Build an Online Data Analysis Tool to Explore World Bank Data

Hong Kong University of Science and Technology

2015 Fall MSc(IT) Program

CSIT 6910 Independent Project

Supervisor: Prof. David Rossiter

Student Name: Liu Yufan
Student No.: 29266796
Email: yliude@connect.ust.hk

Content Table

1. Background

1) Motivation

World bank have designed different indicators to estimate different country's status. These indicators includes 248 countries cover years range from 1960 to 2015. As these indicators are public, everyone can download a copy and do any operation on it.
I would like to learn these indicators, to learn their meaning and their historic trend among different countries. As world bank has created a tool for users to explore and analysis these data, I could use it as a prototype to develop my own analysis tool.
Besides, I am curious about the Scala technology. If I use Scala and its related frameworks or technologies to work on this project, I could have a better understanding of the language and its ecosystem.

2) Expect output

Create a website where user can do basic analysis on World Bank Data.
Learn some of the indicators, understand them, and their historical trend in different countries.
Learn Scala technology with its ecosystem.
Learn Front-end techniques to create beautiful and interactive web pages.

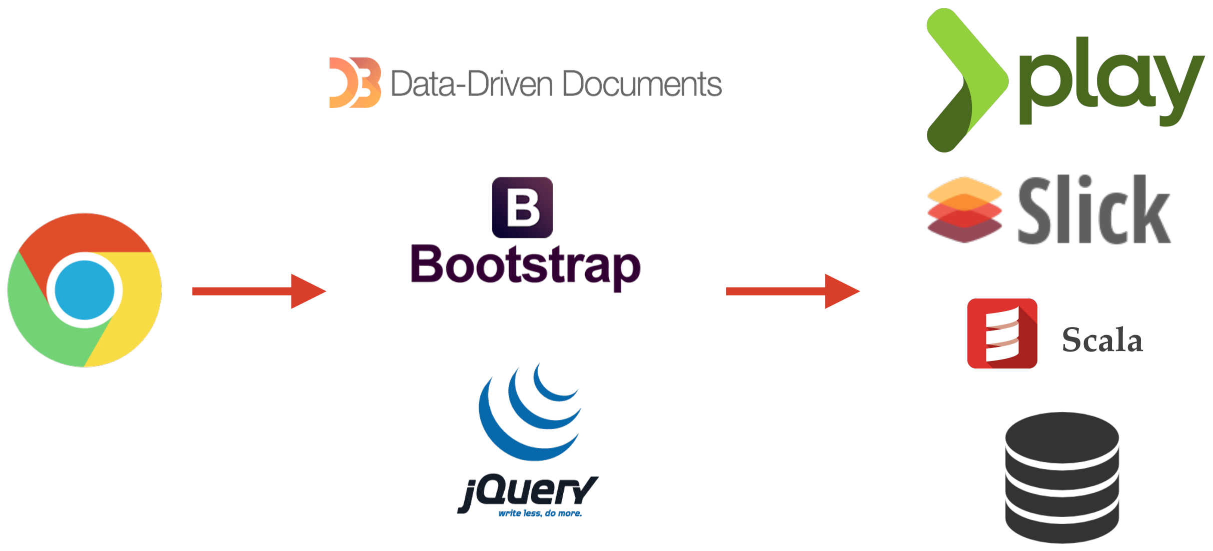
2. Develop Environment

ITEM	NOTE
OS	Mac OS X Yosemite 10.10.5
IDE	Idea Intellij 15.0.1
DataBase	Mysql community server 5.6.25
Default Target Browser	Chrome 47.0.2526.73 (64-bit)
Programming Language	Scala 2.11.7
Build Tool	SBT 0.13.5
Web Server Techniques	TypeSafe Activator 1.3.6
	Play Framework 2.4.2
Database access library	Slick 3.1.0
Front-end Techniques	jQuery 1.11.3
	Bootstrap v3.3.5
	d3js v3.1

3. Architecture

The Project is developed with layered structure. Layered structure can decouple each component. This will make each component easy to test and implement. Developers can switch any component without interferer other components. This can make development process easy to estimate.

Below is the layered structure of this project.



On the left side is the web browser. In this project, chrome is chosen as the default dev/test/production browser. In order to make the website compatible with different browsers, especially IE, I use 3rd party javascript libraries "***html5shiv.min.js***" and "***respond.min.js***" to do the job.

In the middle part, are the front-end techniques used in this project. Beside basic "***Html + CSS + Pure Javascript***" techniques, I use existing 3rd party javascript libraries to speed up development process. Please check the table for their detailed description.

JS Framework	Description
jQuery	It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. It is required by D3js.
Bootstrap	It is used to do layout in this project.
D3js	It is used to render data points to generate graphs.

On the right side are the back-end techniques. All the code are written in Scala. Please check the table for their detailed description.

Server Side Technique	Description
Play Framework	It is a powerful web framework. It provides many built-in functionalities which can speed up develop process. Play is based on a lightweight, stateless, web-friendly architecture. Built on Akka, Play provides predictable and minimal resource consumption (CPU, memory, threads) for highly-scalable applications.
Slick	A functional relational mapping library. This project's data access layer is based on Slick. It's very convenient and easy to use.
Scala	The major programing language. In the upper layer, all the code are written in Scala.
Mysql	Store data.

4. Implementation

1) Get to know the data

There are plenty of data published in the World Bank website. In this project, we choose "***Indicators***" data. For example, to access the "***GDP (current US$)***" indicator data, we can download them from this page: http://data.worldbank.org/indicator/NY.GDP.MKTP.CD?display=default. There are three formats: XML, EXCEL, CSV. In this project, we use CSV format.

For each indicator, there are 3 csv files.

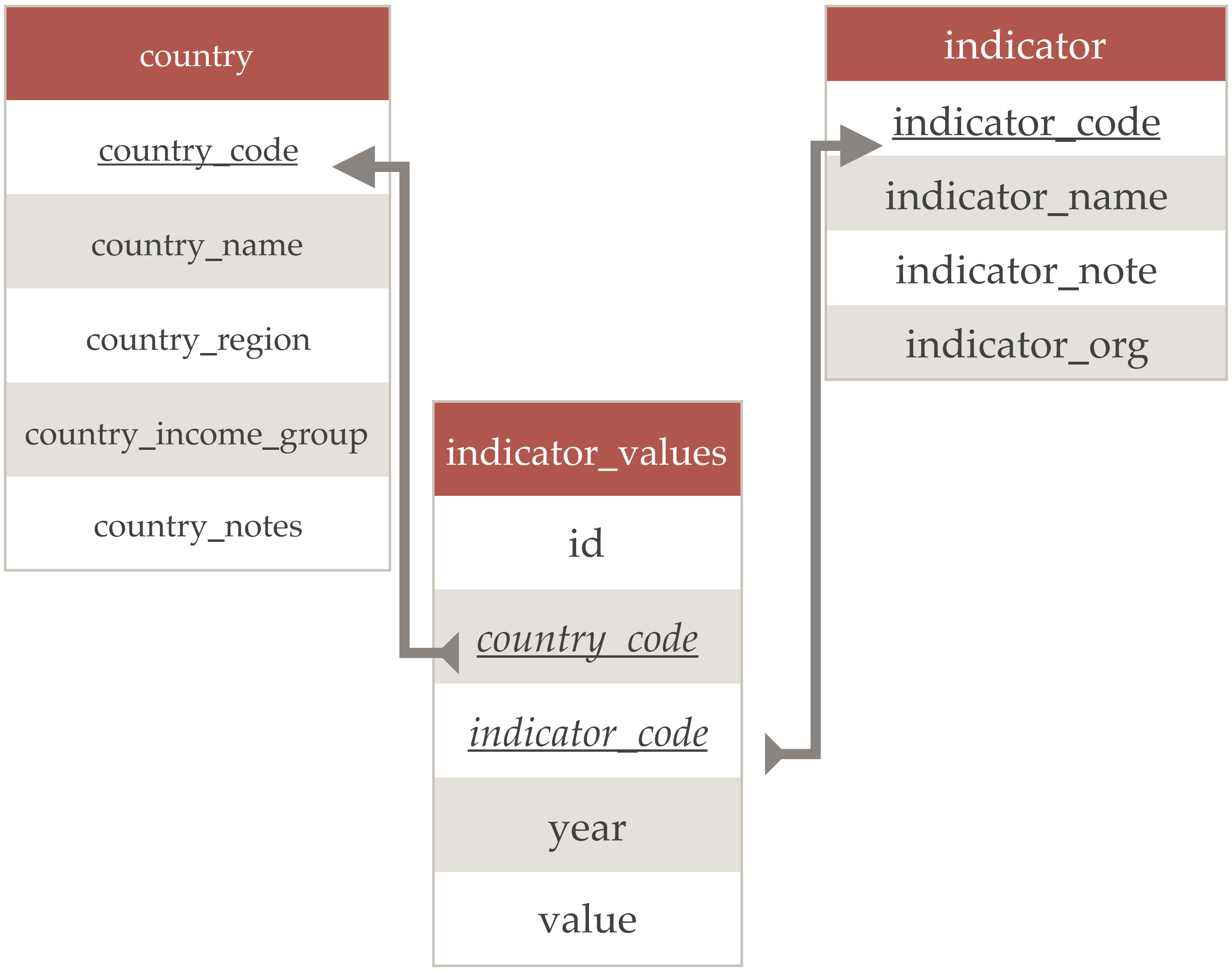
File	Content
All Countries	Include all country meta data.
Indicator Description	Describe the indicator.
Indicator values	Contains all indicator values for each country from 1960 to 2015. One value for each country for each year.

* The "All Countries" file is duplicate because all downloaded indicator zip packages will have one copy. So we only need to store one copy of "All Countries" file.

2) Database Access Layer Design

i) Design E-R Diagram

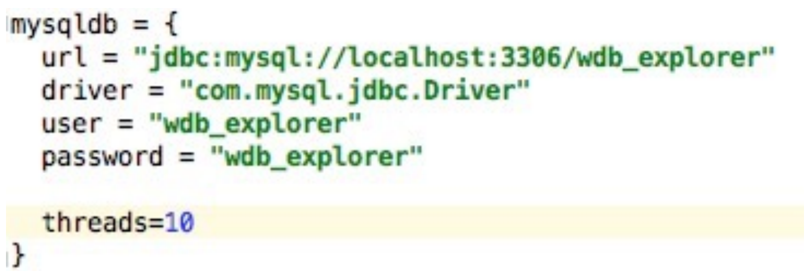
As we have mentioned above, we have only 3 csv files for each indicator. So the E-R diagram is pretty simple and straight-forward.



We create 3 tables. The "***country***" table contains all information about a country. The "***indicator***" table contains all information about a indicator. Then the "***indicator_values***" table contains each indicator_value, one for each country for each year. It reference "***country***" table and "***indicator***" using foreign key "***country_code***" and "***indicator_code***" separately.

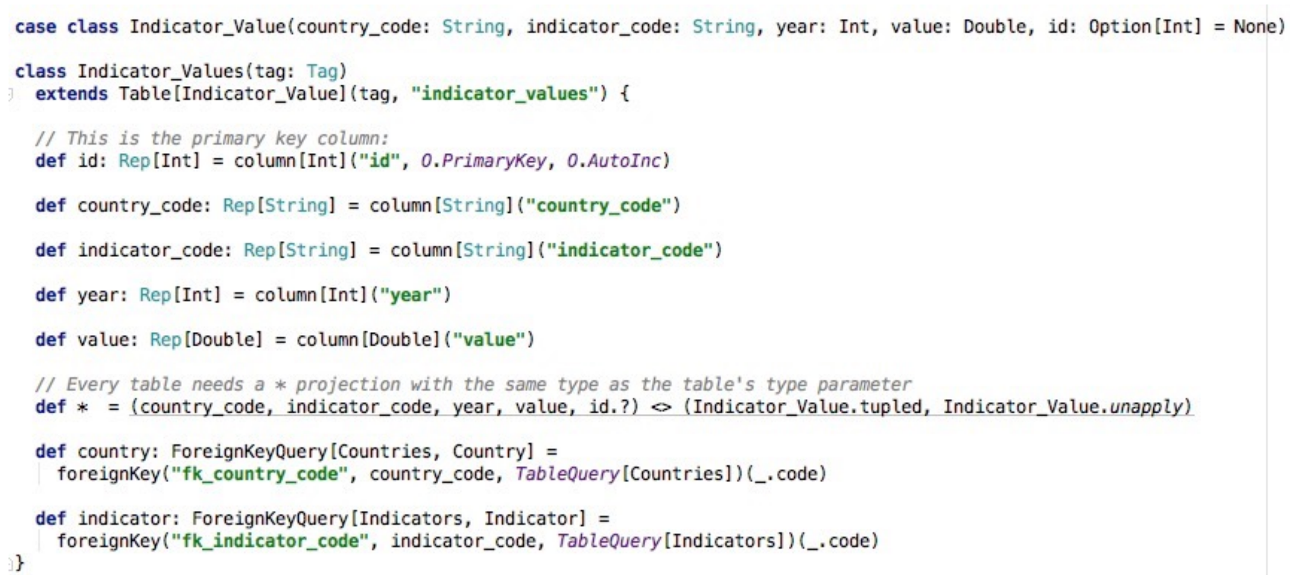
ii) Prepare Database Access

First, we need to create database schema in the MySQL Database.
Then we need to add our database connection information the configuration file, so Slick knows where to connect to.



```
mysqldb = {
  url = "jdbc:mysql://localhost:3306/wdb_explorer"
  driver = "com.mysql.jdbc.Driver"
  user = "wdb_explorer"
  password = "wdb_explorer"

  threads=10
}
```

Finally, we need to define the relational mapping from Scala object to Database Schema. Below is the mapping example for "***indicator_values***" table.



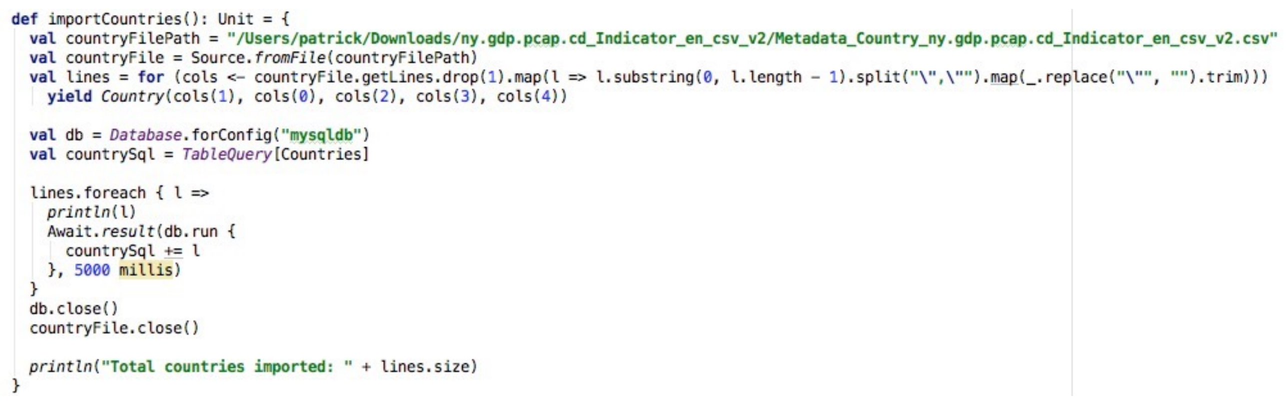
```
case class Indicator_Value(country_code: String, indicator_code: String, year: Int, value: Double, id: Option[Int] = None)

class Indicator_Values(tag: Tag)
  extends Table[Indicator_Value](tag, "indicator_values") {

  // This is the primary key column:
  def id: Rep[Int] = column[Int]("id", O.PrimaryKey, O.AutoInc)

  def country_code: Rep[String] = column[String]("country_code")

  def indicator_code: Rep[String] = column[String]("indicator_code")

  def year: Rep[Int] = column[Int]("year")

  def value: Rep[Double] = column[Double]("value")

  // Every table needs a * projection with the same type as the table's type parameter
  def * = (country_code, indicator_code, year, value, id.?) <> (Indicator_Value.tupled, Indicator_Value.unapply)

  def country: ForeignKeyQuery[Countries, Country] =
    foreignKey("fk_country_code", country_code, TableQuery[Countries])(_.code)

  def indicator: ForeignKeyQuery[Indicators, Indicator] =
    foreignKey("fk_indicator_code", indicator_code, TableQuery[Indicators])(_.code)
}
```

The "***Indicator_Values***" class is the mapping class for the "***indicator_values***" table. The rows fetched are mapped to the "case class ***Indicator_Value***".

III) Import Data into MySQ

After the table is created, the connection information is provided and the relational mapping created, we can import data into MySQL.
Because the "***indicator_values***" table has foreign key constraints, we need to import "***All Countries***" file and "***Indicator***" file first.

To import "***All Countries***" file, we first read data from "***All Countries***" csv file. For each row, we separate the column by colons, then we create a case class to wrap the row. And then we insert this object into MySQL.



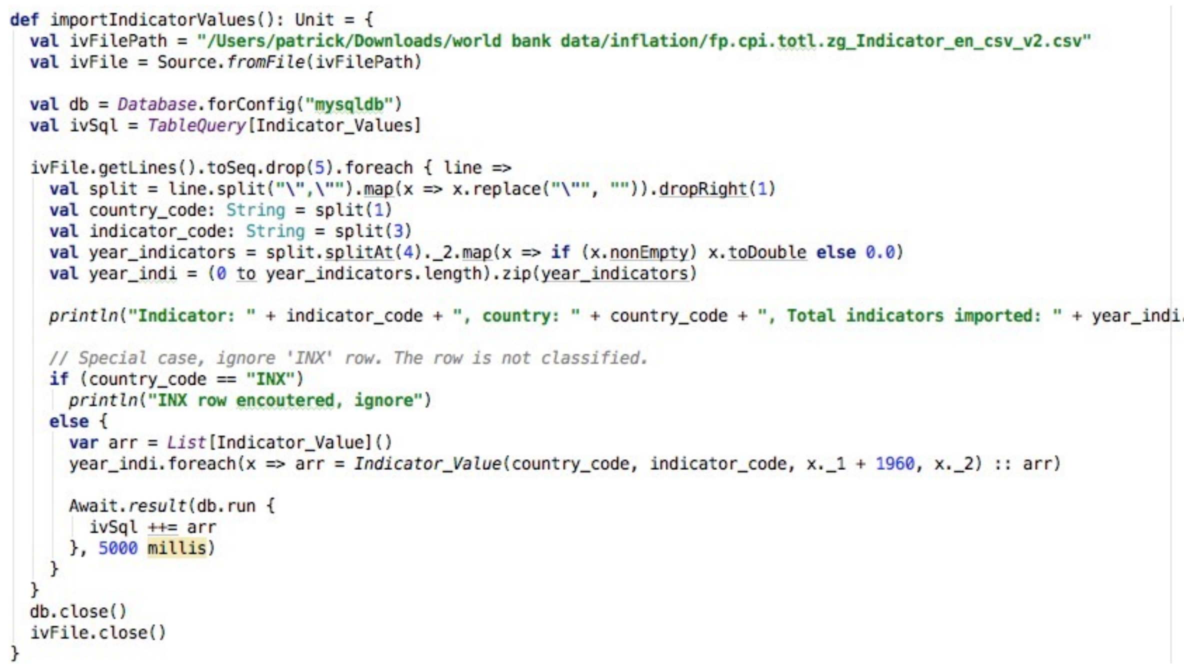
```
def importCountries(): Unit = {
  val countryFilePath = "/Users/patrick/Downloads/ny.gdp.pcap.cd_Indicator_en_csv_v2/Metadata_Country_ny.gdp.pcap.cd_Indicator_en_csv_v2.csv"
  val countryFile = Source.fromFile(countryFilePath)
  val lines = for (cols <- countryFile.getLines.drop(1).map(l => l.substring(0, l.length - 1).split("\",\"").map(_.replace("\"", "").trim)))
    yield Country(cols(1), cols(0), cols(2), cols(3), cols(4))

  val db = Database.forConfig("mysqldb")
  val countrySql = TableQuery[Countries]

  lines.foreach { l =>
    println(l)
    Await.result(db.run {
      countrySql += l
    }, 5000 millis)
  }
  db.close()
  countryFile.close()

  println("Total countries imported: " + lines.size)
}
```

As the insert operation is asynchronous, so we will wait the object is inserted into the database then we insert the second.
(* The operation can be optimized by batch insert.)

Then we can insert the “***indicator***“ file. However, for each indicator, the “***Indicator***” file will contain only one line describing this indicator. And as we only need 6 indicators(GDP, GDP growth, GDP per capital, inflation, import rate, export rate). So we can just insert these indicators in the MySQL console.

```
INSERT INTO `wdb_explorer`.`indicator`
(`indicator_code`,
`indicator_name`,
`indicator_note`,
`indicator_org`)
VALUES
("FP.CPI.TOTL.ZG",
"Inflation, consumer prices (annual %)",
"Inflation as measured by the consumer price index reflects the annual percentage change in the
cost to the average consumer of acquiring a basket of goods and services that may be fixed or
changed at specified intervals, such as yearly. The Laspeyres formula is generally used.",
"International Monetary Fund, International Financial Statistics and data files.");
```

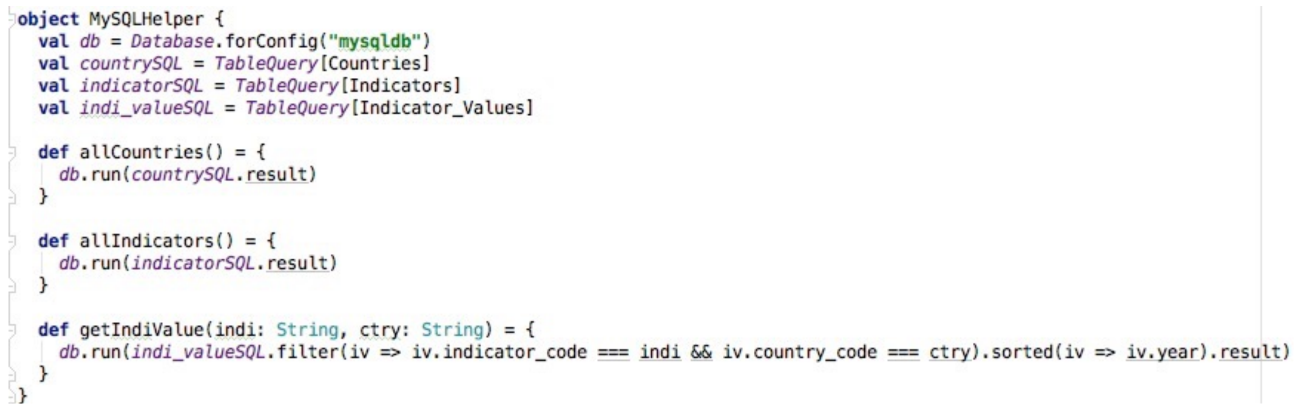
Finally, we can insert the “Indicator Values” file. This step is similar to import “All Countries” Data. After reading data from csv file, we insert the data row by row. For each row, we separate the content by colons, then for each value range from year 1960 to 2015, we can create the Indicator_Value case class and then insert the object.



```scala
def importIndicatorValues(): Unit = {
  val ivFilePath = "/Users/patrick/Downloads/world bank data/inflation/fp.cpi.totl.zg_Indicator_en_csv_v2.csv"
  val ivFile = Source.fromFile(ivFilePath)

  val db = Database.forConfig("mysqldb")
  val ivSql = TableQuery[Indicator_Values]

  ivFile.getLines().toSeq.drop(5).foreach { line =>
    val split = line.split("\",\"").map(x => x.replace("\"", "")).dropRight(1)
    val country_code: String = split(1)
    val indicator_code: String = split(3)
    val year_indicators = split.splitAt(4)._2.map(x => if (x.nonEmpty) x.toDouble else 0.0)
    val year_indi = (0 to year_indicators.length).zip(year_indicators)

    println("Indicator: " + indicator_code + ", country: " + country_code + ", Total indicators imported: " + year_indi.size)

    // Special case, ignore 'INX' row. The row is not classified.
    if (country_code == "INX")
      println("INX row encoutered, ignore")
    else {
      var arr = List[Indicator_Value]()
      year_indi.foreach(x => arr = Indicator_Value(country_code, indicator_code, x._1 + 1960, x._2) :: arr)

      Await.result(db.run {
        ivSql ++= arr
      }, 5000 millis)
    }
  }
  db.close()
  ivFile.close()
}
```

iv) Access the Data

After the data is imported, we can fetch the data we needed.
We create an object *MySQLHelper*, which is responsible for access the data stored in the database.



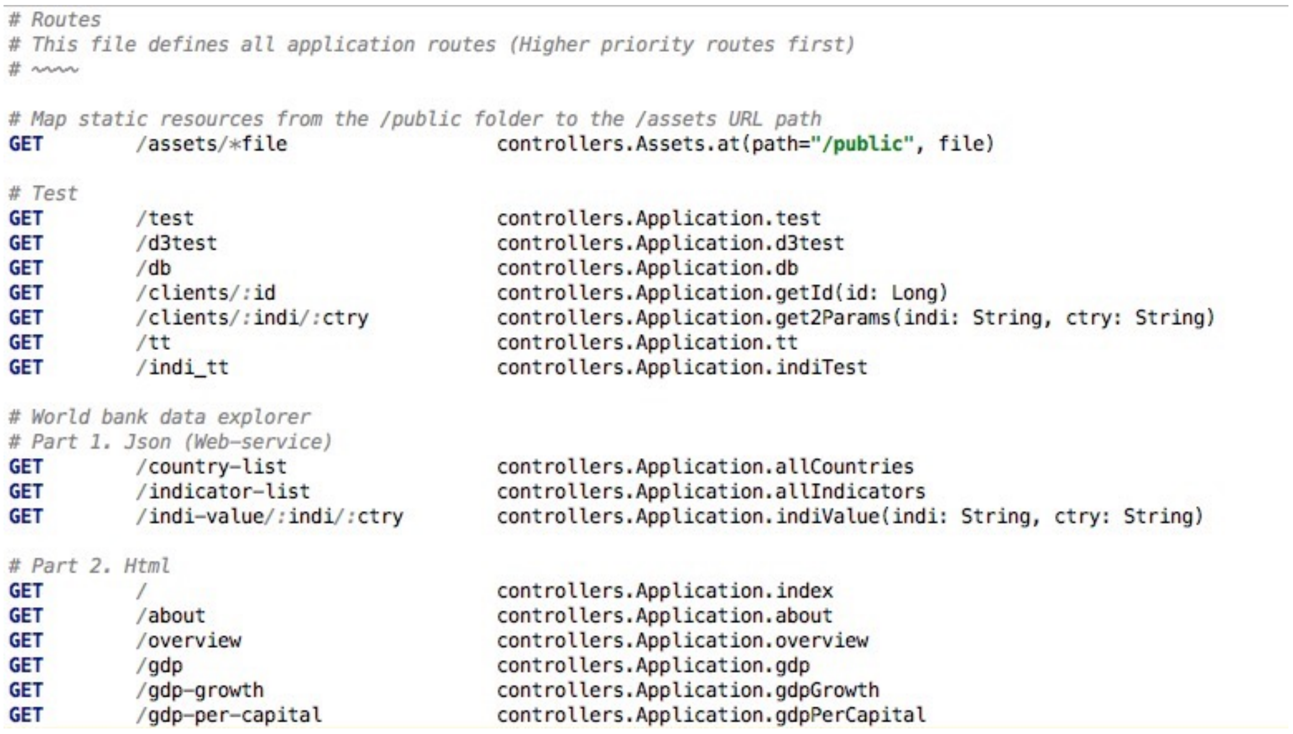
```scala
object MySQLHelper {
  val db = Database.forConfig("mysqldb")
  val countrySQL = TableQuery[Countries]
  val indicatorSQL = TableQuery[Indicators]
  val indi_valueSQL = TableQuery[Indicator_Values]

  def allCountries() = {
    db.run(countrySQL.result)
  }

  def allIndicators() = {
    db.run(indicatorSQL.result)
  }

  def getIndiValue(indi: String, ctry: String) = {
    db.run(indi_valueSQL.filter(iv => iv.indicator_code === indi && iv.country_code === ctry).sorted(iv => iv.year).result)
  }
}
```

(* The code used for testing has been removed, so the final class below contains only the method used in our project.)

Actually, the most frequent used method is `*getIndiValue(indi:String, ctry: String)*`, the *indi* parameter defines which indicator we need, and *ctry* parameter defines which country we are interested. The result is a Future object contains the specified country's indicator values from 1960 to 2015. Then the front-end can render the diagram from these data.

3) Play Framework Design

i) Define Routing Rules



```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# Map static resources from the /public folder to the /assets URL path
GET         /assets/*file               controllers.Assets.at(path="/public", file)

# Test
GET         /test                       controllers.Application.test
GET         /d3test                     controllers.Application.d3test
GET         /db                         controllers.Application.db
GET         /clients/:id                controllers.Application.getId(id: Long)
GET         /clients/:indi/:ctry        controllers.Application.get2Params(indi: String, ctry: String)
GET         /tt                         controllers.Application.tt
GET         /indi_tt                    controllers.Application.indiTest

# World bank data explorer
# Part 1. Json (Web-service)
GET         /country-list               controllers.Application.allCountries
GET         /indicator-list             controllers.Application.allIndicators
GET         /indi-value/:indi/:ctry     controllers.Application.indiValue(indi: String, ctry: String)

# Part 2. Html
GET         /                           controllers.Application.index
GET         /about                      controllers.Application.about
GET         /overview                   controllers.Application.overview
GET         /gdp                        controllers.Application.gdp
GET         /gdp-growth                 controllers.Application.gdpGrowth
GET         /gdp-per-capital            controllers.Application.gdpPerCapital
```

Each row is a routing rule, each rule has 3 parts.
The left part is the method, such as GET / POST / PUT, etc.
The middle part is the url path. There are two kinds of url path here. The first kind is the basic path, for example "*/about*", is an explicit request. The second kind is with parameters, for example, "*/indi-value/:indi/:city*", the "*indi*" and "*city*" parameters can be extracted from the url path, and send them to the request handler.
The right part is the handler. Each handler is a function defined in a Scala object. For example, the "*controllers.Application.indiValue(indi: String, city: String)*" is the function *indiValue(indi: String, city: String)* defined in the object *Application* in package *controllers*.

ii) Serve Static files (js / css / images)

This is a built-in functionality provided by play framework.
We can simply define a routing rule in the "*conf/routes*" file, then can we get the static files easily.
The Route rule:

```
# Map static resources from the /public folder to the /assets URL path
GET       /assets/*file                controllers.Assets.at(path="/public", file)
```

The route rule defines that when a url path matches "*/assets/*file*", the *"file"* argument will be mapped to a file in the "*/public*" directory.

This is how to access a css file in html:

```
<link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/main.css")">
```

We can simply use the *@routes.Assets.at("folder_in_public_foler/file")* to get the target file.

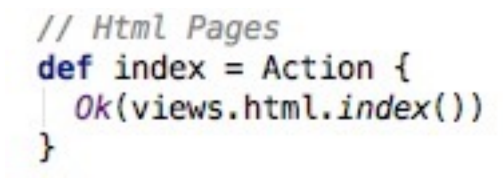
iii) Handle Request

Each request will be sent to the target handler if the request path matches the routing rules. Otherwise, the request is invalid, a 404 page will return.

For example, let's see how the index page request is handled in play framework.
This is the routing rule:

```
GET      /                        controllers.Application.index
```

So when a user input the http://ip-to-our-server, the request will be sent to Application's index method. This is how the *index* method looks like:



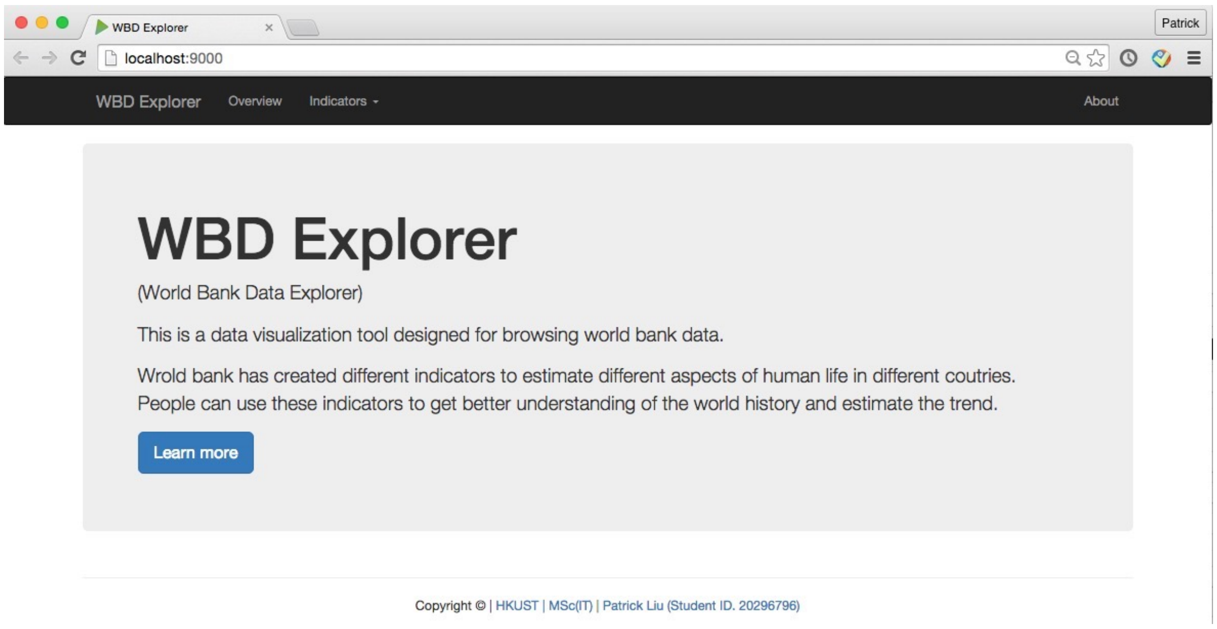
```
// Html Pages
def index = Action {
  Ok(views.html.index())
}
```

We just return the "*index*" page back to the browser.

Now we have data stored in the database, and a web server in the backend. Next, we will see how the front-end page is designed.

4) Front-end Web Page Design

We use "Bootstrap" to do layout.
For example, this is the home page for the project. There is a navigation bar on the top, which contains the `project title`, `overview`, `3 Indicator pages` and `about` page. In the middle part are the content. And there is a footer in the bottom.



WBD Explorer

(World Bank Data Explorer)

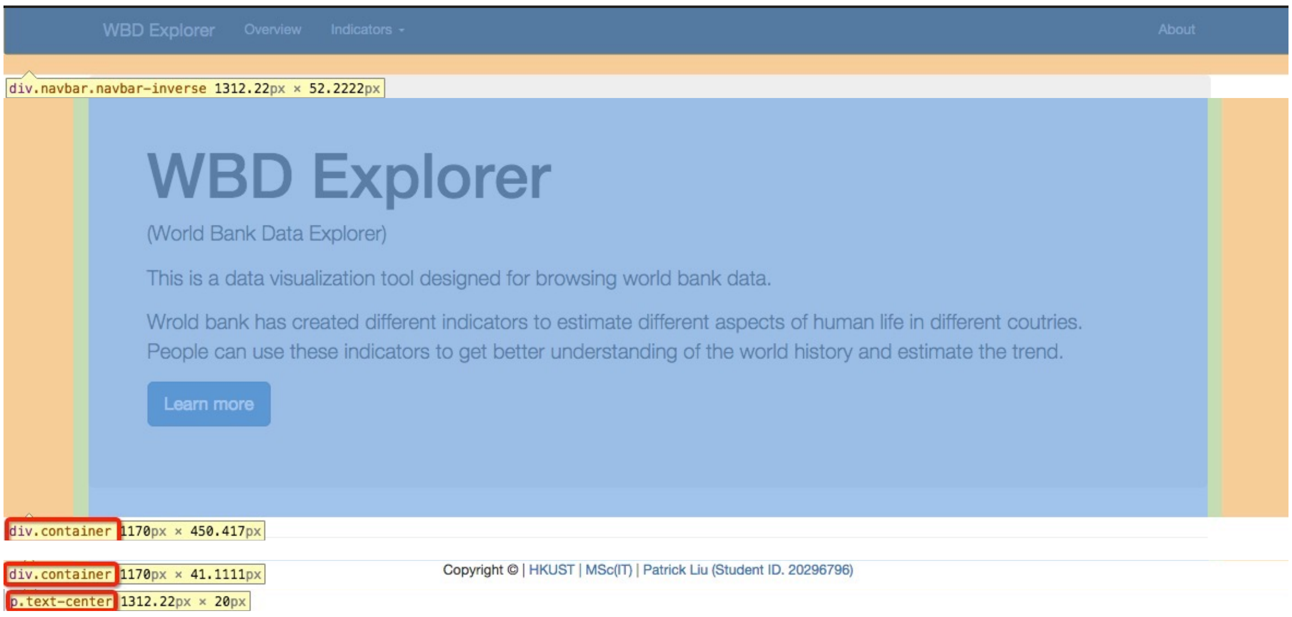
This is a data visualization tool designed for browsing world bank data.

Wrold bank has created different indicators to estimate different aspects of human life in different coutries. People can use these indicators to get better understanding of the world history and estimate the trend.

Learn more

Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

Basically, the webpage is consists of 4 parts, highlighted below:


WBD Explorer
Overview
Indicators
About
div.navbar.navbar-inverse 1312.22px × 52.2222px
Learn more
div.container 1170px × 450.417px
div.container 1170px × 41.1111px
p.text-center 1312.22px × 20px

WBD Explorer

(World Bank Data Explorer)

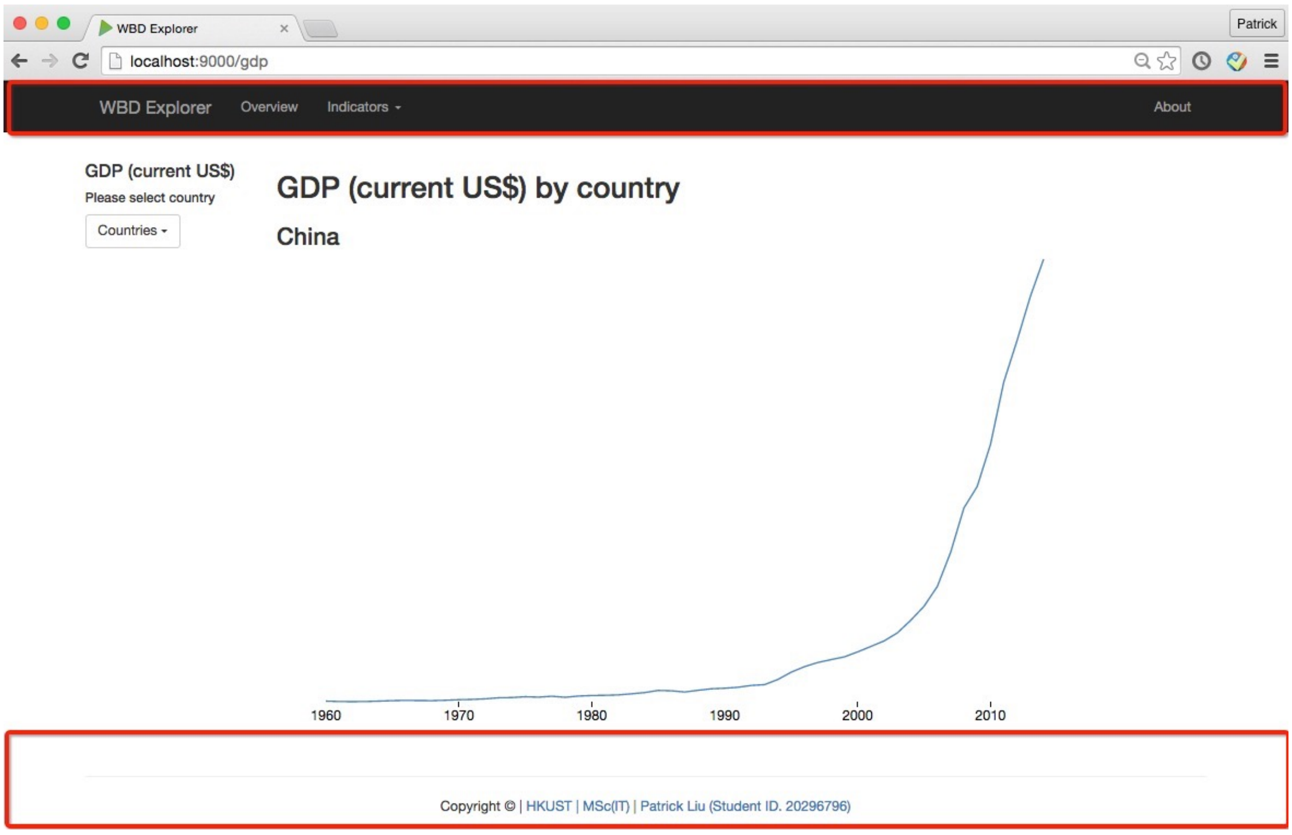
This is a data visualization tool designed for browsing world bank data.

Wrold bank has created different indicators to estimate different aspects of human life in different coutries. People can use these indicators to get better understanding of the world history and estimate the trend.

Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

There is a navigation bar on the top, a container containing all the content in the middle, another container contains a line to separate the content and footer, then finally the footer.

However, the “gdp indicator” page has the similar layout.



Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

If we don't extract the common parts out side of these webpages, we will have duplicates spread through the project. It is hard to maintain.

These two pages have same header and footer. We can create a template html file called "common" html, which contains the duplicated parts. And left the differentiate part open. Then we just need to feed the middle part to generate these different pages. Luckily, this can be implemented by Play's template system easily.

For simplification, this is the "*common.scala.html*":

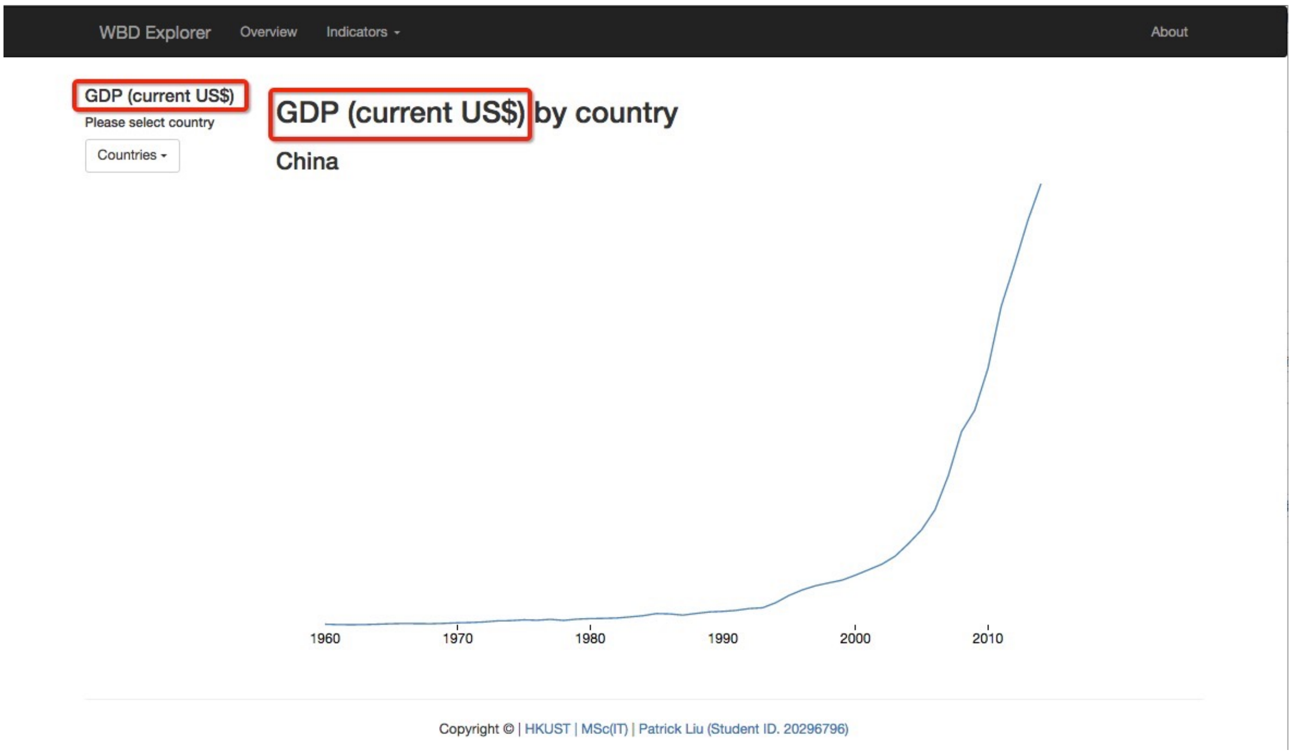
```
@(content: Html)
<!DOCTYPE html>
<html lang="zh-CN">
    <head>
        ... header content ...
    </head>
    <body>
        <div class="navbar navbar-inverse ">
            ... navbar content ...
        </div>

        @content

        <div class="container">
            <hr>
        </div>
        <footer>
            ... footer content ...
        </footer>
    </body>
</html>
```

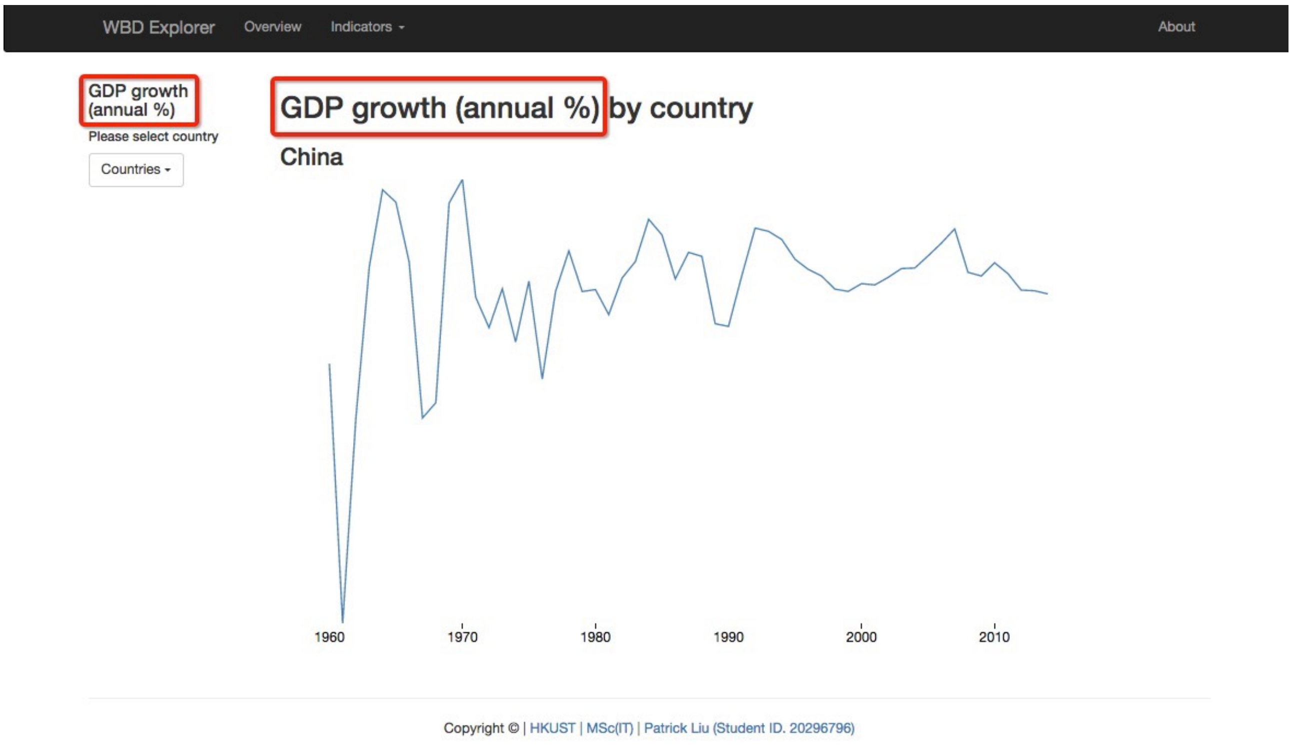
(* The @(content: Html) defines a parameter "*content*" which is a html element, and this element will be placed in the middle part of the "*common*" html file)

Then, in the "*indexscala.html*" file, we can simply add the "*content*" and send it back to browser:

```
@common {
  <div class="container">
    <div class="jumbotron">
      <h1>WBD Explorer</h1>
      <p>(World Bank Data Explorer)</p>
      <p>This is a data visualization tool designed for browsing world bank 
data.</p>
      <p>Wrold bank has created different indicators to estimate different 
aspects of human life in different coutries.
        People can use these indicators to get better understanding of the 
world history and estimate the trend.</p>
      <p><a class="btn btn-primary btn-lg" href="/about" role="button">Learn 
more</a></p>
    </div>
  </div>
}
```
(* The index page called returns a common page, but use the content inside the brace to fill the @content part)

With more observation, we have found out that the "*gdp.scala.html*" page and "*gdp_growth.scala.thml*" page have similar layout too.





The only different is the title and the data points. So we can create another template html from these them.

The new template will based on "*common*" template, but with more common information in the @content part.

The new template "*indi_common.scala.html*"

```
@(indicator: String, baseUrl: String)
@common {
    <div class="container">
        <div class="row">
            <div class="col-md-2">
                <h4>@indicator</h4>
                <h6>Please select country</h6>
                <div class="dropdown">
                    <button class="btn btn-default dropdown-toggle" 
type="button" id="dropdownMenu1" data-toggle="dropdown" aria-haspopup="true" 
aria-expanded="true">
                        Countries
                        <span class="caret"></span>
                    </button>
                    <ul id="country-select" class="dropdown-menu" aria-
labelledby="dropdownMenu1">
                        <li><a href="#">China</a></li>
                        <li><a href="#">Japan</a></li>
                        <li><a href="#">Korea</a></li>
                        <li><a href="#">Brazil</a></li>
                        <li><a href="#">USA</a></li>
                        <li><a href="#">Norway</a></li>
                    </ul>
                </div>
            </div>
            <div id="display" class="col-md-10">
                <h2>@indicator by country</h2>
                <h3 id="current-country"></h3>
            </div>
        </div>
    </div>
}
```

The template fills the @content part with a solid structure, left 2 places for the input arguments. The first argument "*indicator*" defines the title of the page, the second argument "*baseUrl*" defines where can indicator's data can be fetched.

Then we can generate the gdp page with one sentence:
```
@indi_common("GDP (current US$)", "NY.GDP.MKTP.CD")
```

And gdp_growth page with another sentence:
```
@indi_common("GDP growth (annual %)", "NY.GDP.MKTP.KD.ZG")
```

5) Render Data

D3js is used to render data points.
For example, we want to render China's gdp diagram in the past 55 years.

First we have the data:

# id	country_code	indicator_code	year	value
76312'	'CHN'	NY.GDP.MKTP.CD'	'2014'	'10360105247908.3'
76313'	'CHN'	NY.GDP.MKTP.CD'	'2013'	'9490602600148.49'
76314'	'CHN'	NY.GDP.MKTP.CD'	'2012'	'8461623162714.07'
76315'	'CHN'	NY.GDP.MKTP.CD'	'2011'	'7492432097810.11'
…	…	…	…	…

Then we need to do the following to render these data:

I) Find where to attach the graph

```
var svg = d3.select(node).append("svg")
     .attr("width", width + margin.left + margin.right)
     .attr("height", height + margin.top + margin.bottom)
     .attr("transform", "translate(" + margin.left + "," + margin.top + “)");
```

ii) Define the size of the graph

```
var margin = {top: 20, right: 20, bottom: 30, left: 50},
      width = full_width - margin.left - margin.right,
      height = full_height - margin.top - margin.bottom;
```

(iii) Scale and select the data to meet the graph size

```
var x = d3.scale.linear().range([0, width]);

var y = d3.scale.linear().range([height, 0]);

var xAxis = d3.svg.axis()
       .scale(x)
       .tickFormat(d3.format("d"))
       .ticks(5)
       .orient("bottom");

var yAxis = d3.svg.axis()
       .scale(y)
       .orient(“left");
```

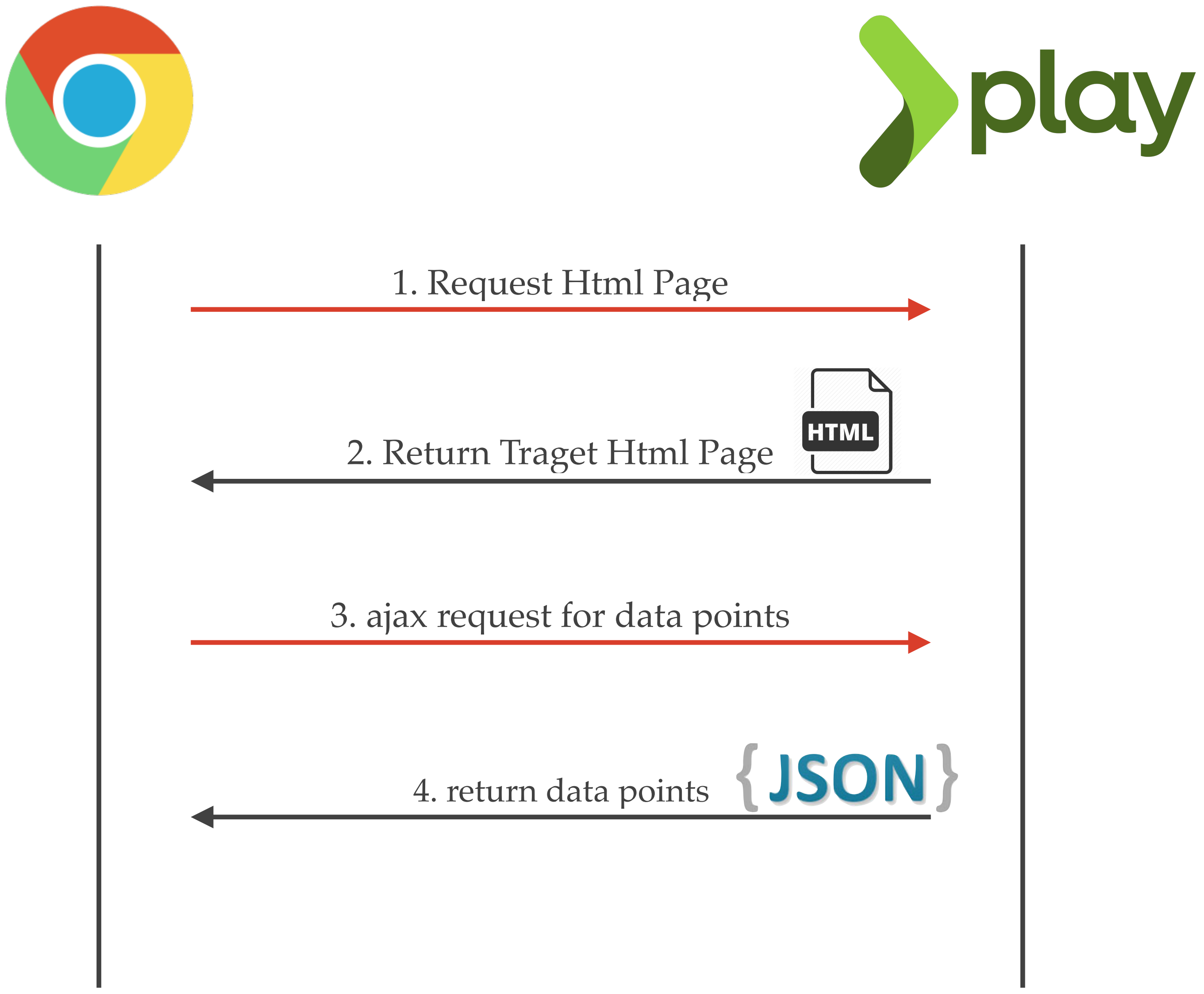
iv) Feed the data

```
d3.json(url, function(error, data) {
if (error) throw error;

// After data is loaded, can we calculate the domain for xAxis and yAxis.
x.domain(d3.extent(data, function(d) { return d.year; }));
y.domain(d3.extent(data, function(d) { return d.value; }));

 svg.append("g")
     .attr("class", "x axis")
     .attr("transform", "translate(" + margin.left + "," + height + ")")
     .call(xAxis);

 svg.append("g")
     .attr("class", "y axis")
     .call(yAxis)
```

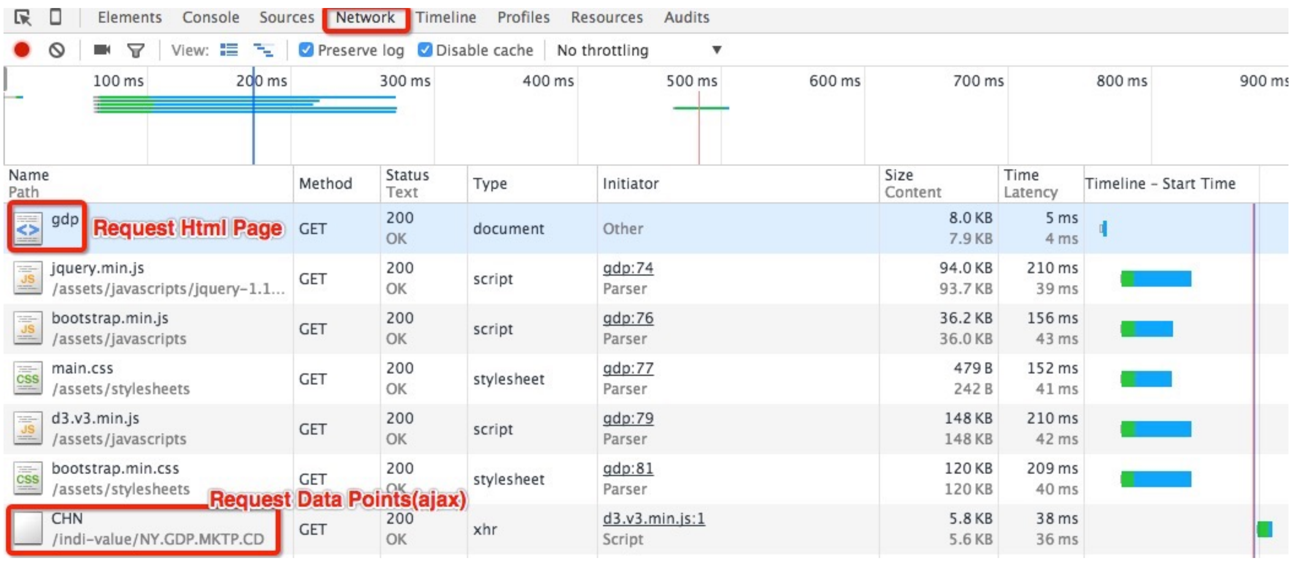
```
            .append("text")
            .attr("transform", "rotate(-90) translate(" + 2*margin.left + "," + 0 + ")")
            .attr("y", 6)
            .attr("dy", ".71em")
            .style("text-anchor", "end")
            .text("Price ($)");

        svg.append("path")
            .datum(data)
            .attr("class", "line")
            .attr("transform", "translate(" + margin.left + "," + 0 + ")")
            .attr("d", line);
  });
```

6) Combine front-end and back-end together

Here is the work-flow between browser and server:



You can see the network request sequence in Chrome's developer console:

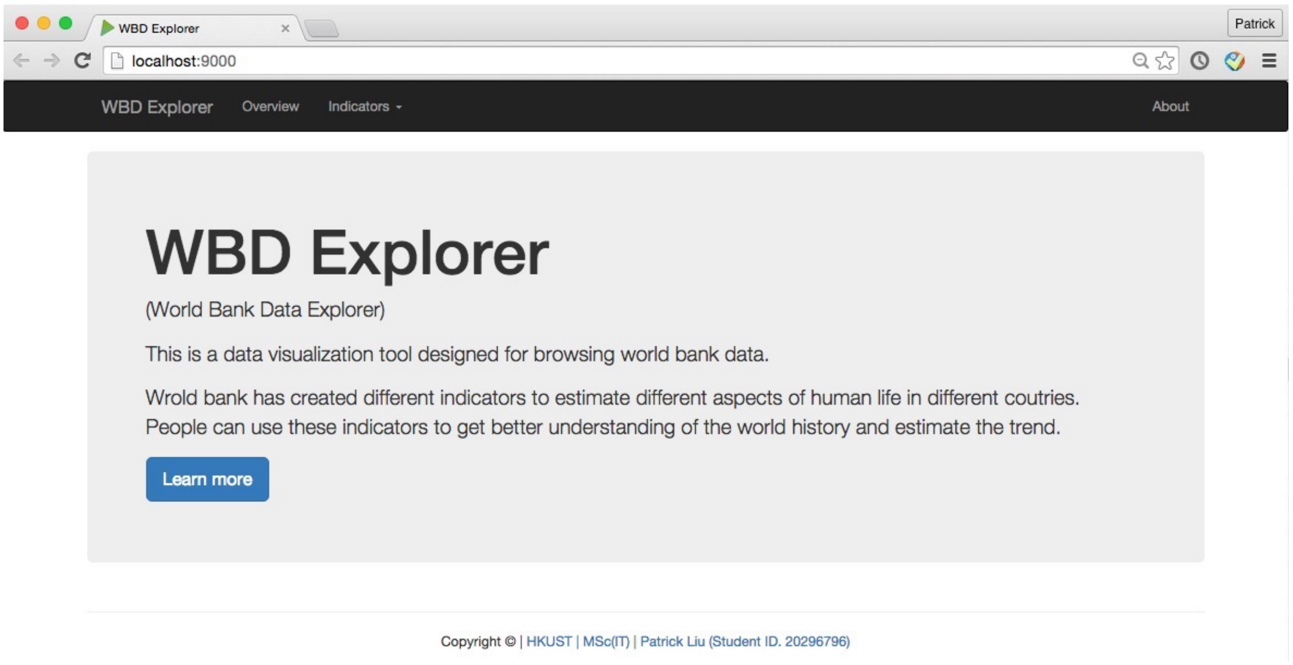


Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline – Start Time
gdp	GET	200 OK	document	Other	8.0 KB 7.9 KB	5 ms 4 ms	
jquery.min.js /assets/javascripts/jquery-1.1...	GET	200 OK	script	gdp:74 Parser	94.0 KB 93.7 KB	210 ms 39 ms	
bootstrap.min.js /assets/javascripts	GET	200 OK	script	gdp:76 Parser	36.2 KB 36.0 KB	156 ms 43 ms	
main.css /assets/stylesheets	GET	200 OK	stylesheet	gdp:77 Parser	479 B 242 B	152 ms 41 ms	
d3.v3.min.js /assets/javascripts	GET	200 OK	script	gdp:79 Parser	148 KB 148 KB	210 ms 42 ms	
bootstrap.min.css /assets/stylesheets	GET	200 OK	stylesheet	gdp:81 Parser	120 KB 120 KB	209 ms 40 ms	
CHN /indi-value/NY.GDP.MKTP.CD	GET	200 OK	xhr	d3.v3.min.js:1 Script	5.8 KB 5.6 KB	38 ms 36 ms	

So basically, the browser send a request to the webServer, the server returns the html page. When the html page is rendered in the browser, the javascript will trigger another ajax request to the web server to fetch data. These data are returned in JSON format, then the javascript will parse the data and feed them to d3js to be rendered.

5. Result

1) Home Page



WBD Explorer

(World Bank Data Explorer)

This is a data visualization tool designed for browsing world bank data.

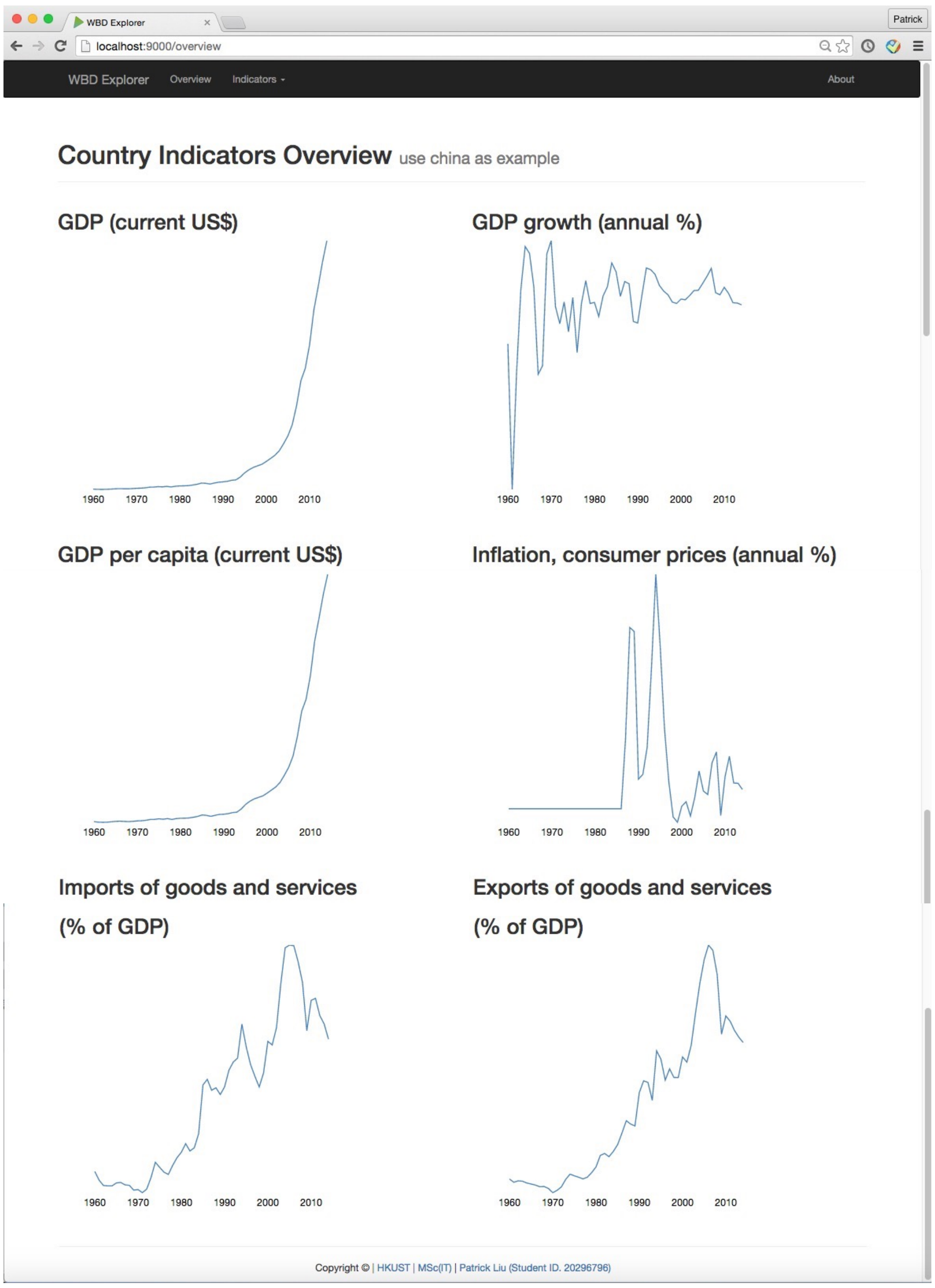
Wrold bank has created different indicators to estimate different aspects of human life in different coutries. People can use these indicators to get better understanding of the world history and estimate the trend.

Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

This is the home page for the project. It describe the project "WBD Explorer"(short for World bank data explorer) and it's basic background. User can click "Learn more" to go to the "About" page.

The navigation bar is always at the top of the website. User can navigate freely. In the bottom, there are links link to “HKUST” and “MSc(IT) program” pages. When user clicks the “Patrick Liu” link, a mail box will pop-up for them to write email to Patrick.
(* This templates is extracted from the “Leetcode” home page.)

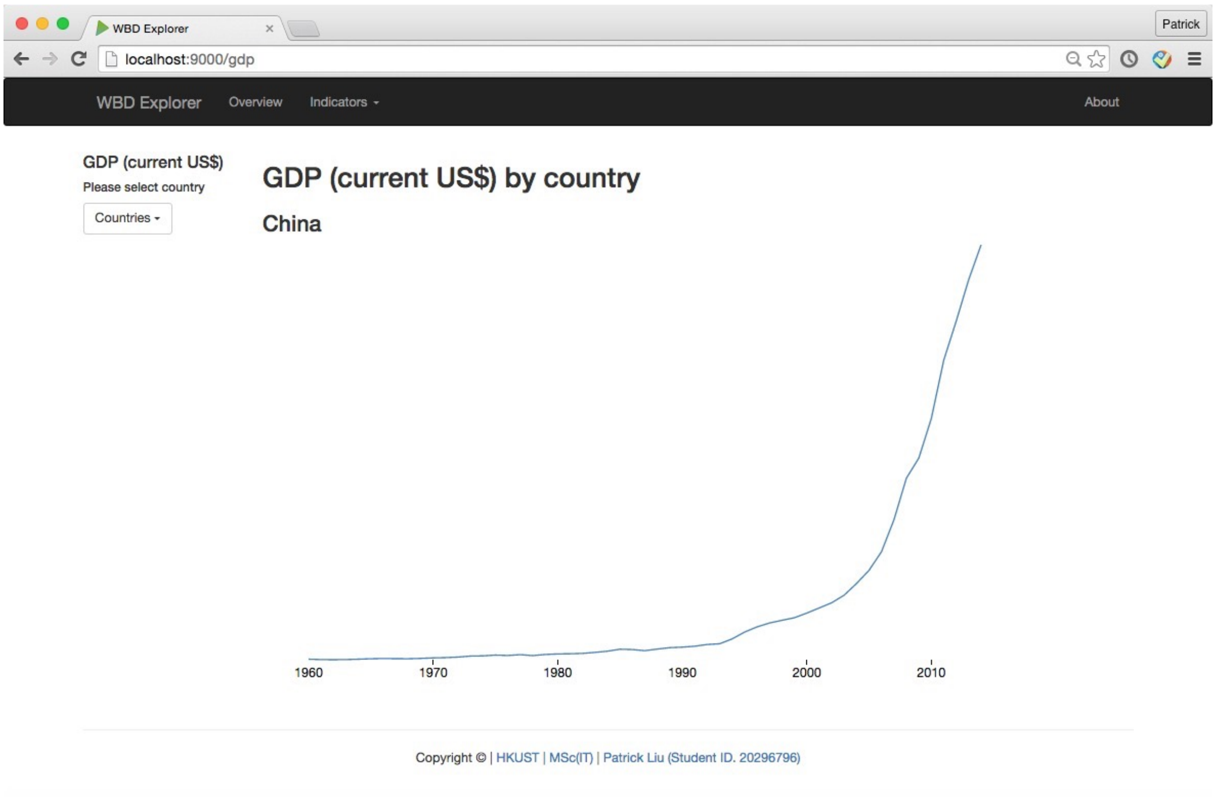
2) Overview Page



WBD Explorer
Patrick
localhost:9000/overview
WBD Explorer
Overview
Indicators
About
Country Indicators Overview use china as example
GDP (current US$)
GDP growth (annual %)
GDP per capita (current US$)
Inflation, consumer prices (annual %)
Imports of goods and services (% of GDP)
Exports of goods and services (% of GDP)
1960
1970
1980
1990
2000
2010
Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

This is the overview page. This page is like a Dashboard. I use China as example. User can see 6 different Indicators. They are GDP (current US$), GDP growth (annual %), GDP per capita (current US $), Inflation, consumer prices (annual %), Imports of goods and services(% of GDP), Exports of goods and services(% of GDP).

3) GDP Page



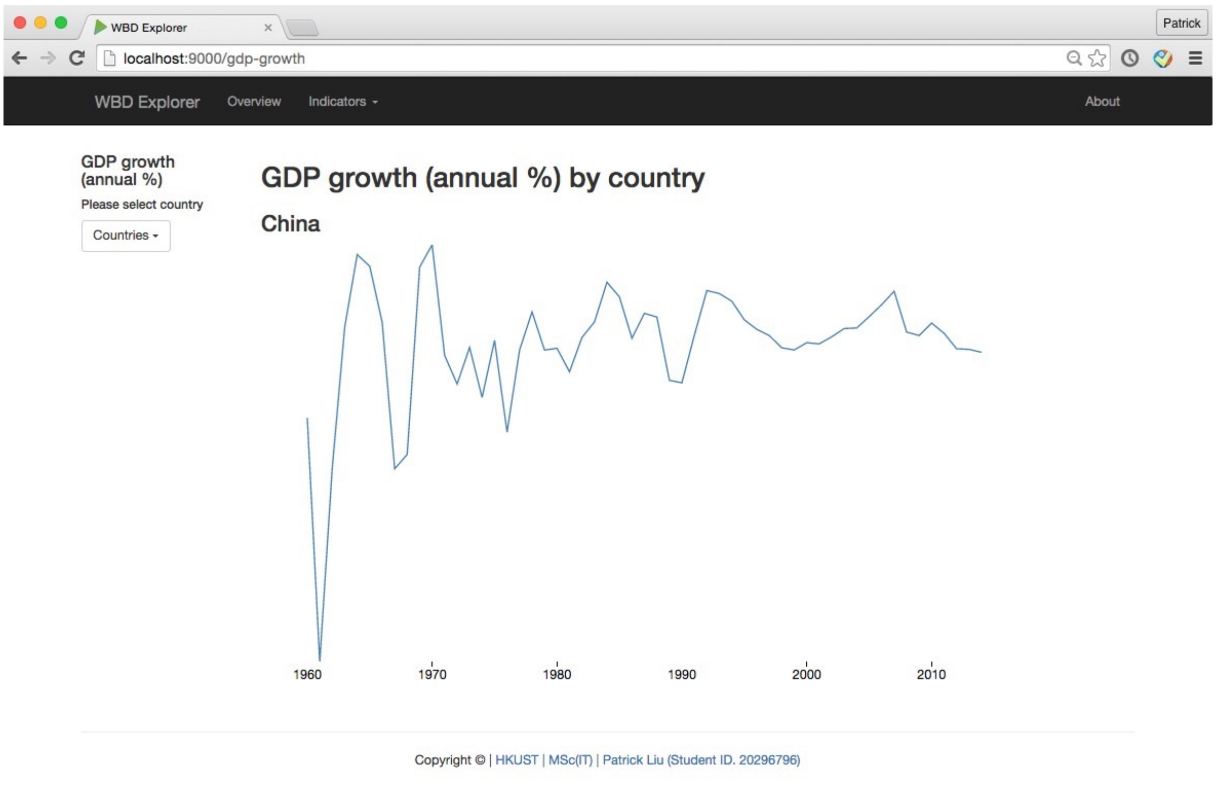
GDP (current US$) by country

China

Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

In this page, user can see a country's GDP (current US$) indicator values from 1960 to 2015 with a bigger picture. User can also select countries in the left drop-down menu.

4) GDP Growth Page



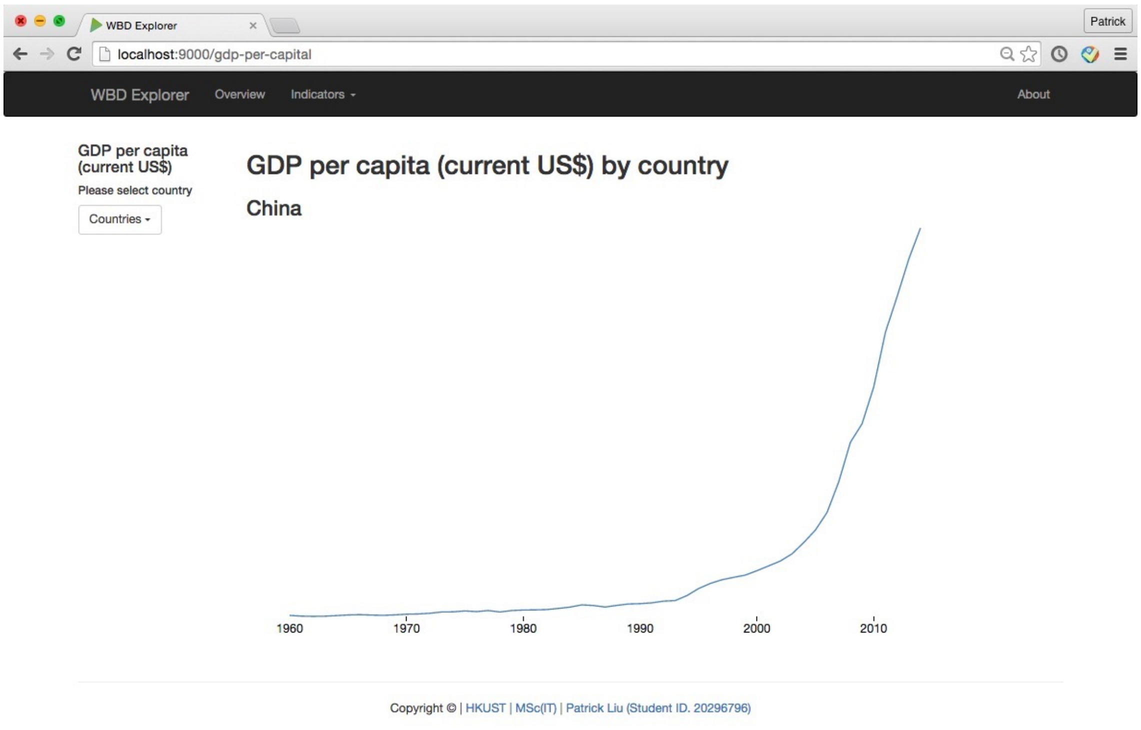
GDP growth (annual %) by country

China

Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

This page is similar to GDP page. User can browse different countries' GDP growth data.

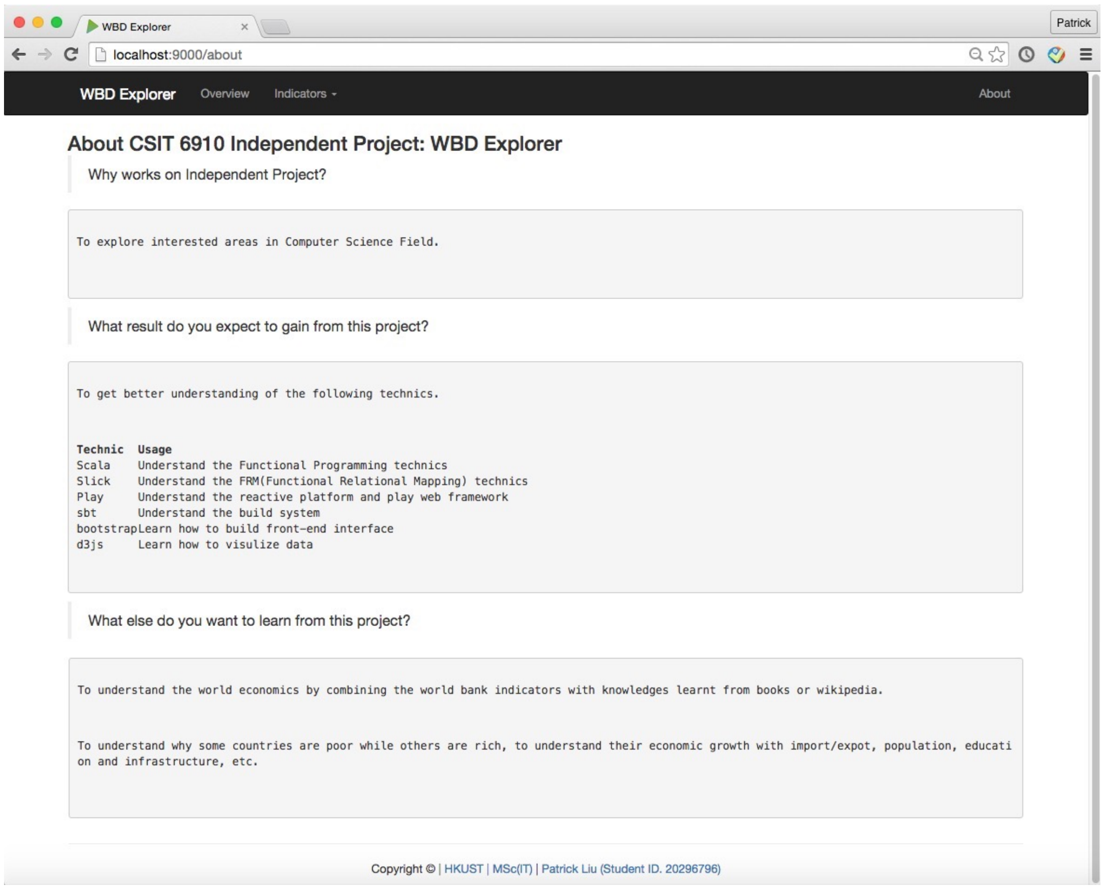
5) GDP Per Capital Page



WBD Explorer
Patrick
localhost:9000/gdp-per-capital
WBD Explorer
Overview
Indicators
About
GDP per capita (current US$)
Please select country
Countries
GDP per capita (current US$) by country
China
1960
1970
1980
1990
2000
2010
Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

This page is similar to GDP page. User can browse different countries' GDP per capital data.

6) About Page



About CSIT 6910 Independent Project: WBD Explorer

Why works on Independent Project?

```
To explore interested areas in Computer Science Field.
```

What result do you expect to gain from this project?

```
To get better understanding of the following technics.


Technic  Usage
Scala    Understand the Functional Programming technics
Slick    Understand the FRM(Functional Relational Mapping) technics
Play     Understand the reactive platform and play web framework
sbt      Understand the build system
bootstrapLearn how to build front-end interface
d3js     Learn how to visulize data
```

What else do you want to learn from this project?

```
To understand the world economics by combining the world bank indicators with knowledges learnt from books or wikipedia.


To understand why some countries are poor while others are rich, to understand their economic growth with import/expot, population, educati
on and infrastructure, etc.
```

Copyright © | HKUST | MSc(IT) | Patrick Liu (Student ID. 20296796)

6. Conclusion

After doing this project. I have created a web site from scratch. It is not easy to learn that much content from zero to one. (jQuery, bootstrap, d3js, Scala, Slick, Play, etc). The whole project has been through many explores and tests. However, I have learnt a lot from this project.
I have learnt the economical function of World bank by analyzing its website, reading its yearly report and wikipedia articles. Then I have learnt basic economic trend of some selected countries by analyzing their indicator values. Meanwhile, I have learnt to use the BootStrap to create a webpage layout quickly and d3js to do data visualization.
Most importantly, I have learnt how to use Scala more efficiently. Though the Slick library is more like a black box due to lack of articles or blogs on it, I have explored and write tests to understand it. And luckily, I have learnt nearly the whole Tyrpesafe Scala ecosystem.

7. Minutes

In this semester I have 2 meetings with Prof. David Rossiter. There should be 3 meetings, but 1 meeting was postponed because I was analyzing the World bank website and its data in economics perspective, so there's nothing more to show in the meeting, so finally the meeting is cancelled.

The 1st meeting minutes:

Item	Detail
Date	2015-10-13
Time	11:10 am
Place	Room 3512
Attending	Prof. Rossiter Liu Yufan
Absent	None
Recorder	Liu Yufan
Approval of minutes	This is the first formal Meeting, so there were no minutes to approve.
Report	In this meeting, we go though the project proposal, estimate the project management time table, discuss the expected output and techniques required, set milestones.

The 2nd meeting minutes:

Item	Detail
Date	2015-11-19
Time	14:00 pm
Place	Room 3512
Attending	Prof. Rossiter Liu Yufan
Absent	None

Item	Detail
Recorder	Liu Yufan
Approval of minutes	This is the first formal Meeting, so there were no minutes to approve.
Report	Demonstrate the project progress with a keynote slides. The techniques required in this project has been explored and tested. However, they are still in pieces, so I have to hurry up to combine them together to become a whole project. Discuss the implementation details, estimate workload and decide that 2 advanced features may not be implemented due to time pressure.

8. Appendix

[1] http://www.scala-lang.org/
[2] http://docs.scala-lang.org/index.html
[3] https://twitter.github.io/scala_school/
[4] https://www.coursera.org/course/progfun
[5] https://www.playframework.com/
[6] http://akka.io/
[7] http://slick.typesafe.com/
[8] http://www.scala-sbt.org/0.13/tutorial/index.html
[9] https://jquery.com/
[10] http://getbootstrap.com/
[11] http://d3js.org/
[12] http://www.worldbank.org/
[13] http://data.worldbank.org/