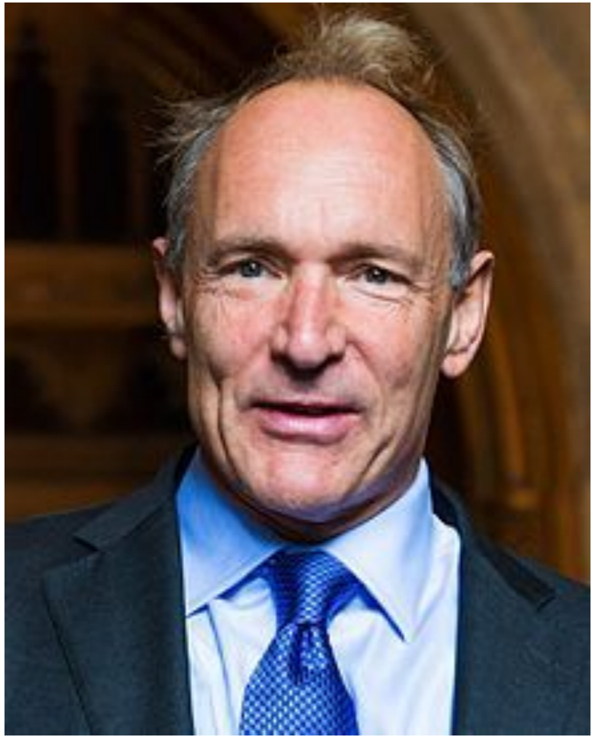
MONTANA
STATE UNIVERSITY

Web Programming

The Web

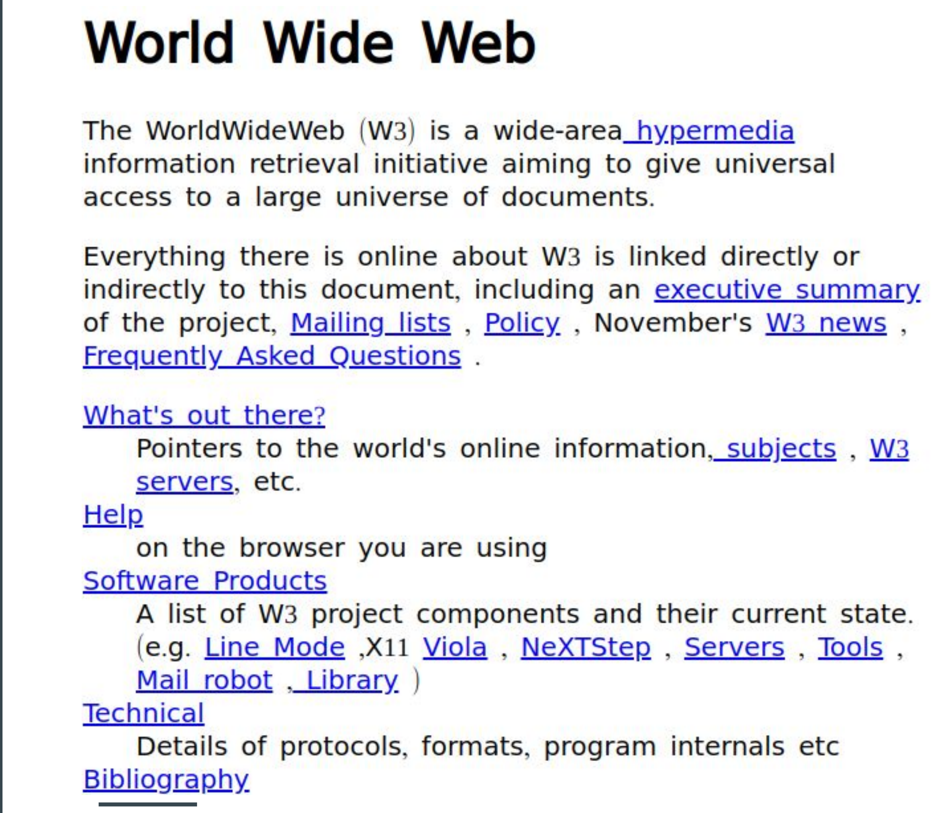
- Maybe you’ve heard of it
- Tim Berners-Lee is considered the father of the web
  - Worked for CERN
  - Developed HTTP and HTML
  - It’s **HIS** fault
- First non-CERN web servers deployed in 1990



The Web

- Recently CERN put the very first web page back up on the net:

  http://info.cern.ch/hypertext/WWW/TheProject.html



World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
: Pointers to the world's online information, subjects , W3 servers, etc.

Help
: on the browser you are using

Software Products
: A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)

Technical
: Details of protocols, formats, program internals etc

Bibliography

The Web

- HTTP: HyperText Transfer Protocol
- A mechanism for transferring hypertext documents from a web server to web clients (typically browsers)

HTTP Request

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

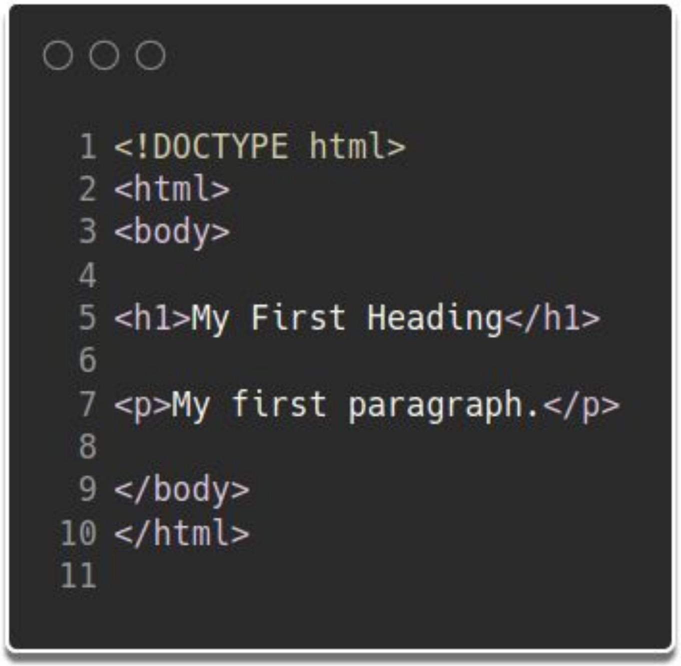
HTTP Response

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

```
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

The Web

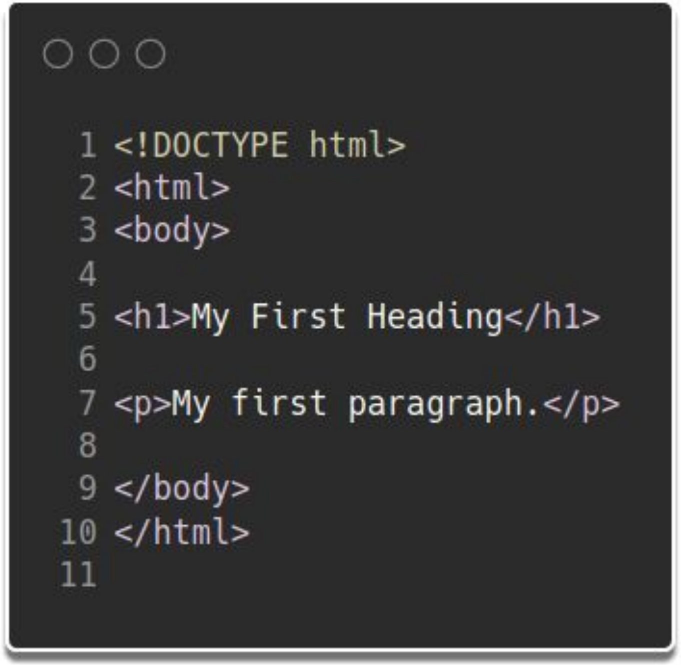
- HTML: HyperText Markup Language
- A language for describing hypertext documents
- Influence by XML
    - but not a proper XML



```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>

```

The Web

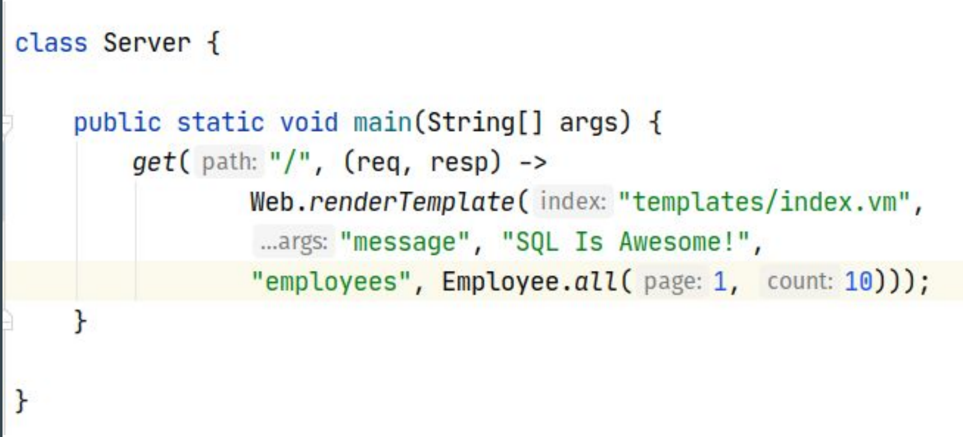
- **NB** - we are going to stay very basic for our HTML
    - You will not be judged on the styling
    - Avoid complex HTML nesting for layout reasons
    - I want to focus on the data in the project



```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>

```

Our Web Toolkit

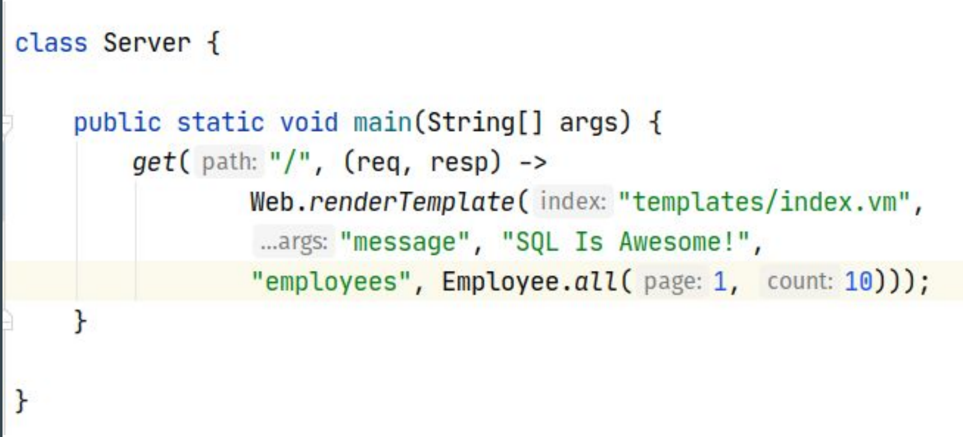
- We are going to be working with a somewhat unique web server setup
  - Java as our language
  - SparkJava as our web server
  - Velocity as our templates
- You will probably not work with this professionally
- *BUT* this is a good platform for demonstrating concepts



```
class Server {

    public static void main(String[] args) {
        get( path: "/", (req, resp) ->
                Web.renderTemplate( index: "templates/index.vm",
                 ...args: "message", "SQL Is Awesome!",
                "employees", Employee.all( page: 1,  count: 10)));
    }

}
```

Spark Java

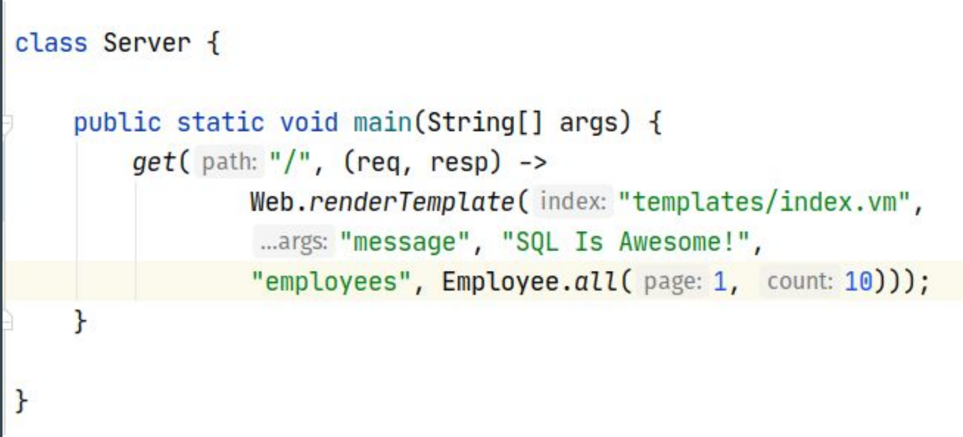
- We are going to be using SparkJava as our web server
  - It’s written in Java
  - It’s pretty simple to work with
  - It doesn’t hide much of the HTTP/HTML loop from you



```
class Server {

    public static void main(String[] args) {
        get( path: "/", (req, resp) ->
                Web.renderTemplate( index: "templates/index.vm",
                ...args: "message", "SQL Is Awesome!",
                "employees", Employee.all( page: 1,  count: 10)));
    }

}
```

Spark Java

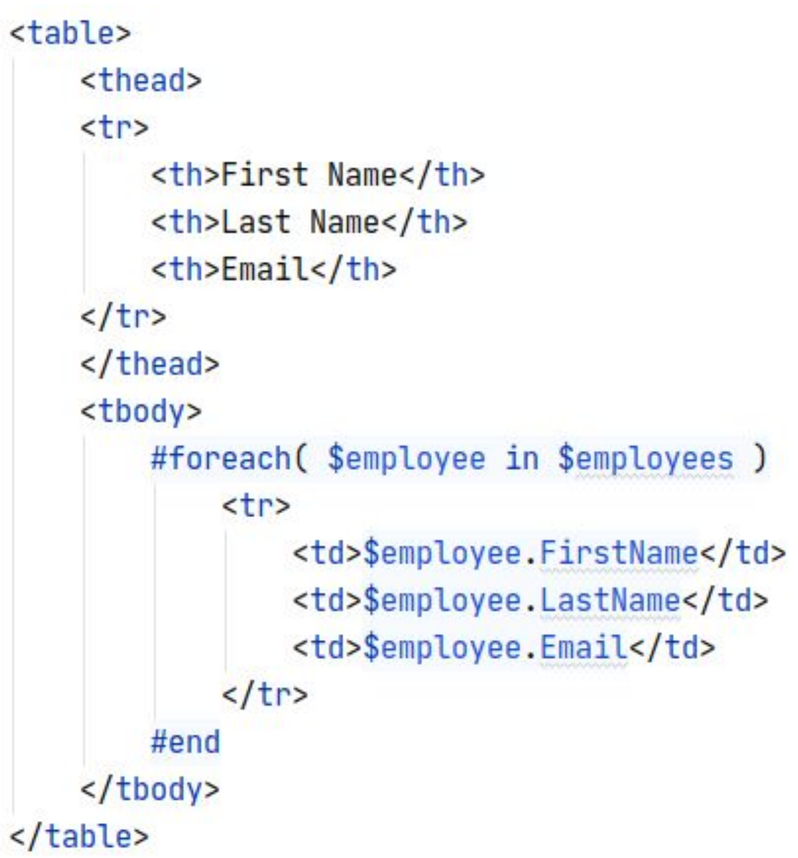
- The core of SparkJava is mapping request URLs to response strings
- In the example we have, we are mapping the path “/” to a string produced by the template *index.vm*



```
class Server {

    public static void main(String[] args) {
        get( path: "/", (req, resp) ->
                Web.renderTemplate( index: "templates/index.vm",
                 ...args: "message", "SQL Is Awesome!",
                "employees", Employee.all( page: 1,  count: 10)));
    }

}
```

Velocity Templates

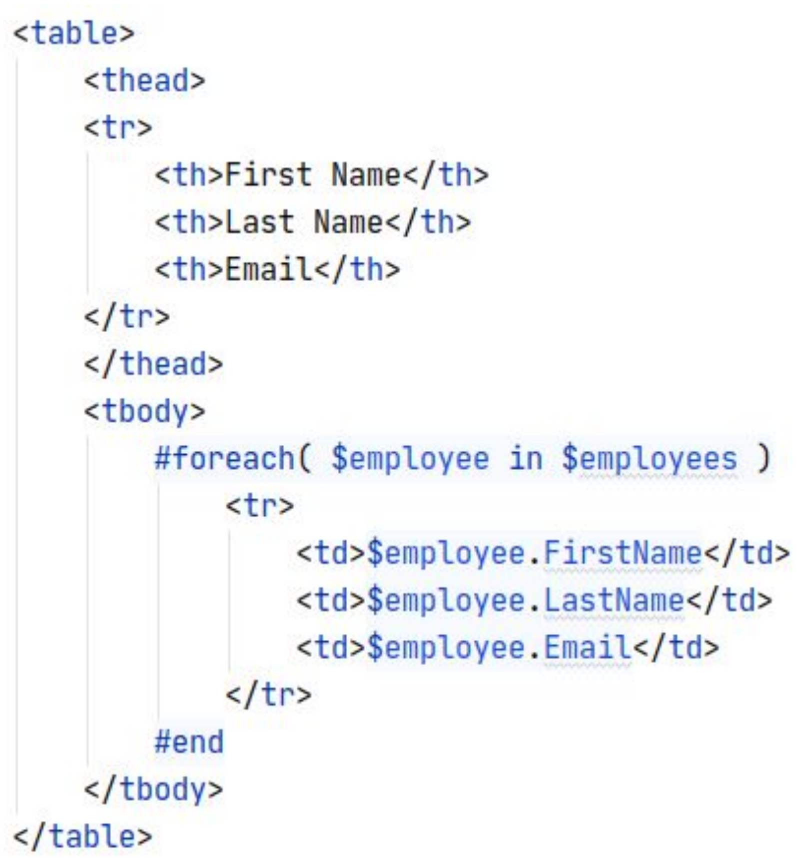
- We are going to be using velocity templates
- Velocity templates are a mature template library for java
- Templates allow you to create dynamic string content more conveniently than concatenating strings together



```
<table>
    <thead>
    <tr>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Email</th>
    </tr>
    </thead>
    <tbody>
        #foreach( $employee in $employees )
            <tr>
                <td>$employee.FirstName</td>
                <td>$employee.LastName</td>
                <td>$employee.Email</td>
            </tr>
        #end
    </tbody>
</table>
```

Velocity Templates

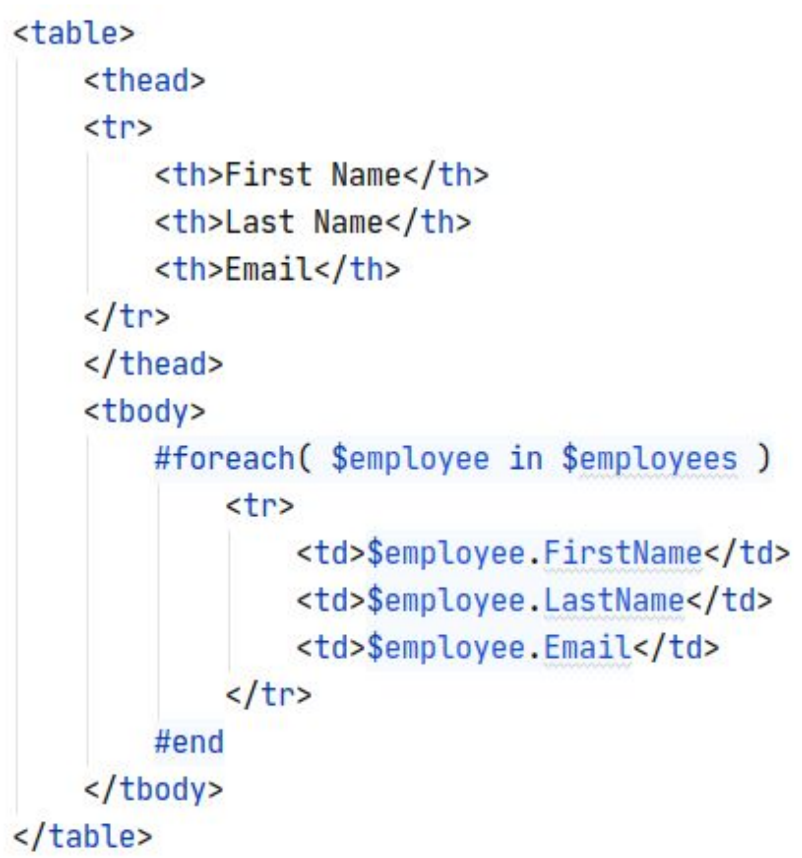
- Velocity template basics:
    - $ - refer to a variable
        - $! - null safe
    - #foreach - a loop macro
    - #if/#elseif/#lse - conditional macro
    - #parse - includes another template



```
<table>
    <thead>
    <tr>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Email</th>
    </tr>
    </thead>
    <tbody>
        #foreach( $employee in $employees )
            <tr>
                <td>$employee.FirstName</td>
                <td>$employee.LastName</td>
                <td>$employee.Email</td>
            </tr>
        #end
    </tbody>
</table>
```

Velocity Templates

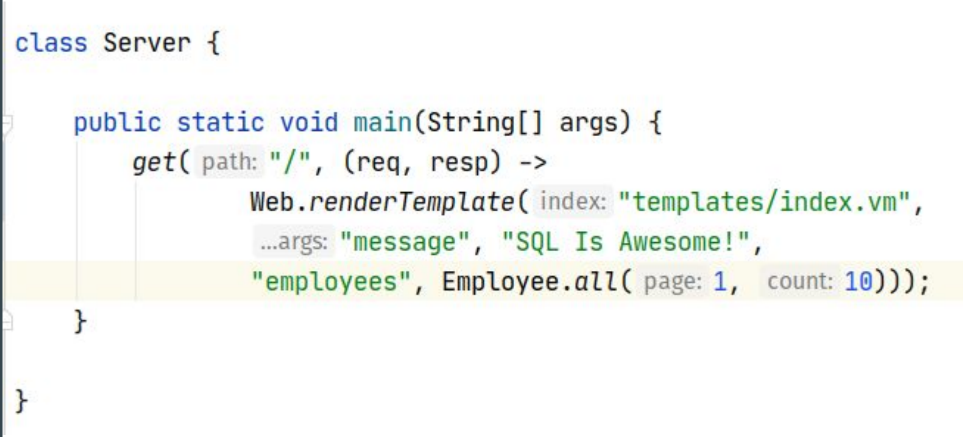
- In this example we are iterating over all the employees we found and rendering a row for each
- Note that in velocity templates you can use properties, rather than java-style getters
    - E.g. getFirstName()



```
<table>
    <thead>
    <tr>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Email</th>
    </tr>
    </thead>
    <tbody>
        #foreach( $employee in $employees )
            <tr>
                <td>$employee.FirstName</td>
                <td>$employee.LastName</td>
                <td>$employee.Email</td>
            </tr>
        #end
    </tbody>
</table>
```

Model Classes

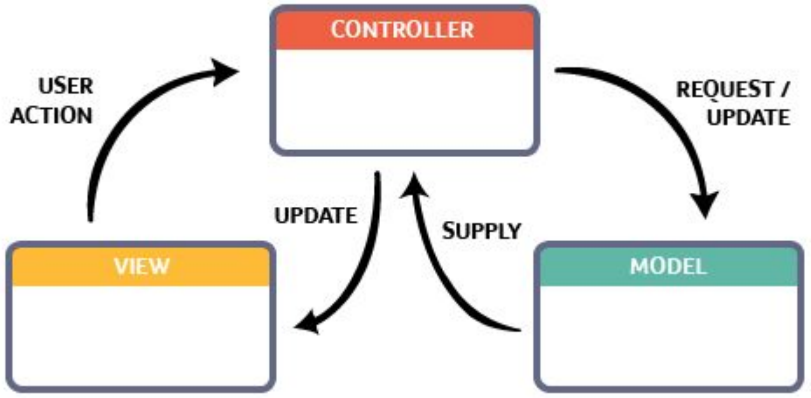
- A large part of the project is going to be writing *Model Objects*
- Model objects are objects that correspond to your database and that expose
  - Database fields
  - Logical operations
- Here we have the Employee model object



```
class Server {

    public static void main(String[] args) {
        get( path: "/", (req, resp) ->
                Web.renderTemplate( index: "templates/index.vm",
                ...args: "message", "SQL Is Awesome!",
                "employees", Employee.all( page: 1,  count: 10)));
    }

}
```

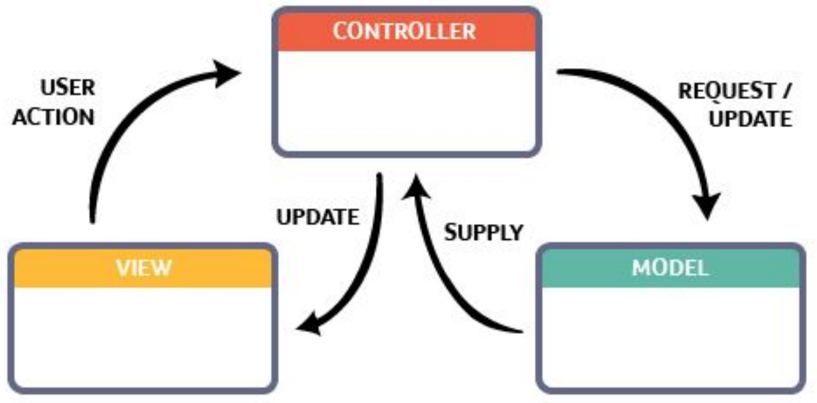
Model, View Controller

- You may have heard the term MVC: Model, View, Controller
- This is a common system model across many different domains, but it applies very well to web programming



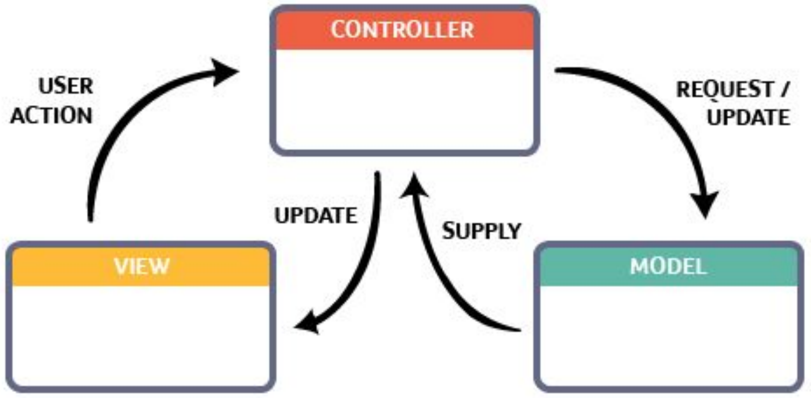
Controller

- Responsible for processing a user action
- Dispatches/converts that action into a request to the Model
- Relays the Models response to the View



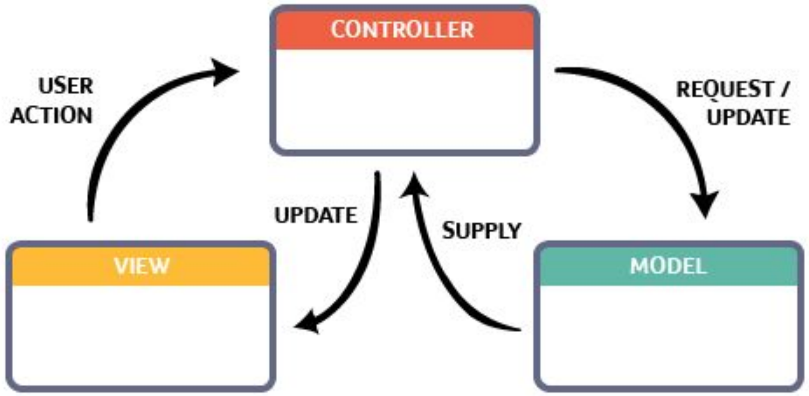
View

- Given a Model, creates an updated User Interface for the user to interact with
- NB: this could be an update *in place* (as with a thick client) or a complete refresh of the UI, as with web pages



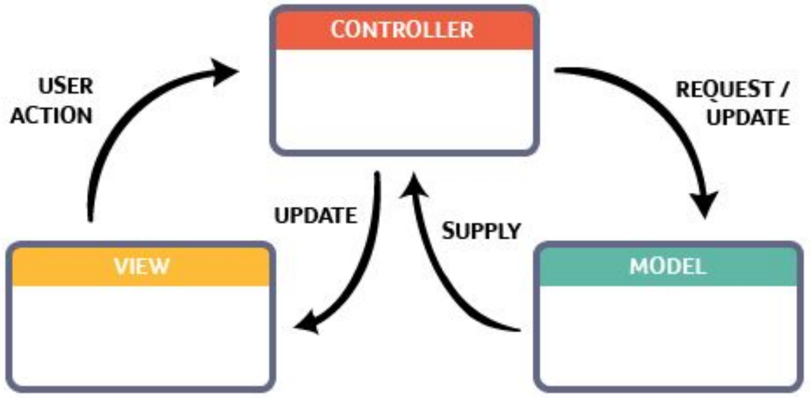
Model

- The Model, sometimes referred to as the *domain model* is the representation of the underlying domain
- In OO languages, typically represents both the data and the actions available on that data in the *domain*



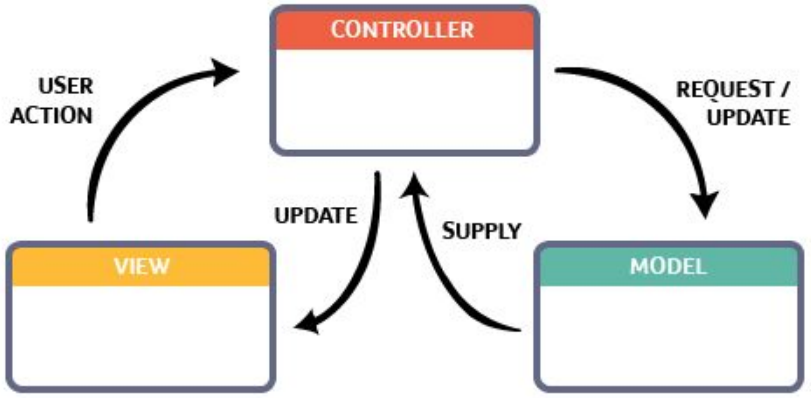
Model Classes

- In our case
- Model: Our java objects that we will create
- View: Velocity templates
- Controller: Controller Java files
  - Demo: Step Through Employee Request



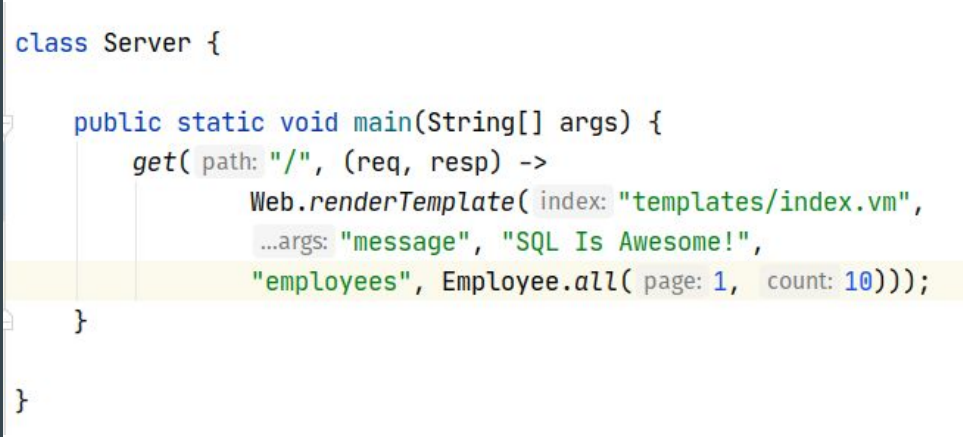
Model Classes

- Note that for our Model, we have a (roughly) 1-1 correspondence with the underlying scheme from chinook db
- In addition to *fields* that map to the database, we also have *methods* that allow retrieval and modification of that data



Model Classes

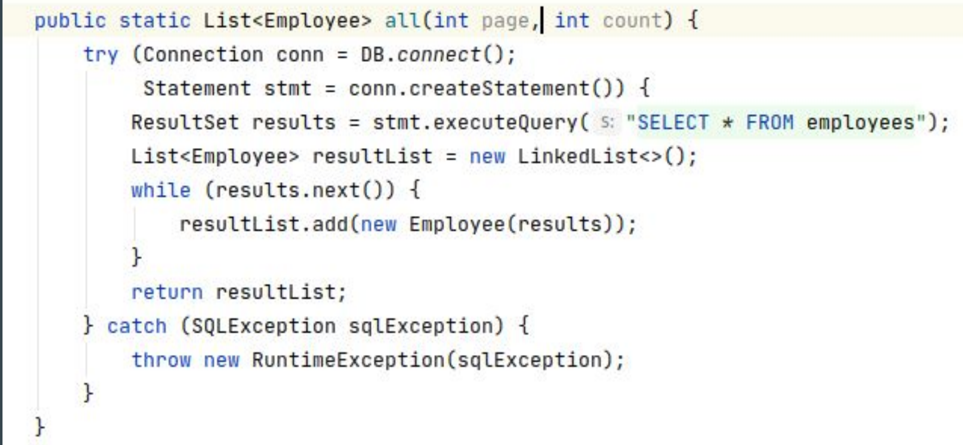
- The Model class will communicate with the database via JDBC (Java Database Connectivity)
- JDBC is a mature API with plenty of tools to work with



```
class Server {

    public static void main(String[] args) {
        get( path: "/", (req, resp) ->
                Web.renderTemplate( index: "templates/index.vm",
                 ...args: "message", "SQL Is Awesome!",
                "employees", Employee.all( page: 1,  count: 10)));
    }

}
```

Model Classes

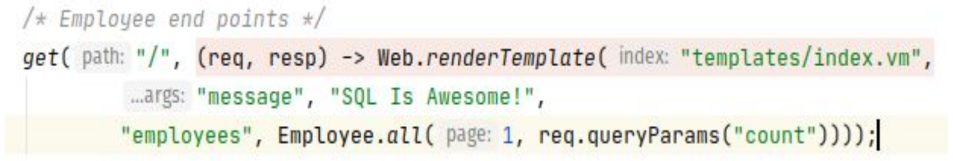
- The example *all()* static function demonstrates a basic JDBC call
    - We connect to the DB
    - Create a statement
    - Execute some SQL
    - Process the results from database rows into java Model objects
    - Return them as a List for display



```java
public static List<Employee> all(int page, int count) {
    try (Connection conn = DB.connect();
         Statement stmt = conn.createStatement()) {
        ResultSet results = stmt.executeQuery( s: "SELECT * FROM employees");
        List<Employee> resultList = new LinkedList<>();
        while (results.next()) {
            resultList.add(new Employee(results));
        }
        return resultList;
    } catch (SQLException sqlException) {
        throw new RuntimeException(sqlException);
    }
}
```

Model Classes

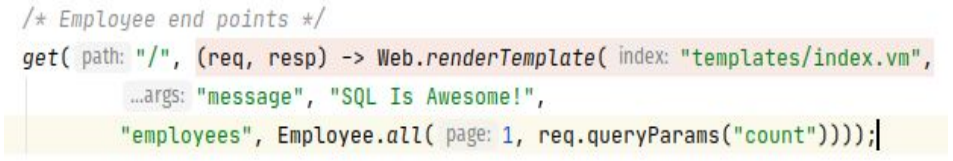
- Let’s implement that count feature together!
- First thing first, we need to get the count parameter from the URL
- We will pass it in as a *query parameter*
  http://localhost:4567/?count=3



```
/* Employee end points */
get( path: "/", (req, resp) -> Web.renderTemplate( index: "templates/index.vm",
        ...args: "message", "SQL Is Awesome!",
        "employees", Employee.all( page: 1, req.queryParams("count"))));
```

Model Classes

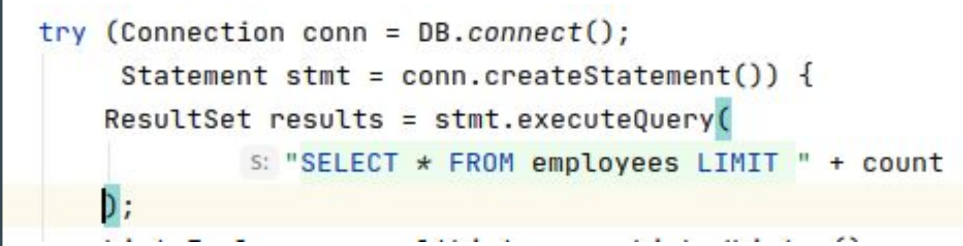
- We can use the *request.queryParams()* method to get the value passed in
- It’s a string, so let’s convert that parameter to a string...



```
/* Employee end points */
get( path: "/", (req, resp) -> Web.renderTemplate( index: "templates/index.vm",
         ...args: "message", "SQL Is Awesome!",
         "employees", Employee.all( page: 1, req.queryParams("count"))));
```

Model Classes

- And then we can update our query to use the LIMIT statement
- Fix a few compilation errors and restart our server...

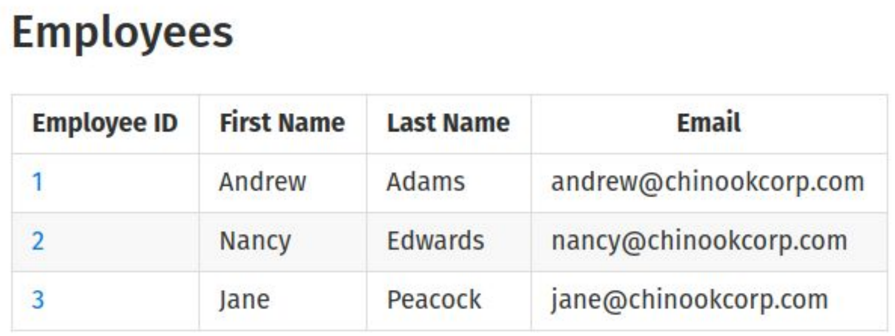


```
try (Connection conn = DB.connect();
     Statement stmt = conn.createStatement()) {
    ResultSet results = stmt.executeQuery(
            "SELECT * FROM employees LIMIT " + count
    );
```

Model Classes

- And presto! The limit works!
- But…
- What if I’m a tricky trickster, and create a URL like this:

  http://localhost:4567/?count=3;DELETE%20FROM%20EMPLOYEES



Employees

Employee ID	First Name	Last Name	Email
1	Andrew	Adams	andrew@chinookcorp.com
2	Nancy	Edwards	nancy@chinookcorp.com
3	Jane	Peacock	jane@chinookcorp.com

Model Classes

- Yikes

Model Classes

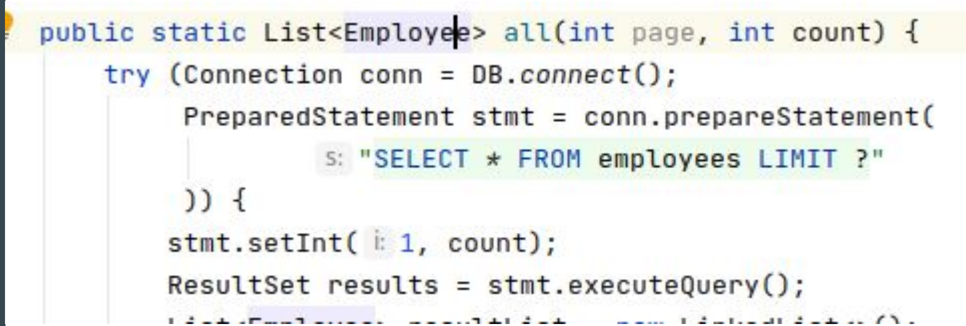
- It turns out that this will *not* delete all tables
- JDBC is smart enough to only execute one SQL statement
- If you want multiple statements, you must batch them
- But not all SQL APIs are as smart

Model Classes

- This is an example of a *SQL Injection Attack*
- String concatenation should never be mixed with user data
- Instead, we need to use a *PreparedStatement*
- This allows you to set values in a query safely

Model Classes

- You specify placeholders for the variables that will be set in the query
- You then call the appropriate set method, with an index, to set the value
- The index is 1 based...



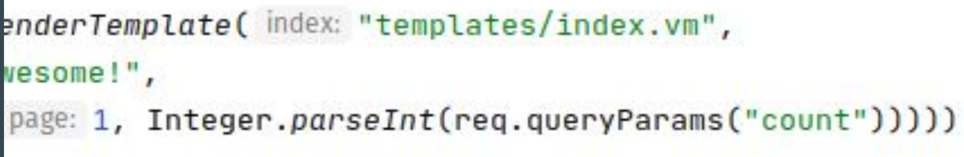
```
public static List<Employee> all(int page, int count) {
    try (Connection conn = DB.connect();
         PreparedStatement stmt = conn.prepareStatement(
                 s: "SELECT * FROM employees LIMIT ?"
         )) {
        stmt.setInt( i: 1, count);
        ResultSet results = stmt.executeQuery();
```

Model Classes

- You specify placeholders for the variables that will be set in the query
- You then call the appropriate set method, with an index, to set the value
- The index is 1 based...

Model Classes

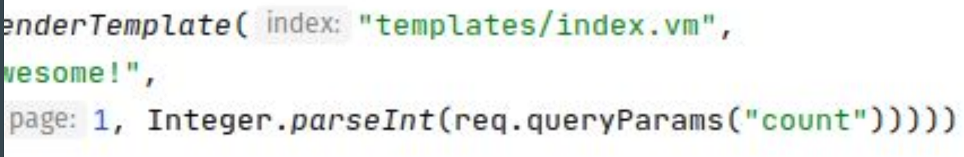
- Fine, we’ve reverted to an int and used a prepared statement, but now the Server isn’t compiling…
- We need to parse the string into a valid integer: *Integer.parseInt()*



```
enderTemplate( index: "templates/index.vm",
wesome!",
 page: 1, Integer.parseInt(req.queryParams("count"))))
```

Model Classes

- And we're done!
- We have a functioning mechanism for limiting the number of employees we show with a dynamic query, driven by a URL parameter
- Not bad!
- Using this, as well as some helpers, you can implement paging in the app



```
enderTemplate( index: "templates/index.vm",
vesome!",
 page: 1, Integer.parseInt(req.queryParams("count"))))
```

Web Programming Summary

- The Web is a wonderful collection of hypertext documents linking together friends and family across the world
    - and also 4chan
- We are going to be using Spark Java as our web server
- We will be using Velocity templates for our HTML templates
- We will be using JDBC to work with the SQLite Database
- String concatenation is bad
- PreparedStatements are safe

MONTANA
STATE UNIVERSITY