Microservices Notes

John Paxton

Table of Contents

Resources

- GitHub Repository
- Broken into three projects:
- Demos: Demonstration code
- Labs: For exercises and experiments
- Solutions: the solved version of the labs

Platform

- Java 8
- Maven
- Spring Tool Suite
- Spring Boot, as assembled by the Spring Initializer.

Dependencies

These are the dependencies included in each of the projects in the repository: - Spring Boot DevTools: Developer tools like auto-restart and increased logging - Spring Web: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container. - Spring Data JPA: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate. - Rest Repositories: Exposing Spring Data repositories over REST via Spring Data REST. - Rest Repositories HAL Explorer: Browsing Spring Data REST repositories in your browser. - Apache Derby Database SQL: An open source relational database implemented entirely in Java. - H2 Database SQL: Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Microservices Pro and Con

Pros

- Flexible
- Focused
- Served by smaller teams
- More nimble
- Able to respond to client needs faster
- Same for iterating over improvements in code

Cons

- Breaking a monolithic service into several smaller services increases memory and filespace footprints
- Counterargument: memory and disk space are cheap
- Potential to overdo microservices and break into too many possible microservices
- Needs between services are less clear
- Requires good communication amongst teams with similar needs

Microservices technologies

Microservices stand on a foundation of proven technologies. These include: - HTTP: HyperText Transport Protocol - JSON: JavaScript Object Notation - XML: eXtended Markup Language - ORM: Object-Relational Mapping - Databases and other data stores

Spring Boot technologies

Spring Boot has specific implementations of the tools above

- HTTP / Web Server (usually Tomcat, but could use others)
- JSON and XML Parsing (Jackson 2 in the case of JSON)
- ORM: Easy integration with the Java Persistence API (JPA) implemented by Hibernate
- Databases: Can talk to several of the most popular databases

Spring Boot projects

The Spring Initializer will build a zipfile with the following assets - pom.xml with all dependencies for the chosen dependencies - Maven project structure - Packages as configured by the initializer - Spring Boot config files

Microservice API Naming patterns

Microservice APIs combine a URL with an HTTP method to determine what work the API will do. For example:

GET /books

would presumably retrieve a list of books.

Microservice APIs are usually organized around nouns, verbs, or a combination of the two.

Nouns

Think of nouns as organizing around a resource:

GET /books
GET /books/1
GET /books?title=The+Great+Gatsby
POST /books
PUT /books/5
PATCH /books/7
DELETE /books 10

Verbs

Verbs organize your API around use cases (instead of resources)

GET /shipping/report/5 POST /shipping/{order number}/update PUT /shipping/{order number} PATCH /shipping/items/add

Combinations of nouns and verbs

Combining the above, we get use cases for resources. This may be a good fit, or it might be too specific. Depends on the resources, use cases, and implementations

Adding a route

Spring Boot makes extensive use of annotations. XML configuration is also available, but this class strongly prefers Java annotations.

Create a controller for a route or group of routes by adding the `@RestController` annotation to a Plain Old Java Object (POJO).

```
package microservices.demos;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloWorldController { }
```

The `@RestController` is picked up by Spring Boot's auto-scan and registers itself as a Spring Component. This enables other annotations within this file.

Add a route with `@GetMapping`:

```
package microservices.demos;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloWorldController {

    @GetMapping("/hello-world")
    public String sayHello() {
        return "Hello, world!";
    }

}
```

`@GetMapping` maps a particular URL (`/hello-world`) to an HTTP GET request. There are similar mappings for other HTTP verbs:

- `@PostMapping`
- `@PutMapping`
- `@PatchMapping`

- `@DeleteMapping`

Testing a route

There are several ways to test routes. We will start with an integration testing level approach. Instead of worrying about mocking out various parts of a system, we will test the integration of various parts.

Testing includes setting up JUnit (included with any project created by the Spring Initializer) and building and configuring an HTTP client.

Note also that in addition to JUnit's assertions, AssertJ is available as well.

```
package microservices.demos;

import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.test.web.server.LocalServerPort;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HelloWorldTest {
    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate client;

    @Test
    public void testHelloWorld() throws Exception {
        String testUrl = String.format("http://localhost:%d/hello-world", port);
        String response = client.getForObject(testUrl, String.class);
    // JUnit assertion
        assertEquals("Hello, world!", response);

    // AssertJ assertion
        assertThat(response).contains("Hello");
    }
}
```

Annotate the class with `@SpringBootTest`. When Maven's test target is run, or when running this file standalone, this annotation sets up a server for us to test against.

Optionally add the `webEnvironment` property to set a random port for the test server.

Get access to the random server port with `@LocalServerPort`

The HTTP client we want to configure is provided by the `TestRestClient` class. Spring Boot will inject a configured instance of that class via the Spring `@Autowired` annotation.

Annotate the test with JUnit's `@Test` annotation.

`TestRestClient` has several methods that allow you to retrieve the body of an HTTP request. They typically take the form `getForObject(url, type)` where type is the Java type we want to convert the body of the message to. This is expressed with the `.class` property.

Later we will see a similar version in `getForEntity()`.

Use assertions to see if you retrieved the value you want.

Retrieving an object

We can have our route return an object, rather than a String. Thanks to Spring Boot's already-configured Jackson JSON conversion library, the object will be automatically JSON-ified (as it were).

Here's a simple object:

```
package microservices.demos;

import java.util.Objects;

public class Greeting {

    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() {
        return id;
    }

    public String getContent() {
        return content;
    }

}
```

And here's a controller for retrieving that object:

```
package microservices.demos;

import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private final AtomicLong counter = new AtomicLong(1);

    @GetMapping("/greeting")
```

```
    public Greeting greeting() {
        return new Greeting(counter.getAndIncrement(), "Hello, World!");
    }
}
```

Note the use of `@RestController` and `@GetMapping`. But what has really changed is we have a method attached to `@GetMapping` which returns a custom object/type, rather than a Java standard String.

Behind the scenes, Jackson 2 will convert the `Greeting` instance into a JSON representation. Jackson follows similar rules to Entity Beans in JPA (though there are differences): public getters can define properties.

Testing retrieving an object

The test will not change all that much. Since we're expecting an Object, we will need to change the second argument to `getForObject` to the appropriate `class` type. And we should change the nature of our test, since it's not looking directly at a String anymore.

```
package microservices.demos;

import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.test.web.server.LocalServerPort;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HelloWorldTest {
    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate client;

    @Test
    public void testHelloWorld() throws Exception {
        String testUrl = String.format("http://localhost:%d/hello-world", port);
        Greeting response = client.getForObject(testUrl, Greeting.class);
    // JUnit assertion
        assertEquals("Hello, world!", response.getContent());

    // AssertJ assertion
        assertThat(response.getContent()).contains("Hello");
    }
}
```

Typical Microservice endpoints

Microservices, particularly those that are arranged around resources/nouns, have a typical set of API endpoints:

GET /books → Returns an array of all books
GET /books/{id} → Returns the book corresponding to {id}
POST /books → creates a new book, returns representation of same
PUT /books/{id} → replaces the book at {id} with whatever is passed as part of the request
PATCH /books/{id} → updates the book at {id} with the content passed as part of the request
DELETE /books/{id} → deletes or otherwise invalidates/hides the book at {id}

Additionally, for the GET methods, it is typical to be able to pass parameters:

GET /books?title=The+Great+Gatsby → Find books with a title of "The Great Gatsby" GET /books?_limit=50&_start=10 → Return only fifty books, start after the first ten in the collection

We will have all the tools to implement these URLs soon.

Working with collections of data

We are implementing `GET /greetings`, that is, GET a collection of a resource.

From the Java perspective, Jackson does not care if you hand it an array or a Java Collection. It knows how to convert both into a JSON Array. (JSON does not have extensive data structures.)

We can add the following to our `GreetingController`:

```
@GetMapping("/greetings")
public List<Greeting> greetings() {
  greetingsList = new ArrayList<>();
  greetingsList.add(new Greeting(1, "Buddy"));
  greetingsList.add(new Greeting(2, "Pal"));
  greetingsList.add(new Greeting(3, "Folks"));
  greetingsList.add(new Greeting(4, "People"));
  greetingsList.add(new Greeting(5, "Everyone"));
  return greetingsList;
}
```

We build a `List`, which could have just as easily been an array, and return its values.

We do not want to construct the List/array every time the `/greetings` URL is accessed. We can use Java annotations to set up `greetingsList` when the controller is initialized.

```
package microservices.demos;

import javax.annotation.PostConstruct;
import java.util.concurrent.atomic.AtomicLong;
import java.util.List;
import java.util.ArrayList;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private final AtomicLong counter = new AtomicLong(1);
  private List<Greeting> greetingsList;

    @PostConstruct
    public void init() {
        greetingsList = new ArrayList<>();
        greetingsList.add(new Greeting(1, "Buddy"));
        greetingsList.add(new Greeting(2, "Pal"));
```

```
        greetingsList.add(new Greeting(3, "Folks"));
        greetingsList.add(new Greeting(4, "People"));
        greetingsList.add(new Greeting(5, "Everyone"));
    }

  @GetMapping("/greetings")
  public List<Greeting> greetings() {
    return greetingsList;
  }

  @GetMapping("/greeting")
  public Greeting greeting() {
    return new Greeting(counter.getAndIncrement(), "Hello, World!");
  }
}
```

We have added the `javax.annotation.PostConstruct` annotation to an `init` method. There is nothing special about the method name. The annotation makes it special. `@PostConstruct` designates a method to run once after the object has been constructed. We can use this as a setup method, to fill out the greetingsList with data.

Testing a collection of data

Testing a collection of data can be simple or complex depending on your approach. Let's try for simple first.

```
@Test
public void testGreetingsAsArray() throws Exception {
  Greeting[] testGreetings = this.restTemplate.getForObject("http://localhost:" + port +
"/greetings", Greeting[].class);
  assertThat(testGreetings.length).isEqualTo(5);
}
```

In this example, we only had to change the class argument to `getForObject`. Changing it to an Array class, rather than a scalar class, is not difficult, and our test will run successfully.

But what is we wanted to specifically test for a `List<Greeting>` or something similar? The complication is that you cannot pass a generic type as part of a `class` property. That is, you cannot do this:

```
List<Greeting> testGreetings = this.restTemplate.getForObject(url, List<Greeting>.class);
```

Generic types cannot be used with the `class` property.

`TestRestTemplate` does provide a tool for testing output as a `Collection`.

```
@Test
public void testGreetingsAsCollection() throws Exception {
  ResponseEntity<List<Greeting>> testGreetings;
  String url = "http://localhost:8080/greetings";

  testGreetings = restTemplate.exchange(url,
                    HttpMethod.GET,
                    null,
                    new ParameterizedTypeReference<List<Greeting>>() {});
  List<Greeting> greetingsList = testGreetings.getBody();
  assertThat(greetingsList.size()).isEqualTo(5);
}
```

Spring provides `ParameterizedTypeReference` to solve the problem of a class type with a late-binding generic. (All generics are, of course, late-binding.) It does not provide an exact match to something like `List<Greeting>.class`, so we cannot use it with `getForObject`. But Spring Boot's `TestRestTemplate` adds an `exchange` method which allows passing a `ParameterizedTypeReference` instead of a `Class`.

This is a lower-level method with greater configuration possibilities. We can pass in the URL, an enum for the HTTP method we're using (`GET`), an `HttpEntity` which could configure headers and body (we left this null) and the response type as a `ParameterizedTypeReference`.

That returns a `ResponseEntity` wrapped around the subtype of the `ParameterizedTypeReference`. A call to ResponseEntity's `getBody` method will return the body of the HTTP response as the appropriate Java type.

Query Parameters

Query parameters are parameters passed as part of the URL. (Contrast with the request body, which is the portion passed in the body of the request, often thought of as POST params.) We can pass and parse query parameters to our controller via the `@RequestParam` annotation.

```
@GetMapping("/greeting")
public Greeting greeting(@RequestParam(value = "name") String name) {
  return new Greeting(10, String.format("Hello, %s!", name));
}
```

`@RequestParam` can take a few useful arguments:

- name: What's the name of the parameter we are interested in?
- required (boolean): Is this parameter required? Defaults to true.
- defaultValue (String or appropriate): If this value is required and not present, what should we use as a default value?

If you want to retrieve all the parameters in the query string, you could use this code:

```
@RequestParam Map<String, String> params
```

Testing request parameters

To specifically test query string parameters, you can simply add them directly to the URL string.