Pivotal

Creating a Complete RESTful Application

Using Spring MVC to create RESTful Web Services

Objectives

After completing this lesson, you should be able to

- Create controllers to support the REST endpoints for various verbs
- Process arguments from the request and from the URL
- Utilize message converters to return data as JSON or XML
- Utilize RestTemplate to invoke RESTful services

Pivotal

Agenda

- **REST Introduction**
- Spring MVC support for RESTful applications
- RESTful Clients: **`RestTemplate`**
- Lab
- Advanced Topics

Pivotal

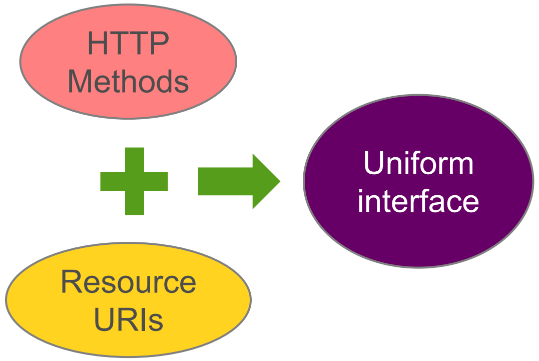
REST Introduction

- Why REST?
  - Programmatic clients leveraging HTTP
    - Mobile applications, Microservices
    - Browsers: SPA, AJAX

- REST is an *architectural style* that describes best practices to expose web services over HTTP
  - *REpresentational State Transfer*, term by Roy Fielding
  - HTTP as *application* protocol, not just transport
  - Emphasizes scalability
  - *Not* a framework or specification

REST Principles (1)

- Expose *resources* through URIs
  - Model nouns, not verbs
  - http://springbank.io/banking/accounts/123456789
- Resources support limited set of operations
  - GET, PUT, POST, DELETE in case of HTTP
  - All have well-defined semantics
- Example: update an order
  - PUT to /orders/123
  - don't POST to /order/edit?id=123



Pivotal

REST Principles (2)

- Clients can request particular representation
  - Resources can support multiple representations
  - HTML, XML, JSON, …
- Representations can link to other resources
  - Allows for extensions and discovery, like with web sites
- Hypermedia As The Engine of Application State
  - HATEOAS: Probably the world's worst acronym!
  - RESTful response contain the links you need – just like HTML pages do

More on HATEOAS in *Advanced Section*

REST Principles (3)

- Stateless architecture
  - No HttpSession usage
  - GETs can be cached on URL
  - Requires clients to keep track of state
  - Part of what makes it scalable
  - Looser coupling. Between client and server

- HTTP headers and status codes communicate result to clients
  - All well-defined in HTTP Specification

Why REST?

- Benefits of REST
  - Every platform/language supports HTTP
    - Unlike, for example, SOAP + WS-* specs
  - Easy to support many different clients
    - Scripts, Browsers, Applications
  - Scalability
  - Support for redirect, caching, different representations, resource identification, …
  - Support for multiple formats
    - JSON and Atom are popular choices

REST and Java: JAX-RS

JAX-RS

- JAX-RS is a Java EE 6 standard for building RESTful applications
  - Focuses on programmatic clients, not browsers
- Various implementations
  - Jersey (RI), RESTEasy, Restlet, CXF
  - All implementations provide Spring support
- Good option for full REST support using a standard

Pivotal

REST and Java: Spring-MVC

- Spring-MVC provides REST support as well
  - Using familiar and consistent programming model
  - Spring MVC does not implement JAX-RS
- Single web-application for everything
  - Traditional web-site: HTML, browsers
  - Programmatic client support (RESTful web applications, HTTP-based web services)
- `RestTemplate` for building programmatic clients in Java

Agenda

- REST Introduction
- **Spring MVC support for RESTful applications**
- RESTful Clients: `RestTemplate`
- Lab
- Advanced Topics

Pivotal

Spring-MVC and REST

- Requirements for using Spring MVC to support REST
  - Map requests based on HTTP method
  - Define response status
  - Message Converters
  - Access request and response body data
  - Build valid Location URIs *

** For HTTP POST responses*

Agenda

- REST Introduction
- **Spring MVC support for RESTful applications**
  - **HTTP GET**
  - HTTP PUT
  - HTTP POST
  - HTTP DELETE
- RESTful Clients: **`RestTemplate`**
- Lab
- Advanced Topics

Pivotal

HTTP GET: Fetch a Resource

- **Requirement**
  - Respond *only* to GET requests
  - Return requested data in the HTTP Response
  - Determine requested response format

```
GET /store/orders/123
Host: shop.spring.io
Accept: application/json,
...
...
```

```
HTTP/1.1 200 OK
Date: …
Content-Length: 756
Content-Type:
application/json

{
   "id": 123,
   "total": 200.00,
   "items": [ … ]
}
```

Request Mapping Based on HTTP Method

- Map HTTP requests based on method
  - Allows same URL to be mapped to multiple Java methods
  - **`@GetMapping`**, **`@PostMapping`**, **`@PutMapping`**, **`@PatchMapping`**, **`@DeleteMapping`**

- Examples:

```
// Get all orders (for current user typically)
@GetMapping(path="/orders")

// Create a new order
@PostMapping(path="/orders")
```

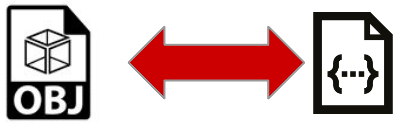
`@RequestMapping` Annotation

- `@GetMapping` is a "*composed*" annotation
  - `@RequestMapping(path="/accounts",`
    `method=RequestMethod.GET)`
  - Equivalent to `@GetMapping("/accounts")`

- `@RequestMethod` enumerators
  - `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, `OPTIONS`, `TRACE`

For HEAD, OPTIONS, TRACE *must* use `@RequestMethod`

Recall: Message Converters



- Message Converters
  - Implement **`HttpMessageConverter`**
  - Setup automatically by Spring Boot
- HTTP **`GET`** method returns a Java object
  - Converter generates data in HTTP response body
    - Typically XML or JSON
    - No need to convert objects manually
    - **`Accept`** header defines response format (and converter to use)
- Also used for HTTP **`POST`** and **`PUT`**
  - Convert *incoming* request body to a Java object

Recall: `@ResponseBody`

- Use converters for response data by annotating return data with **`@ResponseBody`**
- Converter handles rendering a response
  - *No ViewResolver and View involved any more!*

```
@GetMapping("/orders/{id}")
public @ResponseBody Order getOrder(@PathVariable("id") long id) { ... }
```

NOTE: If you forget **`@ResponseBody`**, Spring MVC defaults to finding a View (and fails)

Recall: `@RestController` Simplification

```
@Controller
public class OrderController {

  @GetMapping("/orders/{id}")
  public @ResponseBody Order getOrder(@PathVariable long id) {
    return orderService.findOrderById(id);
  }
}
```

```
@RestController
public class OrderController {

  @GetMapping("/orders/{id}")
  public Order getOrder(@PathVariable long id) {
    return orderService.findOrderById(id);
  }
}
```

No need for `@ResponseBody` on `GET` methods

Retrieving a Representation: GET

Complete Example

```
GET /store/orders/123
Host: shop.spring.io
Accept: application/json
...
```

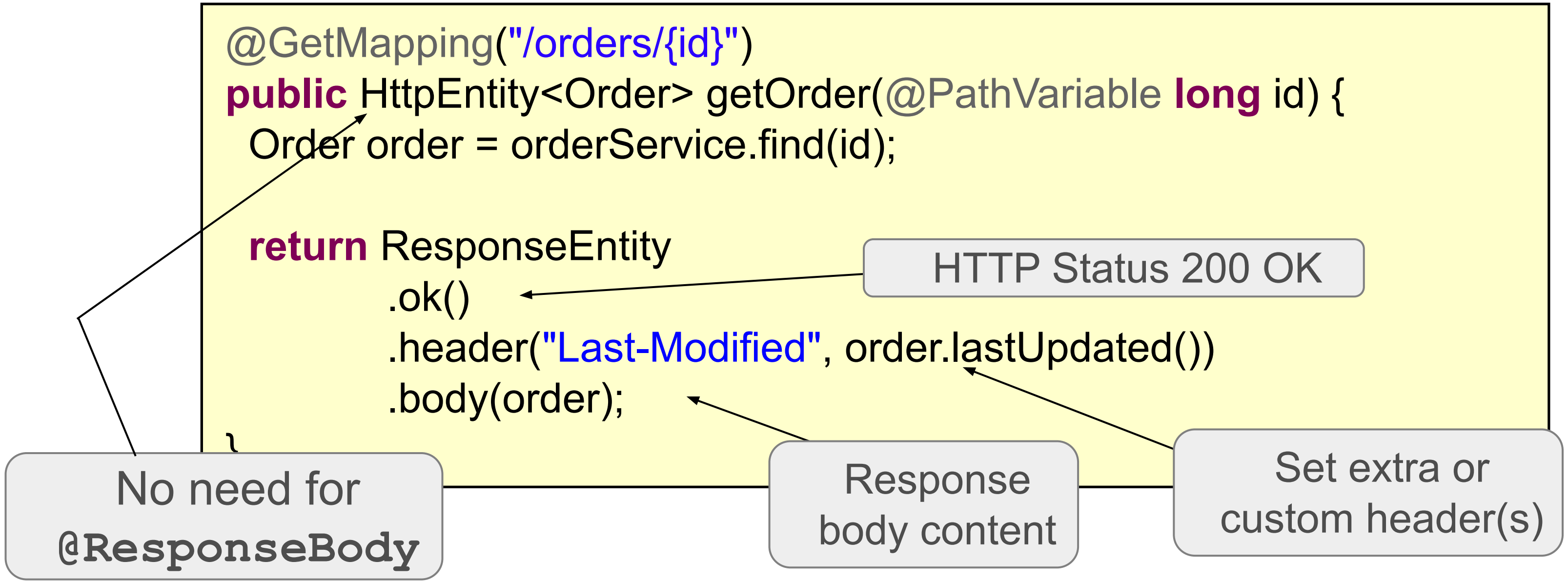
```
HTTP/1.1 200 OK
Date: ...
Content-Length: 1456
Content-Type:
application/json


{
    "id": 123,
    "total": 200.00,
    "items": [ … ]
}
```

```
@GetMapping("/orders/{id}")
public Order getOrder(@PathVariable("id") long id) {
  return orderService.findOrderById(id);
}
```

```
@RequestMapping(path="/orders/{id}", method=RequestMethod.GET)
```

Recall: Setting Response Data

- Can use **`ResponseEntity`** to generate a Response
  – Avoids use of **`HttpServletResponse`** (easier to test)



```
@GetMapping("/orders/{id}")
public HttpEntity<Order> getOrder(@PathVariable long id) {
  Order order = orderService.find(id);

  return ResponseEntity
             .ok()
             .header("Last-Modified", order.lastUpdated())
             .body(order);
}
```

Pivotal

Agenda

- REST Introduction
- **Spring MVC support for RESTful applications**
  - HTTP GET
  - **HTTP PUT**
  - HTTP POST
  - HTTP DELETE
- RESTful Clients: `RestTemplate`
- Lab
- Advanced Topics

Pivotal

HTTP PUT: Update a Resource

- **Requirement**
  - Respond *only* to PUT requests
  - Access data in the HTTP Request
  - Return empty response, status 204

```
PUT /store/orders/123/items/abc
Host: www.mybank.com
Content-Type: application/json

{
   "id": abc,
   "cost": 35.00,
   "product": SKU1234, ...
}
```

```
HTTP/1.1 204 No Content
Date: …
Content-Length: 0
...
```

Successful update – *nothing* to return

Pivotal

HTTP Status Code Support

- Web apps just use a handful of status codes
  - Success: 200 OK
  - Redirect: 30x for Redirects
  - Client Error (Invalid URL): 404 Not Found
  - Server Error: 500 (such as unhandled Exceptions)
- RESTful applications use many additional codes
  - Created Successfully: 201
  - Client Error: 400
  - HTTP method not supported: 405
  - Cannot generate requested response body format: 406
  - Request body not supported: 415

For a full list: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

`@ResponseStatus`

- To return a status code *other* than 200
  - Use **`HttpStatus`** enumerator
- **Note: `@ResponseStatus`** on ***void*** methods
  - Turns *off* view-processing subsystem
  - Method returns a response with empty body (*no-content*)

```
@PutMapping("/orders/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrder(...) {
  // Update order
}
```

Can also set error response codes – see Advanced section

Pivotal

`@RequestBody`

- Use message converters for incoming request data
  - Correct converter chosen automatically
    - Based on **`Content-Type`** of request
  - **`updatedOrder`** could be mapped from XML (with JAXB2) or from JSON (with GSON or Jackson)
    - Annotate Order class (if need be) for JAXB/Jackson to work

```
@PutMapping("/orders/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)  // 204
public void updateOrder(@RequestBody Order updatedOrder,
                    @PathVariable("id") long id) {
  // process updated order data and return empty response
  orderManager.updateOrder(id, updatedOrder);
}
```

Pivotal

Updating Existing Resource: PUT

Complete Example

```
PUT /store/orders/123/items/abc
Host: shop.spring.io
Content-Type: application/json

{
    "id": abc,
    "cost": 35.00,
    "product": SKU1234, ...
}
```

```
HTTP/1.1 204 No Content
Date: …
Content-Length: 0
```

```
@PutMapping("/orders/{orderId}/items/{itemId}")
@ResponseStatus(HttpStatus.NO_CONTENT)  // 204
public void updateItem(@PathVariable long orderId,
                       @PathVariable String itemId,
                       @RequestBody Item item) {

  orderService.findOrderById(orderId).updateItem(itemId, item);
}
```

```
@RequestMapping(path="/orders/...", method=RequestMethod.PUT)
```

Pivotal

Agenda

- REST Introduction
- **Spring MVC support for RESTful applications**
  - HTTP GET
  - HTTP PUT
  - **HTTP POST**
  - HTTP DELETE
- RESTful Clients: **`RestTemplate`**
- Lab
- Advanced Topics

Pivotal

HTTP POST: Create a new Resource

- **Requirement**
  - Respond *only* to POST requests
  - Access data in the HTTP Request
  - Return “created”, status 201
  - Generate *Location* header for newly created resource

```
POST /store/orders/123/items
Host: shop.spring.io
Content-Type: application/json

{
   "cost": 50.00,
   "product": SKU9988, ...
}
```

```
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://shop.spring.io/

store/orders/123/items/abc
```

Pivotal

Creating a new Resource: `POST` (2)

- We can already implement most of this requirement
  - But how do we return the new Item location?

```
@PostMapping(path="/orders/{id}/items")
public ??? createItem (@PathVariable("id") long id, @RequestBody Item newItem) {
  // Add the new item to the order
  orderService.findOrderById(id).addItem(newItem);

  return ???
}
```

Pivotal

Building URIs

- HTTP POST is required to return the location of the newly created resource in the response header
- How to create a URI?
  - **`UriComponentsBuilder`**
    - Allows explicit creation of URI
    - *But* requires hard-coded URLs
  - **`ServletUriComponentsBuilder`**
    - Subclass of **`UriComponentsBuilder`**
    - Provides access to the URL that invoked the current controller method

ServletUriComponentsBuilder

- Use this in our Controller method

```
// Must be in a Controller method
//   Example:  POST /orders/12345/items

URI location = ServletUriComponentsBuilder
     .fromCurrentRequestUri()
     .path("/{itemId}")
     .buildAndExpand("item A")
     .toUri();

return ResponseEntity.created(location).build();

  // .../orders/12345/items/item%20A
```

Framework puts request URL in current thread – which builder can access

Note: leading **/**

Note: **space**

Space converted to **%20**

Creating a new Resource: `POST`

Complete Example

```
@PostMapping("/orders/{id}/items")
public ResponseEntity<Void> createItem
    (@PathVariable long id, @RequestBody Item newItem) {
  // Add the new item to the order
  orderService.findOrderById(id).addItem(newItem);

  // Build the location URI of the new item
  URI location = ServletUriComponentsBuilder
      .fromCurrentRequestUri()
      .path("/{itemId}")
      .buildAndExpand(newItem.getId())
      .toUri();

  // Explicitly create a 201 Created response
  return ResponseEntity.created(location).build();
}
```

`@ResponseStatus` not needed

Assume this call also set an item-id

```
@RequestMapping(path="/orders/...", method=RequestMethod.POST)
```

Pivotal

Agenda

- REST Introduction
- **Spring MVC support for RESTful applications**
  - HTTP GET
  - HTTP PUT
  - HTTP POST
  - **HTTP DELETE**
- RESTful Clients: **`RestTemplate`**
- Lab
- Advanced Topics

Pivotal

HTTP `DELETE`: Delete existing Resource

- **Requirement**
  - Respond *only* to DELETE requests
  - Return empty response, status 204

```
DELETE
/store/orders/123/items/abc
Host: shop.spring.io
Content-Length: 0
...
```

```
HTTP/1.1 204 No
Content
Date: …
Content-Length: 0
```

Pivotal

Deleting a Resource: DELETE

Complete Example

```
DELETE
/store/orders/123/items/abc
Host: shop.spring.io
...
```

```
HTTP/1.1 204 No Content
Date: …
Content-Length: 0
```

```
@DeleteMapping("/orders/{orderId}/items/{itemId}")
@ResponseStatus(HttpStatus.NO_CONTENT)  // 204
public void deleteItem(@PathVariable long orderId,
                       @PathVariable String itemId) {
  orderService.findOrderById(orderId).deleteItem(itemId);
}
```

```
@RequestMapping(path="/orders/...", method=RequestMethod.DELETE)
```

Putting it all Together

- Many new concepts
  - HTTP Message Converters
  - `@RequestBody`, `@ResponseBody`
  - `@RestController`
  - `@ResponseStatus`
  - `HttpEntity`, `ResponseEntity`
  - `ServletUriComponentsBuilder`

Agenda

- REST Introduction
- Spring MVC support for RESTful applications
- **RESTful Clients: `RestTemplate`**
- Lab
- Advanced Topics

Pivotal

RestTemplate

- Provides access to RESTful services
  - Supports all the HTTP methods

HTTP Method	RestTemplate Method
DELETE	delete(String url, Object… urlVariables)
GET	getForObject(String url, Class<T> responseType, Object… urlVariables)
HEAD	headForHeaders(String url, Object… urlVariables)
OPTIONS	optionsForAllow(String url, Object… urlVariables)
POST	postForLocation(String url, Object request, Object… urlVariables)
	postForObject(String url, Object request, Class<T> responseType, Object… uriVariables)
PUT	put(String url, Object request, Object… urlVariables)
PATCH	patchForObject(String url, Object request, Class<T> responseType, Object… uriVariables)

Defining a `RestTemplate`

- Just call constructor in your code
  - Sets up default *HttpMessageConverters* internally
    - Same as on the server, depending on classpath

```
RestTemplate template = new RestTemplate();
```

`RestTemplate` Usage Examples

```
RestTemplate template = new RestTemplate();
String uri = "http://example.com/store/orders/{id}/items";

// GET all order items for an existing order with ID 1:
OrderItem[] items =
        template.getForObject(uri, OrderItem[].class, "1");

// POST to create a new item
OrderItem item = // create item object
URI itemLocation = template.postForLocation(uri, item, "1");

// PUT to update the item
item.setAmount(2);
template.put(itemLocation, item);

// DELETE to remove that item again
template.delete(itemLocation);
```

{id} = 1

{id} = 1

Pivotal

Using `ResponseEntity`

- Access response headers and body

```
String uri = "http://example.com/store/orders/{id}";

ResponseEntity<Order> response =
    restTemplate.getForEntity(uri, Order.class, "1");

assert(response.getStatusCode().equals(HttpStatus.OK));

long modified = response.getHeaders().getLastModified();

Order order = response.getBody();
```

{id} = 1

Access HTTP Response yourself

Pivotal

`RequestEntity` *and* `ResponseEntity`

- Setup your own request as well

```
// POST with HTTP BASIC authentication
RequestEntity<OrderItem> request = RequestEntity
    .post(new URI(itemUrl))
    .getHeaders().add(HttpHeaders.AUTHORIZATION,
            "Basic " + getBase64EncodedLoginData())
    .accept(MediaType.APPLICATION_JSON)
    .body(newItem);

ResponseEntity<Void> response =
        restTemplate.exchange(request, Void.class);

assert(response.getStatusCode().equals(HttpStatus.CREATED));
```

Pivotal

Summary

REST

- REST is an architectural style that can be applied to HTTP-based applications
  - Useful for supporting diverse clients and building highly scalable systems
- Spring-MVC adds REST support using a familiar programming model (but *without* Views)
  - **`@ResponseStatus`**, **`@RequestBody`**, **`@ResponseBody`**
  - **`HttpEntity`**, **`ResponseEntity`**
  - **`ServletUriComponentsBuilder`**
  - HTTP Message Converters
- Clients use *RestTemplate* to access RESTful servers

Lab: Creating a full RESTful service

Lab project:
38-rest-ws

Anticipated Lab time:
50 Minutes

Optional Topics: Spring REST additional details, HATEOS

Pivotal

Agenda

- REST Introduction
- Spring MVC support for RESTful applications
- RESTful Clients: `RestTemplate`
- Lab
- **Advanced Topics**
  - **More on Spring REST**
  - Spring HATEOAS

Pivotal

`@ResponseStatus` & Exceptions

- Can also annotate exception classes with this
  - Given status code used when an unhandled exception is thrown from any controller method

```
@ResponseStatus(HttpStatus.NOT_FOUND)  // 404
public class OrderNotFoundException extends RuntimeException {
   …
}
```

```
@GetMapping(value="/orders/{id}")
public Order showOrder(@PathVariable("id") long id, Model model) {
   Order order = orderService.findOrderById(id);
   if (order == null) throw new OrderNotFoundException(id);
   return order;
}
```

NOTE: Works with both REST and View based controller methods

Pivotal

@ExceptionHandler

- For existing exceptions you cannot annotate, use **@ExceptionHandler** method on controller
  - Method signature similar to request handling method
  - Also uses **@ResponseStatus**

```
@ResponseStatus(HttpStatus.CONFLICT)  // 409
@ExceptionHandler({DataIntegrityViolationException.class})
public void conflict() {
    // could add the exception, response, etc. as method params
}
```

Spring MVC offers several ways to handle exceptions, for more details see http://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc

`HiddenHttpMethodFilter`

- HTML forms do not support PUT or DELETE
  - Not even in HTML 5
- So use a POST
  - Put PUT or DELETE in a *hidden* form field
- Deploy a special filter to intercept the message
  - Restores the HTTP method you wanted to send
  - Request looks like a PUT or a DELETE to any Controller

See *HiddenHttpMethodFilter* online documentation

Agenda

- REST Introduction
- Spring MVC support for RESTful applications
- **RESTful Clients: `RestTemplate`**
- Lab
- Advanced Topics
  - More on Spring REST
  - **Spring HATEOAS**

HATEOAS - Concepts

- REST clients need *no* prior knowledge about how to interact with a particular application/server
  - SOAP web-services need a WSDL
  - SOA processes require a fixed interface defined using interface description language (IDL)
- Clients interact entirely through hypermedia
  - Provided dynamically by servers
- Serves to *decouple* client and server
  - Allows the server to evolve functionality independently
  - Unique compared to other architectures

HATEOAS Example (http://restcookbook.com/Basics/hateoas)

```
<account>
   <account-number>12345</account-number>
   <balance currency="usd">100.00</balance>
   <link rel="self" href="/accounts/12345" />
   <link rel="deposit" href="/accounts/12345/deposits" />
   <link rel="withdraw" href="/accounts/12345/withdrawls" />
   <link rel="transfer" href="/accounts/12345/transfers" />
   <link rel="close" href="/accounts/12345/close" />
</account>
```

Spring HATEOAS provides an API for generating these links in MVC Controller responses

```
<account>
   <account-number>12345</account-number>
   <balance currency="usd">-25.00</balance>
   <link rel="self" href="/accounts/12345" />
   <link rel="deposit" href="/accounts/12345/deposits" />
</account>
```

Note: links *change* as state changes

No standard for links yet - examples uses link style from *Hypertext Application Language* (HAL), one possible representation

Pivotal

Managing Links

- Use Link class
  - Holds an href and a rel (relationship)
  - Self implies the current resource
  - Link builder derives URL from Controller mappings

```
// A link can be built with a relationship name
//   Use withSelfRel() for a self link
Link link = ControllerLinkBuilder.linkTo(AccountController.class)
    .slash(accountId).slash("transfer")).withRel("transfer");

link.getRel();    // => transfer
link.getHref();   // => http://.../account/12345/transfer
```

Converting to a Resource

- Wrap return value of REST method in a **`Resource`**
  - Converted by **`@ResponseBody`** to XML/JSON with links
    - Only HAL supported currently

```
@Controller
@EnableHypermediaSupport(type=HypermediaType.HAL)
public class OrderController {

  @GetMapping(value="/orders/{id}")
  public @ResponseBody Resource<Order> getOrder(@PathVariable("id") long id) {
    Links[] = ...;  // Some links (see previous slide)
    return new Resource<Order>
        (orderService.findOrderById(id), links);
  }
}
```

Pivotal

Spring HATEOAS

- Spring Data sub-project for REST
  - For generating links in RESTful responses
  - Supports ATOM (newsfeed XML) and HAL (Hypertext Application Language) links
  - Many other features besides examples shown here
- For more information see
  - http://projects.spring.io/spring-hateoas/
  - http://spring.io/guides/gs/rest-hateoas/