Module 15, part 2

- Review of Previous Week
- Servers and long running processes.
- Request and response
- Lambdas
- What is a request handler?
- JSON, how and why.
- What is Spark? What is Gradle?
- APIs and API Specs.
- Homework.

Review of Previous Week

- Server and Client Architecture
- Requests and Responses
- Protocols
- HTTP

This week we're building a Web Server

Servers and long running processes.

So far we've been working with processes that start and stop once they complete their task. But what about processes that have no known completion? For example, like an HTTP Server. The server doesn't know when a user is going to visit a webpage, so instead it just starts and waits for a request. And as long as nothing goes wrong, it will keep running and listen for more requests.

Many clients connect to a single server on the web

Request and response

This is the way clients and servers communicate. First, the client sends a `request`. Next, the server reads the request and returns a `response`.

HTTP request and response

A request is made up of the following parts:

- URL/Path
- Method
- Body (sometimes)

A response is made up of the following parts:

- Status code
- Content type (html, json, xml, etc.)
- Body

Lambdas

```
Route getStudentsRoute = (request, response) -> {
  response.type("application/json");

  List<Student> students = school.getStudents();
  String json = toJson(students);

  return json;
};
```

What is a request handler?

A request handler is the code that is designated to run when a specific request is made to your server.

```
get("/api/students", (request, response) -> {
  response.type("application/json");
  List<Student> students = school.getStudents();
  return toJson(students);
});
```

```
post("/api/students", (request, response) -> {
  response.type("application/json");

  UpdateStudentRequest updates = fromJson(
      request.body(),
      UpdateStudentRequest.class
  );

  Student studentToAdd = new Student(
    updates.firstName,
    updates.lastName
  );

  school.addStudent(studentToAdd);

  return toJson(studentToAdd);
});
```

What is Spark?

In the previous slides we were calling methods like `post` and `get`, but where do they come from? This is where Spark comes in. Spark simplifies on the common tasks that all servers need to do so that you can focus on what is unique about your application.

JSON, why.

JSON stands for "JavaScript Object Notation" and it is a format for encoding data.

```
class Student {
  public String firstName;
  public String lastName;
  public int grade;
}

Student me = new Student("Marcos", "Minond", 99);
```

```
{
  "firstName": "Marcos",
  "lastName": "Minond",
  "grade": 99,
}
```

JSON, how

Along with Spark, we have provided the `toJson` and `fromJson` methods to help you convert to and from JSON. Under the hood, these methods are using a library called `Gson`.

What is Gradle?

Gradle is what is known as a *build tool*. A build tool is a program that helps with certain tasks, like running code, running test, and downloading dependecies (third-party code).

Gradle is what we use to glue together Spark, Gson, and our own code.

APIs

API stands for Application Programming Interface, meaning the tools and functions for application software development. The Java Standard Library has APIs like `System.out.println()`.

Web APIs are kind of like functions on the internet that we can use in apps.

What is an API Spec?

Spec is short of "specification". A spec is documentation, so an API Spec is documentation for the API.

Why are API Specs needed?

Code needs documentation, and so do web APIs. If we write a thorough API spec, the client and server code can be written at the same time.

Frontend developers can use "mock data" to simulate the server. Backend developers can use "mock data" to simulate a client. When both are finished, the client and server will send the same sort of data.

What's in an API Spec?

1. `Resources`: Each resource represents a different entity, or a different class.
    - Example: a grading system would have a `student` resource because we need to get/set data involving students.
    - Example: a restaurant system would have a `reservations` resource to keep track of reserved tables.
2. `Endpoints`: Each endpoint is a combination of a path and HTTP method.
    - Example: a public library might have an endpoint like `POST /api/events/checkout` to create a new record of checking out some books.

Let's see an example

GET /api/students/{id}

Description: returns information about a specific student.

Example response body:

```
None
```

Example response body:

```
{
  "id": 2,
  "firstName": "Eric",
  "lastName": "Fortney",
  "grade": 90
}
```

Things to notice

- HTTP method (`GET`)
- Path (`/api/students/{id}`)
- Request parameter: `{id}` should be replaced by an ID number for a student
    - Example: `GET /api/students/5`
- Details about the `request body` and `response body`

A note about example requests/responses

- Not actual data on the server, just examples
- Pay attention to the JSON format
  - What are the data types?
  - What are the parameter names?

Practice (1 of 2)

Suppose we are building a system to keep track of flights. What resources should the API have?

Practice (2 of 2)

Suppose we are writing an API spec for a social media platform. What would each of these endpoints do?

- `POST /posts`
- `GET /posts/{postId}`
- `PUT /posts/{postId}`
- `GET /posts/{postId}/comments`
- `POST /posts/{postId}/comments`
- `PUT /posts/{postId}/comments/{commentId}`

Homework (1 of 3)

Experiment with adding students to the school

- You can find the hardcoded students in the generateSchool method in the Main.java file.

Homework (2 of 3)

Finish Implementing `PUT /api/students/:id/grade` endpoint

- Add public setGrade method to `Student` class.
- Follow example from PUT "/api/students/:id" endpoint.
- Get the student from the school by Id.
- Update the student grade with the setGrade method you added.
- Return the student as JSON.

Homework (3 of 3)

Implement `DELETE /api/students/:id` endpoint

- Add public `removeStudentById` method to `School` class.
- Add delete method with string route as first arg and lambda `(request, response) -> { }` as second arg.
- Follow example from POST "/api/students", instead of adding use the `removeStudentById` method.
- Return the deleted student as JSON.

Additional resources

1. Spark documentation, http://sparkjava.com/documentation

Reference list

1. Spark framework, http://sparkjava.com/
2. Google Gson library, https://github.com/google/gson
3. Gradle build tool, https://gradle.org/