Refactoring Code to Load a Document

Much modern web server code talks to upstream services which return JSON data, do a little munging of that JSON data, and send it over to rich client web pages using fashionable single page application frameworks. Talking to people working with such systems I hear a fair bit of frustration of how much work they need to do to manipulate these JSON documents. Much of this frustration could be avoided by encapsulating a combination of loading strategies.

---

17 December 2015

Martin Fowler

Find **similar articles** at the tag: refactoring

When I'm loading a document - be it JSON, XML, or any other hierarchy of data - I have to choose how to represent it in my program. I don't write much of the loading code myself, there are libraries that are out there to do that for me, but I still need to choose where that data ends up. That choice should be based on how I'm going to use the data. But I frequently run into programs that load everything into a complex object structure, often ending up with unnecessary code that's tedious to keep up to date.

Initial Code

To illustrate some different approaches I'll use an example service that provides information about assortments of music albums. The data it consumes looks something like this.

```
  {
    "albums": [
      {
        "title": "Isla",
        "artist": "Portico Quartet",
        "tracks": [
          {"title": "Paper Scissors Stone", "lengthInSeconds": 327},
          {"title": "The Visitor", "lengthInSeconds": 330},
          {"title": "Dawn Patrol", "lengthInSeconds": 359},
          {"title": "Line", "lengthInSeconds": 449},
          {"title": "Life Mask (Interlude)", "lengthInSeconds": 75},
          {"title": "Clipper", "lengthInSeconds": 392},
          {"title": "Life Mask", "lengthInSeconds": 436},
          {"title": "Isla", "lengthInSeconds": 310},
          {"title": "Shed Song (Improv No 1)", "lengthInSeconds": 503},
          {"title": "Su-Bo's Mental Meltdown", "lengthInSeconds": 347}
        ]
      },
      {
        "title": "Horizon",
        "artist": "Eyot",
        "tracks": [
          {"title": "Far Afield", "lengthInSeconds": 423},
          {"title": "Stone upon stone upon stone", "lengthInSeconds":
479},
          {"title": "If I could say what I want to", "lengthInSeconds":
167},
          {"title": "All I want to say", "lengthInSeconds": 337},
          {"title": "Surge", "lengthInSeconds": 620},
          {"title": "3 Months later", "lengthInSeconds": 516},
          {"title": "Horizon", "lengthInSeconds": 616},
          {"title": "Whale song", "lengthInSeconds": 344},
          {"title": "It's time to go home", "lengthInSeconds": 539}
        ]
      }

    ]
  }
```

The service is written in Java, using the Jackson library to read the JSON. The usual way to do this is to define a series of Java classes and use Jackson's nifty data binding feature to map the JSON data into the fields of the class. To handle this data we'd need these classes.

class Assortment…

```
  private List<Album> albums;

  public List<Album> getAlbums() {
    return Collections.unmodifiableList(albums);
  }
```

class Album…

```
  private String artist;
  private String title;
  private List<Track> tracks;

  public String getArtist() {
    return artist;
  }
  public String getTitle() {
    return title;
  }
  public List<Track> getTracks() {
    return Collections.unmodifiableList(tracks);
  }
```

class Track…

```
  private String title;
  private int lengthInSeconds;
  public String getTitle() {
    return title;
  }
  public int getLengthInSeconds() {
    return lengthInSeconds;
  }
```

Jackson uses reflection to automatically map JSON data to the corresponding Java objects. Thanks to this, I don't have to write any code to load up the objects. I do, however, have to define the objects that the JSON loads into.

> I can use either public fields, or private fields with public getters. The getters make for more verbose code, but I prefer to use them as I like to follow the Uniform Access Principle. IntelliJ writes the getters for me, which further reduces the annoyance.

With the classes defined like this, then loading the JSON data is a simple method call

class Service...

```
  public String tuesdayMusic(String query) {
    try {
      Assortment data =
Json.mapper().readValue(dataSource.getAlbumList(query),
Assortment.class);
      return Json.mapper().writeValueAsString(data);
    } catch (Exception e) {
      log(e);
      throw new RuntimeException(e);
    }
  }
```

> *The JSON data comes from some data source via the call `dataSource.getAlbum(query)`. This could be a call to another service, access to a JSON-oriented database, read from a file, or any other source that doesn't concern me at the moment.*

class Json...

```
  public static ObjectMapper mapper() {
    JsonFactory f = new
JsonFactory().enable(JsonParser.Feature.ALLOW_COMMENTS);
    return new ObjectMapper(f);
  }
```

While I don't have to write any code to load the objects from the JSON data, I do have to write the object definitions. That's not much in this case, but I've run into cases where there's a hundred classes needed to represent the JSON data. If there's not much use being made of these objects, then writing such class definitions is annoying busywork.

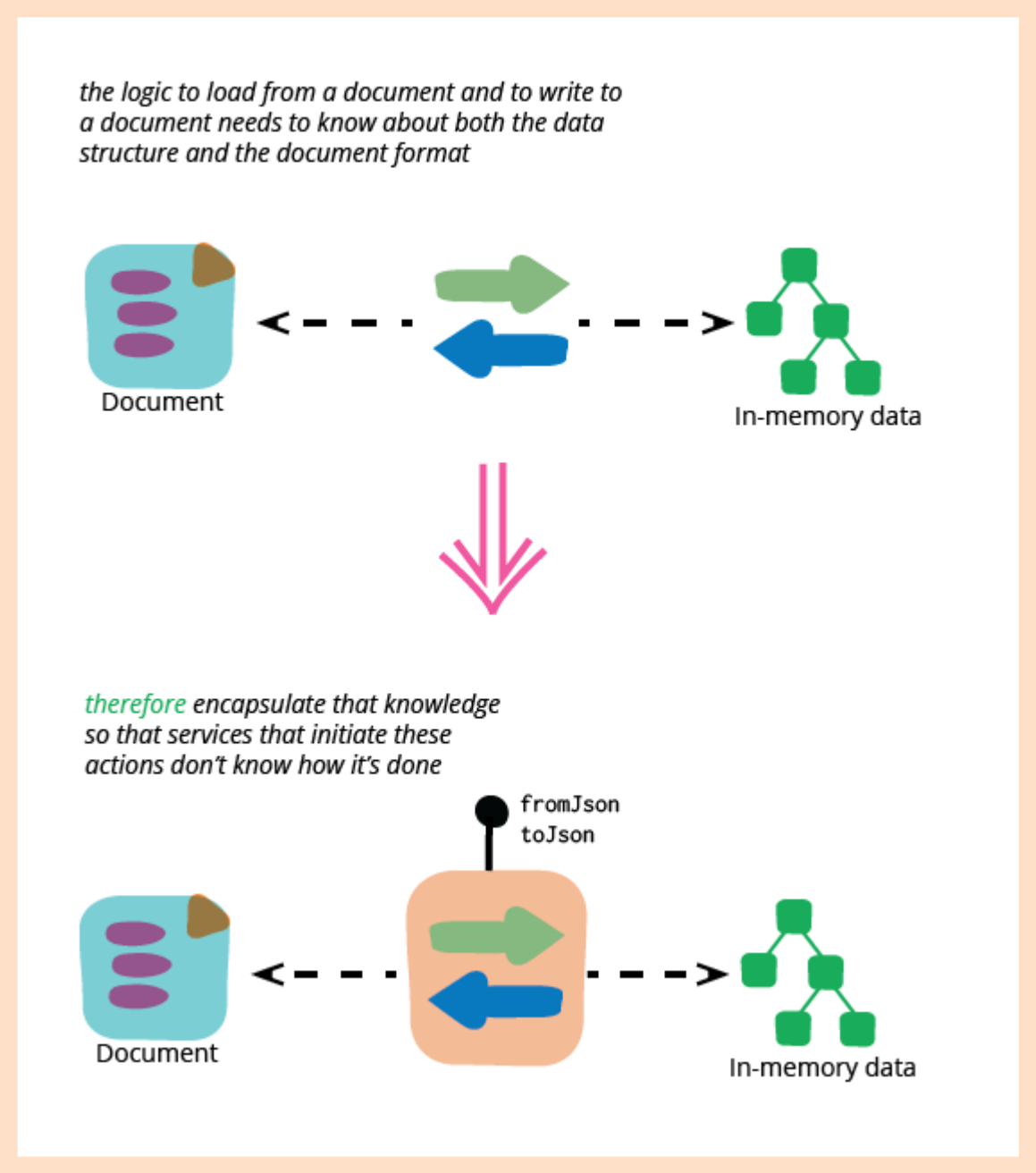
The case where it's justified is where we're using all the objects and methods defined in the databinding. So if I'm defining ten classes and 200 public methods, and only using five of them, that's a sign that something is wrong. To examine the alternatives, we need to look a few different cases and consider the various alternative routes we can take.

But whatever the alternative, I begin with the same first step.

Encapsulating the assortment

Whenever I make a refactoring, I look to see how I can reduce the visibility of change by

encapsulating the bulk of the change behind some suitable methods. Since I'm working with assortments, this means moving the responsibility for translating the assortment to and from JSON into the assortment itself.



I begin this by using Extract Method on the code that loads the assortment.

class Service...

```
  public String tuesdayMusic(String query) {
    try {
      Assortment data = loadAssortment(query);
      return Json.mapper().writeValueAsString(data);
    } catch (Exception e) {
      log(e);
      throw new RuntimeException(e);
```

```
    }
  }
  private Assortment loadAssortment(String query) throws
java.io.IOException {
    return Json.mapper().readValue(dataSource.getAlbumList(query),
Assortment.class);
  }
```

I'm not a fan of checked exceptions, so my default is to wrap them in a runtime exception as soon as they rear their ugly head.

class Service...

```
  private Assortment loadAssortment(String query) {
    try {
      return Json.mapper().readValue(dataSource.getAlbumList(query),
Assortment.class);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
```

I want the new method to take a string of JSON data, so I adjust the arguments to the new method.

class Service...

```
  public String tuesdayMusic(String query) {
    try {
      Assortment data = loadAssortment(dataSource.getAlbumList(query));
      return Json.mapper().writeValueAsString(data);
    } catch (Exception e) {
      log(e);
      throw new RuntimeException(e);
    }
  }
  private Assortment loadAssortment(String json) {
    try {
      return Json.mapper().readValue(json, Assortment.class);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
```

With the behavior nicely extracted, I can then move it to the assortment class as a factory method.

class Service...

```
  public String tuesdayMusic(String query) {
    try {
      Assortment data =
Assortment.fromJson(dataSource.getAlbumList(query));
      return Json.mapper().writeValueAsString(data);
    } catch (Exception e) {
      log(e);
      throw new RuntimeException(e);
    }
  }
```

class Assortment...

```
  public static Assortment fromJson(String json) {
    try {
      return Json.mapper().readValue(json, Assortment.class);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
```

I couldn't do this using IntelliJ's Move Method refactoring, but it was easy to do it manually.

I used the same sequence of steps to extract and move the code to emit the JSON.

class Service...

```
  public String tuesdayMusic(String query) {
    try {
      Assortment data =
Assortment.fromJson(dataSource.getAlbumList(query));
      return data.toJson();
    } catch (Exception e) {
      log(e);
      throw new RuntimeException(e);
    }
  }
```

class Assortment...

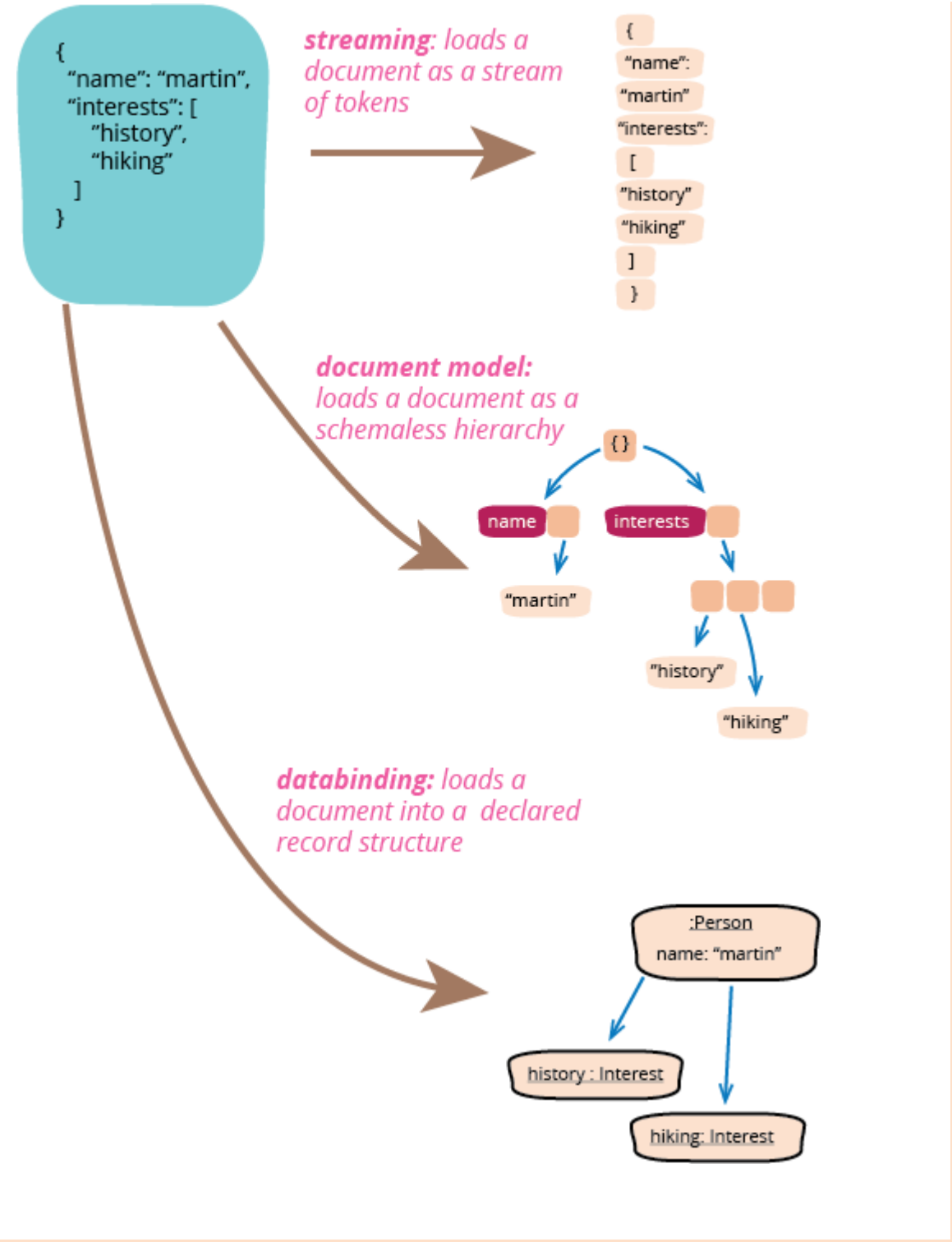
```
  public String toJson() {
    try {
```

```
      return Json.mapper().writeValueAsString(this);
    } catch (JsonProcessingException e) {
      throw new RuntimeException(e);
    }
  }
```

If I had other service methods that were using assortments like this, I'd now adjust all their calls to use these new methods. Once I'm done I have objects that encapsulate their mechanism for serialization, which makes it much easier for me to change it.

> *The essence of encapsulation is turning a design decision into a secret -- Martin Fowler*

Some people might find it odd that I used the phrase "encapsulate their mechanism for serialization". When people are taught about object-orientation, they are often told that data is encapsulated, so it might sound odd to be thinking of behavior like this as something that should be encapsulated. However the whole point of encapsulation is taking some decision, which may be a choice of behavior or data structure, and turning it into a secret so it's hidden from the outside which doesn't need to know what's going on behind the encapsulation boundary. In this case I don't want any code outside the assortment to know how the assortment relates to the JSON, so I encapsulate the code that does the loading within the assortment class.



The lowest level approach is **streaming**. In this style the document is chopped up into tokens and I configure the streamer with code to operate on tokens one at a time. An example is XML's SAX (which treats tokens as events, but it's the same idea). Streaming's big advantage is performance, minimizing both memory usage and processing time within the tokenizer. The downside is more work on the consumer, who has to figure out how to turn a stream of tokens into a useful structure. I don't tend to use streaming unless I'm dealing with a document large enough to cause problems loading it into memory. It would be unlikely to pay off from a RESTful get, for example.

The initial code in this example uses a **databinding** approach. I create my own record structure, in this case the objects for Assortment, Album, and Track, and a mapping layer reads data from the document and populates my record structure. This works well when the mapping isn't too complicated and I want to use specific elements of the document structure, as I do for the internal client. Jackson's data binding is one

example, another is the JAXB standard for processing XML in Java.

The third case is to read the document data into a **document model**, a tree that is designed to be a simple and faithful representation of the document. XML's Document Object Model (DOM) is an example of this. A document model is generic - I don't have to specify a record structure of Assortment, Album, and Track, instead the reader will create a generic structure of nodes that captures this information. A document model can use a language's regular collections, commonly in a List And Hash data structure, or provide its own model, such as the XML Document Object Model. Jackson's document model is a hierarchy of its own `JsonNode` classes

Passing through the JSON

Now I've encapsulated the assortment's JSON handling I can start looking at the different ways I might refactor it, depending on how it's used.

The simplest case is that the service only requires the same JSON that was supplied to the assortment. In this case I can just store the JSON string itself in the assortment and not bother with jackson at all.

class Assortment...

```
  private String json;

  public static Assortment fromJson(String json) {
    Assortment result = new Assortment();
    result.json = json;
    return result;
  }
  public String toJson() {
    return json;
  }
```

I can then remove the album and track classes entirely.

Indeed in this case I'd remove the assortment class entirely, and just have the service return the result of the data source call directly.

class Service...

```
  public String tuesdayMusic(String query) {
    try {
      return dataSource.getAlbumList(query);
    } catch (Exception e) {
```

```
      log(e);
      throw new RuntimeException(e);
    }
  }
```

Internal client

Next I'll consider the case where I have a client in the server process that requires some of the information gathered from the JSON data.

class SomeClient…

```
  public List<String> doSomething(Assortment anAssortment) {
    List<String> titles = anAssortment.getAlbums().stream()
            .map(a -> a.getTitle())
            .collect(Collectors.toList());
    return somethingCleverWith(titles);
  }
```

This case requires some of my java code to manipulate the JSON data, so I need more than a string. However, if this is the only client, then I don't need to build the entire object tree I showed earlier. Instead I can have Java classes that look like this.

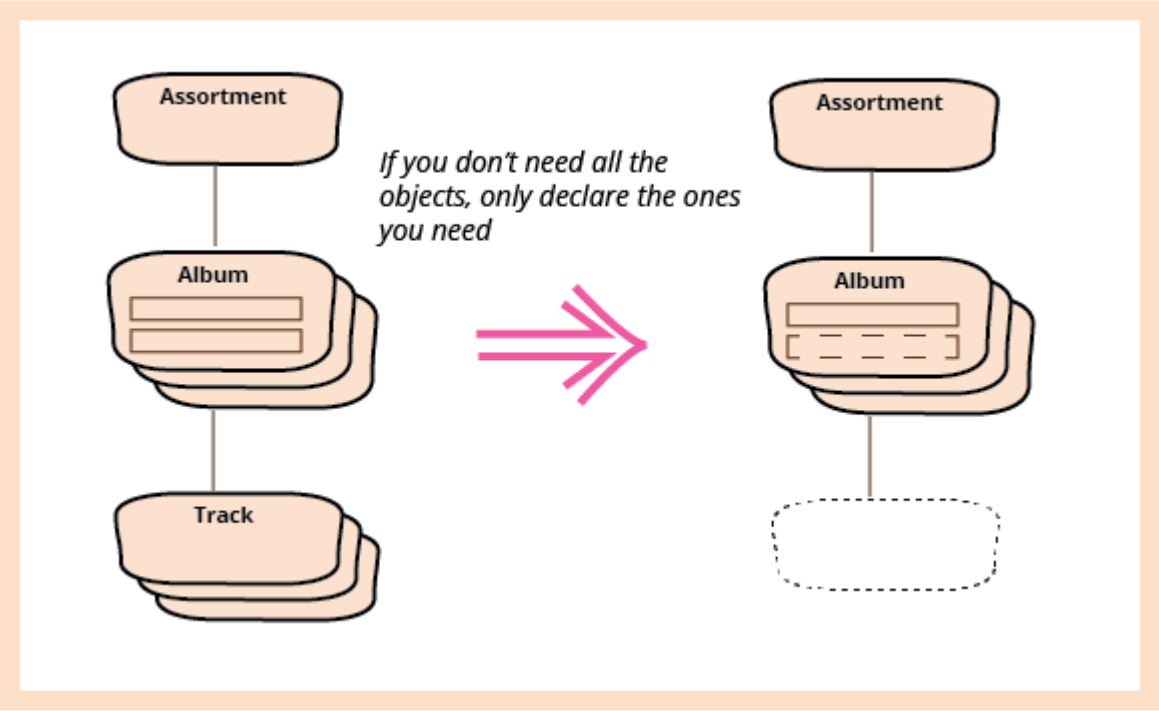
class Assortment…

```
  private List<Album> albums;

  public List<Album> getAlbums() {
    return Collections.unmodifiableList(albums);
  }
```

class Album…

```
  private String title;

  public String getTitle() {
    return title;
  }
```

I can delete the track class completely.



If I only need part of the JSON data, there's no point in importing it all. Specifying just the bits I need via databinding is a really good way to get hold of a reduced set of data like this. Libraries that use databinding like this usually have a configuration parameter that indicates how the databinding should treat fields in the JSON that don't have a binding in the target records. By default, Jackson throws an `UnrecognizedPropertyException`, but it's easy to change this by disabling the `FAIL_ON_UNKNOWN_PROPERTIES` feature with a call like

```
anObjectMapper.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES)
```

By declaring a class structure with only the properties I need, I'm avoiding unnecessary effort - the act of declaring the properties is as good a way of any as picking out the fields I want.

This is a case where another java class wants to get an assortment's album titles with a java API call. A similar case is where a client wants a subset of the JSON data, the regular `toJson` call will only now return the album titles.

```
{"albums":[
   {"title":"Isla"},
   {"title":"Horizon"}
]}
```

If I'm using only a subset of the data from an input document, I usually find it best to arrange things so I only tie myself to the data I'm using. If I define a full object structure to map to, then my code will break should the supplier add a field to the JSON that I could safely ignore. By defining only the structure I use, I'm creating a Tolerant Reader, which makes my program much more resilient to changes in the input data.

Java API and full JSON

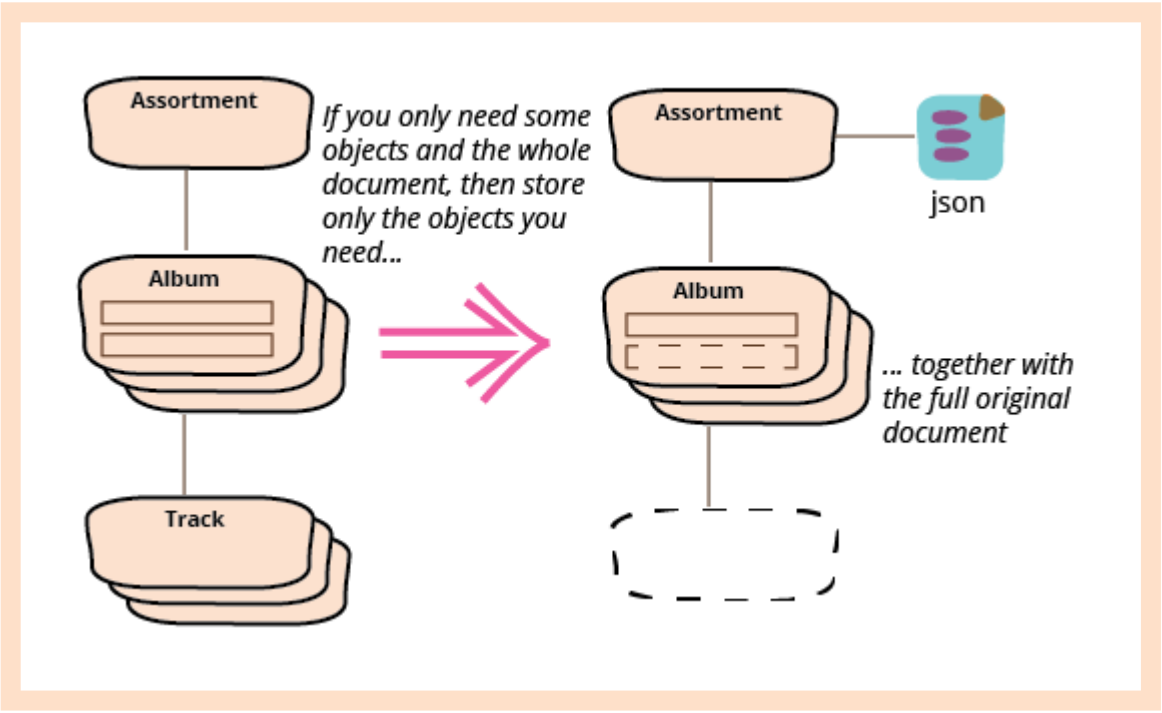
This naturally leads to my third case, what if we want a service call to return the full JSON document, but at the same time we want the list of album titles through a Java API?

The answer for this case is a combination of the two previous cases, store both the original JSON string and whatever java objects are needed for the API. So in this case I add the string to the factory method.

class Assortment…

```
  private String doc;
  public static Assortment fromJson(String json) {
    try {
      final Assortment result = Json.mapper()

.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES)
                .readValue(json, Assortment.class);
      result.doc = json;
      return result;
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
  public String toJson() {
    return doc;
  }
```

Once I've done that, I can delete unneeded methods and classes from the Java data structure, just as I did when only using a Java API.



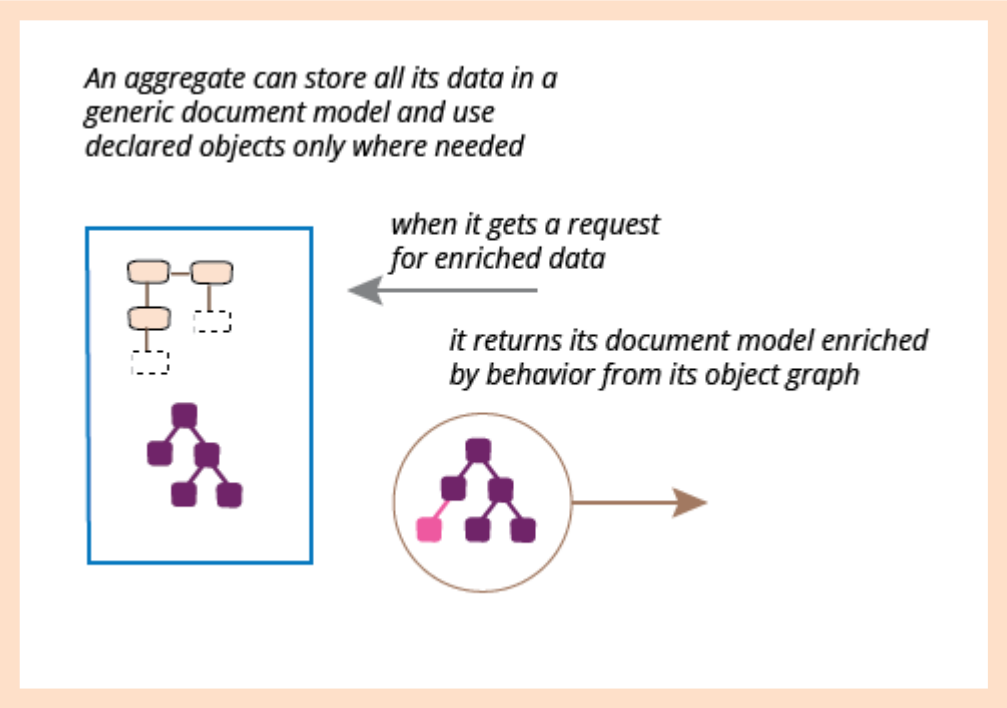
Enriching the output JSON document

The scenarios so far have treated the JSON document as a complete carrier of the information we need, but sometimes we use the server process to enrich that information. Consider the case where we'd like information about the length of each album. With the complete object structure this is easy.

class Album…

```
  public int getLengthInSeconds() {
    return
tracks.stream().collect(Collectors.summingInt(Track::getLengthInSeconds));

  }
```

This method both provides the information to java clients using java method calls, and also adds it automatically to the JSON output when I use jackson's databinding to create the output JSON. The only problem is that this forces me to declare the entire JSON structure as java classes, not a big problem for this small example but a problem when there's tens or hundreds of otherwise unnecessary class definitions. I can avoid this by using and enriching a document model.



I'll start the refactoring from the original starting point of a full set of class definitions, together with the method for the album lengths. My first move is, as with the previous example, add an additional JSON document when reading. However this time I'll read it as a Jackson tree model, and I won't yet modify the output code.

class Assortment...

```
  private List<Album> albums;
  private JsonNode doc;

  public List<Album> getAlbums() {
    return Collections.unmodifiableList(albums);
  }
  public static Assortment fromJson(String json) {
    try {
      final Assortment result = Json.mapper().readValue(json,
Assortment.class);
      result.doc = Json.mapper().readTree(json);
      return result;
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
  public String toJson() {
    try {
      return Json.mapper().writeValueAsString(this);
    } catch (JsonProcessingException e) {
      throw new RuntimeException(e);
```

```
    }
  }
```

My next step is to create a method that will output the soon-to-be-enriched JSON document model. I start with a simple accessor and test that it just outputs the current document.

class Assortment...

```
  public String enrichedJson() {
    return doc.toString();
  }
```

class Tester...

```
  @Test
  public void enrichedJson() throws Exception {
    JsonNode expected = Json.mapper().readTree(new
File("src/test/enrichedJson.json"));
    JsonNode actual =
Json.mapper().readTree(getAssortment().enrichedJson());
    assertEquals(expected, actual);
  }
```

For the moment the test just probes that the output document is the same as the input, but I now have a hook where I can gradually add steps of enrichment to the output. It's not a big deal here, as I only have the one, but if there are several enrichments to the document, then such a test allows me to add them one at a time.

So now it's time to set up the enrichment code.

class Assortment...

```
  public String enrichedJson() {
    JsonNode result = doc.deepCopy();
    getAlbums().forEach(a -> a.enrichJson(result));
    return result.toString();
  }
```

class Album...

```
  public void enrichJson(JsonNode parent) {
    final ObjectNode albumNode = matchingNode(parent);
    albumNode.put("lengthInSeconds", getLengthInSeconds());
  }
```

```
  private ObjectNode matchingNode(JsonNode parent) {
    final Stream<JsonNode> albumNodes =
StreamSupport.stream(parent.path("albums").spliterator(), false);
    return (ObjectNode) albumNodes
      .filter(n -> n.path("title").asText().equals(title))
      .findFirst()
      .get()
      ;
  }
```

My basic approach for enrichment is to traverse the java records, asking each one to enrich its corresponding tree node.

> In general I prefer using collection pipelines as much as possible, but have to confess the whole business with `StreamSupport` and `spliterator` is a right pain. Hopefully over time Jackson will support streams directly, to avoid having to do this.
>
> Rather than modify the document embedded in the assortment, I prefer to create a new document. In general I prefer to keep data as read and make updates on demand. If the cost of building the new JSON document is high, I can always cache the result.

If I have a bunch of enrichments, I can add them one at a time with this mechanism, testing for the new data with each change. Then, once I'm done, I can swap the databound output for the enriched document.

class Assortment…

```
  public String toJson() {
    JsonNode result = doc.deepCopy();
    getAlbums().forEach(a -> a.enrichJson(result));
    return result.toString();
  }
```

Once I've done that I can merrily prune my class structure, removing all the now-dead wood of unused data, methods, and classes. Not very many in this example, but it's fun to rip 37 classes out your code this way.

Using only the tree model

I can prune even more from the Java class declarations by using the tree model as the basis for enrichment. In this approach I enrich purely by walking the tree of the document and not involving domain objects at all. In this case the enrichment code would look like this.

class Assortment...

```
  public String toJson() {
    JsonNode result = doc.deepCopy();
    enrichDoc(result);
    return result.toString();
  }
  private void enrichDoc(JsonNode doc) {
    for(JsonNode n : doc.path("albums")) enrichAlbum((ObjectNode)n);
  }
  private void enrichAlbum(ObjectNode albumNode) {
    int length = 0;
    for (JsonNode n : albumNode.path("tracks")) length +=
n.path("lengthInSeconds").asInt();
    albumNode.put("lengthInSeconds", length);
  }
```

This kind of code should look familiar to those who are used to manipulating List And Hashs in dynamic languages. On the whole, however, I don't prefer this in Java. Having and using the java objects is generally the best place to put behavior, particularly as it grows in volume and complexity.

One exception where I would use this style, however, is if there are many layers of navigation down the hierarchy through layers of otherwise unneeded classes, although there I would be inclined to create objects using just the lower-level nodes.

Creating objects deep in the document structure

What if we have a large document that would need many classes to represent, most of which are unnecessary, but has some important stuff down towards the leaves of the tree? With the earlier example about lists of album titles, I could support them and shave off the tracks by databinding to a subset of the JSON data. But what if I only wanted something lower down?

For my example, let's imagine a client that wants a list of tracks. Since there's only an album class in between the track and the assortment, I would handle that case by databinding. But let's pretend there are a dozen unneeded layers in the way, what then?

In that situation I would want to avoid databinding all those layers, but I still want proper objects for the client. To illustrate how I might handle this, let's assume that so far I only have a client who wants the entire JSON string, so I followed the earlier refactoring path

and store only that in my assortment.

class Assortment...
```
  private String json;

  public static Assortment fromJson(String json) {
    Assortment result = new Assortment();
    result.json = json;
    return result;
  }
  public String toJson() {
    return json;
  }
```

That's good, a nice minimal solution to the problem. But then I get a new feature that requires a new Java API over the document.

class SomeClient...
```
  public String doSomething(Assortment anAssortment) {
    final List<Track> tracks = anAssortment.getTracks();
    return somethingCleverWith(tracks);
  }
```

Again, for this document I'd just switch to databinding, but instead we'll keep up the pretense that instead of a single Album class between the assortment and track, I actually have a dozen things - enough to deter me from databinding.

I want to use the document tree, so my first move is to refactor from the string to the tree.

class Assortment...
```
  private JsonNode doc;

  public static Assortment fromJson(String json) {
    Assortment result = new Assortment();
    try {
      result.doc = Json.mapper().readTree(json);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
    return result;
  }
  public String toJson() {
```

```
    return doc.toString();
  }
```

With this bit of preparatory refactoring done, I can then look to create the tracks. I do this by selecting just the track nodes and then using databinding with the track nodes as the source.

class Assortment...

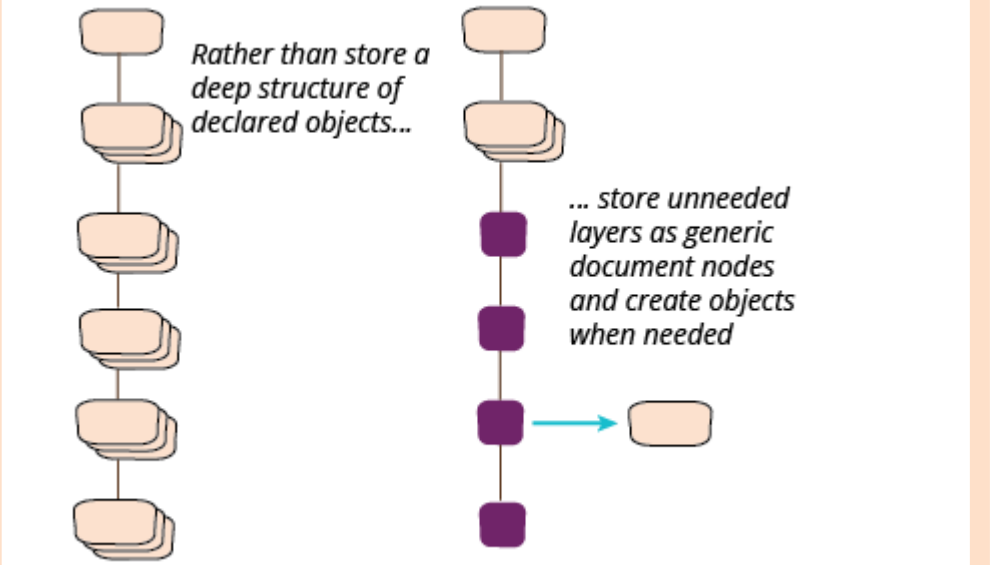
```
  public List<Track> getTracks() {
    return StreamSupport.stream(doc.path("albums").spliterator(), false)
            .map(a -> a.path("tracks"))
            .flatMap(i -> StreamSupport.stream(i.spliterator(), false))
            .map(Track::fromJson)
            .collect(Collectors.toList())
            ;
  }
```

class Track...

```
  private String title;
  private int lengthInSeconds;
  public String getTitle() {
    return title;
  }
  public int getLengthInSeconds() {
    return lengthInSeconds;
  }
  public static Track fromJson (JsonNode node) {
    try {
      return Json.mapper().treeToValue(node, Track.class);
    } catch (JsonProcessingException e) {
      throw new RuntimeException(e);
    }
  }
```

I've shown this with a single record, but it's just as easy to do this with little sub-trees, allowing me to pluck out little bits of important information from a larger document. This approach is handy if I'm doing a more serious restructuring of the document, I can use these kinds of local APIs as data sources to build an output Data Transfer Object which I can then serialize to an appropriate output representation

I can also do a similar refactoring from a full databound model.



Wrapping Up

A large part of our programming effort lies in manipulating and processing data that comes in through documents of hierarchic data. It's good to remember that there are several ways to process this information and choose the right mix to keep a code base small and flexible. It's common for people to forget that encapsulation means hiding data structures and processing methods, and we should not expect that a module's interface should correspond to its internal storage.

Acknowledgements

My colleagues Carlyle Davis, Chris Birch, and Peter Hodgson discussed this article with me as I was drafting it

Significant Revisions

17 December 2015: Published final initial installment

15 December 2015: Published section enriching output json

14 December 2015: Published first installment