Restlets

COMP 30220: Distributed Systems

Lecturer: Rem Collier

Email: rem.collier@ucd.ie

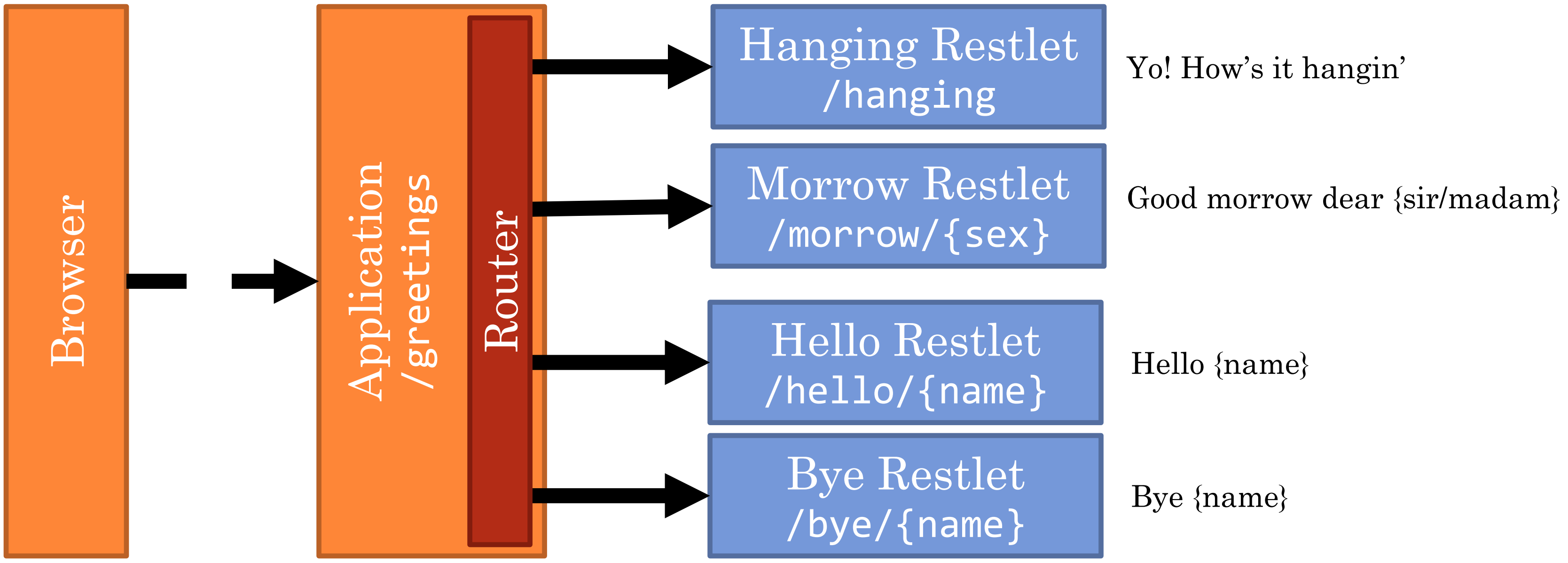
What are Restlets?

- An open source Java framework for building RESTful Services:
  - Source code on GitHub: https://github.com/restlet
  - Documentation & Download: http://www.restlet.com
- Restlets are seen as an easy and intuitive way to build RESTful web services.
  - Each Restlet maps to a single resource, and it is responsible for handling all operations applied to that resource.
  - An advanced feature of Restlets is the ability to map resources to annotated interfaces.
- The Restlet API comes with a built in web server.
  - It is not recommended for "production" use, but it will do the job for us…
  - For production use, Restlets can be combined with servlet containers, such as Tomcat or Jetty.

Creating a Restlet Service

- There are a number of ways to use the Restlet API, here, we will adopt the following approach:
  - REST Services are organised into **Application**s.
  - An application is a set of related resources (**Restlets**) possibly linked together through a **Router.**
  - Each application can be associated with a different base segment of the URL: *http://<myhost>:<myport>/<application>/<resource-path>*
  - The Router maps each resource path to a Restlet that represents a single resource.
  - The resource path can be parameterised to allow a single URL to refer to multiple resource instances: *http://<host>:<port>/<app>/<res-path>/<res-id(s)>*

Greetings Example: Architecture



Greeting Example: **HangingRestlet**

- This Restlet returns the string “Yo! How’s it hangin’” if a GET operation is used, and “Forbidden (403) Error” otherwise.

```java
public class HangingRestlet extends Restlet {
    @Override
    public void handle(Request request, Response response) {
        if (request.getMethod().equals(Method.GET)) {
            response.setEntity(“Yo! How's it hangin’”, MediaType.TEXT_PLAIN);
            response.setStatus(Status.SUCCESS_OK);
        } else {
            response.setStatus(Status.CLIENT_ERROR_FORBIDDEN);
        }
    }
}
```

Greetings Example: GreetingsApplication

- This class combines multiple restlets into a single application that will be deployed on a single node.

```
public class GreetingsApplication extends Application {
    public Restlet createInboundRoot() {
        Router router = new Router(getContext());
        router.attach("/hanging", new HangingRestlet());
        return router;
    }
}
```

- The core component is a router which is responsible for passing HTTP requests to the relevant Restlet.
  - Routes are specified by attaching restlets to the router.

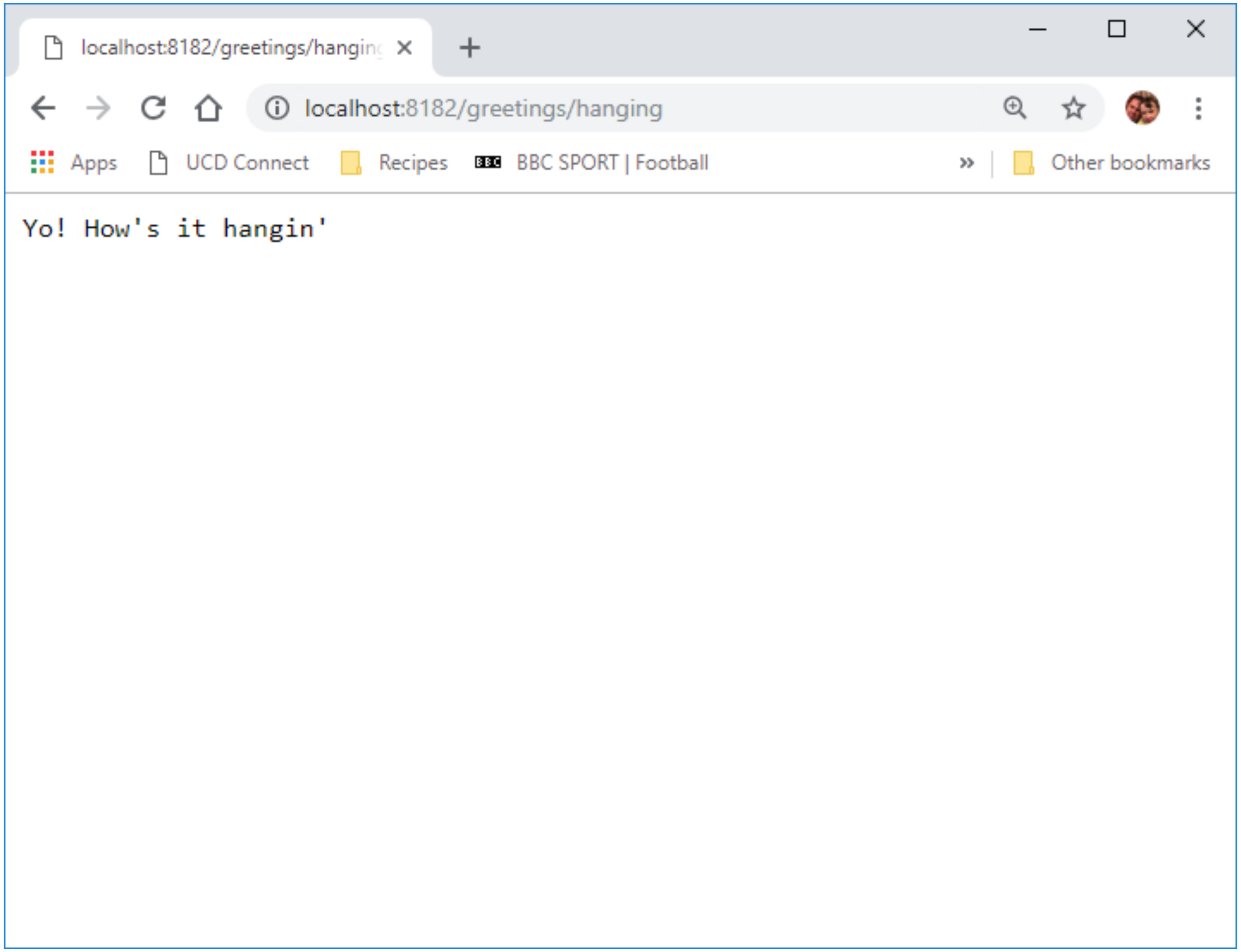
Deploying the Greetings Example

- To deploy a Restlet application, you use the Component class of the Restlet API.

```
public class GreetingsServer {
    public static void main(String[] args) throws Exception {
        Component component = new Component();
        component.getServers().add(Protocol.HTTP, 8182);
        component.getDefaultHost().
            attach(“/greetings", new GreetingsApplication());
        component.start();
    }
}
```

- This class can be used to create one or more endpoints that can be used to access one or more REST applications.
  - If you are hosting only one application, you can omit the “application” segment of the url by using an empty string when attaching the application (e.g. `http://localhost:8182/hanging`)

Greetings Client



localhost:8182/greetings/hangin
localhost:8182/greetings/hanging
Apps
UCD Connect
Recipes
BBC SPORT | Football
Other bookmarks

```
Yo! How's it hangin'
```

Greeting Example: **HelloRestlet**

- This Restlet returns the string “Hello {name}” if a GET operation is used, and “Forbidden (403) Error” otherwise.
  - Note: {name} is a parameter of the URL: /hello/{name}

```
public class HelloRestlet extends Restlet {
    @Override
    public void handle(Request request, Response response) {
        if (request.getMethod().equals(Method.GET)) {
            String name = (String) request.getAttributes().get(“name”);
            response.setEntity(“Hello” + name, MediaType.TEXT_PLAIN);
            response.setStatus(Status.SUCCESS_OK);
        } else {
            response.setStatus(Status.CLIENT_ERROR_FORBIDDEN);
        }
    }
}
```

Revised: GreetingsApplication

- We now need to attach the second restlet to the application:

```
public class GreetingsApplication extends Application {
    public Restlet createInboundRoot() {
        Router router = new Router(getContext());
        router.attach("/hanging", new HangingRestlet());
        router.attach("/hello/{name}", new HelloRestlet());
        return router;
    }
}
```

- The core component is a router which is responsible for passing HTTP requests to the relevant Restlet.
  - Routes are specified by attaching restlets to the router.

WORKING WITH DATA

Working with More Complex Data

- So far, we have worked with simple textual data, but REST services typically handle more complex data types.
  - For example, a student record could be represented by the JSON object below:

```
{
   “student-number” : “96123456”,
   “firstname” : “Rem”,
   “surname” : “Collier”,
   “gender” : “M”,
   “dateofbirth” : “04-07-1973”,
   “home-address” : “36 Elm Street”
}
```

- This is fine as a representation format, but when it is sent, it is as text, not JSON…

Working with More Complex Data

- What we would like is to be able to work with an equivalent structure (such as a **data class**) in Java:

```
public class StudentRecord {
    public String student_number;
    public String firstname;
    public String surname;
    public String gender;
    public String dateofbirth;
    public String home_address;
}
```

- But: how to transform the data between JSON and the Java object?

The Manual Option?

- We can always manually craft a solution:

```
public String toJson() {
    return "{\"student_number\":\"" + student_number +
        "\", \"firstname\":\"" + firstname +
        "\", \"surname\":\"" + surname +
        "\", \"sex\":\"" + sex +
        "\", \"dateofbirth\":\"" + dateofbirth +
        "\", \"home_address\":\"" + home_address + "\"}";
}
```

- What about the JSON -> Java translation?
    - We have to cater for different data types – strings, numbers, Boolean values, null, arrays, and objects…
    - We have to cater for encoding schemes – escaped quotes etc.
    - We have to manually construct a parser to tokenize the string and map the contents to the object equivalents!!!!

The Manual Option?

- Wait a minute – there is something we are missing:
  - The Java representation includes types and names of fields.
  - The JSON representation includes names but not types.

- Java has this beautiful library called the Reflection API:
  - Programs can introspectively analyse their own structure.
  - What methods a class has, how they are annotated, the parameter and return types, what fields exist and their types, …
  - Why don’t we use the strongly-typed Java representation as the model, build a general JSON parser and then parse the JSON using the Java model as the schema?
  - Oh! Somebody already did it in 2008 – the Google Gson library: https://github.com/google/gson

Using Gson To Create JSON

- Step 1: Create an instance of the Gson parser.
  - One instance is enough per program:

```
Gson gson = new Gson();
```

- Step 2: Convert an object into Json:

```
String json_data = gson.toJson(record);
```

- Step 3: Send the data

```
out.println(json_data);
gson.toJson(json, StudentRecord.class);
```

Using Gson to Convert JSON

- Step 1: Create an instance of the Gson parser.
  - One instance is enough per program:

```
Gson gson = new Gson();
```

- Step 2: Convert a Json object into a Java object:

```
StudentRecord record =
    (StudentRecord) gson.toJson(json,StudentRecord.class);
```

- Step 3: Do something…

```
studentDB.store(record);
```

Sending A Json Response

- Lets consider a simple Time API:
  - This API contains a single resource "/time" which returns a JSON representation of the current time.

- To implement this, we can use the `LocalDateTime` class to get the current time:
  - `LocalDateTime now = LocalDateTime.now();`

- To send the resultant object back (in JSON format), we must:
  - Use Gson to convert the object to JSON:

    ```
    gson.toJson(now)
    ```
  - Modify the content type of the response to indicate that JSON is being returned:

    ```
    response.setEntity(the_json, MediaType.APPLICATION_JSON);
    ```

- If we want to more control over the format of the JSON response, we must create a data class for it…

Sending A Json Response

```
package time;

public class TimeRestlet extends Restlet {
  private static Gson gson = new Gson();

  @Override
  public void handle(Request request, Response response) {
    if (request.getMethod().equals(Method.GET)) {
      LocalDateTime now = LocalDateTime.now();
      response.setEntity(gson.toJson(now), MediaType.APPLICATION_JSON);
      response.setStatus(Status.SUCCESS_OK);
    } else {
      response.setStatus(Status.CLIENT_ERROR_FORBIDDEN);
    }
  }
}
```

Receiving JSON in a Request

- When you send JSON, it must be part of the body of the HTTP Request.
  - Normally, this means it should be part of a POST or PUT request.

- Lets play with a calculator example.
  - We can model a calculator resource as a calculation resource.
  - We can create a calculation by sending a JSON representation of the calculation to the service.
  - The service then generates a result which it associates with the calculation.
  - The updated calculation is returned as a response.
  - To be pedantic, the service should be stored by the service, associated with a unique id, and made accessible as a resource (e.g. `/calculation/{id}`) – we will skip this for simplicity.

Receiving Json in a Request

- We can model our calculation using the following class:

```
package calculation;

public class Calculation {
  public Calculation(char operator, int left, int right) {
    this.operator = operator;
    this.left = left;
    this.right = right;
  }

  public Calculation() {}

  public char operator;
  public long left;
  public long right;
  public long result;
}
```

- NOTE: the default constructor is required for Gson to work.

Receiving Json in a Request

- We can create a simple Calculator class to perform calculations:

```
package calculation;

public class Calculator {
  public Calculation perform(Calculation calculation) {
    long result = -1;
    switch (calculation.operator) {
    case '+’:
      result= calculation.left + calculation.right;
      break;
    case ‘-’: //...
    }
    calculation.result = result;
    return calculation;
  }
}
```

Receiving JSON in a Request

- Our Restlet must then:
  - Extract the JSON data in the body of the POST Request
  - Convert the JSON to a Calculation object
  - Invoke the perform(…) operation on the calculator
  - Return the updated calculation to the client

- In terms of code, we can access the request body through the following method:
  - `request.getEntityAsText()`

- The valid range of response types are defined in the `MediaType` class, for example:
  - `MediaType.APPLICATION_JSON`

Receiving Json in a Request

```
package calculation;

public class CalculationRestlet extends Restlet {
  private static Gson gson = new Gson();
  private static Calculator calculator = new Calculator();

  @Override
  public void handle(Request request, Response response) {
    if (request.getMethod().equals(Method.POST)) {
      Calculation calculation = gson.fromJson(
                    request.getEntityAsText(), Calculation.class);
      response.setEntity(
                    gson.toJson(calculator.perform(calculation)),
                    MediaType.APPLICATION_JSON);
      response.setStatus(Status.SUCCESS_OK);
    } else {
      response.setStatus(Status.CLIENT_ERROR_FORBIDDEN);
    }
  }
}
```

Interacting as a Client

- Client support is provided through the `ClientResource` class.
  - This class has methods that implement each of the main HTTP verbs.
  - The return value for each of these methods is a Representation object, which contains the response from the service.
  - This can be manipulated in a number of ways.

- For example, to invoke the "Hanging" Greeting Service, we can use the following code:

```
new ClientResource("http://localhost:8182/greetings/hanging")
                                        .get().write(System.out);
```

- For POST and PUT operations, the associated method takes one argument – the contents of the Request.

Interacting as a Client

```
package calculation;

public class CalculationClient {
  public static final String JSON_TEXT = "{ left:5, operator:'+', right:5 }";
  public static Gson gson = new Gson();

  public static void main(String[] args) throws Exception {
    new ClientResource("http://localhost:8182/calculation")
                          .post(JSON_TEXT).write(System.out);

    Calculation calculation = new Calculation('-', 8, 3);
    new ClientResource("http://localhost:8182/calculation")
                          .post(gson.toJson(calculation)).write(System.out);

    String json = new ClientResource("http://localhost:8182/calculation")
                          .post(gson.toJson(calculation)).getText();
    calculation = gson.fromJson(json, Calculation.class);
    System.out.println("The answer is: " + calculation.result);
  }
}
```