Tutorial for Developing Mobile Application based MVC model

1. Overview

For the thesis with research question is "Develop a generic cross-platform approach for developing client-server mobile application using a generic MVC-based model in design". In detail, I will propose a general cross-platform MVC framework which is implemented for specific mobile platform as **Figure 1** General cross-platform mobile application. Currently, I proposed a general cross-platform MVC framework, implemented it on Android and made an application demo using this framework. And I want to evaluate whether the framework can be considered as a generic cross-platform approach for developing client-server mobile application.

This document is a tutorial for developing a client-server mobile application using this proposed framework. It includes seven sections: Motivation, Target Audience, How to install, Introduction framework, Design framework, Demo using framework, and Survey. And you can also review the framework more detail in attached source code.

After all, please help me to answer questions in survey via the following link
https://docs.google.com/forms/d/1eDVccBL4Rq-B8wVpk6k3r8oLl2cx05O9fdtTJNsqJZo/viewform

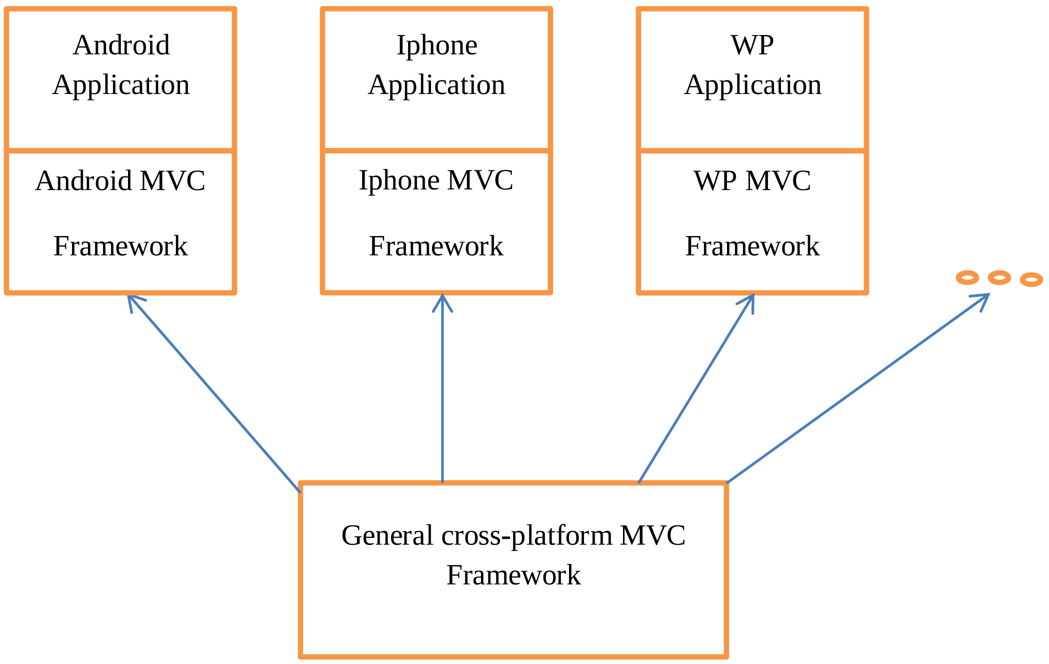


Figure 1 General cross-platform mobile application

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 technology 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.
- In Eclipse, we choose File-Import to import two projects sequentially. Then, right-click project Demo_MVC_Android, choose Properties-> Android, then add project MVC_Android_Framework as reference.
- Run Demo_MVC_Android application.

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 2: The generic model based MVC for mobile application**.

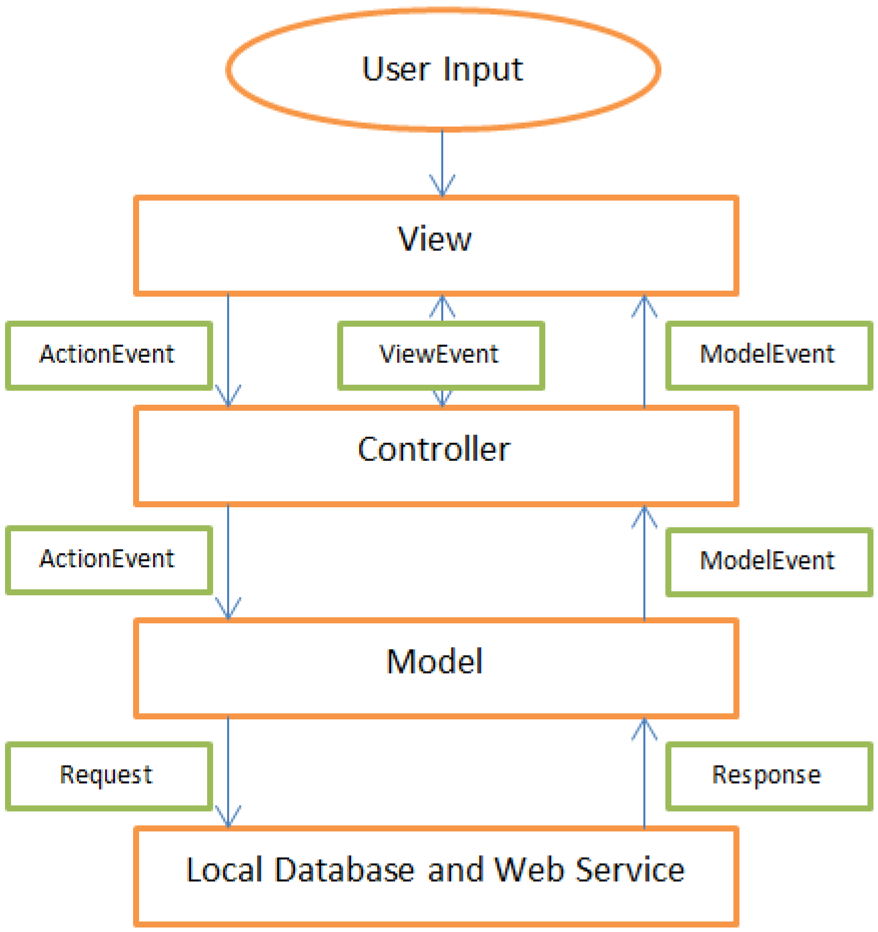


Figure 2: The generic model based MVC for mobile application

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, we design a class diagram in **Figure 3**.

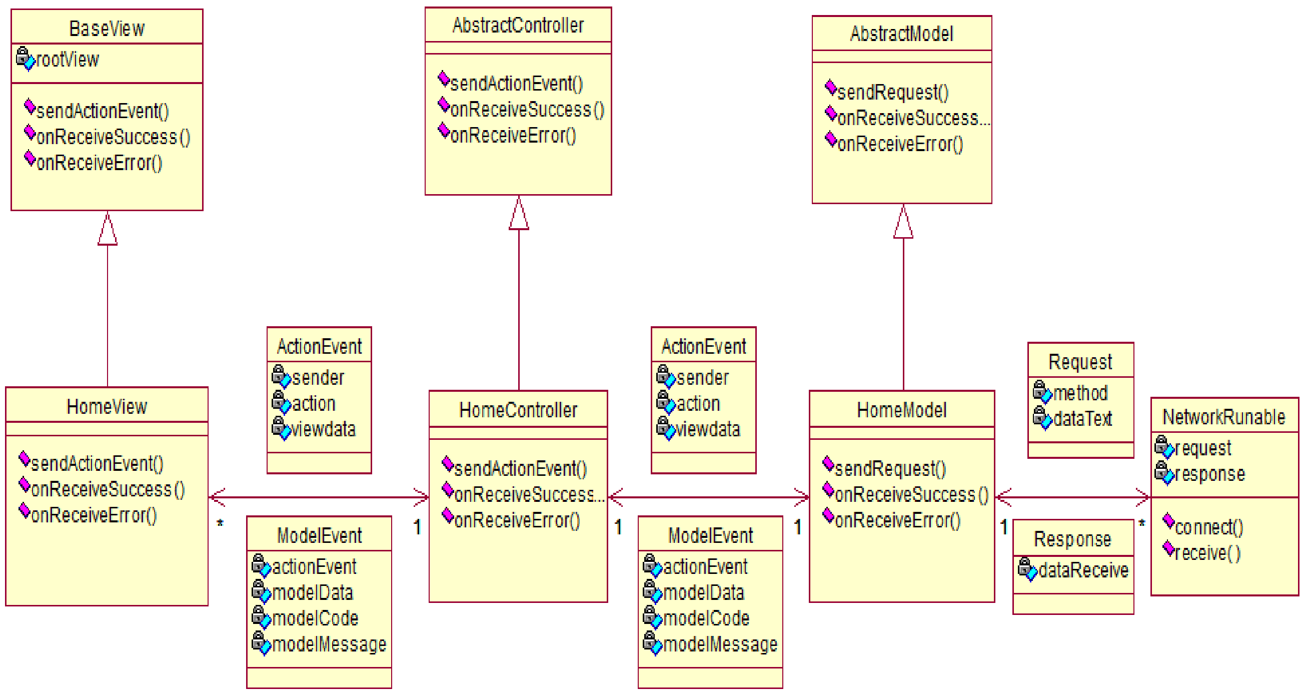


Figure 3 : The class diagram for general MVC model

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 will be 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, ModelEvent 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 4** The sequence diagram of the proposed model based MVC show the processing flow of this model for doing a request.

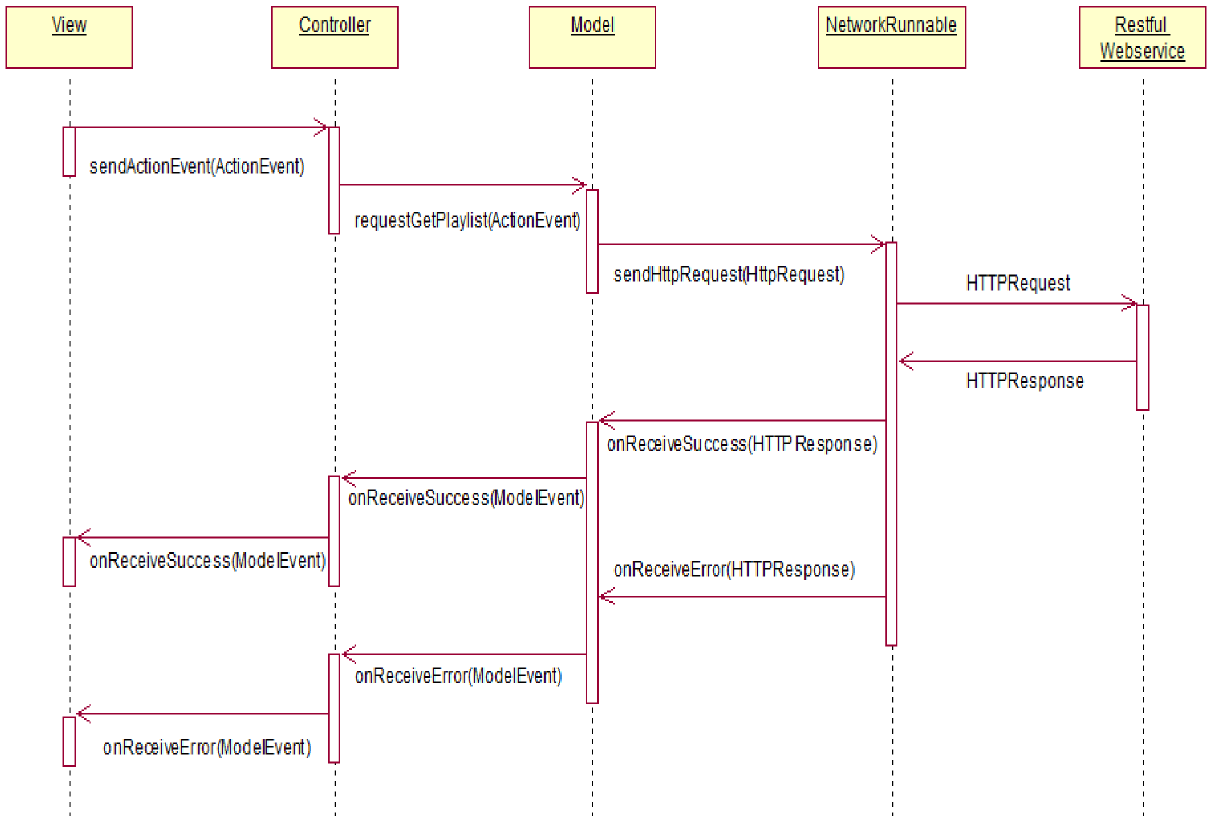

View
Controller
Model
NetworkRunnable
Restful Webservice
sendActionEvent(ActionEvent)
requestGetPlaylist(ActionEvent)
sendHttpRequest(HttpRequest)
HTTPRequest
HTTPResponse
onReceiveSuccess(HTTPResponse)
onReceiveSuccess(ModelEvent)
onReceiveSuccess(ModelEvent)
onReceiveError(HTTPResponse)
onReceiveError(ModelEvent)
onReceiveError(ModelEvent)

Figure 4 The sequence diagram of the proposed model based MVC

6. Demo using framework

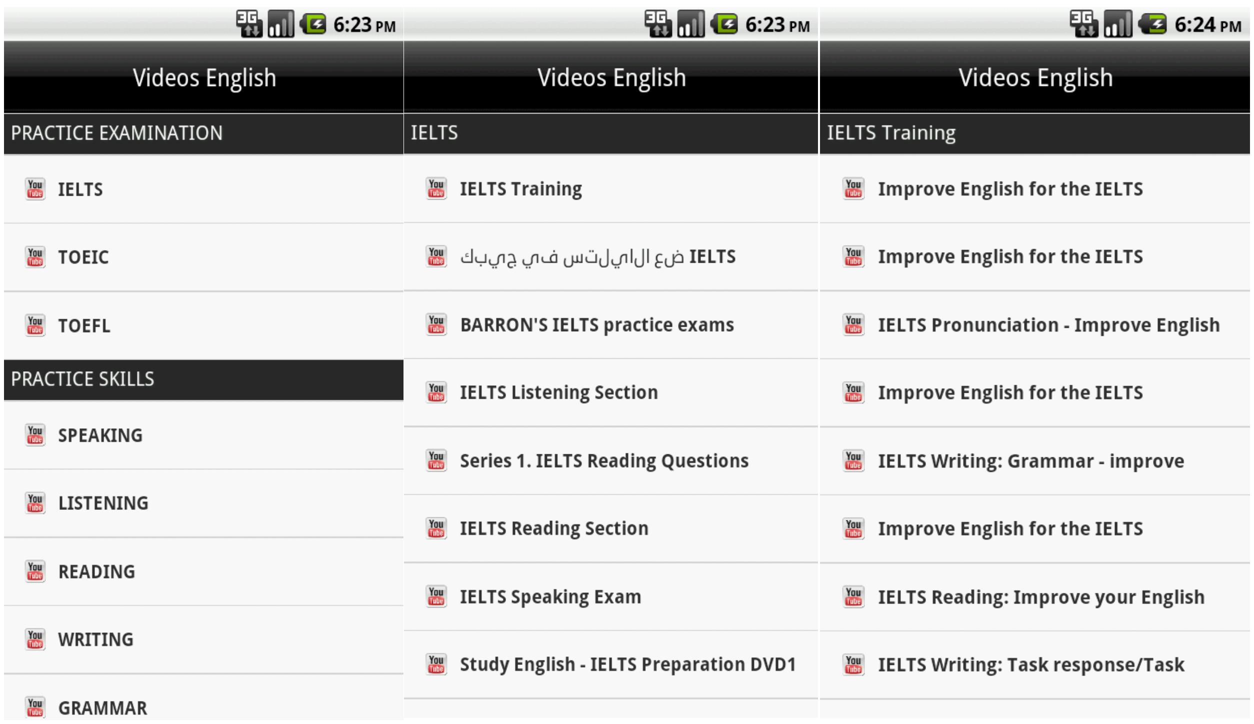
a. Problem:

And now, we use the framework above to develop an application called English Videos By Youtube on two both mobile platforms Android and Iphone in one development process. The application will display videos links of Youtube categorized by playlists, and categories. The application includes three screens:

HomeView: 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.



6:23 PM

Videos English

PRACTICE EXAMINATION

- IELTS
- TOEIC
- TOEFL

PRACTICE SKILLS

- SPEAKING
- LISTENING
- READING
- WRITING
- GRAMMAR

6:23 PM

Videos English

IELTS

- IELTS Training
- IELTS ضع الايلتس في جيبك
- BARRON'S IELTS practice exams
- IELTS Listening Section
- Series 1. IELTS Reading Questions
- IELTS Reading Section
- IELTS Speaking Exam
- Study English - IELTS Preparation DVD1

6:24 PM

Videos English

IELTS Training

- Improve English for the IELTS
- Improve English for the IELTS
- IELTS Pronunciation - Improve English
- Improve English for the IELTS
- IELTS Writing: Grammar - improve
- Improve English for the IELTS
- IELTS Reading: Improve your English
- 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 2**, 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.

Request	Datasource
requestListCategories	The data file from server.
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}

To reduce the time for development, the designer will design common screens for both platforms while developing Webservice.

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:**

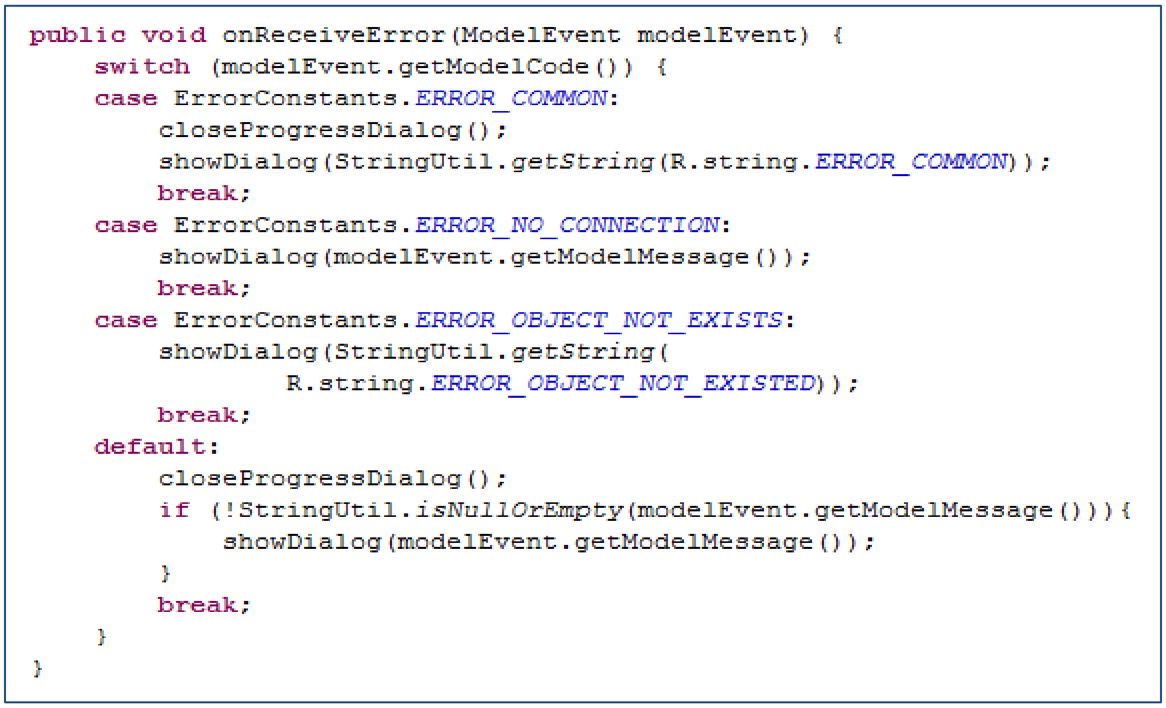
Based on the class diagram in **Figure 3**, implementing an application using framework is overriding functions from super class. In fact, the framework provides available classes, packages such as BaseView, AbstractController, AbstractModel, Network package, Data Transfer Objects (ActionEvent and ModelEvent), 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 HomeView screen, we will implement HomeView, HomeController, HomeModel, and ListCategoryDTO. The following sections will explain more detail about these classes.

- **View:**

As the class diagram Figure 3, in this layer we have to implement common functions as following:

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

Firstly, we must implement the common function onReceiveError in base class BaseView because it handle errors for all views such as common errors, error no connection, error object not exist... [Figure 5]

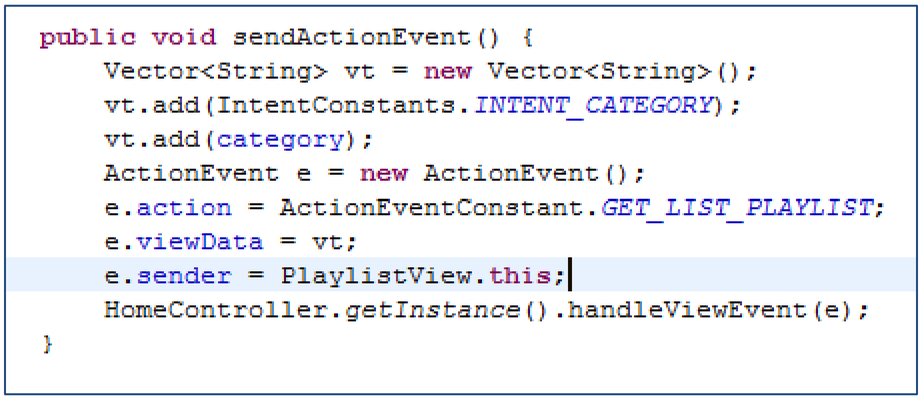


```
public void onReceiveError(ModelEvent modelEvent) {
    switch (modelEvent.getModelCode()) {
    case ErrorConstants.ERROR_COMMON:
        closeProgressDialog();
        showDialog(StringUtil.getString(R.string.ERROR_COMMON));
        break;
    case ErrorConstants.ERROR_NO_CONNECTION:
        showDialog(modelEvent.getModelMessage());
        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 5 Handling errors in BaseView

After that, we implement views such as HomeView, PlaylistView, PlaylistItemView. Each view includes send data 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().

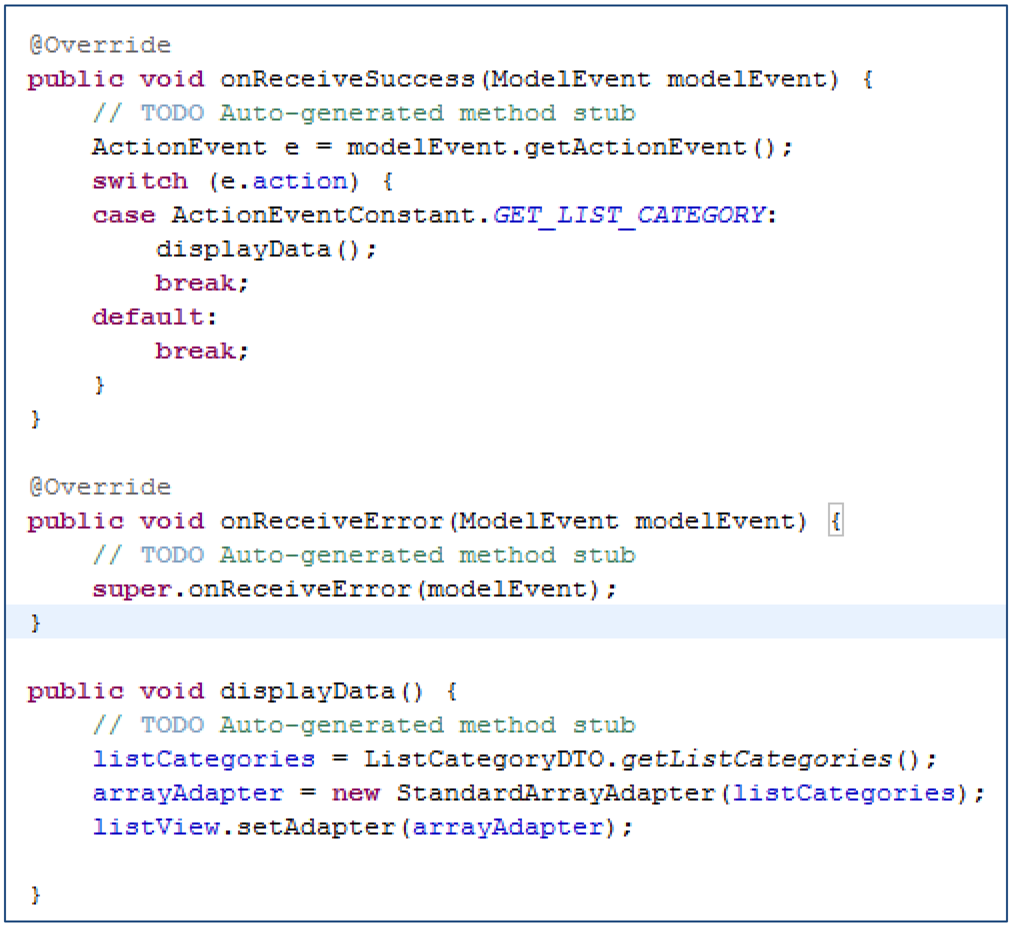
The following code in Figure 6 is an example of sendActionEvent() for request playlists by the chosen category. An ActionEvent object includes basic fields such as *action* as an identifier of an ActionEvent, *viewData* includes name and value of parameters, *sender* is view object to identify who sent this ActionEvent.



```
public void sendActionEvent() {
    Vector<String> vt = new Vector<String>();
    vt.add(IntentConstants.INTENT_CATEGORY);
    vt.add(category);
    ActionEvent e = new ActionEvent();
    e.action = ActionEventConstant.GET_LIST_PLAYLIST;
    e.viewData = vt;
    e.sender = PlaylistView.this;
    HomeController.getInstance().handleViewEvent(e);
}
```

Figure 6 SendActionEvent in View

And the **Figure 7** 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_CATEGORY:
        displayData();
        break;
    default:
        break;
    }
}

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

public void displayData() {
    // TODO Auto-generated method stub
    listCategories = ListCategoryDTO.getListCategories();
    arrayAdapter = new StandardArrayAdapter(listCategories);
    listView.setAdapter(arrayAdapter);

}
```

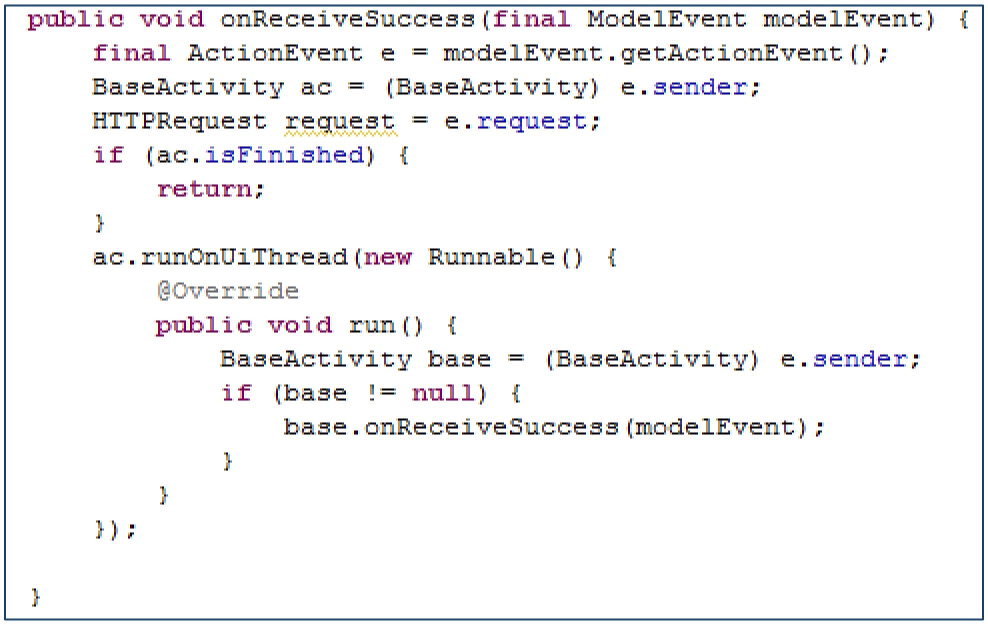
Figure 7 Update data for UI in View

❖ **Controller:**

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

Functions	Class	Description
sendActionEvent	HomeController(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	HomeController(Demo)	Handle switching views in application.

In AbstractController, we implement 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 8. And onReceiveError also handle similarly, however, we can log the error to server in this function depending on our need.

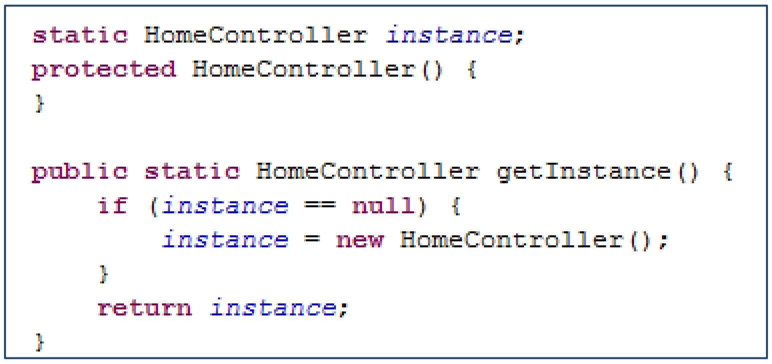


```
public void onReceiveSuccess(final ModelEvent modelEvent) {
    final ActionEvent e = modelEvent.getActionEvent();
    BaseActivity ac = (BaseActivity) e.sender;
    HTTPRequest request = e.request;
    if (ac.isFinished) {
        return;
    }
    ac.runOnUiThread(new Runnable() {
        @Override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

}
```

Figure 8: onReceiveSuccess in AbstractController

In application demo, one instance controller HomeController inherited from AbstractController is initialized by Single Instance design pattern (**Figure 9**). It makes easier to be invoked by other objects.

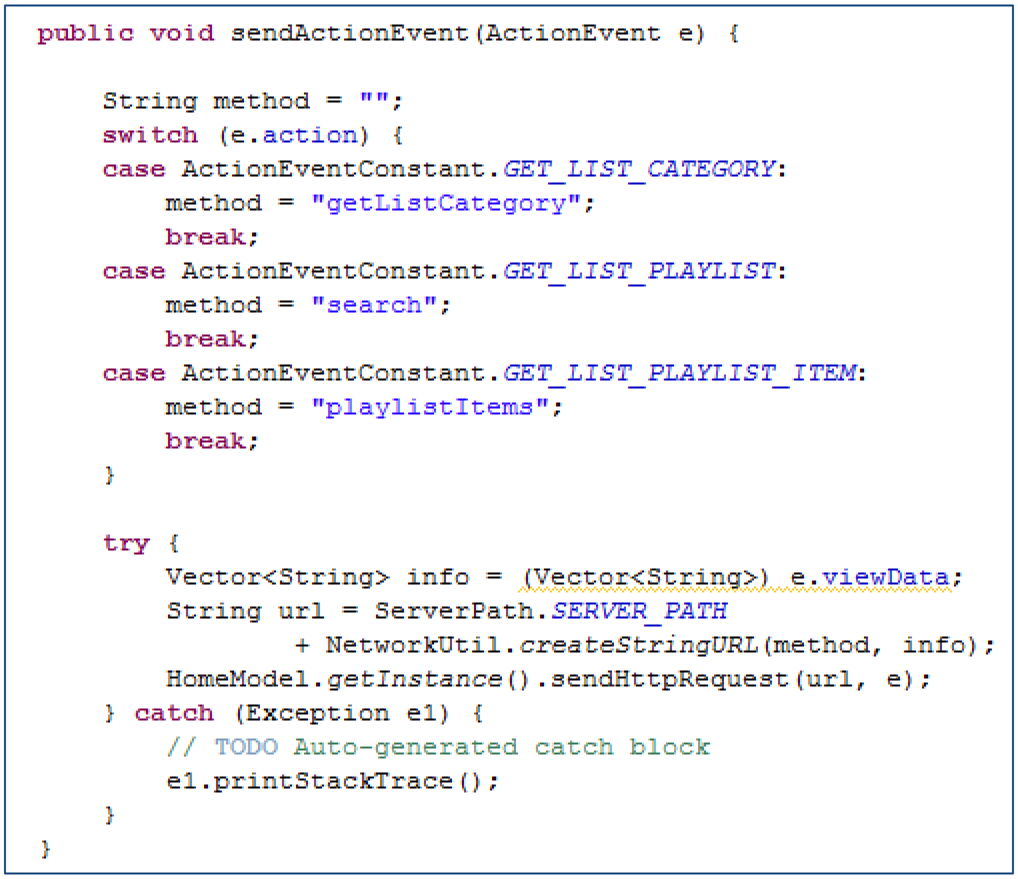


```
static HomeController instance;
protected HomeController() {
}

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

Figure 9: Single Instance of Controller

The HomeController receives ActionEvents and forward them to model via instance of model. The ActionEvents sent from views are distinguished by field *action.*as **Figure 10**:



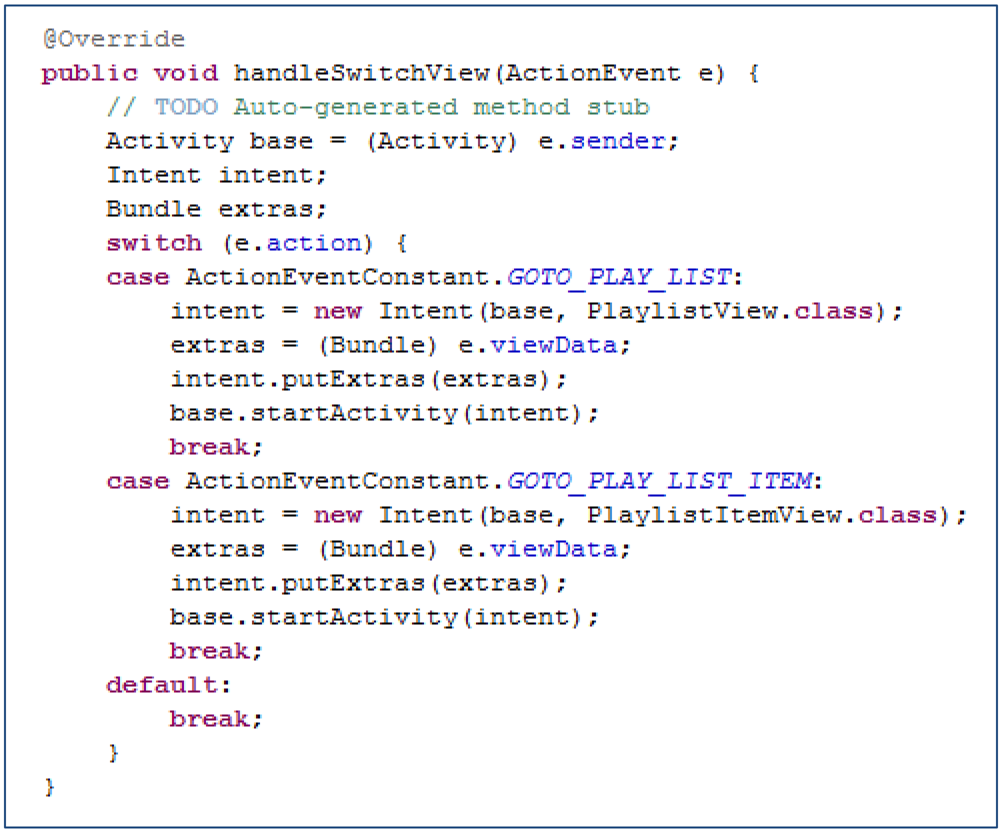
```java
public void sendActionEvent(ActionEvent e) {

    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 10: SendActionEvent in Controller

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



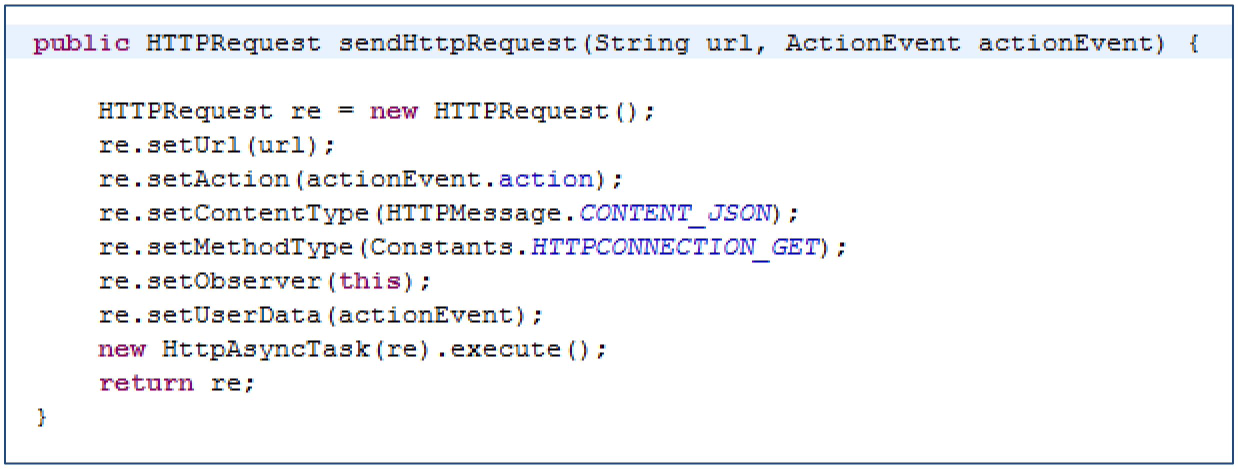
```
@Override
public void handleSwitchView(ActionEvent e) {
    // TODO Auto-generated method stub
    Activity base = (Activity) e.sender;
    Intent intent;
    Bundle extras;
    switch (e.action) {
    case ActionEventConstant.GOTO_PLAY_LIST:
        intent = new Intent(base, PlaylistView.class);
        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 3**, 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	HomeModel(Demo)	Handle receiving response from server in success
onReceiveError	HomeModel(Demo)	Handle receiving response from server in error

Firstly, AbstractModel need to implement sendHttpRequest which creates HTTPRequest object and forwards it to an independent thread called HttpAsyncTask as **Figure 11**.



```
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 11: SendHttpRequest in AbstractModel

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

After receiving the response from an independent thread, the HomeModel 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 HomeController as code below.

```java
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 HomeController.

```java
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.getErrMessage());
    HomeController.getInstance().onReceiveError(model);

}
```

7. Survey

After reviewing the framework and application demo, please help me to answer this survey. Your response is extremely important for me to evaluate the proposed framework for my thesis, Thanks for your help.

https://docs.google.com/forms/d/1eDVccBL4Rq-B8wVpk6k3r8oLl2cx05O9fdtTJNsqJZo/viewform