# Project 4: Java on Rails

COMP 121, Spring 2021

Test case due: Mon, Apr 12 @ 11:59pm

Main project due: Tue, Apr 20 @ 11:59pm

## Introduction

In this project you will implement a web server that uses a model-view-controller architecture. We'll call the system *jrails*, because it uses ideas from Ruby on Rails. However, because Java is, well, Java, the framework will be noticeably cruftier than Rails. On the other hand, we'll only implement a tiny fraction of what goes into a real web server framework.

The code for this project is structured a bit differently than for previous projects. The MVC framework source code, which is what you'll be editing, is in the `jrails/` directory. The code in that directory will be compiled to `jrails.jar`, which can then be added to the classpath when compiling a web app that uses the framework.

To make this process easier, we've given you a build setup for gradle, which is a popular and powerful build tool for Java. It's a bit complex to set up the build, but you shouldn't need to modify the build process at all. Here's how to use the build process:

- `./gradlew tasks` - list all the build tasks
- `./gradlew classes` - build jrails and the app
- `./gradlew run` - run the app
- `./gradlew testClasses` - build tests
- `./gradlew test` - run tests

To help you more easily understand the structure of jrails, we've included an example app spread across four files in the `books` app directory: `Book.java`, `BookController.java`, `BookView.java`, and `Main.java`. (These files are in `books/src/main/java/books`.) Once you implement jrails, this particular app will let users keep track of a list of books. When you run

```
./gradlew run
```

you will eventually see something like this on the console:

```
> Task :books:run
Starting server...point your web browser to http://localhost:8000
<===========--> 87% EXECUTING [7s]
> :books:run
```

(Note that the text prints slightly out of order.) At this point, the web server has started on port 8000 on the local machine. You can connect to it by firing up a web browser and going to

```
http://localhost:8000/test
```

This should display a simple web page in your browser, and the console should show that the request was received. If you want to play around a bit more with the server, if you go to any other URL that's not routed (see below), the web server will dump some information about the request onto the console. For example, I noticed that when I go to a web page on this server, my web browser tries to get `/favicon.ico` in addition to the page I requested. It's fine for the server to just drop such requests on the floor.

## Models

Like most MVC web app frameworks, jrails includes *models* that represent the database. More specifically, a model is any class that subclasses `jrails.Model`, which uses reflection to provide primitive database functionality. For example, here is the model from the book app:

```
import jrails.*;
public class Book extends Model {
    @Column public String title;
    @Column public String author;
    @Column public int num_copies;
}
```

Because `Book extends Model`, conceptually there is a database table corresponding to the `Book` class. That table has three columns, `title`, `author`, and `num_copies`, with the corresponding types `String`, `String`, and `int`, respectively.

Here is some code that uses the above model, with inline comments to explain what's happening:

```
Book b = new Book();
b.title = "Programming Languages: Build, Prove, and Compare";
b.author = "Norman Ramsey";
b.num_copies = 999;
// The book b exists in memory but isn't saved to the db
b.save(); // now the book is in the db
b.num_copies = 42; // the book in the db still has 999 copies
b.save(); // now the book in the db has 42 copies
Book b2 = new Book();
b2.title = "Programming Languages: Build, Prove, and Compare";
b2.author = "Norman Ramsey";
b2.num_copies = 999; // hm, same as other book
b2.save(); // a second record is added to the database
assert(b.id() != b2.id()); // every row has a globally unique id (int) column, so we can tell them apart
Book b3 = Model.find(Book.class, 3); // finds the book with id 3 in the db, if any
List<Book> bs = Model.all(Book.class); // returns all books in the db
b.destroy(); // remove book b from db
```

Your job for this part of the project is to implement the `Model` class in jrails to achieve this functionality. Here are the details:

- In general, **jrails should throw an exception if the app code has an error.** (Warning: as a general principle, it's okay for web apps to show stack traces locally, but they should not send them in response to a request that crashes the server, since that could reveal information to an adversary.)
- Below, we'll write "model" to refer to a subclass of `Model`.
- A model class has zero or more public fields annotated with `@Column`. The only possible types for `@Column` fields are `String`, `int`, and `boolean`. It is an error for any other type to be used with `@Column`.
- When `save` is called on a model, you should write the contents of the model to a file on disk. The exact name and format of this file is up to you. We won't look at it directly. I'd suggest a text file in comma-separated value format, where each line represents a database row. Be careful when writing strings to the file in case they themselves have commas. Note that a `String @Column` could be `null`, and `null` should be serialized differently than an empty string. It's up to you whether to have one global db file or several different ones for individual tables. You probably need some logic to create the db file if it does not already exist.
- Initially, a model that has been created but not saved has an id of 0. Just before a model is written to disk, is gets a globally unique id assigned to it, which can subsequently be accessed by calling the `id()` method of the object and is also written to disk. You'll need to keep track of ids on disk somewhere. If you prefer to have unique ids per table, that's okay too.
- When `save`ing a model with a zero id field, you should *add* a new row to the db file on disk. When `save`ing a model with a non-zero id field, you should *replace* the previous record in the database. It is an error to try to `save` a model with a non-zero id that is not already in the database. It will likely be somewhat annoying to implement replacement, because you'll need to change text in the middle of a file, which is always a bit awkward. You can safely assume that the entire db will fit in memory, though, so you can always read the whole file into some in-memory format and then write it out.
- The `int id()` method returns the id of the model.
- The `<T> T find(Class<T> c, int id)` method looks through the db for a row of the given id for the model `c`. If one exists, it *materializes* the row by creating a fresh instance of the model and initializing its fields according to the database entry. You can assume the model has only the no-argument constructor. If there is no db entry with the given id, then `find` returns `null`. The fancy type signature here means that `find` returns an instance of whatever class is passed as the first argument.
- The `<T> List<T> all(Class<T> c)` class method returns a `List` (possibly empty) of all the (materialized) db rows for the model. Similarly to `find`, the fancy type means `all` returns a list of instances of whatever class is passed as the first argument.
- Calling the `void destroy()` method removes the receiver from the db. If the receiver has not been saved to the db previously, this method raises an exception.
- Calling `Model.reset()` should remove all rows from the database. We will use this method to make grading your project easier.

## Views

The *views* for jrails are HTML pages that are sent back to the client web browser. Ruby on Rails uses HTML with embedded Ruby code, but developing such a system is a bit too complex for this project. Instead, we will create HTML by invoking methods in Java. More specifically, here is an example view from the book project:

```
    public static Html show(Book b) {
        return p(strong(t("Title:")).t(b.title)).
                p(strong(t("Author:")).t(b.author)).
                p(strong(t("Copies:")).t(b.num_copies)).
                t(link_to("Edit", "/edit?id=" + b.id())).t(" | ").
                t(link_to("Back", "/"));
    }
```

Here the `show` method takes a `Book` and returns an `Html`, which is a data structure representation of HTML. The `View` superclass provides a variety of methods for constructing HTML, e.g., the `p(...)` method constructs HTML with paragraph tags `<p>...</p>`, where the child `...` is more HTML that is created by calling some other methods inherited from `View`. Moreover, an `Html` object has as instance methods all the same tag methods as `View`, which are used to sequence HTML. For example, calling `p(a).p(b)` returns in `<p>a</p><p>b</p>` for some `a` and `b`.

Your next task is to implement `View` and `Html` such that calling `toString` on an `Html` returns a string containing the corresponding HTML. **These methods should not modify the current HTML object, i.e., they should be purely functional.** Here are the tags you need to support:

- Standard HTML tags `br`, `p`, `div`, `strong`, `h1`, `tr`, `th`, `td`, `table`, `thead`, `tbody`, `textarea`. For `textarea`, don't forget to include the `name` argument as an attribute (see the example form in `JServer.java`).
- There should be no spaces between the <, >, and output HTML tags. The tags should all be in lowercase. Self-closing tags, specifically `br`, should be rendered with the slash, i.e., `<br/>`. There should only be one space before any attributes. The value of any attributes should be quoted with double quotes.
- `t(Object o)` should call `o.toString()` and convert the result into an `Html`.
- `empty()` should return HTML corresponding to an empty string. This method is not part of `Html`.
- `seq(Html h)` should sequence the HTML `h` after `this`. This method is not part of `View`.
- `link_to(String text, String url)` should return HTML corresponding to a link to URL `url` with text `text`, e.g., `link_to("Show", "/show?id=42")` should produce `<a href="/show?id=42">Show</a>`.
- `form(String action, Html child)` should return HTML corresponding to a form that sends a POST request in UTF-8 to URL `action` and contains `child` as its body. For example, `form("/create", html)` should produce `<form action="/create" accept-charset="UTF-8" method="post">HTML</form>`, where `HTML` is the contents of `child`.
- `submit(String value)` should return an HTML submit button with the given value, e.g., `submit("Save")` should produce `<input type="submit" value="Save"/>`.

For example, the following code:

```
Book b = new Book();
b.title = "Programming Languages: Build, Prove, and Compare";
b.author = "Norman Ramsey";
b.num_copies = 999;
String s = BookView.show(b);
```

results in the following HTML (spaces and line breaks added for clarity):

```
<p><strong>Title:</strong>Programming Languages: Build, Prove, and Compare</p>
<p><strong>Author:</strong>Norman Ramsey</p>
<p><strong>Copies:</strong>999</p>
<a href="/edit">Edit</a> | <a href="/">Back</a>
```

## Controllers

When a web request comes in, it's eventually passed to a *controller*, which handles the request and returns an HTML page in response. For example, here is a controller from our example app:

```
    public static Html show(Map<String, String> params) {
        int id = Integer.parseInt(params.get("id"));
        Book b = (Book) Book.find(id);
        return BookView.show(b);
    }
```

The input to *all* controller methods is a hash of parameter names to values, both of which are strings, and every controller method is public and `static` and returns an `Html`. In this particular case, the router (discussed next), will map the URL `http://localhost:8000/show?id=42` to a call to `show` where `id` is mapped to `"42"`. Then the body of `show` does some computation, in this case retrieving a book from the database, and returns the corresponding HTML page by calling one of the view methods.

If you look through the controller methods for the example app, you'll see that they support a basic CRUD interface, i.e., create, read, update, and delete.

There's actually no code you need to implement to support controllers. There is a superclass, `Controller`, but for purposes of this project it can be empty. (For a more full-featured web framework, we'd probably want to add some functionality to it.)

## The Router

As we just saw, different HTTP requests are handled by different controllers. Among other things, each HTTP request has a verb (e.g., `GET`, `POST`) and a *path* (e.g., `/show` from the URL `localhost:8000/show?id=42`). The job of the *router* is to map such requests to controller methods.

The router has to be configured on a per-app basis, which is done with a series of calls to `addRoute`. For example, here is the routing for the example app:

```
        JRouter r = new JRouter();
        r.addRoute("GET", "/", BookController.class, "index");
        r.addRoute("GET", "/show", BookController.class, "show");
        r.addRoute("GET", "/new", BookController.class, "new_book");
        r.addRoute("GET", "/edit", BookController.class, "edit");
        r.addRoute("POST", "/create", BookController.class, "create");
        r.addRoute("POST", "/update", BookController.class, "update");
        r.addRoute("GET", "/destroy", BookController.class, "destroy");
```

For example, the second call to `addRoute` tells the router to map a `GET` request for `show` to the `BookController`'s `show` method.

You need to implement the following behavior for `JRouter`:

- A `JRouter` should maintain a list of routes in internal state (so you'll need to add at least one field).
- Calling `void addRoute(String verb, String path, Class clazz, String method)` should add a route from HTTP `verb` for `path` to the `clazz` class's `method`.
- Calling `String getRoute(String verb, String path)` should return a string of the form `"clazz#method"` if such a route exists, or `null` otherwise. For example, after the sequence of `addRoute` calls above, calling `r.getRoute("GET", "/show")` should return `"BookController#show"`.
- Calling `Html route(String verb, String path, Map<String, String> params)` should look up the controller method corresponding to `verb` and `path`, call it with the given `params`, and return the result. If no such route exists, this method should raise an `UnsupportedOperationException`.

## The HTTP Server

Finally, jrails include a class `JServer` with a method `void start(JRouter r)` that starts up an HTTP server on port 8000, listens for requests, and routes any requests received through `r`, sending the result back to the web browser. We've written this class for you, and you shouldn't need to modify it.

Then each application sets up its routes and calls `start` to launch the web server. We've put all the necessary code for the example app in a class called `Main`, which you can run as described above.

## Errors

In general, when web servers encounter errors, they often send some nice web page back to the web browser indicating something went wrong. But for this project, we're not going to do that. Instead, your web server is just going to crash with an exception. That's okay for this project, though it wouldn't be great for a real-world system.

## How to Test Your Code

Although on the surface this project seems hard to test—e.g., you might think you have to write tests that connect over the network to a web server—in fact all the model/view/router code is designed to be tested locally. You'll notice that the only code involving networking is in `JServer`, which you didn't write and therefore you don't have to test.

Thus, as part of this project, you must write at least one test case and post it on Campuswire to share with the class. (Warning: We don't guarantee that posted test cases are correct!) Since there are several different parts of the project, we'll use the following rules to split up the tests:

| Last digit of Tufts ID | Class for your test |
|---|---|
| 0-3 | `Model.java` |
| 4-7 | `View.java` and `Html.java` |
| 8-9 | `Router.java` |

To use the testing framework for your project, put your tests in the appropriate file in `jrails/src/test/java/jrails/`. We've gotten you started with a few basic tests already.

## What to turn in

Put all your code in the code skeleton we have supplied, and upload your solution using Gradescope.