
MONTANA
STATE UNIVERSITY

Web CRUD

•••

Create, Read, Update, Delete

Last Lecture

- The last lecture was a whirlwind introduction to web programming and DB integration
  - Looked at requests
  - Looked at JDBC
  - Learned about SQL Injection

This Lecture

- This lecture we will look in depth at the patterns used for CRUD operations on the web
- Recall
  - C - Create
  - R - Read
  - U - Update
  - D - Delete

Note

- For this project, we are going to use a very simplified web framework, created by me
- Heavily influenced by ruby on rails
- Neither pretty, nor complete
- Optimized for simplicity & getting out of the way
- You will be mostly writing JDBC

URL Layout

- We will be using a standard URL layout
  - Based on long-standing patterns in web apps
- This is sometimes referred to as REST-ful URLs
- That is not correct, but be familiar with the term

```
C
/employees/new - Create a new employee

R
/employees - List employees
/employees/1 - Show employee 1

U
/employees/1/edit - Edit employee 1

D
/employees/1/delete - Delete employee 1
```

URL Layout

- GET & POST are two *HTTP methods*
- Issuing a GET or a POST to a URL is a *different* sort of request
- *GET* - Get the resource at the URL
- *POST* - Update the resource at the URL

```
C
/employees/new - Create a new employee

R
/employees - List employees
/employees/1 - Show employee 1

U
/employees/1/edit - Edit employee 1

D
/employees/1/delete - Delete employee 1
```

URL Layout

- Consider the URL /employees/1/edit
- If we issue a GET to that URL, we are asking for an edit UI for that resource
- If we issue a POST to that URL, we are asking the server to update the resource

```
C
/employees/new - Create a new employee

R
/employees - List employees
/employees/1 - Show employee 1

U
/employees/1/edit - Edit employee 1

D
/employees/1/delete - Delete employee 1
```

URL Layout

- In HTML, a link issues a GET
- A form can issue a GET or a POST
- We use forms for the new employee UI and edit UI that POST back to their URL
    - What about delete? We'll talk about that in a bit...

```
C
/employees/new - Create a new employee

R
/employees - List employees
/employees/1 - Show employee 1

U
/employees/1/edit - Edit employee 1

D
/employees/1/delete - Delete employee 1
```

URL Layout

- It turns out that there are many other HTTP methods:
  - PUT - replace the resource
  - HEAD - HTTP headers only
  - DELETE - delete the resource
  - PATCH - partially update the resource
  - OPTIONS - get communication options for the resource

```
C
/employees/new - Create a new employee

R
/employees - List employees
/employees/1 - Show employee 1

U
/employees/1/edit - Edit employee 1

D
/employees/1/delete - Delete employee 1
```

URL Layout

- Unfortunately, HTML as it currently exists makes it very difficult to use anything other than GET or POST
- For deletes, I would prefer to issue this request:

  DELETE /employees/1

```
C
/employees/new - Create a new employee

R
/employees - List employees
/employees/1 - Show employee 1

U
/employees/1/edit - Edit employee 1

D
/employees/1/delete - Delete employee 1
```

URL Layout

- Let me tell you about a little something I like to call… htmx
- A javascript library I wrote to fix HTML
  - Gives you access to things like DELETE!
- We will talk more about this when we discuss AJAX

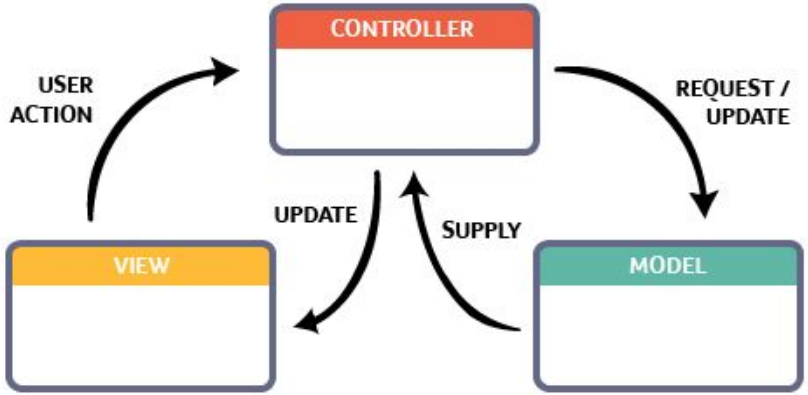
URL Layout

- So this is the URL pattern we are going to use for all the entities in our system
- /tracks
  /artists
  etc...

```
C
/employees/new - Create a new employee

R
/employees - List employees
/employees/1 - Show employee 1

U
/employees/1/edit - Edit employee 1

D
/employees/1/delete - Delete employee 1
```

Controller Code - Create

- Recall the Model View Controller concept
  - Model - The model classes, know how to work with the database
  - View - The velocity templates that render HTML
  - Controller - The code in the Server class



Controller Code - Create

- There are two aspects of creating an entity
  - Show the create UI
  - Do the creation of the element
- Correspondingly, we have two routes
  - One GET route
    /employee/new
  - One POST route
    /employee/new

```
/* CREATE */
get( path: "/employees/new", (req, resp) -> {
    Employee employee = new Employee();
    return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", employee);
});

post( path: "/employees/new", (req, resp) -> {
    Employee emp = new Employee();
    Web.putValuesInto(emp,  ...properties: "FirstName", "LastName");
    if (emp.create()) {
        Web.message( s: "Created An Employee!");
        resp.redirect( location: "/employees/" + emp.getEmployeeId());
        return "";
    } else {
        Web.message( s: "Could Not Create An Employee!");
        return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", emp);
    }
});
```

Create - GET form

- The GET is pretty simple
  - Create a new employee object
  - Render the employees/new.vm template
- The new template has a form in it that POST back to the same URL
- The form body is extracted to a form.vm file so it can be shared with the edit functionality

```
/* CREATE */
get( path: "/employees/new", (req, resp) -> {
    Employee employee = new Employee();
    return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", employee);
});

post( path: "/employees/new", (req, resp) -> {
    Employee emp = new Employee();
    Web.putValuesInto(emp,  ...properties: "FirstName", "LastName");
    if (emp.create()) {
        Web.message( s: "Created An Employee!");
        resp.redirect( location: "/employees/" + emp.getEmployeeId());
        return "";
    } else {
        Web.message( s: "Could Not Create An Employee!");
        return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", emp);
    }
});
```

Create - POST

- The POST logic is much more complex
- Note that the new template has a form in it that POSTs back to the same URL
- This URL is *overloaded* with a GET and a POST operation
  - It is a *resource* that can be *manipulated* via different actions

```
/* CREATE */
get( path: "/employees/new", (req, resp) -> {
    Employee employee = new Employee();
    return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", employee);
});

post( path: "/employees/new", (req, resp) -> {
    Employee emp = new Employee();
    Web.putValuesInto(emp,  ...properties: "FirstName", "LastName");
    if (emp.create()) {
        Web.message( s: "Created An Employee!");
        resp.redirect( location: "/employees/" + emp.getEmployeeId());
        return "";
    } else {
        Web.message( s: "Could Not Create An Employee!");
        return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", emp);
    }
});
```

Create - POST

- Again, the form body is extracted to a form.vm file so it can be shared with the edit functionality
- On the POST, we put the values from the request into a new employee object and attempt to create it

```
/* CREATE */
get( path: "/employees/new", (req, resp) -> {
    Employee employee = new Employee();
    return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", employee);
});

post( path: "/employees/new", (req, resp) -> {
    Employee emp = new Employee();
    Web.putValuesInto(emp,  ...properties: "FirstName", "LastName");
    if (emp.create()) {
        Web.message( s: "Created An Employee!");
        resp.redirect( location: "/employees/" + emp.getEmployeeId());
        return "";
    } else {
        Web.message( s: "Could Not Create An Employee!");
        return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", emp);
    }
});
```

Create - POST

- *create()* will *validate()* the object and, if it is valid, create it, otherwise return false
- On a successful create, we *redirect* to the URL that shows the newly created object
- On failure, we re-render the create UI and display the errors

```
/* CREATE */
get( path: "/employees/new", (req, resp) -> {
    Employee employee = new Employee();
    return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", employee);
});

post( path: "/employees/new", (req, resp) -> {
    Employee emp = new Employee();
    Web.putValuesInto(emp,  ...properties: "FirstName", "LastName");
    if (emp.create()) {
        Web.message( s: "Created An Employee!");
        resp.redirect( location: "/employees/" + emp.getEmployeeId());
        return "";
    } else {
        Web.message( s: "Could Not Create An Employee!");
        return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", emp);
    }
});
```

Create - POST

- *Whew!*
- Seems like a lot, but it's a simple enough pattern when you get used to it
- Update uses a very similar pattern

```
/* CREATE */
get( path: "/employees/new", (req, resp) -> {
    Employee employee = new Employee();
    return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", employee);
});

post( path: "/employees/new", (req, resp) -> {
    Employee emp = new Employee();
    Web.putValuesInto(emp,  ...properties: "FirstName", "LastName");
    if (emp.create()) {
        Web.message( s: "Created An Employee!");
        resp.redirect( location: "/employees/" + emp.getEmployeeId());
        return "";
    } else {
        Web.message( s: "Could Not Create An Employee!");
        return Web.renderTemplate( index: "templates/employees/new.vm",  ...args: "employee", emp);
    }
});
```

Read

- Reads are simple compared to Create:
  - Just GETs
  - No URL reuse
- /employees renders a list of employees
- /employees/:id renders a specific employee

```
/* READ */
get( path: "/employees", (req, resp) -> {
    List<Employee> employees = Employee.all( page: 1,  count: 10);
    return Web.renderTemplate( index: "templates/employees/index.vm",
            ...args: "employees", employees);
});

get( path: "/employees/:id", (req, resp) -> {
    Employee employee = Employee.find(Integer.parseInt(req.params(":id")));
    return Web.renderTemplate( index: "templates/employees/show.vm",
            ...args: "employee", employee);
});
```

Update

- Almost identical pattern to the CREATE logic
- Note that the URLs are now /employee/:id/edit
- GET to display the edit form
- POST to update the object in the database
- Once again, we are treating the employee as a *resource*

```
/* UPDATE */
get( path: "/employees/:id/edit", (req, resp) -> {
    Employee employee = Employee.find(Integer.parseInt(req.params(":id")));
    return Web.renderTemplate( index: "templates/employees/edit.vm",
            ...args: "employee", employee);
});

post( path: "/employees/:id", (req, resp) -> {
    Employee emp = Employee.find(Integer.parseInt(req.params(":id")));
    Web.putValuesInto(emp, ...properties: "FirstName", "LastName");
    if (emp.update()) {
        Web.message( s: "Updated Employee!");
        resp.redirect( location: "/employees/" + emp.getEmployeeId());
        return "";
    } else {
        Web.message( s: "Could Not Update Employee!");
        return Web.renderTemplate( index: "templates/employees/edit.vm",
                ...args: "employee", emp);
    }
});
```

DELETE

- Delete logic is also simple:
  - Find the given object
  - Delete it from the DB
  - Redirect to the list of objects
- This implementation isn't ideal
  - We are ysing a GET to modify a resource
  - No confirmation?
  - In production, we would confirm this action and perhaps use htmx to issue a DELETE

```
/* DELETE */
get( path: "/employees/:id/delete", (req, resp) -> {
    Employee employee = Employee.find(Integer.parseInt(req.params(":id")));
    employee.delete();
    Web.message( s: "Deleted Employee " + employee.getEmail());
    resp.redirect( location: "/employees");
    return "";
});
```

CRUD Summary

- Using a standard web URL pattern, we can implement the basic CRUD functionality for tables
- Create and Update look similar, require the most logic
- Delete is easy, but probably would not be implemented this way in production
    - Htmx is rad
- Read is pretty easy
    - We will add search next time!

Project Note

- Again I want to emphasize: this is NOT a web app development class
- I want you to be writing mostly Model/JDBC code
- We will be using simplified HTML and template logic
    - E.g. Tables for layout
- I do want you to understand the general web patterns, however
    - Be comfortable with web terminology
    - Understand the URL patterns
    - Know what a request and response are
    - Maybe learn a bit of htmx!


MONTANA
STATE UNIVERSITY