Multiplayer Game Programming

Chapter 9
Server Architecture / MVC

Chapter 9
Objectives

- **Server side Architecture**
  - **How is the MVC architecture applied to HTTP servers?**
  - How does architecture help debug client server issues?

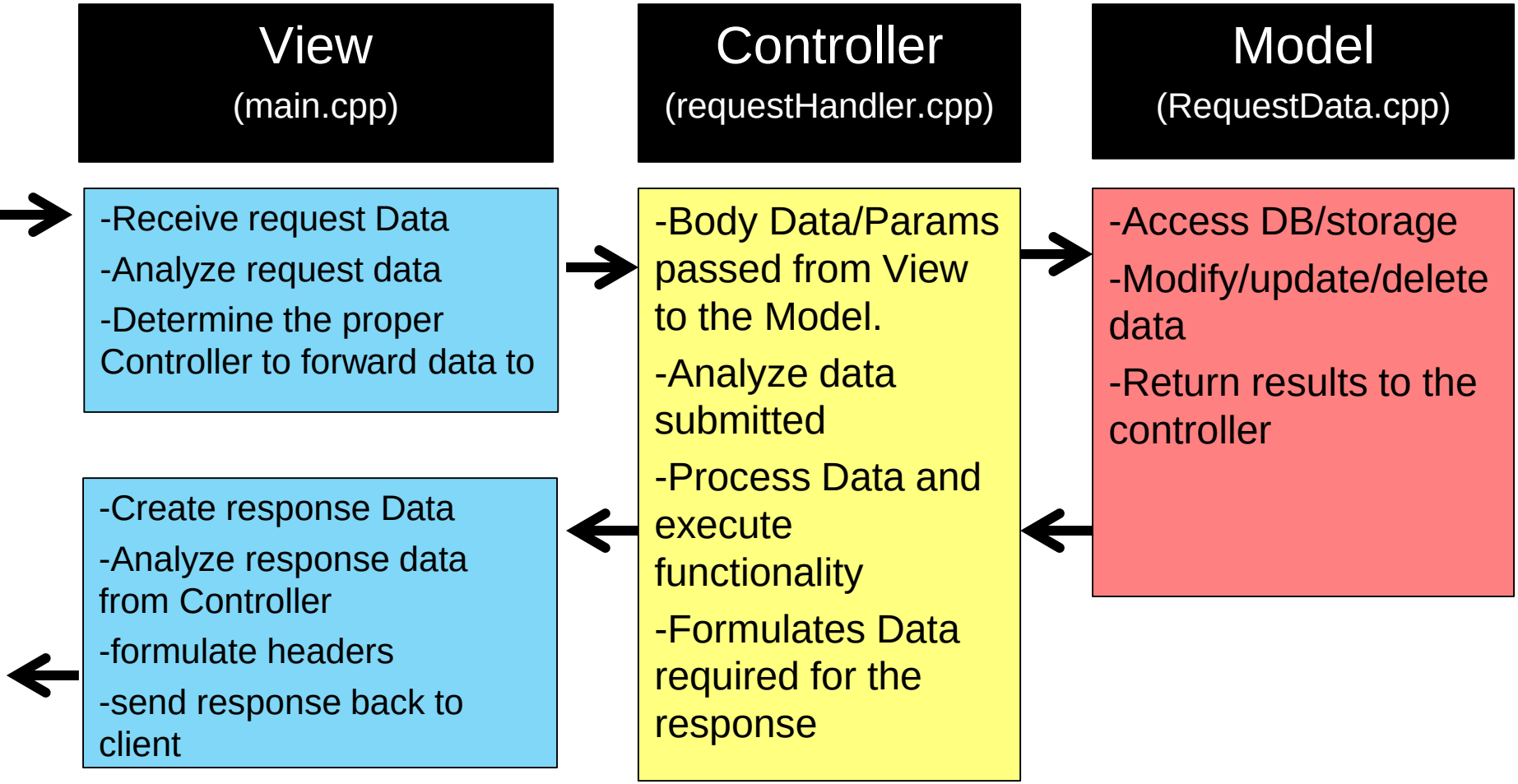
Model-Controller Architecture

- MVC
  - Model View Controller architecture.
  - Often implemented with REST
  - Separates processing the request, the processing logic, and the model data
  - Allows developers to debug and analyze efficiency of the server.
    - When a bug is discovered we can easily evaluate what the issue is and narrow down the file or function of the file to check quickly.

Client-Server Architecture

- Although Networking code isn't a direct application of MVC it's adaption still contains 3 sections.
  - **Model**: this is the class object representation of the requests data. WHERE data is stored.
    - This typically match 1 to 1 the data properties names from client to server.
      - Exceptions can be made to hold additional information so long as it is documented and marked not serializable.
  - **View**: this is facilitated by the communication of data in terms of our network code processing.
    - Essentially it's the main code of processing the requests URI and headers and determining how to process the request.
      - Generally this is the processing of the request data and the response data.
      - What the interface sees.
      - This is the 1 section visible to the user.
  - **Controller**: this is the individual processing of request data.
    - This uses the data supplied through the view to the model and processes the logic of the API.

Server Side Architecture



Server Side Architecture VIEW

```
13  using namespace web::http::experimental::listener;
14  using namespace std;
15
16  int main()
17  {
18      http_listener listener(L"http://localhost:8777/SLCGame311");
19
20      listener.support(methods::GET, handle_get);
21      listener.support(methods::POST, handle_post);
22
23      try
24      {
25          listener.open()
26              .then([&listener]() { wcout << ("\nstarting to listen :: \n") << listener.uri().to_string().c_str(); })
27              .wait();
28      }
29      catch (exception const & e)
30      {
31          wcout << e.what() << endl;
32      }
33
34      // infinite while loop to ensure our application continues to run and doesn't reach the end
35      while (true);
36
37      return 0;
38  }
39
```

View
(main.cpp)

GetRequestView.cpp/.h

PostRequestView.cpp/.h

- Our main.cpp Is the main **VIEW** entry point receiving our **POST** and **GET** requests.
- The Data for the Request is formulated into an http_request object and passed to each individual controller.

Server Side Architecture VIEW

```cpp
#include "GetRequestView.h"
#include "PlayerDiedController.h"

void handle_get(http_request request)
{
    wstring APIPath = request.absolute_uri().to_string();

    wcout << "\nAPI PATH:" << APIPath;
    cout << "\nhandle GET\n";

    if (wcscmp(APIPath.c_str(), L"/SLCGame311/PlayerDied") == 0)
    {
        PlayerDiedController Controller;
        Controller.Process(request);
    }

    request.reply(status_codes::BadRequest, "Requested API not found.");
}
```

VIEW
(GetRequestView.cpp/.h)
&
(PostRequestView.cpp/.h

- 1) GET VIEW processes the API endpoint and determines where to forward the request for processing.
- 2) This creates a Controller for the specific request.
  - The program contains individual controllers for each request.

Server Side Architecture VIEW

```
void PlayerDiedController::Process(http_request request)
{
    PlayerDiedRequest playerDiedReq;
    PlayerDiedModel playerDiedModel;

    // validate headers
    bool success = playerDiedReq.ValidateHeaders(request);
    if (!success)
    {
        request.reply(status_codes::FailedDependency, "Error, Missing or Incorrect Header Information");
    }

    // parse request to populate Model Data
    success = playerDiedReq.ProcessRequest(request, playerDiedModel);
    if (!success)
    {
        request.reply(status_codes::BadRequest, "Error, Unable to Process the Request");
    }

    // Complete Any Logic Here


    // Process and send the response
    PlayerDiedResponse Response;
    success = Response.ProcessResponse(playerDiedModel);
    if (!success)
    {
        request.reply(status_codes::BadRequest, "Error, Unable to Process Response Data");
    }

    Response.SendResponse(request);
}
```

CONTROLLER

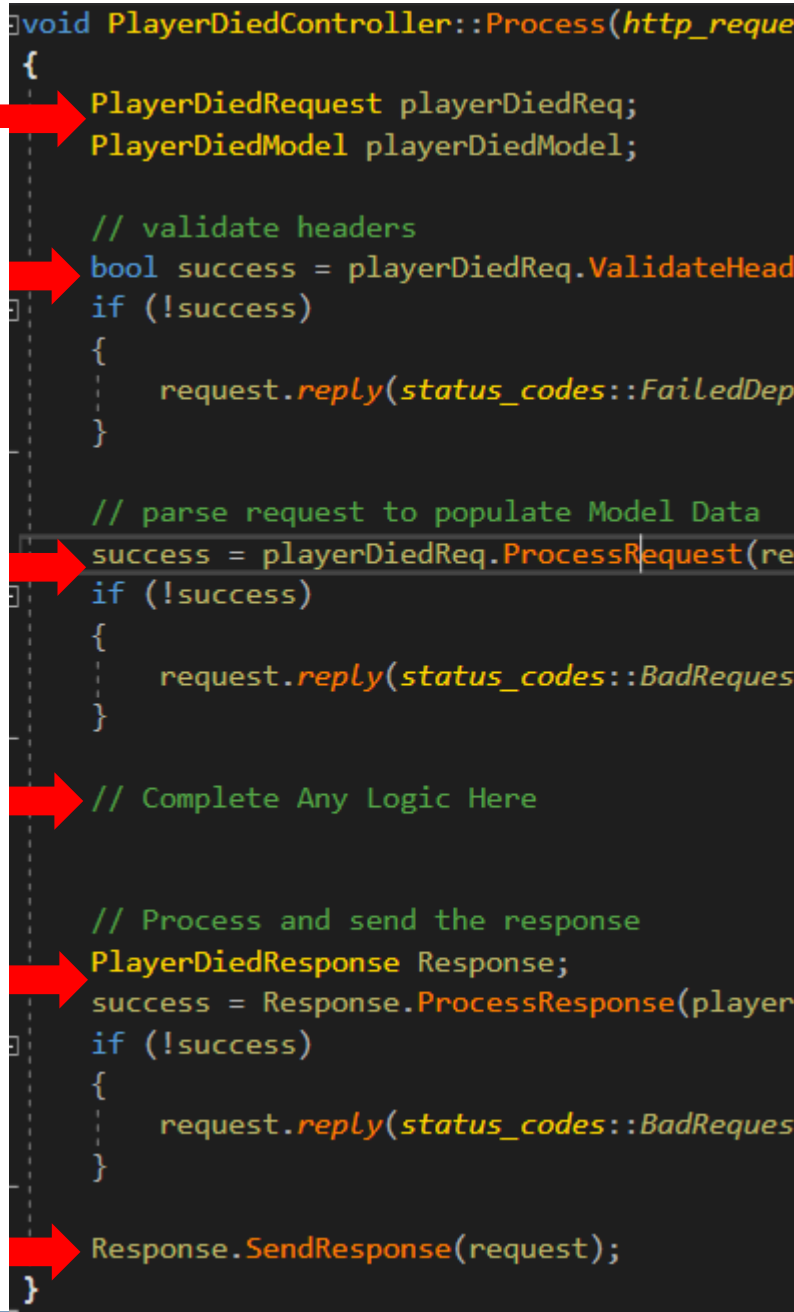
(playerDiedController.cpp/.h)

&

(LoiginController.cpp /.h

Server Side Architecture CONTROLLER

- The Controller Is the main Logic of our MVC Architecture.
- It's also the central connections between the 3 parts.
- The View forwards in the http_request object.
- The Controller Creates a Custom Request Object designed to parse the Headers and/or Body of a request for Data.
  - PlayerDiedRequest in this example.
- The Controller also creates a Custom Model to hold all the data used for the request, logic, and response.
  - The Controller often processes the data towards some sort of functionality after the request data is retrieved.
- Lastly the Controller creates a Custom Response object designed to populate the body and headers of the response and communicate back to the client



Server Side Architecture

```
bool PlayerDiedRequest::ValidateHeaders(http_request request)
{
    bool success = true;

    if (!request.headers().has(L"UserName"))
    {
        success = false;
    }

    return success;
}

bool PlayerDiedRequest::ProcessRequest(http_request request, PlayerDiedModel &playerDiedModel)
{
    //RequestBody = json::value::object();
    bool success = true;

    // grab all the headers
    http_headers requestHeaders = request.headers();

    // parse out the userID
    playerDiedModel.UserID = requestHeaders[L"UserName"];

    return success;
}
```

View

(main.cpp)

*Request.cpp/.h

Our Requests have two main functions, **ValidateHeaders** and **ProcessRequest**.

Each function as their names imply.

Note the **PlayerDiedModel** passed in to the request.

This is the model which is populated from the controller.

Server Side Architecture

Model
(PlayerDiedModel.h)
(LoginModel.h)

```
1   #pragma once
2
3   using namespace std;
4
5   struct LoginModel
6   {
7       // not serialized
8       wstring Name;
9
10      // object specific variables:
11      int SessionToken;
12      wstring ResponseStr;
13  };
14
```

```
1   #pragma once
2
3   using namespace std;
4
5   struct PlayerDiedModel
6   {
7       // object specific variables:
8       wstring UserName;
9   };
10
11
```

- The Model Data is used to house request and response information.
  - Sometimes this is handled through classes with encapsulation applied to protect the data.

Server Side Architecture

```
bool PlayerDiedResponse::ProcessResponse(PlayerDiedModel model)
{
    bool success = true;
    // set the UserID inside of our Response
    ResponseBody[L"UserName"] = json::value::string(model.UserName);

    return success;
}

void PlayerDiedResponse::SendResponse(http_request request)
{
    // send the JSON
    request.reply(status_codes::OK, ResponseBody);
}
```

View

(main.cpp)

*Response.cpp/.h

Our Response also has two main functions, **ProcessResponse** & **SendResponse** which are also aptly named.

The Process response takes in the Model data which houses the data modified through the initial request and any modifications performed by the Controller.

Summary: Debugging Server Code

- When debugging an error on the server request processing we can break down what occurred and determine where to investigate.
  - If it's a logical error:
    - it's likely the controller.
  - If it's a mismatched data error:
    - It's likely the model and we need to cross compare the clients model to the servers model structure.
  - If it's a denied request:
    - It's likely an issue with the View
- Another advantage to having modular components to our architecture and separating code is that it makes it easier for multiple programmers to be working on different sections of the same request at the same time and avoid conflicts in code.