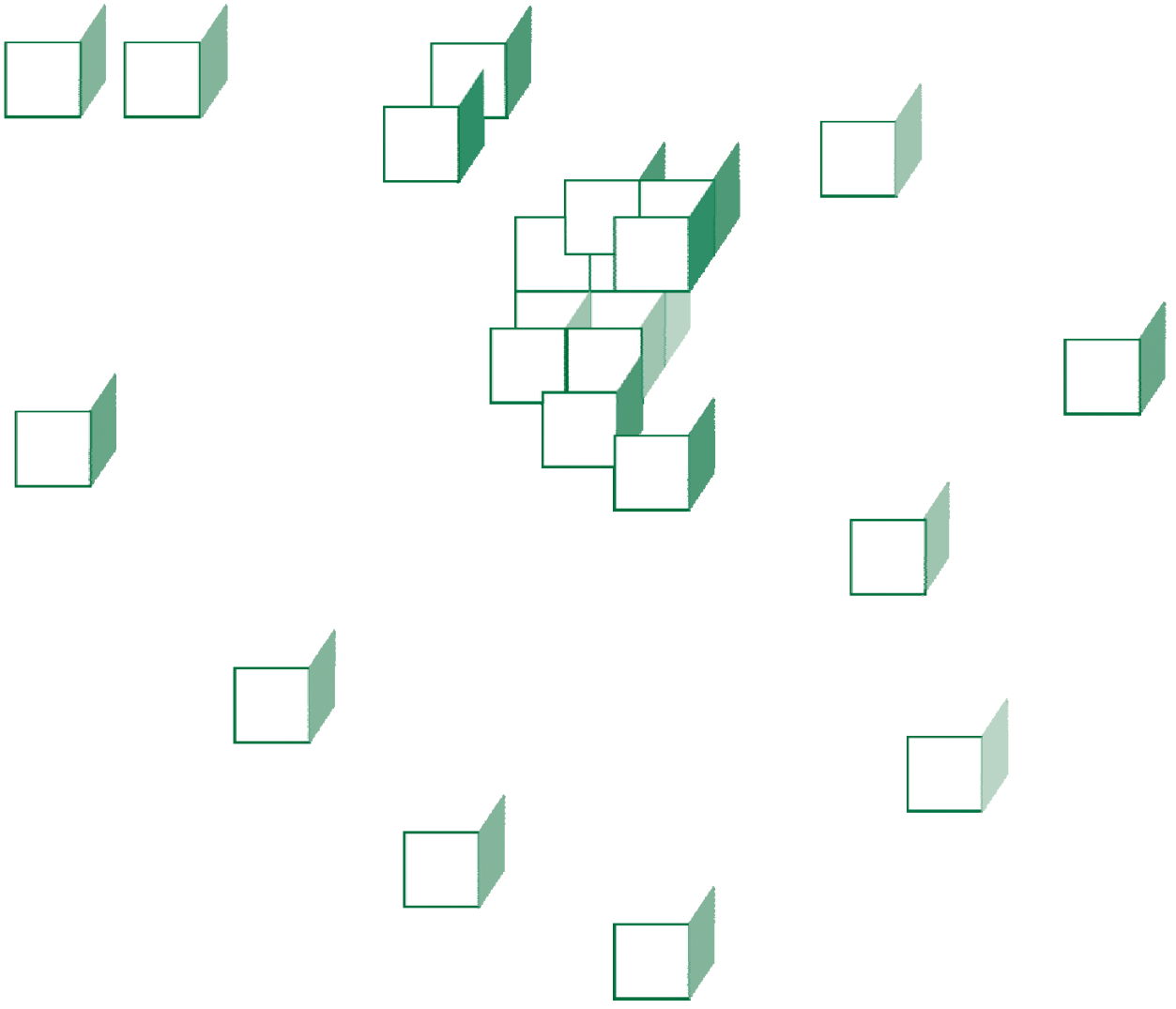
Chapter 4

Entities and Complex Content



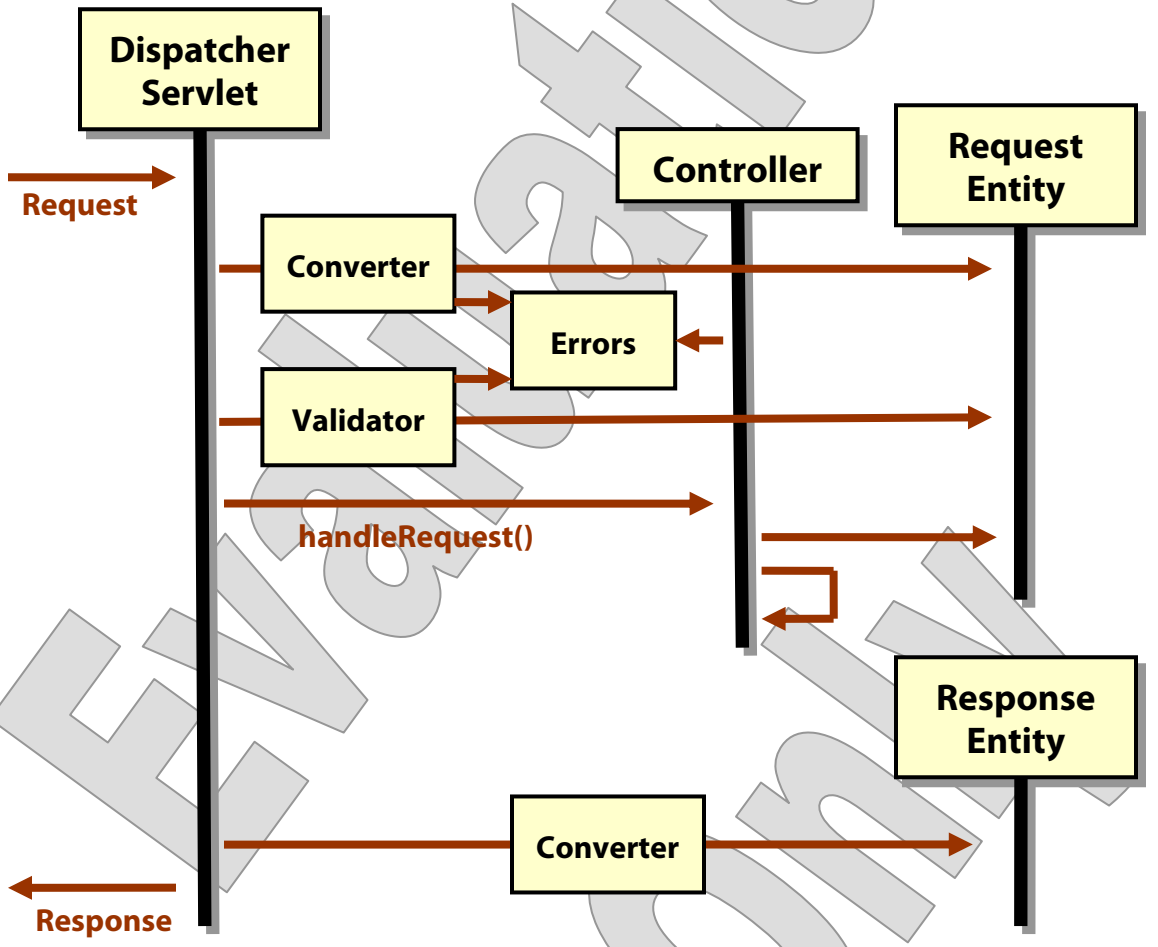
OBJECTIVES

After completing "Entities and Complex Content," you will be able to:

- Explain the use of certain built-in message converters for processing of HTTP entity content as
  - XML
  - JSON
- Use the **JacksonMappingHttpMessageConverter** to consume and to produce JSON representations in your web services.
- Create custom message converters to manage HTTP entities of other formats.

The Request/Response Cycle, Revisited

- A few chapters back we saw a summary interaction diagram of the Spring MVC request-handling cycle.
- Here is perhaps a better and more relevant diagram from the perspective of the REST developer:



  - We leave out many lower-level details; but observe the importance of the **Converter** and **Validator** to the process.
  - Both log to a common **Errors** object, which you can consult, and possibly change your intended page flow based on what you find.

Converters and Formatters

- **Spring 3.0 introduced a system that ultimately will replace PropertyEditors, using Converters and Formatters.**
  - **Converter** represents generalized type-conversion – that is, not just for converting "strings to things" and "things to strings," but also between different internal data representations.

```
public interface Converter<S,T>
{
  public T convert (S);
}
```

  - **Formatter** is the more direct replacement for **PropertyEditor**, as it assumes we're talking about **parsing** and **printing** strings.

```
public interface Formatter<T>
  extends Printer<T>, Parser<T>
{
  public T parse (String, Locale)
    throws ParseException;
  public String print (T, Locale);
}
```

- **Built-in (and possibly custom) converters and formatters are used heavily in Spring MVC web applications, and can have their place in REST services, too.**
  - They are used for all your **@RequestParam** and **@PathVariable** parameters, as well as for **headers** and **cookies** – and basically for any conversion tasks except for those involving the HTTP entity.

HttpMessageConverter

- We've seen a thing called **HttpMessageConverter**, a few times now, and now we can see that this is a specialized converter that treats the entire HTTP entity – whether request or response – and converts to and from a Java representation.

```
public interface HttpMessageConverter<T>
{
  boolean canRead (Class<?>, MediaType);
  boolean canWrite( Class<?>, MediaType);
  List<MediaType> getSupportedMediaTypes ();
  T read (Class<T>, HttpInputMessage)
    throws IOException,
           HttpMessageNotReadableException;
  void write (T t, HttpOutputMessage outputMessage)
    throws IOException,
           HttpMessageNotReadableException;
}
```

  - Type **T** is of course the Java type you will use to represent entity content.
  - **canRead**, **canWrite**, and **getSupportedMediaTypes** support discovery and selection of converters for a specific message.
  - **read** and **write** do the actual conversion.
  - Notice that a converter can offer to read requests, write responses, or both – so there can be **read-only** and **write-only** converters.

- Spring derives an **HttpMessageConverter** whenever you indicate that you want to interact directly with the HTTP entity …
  - With **@RequestBody** and **@ResponseBody**
  - With **HttpEntity<T>** and **ResponseEntity<T>**

Using <mvc:annotation-driven>

- **While it's possible to configure various Spring-3.x features with traditional Spring-beans elements, it's more common to use a shorthand made available as part of a new namespace:**

```
http://www.springframework.org/schema/mvc
```

- **Using the conventional prefix mvc:, you can simply declare:**

```
<mvc:annotation-driven/>
```

- **This gives you all the usual default configuration of the DispatcherServlet – annotation-driven request handler, default view resolver, exception resolver, etc.**
- **It also puts the whole family of Spring-3 features in play:**
  - A default **ConversionService**
  - The system of **HttpMessageConverters**
  - Support for JSR-303 validation using the **@Valid** annotation, as we'll discuss in the next chapter
- **The element also supports some customization attributes:**
  - Install a **conversion service**, to support custom converters and formatters
  - Declare a **global validator**
  - Declare request-handling **intercepters** (discussed in a later chapter)
  - Register custom **HttpMessageConverter**s

Built-In HttpMessageConverters

- **With <mvc:annotation-driven/>, the following mesasge converters are pre-registered:**
  - For simple string and binary representations, as we've been doing:

```
ByteArrayHttpMessageConverter
StringHttpMessageConverter
ResourceHttpMessageConverter
```

  - For XML conversion, using JAXP or JAXB:

```
SourceHttpMessageConverter
Jaxb2RootElementHttpMessageConverter
```

  - For JSON conversion:

```
MappingJacksonHttpMessageConverter
```

  - Miscellaneous and more or less useful for different applications:

```
FormHttpMessageConverter
AtomFeedHttpMessageConverter
RssChannelHttpMessageConverter
```

Working with XML

- **So there are two approaches to supporting XML entities.**
- **SourceHttpMessageConverter works with a JAXP Source object as its type T.**
  - The **Java API for XML Processing**, or **JAXP**, offers low-level APIs to XML content, parsing, and production, and supports various XML standards including the XML **DOM**, **SAX**, **XPath**, and **XSLT**.
  - It defines a system of **Source** and **Result** objects, partly as a way to support XSLT transformation but also as a way of piping content between different XML representations: as in from a DOM tree to a series of SAX events, or from a character stream to a DOM tree.
  - **Source** especially has become a major plug-in point for other APIs over the years.
- **Jaxb2RootElementHttpMessageConverter instead supports JAXB binding based on XML-Schema-to-Java type mappings.**
  - The **Java API for XML Binding**, or **JAXB**, offers a much higher-level programming model by translating XML content to a strongly typed Java object model, based on XML Schema definitions (or by generating an XML Schema to match a Java type model).
  - JAXB defines **annotations** for Java classes, fields, and methods that inform a processor as to the correct XML representation of Java content.
  - Many other tools and APIs support this annotation system, even if they don't do JAXB-style marshalling, un-marshalling, and binding proper.

Working with JSON

- The **JacksonMappingHttpMessageConverter** uses the **Jackson** utility to parse and to produce JavaScript Object Notation.
  - It can both **read** and **write**.
  - It supports **"application/json"** as a content type.
  - It can work with ordinary **JavaBeans**, but will also respect **JAXB annotations** if it encounters them on candidate classes.
- **Once this converter is working, it's great.**
- **It can be confusing to set up, for a couple of reasons.**
- **For one thing, while the converter itself is configured automatically, it will only make itself available for work if a Jackson implementation is found on the runtime class path.**
  - Various versions of Spring MVC require different versions of Jackson.
  - We're using **Jackson 1.9**.
  - If an appropriate version is not detected, the converter will just sit out the game – and quietly, oh! so quietly ...
- **Also, as you can imagine, there are lots of opportunities for JSON parsing to fail – a missing quotation mark or curly brace will upset the whole apple cart.**
- **What you might not imagine is that this converter will swallow such parsing exceptions, simply failing to convert with no log output, and causing the handler to return a bland HTTP 406.**
  - It will not even tell you where in the stream the failure occurs.

Working with JSON

DEMO

- **Let's add JSON support to the Ellipsoid service.**
  - It already has a web application co-deployed with it, so we can exercise a JSON interface from JavaScript on a new web page.
  - Do your work in **Demos_JSON**.
  - The completed demo is found in **Examples_Ellipsoid_Step3**.
- **The starter application now has an HTML page that we've not seen so far – docroot/allInOne.html – that uses XMLHttpRequest to send an Ajax request to our service.**

```
var ellipsoidString =
  JSON.stringify(ellipsoidIn);
var XHR = new XMLHttpRequest();
XHR.onreadystatechange = function()
  {
    if (XHR.readyState == 4)
    {
      var ellipsoidOut = eval("(" +
        XHR.responseText + ")");
      ...
    }
  }

var address =
  "http://localhost:8080/Demos_JSON/REST/Analyze";
XHR.open("post", address, true);
XHR.setRequestHeader
  ("Content-Type", "application/json");
XHR.send(ellipsoidString);
```

- It calls the JavaScript function on any change to the semi-axis values in the form fields.
- This allows it to update the derived fields on the page immediately.

1. Open **src/cc/math/EllipsoidService.java**, and add a new method that takes an ellipsoid and echoes it back:

```
public Ellipsoid getEllipsoid
  (Ellipsoid ellipsoid)
{
  return ellipsoid;
}
```

- Now, what good will that do?
- In ordinary Java programming, probably none at all.
- But if we can prompt Spring to **parse** something into an **Ellipsoid** instance, based only on the writeable properties **a**, **b**, and **c**, then when it **produces** the object again, it will have all the calculated properties **volume**, **type**, and **definition**.
- So, query and response – that's worth something to somebody!

2. Annotate the method as a request handler:

```
@RequestMapping("/Analyze")
@ResponseBody
public Ellipsoid getEllipsoid
```

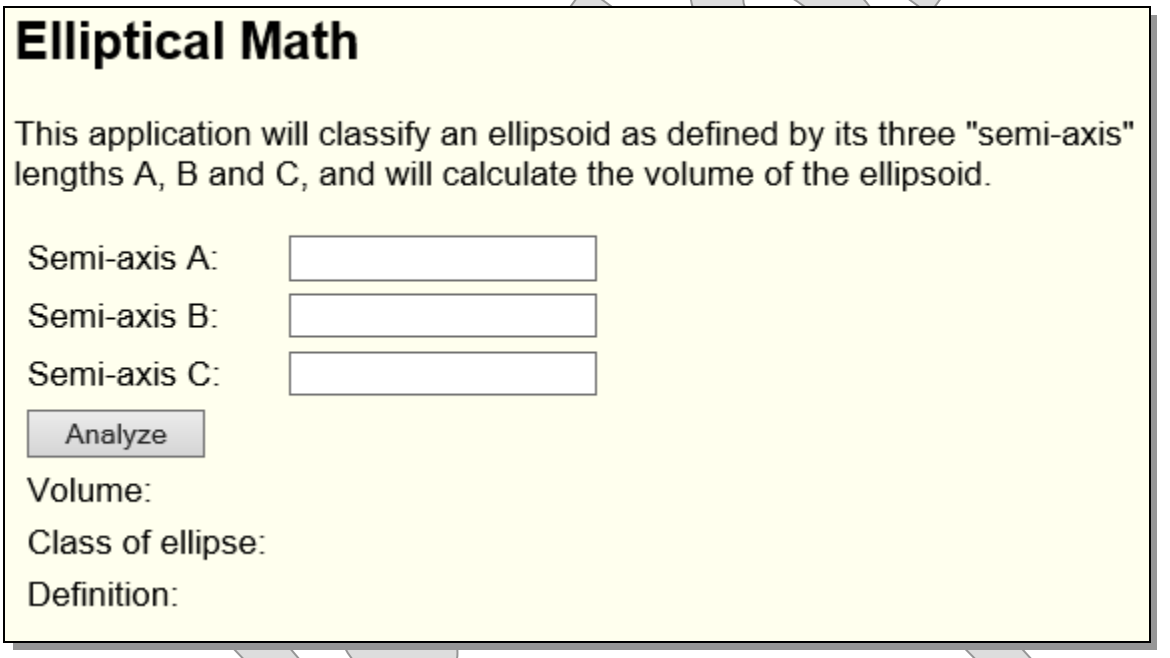
DEMO

3. Deploy the application and try it out in a browser:

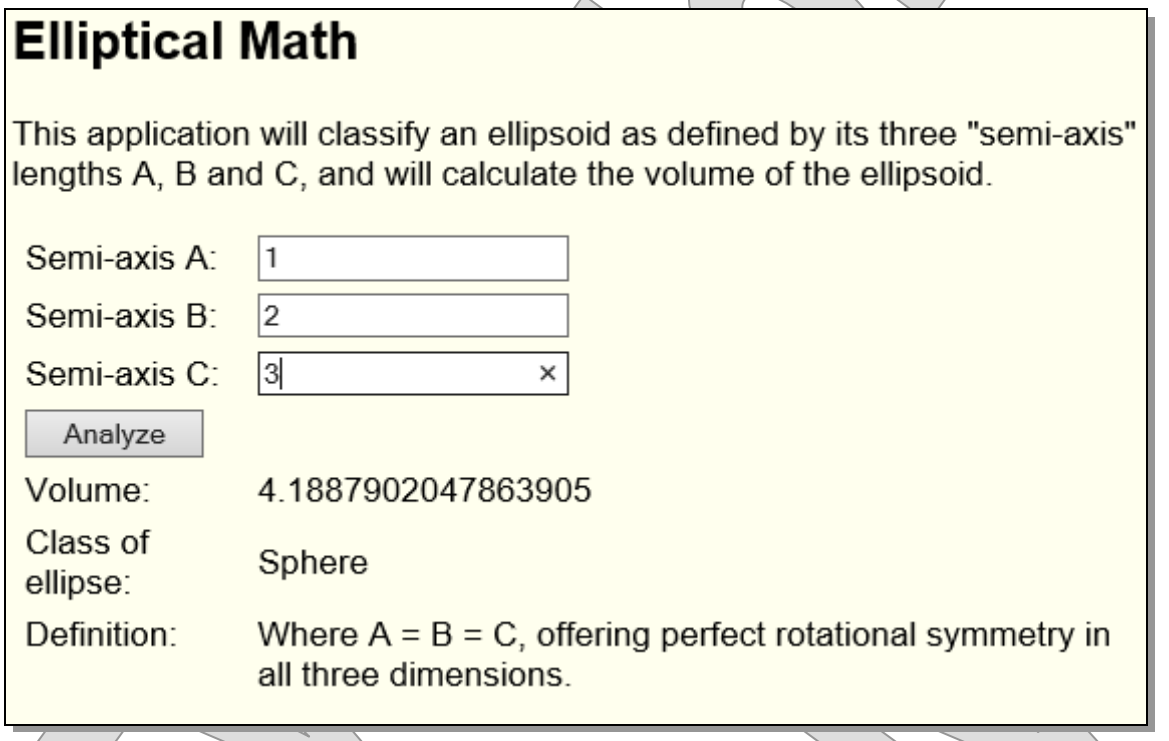
`http://localhost:8080/Ellipsoid/allInOne.html`

or

`http://localhost:8080/Demos_JSON/allInOne.html`



4. Fill in all three values, and when you have numbers in all three, the resident JavaScript function will send a request, and you'll see the full ellipsoid information including values filled out in the lower part of the page:



   - Wait, though: are those values right?
   - It would be a strange piece of software indeed that classified an ellipsoid with dimensions 1, 2, and 3 as a sphere. What's going on?

5. Use HTTPPad to be more certain of the HTTP request and response:

```
POST /Demos_JSON/REST/Analyze HTTP/1.1
Content-Type: application/json

{"a":1,"b":2,"c":3}

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{"a":1.0,"b":1.0,"c":1.0,"type":"Sphere",
"definition":"Where A = B = C, offering perfect
rotational symmetry in all three dimensions.",
"volume":4.1887902047863905}
```

- Look again at the service method signature:

```
@RequestMapping(value="/Analyze")
@ResponseBody
public Ellipsoid getEllipsoid
  (Ellipsoid ellipsoid)
```

  - This method has a signature very similar to the others in this service – and, remember, they are using traditional command objects to parse HTTP query strings or form data.
  - That's not what we're looking for anymore; we want to parse the HTTP request body as JSON.

DEMO

- **In fact, we didn't provide any data in our request, at all!**
  - Our **JSON** entity was **ignored**.
  - Form **binding** to the command object also **failed** – but this is not a fatal error in ordinary data binding; it just means no form parameters matched.
  - As a result we're getting the **default** ellipsoid, which happens to be a sphere of radius 1.0.
  - See **src/cc/math/Ellipsoid.java** for the proof:

```
private double a = 1;
private double b = 1;
private double c = 1;
```

6. Annotate the method parameter to clarify that it should consume the HTTP request entity:

```
public Ellipsoid getEllipsoid
  (@RequestBody Ellipsoid ellipsoid)
```

7. Build again, and test. Now you should see more sensible values that indicate that the JSON entity is being parsed and used:

Volume:	25.132741228718345
Class of ellipse:	Triaxial ellipsoid
Definition:	Where none of A, B and C are equal; the most general classification of an ellipsoid.

Suggested time: 30 minutes

In this lab you will add operations to the Tracking service that allow the caller to:

- Request a detailed record of an existing order
- Place a new order by submitting the entire record
- Add lines to an existing order

Detailed instructions are found at the end of the chapter.

Custom Message Converters

- **It's not especially painful to implement HttpMessageConverter yourself.**
  - Decide if you need or want a read-only, write-only, or read-write converter, and implement **canRead** and **canWrite** accordingly.
  - Implement **getSupportedMediaTypes** – often with a hard-coded list of the one MIME type that interests you.
  - Implement **read** and/or **write**, and the parsing or printing logic is up to you – and is nothing specific to HTTP or REST. Often it will be code borrowed from elsewhere in your application.
- **You can register your converter easily, using a child element of the standard configuration element:**

```
<mvc:annotation-driven>
  <mvc:message-converters>
    <bean class="com.me.MyConverterClass" />
  </mvc:message-converters>
</mvc:annotation-driven>
```

- **Remember, your converter will be used under specific conditions:**
  - If you **canRead**, then the request's **content type** must be one of your **supported types** and one of the types that the method **consumes**.
  - If you **canWrite**, then one of the types you promise to support must be in both the set of **acceptable types** based on the request and the set of types that the method **produces**.

A Custom Converter

DEMO

- **We'll look at a couple examples of entity providers in this demonstration, and enable the second of them to give an application the ability to provide a compressed response.**
  - Do your work in **Demos_Converter**.
  - The completed demo is in **Examples_DNA_Step2**.
- **The service itself is nearly trivial: one class acts as application and root resource, and accepts and provides instances of another:**

- **What's more interesting is the question of in what form the information will be accepted and provided.**
  - **DNASequence** is just a wrapper around a string value.
  - But Spring doesn't know that, so we have to manage entity consumption and production using the custom converter **DNASequenceTextConverter**.
  - You'll build a second, **write-only converter** that will be able to render a sequence in a more compact, binary form.

A Custom Converter

DEMO

1. Open **src/cc/rest/DNAAnalysis.java** and see two service methods to read and write an instance of **DNASequence**:

```
@Controller
public class DNAAnalysis
{
  private DNASequence current;

  @RequestMapping
    (method=RequestMethod.POST, value="Analyze")
  @ResponseBody
  public String analyze
    (@RequestBody DNASequence sequence)
  {
    current = sequence;
    return "";
  }

  @RequestMapping
    (method=RequestMethod.GET, value="Retrieve",
     produces={MediaType.TEXT_PLAIN_VALUE})
  @ResponseBody
  public DNASequence retrieve ()
  {
    return current;
  }
}
```

   - Note that **retrieve** states explicitly that it **produces** plain text.
   - **analyze** is silent on the question of MIME type – which means it's open to anything.

DEMO

2. Open **src/cc/rest/DNASequenceTextConverter.java**, and review the code there. This class reads and writes "text/plain":

```
public class DNASequenceTextConverter
  implements HttpMessageConverter
{
```

   – It can read and write a specific MIME type for a a specific class:

```
  public boolean canRead
    (Class type, MediaType mediaType)
  {
    return type.equals (DNASequence.class) &&
      mediaType.equals (MediaType.TEXT_PLAIN);
  }

  public boolean canWrite
    (Class type, MediaType mediaType)
  {
    return type.equals (DNASequence.class) &&
      mediaType.equals (MediaType.TEXT_PLAIN);
  }
```

   – It supports just one media type:

```
  @SuppressWarnings("unchecked")
  public List getSupportedMediaTypes ()
  {
    List result = new ArrayList();
    result.add (MediaType.TEXT_PLAIN);
    return result;
  }
```

- It reads the whole entity as a string and makes that the **codons** property of a new **DNASequence**:

```
public Object read
  (Class type, HttpInputMessage message)
    throws IOException,
      HttpMessageNotReadableException
{
  DNASequence result = new DNASequence ();
  result.setCodons (new BufferedReader
    (new InputStreamReader (message.getBody ()))
      .readLine ());

  return result;
}
```

- And it simply prints the **codons** string to create a new entity:

```
public void write (Object object,
  MediaType mediaType, HttpOutputMessage message)
    throws IOException,
      HttpMessageNotWritableException
{
  new PrintStream (message.getBody ()).println
    (((DNASequence) object).getCodons ());
}
}
```

3. See the configuration in **docroot/WEB-INF/REST-servlet.xml**:

```
<mvc:annotation-driven>
  <mvc:message-converters>
    <bean class=
      "cc.rest.DNASequenceTextConverter" />
  </mvc:message-converters>
</mvc:annotation-driven>
```

4. Deploy the application, and test using HTTPPad. Use the script in **Http/TestScript.txt**:

```
POST /@ROOT@/REST/Analyze HTTP/1.1
Content-type: text/plain
Accept: text/plain

GCCATTGGCACCTAAGCCAT...

HTTP/1.1 200 OK
Content-Type: text/plain;charset=ISO-8859-1

(No message body.)


GET /@ROOT@/REST/Retrieve HTTP/1.1

HTTP/1.1 200 OK

GCCATTGGCACCTAAGCCAT...


GET /@ROOT@/REST/Retrieve HTTP/1.1
Accept: application/octet-stream

HTTP/1.1 406 Not Acceptable
Content-Type: text/html;charset=utf-8

<HTML content here>
```

   - So Spring refuses to invoke **retrieve**, because we say explicitly that we only produce plain text, and the request insists on binary data.
   - We'll add a second converter now to support that last request.

A Custom Converter

DEMO

5. Add the required type to the production for the **retrieve** method:

```
@Produces({MediaType.TEXT_PLAIN,
  MediaType.APPLICATION_OCTET_STREAM_VALUE})
public DNASequence retrieve ()
```

6. Open **DNASequenceBinaryConverter.java**, which currently has a plain old helper method that knows how to compress a given DNA sequence from text format to a byte array holding two bits per codon. The full code of this method isn't important, and isn't reproduced here. You will modify this method in spots.

7. Make the class implement **HttpMessageConverter**.

```
public class DNASequenceBinaryConverter
  implements HttpMessageConverter
```

8. Implement **canRead** to return **false**, as we won't try to parse binary content, just to produce it.

```
public boolean canRead (Class c, MediaType t)
{
  return false;
}
```

9. Implement **canWrite** to return **true** for our specific class and MIME type. This is a decent candidate for copy-and-paste from the text converter:

```
public boolean canWrite
  (Class type, MediaType mediaType)
{
  return type.equals (DNASequence.class) &&
    mediaType.equals
      (MediaType.APPLICATION_OCTET_STREAM);
}
```

10. **getSupportedMediaTypes** can also be copied in and modified: the only change is the media type we offer to support:

```
public List getSupportedMediaTypes ()
{
  List result = new ArrayList();
  result.add
    (MediaType.APPLICATION_OCTET_STREAM);
  return result;
}
```

11. **read** is a no-op:

```
public Object read (Class c, HttpInputMessage m)
  throws IOException,
    HttpMessageNotReadableException
{
  return null;
}
```

12. Now you can modify the **write** method so that it works as an implementation of **HttpMessageConverter.write**. First, change the method signature to match. (If using Eclipse, you've probably already had the IDE generate the unimplemented methods, so you can just merge your two **write** methods now.)

```
public void write (Object object,
  MediaType mediaType, HttpOutputMessage message)
    throws IOException,
      HttpMessageNotWritableException
```

13. Now your parameter **object** is not strongly typed, as **sequence** was in the starter implementation. But you won't get a call to this method unless Spring has first checked your **canWrite**, and that narrows the type of **object**. Replace the one occurrence of **sequence** with a downcast of **object**:

```
String codons =
  ((DNASequence) object).getCodons ();
```

14. There is another type shift in the new signature: the method is implemented to use an **OutputStream** called **out**. You now take an **HttpOutputMessage**, and this can provide a reference to an output stream that writes to the response body. So replace the two occurrences of **out** with calls to **getBody**:

```
    ...
    if (rotation == 6)
    {
      message.getBody ().write (compressed);
      compressed = 0;
    }
  }

  if (codons.length () % 4 != 0)
    message.getBody ().write (compressed);
}
```

A Custom Converter

DEMO

15. In **REST-servlet.xml**, register your converter:

```
<mvc:annotation-driven>
  <mvc:message-converters>
    <bean class=
      "cc.rest.DNASequenceTextConverter" />
    <bean class=
      "cc.rest.DNASequenceBinaryConverter" />
  </mvc:message-converters>
</mvc:annotation-driven>
```

16. Deploy the application and test again. Now, you should see the binary-data request is granted – though HTTPPad is not real good at representing the information:

```
GET /@ROOT@/REST/Retrieve HTTP/1.1
Accept: application/octet-stream

HTTP/1.1 200 OK

¯QƒÅkÔ`ñ5X¼F
¯QƒÅkÔ`ñ5X¼F
¯QƒÅkÔ`ñ5X¼F
¯QƒÅkÔ`ñ5X¼F
¯QƒÅkÔ`ñ5X¼F
¯QƒÅkÔ`ñ5X¼F
...
```

SUMMARY

- They almost hit the REST nail on the head with the formatter and converter system in Spring 3!
- But ultimately an alternate system of **HttpMessageConverter**s was called for, and it is simple and straightforward to use.
- Built-in converters (by which we mean those that are configured automatically with `<mvc:annotation-driven>`) cover most common REST content types quite well:
  - Plain text
  - Raw binary content
  - Image formats
  - XML
  - JSON
- Custom converter implementation isn't trivial, but it's not that hard, either, and from there it's possible to support any content type, for reading or for writing, just about any way you might want to do.