tags						projects			
rest	hateoas	hypermedia	security	testing	oauth	spring-framework	spring-hateoas	spring-security	spring-security-oauth

Building REST Services with Spring

REST has quickly become the de-facto standard for building web services on the web because they're easy to build and easy to consume.

There's a much larger discussion to be had about how REST fits in the world of microservices, but - for this tutorial - let's just look at building RESTful services.

Why REST? *REST In Practice* proffers, to borrow Martin Fowler's phrasing, "the notion that the web is an existence proof of a massively scalable distributed system that works really well, and we can take ideas from that to build integrated systems more easily." I think that's a pretty good reason: REST embraces the precepts of the web itself, and embraces its architecture, benefits and all.

What benefits? Principally all those that come for free with HTTP as a platform itself. Application security (encryption and authentication) are known quantities today for which there are known solutions. Caching is built into the protocol. Service routing, through DNS, is a resilient and well-known system already ubiquitously support.

REST, however ubiquitous, is not a standard, *per se*, but an approach, a style, a *constraint* on the HTTP protocol. Its implementation may vary in style, approach. As an API consumer this can be a frustrating experience. The quality of REST services varies wildly.

Dr. Leonard Richardson put together a maturity model that interprets various levels of compliance with RESTful principles, and grades them. It describes 4 levels, starting at **level 0**. Martin Fowler has a very good write-up on the maturity model

- **Level 0**: the Swamp of POX - at this level, we're just using HTTP as a transport. You could call SOAP a **Level 0** technology. It uses HTTP, but as a transport. It's worth mentioning that you could also use SOAP on top of something like JMS with no HTTP at all. SOAP, thus, is *not* RESTful. It's only just HTTP-aware.
- **Level 1**: Resources - at this level, a service might use HTTP URIs to distinguish between nouns, or entities, in the system. For example, you might route requests to `/customers`, `/users`, etc. XML-RPC is an example of a **Level 1** technology: it uses HTTP, and it can use URIs to distinguish endpoints. Ultimately, though, XML-RPC is not RESTful: it's using HTTP as a transport for something else (remote procedure calls).
- **Level 2**: HTTP Verbs - this is the level you want to be at. If you do **everything** wrong with Spring MVC, you'll probably still end up here. At this level, services take advantage of native HTTP qualities like headers, status codes, distinct URIs, and more. This is where we'll start our journey.

- **Level 3**: Hypermedia Controls - This final level is where we'll strive to be. Hypermedia, as practiced using the HATEOAS ("HATEOAS" is a truly welcome acronym for the mouthful, "Hypermedia as the Engine of Application State") design pattern. Hypermedia promotes service longevity by decoupling the consumer of a service from intimate knowledge of that service's surface area and topology. It **describes** REST services. The service can answer questions about what to call, and when. We'll look at this in depth later.

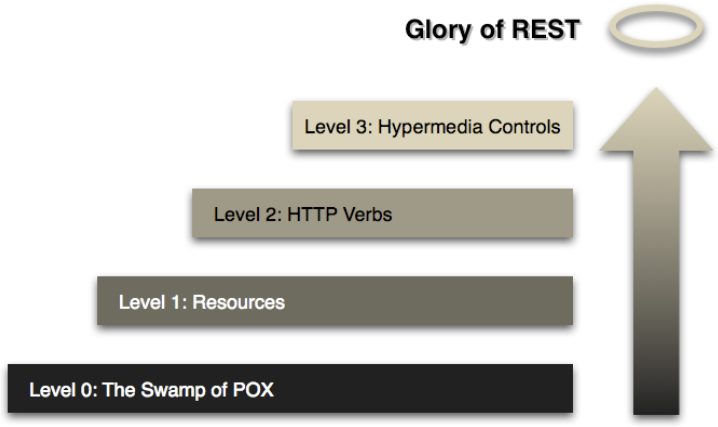

Figure 1. Leonard Richardson's Maturity Model

Getting Started

As we work through this tutorial, we'll use Spring Boot Spring Boot removes a lot of the boilerplate typical of application development. You can get started by going to the Spring Initializr and selecting the checkboxes that correspond to the **type** of workload your application will support. In this case, we're going to build a **web** application. So, select "Web" and then choose "Generate." A `.zip` will start downloading. Unzip it. In it, you'll find a simple, Maven or Gradle-ready directory structure, complete with a Maven `pom.xml` and a Gradle `build.gradle`. Delete the build artifact you don't want to use. Most people are using Maven these days, so - where appropriate - the examples in this tutorial will be Maven based. However, if you haven't looked at Gradle, do. It's **very** nice.

Spring Boot can work with any IDE. You can use Eclipse, IntelliJ IDEA, Netbeans, etc. The Spring Tool Suite is an open-source, Eclipse-based IDE distribution that provides a superset of the Java EE distribution of Eclipse. It includes features that making working with Spring applications even easier. It is, by no means, required. But consider it if you want that extra **oomph** for your keystrokes. Here's a video demonstrating how to get started with STS and Spring Boot. This is a general introduction to familiarize you with the tools.

The Story so Far…

All of our examples will be based on Spring Boot. We'll reprint the same setup code for each example. Our example models a simple bookmark service, à la Instapaper or other cloud-based bookmarking services. Our bookmark service simply collects a URI, and a description. All bookmarks belong to a user account. This relationship is modeled using JPA and Spring Data JPA repositories in the `model` module.

We won't dive too much into the code. We're using two JPA entities to model the records as they'll live in a database. We're using a standard SQL database to store our records so that the domain is as immediately useful to as large an audience as possible.

The first class models our user account. Aptly, with a JPA entity called `Account`.

Note	Amazingly, the following class is one of the *noisiest* - mostly because of the Java language's verbosity.

`model/src/main/java/bookmarks/Account.java`

```
link:./model/src/main/java/bookmarks/Account.java[]
```

Each `Account` may have no, one, or many `Bookmark` entities. This is a 1:N relationship. The code for the `Bookmark` entity is shown below:

`model/src/main/java/bookmarks/Bookmark.java`

```
link:./model/src/main/java/bookmarks/Bookmark.java[]
```

We'll use two Spring Data JPA repositories to handle the tedious database interactions. Spring Data repositories are typically interfaces with methods supporting reading, updating, deleting, and creating records against a backend data store. Some repositories also typically support data paging, and sorting, where appropriate. Spring Data synthesizes implementations based on conventions found in the naming of the methods in the interface. There are multiple repository implementations besides the JPA ones. You can use Spring Data MongoDB, Spring Data GemFire, Spring Data Cassandra, etc.

One repository will manage our `Account` entities, called `AccountRepository`, shown below. One custom finder-method, `findByUsername`, will, *basically*, create a JPA query of the form `select a from Account a where a.username = :username`, run it (passing in the method argument `username` as a named parameter for the query), and return the results for us. Convenient!

`model/src/main/java/bookmarks/AccountRepository.java`

```
link:./model/src/main/java/bookmarks/AccountRepository.java[]
```

Here's the repository for working with `Bookmark` entities.

`model/src/main/java/bookmarks/BookmarkRepository.java`

```
link:./model/src/main/java/bookmarks/BookmarkRepository.java[]
```

The `BookmarkRepository` has a similar finder method, but this one dereferences the `username` property on the `Bookmark` entity's `Account` relationship, ultimately requiring a join of some sort. The JPA query it generates is, **roughly**, `SELECT b from Bookmark b WHERE b.account.username = :username`.

Our application will use Spring Boot. A Spring Boot application is, at a minimum, a `public static void main` entry-point and the `@EnableAutoConfiguration` annotation. This tells Spring Boot to help out, wherever possible. Our `Application` class is also a good place to stick of odds and ends, like `@Bean` definitions. Here's what our simplest `Application.java` class will look like:

```
link:./rest/src/main/java/bookmarks/Application.java[]
```

Once started, Spring Boot will call all beans of type `CommandLineRunner`, giving them a callback. In this case, `CommandLineRunner` is an interface with one **abstract** method, which means that - in the world of Java 8 - we can substitute its definition with a lambda expression. All the examples in this tutorial will use Java 8. There is no reason, however, that you couldn't use Java 6 or 7, simply substituting the more concise lambda syntax for a slightly more verbose anonymous inner class implementing the interface in question.

HTTP is the Platform

HTTP URIs are a natural way to describe hierarchies, or relationships. For example, we might start our REST API at the account level. All URIs start with an account's username. Thus, for an account named `bob`, we might address that account as `/users/bob` or even just `/bob`. To access the collection of bookmarks for that user, we can **descend** one level down (like a file system) to the `/bob/bookmarks` resource.

REST does **not** prescribe a representation or encoding. REST, short for **Representational STate Transfer**, defers to HTTP's content-negotiation mechanism to let clients and services agree upon a mutually understood representation of data coming from a service, if possible. There are many ways to handle content negotiation, but in the simplest case, a client sends a request with an `Accept` header that specifies a comma-delimited list of acceptable mime types (for example: `Accept: application/json, application/xml, */*`). If the service can produce any of those mime types, it responds with a representation in the first understood mime type.

We can use HTTP verbs to manipulate the data represented by those URIs.

- the HTTP `GET` verb tells the service to **get**, or retrieve, the resource designated by a URI. How it does this is, of course, implementation specific. The backend code might talk to a database, a file system, another webservice, etc. The client doesn't need to be aware of this, though. To the client, all resources are HTTP resources, and in the world of HTTP, there's only one way to ask for data: `GET` . `GET` calls have no body in the request, but typically return a body. The response to an HTTP `GET` request for `/bob/bookmarks/6` might look like:

```
link:./snippets/simple-json-response.txt[]
```

- the HTTP `DELETE` verb tells the service to remove the resource designated by a URI. Again, this is implementation specific. `DELETE` calls have no body.
- the HTTP `PUT` verb tells the service to update the resource designated by a URI with the body of the enclosed request. Thus, to update the resource at `/bob/bookmarks` , I might send the same JSON representation returned from the `GET` call, with updated fields. The service will **replace** the value.
- the HTTP `POST` verb tells the service to **do something** with the enclosed body of the request. There's no hard and fast rules here, but typically an HTTP `POST` call to `/bob/bookmarks` will **add**, or **append**, the enclosed body to the collection (database, filesystem, whatever) designated by the `/bob/bookmarks` URI. It can be a little confusing, though. An HTTP `POST` to `/bob/bookmarks/1` , on the other hand, might be treated in the same way as an HTTP `PUT` call; the service could take the enclosed body and use it to **replace** the resource designated by the URI.

Of course, sometimes things don't go to plan. Perhaps the browser timed out, or the service has timed out, or the service encounters an error. We've all gotten the annoying 404 ("Page not found") error when attempting to visit a page that doesn't exist or couldn't be routed to correctly. That 404 is a **status code**. It conveys information about the state of the operation. There are **many** status codes divided along ranges for different purposes. When you make a request to a webpage in the browser, it is an HTTP `GET` call, and - if the page shows up - it will have returned a 200 status code. 200 means `OK` ; you may not know it, but it's there.

- Status codes in the **100x range** (from 100-199) are **informational**, and describe the processing for the request.
- Status codes in the **200x range** (from 200-299) indicate the action requested by the client was received, understood, accepted and processed successfully
- Status codes in the **300x range** (from 300-399) indicate that the client must take additional action to complete the request, such as following a **redirect**
- Status codes in the **400x range** (from 400-499) is intended for cases in which the client seems to have erred and must correct the request before continuing. The aforementioned 404 is an example of this.
- Status codes in the **500x range** (from 500-599) is intended for cases where the server failed to fulfill an apparently valid request.

Building a REST service

The first cut of a bookmark REST service should at least support reading from, and adding to, an account's bookmarks, as well as reading individual ones. Below is the first cut at our REST service:

`rest/src/main/java/bookmarks/Application.java`

```
link:./rest/src/main/java/bookmarks/Application.java[]
```

`BookmarkRestController` is a simple Spring MVC `@RestController` -annotated component. `@RestController` exposes the annotated bean's methods as HTTP endpoints using metadata furnished by the `@RequestMapping` annotation on each method. A method will be put into service if an incoming HTTP request matches the qualifications stipulated by the `@RequestMapping` annotation on the method.

`@RestController` , when it sits at the type level, provides defaults for all the methods in the type. Each individual method may override most of the type-level annotation. Some things are **contextual**. For example, the `BookmarkRestController` handles **all** requests that start with a username (like `bob`) followed by `/bookmarks` . Any methods in the type that further qualify the URI, like `readBookmark` , are **added** to the root request mapping. Thus, `readBookmark` is, in effect, mapped to `/{userId}/bookmarks/{bookmarkId}` . Methods that don't specify a path just inherit the path mapped at the type level. The `add` method responds to the URI specified at the type level, but it **only** responds to HTTP requests with the verb

The `{userId}` and `{bookmarkId}` tokens in the path are **path variables**. They're globs, or wildcards. Spring MVC will extract those portions of the URI, and make them available as arguments of the same name that are passed to the controller method and annotated with `@PathVariable` . For an HTTP `GET` request to the URI `/bob/bookmarks/4234` , the `@PathVariable String userId` argument will be `"bob"` , and the `@PathVariable Long bookmarkId` will be coerced to a `long` value of `4234` .

These controller methods return simple POJOs - `Collection<Bookmark>` , and `Bookmark` , etc., in all but the `add` case. When an HTTP request comes in that specifies an `Accept` header, Spring MVC loops through the configured `HttpMessageConverter` until it finds one that can convert from the POJO domain model types into the content-type specified in the `Accept` header, if so configured. Spring Boot automatically wires up an `HttpMessageConverter` that can convert generic `Object` s to JSON, absent any more specific converter. `HttpMessageConverter` s work in both directions: incoming requests bodies are converted to Java objects, and Java objects are converted into HTTP response bodies.

Use `curl` (or your browser) to see the JSON response from `http://localhost:8080/jhoeller/bookmarks` .

The `add` method specifies a parameter of type `Bookmark` - a POJO. Spring MVC will convert the incoming HTTP request (containing, perhaps, valid JSON) to a POJO using the appropriate `HttpMessageConverter` .

The `add` method accepts incoming HTTP requests, saves them and then sends back a `ResponseEntity<T>` . `ResponseEntity` is a wrapper for a response and, optionally, HTTP headers and a status code. The `add` method sends back a `ResponseEntity` with a status code of 201 (`CREATED`) and a header (`Location`) that the client can consult to learn how the newly created record is referencable. It's a bit like extracting the just generated primary key after saving a record in the database.

There are paths not taken. By default Spring Boot sets up a pretty generous collection of `HttpMessageConverter` implementations suitable for common use, but it's easy to add support for other, perhaps more compact, network-efficient formats (like the Google Protocol Buffers implementation in Spring 4.1) using the usual Spring MVC configuration.

Spring MVC natively supports file uploads via controller arguments of type `MultipartFile multipartFile` .

Spring MVC makes it easy to write service-oriented code whose shape is untainted by `HttpServlet` APIs. This code can be easily unit tested, extended through Spring AOP. We'll look at how to unit test these Spring MVC components in the next section.

Using HTTP to signal Errors

All the methods in the REST service call `validateUser` which in turn verifies that the user in question exists and - if it doesn't - throws a `UserNotFoundException` . The definition of this exception is shown. It's annotated with a Spring MVC `@ResponseStatus(HttpStatus.NOT_FOUND)` which tells Spring MVC to send back an HTTP status code (404) whenever this exception is triggered. This is a *really* nice arrangement: you can think in terms of your business domain in your code *and* you get smart handling that maps nicely to how HTTP signals errors to the client, using status codes. Spring MVC

provides a *lot* of different places to cleanly layer in generic, application-global and controller-local error handling logic.

Testing a REST Service

Spring MVC provides great support for unit testing HTTP endpoints. It provides a very nice middle ground between unit-testing and integration-testing in that it lets you stand up the entire Spring MVC `DispatcherServlet` -based machinery - including validators, `HttpMessageConverter` s, and more - and then run tests against them *without* actually starting up a *real* HTTP service: the best of both worlds! It's an integration-test in that the logic you care about is actually being exercised, but it's a unit-test in that you're not actually waiting for a web server to initialize and start servicing requests.

Here's the unit-test for the `ReservationRestController` shown below. This should look familiar to anybody who has written a JUnit unit test before. At the top, we use the `@SpringApplicationConfiguration(classes = Application.class)` annotation (from Spring Boot) to tell the `SpringJUnit4ClassRunner` where it should get information about the Spring application under test. The `@WebAppConfiguration` annotation tells JUnit that this is a unit test for Spring MVC web components and should thus run under a `WebApplicationContext` variety, not a standard `ApplicationContext` implementation.

```
rest/src/test/java/bookmarks/BookmarkRestControllerTest.java


  link:./rest/src/test/java/bookmarks/BookmarkRestControllerTest.java[]
```

The first thing to look at is this `@Before` -annotated `setup` method. The first thing the `setup` method does is instantiate a `MockMvc` which requires a reference to the application's `WebApplicationContext` . The `MockMvc` is the center piece: all tests will invariably go through the `MockMvc` type to mock HTTP requests against the service. And… that's it! Look at any of the various tests. We can use the static imports on `org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*` to chain together HTTP requests and verify the responses.

All tests specify a `application/json` `content-type` and expect responses of that `content-type` , as well. The tests use the `MockMvcResultMatchers#jsonPath` method to validate the structure and contents of the JSON responses. This, in turn, uses the Jayway JSON Path API to run X-Path-style traversals on JSON structures, as we do in various places in the unit tests.

Building a HATEOAS REST Service

The first cut of the API works very well. If this service were well documented, it would be workable for REST clients in many different languages. It is a clean API, in that it takes advantage of some of the primitives that HTTP provides, in a well-understood way. One measure of an API is by its compliance with the uniform interface principle. HTTP REST APIs like the one we have so far stack up pretty well. Each message includes enough information to describe how to process the message. For example, a client might decide which parser to invoke based on the `Content-Type` header in the request message. The state in the system is mapped into uniquely identifying resource URIs. State is addressable. Mutations in state are done through known HTTP verbs (`POST` , `GET` , `DELETE` , `PUT` , etc.). Thus, when a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.

But, we can do better. The services as they stand are adequate to the task but lack.. **staying power**. As Wikipedia says: Clients must know the API a priori. Changes in the API break clients and they **break** the documentation about the service. Hypermedia as the engine of application state (a.k.a. **HATEOAS**) is one more constraint that addresses and removes this coupling. Clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g., by hyperlinks within hypertext). Except for simple fixed entry points to the application, a client does not assume that any particular action is available for any particular resources beyond those described in representations previously received from the server.

Let's look at a revised cut of this API (shown below), this time embracing HATEOAS with Spring HATEOAS. It is a slight simplification to say that Spring HATEOAS makes it easy to provide links - metadata about payloads being returned to the

client - but that is how we will approach it. Fundamentally, all we will do is **wrap** our response payloads using Spring HATEOAS' `ResourceSupport` type. `ResourceSupport` accumulates `Link` objects which in turn describe useful, related resources. For example, a resource describing an account in an e-commerce solution could have a link to the resource for that account's orders, a link to that account's current shopping cart, and a link that can be used to retrieve that resources state again.

```
hateoas/src/main/java/bookmarks/Application.java
```

```
link:./hateoas/src/main/java/bookmarks/Application.java[]
```

These `Link` s, by the way, are of the same sort as the `<link/>` element so often used in HTML pages to import CSS stylesheets. They have an `href` attribute and a `rel` attribute. The `href` attribute points to where the CSS stylesheet lives, and the `rel` tells the client (the browser) **why** this resource is important (because it's a stylesheet to be used in rendering the page). It's not hard to translate a `link` element into JSON, either.

The `BookmarkResource` type **wraps** a `Bookmark` and provides a nice, centralized place to keep link-building logic. At a minimum, a resource should provide a link to itself (usually a link whose `rel` value is `self`). We could simply write out `http://127.0.0.1:8080/\{userId\}/bookmarks/\{id\}` (substituting the path variables for appropriate values), but this will fail as soon as we move to a different host, port, and context root. We could use the `ServletUriComponentsBuilder` to simplify some of this work. But why should we? After all, Spring MVC *already knows* about this URI. It's **in** the `@RequestMapping` information on every controller method! Spring HATEOAS provides the convenient static `ControllerLinkBuilder.linkTo` and `ControllerLinkBuilder.methodOn` methods to extract the URI from the controller metadata itself - a marked improvement and in keeping with the DRY (do not repeat yourself) principle. The example shows how to build a `Link` object directly, specifying an arbitrary value for `href` (in this case, the URI for the bookmark itself), how to build a link based on the `@RequestMapping` metadata on a Spring MVC controller, and how to build a link based on the `@RequestMapping` metadata on a specific Spring MVC controller method.

With this in place, the only remaining changes substitute `Bookmark` types for `BookmarkResource` types.

Improved Error Handling with `VndErrors`

HATEOAS gives clients improved metadata about the service itself. We can improve the situation for our error handling, as well. HTTP status codes tell the client - broadly - that something went wrong. HTTP status codes from 400-499, for example, tell the client that the client did something wrong. HTTP status codes from 500-599 tell the client that the server did something wrong. If you know your status codes, then this can be a *start* in understanding how to work with the API. But, we can do better. After all, before a REST client is up and running, *somebody* needs to develop it, and useful error messages can be invaluable in understanding an API. Errors that can be handled in a consistent way are even better!

`VndError` is a popular, de-facto standard mime-type that describes the encoding of errors to the client of a REST service. It works well with errors in the 40x and 50x range of errors. Spring HATEOAS provides `VndError` types that you can use to report errors back to your client.

Our revised service introduces a new class, `BookmarkControllerAdvice` , that uses Spring MVC's `@ControllerAdvice` annotation. `@ControllerAdvice` are a useful way to extricate the configuration of common concerns - like error handling - into a separate place, away from any individual Spring MVC controller. Spring MVC, for example, defines the `@ExceptionHandler` method that ties a specific handler method to any incident of an `Exception` or a HTTP status code. Here, we're telling Spring MVC that any code that throws a `UserNotFoundException` , as before, should eventually be handled by the `userNotFoundExceptionHandler` method. This method simply wraps the propagated `Exception` in a `VndErrors` and returns it to the client. In this example, we've removed the `@ResponseStatus` annotation from the exception itself and centralized it in the `@ControllerAdvice` type. `@ControllerAdvice` types are a convenient way to centralize all sorts of logic, and - unlike annotating exception types - can be used even for exception types to which you don't have the source code.

Securing a REST Service

Thus far we've proceeded from the assumption that all clients are trustworthy, and that they should have unmitigated access to all the data. This is rarely actually the case. An open REST API is an insecure one. It's not hard to fix that, though. Spring Security provides primitives for securing application access. Fundamentally, Spring Security needs to have some idea of your application's users and their privileges. These privileges, or **authorities**, answer the question: what may an application user see, or do?

At the heart of Spring Security is the `UserDetailsService` interface, which has **one job**: given a username, produce a `UserDetails` implementation, `UserDetails` implementations must be able to answer questions about an account's validity, its password, its username, and its authorities (represented by instances of type `org.springframework.security.core.GrantedAuthority`).

```
link:./snippets/user-details-service.txt[]
```

Spring Security provides many implementations of this contract that adapt existing identity providers, like Active Directory, LDAP, `pam`, CAAS, etc. Spring Social even provides a nice integration that delegates to different OAuth-based services like Facebook, Twitter, etc., for authentication.

Our example already has a notion of an `Account`, so we can simply adapt that by providing our own `UserDetailsService` implementation, as shown below in the `WebSecurityConfiguration` `@Configuration`-class. Spring Security will ask this `UserDetailsService` if it has any questions about an authentication request.

Client Authentication and Authorization on the Open Web with Spring Security OAuth

We can authenticate client requests in a myriad of ways. Clients could send, for example, an HTTP-basic username and password on each request. They could transmit an x509 certificate on each request. There are indeed numerous approaches that could be used here, with numerous tradeoffs.

Our API is meant to be consumed over the open-web. It's meant to be used by all manner of HTML5 and native mobile and desktop clients that we intend to build. We shall use diverse clients with diverse security capabilities, and any solution we pick should be able to accommodate that. We should also decouple the user's username and password from the application's session. After all, if I reset my Twitter password, I don't want to be forced to re-authenticate every client signed in. On the other hand, if someone **does** hijack one of our clients (perhaps a user has lost a phone), we don't want the party that stole the device to be able to lock our users out of their accounts.

OAuth provides a clean way to handle these concerns. You've no doubt used OAuth already. One common case is when installing a Facebook plugin or game. Typically the flow looks like this:

- user finds a game or piece of functionality on the web that requires access to a user's Facebook data in order to function. One common example is "Sign in With Facebook"-style scenarios.
- a user clicks "install," or "add," and is then redirected to a trusted domain (for example: Facebook.com) where the user is prompted to grant certain permissions (like "Post to wall," or "Read Basic information")
- the user confirms these permissions and is subsequently redirected back to the source application where the source application now has an access token. It will use this access token to make requests to Facebook on your behalf.

In this example, any old client can talk to Facebook and the client has, at the end of the process, an access token. This access token is transmitted via all subsequent REST requests, sort of like an HTTP cookie. The username and password need not be retransmitted and the client may cache the access token for a finite or infinite period. Users of the client need not re-authenticate every time they open an application, for example. Even better: access tokens are specific to each client. They may be used to signal that one client needs more permissions than others. In the flow above, requests always ended up at Facebook.com where - if the user is not already signed into Facebook, he or she will be prompted to login and then assign permissions to the client. This has the benefit of ensuring that any sensitive information, like a username and password, is never entered in the wild in untrusted applications that might maliciously try to capture that username and password. Our application, will not be available to any old client. We can be sure that any client we deploy is

friendly, as it will be one of **our** clients. OAuth supports a simpler flow whereby a user authenticates (typically by sending a username and password) from the client and the service returns an OAuth access token directly, sidestepping the need for a redirect to a trusted domain. This is the approach we will take: the result will be that our clients will have an access token that's decoupled from the user's username and password, and the access token can be used to confer different levels of security on different clients.

Setting up OAuth security for our application is easy. The `OAuth2Configuration` configuration class describes one client (here, one for a hypothetical Android client) that needs the `ROLE_USER` and the `write` scope. Spring Security OAuth will read this information from the `AuthenticationManager` that is ultimately configured using our custom `UserDetailsService` implementation.

OAuth is very flexible. You could, for example, deploy an authorization server that's shared by many REST APIs. In this case, our OAuth implementation lives adjacent to our bookmarks REST API. They are one and the same. This is why we've used both `@EnableResourceServer` and `@EnableAuthorizationServer` in the same configuration class.

You Said Something about the Open Web?

We expect that some of the clients to our service will be HTML5-based. They will want to talk to our REST API from different domains, and in most browsers this runs afoul of cross-site-scripting security measures. We can explicitly enable XSS for well-known clients by exposing CORS (cross-origin request scripting) headers. These headers, when present in the service responses, signal to the browser that requests of the origin, shape and configuration described in the headers **are** permitted, even across domains. Our API is decorated with a simple `javax.servlet.Filter` that adds these headers on every request. In the example, we're delegating to a property - `tagit.origin` - if provided or a default of `http://localhost:9000` where we might have, for example, a JavaScript client making requests to the service. We could just as easily read this information from a datastore which we can change without recompiling the code. We've only specified a single origin here, but we could just as easily specified numerous clients.

Using HTTPS (SSL/TLS) to prevent Man-in-the-Middle Attacks

Spring Boot provides an embedded web server (Apache Tomcat, by default) that can be configured programmatically to do anything that the standalone Apache Tomcat webserver can do. In the past, it required several tedious steps to configure HTTPS (SSL/TLS). Now, Spring Boot makes it super simple to do that declaratively. First, create the following:

```
security/src/main/resources/application-https.properties


  link:security/src/main/resources/application-https.properties[]
```

Note	This property file is only activated when the app is run when **SPRING_PROFILES_ACTIVE** configured with profile **https**.

HTTPS requires a signed certificate certificate and a certificate password which we provide using property values. To do so, we can use the JDK's **keytool** like this:

```
$ keytool -genkey -alias bookmarks -keyalg RSA -keystore src/main/resources/tomcat.keystore
Enter keystore password: password
Re-enter new password: password
What is your first and last name?
  [Unknown]:  Josh Long
What is the name of your organizational unit?
  [Unknown]:  Spring Team
What is the name of your organization?
  [Unknown]:  Pivotal
What is the name of your City or Locality?
  [Unknown]:  IoT
What is the name of your State or Province?
  [Unknown]:  Earth
What is the two-letter country code for this unit?
```

```
    [Unknown]:  US
Is CN=Josh Long, OU=Spring Team, O=Pivotal, L=IoT, ST=Earth, C=US correct?
    [no]:  yes

Enter key password for <learningspringboot>
        (RETURN if same as keystore password): <RETURN>
```

Warning	keystore files MUST be on the file system for embedded Tomcat to read them. They can NOT be embedded in JAR files. (And frankly, that type of security item should NOT be part of your deliverable anyway.)

This will create a keystore underneath `src/main/resources`. Now if you run the app, it will use the keystore to run the embedded Tomcat servlet container using SSL/TLS.

As stated earlier, this mode is only available using the Spring profile named `https`. This means that you can run and test the application **without** the profile applied or switch it on just as easily.

Putting it All Together

```
security/src/main/java/bookmarks/Application.java

  link:./security/src/main/java/bookmarks/Application.java[]
```

With Spring Security OAuth in place, we can rework our API a little bit. It makes little sense to have `userId`s strewn throughout the URIs for our API. After all, the user context is *implied* in the secure access of our endpoints. Instead of requiring a `userId` in the path, the secure Spring MVC handler methods expect a `javax.security.Principal` that Spring Security injects on our behalf. This principal offers a `javax.security.Principal#getName` method that can be used in place of the `userId`.

Conclusion

REST is the most natural way for ubiquitous, disparate clients to communicate. It works because HTTP works. Once you understand how REST fits into an application, it's not hard to envision an architecture with numerous, singly focused APIs all exposed over REST, load-balanced as any other HTTP service would be. It is not surprising, then, that REST (often, but not necessarily) forms a critical foundation for *microservices*, which we'll look at in more depth in an upcoming tutorial.