**1. Overview**

In this lab, we'll be discussing **what HAL is and why it's useful, before introducing the HAL browser**.

We'll then use Spring to build a simple REST API with a few interesting endpoints and populate our database with some test data.

Finally, using the HAL browser, we'll explore our REST API and discover how to traverse the data contained within.

**2. HAL and the HAL Browser**

[JSON Hypertext Application Language](http://stateless.co/hal_specification.html), or HAL, is a simple format that**gives a consistent and easy way to hyperlink between resources in our API**. Including HAL within our REST API makes it much more explorable to users as well as being essentially self-documenting.

It works by returning data in JSON format which outlines relevant information about the API.

The **HAL model revolves around two simple concepts.**

*Resources,* which contain:

* Links to relevant URIs
* Embedded Resources
* State

*Links:*

* A target URI
* A relation, or rel, to the link
* A few other optional properties to help with depreciation, content negotiation, etc

The HAL browser was created by the same person who developed HAL and **provides an in-browser GUI to traverse your REST API**.

We'll now build a simple REST API, plug in the HAL browser and explore the features.

**3. Dependencies**

Below is the single dependency needed to integrate the HAL browser into our REST API. You can find the rest of the dependencies for the API in the [GitHub code](https://github.com/eugenp/tutorials/blob/master/persistence-modules/spring-data-rest/pom.xml).

Firstly, [the dependency](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.data%22%20AND%20a%3A%22spring-data-rest-hal-browser%22) for Maven-based projects:

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-rest-hal-explorer</artifactId>

<version>3.4.1.RELEASE</version>

</dependency>

If you're building with Gradle, you can add this line to your *build.gradle*file:

compile group: 'org.springframework.data', name: 'spring-data-rest-hal-explorer', version: '3.4.1.RELEASE'

**4. Building a Simple REST API**

**4.1. Simple Data Model**

In our example, we'll be setting up a simple REST API to browse different books in our library.

Here, we define a simple book entity which contains appropriate annotations so that we can persist the data with Hibernate:

@Entity

**public** **class** **Book** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** **long** id;

@NotNull

@Column(columnDefinition = "VARCHAR", length = 100)

**private** String title;

@NotNull

@Column(columnDefinition = "VARCHAR", length = 100)

**private** String author;

@Column(columnDefinition = "VARCHAR", length = 1000)

**private** String blurb;

**private** **int** pages;

// usual getters, setters and constructors

}

**4.2. Introducing a CRUD Repository**

Next, we'll need some endpoints. To do this, we can**leverage the *PagingAndSortingRepository***and specify that we want to get data from our *Book*entity.

This Class**provides simple CRUD commands, as well as paging and sorting capabilities** right out of the box:

@Repository

**public** **interface** **BookRepository** **extends** **PagingAndSortingRepository**<Book, Long> {

@RestResource(rel = "title-contains", path="title-contains")

Page<Book> **findByTitleContaining**(@Param("query") String query, Pageable page);

@RestResource(rel = "author-contains", path="author-contains", exported = false)

Page<Book> **findByAuthorContaining**(@Param("query") String query, Pageable page);

}

If this looks a bit strange, or if you'd like to know more about Spring Repositories, you can read more [here](https://www.baeldung.com/spring-data-repositories).

We've extended the repository by adding two new endpoints:

* *findByTitleContaining –*returns books that contain the query included in the title
* *findByAuthorContaining –* returns books from the database where the author of a book contains the query

Note that our **second endpoint contains the *export = false*attribute**. This attribute **stops the HAL links being generated for this endpoint**, and won't be available via the HAL browser.

Finally, we'll load our data when Spring is started by defining a class which implements the *ApplicationRunner* interface. You can find the code on [GitHub](https://github.com/eugenp/tutorials/blob/master/persistence-modules/spring-data-rest-2/src/main/java/com/baeldung/halbrowser/config/DBLoader.java).

**5. Installing the HAL Browser**

The setup for the HAL browser is remarkably easy when building a REST API with Spring. As long as we have the dependency, Spring will auto-configure the browser, and make it available via the default endpoint.

All we need to do now is press run and switch to the browser. The HAL browser will then be available on *http://localhost:8080/*

**6. Exploring Our REST API With the HAL Browser**

The **HAL browser is broken down into two parts – the explorer and the inspector**. We'll break down and explore each section separately.

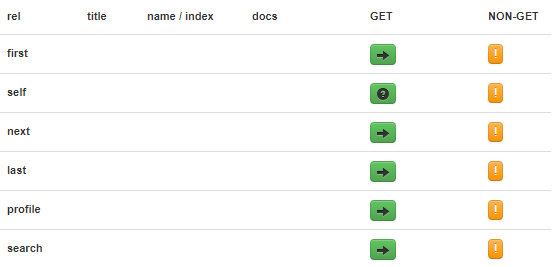
**6.1. The HAL Explorer**

As it sounds, the explorer is devoted to **exploring new parts of our API relative to the current endpoint**. It contains a search bar, as well as text boxes to display **Custom Request Headers and Properties** of the current endpoint.

Below these, we have the links section and a clickable list of Embedded Resources.

**6.2. Using Links**

If we navigate to our */books*endpoint we can view the existing links:



These **links are generated from the HAL** in the adjacent section:

"\_links": {

"first": {

"href": "http://localhost:8080/books?page=0&size=20"

},

"self": {

"href": "http://localhost:8080/books{?page,size,sort}",

"templated": true

},

"next": {

"href": "http://localhost:8080/books?page=1&size=20"

},

"last": {

"href": "http://localhost:8080/books?page=4&size=20"

},

"profile": {

"href": "http://localhost:8080/profile/books"

},

"search": {

"href": "http://localhost:8080/books/search"

}

},

If we move to the search endpoint, we can also view the custom endpoints we created using the *PagingAndSortingRepository:*

{

"\_links": {

"title-contains": {

"href": "http://localhost:8080/books/search/title-contains{?query,page,size,sort}",

"templated": true

},

"self": {

"href": "http://localhost:8080/books/search"

}

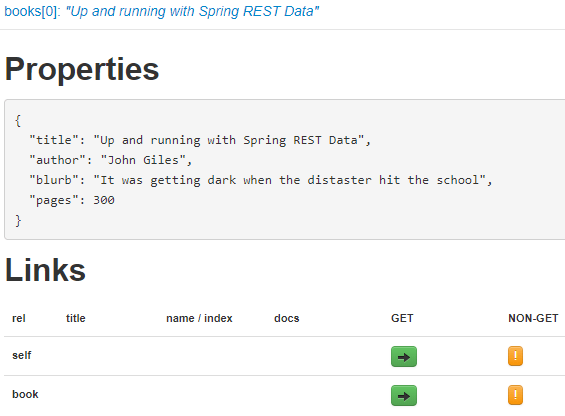
}

}

The HAL above shows our ***title-contains* endpoint displaying suitable search criteria.** Note how the *author-contains* endpoint is missing since we defined that it should not be exported.

**6.3. Viewing Embedded Resources**

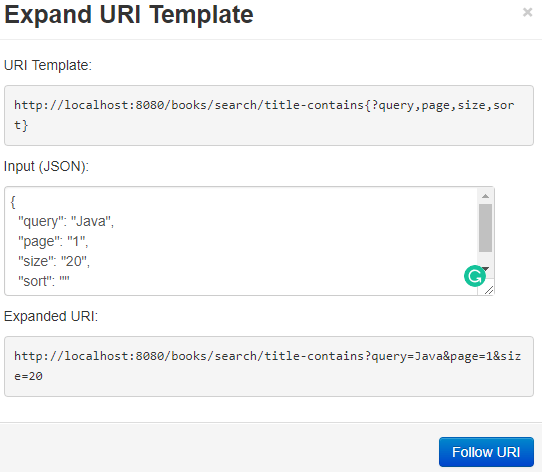
Embedded Resources show the**details of the individual book records** on our */books* endpoint. Each resource also contains its own *Properties*and *Links*section:



**6.4.  Using Forms**

The question mark button in the GET column within the links section denotes that a form modal can be used to enter custom search criteria.

Here is the form for our *title-contains*endpoint:



Our custom URI returns the **first page of 20 books where the title contains the word ‘Java'.**

**6.5. The Hal Inspector**

The inspector makes up the right side of the browser and contains the **Response Headers and Response Body.**This **HAL data is used to render the Links and Embedded Resources**that we saw earlier in the tutorial.

**7. Conclusion**

In this lab, we've summarized what HAL is, why it's useful and why it can help us to create **superior self-documenting REST APIs.**

We have built a simple REST API with Spring which implements the *PagingAndSortingRepository*, as well as defining our own endpoints. We've also seen how to **exclude certain endpoints from the HAL browser**.

After defining our API, we populated it with test data and explored it in detail with the help of the HAL browser. We saw how the HAL browser is structured, and the UI controls which allowed us to step through the API and explore its data.