

Tutorial for Developing Mobile Application based MVC model

1. Overview

This is a tutorial for developing a client-server mobile application in Android using MVC framework. In this tutorial I propose a general cross-platform MVC framework which can be implemented for specific mobile platform such as Android, Iphone, Window Phone as [Figure 1](#). Currently, the framework has been implemented for both mobile platforms Android and Iphone. This tutorial will introduce to you about the general cross-platform MVC framework for developing client-server mobile application. Besides that, it guides you how to implement an Android application using this framework. The tutorial includes seven sections: Overview, Target Audience, How to install, Introduction framework, Design framework, Demo using framework, Your practice and survey.

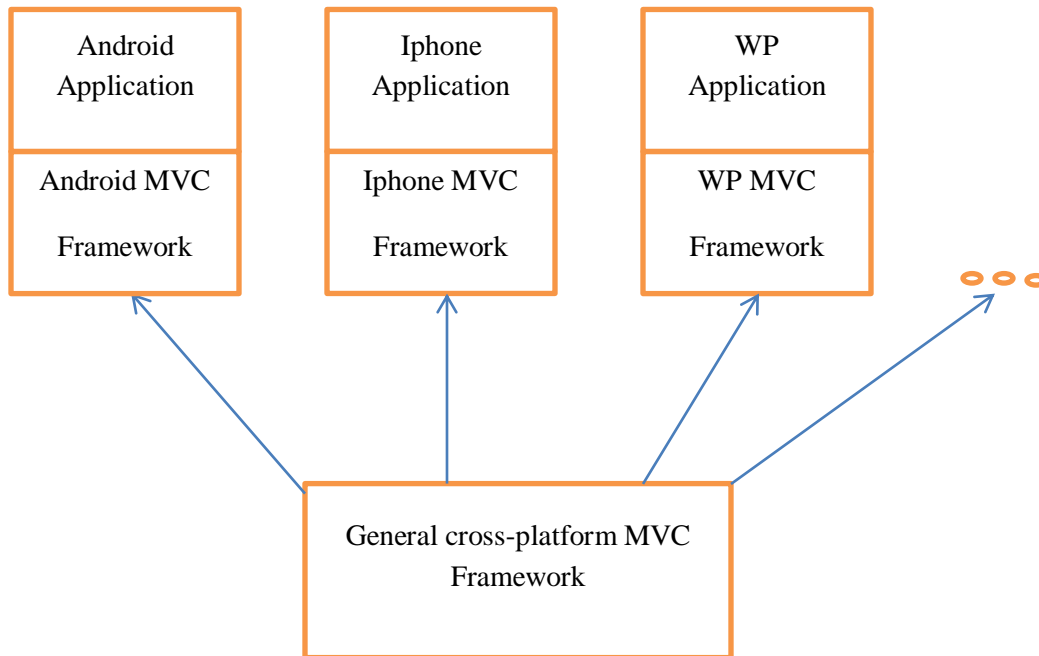


Figure 1 General cross-platform mobile application

2. Target Audience

This tutorial is directed to any developer interested in learning how to develop a mobile application using MVC framework in design.

You need to know how to program in Java and use Eclipse. Also, you should be familiar with building Android mobile apps. A basic knowledge of MVC model is also required.

3. How to install framework and application demo.

Make sure you have installed the following tools and packages for the tutorial:

Eclipse

- **Eclipse.** Install the tool from [here](#).

Android

- **Android Development Tools (ADT).** Install the Android tools (plugin) for the Eclipse IDE by following the steps described here: [Installing the Eclipse Plugin](#).

From within Eclipse, open the Android SDK manager to install the latest SDK platform tools. It is recommended that you install the latest Google API. The example shown in this tutorial was verified using API level 8.

Installing framework and application demo

- Source code for Android MVC framework can be downloaded [here](#). It includes two folders.
 - MVC_Android_Framework: Framework MVC for developing Android mobile application.
 - Demo_MVC_Android: A application demo using MVC_Android_Framework as a library.
- After downloading the source code, in eclipse, we choose File->Import to import MVC_Android_Framework and Demo_MVC_Android sequentially. Then, right-click project Demo_MVC_Android, chooses Properties-> Android, then add project MVC_Android_Framework as reference ([Figure 2](#))
- Run Demo_MVC_Android application.

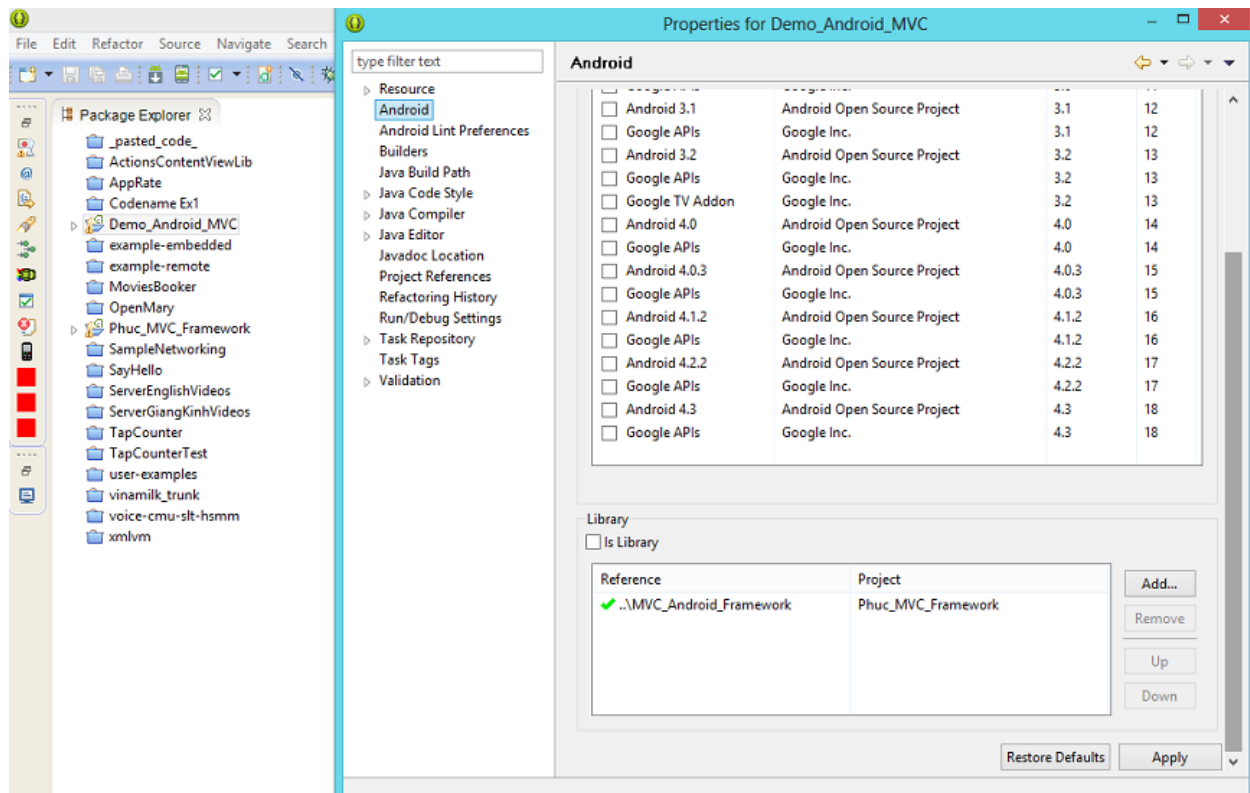


Figure 2 Reference project

4. Introduction framework

To understand deeply about source code of MVC_Android_Framework, firstly we need to learn about the general architecture to implement framework in [Figure 3](#).

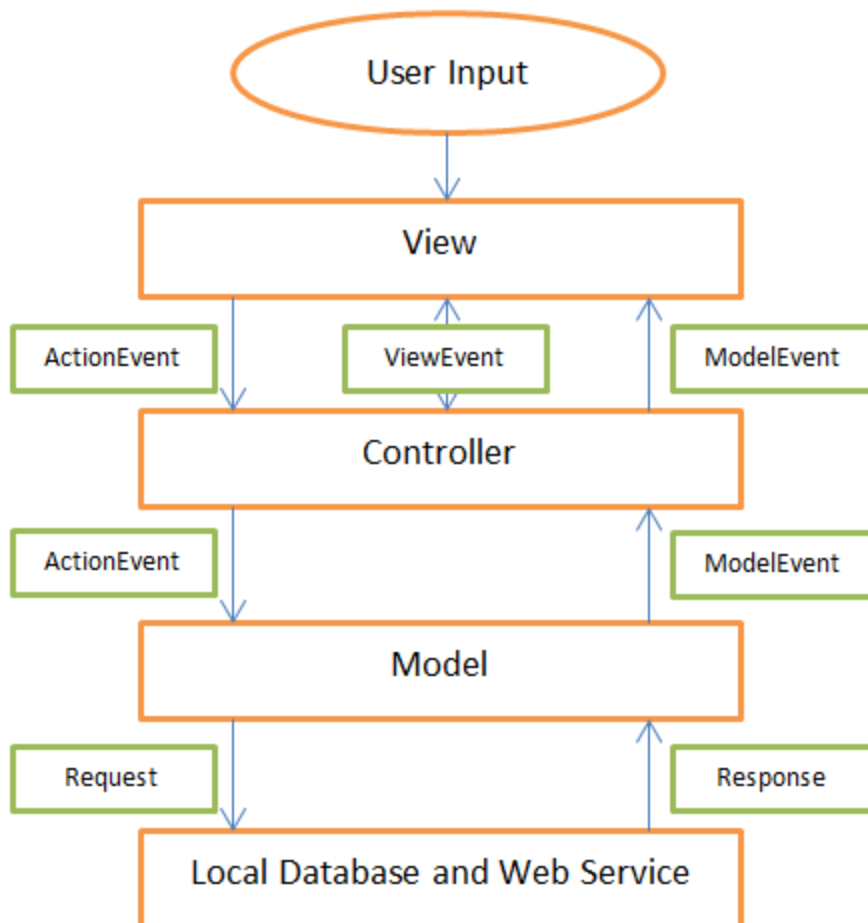


Figure 3: The generic model based MVC for mobile application

In this model, we see that there is obvious separation between four layers View, Controller, Model, and Database.

View: receives user input, validate user input and then create action event to send to controller layer.

Controller: mediates to receive ActionEvent from view and send to Model, besides that controller also receives view events which handle events between view objects such as navigating views.

Model: receive ActionEvent from controller, and generate a request to retrieve or update database. When receiving response, it creates an object called ModelEvent to pass by controller.

Database: In this model, database plays role as resource, it can be local database as well as web service. It receive request from model, handle request and return response for model.

5. Design framework

Based on the general model MVC above, I designed a class diagram in [Figure 4](#).

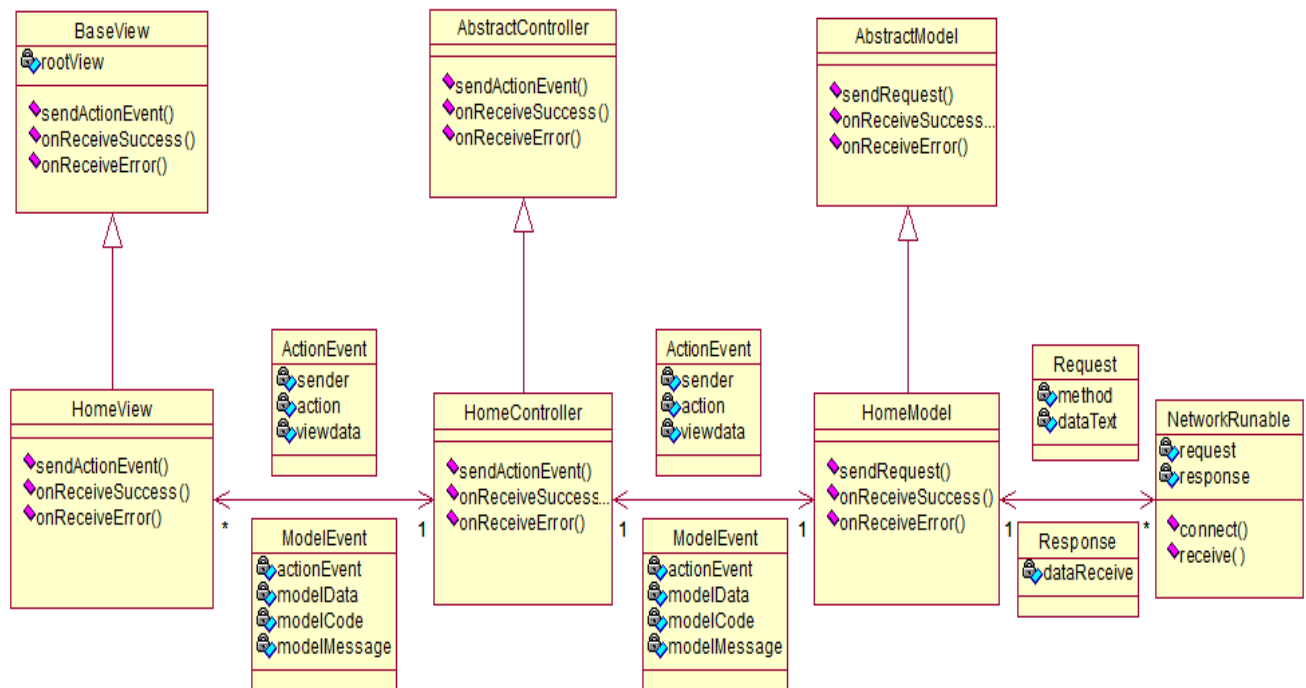


Figure 4 : The class diagram for general MVC model

As can be seen from the class diagram, each layer has its base class which enhances reusability and extensibility. In addition, there is a data flow implemented by data objects as ActionEvent, ModelEvent, Request, and Response run sequentially through layers in bi-direction.

In the diagram, HomeView, HomeController, HomeModel are the three of implementation of the three main classes. And we will describe in more detail about the processing flow of this design. Firstly, HomeView receives the user input through interface, then create a ActionEvent object and send to HomeController through HomeController static instance. The ActionEvent object includes important attributes such as sender (in case is HomeView), action to distinguish requests (for example, request login or signup...), viewdata stores parameter values from user input. Next, HomeController receives ActionEvent object, then it forwards to HomeModel through HomeModel static instance. The responsibility of HomeModel is to create a request from ActionEvent object and send to server through a thread called NetworkRunnable which is

independent with main thread. After the response is received from server in this thread, it will send to HomeModel through a listener. A ModelEvent object will be created at HomeModel with some added information such as modelData stores data from server, modelCode to know a request is success or fail, modelMessage stores a message from server. In case response is success, HomeModel sends ModelEvent to HomeController static instance through onReceiveSuccess, otherwise onReceiveError. Finally, HomeController also sends ModelEvent to HomeView through sender variable in ActionEvent object to update data for UI. The [Figure 5](#) The sequence diagram of the proposed model based MVC show the processing flow of this model for doing a request.

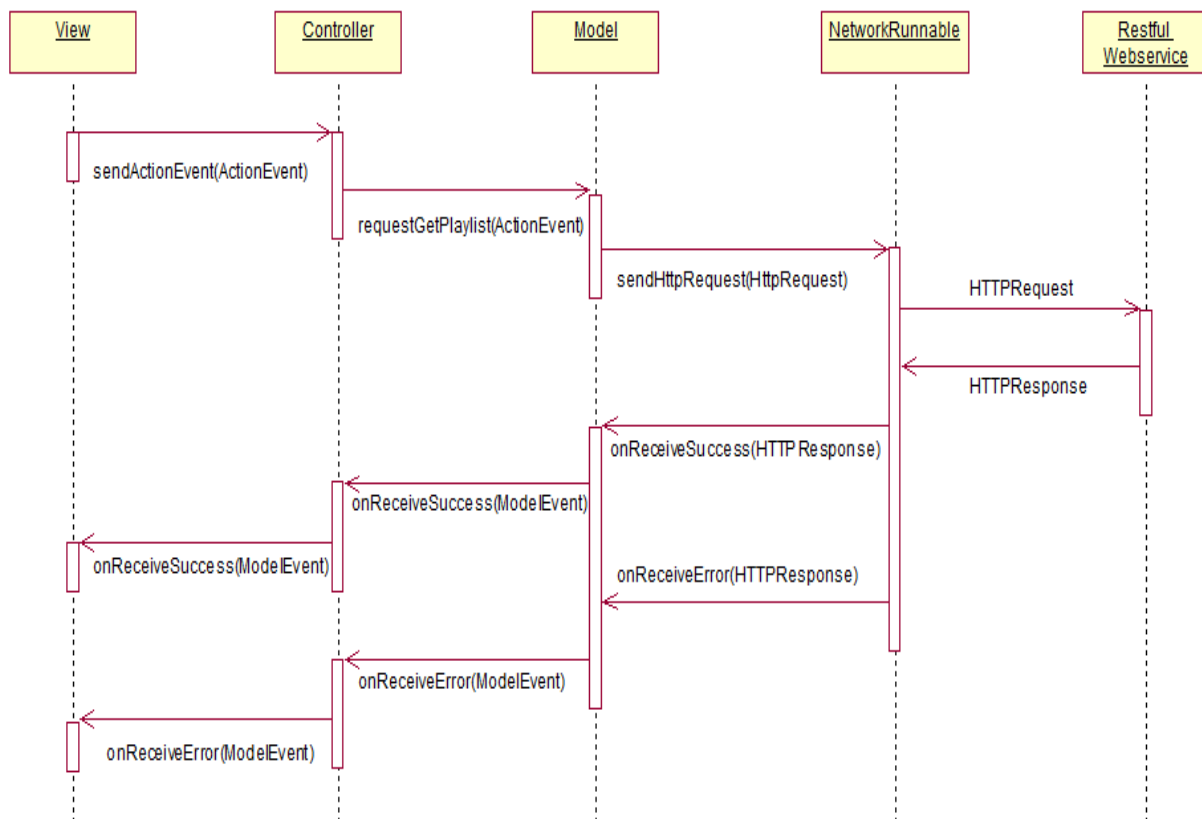


Figure 5 The sequence diagram of the proposed model based MVC

For implementing the complex application with many features and many requests, the framework can be scaled as [Figure 6](#). The view can obtain many requests, a controller manages many views, a controller has a corresponding model. With the complex application, we need to separate controllers for managing the certain features.

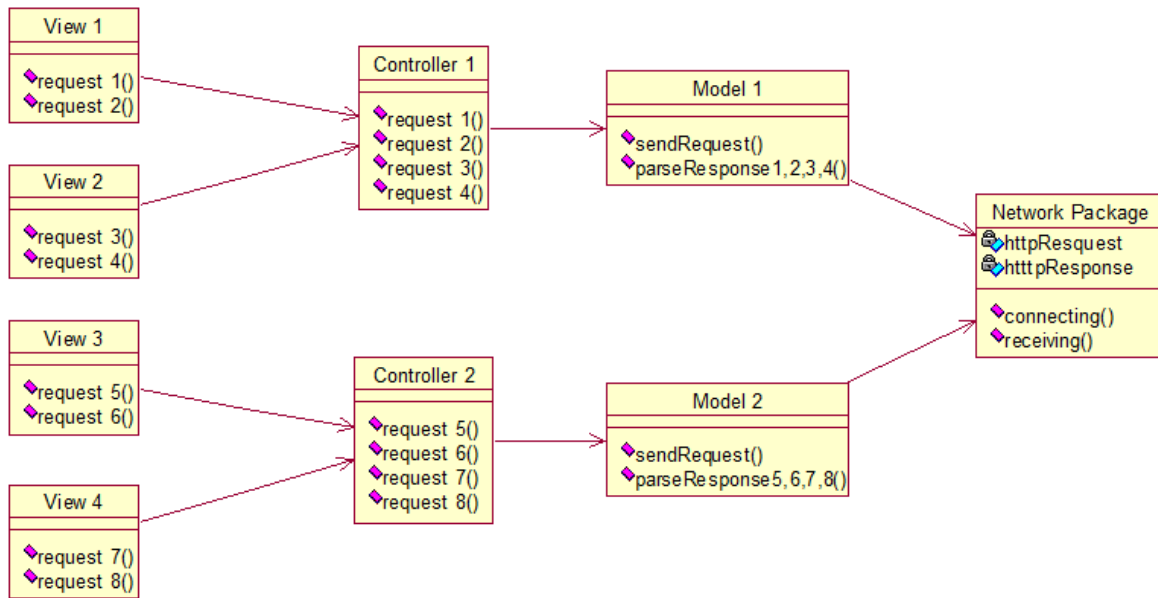


Figure 6 The structure of application using the proposed framework.

6. Demo using framework

























a. Problem:

And now, we use the framework above to develop an application called English Videos By Youtube. The application will display videos links of Youtube categorized by playlists, and categories. The application includes three screens:

YoutubeView: The first screen from the left, list categories of application.

PlaylistView: The middle screen is displayed when we click one of category items, lists playlist of the chosen category.

PlaylistItemView: The rightest screen is displayed when we click one of playlist items, list videos of the chosen playlist.

| Videos English | | Videos English |
|---|--|---|
| PRACTICE EXAMINATION | | IELTS Training |
|  IELTS |  IELTS Training |  Improve English for the IELTS |
|  TOEIC |  IELTS ضع الاليتس في جي ب ك |  Improve English for the IELTS |
|  TOEFL |  BARRON'S IELTS practice exams |  IELTS Pronunciation - Improve English |
| PRACTICE SKILLS | | |
|  SPEAKING |  IELTS Listening Section |  Improve English for the IELTS |
|  LISTENING |  Series 1. IELTS Reading Questions |  IELTS Writing: Grammar - improve |
|  READING |  IELTS Reading Section |  Improve English for the IELTS |
|  WRITING |  IELTS Speaking Exam |  IELTS Reading: Improve your English |
|  GRAMMAR |  Study English - IELTS Preparation DVD1 |  IELTS Writing: Task response/Task |

You can run this project with Demo_Android_MVC source

b. Analysis:

This section shows the steps for using the proposed framework to develop this application in both platforms Android and Iphone in one development process.

As the general MVC model in [Figure 3](#), we need to build four layers for this application includes View, Controller, Model, Webservice. Therefore, the first task is to build a Webservice with published APIs. In this case, to make the problem simpler, we access to Youtube Webservice with available APIs as following table.

Table 1 Youtube API requests

| Request | Datasource |
|-----------------------|------------------------------------|
| requestListCategories | The temporary data. |
| requestGetPlaylist | Youtube API with the link as below |

| | |
|------------------------|---|
| | https://www.googleapis.com/youtube/v3/search?part=snippet&q=ielts&type=playlist&key={YOUR_API_KEY} |
| requestGetPlaylistItem | Youtube API with the link as below https://www.googleapis.com/youtube/v3/playlistItems?part=snippet&playlistId=PLE1F8BA159C48CBCF&key={YOUR_API_KEY} |

After developing Webservice and designing common screens finished, we will implement mobile application using the above framework. This step will be mentioned more detail in next section.

c. **Implement:**

Based on the class diagram in [Figure 4](#), implementing an application using framework is overriding functions from super class. In fact, the framework provides available classes, packages such as BaseView, AbstractController, AbstractModel, Network package, Data Transfer Objects (ActionEvent and ModelEvent), Json library for parsing data, and other classes. You can review more detail in source code “MVC_Android_Framework”... With the available framework, it is not difficult for us to implement an application using framework. In detail, as in this demo application which includes three screens with three APIs requests connecting to Webservice. With this framework, for each view, we will just implement some classes. For YoutubeView screen, we will implement YoutubeView, YoutubeController, YoutubeModel, and ListCategoryDTO. Other screens can use the same controller and model. The following sections will explain more detail about these classes.

❖ **View:**

As the class diagram [Figure 4](#), in this layer we have to implement common functions as following:

| Functions | Class | Description |
|------------------|---------------------|--|
| sendActionEvent | YoutubeView(Demo) | Create ActionEvent and forward it to Controller |
| onReceiveSuccess | YoutubeView(Demo) | Handle receiving response from server in success |
| onReceiveError | BaseView(Framework) | Handle receiving response from server in error |

Firstly, we can review the common function `onReceiveError` which is implemented in base class `BaseView` because it handle errors for all views such as common errors, error no connection, error object not exist... [Figure 7]

```
public void onReceiveError(ModelEvent modelEvent) {
    switch (modelEvent.getModelCode()) {
        case ErrorConstants.ERROR_COMMON:
            closeProgressDialog();
            showDialog(StringUtil.getString(R.string.ERROR_COMMON));
            break;
        case ErrorConstants.ERROR_NO_CONNECTION:
            showDialog(modelEvent.getModelMessage());
            break;
        case ErrorConstants.ERROR_OBJECT_NOT_EXISTS:
            showDialog(StringUtil.getString(
                R.string.ERROR_OBJECT_NOT_EXISTED));
            break;
        default:
            closeProgressDialog();
            if (!StringUtil.isNullOrEmpty(modelEvent.getModelMessage())) {
                showDialog(modelEvent.getModelMessage());
            }
            break;
    }
}
```

Figure 7 Handling errors in BaseView

After that, we implement views such as `YoutubeView`, `PlaylistView`. Each view must send request via `sendActionEvent()`, and update UI with the returned data via `onReceiveSuccess`, and handle error via `onReceiveError()`. As mentioned above, we will override `onReceiveError` from `BaseView` which is implemented in framework. And we just implement two basic functions `sendActionEvent()`, and `onReceiveSuccess()`.

The following code in Figure 8 is an example of `sendActionEvent()` for request playlists by the chosen category. An `ActionEvent` object includes basic fields such as *action* as an identifier of an `ActionEvent`, *viewData* includes name and value of parameters, *sender* is view object to identify who sent this `ActionEvent`.

```

public void sendActionEvent() {
    Vector<String> vt = new Vector<String>();
    vt.add(IntentConstants.INTENT_CATEGORY);
    vt.add(category);
    vt.add(IntentConstants.INTENT_KEY);
    vt.add(ServerPath.KEY);
    vt.add(IntentConstants.INTENT_PART);
    vt.add("snippet");
    vt.add(IntentConstants.INTENT_TYPE);
    vt.add("playlist");
    vt.add(IntentConstants.MAX_RESULTS);
    vt.add("20");
    ActionEvent e = new ActionEvent();
    e.action = ActionEventConstant.GET_LIST_PLAYLIST;
    e.viewData = vt;
    e.sender = PlaylistView.this;
    YoutubeController.getInstance().sendActionEvent(e);
}

```

Figure 8 SendActionEvent in View

And the [Figure 9](#) is the same code for update UI for the returned data via onReceiveSuccess. The returned data is accessed from function getModelData of ModelEvent object. In this case, the data is filled in listview.

```

@Override
public void onReceiveSuccess(ModelEvent modelEvent) {
    // TODO Auto-generated method stub
    ActionEvent e = modelEvent.getActionEvent();
    switch (e.action) {
        case ActionEventConstant.GET_LIST_PLAYLIST:
            displayData(modelEvent);
            break;
        default:
            break;
    }
}

@SuppressWarnings("unchecked")
public void displayData(ModelEvent modelEvent) {
    // TODO Auto-generated method stub
    playList = (List<PlaylistDTO>)modelEvent.getModelData();
    arrayAdapter = new StandardArrayAdapter(playList);
    listView.setAdapter(arrayAdapter);
}

@Override
public void onReceiveError(ModelEvent modelEvent) {
    // TODO Auto-generated method stub
    super.onReceiveError(modelEvent);
}

```

Figure 9 Update data for UI in View

❖ Controller:

In this layer, we must implement common functions as following:

| Functions | Class | Description |
|------------------|--------------------------------|--|
| sendActionEvent | YoutubeController(Demo) | Receive ActionEvent from View and forward to Model |
| onReceiveSuccess | AbstractController (Framework) | Handle receiving response from server in success |
| onReceiveError | AbstractController(Framework) | Handle receiving response from server in error |
| handleSwitchView | YoutubeController(Demo) | Handle switching views in application. |

In AbstractController, we can review two common functions onReceiveSuccess and onReceiveError. In onReceiveSuccess, we has field *e.sender* determines which object view is received data. As the request is executed in thread independent with UI thread, so updating data must be switch to UI thread as [Figure 10](#). And onReceiveError also handle similarly, however, we can log the errors to server in this function depending on our need.

```

public void onReceiveSuccess(final ModelEvent modelEvent) {
    final ActionEvent e = modelEvent.getActionEvent();
    BaseActivity ac = (BaseActivity) e.sender;
    HTTPRequest request = e.request;
    if (ac.isFinished) {
        return;
    }
    ac.runOnUiThread(new Runnable() {
        @Override
        public void run() {
            BaseActivity base = (BaseActivity) e.sender;
            if (base != null) {
                base.onReceiveSuccess(modelEvent);
            }
        }
    });
}

```

Figure 10: onReceiveSuccess in AbstractController

In application demo, one instance controller YoutubeController inherited from AbstractController is initialized by Single Instance design pattern (Figure 11). It makes easier to be invoked by other calls.

```

static HomeController instance;
protected HomeController() {
}

public static HomeController getInstance() {
    if (instance == null) {
        instance = new HomeController();
    }
    return instance;
}

```

Figure 11: Single Instance of Controller

The YoutubeController receives ActionEvents and forward them to model via instance of model. The ActionEvents sent from views are distinguished by field *action*. as Figure 12:

```

public void sendActionEvent(ActionEvent e) {

    String method = "";
    switch (e.action) {
        case ActionEventConstant.GET_LIST_CATEGORY:
            method = "getListCategory";
            break;
        case ActionEventConstant.GET_LIST_PLAYLIST:
            method = "search";
            break;
        case ActionEventConstant.GET_LIST_PLAYLIST_ITEM:
            method = "playlistItems";
            break;
    }

    try {
        Vector<String> info = (Vector<String>) e.viewData;
        String url = ServerPath.SERVER_PATH
            + NetworkUtil.createStringURL(method, info);
        HomeModel.getInstance().sendHttpRequest(url, e);
    } catch (Exception e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
}

```

Figure 12: SendActionEvent in Controller

In addition, the controller also handles switching views in application.

```

@Override
public void handleSwitchView(ActionEvent e) {
    // TODO Auto-generated method stub
    Activity base = (Activity) e.sender;
    Intent intent;
    Bundle extras;
    switch (e.action) {
        case ActionEventConstant.GOTO_PLAY_LIST:
            intent = new Intent(base, PlaylistView.class);
            extras = (Bundle) e.viewData;
            intent.putExtras(extras);
            base.startActivity(intent);
            break;
        case ActionEventConstant.GOTO_PLAY_LIST_ITEM:
            intent = new Intent(base, PlaylistItemView.class);
            extras = (Bundle) e.viewData;
            intent.putExtras(extras);
            base.startActivity(intent);
            break;
        default:
            break;
    }
}

```

❖ **Model:**

As the [Figure 4](#), we need to implement following functions as following table:

| Functions | Class | Description |
|------------------|--------------------------|---|
| sendHttpRequest | AbstractModel(Framework) | Create request with URL and forward it to an independent thread connecting to restful web service |
| onReceiveSuccess | YoutubeModel(Demo) | Handle receiving response from server in success |
| onReceiveError | YoutubeModel(Demo) | Handle receiving response from server in error |

Firstly, you can review the function `sendHttpRequest` in `AbstractModel` which creates `HttpRequest` object and forwards it to an independent thread called `HttpAsyncTask` as [Figure 13](#).

```
public HttpRequest sendHttpRequest(String url, ActionEvent actionEvent) {  
  
    HttpRequest re = new HttpRequest();  
    re.setUrl(url);  
    re.setAction(actionEvent.action);  
    re.setContentType(HTTPMessage.CONTENT_JSON);  
    re.setMethodType(Constants.HTTPCONNECTION_GET);  
    re.setObserver(this);  
    re.setUserData(actionEvent);  
    new HttpAsyncTask(re).execute();  
    return re;  
}
```

[Figure 13](#): `SendHttpRequest` in `AbstractModel`

Then, we implement an instance `YoutubeModel` which inherits from `AbstractModel` with following functions:

After receiving the response from an independent thread, the `YoutubeModel` handle the response in `onReceiveSuccess` in success or `onReceiveError` in error.

For `onReceiveSuccess`: The server responses to the client `DataText` with `Json` format in `HTTPMessage` object. We need to parse `JsonFormat` to `Data Transfer Object PlaylistDTO` to keep this data. Then it forwards to `YoutubeController` as code below.

```

public void onReceiveSuccess(HTTPMessage mes) {
    ActionEvent actionEvent = (ActionEvent) mes.getUserData();
    ModelEvent model = new ModelEvent();
    model.setDataText(mes.getDataText());
    model.setCode(mes.getCode());
    model.setParams(((HTTPResponse) mes).getRequest().getDataText());
    model.setActionEvent(actionEvent);
    if (StringUtil.isNullOrEmpty((String) mes.getDataText())) {
        model.setModelCode(ErrorConstants.ERROR_COMMON);
        HomeController.getInstance().onReceiveError(model);
        return;
    }
    JSONObject json;
    switch (mes.getAction()) {
case ActionEventConstant.GET_LIST_PLAYLIST:
        try {
            json = new JSONObject((String) mes.getDataText());
            JSONObject result = json.getJSONObject("error");

            if (result == null) {
                JSONArray response = json.getJSONArray("items");
                List<PlaylistDTO> list = PlaylistDTO.parseListPlayList(response);
                model.setModelData(list);
                HomeController.getInstance().onReceiveSuccess(model);
            } else {
                model.setModelMessage(result.getString("message"));
                HomeController.getInstance().onReceiveError(model);
            }
        } catch (Exception ex) {
            model.setModelCode(ErrorConstants.ERROR_COMMON);
            HomeController.getInstance().onReceiveError(model);
        }
        break;
    }
}

```

For onReceiveError: Usually the error is no connection or timeout, we need to response error for YoutubeController.

```

public void onReceiveError(HTTPResponse response) {
    ActionEvent actionEvent = (ActionEvent) response.getUserData();
    ModelEvent model = new ModelEvent();
    model.setDataText(response.getDataText());
    model.setParams(((HTTPResponse) response).getRequest().getDataText());
    model.setActionEvent(actionEvent);
    model.setModelCode(ErrorConstants.ERROR_NO_CONNECTION);
    model.setModelMessage(response.getErrorMessage());
    HomeController.getInstance().onReceiveError(model);
}

```


7. Your Practice and Survey

This section gives some case studies for learning and using the proposed framework for mobile MVC which introduced in the previous sections and source code.

To understand the framework clearly and answer the questions in survey, you need to finish the following case studies in this document. Based on the design above and the source code

Demo_Android_Framework, you will implement some simple tasks and answer the questions in survey [here](#).

a) Creating PlaylistItems view:

In Demo_Android_Framework project, it includes the sample Playlist view which list playlists requesting from the requestGetPlaylist as in Table 1 Youtube API requests . The request is implemented through three layers View->Controller->Model follow steps presented as section Demo using framework-> Implement and in Demo source code. And now, similarly you need to make a requestGetPlaylistItems for PlaylistItems view with steps like Playlist view.

b) Evaluating about the ability to discover and fix errors:

It supposes that there is a change of JSON response structure of the requestGetPlaylist. How to modify and fix this error?

c) Extending features:

Currently, the application only makes requests from Youtube Webservice to get video links. But now, it supposes that you want to make requests from other Webservices such as DailyMotion Video to get more video links ([API Requests](#)). Based on the demo application, how do you extend the application to get more video links from other Webservices?

d) Evaluating the cross-platform ability:

Please you review the general framework [Figure 3: The generic model based MVC for mobile application](#), the class diagram [Figure 4 : The class diagram for general MVC model](#) and the source code of framework again to evaluate the cross-platform ability. It supposed that the framework will be implemented on other platforms such as iOS and Window Phone. Is it easy for you to implement this application demo to iOS and Window Phone? Ofcourse you can also know a little bit about programming iOS and Window Phone.