
Spring Restbucks



This project is a sample implementation of the Restbucks application described in the book REST in Practice by Jim Webber, Savas Parastatidis and Ian Robinson. It's a showcase for bringing different Spring eco-system technologies together to implement a REST web service. The application uses HAL as the primary representation format. The server implementation is accompanied by a hypermedia-aware Android client that adapts to changes on the server dynamically.

Quickstart

From the command line do:

```
1 git clone https://github.com/odrotbohm/spring-restbucks.git
2 cd spring-restbucks/server
3 mvn clean package
4 java -jar target/*.jar
```

The application ships with the HAL browser embedded, so simply browsing to <http://localhost:8080/browser/index.html> will allow you to explore the web service.

Note, that the curie links in the representations are currently not backed by any documents served but they will be in the future. Imagine simple HTML pages being served documenting the individual relation types.

IDE setup

For the usage inside an IDE do the following:

1. Make sure you have an Eclipse with m2e installed (preferably STS).
2. Install Lombok.
 1. Download it from the project page.
 2. Run the JAR (double click or `java -jar ...`).
 3. Point it to your Eclipse installation, run the install.
 4. Restart Eclipse.
3. Import the checked out code through *File > Import > Existing Maven Project...*

Project description

The project uses:

- Spring Boot
- Spring (MVC)
- Spring Data JPA
- Spring Data REST
- Spring HATEOAS
- Spring Plugin
- Spring Security
- Spring Session

The implementation consists of mainly two parts, the `order` and the `payment` part. The `Orders` are exposed as REST resources using Spring Data REST's capability to automatically expose Spring Data JPA repositories contained in the application. The `Payment` process and the REST application protocol described in the book are implemented manually using a Spring MVC controller (`PaymentController`).

Here's what the individual projects used contribute to the sample in from a high-level point of view:

Spring Data JPA

The Spring Data repository mechanism is used to reduce the effort to implement persistence for the domain objects to the declaration of an interface per aggregate root. See `OrderRepository` and `PaymentRepository` for example. Beyond that, using the repository abstract enables the Spring Data REST module to do its work.

Spring Data REST

We're using Spring Data REST to expose the `OrderRepository` as REST resource without additional effort.

Spring HATEOAS

Spring HATEOAS provides a generic `Resource` abstraction that we leverage to create hypermedia-driven representations. Spring Data REST also leverages this abstraction so that we can deploy `ResourceProcessor` implementations (e.g. `PaymentOrderResourceProcessor`) to enrich

the representations for `Order` instance with links to the `PaymentController`. Read more on that below in the Hypermedia section.

The final important piece is the `EntityLinks` abstraction that allows to create `Link` instance in a type-safe manner avoiding the repetition of URI templates and parts all over the place. See `PaymentLinks` for example usage.

Spring Plugin

The Spring Plugin library provides means to collect Spring beans by type and exposing them for selection based on a selection criterion. It basically forms the foundation for the `EntityLinks` mechanism provided in Spring HATEOAS and our custom extension `RestResourceEntityLinks`.

Spring Security / Spring Session

The `spring-session` branch contains additional configuration to secure the service using Spring Security, HTTP Basic authentication and Spring Session's HTTP header based session strategy to allow clients to obtain a security token via the `X-Auth-Token` header and using that for subsequent requests.

If you check out the branch and run the sample you should be able to follow this interaction (I am using HTTPie here)

```
1 $ http :8080
2 HTTP/1.1 401 Unauthorized...
3
4 WWW-Authenticate: Basic realm="Spring RESTBucks"
```

You can now authenticate using the default credentials (the password `password` is configured in `application.properties`).

```
1 $ http :8080 --auth=user:password
2 HTTP/1.1 200 OK
3 Content-Type: application/hal+json;charset=UTF-8-
4
5 X-Auth-Token: ef005d62-b69b-4675-b920-d340a238e857
```

Now you can use the presented auth token in further requests:

```
1 $ http :8080 X-Auth-Token:ef005d62-b69b-4675-b920-d340a238e857
2 HTTP/1.1 200 OK
3 Content-Type: application/hal+json;charset=UTF-8-
```

Documentation / Client Stub Generation

The `restdocs` branch contains the test for the order payment process augmented with setup to generate Asciidoctor snippets documenting the executed requests and expectations on the responses. These snippets are included from the general Asciidoctor documents in `src/main/asciidoc`, turned into HTML and packaged into the application itself during the build (run `mvn clean package`). The docs are then pointed to by a CURIE link in the HAL response (see `Restbucks.curieProvider()`) so that they appear in the `docs` column in the HAL browser (run `mvn spring-boot:run` and browse to `http://localhost:8080`) the service ships.

The use of the Spring Cloud Contract extension also causes WireMock stubs to be generated by those tests and placed into `generated-snippets/stubs/...`. The `pay-order` subfolder for the individual setups:

Open questions

- How to package the stubs up so that they can be used by a client project?
- How do the client tests select a particular flow to be used for stubbing?
- How to make sure WireMock “understands” the state transitions, i.e. responses to requests to a resource changing based on intermediate requests?
- How to set up WireMock so that the client doesn’t necessarily have to select a particular flow to operate against?

Miscellaneous

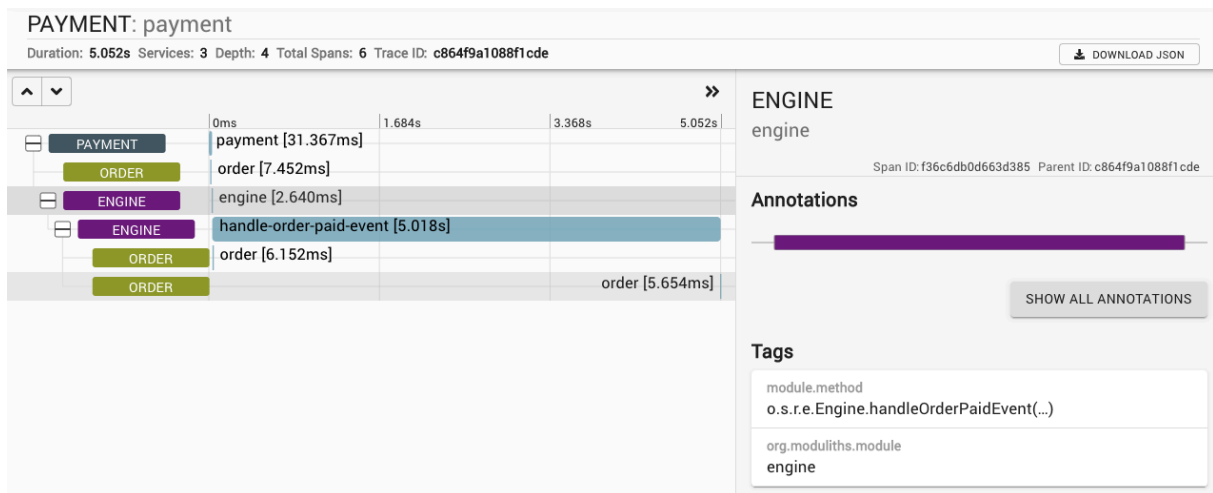
The project uses Lombok to reduce the amount of boilerplate code to be written for Java entities and value objects.

Observability

The project uses Spring Moduliths’ observability features to inspect the runtime interaction between the logical modules of RESTBucks. To use and see this, run the application with the `observability` Maven profile enabled:

```
1 $ mvn spring-boot:run -Pobservability
```

That profile adds some dependencies, like Spring Cloud Sleuth as well as its integration with Zipkin. Make sure you have Zipkin running as described here. Interactions with the system will now expose the logical module invocation and their choreography



See how the triggering of the payment for an order changes the order state, kicks of the preparation engine and tweaks the order's state in turn at the start and end of the process.

Hypermedia

A core focus of this sample app is to demonstrate how easy resources can be modeled in a hypermedia driven way. There are two major aspects to this challenge in Java web-frameworks:

1. Creating links and especially the target URL in a clean and concise way, trying to avoid the usage of Strings to define URI mappings and targets and especially the repetition of those. On the server side, we'd essentially like to express "link to the resource that manages **Order** instances" or "link to the resource that manages a single **Order** instance."
2. Cleanly separate resource functionality implementation but still allowing to leverage hypermedia to advertise new functionality for resources as the service implementation evolves. This essentially boils down to an enrichment of resource representations with links.

In our sample the core spot these challenges occur is the **payment** subsystem and the **PaymentController** in particular.

ALPS

The repository currently contains an **alps** branch that is based on a feature branch of Spring Data REST to automatically expose resources that server ALPS metadata for the resources exposed.

```
1 git checkout alps
2 mvn spring-boot:run
3 curl http://localhost:8080
```

This will return:

```
1 {
2   "_links" : {...
3
4     "profile" : {
5       "href" : "http://localhost:8080/alps"
6     }
7   }
8 }
```

You can then follow the [profile](#) link to access all available ALPS resources, such as the one for [orders](#), a link relation also listed in the response for the root resource.

Using the AOT mode for the server application

You can use Spring Native's AOT support to build the application for optimized Spring Boot startup that pre-processes the Spring configuration at runtime but still run the app as JVM application. To achieve that, run the build using the [aot](#) profile. Then activate the AOT mode of the application on startup.

```
1 $ ./mvnw -Paot...
2
3 $ java -Dspring.aot.enabled=true -jar target/*.jar
```

Building native images for the server application

Make sure you have GraalVM 21+ installed (on a Mac: `brew install graalvm`), and your console shows the following output when running `java -version`:

```
1 java 21 2023-09-19
2 Java(TM) SE Runtime Environment Oracle GraalVM 21+35.1 (build 21+35-
   jvmci-23.1-b15)
3 Java HotSpot(TM) 64-Bit Server VM Oracle GraalVM 21+35.1 (build 21+35-
   jvmci-23.1-b15, mixed mode, sharing)
```

The image can then be built as follows:

```
1 $ ./mvnw -Pnative
```

After the build has completed, start the binary:

```
1 $ ./target/spring-restbucks.....
2
```

```
3 [main] org.springframework.samples.restbucks.Restbucks : Started
Restbucks in 0.366 seconds (process running for 0.446)
```

Using Class Data Sharing

Class Data Sharing (CDS) is an execution optimization technique available on recent JVMs (JDK 21 or newer). To make use of that make sure you have a recent JDK installed.

```
1 $ java -version
2 openjdk version "22" 2024-03-19
3 OpenJDK Runtime Environment Temurin-22+36 (build 22+36)
4 OpenJDK 64-Bit Server VM Temurin-22+36 (build 22+36, mixed mode)
```

Next, create a CDS base image for that JDK:

```
1 java -Xshare:dump
```

Finally, execute `cds.sh` located in the root of the server project. It will perform the following steps:

1. Build the application with the AOT optimizations applied.
2. Unpack the JAR file and create the index files necessary for CDS execution
3. Perform an initial training run that produces the metadata file necessary for the running application.
4. Finally, start the application in CDS-enabled mode.

You should see the startup time of the final run to roughly be half of the time the non-CDS run.

```
1 $ ./cds.sh.....
2
3 : Started Restbucks in 1.475 seconds (process running for 1.636)
```

The Android client

The Android sample client can be found in `android-client` and is a most rudimentary implementation of a client application that leverages hypermedia elements to avoid strong coupling to the server.

Quickstart

1. Have Android Studio installed.
2. Import the project from `android-studio`.
3. Make sure the server runs.

-
4. In Android Studio, run the application in the simulator.
 5. In the application, browse existing orders, trigger payments and cancellations.

The main abstraction working with hypermedia elements is [HypermediaRemoteResource](#). It allows to define client behavior conditionally based on the presence of links in the representations. For example, see this snippet from [OrderDetailsActivity](#):

```
1 resource.ifPresent("restbucks:cancel") {  
2  
3     with(cancel_button) {  
4  
5         visibility = View.VISIBLE  
6  
7         setOnClickListener {  
8  
9             CancelOrder(Link(selfUri)).execute().get()  
10  
11             it.context.startActivity(Intent(it.context, MainActivity::class.  
12                 java))  
13         }  
14     }
```

The closure is only executed if the link with a relation `restbucks:cancel` is actually present. It then enables the button and registers `CancelOrder` action for execution on invocation.

TODO - complete