOLELY because of the "1.2.7 BCMSMenuViewController". This controller is responsible for the main menu of the application, so that if user clicks one of the menu item, it should push that controller onto the navigation stack. Therefore the Infrastructure Assembly simply needs to mock BCMSIncidentsViewController, BCMSConveneRoomsController, BCMSContactsController, BCMSMoreViewController, and BCMSSettingsController, and each only needs a blank view because Infrastructure doesn't care what these controllers are.

1.3 Application Management

1.3.1 Handling Rotation

The application must support both the Portrait and Landscape orientation. According to the storyboard, the Portrait and Landscape orientations are extremely similar and have no difference in layout at all. The only difference is the width and height of the screen. Therefore simple Autosizing is enough to handle rotation. See here for a tutorial on configuring Autosizing attributes so that rotation can be handled automatically. Basically all sub view's height and width should increase or decrease in proportion to size changes in the corresponding super view. The control is done with the following autoresizing mask values set on each view:
UIViewAutoResizingFlexibleHeight
UIViewAutoResizingFlexibleWidth

Note that in addition to configuring the Autosizing properties for static UI view on Interface Builder, all programmatically created views (such as labels and buttons in table view cells) of this module should also have their ‘autoresizingMask’ property set with the above mask values.

Each custom controller of this module should override the ‘UIViewController.shouldAutorotateToInterfaceOrientation’ method to return true at all time, to allow rotation at any direction.

1.3.2 UI Design

Since the landscape prototype already includes all pages in landscape orientation, and that the portrait orientation is almost the same as the landscape orientation, it’s easy for the assembler to just refer heavily to the prototype when drawing each view using Interface Builder. It is suggested that the assembler open each nib file in the prototype, see its layout, and draw a similar layout in a portrait view.

Unless there's something special, the assembly specifications of this module won't mention the specifics of building the views of this application for the reason above.

1.3.3 *Prototype Code*

The code in the prototype is a very good reference for the assemblers, especially for how each UI control is connected to the properties of each controller.

The classes of this application all have a corresponding existing namesake class in the prototype (if there's exception, it will be explicitly mentioned)

1.3.4 *Popping and Pushing of the Navigation Controller Stack*

Whenever a controller needs to navigate the user to another controller, it should push the new controller onto the stack like this:

```
[self.navigationController pushViewController:nextController  animated:YES];
```

For the "Back" button and "Menu" button shown in the storyboard, their handling methods should simply pop the current controller out of the stack like this:

```
[self.navigationController popViewControllerAnimated:YES];
```

The prototype code use NSNotification to indirectly perform the popping and pushing. This approach is NOT adopted.

1.3.5 *Navigation Bar and Back Button*

For all "Back" button, "Close", "Menu" button (except the "Menu" shown on the Incident Details page) shown in the storyboard, they are actually the leftBarButtonItem of the Navigation Bar of the current controller. See Updating the Navigation Bar from here for an introduction of the navigation bar. The prototype doesn't use navigation bar at all. This should NOT be adopted in this application.

Because we can't use the default back button (which shows the parent controller's title, instead of 'Menu' or 'Back'), we have to create our custom bar buttons.

Therefore for all pages that have a back button, their corresponding controller's viewDidLoad() method should create a UIBarButtonItem using an image, set its handling method, and assign it to self.navigationItem.leftBarButtonItem.

To display the title of each page as per storyboard, each controller's title property should be set appropriate when that controller is created, which is usually done in didSelectRowAtIndexPath() method of each controller because table view is heavily used.

1.3.6 *Use of Custom Table Cell*

This application follows the approach of the prototype to use custom table cell (inherited from UITableViewCell) to realize the custom layout of table cell.

Note that these custom table cells have their corresponding XIB files too. The assemblers can refer to these XIB files when building the portrait versions of these files.

1.3.7 *Data Loading*

For each page that will display dynamic data, its viewWillAppear() should be responsible for calling BCMSBackEndFacade to get the data. If the controller has a table view, this method should call the reloadData() of the table view.

1.3.8 *No Pagination*

Whenever a call is made to a method of BCMSBackEndFacade that supports pagination, pageSize should be set to -1 to tell the back-end service to return all data without pagination.

1.3.9 *Authentication Token*

Whenever there's the need of a call to BCMSBackEndFacade that requires an authentication token, the SessionContainer.getAuthToken() should be used.

1.3.10 *Perform Asynchronous Operation on Back End Services*

Whenever the application performs a call on BCMSBackEndFacade (except on refreshData()), it should be done at the back-ground, and at the same time a modal view should be shown to indicate that the operation is underway. After the back-ground operation completes, the modal view will be dismissed. See here for a tutorial on how to present a view controller modally.

In short, this can be done by the controller that performs the operation like this, using the Grand Central Dispatch of iOS:

1. Create a SpinnerModalViewController
2. Call self.presentModalViewController() passing in the created SpinnerModalViewController to show the modal view controller
3. Use dispatch_async() to execute the back-end operation like this:

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
    perform the back-end operation;
    dispatch_async(dispatch_get_main_queue(), ^{
        Call self's dismissModalViewControllerAnimated() to dismiss the modal
        view; // This step MUST be done in the main queue because UI
        components are not thread-safe.
    }
}
```

1.3.11 *SessionAwareController*

To implement session timeout, a SessionAwareController is defined. All controllers of this module should extend from this controller. See the Infrastructure Assembly for details of this class.

1.3.12 *Protocols to follow*

All controllers that have a table view should follow UITableViewDelegate and UITableViewDataSource protocols.

All controllers that have UITextField should follow UITextFieldDelegate protocol and the implement its textFieldShouldReturn() method to resign first responder properly.

1.3.13 *Table View*

The UI of this module uses UITableView heavily. Therefore it's essential for assemblers to fully understand this view in order to understand this module. See here for a tutorial on Table View.

For controllers that have a table view, the viewWillAppear() should always call the reloadData() of the table view.

1.3.14 *Threading*

The UI components of Cocoa touch are not thread-safe, but they are always accessed by the main event handling thread. The back-end services are thread-safe, so there's no thread-safety concern when calling them. The threads that run NSTimer and perform asynchronous operations in 1.3.10 are all managed by iOS safely, and there is no shared data between threads that could be written at the same time.

Therefore, this application is thread-safe to use.

1.3.15 *Init Method*

All classes of this module should have a parameter-less init method.

1.3.16 *Memory Management*

Since the version of iOS is 4+, we can't use the more advanced Automatic Reference Counting. Instead, programmatic retain and release should be used. See here for a quick tutorial. Basically, whoever creates an object through 'alloc' and 'init' should call a release on that object at the end when it's no longer used. Whoever 'retain' an object should also call a release on that object at the end when it's no longer used. The 'release' operations should be done in a 'dealloc' method of each class. The 'dealloc' method should call super class's dealloc after releasing like this;

```
-(void) dealloc {
    [label release];
    [message release];

    [super dealloc];
}
```

For all controllers of this class, a 'viewDidUnload' method should be implemented, which will set all its UI control properties to nil and call super class's namesake method like this:

```
- (void)viewDidUnload {
   self.label = nil;
   self.message = nil;
   [super viewDidUnload];
}
```

1.3.17 Persistence

There is no persistence in this module. The persistence is handled by the back-end services.

1.3.18 Transactions

There is no transaction in this module.

1.3.19 Configuration

All configurations will be done via a "config.plist" file. This file should include the following keys:

Key	**Value**
BCMConveneRoomService	The class name of the implementation of the protocol whose name is the key.
BCMHelpDocumentService	The class name of the implementation of the protocol whose name is the key.
BCMIncidentAttachmentService	The class name of the implementation of the protocol whose name is the key.
BCMIncidentCategoryService	The class name of the implementation of the protocol whose name is the key.
BCMIncidentNoteService	The class name of the implementation of the protocol whose name is the key.
BCMIncidentService	The class name of the implementation of the protocol whose name is the key.
BCMLookupService	The class name of the implementation of the protocol whose name is the key.
BCMUserService	The class name of the implementation of the protocol whose name is the key.
CacheRrefeshIntervalInSeconds	The interval of refreshing the cache in seconds.
SessionTimeoutInSeconds	The session timeout period in seconds. Default to 15 minutes (15 * 60 seconds)
TermOfUse	The text of Term of Use
PrivacyPolicies	The text of Privacy Policies
LegalNotices	The text of Legal Notices
ISM	The role name of ISM
BCMSAdministrator	The role name of BCMS Administrator
PlanWebApplicationURL	The URL of the Plan Web Application
WebApplicationURL	The URL of the BCMS Web Application

All configuration parameters are required.

The plist file can be loaded like this:

```
NSMutableDictionary* dict =  [ [ NSMutableDictionary alloc ]
initWithContentsOfFile:@"/config.plist" ];
```

1.3.20 Logging

NSLog should be used to perform logging, specifically:

- The system will log all the errors, exceptions, warning and debug information during the application execution.
- The following data will be logged in each method of the application:
  - Method name (String, max 100 chars, non empty),
  - Description of problem (String, max 1024 chars, non empty),
  - Input arguments (if any, String, max 4096 chars, can be empty),
  - Output results (if any, String, max 4096 chars, can be empty),
  - Date and timestamp (String, full date/time format, max 30 chars, non empty),
- Additionally, the application will log all successful and failed login attempts for all the users. The following data will be logged in addition to the regular logging information:
  - Username (String, max 50 chars, non empty),
  - Login result (String, max 7 chars, non empty: “success” or “fail”).

The logged information must not contain user passwords.

1.3.21 Exception Handling Overview

The BCMSBackEndFacade should throw the back-end services’ exceptions directly to the caller. These exceptions all have one common base class: ‘BCMServiceException’. So this module only needs to catch this exception. The information in the exception will be shown in a readable manner using NSAlert.

For errors occur in this module, there’s no need to throw any exception. The module simply needs to use NSAlert to show a readable error message.

The alert can be shown using NSAlert like this:

```
NSAlert *alert = [[NSAlert alloc] init]

[alert setMessageText:@"Hi there."];

[alert runModal];
```

After showing the alert, the module should stop doing the current operation.

1.3.22 Internationalization

There is no internationalization requirement.

1.3.23 Auditing

None. The auditing is done at the server side.

1.3.24 Scalability

There are no specific scalability requirements.

1.3.25 Security

Data Security
The password is saved safely in the local device if the user chooses to. See 1.2.5.1 of Infrastructure Assembly for details.

Session Timeout
Automatic session timeout is implemented. See 1.2.5.4 of Infrastructure Assembly for details.

Authorization

As per ARS 3.3.1.2, the following operations can only be performed by ISM and BCMS Administrator roles:
Create New Incident
Update Incident's Additional Information
Upload Incident's Attachment
Delete Incident's Attachment

Therefore the related controller should check programmatically if the privilege from SessionContainer.getPrivilege() matches the 2 role names. The 2 role names are available from plist under key "ISM" and "BCMSAdministrator". If the user's privilege isn't one of these roles and he/she wants to perform an operation above, a NSAlert should be shown to inform the user of the lack of authority.

1.3.26 GUI Requirements

Please refer to ARS 3.1 for GUI requirements.

1.3.27 Exotic Technology

Cocoa touch and objective-c can be considered unfamiliar technology at TopCoder. So there is a risk of developing with them.

1.3.28 Existing System

There is no existing system to build upon.

1.4 Deployment Constraints

The finally integrated module is one XCode project, which doesn't need to be made into a deployable package. As long as it can be run on the simulator, it's fine.

1.4.1 Environment

- Objective-C 2.0
- iOS 4+
- Cocoa Touch
- XCode 4.2

1.5 Development Standards:

The assembly solution must adhere to the guidelines as outlined here: http://apps.topcoder.com/wiki/display/tc/Assembly+Competition+Tutorial

1.6 Interfaces Classes Overview

See the TCUML file for all classes.

2. User Interface

The various Cocoa touch UI components for the user interface, developed by XCode and Interface Builder.

3. Included Documentation

3.1 Architecture Documentation

- Interface Diagrams
- Sequence Diagrams
- Application Design Specification
- Assembly Specification