Support both XML and JSON

00:00: Welcome to the fifth module of Course Two: Supporting Both XML and JSON. The decision to support multiple media types in the same API will definitely be an API-specific decision. But content negotiation is very well supported by spring so we will try to leverage that and support both XML, JSON, as well as some other media types as we work our way through the course. So in this module, we're going to learn how to first add XML support in the API, very basic XML support. We're then going to start exploring content negotiation, we're gonna see how that works by default, and we're going to dive a little bit deeper than usual into the actual mechanism that spring uses to do content negotiation and do it well. And finally, we're definitely going to tune the XML marshalling and unmarshaling processes in the web configuration the same way we tuned the JSON marshaller and unmarshaller.

01:02: Let's start with adding the XML dependency here in Maven, and we're actually going to use the Jackson XML module, which is quite an interesting option we have starting with spring 4.1. So, previously we had other libraries such as the extreme library but using any of those libraries traditionally meant that we needed to annotate our DTOs with extra notations and it also meant that we needed to learn and configure an extra library, which is not ideal. So using the Jackson first class support for XML is hugely powerful because we're going to still be using Jackson and we will benefit from all the well known configuration that we already have a good grasp on with Jackson. So let's add this here.

01:52: So, very simple, exactly like the Jackson dependencies we are adding here the Jackson data format XML dependency and that brings out of the box XML support to our API. We are actually going to add another dependency, which is going to replace the default out of the box stacks implementation that we get in the JDK, and the main reason we're gonna go with another stacks implementation is that it's much more well maintained and generally a better more mature implementation. So let's add the dependency here. The library is called Woodstocks and it's quite well maintained and super fast. And we're all set. This is the full XML support that we need in terms of libraries to enable XML support in the API.

02:39: Okay, so now that we have added the XML libraries in Maven, they're now available at run time so they're also available to spring. And so exactly like Jackson spring will detect the new library and it will enable XML support out of the box. We will of course configure that unit in going to more detail but it is important to understand that out of the box is a decent experience and we are going to see what that looks like right now.

03:05: So let's now consume the privileges collection resource. Okay, so we are getting the 200 okay here as expected and so let's now see what the body looks like and notice that we are not providing any accept header. So we're not informing the API what kind of media type we want for our representations. We are just asking without any specific information there. Okay. So you can see that now we're getting XML instead of JSON, and why is that? Well, the out of the box configuration is good but we don't really have control over which of the HTTP message converters actually goes first. So in this case the XML message converter went first and so we are getting XMLE for not specifying anything. We can definitely have full control over the order of those HTTP message converters and so we can determine which goes first and which is the default, but it's always, always, always good practice to use the accept header and to specifically ask for a media type for our representation. So, even if we only support JSON, it's always good from the point of

view of the client to ask for specifics because who knows in the future the API might support XML, and if you're not asking for specific as a client of an API then you might get something else in the future.

04:33: So let's see how that looks if we now set the accept header. So we're gonna define it now. We're also going to save it as a favorite and let's go. And here we are. We are getting the right representation now, we are getting JSON and so that is basic content negotiation from the point of view of the client: Asking for a specific media type by using the accept header and getting that representation correctly delivered back to the client. Understanding content negotiation at the surface level is good and knowing that spring adheres to the HTTP spec quite well out of the box is also great but let's actually see what happens when spring processes the request. Let's actually dig into the strategy that's responsible with doing the content negotiation itself and let's see how that actually works. So we're gonna put a break point right here and we're going to repeat that request from the client side.

05:35: Here's the break point being triggered by our new request and we're going to basically just run through this method and go over the exact details of what's happening and how content negotiation is actually done by spring. And as you can see this is not a complex implementation here but it's quite interesting to follow and it's also quite important to realize how the strategies work because you may need to implement a different kind of strategy and there are a few to choose from. For example, if you look at the implementations of this interface here, so you can see that we do have a few negotiation strategies. Actually the interesting ones are here, the parameter and the path extension-based strategies. These are two strategies that are not really well suited for a REST API, but they're quite useful in other scenarios, in MVC-focused type of scenarios.

06:29: Okay, let's run through these and let's see exactly what happens. So obviously the first thing we need to do in a header-content negotiation strategy is to look at the header, specifically the accept header here and let's open up the variables here and see what happens. So the accept header is application/json which is what we are sending from the client as expected. Now after checking that the accept header actually has some text because if you're not sending anything from the client then this content negotiation strategy doesn't have enough information to actually make a determination in terms of what media type we need. Spring will then parse the media type out of this application/json string. So it's going to try to determine what the media type we are requesting is, and of course in this case application/json will map very simply. However, the browser usually sends a much more convoluted, much more complex type of accept header.

07:29: So this is why parsing the media type is quite a complex process in some cases. In this case, it's not going to be so we're gonna see what the media type is here. It's application/json but this time it's an actual media type object. And as I was saying the browser usually does send more complex media types here so that's why we need to sort by specificity and quality, which are two artifacts in this potentially long string in the accept header, and there we go. We are returning back a list of media types sorted according to their priority. Let's now have a look at what the XML output looks like when we start consuming the API with the default configuration without any sort of tuning. So we are going to retrieve the privileges collection resource and we're going to have a look at the formatting of the output. As you can see, the output looks valid, it's definitely XML;

however, it's not well configured. And remember we saw this with JSON output at the very beginning of the course before we had the chance to actually tune the configuration and add those extra properties on the HTTP message converter and so that's exactly what's happening here.

08:48: Without any sort of tuning of the marshallers, we are getting the raw JSON output which is not pretty printed. It doesn't have any sort of indentation. We're also going to have a look at the create privilege operation where we are using XML, we are passing in what looks to be a valid privilege; however, notice this extra field here which does not map to anything in the API and so that means it's going to be an unknown field and without any extra configuration the API will unfortunately accept this extra field and will simply ignore it. So let's see how the create actually does work even when we're passing in this extra information which doesn't have any meaning to the API. And here we go, we are getting back the 201 created, which means that the request went through and the creation actually happened. And so what we want to do now is we want to configure the XML part of the application, the exact same way we configured the JSON aspect. We want to make sure that we're getting pretty printed XML out of the API and we want to make sure that unknown properties like this one are not accepted and ignored are actually rejected as a bad request from the client.

10:07: Now that we've seen how the XML support behaves out of the box, let's actually start adding some extra configuration to it the same way we did with the Jackson configurations. So here we have the existing Jackson configuration where we were configuring the indent output so that we do get pretty printed well-indented output back on the client side and we were also configuring the unmarshaller here to fail when any unknown property occurs. So let's now do the exact same configuration pretty-print or indented output and failing on unknown properties. And because we're dealing with Jackson we have the great advantage of not having to, again, learn another library or do some completely different style of configuration. So we can have the benefit of simply copying this one, adapting it a bit to instead of finding the Jackson JSON converter find the XML converter and then configure it in the exact same way. And this is a hugely powerful thing because we can leverage the existing configuration out of Jackson and do the exact same configuration for XML.

[pause]

11:34: And there we go. We can see how similar these two configurations are. They're exactly the same only we're looking for a slightly different implementation class here, which is super cool and saves a lot of time especially if you have more complex configurations. These two properties, not so complex. However, when you have 15 or 20 properties, not spending the time to do the same type of configuration across two completely different libraries is a huge time saver. So, now that the new configuration has been deployed let's see what the XML output looks like after we turned on pretty print or indentation of the output. So, we get back the 200 okay and here we go. We get back a nicely, indented output where we really don't need the highlighted section here, we can just read the output as it is and it's quite well indented and quite readable.

12:34: Next, let's have a look at the create operation, we're trying to create a privilege but notice how we're passing this extra element here in the XML representation of our new privilege and we

saw before how this was simply accepted and ignored and we're gonna see now how the new configuration makes the marshalling process who reject this new field and essentially mark the request correctly as a bad request. And here we go, we're getting the 400 bad request as we should.
Let's wrap up with the major takeaways of this model. And of course the main thing is we added XML support into the API but the interesting part about that is that we use the Jackson XML support, the newly introduced XML support available starting with Spring 4.1, and we tuned that XML marshalling and unmarshalling exactly the same way we tuned the JSON support. So, that was one of the major advantages of using the XML support out of Jackson and not another third party library such as extreme.

13:39: Alright, hope you're excited. See you in the next one.