UNIVERSITY OF CALIFORNIA, MERCED

BERKELEY · DAVIS · IRVINE · LOS ANGELES · MERCED · RIVERSIDE · SAN DIEGO · SAN FRANCISCO — SANTA BARBARA · SANTA CRUZ

# CSE 30: Data Structures
# Laboratory 2

Fall 2019

## 1 Introduction

In this lab, you will get more practice using the web application sample code that we have been exploring in class. The code has been significantly modified but it can still be built by typing `make`, and it still produces an executable called `server`, which is found in the `bin` folder. The rest of the document will walk you through the code, including both frontend and backend. The last section contains some exercises.

## 2 Front End and Back End

Wikipedia defines front end and back end as " the separation of concerns between the presentation layer (front end), and the data access layer (back end) of a piece of software". In other words, the front end is responsible for formatting and displaying the data (presentation), while the backend does the computation work to get the right data to the front end (data access).

It is important to keep these two entities separate, and never have business logic on the front end, or presentation work on the back end (even though some people find this to be acceptable). The reason why the front end should not do business logic, which is basically any kind of computational work needed for the product, is because the front end is the user interface, and user interfaces change. We may decide to replace our web interface with a mobile app, native to iOS, and a separate app for Android. Now, any business logic that was implemented by the web interface, will have to be implemented in the two (or possibly more) native apps that replace the web interface. In summary, the interface should only ever ask for data from the server, and it should visualize/present the data.

Speaking of data, when the server receives a request for data, it should send it in a generic format, understandable to anyone. Imagine if the server gets asked for a list of players currently in Real Madrid, and it returns this:

```
<ul>
    <li>1 GK Alphonse Areola</li>
    <li>2 DF Dani Carvajal</li>
    <li>3 DF Eder Militao</li>
    <li>4 DF Sergio Ramos</li>
    <li>5 DF Raphael Varane</li>
    <li>6 DF Nacho</li>
    <li>7 FW Eden Hazard</li>
    <li>8 MF Toni Kroos</li>
    <li>9 FW Karim Benzema</li>
    <li>10 MF Luka Modric</li>
    <li>11 FW Gareth Bale</li>
    <li>12 DF Marcelo</li>
    <li>13 GK Thibaut Courtois</li>
</ul>
```

That would be great if we had a web interface, because we can just put that text on a web page, completely unchanged. But what if we had a native app? This HTML code will not be very useful as it can not be displayed directly, so to use it, we would have to process it first, to extract the names, positions, and jersey numbers, but leave out all the HTML tags. This is an example of the backend doing presentation work, and that should be left to the front end. The back end should just get the data and send it in JSON format. That would look like this:

```
data =[
    {"jersey" : 1, "position" : "GK", "name" : "Alphonse Areola"},
    {"jersey" : 2, "position" : "DF", "name" : "Dani Carvajal"},
    {"jersey" : 3, "position" : "DF", "name" : "Eder Militao"},
    {"jersey" : 4, "position" : "DF", "name" : "Sergio Ramos"},
    {"jersey" : 5, "position" : "DF", "name" : "Raphael Varane"},
    {"jersey" : 6, "position" : "DF", "name" : "Nacho"},
    {"jersey" : 7, "position" : "FW", "name" : "Eden Hazard"},
    {"jersey" : 8, "position" : "MF", "name" : "Toni Kroos"},
    {"jersey" : 9, "position" : "FW", "name" : "Karim Benzema"},
    {"jersey" : 10, "position" : "MF", "name" : "Luka Modric"},
    {"jersey" : 11, "position" : "FW", "name" : "Gareth Bale"},
    {"jersey" : 12, "position" : "DF", "name" : "Marcelo"},
    {"jersey" : 13, "position" : "GK", "name" : "Thibaut Courtois"}
]
```

The above data is in JSON format. Specifically, in is an array/list of JSON records, where each record contains the `jersey`, `position`, and `name` of a particular player. So for example, if we wanted to get the details for Alphonse Areola, we type the following, typically in JavaScript:

```
    var name = data[0]["name"];
    var pos = data[0]["position"];
    var num = data[0]["jersey"];

    // We can now build a string out of the variables above

    var result = "The player " + name + " plays as a " + pos + " and wears the number "
        + num + " jersey";
```

Now, we have the situation where the front end is responsible for the presentation, and the backend is responsible for providing the data. Every app we write in this course, will follow this principle, namely it will be separated into a back end and a front end, and they will both do the correct job. Applications of this kind are called RESTful applications, and people who write them are called full-stack developers. In real life, we rarely get a situation where one person creates the front end and the back end, but a full-stack developer has the ability to work on either end.

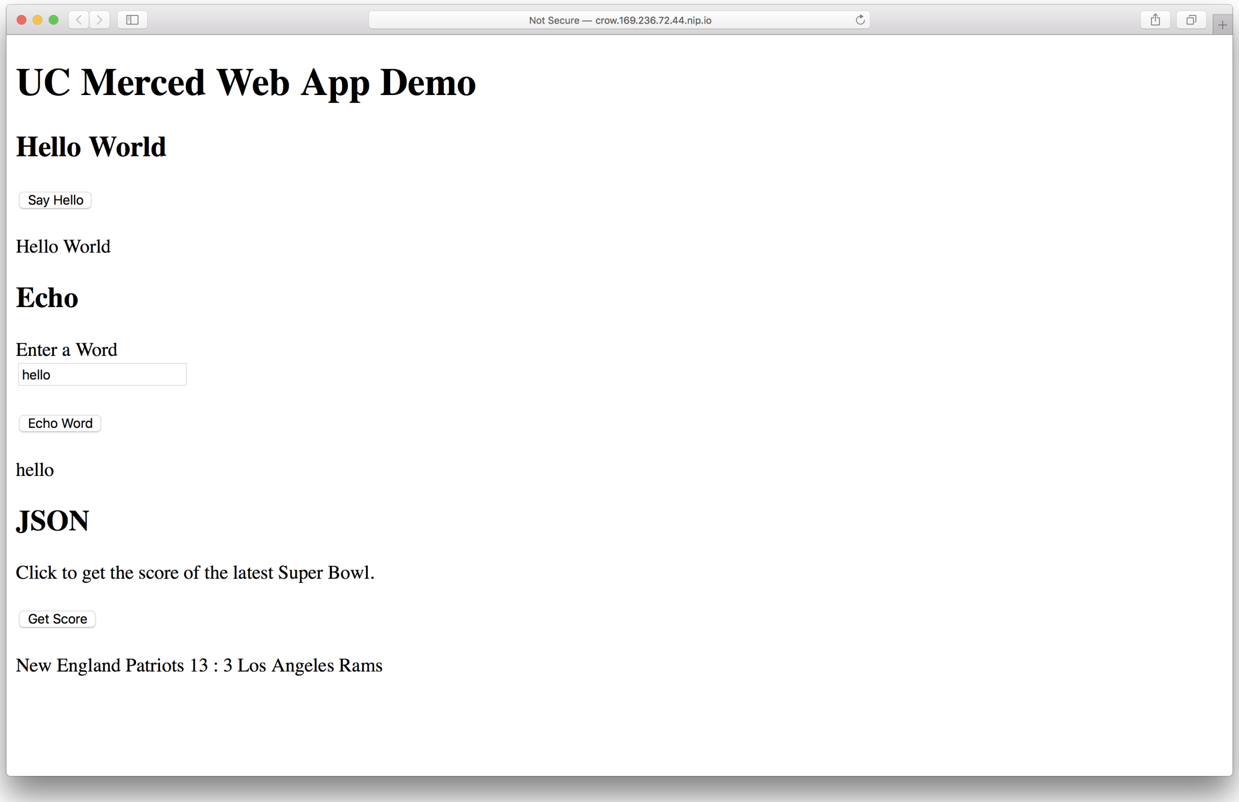
## 3 The UC Merced Crow Application

The demo application, that we have been looking at in class, as well as in the labs last week, is an example of a RESTful application. We call it a Crow Application, because it uses a framework called Crow to implement web services. There are also other frameworks and libraries, such as jQuery, and a fair amount of original code, but the main component is Crow.

When we build the application and run it, we get a web server running on our computer on port 18080. To access this web server, we can go to a web browser and visit the address:

```
127.0.0.1:18080
```

The app looks like this:


Not Secure — crow.169.236.72.44.nip.io

UC Merced Web App Demo

Hello World

Say Hello

Hello World

Echo

Enter a Word

hello

Echo Word

hello

JSON

Click to get the score of the latest Super Bowl.

Get Score

New England Patriots 13 : 3 Los Angeles Rams

It is divided into three sections. The first one just has a button, and when the user clicks it, the text "Hello World" appears. The second section is called "Echo", so it repeats whatever word the user enters in the textbox. The third section is called JSON, and when the user clicks the button, the score from the latest Super Bowl appears.

### 3.1 The Front End

How does this all happen? We will go over the code line-by-line, file-by-file, highlighting the important stuff. Starting from the frontmost part of the front end, the HTML interface.

### 3.1.1 The `index.html` File

Here is a the HTML code that produces this interface. The file is called `index.html`, and it is in the templates folder.

```
1  <!DOCTYPE html>
2
3  <html>
4
5  <head>
6      <meta charset="UTF-8">
7      <title>CrowApp Demo</title>
8      <script src="jquery.js"></script>
9      <script src="main.js"></script>
10
11 </head>
12
13 <body>
14     <h1>CrowApp Demo</h1>
15     <h2>Hello World</h2>
16     <button onclick="clickHello()">Say Hello</button><br><br>
17     <div id="hello-area"></div>
18
19     <h2>Echo</h2>
20     Enter a Word<br>
21     <input id="echo-input"><br><br>
22     <button onclick="clickEcho()">Echo Word</button><br><br>
23     <div id="echo-area"></div>
24
25     <h2>JSON</h2>
26     Click to get the score of the latest Super Bowl.<br><br>
27     <button onclick="clickScore()">Get Score</button><br><br>
28     <div id="score-area"></div>
29 </body>
30
31 </html>
```

As always, HTML files are divided into a `<head>` section, and a `<body>` section. In the head section, we only specify the title of the page (line 7), which is the text that will be displayed on the tab of the browser. In lines 8 and 9, we import some JavaScript files. The file `main.js` contains definitions of the functions that need to be executed when buttons on the interface are clicked. Such functions are called event handlers. The `jquery.js` file is a popular JavaScript library that makes it easy to interact with and manipulate HTML elements.

The contents of the page are all under the `<body>` section. Line 14 says that the main heading of the page (the `<h1>` tag), should say "CrowAppDemo", followed immediately by a smaller heading, hence the `<h2>` tag, which says "Hello World". Line 16 specifies that a button should appear next on the interface. The text on the button will be "Say Hello", and when the user click it, we will execute the `clickHello()` function, which is defined in `main.js`. More on that soon. Line 16 ends with two `<br>` tags, indicating that we should leave two blank lines before we render the next element. Finally, line 17 specifies an empty placeholder section of the site. This placeholder `<div>` can be referred to by id, which is `hello-area`. This is where the "Hello World" message will appear, after the user clicks on the "Say Hello" button.

The second heading, on line 19, specifies the beginning of the second section. Unlike the first section, where the user simply clicked a button and something appeared, here we need to provide some information to the server. This is done by using a text field, defined on line 21. In HTML, the `<input>` tag creates a text field, and since our JavaScript code will be reading the value from that text field, we need to give it an id, so that the JavaScript code can refer to it. In this case (still on line 21), the id is `echo-input`. The third section of the webpage is exactly the same as the first one, in terms of interface elements, so we will not discuss it further.

### 3.1.2 The `main.js` File

This is the JavaScript file, which is responsible for all the interactive aspects of our interface. It is found in the `static` folder.

```
1  function clickHello(){
2    $.get("/hello", {}, function(response){
3      $("#hello-area").html(response);
4    });
5  }
6
7  function clickEcho(){
8    var theWord = $("#echo-input").val();
9
10   $.get("/echo", {word: theWord}, function(response){
11     $("#echo-area").html(response);
12   });
13 }
14
15 function clickScore(){
16   $.get("/superbowl", {}, function(response){
17     var details = JSON.parse(response);
18
19     var displayScore = details["home_team"] + " " + details["home_score
         "] + " : " + details["away_score"] + " " + details["away_team"];
20
21     $("#score-area").html(displayScore);
22   });
23 }
```

This file is divided into three function definitions. The first one, lines 1-5, is the event handler for the "Say Hello" button. It performs a GET request against the `/hello` endpoint on the server.

The jQuery function `$.get()` performs the request, and it needs three arguments. The first one is a string, specifying the endpoint on the server that we need to connect to. The second one is a JSON object, containing any data that we need to pass to the server in order for it to be able to perform its task. In this case it is empty. The third argument is the function that we need to call after we have successfully received a response from the server. This function should always have a parameter called `response`, so we are basically telling the interface what to do with the response. In this case, we happened to know that the response from the `/hello` endpoint is just a string, so we can directly inject it into the placeholder area in the first section. This happens on line 3.

The second event handler is defined in lines 7-13. It performs a request against an endpoint called `/echo`. We know (from the server/backend) documentation, that the `/echo` endpoint expects to get a string named `word`. So we first read what the user has entered in the `echo-input` text box and we save that in a variable called `theWord`, line 8. We then pass that along to the server on line 10, where we also specify what the interface should do with the response.

Finally, the last event handler, defined on lines 15-23 gets the score of the latest Super Bowl game, by requesting that information from an endpoint called `/superbowl`. Because the score of a football game requires the server to transmit multiple pieces of information, the server sends a JSON object instead of a string. We first convert the `response` string to a JSON, line 17, we build a custom string, line 19, and we display it in the placeholder area, line 21. Notice that when we build the custom string, we can access individual elements of the data by name, such as `data["home_team"]`. This is the power of JSON and the principle of separating backend logic from frontend business.

## 3.2 The Back End

The backend is a piece of software that is available over the internet. Clients can ask the backend for information by accessing the various endpoints implemented on the backend.

### 3.2.1 The `app.cpp` File

This is where the endpoints needed by the frontend, and possibly others, have been defined. The file `app.cpp` is in the `src` folder.

```cpp
#include <server.h>
#include <myLib.h>

using namespace ucm;

int main(int argc, char** argv){

    CrowServer server(argc, argv);

    server.renderHTML("/", "index.html");

    server.route("/hello", [](const request& req, response& res){
        std::string phrase = sayHello();

        res.sendHTML(phrase);
    });

18      server.route("/echo", [](const request& req, response& res){
19          if (req.url_params.hasKey("word")){
20              std::string word = req.url_params.get("word");
21
22              std::string answer = echoWord(word);
23
24              res.sendHTML(answer);
25          }
26          else{
27              res.sendError400();
28          }
29      });
30
31      server.route("/superbowl", [](const request& req, response& res){
32          json data = getSuperBowlScore();
33
34          res.sendJSON(data);
35      });
36
37      server.run();
38  }
```

In the `main` function of our C++ app, we simply declare a server instance (line 8), we register some endpoints on it, (lines 10, 12, 18, and 31), and we finally run the server (line 37). As you have seen, when the server runs, it runs forever, listening to requests on the port, unless we quit it. So it is different from the programs you have been writing before, in a sense that previous programs were more like scripts, they run once and terminate, usually within milliseconds. This is an actual piece of software, that is always on and always working.

The first endpoint, on line 10, is a special kind of endpoint. Instead of serving data in JSON format, it serves an HTML file, called `index.html` and it serves it on the root of the server. This means that, when we go to the root of the server, we will be given the HTML page, which of course contains JavaScript, so the server also dishes out the `main.js` file, and a bunch of other things in the `static` folder. That folder is generally for things that need to be available, in order for our app to work.

The first endpoint that actually returns information, rather than a webpage is on line 12. The first argument specifies that this is the `/hello` endpoint, which we can visit by going to `127.0.0.1:18080/hello` in a browser. Of course, we will not be visiting this endpoint manually in the browser, our JavaScript code will be doing that for us and will be visualizing the response appropriately. The second thing we specify on line 12, is the C++ function that needs to be executed when a client connects to this endpoint. That funny C++ syntax: `[](const request& req, response& res){}` is called a lambda function, which is the same as a regular function, only it does not have a name. We only specify its arguments, which are basically `request` and `response`. If the client is passing over any information along with the request, we will find it in the `req` object. Otherwise, as it is the case with the `/hello` endpoint, we do not make any use of the `req` object. The `res` object is used to send a response, as you can see in line 15. We send back the value of the variable `phrase`, which is the result of calling the `sayHello` function.

That is correct. Endpoints do not calculate the values they should be returning in place. Instead, they make use of functions, declared and defined in `myLib.h` and `myLib.cpp`, respectively. As you have seen in class, the reason we separate the functions in their own file, is so that we can test these functions without having to run the webserver and interact with the interface in ordet to see if the functions are behaving correctly. They are now their own individual units, so that they can be unit tested.

The second endpoint, defined on lines 18-29, is a little bit more involved because it needs to accept inputs with the request. On line 19, we check if the client making the request has sent something called `"word"`. If that is the case, we get the value of this `"word"`, line 20, we use it to compute the result we need to return, line 22, and we return it, line 24. If however, the client has not send a key-value pair, with a key named `"word"`, then we return an error, with code 400, meaning it's a bad request.

The last endpoint, which returns the Super Bowl result, is a little bit more simple. It gets the data as JSON from the `getSuperBowlScore()` function, and directly sends it to the client, as JSON. We will see how to build JSON objects in C++, when we look as the file `myLib.cpp`. First, we have the header file, `myLib.h`, which is in the `inc` folder, while all the C++ files are in the `src` folder.

### 3.2.2 The `myLib.h` File

As seen in class, header files only provide function signatures, without implementation. This way you can quickly see what functions are available in the library, so you can get an idea of *what* it does. You just will not know *how* it does it. More on that later.

```cpp
#ifndef MYLIB
#define MYLIB

#include <string>
#include <server.h>

std::string sayHello();

std::string echoWord(std::string);

ucm::json getSuperBowlScore();

#endif
```

You can basically see that the three functions, one for each endpoint, are declared here. From the names of the functions, the return types, and the arguments they take in, it is not hard to see what each one is supposed to do.

For the implementation of each of these functions, we go to the `myLib.cpp` file.

## 3.3 The `myLib.cpp` File

```cpp
#include <myLib.h>

std::string sayHello(){
    return "Hello World";
}
```

```
 6
 7  std::string echoWord(std::string word){
 8      return word;
 9  }
10
11  ucm::json getSuperBowlScore(){
12          ucm::json data;
13
14          data["home_team"] = "New England Patriots";
15          data["home_score"] = 13;
16          data["away_team"] = "Los Angeles Rams";
17          data["away_score"] = 3;
18
19          return data;
20  }
```

The implementation of the first two functions is easy enough so we will not discuss it any further. The third function, defined on lines 11-20, illustrates how to create a JSON object in C++. On line 12, we declare an empty JSON, lines 14-17 we populate it with data, and finally on line 19, we return it.

At this point, some of you may think we are done, but we still have yet to do the most important thing. The unit tests.

### 3.3.1 The `test.cpp` File

```
#include <igloo/igloo.h>

#include "myLib.h"

using namespace igloo;

Context(HelloFunction) {

  Spec(TheOnlyTestPossible) {
    Assert::That(sayHello(), Equals("Hello World"));
  }
};

Context(EchoFunction) {

  Spec(EchoHello) {
    Assert::That(echoWord("hello"), Equals("hello"));
  }

  Spec(EchoWorld) {
    Assert::That(echoWord("world"), Equals("world"));
  }
};
```

```

bool compare(const ucm::json& lhs, const ucm::json& rhs){
  return true;
}

Context(SuperBowlFunction){
  Spec(OnlyPossibleTest){
    ucm::json score;
    score["home_team"] = "New England Patriots";
        score["home_score"] = 13;
        score["away_team"] = "Los Angeles Rams";
        score["away_score"] = 3;
    Assert::That(compare(getSuperBowlScore(),score));
  }
};


int main() {
  // Run all the tests defined above
  return TestRunner::RunAllTests();
}
```

The contents of the file are self-explanatory. Each function is tested in the manner that it can be tested, and all the tests are trivial.

Even though we have stepped through the contents of the project in the order that we have (front to back), often times when we start a new project from scratch, we will implement things in the reverse order of what was presented here, meaning unit tests first. Once again, more on that later.

## 4 Exercises

Please complete the following tasks before the deadline specified on CatCourses. Download the "crow-server" application from CatCourses, and make the changes below. Upload the modified project as a zip archive back to CatCourses.

1. Change the title of the webpage from "CrowApp Demo", to "UC Merced Web App Demo".
2. Modify the heading of the last section of the webpage from "JSON" to "Super Bowl".
3. When the user clicks the "Get Score" button, make it display the score as a sentence of the form "The [winning team] beat the [losing team] by [winning points] points to [losing points]". So in the example we have, instead of displaying "New England Patriots 13 : 3 Los Angeles Rams", we want to get "The New England Patriots beat the Los Angeles Rams by 13 points to 3".
4. Modify the echo function so that instead of returning the same word to you, the server returns the word in reverse. For example, if we provide the word "hello", the server should give us "olleh".
5. Create a fourth section on the website, call it "Upper Case". The user should be able to input a string, which we pass along to the server, and the server gives it to us in two copies, one all uppercase, and one all lowercase. Display the massage under the button: "Uppercase [WORD] : Lowercase [word]".